Kimmo Mantere

Efterbehandling av 3D-grafik i spelmotorer

Kandidatavhandling

Handledare: Jan Westerholm

Institutionen för informationsteknologi

Åbo Akademi

Åbo 2012

Referat

I denna avhandling kommer jag att beskriva vad efterbehandling i spelmotorer handlar om och hur efterbehandling inom tredimensionell datorgrafik fungerar generellt. Innan jag går in på det tekniska berättar jag allmänt om videospel, spelmotorer, och deras komponenter. Inledningsvis börjar jag med att förklara vad en fragmentskuggare är, ty den är en väsentlig del av den teori som kan behövas för att förstå hur efterbehandling fungerar. Därefter ger jag en mera detaljerad beskrivning av olika efterbehandlingseffekter som används i dagens videospel. Jag kommer att diskutera olika implementationsmodeller och reder konsekvent ut vilken är den önskade effekten och teorin bakom idén för varje implementation. Därefter beskriver jag vilka eventuella förutsättningar varje metod kräver ur spelutvecklarens synvinkel och vad allt man måste ta i beaktan. Sedan försöker jag identifiera för- och nackdelarna hos de olika metoderna. Jag tar även en titt på effektiviteten och jämför vissa implementationer för några av efterbehandlings-effekterna. Slutligen kommenterar jag hur lyckad implementationen är och huruvida resultatet på skärmen motsvarar den effekt man ville åstadkomma. I denna uppsats gås igenom några tekniker och effekter för HDR samt kantutjämning.

Sökord: efterbehandling, rendering, fragmentskuggare

Innehållsförteckning

1. Inledning..... 1

2. Fragmentskuggaren..... 3

3. HDR..... 5
3.1 Varför behövs ett större dynamiskt omfång? 5
3.2 Tonmappning..... 7
3.3 Lokal tonmappning..... 9
3.4 Bloomeffekt..... 12
3.5 Stjärnor och skuggstrålar…… 14

4. Kantutjämning..... 16
4.1 Supersampling..... 17
4.2 Multisampling.....18
4.3 Morfologisk kantutjämning..... 19
4.4 FXAA…….. 20

5. Diskussion……..... 21

Litteraturlista……. 22

1. INLEDNING

Inom videospelbranschen satsas det mycket pengar. Då man tittar på budgeter som stora speltillverkare använder talar man om miljoner. Att starta ett företag inom spelindustrin kräver stora investeringar och därför kan även förlusterna vara höga. För att skapa framgångsrika spel räcker det inte med att göra ett spel utgående från vad som tidigare har fungerat väl utan det krävs nya innovationer [1]. Konsumenterna har ofta höga förväntningar på spelets utseende, och för att göra spelen attraktiva är man tvungen att satsa på grafiken i dem.

Spelen blir allt tyngre och kraven på hårdvaran som ska köra dem blir större. Videospel är bland de mest resurskrävande applikationer för datorer idag och till exempel grafikprocessorn utvecklas hand i hand med spelindustrins framsteg [2]. Från att i början spelutvecklingen har gått ut på att manuellt skriva instruktioner för att visa en enskild bildpunkt (eller ett tecken) på skärmen, har processen därefter automatiserats till stor del. De tekniska framstegen man har tagit inom spelindustrin har varit så stora att i dagens läge är största skillnaden mellan ett videospel och en film den att ett spel är interaktiv, dvs. användaren har möjligheten att påverka vad som händer på skärmen. Att videospel allt mera liknar filmer märks även på utvecklingssidan.

Ett stort spelföretag kan lätt ha över hundra anställda som arbetar på fulltid. Det finns flera uppgifter inom t.ex. design, programmering och testning. Då stora titlar tillverkas kan bl.a. orkester, kör och röstskådespelare hyras. För att artisterna skall kunna uttrycka sina idéer skall utvecklingsmiljön ge friheten att låta bli att överväga huruvida någonting är möjligt att göra. Då kan man koncentrera sig enbart på den kreativa delen av processen. Detta ställer höga krav på spelmotorn som fungerar som kärnan för spelutvecklingen. Dess uppgift är framförallt att underlätta arbetet, så att man inte behöver sätta tid på att tänka på programmeringsfrågor utan fokusera mera på att skapa innehåll till projektet. För att man ska kunna ha full kontroll behövs det dock åtminstone ett skriptspråk. [3]

Spelmotorn är ansvarig för flera olika slags funktionaliteter såsom ljud, nätverk, minneshantering, animation, rendering och fysiksimulering. Spelmotorn ger en

bred uppsättning av verktyg för de ovannämnda områdena. Tillsammans bör de bilda en väl integrerad utvecklingsmiljö bestående av robusta mjukvarukomponenter. Grafikmotorn ska ta hand om återgivningen (renderingen) av scenen, den virtuella världen med sina objekt som artisterna konstruerar.

Till grafikmotorn räknas ofta även animationerna och fysikberäkningarna. Därför kan man istället tala om renderingsmotorn då man explicit menar den del där modellerna återges (renderas) på skärmen. Den tredimensionella renderingsmotorn består av sina egna faser såsom rastrering, klippning, skuggning, färgning, projicering och lysning.

Till efterbehandlingen räknas de effekter och modifieringar som görs då själva geometrin är färdigt beräknad och en färdig bild av hela scenen i någon form finns tillgänglig, men innan den slutligen kastas ut på skärmen. Det är alltså frågan om kronan på verket som appliceras före den slutliga bilden är färdig. Till dessa effekter hör bl.a. skärpedjup, skuggstrålar, bloomeffekt och supersampling.

Trenden inom spelindustrin har redan länge varit att göra spelen fotorealistiska och med efterbehandling kan man åstadkomma häpnadsvärt tilltalande resultat. Efterbehandlingseffekterna begränsar sig trots allt inte enbart på simulering av fenomen i verkligheten. Med efterbehandling är det även möjligt att åstadkomma andra estetiskt vackra resultat. Större spelföretag har sin egen uppsättning av formgivare som använder sig av olika designverktyg. Då är det viktigt att grafikmotorn stöder och tillåter olika slags effekter och ger artisterna möjligheten att bygga upp precis en sådan värld som de hade föreställt sig i tankarna.

Inom spelutveckling ställs det höga krav på effektiviteten av metoderna som används för grafikbehandlingen. Dagens spel är i hög grad interaktiva och scenen skall renderas flera gånger per minut för att sekvensen av bilder på displayen kan uppfattas som sammanhängande rörelser. Om rutan inte uppdateras med en tillräcklig frekvens kommer detta lätt att uppfattas som störande ryckighet.

Då man planerar videospel för spelkonsoler är hårdvaran given och då kan man enkelt bestämma minimikrav för effektiveten genom att specificera en acceptabel

siffra på hur många bilder per sekund som ska genereras. När man har en persondator som plattform blir det svårare att identifiera klara gränser, ty prestandan hos en konsuments dator kan variera mycket beroende på hur ny den är och vilken grafikkort den har.

Beroende på spelgenren kan responsivitetskraven variera, t.ex. i förstapersonsskjutspel krävs det snabba drag och reflexer av spelaren, då kan inga fördröjningar accepteras för att spelupplevelsen inte ska förstöras. Därför kan tredimensionella spel ofta betraktas som mindre realtidssystem.

2. FRAGMENTSKUGGAREN

I detta kapitel förklarar jag vad en fragmentskuggare (eller pixelskuggare) är, vad den gör och när den används. Detta är ett grundläggande begrepp som det är värt att känna till för att bättre förstå hur de flesta efterbehandlingseffekter fungerar.

Tredimensionell rendering inom datorgrafik är den process som utförs för att omvandla en tredimensionell modell till en tvådimensionell bild, den som visas på skärmen. En skuggare (engelskans *shader*) är ett datorprogram som används för att beräkna olika slags renderingseffekter. Grafikprocessorn består av en hel del programmerbara beräkningsenheter som kan exekvera parallellt och skuggare används för att programmera dem. För grafikbearbetning finns det flera olika typer av skuggare: pixelskuggare, vektorskuggare, tessellationsskuggare och geometriskuggare till exempel. [4]

Traditionellt börjar renderingen med att centralenheten först skickar instruktioner och data angående geometrin till grafikprocessorn. Vektorskuggaren utför transformeringar och belysningsberäkningar, medan geometriskuggaren kan utföra ändringar i geometrin på scenen. Sedan delas geometrin i mindre trianglar. Dessa delas vidare i 2 x 2 pixlars matriser som bildar en stor bildpunktsmatris (eller rambuffert). Den lagrade tvådimensionella informationen i bildpunktsmatrisen är användbar för att visa bilden på skärmen. Innan bilden

slutligen skickas till displayen kan pixelskuggaren utföra sina beräkningar och modifiera resultatet. [4][5]

En pixelskuggare, även kallad fragmentskuggare, är den sista komponenten i den tredimensionella datorgrafikbehandlingen. I denna fas beräknas färger och andra attribut för varje bildpunkt. Bland annat kan pixelskuggaren ändra Z-bufferten, d.v.s. djupet på pixeln, i Z-riktningen i det tredimensionella koordinatsystemet från kamerans (åskådarens) perspektiv. Som input får fragmentskuggaren RGB-värden som motsvaras av färgerna: röd, grön och blå, med hjälp av vilka man kan uttrycka alla regnbågens färger. Dessutom får den tillgång till ett värde för en s.k. alfakanal som anger hur genomskinlig den aktuella pixeln är. Eftersom fragmentskuggaren alltid opererar på enskilda bildpunkter utan kännedom om scenens geometri kan den inte ensam producera särskilt komplexa effekter. Å andra sidan är det möjligt att bearbeta scenen som en helhet, ty objekten behandlas inte längre isolerade från varandra. [4][5]

Effekter som kan appliceras i pixelskuggaren varierar från att tilldela belysningsvärden till att producera olika fenomen såsom speglingar eller genomskinlighet. I regel kallas dessa för efterbehandlingseffekter av den logiska orsaken att en pixelskuggare bearbetar på en nästan färdigt renderad bild. Ett sätt att beräkna det slutliga värdet på färgen för en enskild bildpunkt kan t.ex. vara att betrakta de närliggande pixlarna från en hjälpbuffert och utgående från dessa avgöra hur den aktuella pixeln ska påverkas. [4][5]

För att övervinna problemet med att fragmentskuggaren har tillgång till bara den aktuella pixeln kan en textur användas. Idén är att först rendera hela scenen i en textur och sedan låta pixelskuggaren komma åt en godtycklig position av texturen med hjälp av texturkoordinater. För att snabba upp processen kan man direkt rendera till ett rambuffertobjekt. Här har man också en möjlighet att rendera med en högre bildupplösning än den fysiska skärmen klarar av. Detta visar sig vara behändigt bl.a. för en kantutjämningsmetod som kallas supersampling. [4][5]

I detta kapitel förklarade jag att en fragmentskuggare är ett program som körs på beräkningsenheterna i en grafikprocessor. Programmet körs för varje pixel för att

göra ytterligare modifieringar på dess färg. Pixelskuggaren exekveras sist och därför talar man om efterbehandling då man programmerar fragmentskuggare.

3. HDR

Stort dynamiskt omfång, HDR (engelskans *high dynamic range*) eller högt dynamiskt omfång kallas den teknik som används för att representera en större utbredning av färger än den som används inom traditionell avbildning. Avsikten är att förstärka mörkheten och ljusheten i en scen utan att förlora detalj. Cryteks Cryengine 1 var den första spelmotorn som implementerade högt dynamiskt omfång och därefter har tekniken mer eller mindre etablerats som grundfunktionalitet i moderna spel. I detta kapitel kommer jag att gå igenom hur och varför HDR används samt hur den informationen kan vidare konverteras till apparatur med lågt dynamiskt område (LDR, engelskans *low dynamic range*) såsom gamla CRT- och LCD-displayer. [7]

För HDR-databehandling används flyttalstexturer, i allmänhet med antingen 16 eller 32 bitar per färgkanal. Eftersom resursallokeringen av videospel överlag är stor och en 16-bitars representation redan ger goda resultat är den att föredra. Om man inte vill skapa en skild karta över ljusheten kan man alternativt använda alfakanalen i den aktuella omgivningskartan som en multiplikator. Denna alfakanal kan då användas i fragmentskuggaren för att multiplicera färgintensiteterna. [5]

3.1 Varför behövs ett större dynamiskt omfång?

I de flesta vardagsapplikationer används en åtta bitars representation av färger som oftast räcker till. Färgen för pixlar bestäms utgående från tre åtta bitars heltalsvärden som motsvarar grundkomponenterna rött, grönt och blått. Med åtta bitar kan man uttrycka värden från 0 till 255, vilket räcker till att visa de flesta förnimbara färgnyanserna. Människans öga kan dock se mångdubbelt så många

ljushetsnivåer än en sådan färgpresentation klarar av att visa. I "normala" renderingsomständigheter är genomsnittliga ljusstyrkan liknande över hela scenen och lågt dynamiskt omfång ger tillräcklig noggrannhet för att uppvisa kontrasterna. [5] [8]

I verkliga livet finns det stora variationer i ljusintensiteter och dessa kontraster kan helt enkelt inte uttryckas med endast åtta bitar. Till exempel skillnaden i nivån av luminansen hos solljus och månljus ligger på en faktor nära tusen. Om detta inte var nog kan man ännu konstatera att en stjärnhimmel utan måne är över en miljon gånger mörkare än en himmel med fullmåne. För att replikera en scen av en värld med mycket varierande ljusstyrkor kommer LDR-teknikens precision inte att räcka till. För en detaljrik scen med varierande ljusstyrkor krävs ett ökat tonomfång. [5] [8]

Typiska ljusfenomen som är lätta att känna igen från omvärlden är bl.a. spegelreflexion och brytning (refraktion). Andra fenomen har sitt ursprung i hur människans öga fungerar; dessa är lättast att illustrera med exempel. Tänk dig en omgivning som är utom räckhåll av ljus, ett garage, en grotta eller en tunnel exempelvis. Utifrån sett, i dagsljus, skulle en sådan plats verka så gått som svart, och det vore omöjligt att identifiera objekt som fanns där. Om vi tvärtom befann oss i det mörka utrymmet, kunde vi se hur bra som helst vad som fanns där, medan omvärlden skulle verka ytterst starkt belyst. Detta beror på att ögat (och kameran) kan förnimma bara en begränsat omfång av ljushet. En kamera och människans öga uppskattar den genomsnittliga ljushetsnivån och detta sköts av en exponeringsmekanism som avgör hur mycket ljusenergi man kan se. [5]

Ett statiskt kontrastförhållande är förhållandet i luminans mellan det mörkaste och ljusaste färgen, enligt ekvation 3.1 [9], som kan produceras samtidigt:

$$Kontrastförhållande = \frac{[max\,luminans]\,\mathrm{cd/m^2}}{[min\,luminans]\,\mathrm{cd/m^2}} \tag{3.1}$$

Dynamiskt kontrastförhållande däremot anger samma förhållande utan tidsbegränsning. Den dynamiska kontrasten hos människans öga ligger omkring 1 000 000:1 – 10 000 000:1 vilket nås genom anpassning i regnbågshinnan, dvs.

justering av pupillens storlek beroende på ljusintensiteten. Denna fördröjning märks i vardagen t.ex. då vi vaknar och tänder ljusen eller när någon tar ett foto med blixt; det tar en tid att vana sig till en förändring i belysningen. Statiska kontrastförhållandet för ögat är någonstans mellan 300:1 och 10 000:1. Olika referenser ger mycket varierande värden på kontrastförhållanden beroende på mätteknik och mätningsomständigheter. [10][11]

3.2 Tonmappning

Faktum är att fåtal konsumenter har skaffat ännu åt sig HDR-displayer som direkt kunde visa en utvidgad mängd av färger och kontraster. Kontrastförhållandet, dvs. luminansen i jämförelse mellan mörkaste och ljusaste värdet en skärm klarar av att visa (svart respektive vitt), i dagens LED-monitorer är typiskt ungefär 1000:1 medan t.ex. en plasma-TV som är betydligt dyrare kan nå 4000:1 i statiskt kontrastförhållande. Dynamisk kontrast kan i teorin vara oändlig, ty bakgrundsbelysningen i skärmar går att dimma ner och släckas helt, vilket då ger ett värde på 0 cd/m² i luminans. Detta är dock inte till stor nytta i en scen där ytterst mörka och ljusa ljushetsnivåer uppträder samtidigt. [9]–[11]

Extrautrymmet som gått till att spara det högre dynamiska omfånget har dock inte gått till spillo. HDR-data kan utnyttjas för att rekonstruera detaljer och fenomen på vanlig LDR-apparatur. HDR-data anpassas för avbildning till LDR-anordningar genom att jämna ut kontraster i den ursprungliga bilden. Denna teknik kallas tonmappning och processen utförs av en tonmappningsoperator. Det är visserligen möjligt att approximativt replikera enskilda fenomen också utan HDR-rendering, men det har en kostnad i kvalité. [6]

I en linjär modell för direkt överföring av HDR-data till en LDR-display kommer mörka delar att klippas till svart och ljusa till vitt. Detta innebär att alla detaljer som fanns inom det mörka eller ljusa området, under eller över ett visst gränsvärde, kommer att förloras. Det finns dock andra, bättre modeller som också behandlar varje pixel enligt en och samma funktion. Sådana tonmappningopreatorer kallas globala. Till exempel sigmoidfunktionen, en ”S”-

formad kurva, är adapterad så att den mappar hela området av tillgängliga ljusintensiteter. Den har visserligen en märkbar nackdel, nämligen kontraster kommer att jämnas ut kraftigt. [8]

En kompromiss mellan en linjär modell och en sigmoidfunktion är att låta kontrasten vara lägre inom mörka och ljusa delar, så att det mest intressanta området får nöjaktig precision för variation i ljushet. En sådan tonmappningsfunktion kan enkelt konstrueras som ett logaritmiskt medelvärde för luminansen i scenen så att kontrasten bevaras i mellanområdet. En funktion för tonmappningsoperatorn kan också härledas ur ett histogram för statistiken av ljusintensiteterna i scenen. Resultatet av några globala tonmappningsoperatorer är illustrerade i Figur 1 där svagheterna i en linjär och en sigmoidfunktion (vänster) framgår i jämförelse med mer sofistikerade operatorer (höger). [8]

linjär

logaritmisk

sigmoid

histogram

Figur 1: Jämförelse av olika globala tonmappningsoperatorer. (egen översättning [8])

I Cryteks spelmotor för spelet Far Cry, Cryengine 1 som var första produkten på spelmarknaden som implementerade högt dynamisk omfång [7], användes Erik Reinhards tonmappningsoperator [12]. Algoritmen börjar med att räkna ut ett

logaritmiskt medelvärde för luminansen i scenen enligt ekvation 3.2 [12] på grafikkortet:

$$Lum_{avg} = \exp\left(\frac{1}{N}\sum_{x,y}\log(\delta + Lum(x,y))\right) \tag{3.2}$$

Lum(*x, y*) står för luminansen för den aktuella pixeln, *N* står för totala antalet pixlar i scenen och δ är en konstant för att undvika singularitet i fall det råkade finnas helsvarta pixlar. Därefter sker skalning till en önskad medelluminans α enligt ekvation 3.3 [12]:

$$Lum_{scaled}(x,y) = \frac{\alpha}{Lum_{avg}} Lum(x,y) \tag{3.3}$$

Slutligen appliceras själva tonmappningsoperatorn enligt ekvation 3.4 [12] som är en sigmoid funktion och ger en ”S”-formad kurva:

$$Color(x,y) = \frac{Lum_{scaled}(x,y)}{1 + Lum_{scaled}(x,y)} \tag{3.4}$$

Denna funktion garanterar att återge hela omfånget av luminanser i det område som går att visas på skärmen. För att simulera ögats eller kamerans anpassning kan Lum_{avg} istället ersättas med ett adapterat värde för ljusstyrkan som konvergerar mot samma medelvärde på luminansen. Med en sådan exponeringskontrol kan man dynamiskt uttrycka mycket varierande belysningsomständigheter. Förenklat, på en skala från 0 till 1, ska man justera genomsnittliga ljusheten till 0,5 för att få ett passligt exponeringvärde för att simulera ögats beteende. Då ögat automatiskt gör denna anpassning tar det en stund och därför ska även en exponeringstid användas i algoritmen för att replikera fördröjningen korrekt. [5][8][12][13]

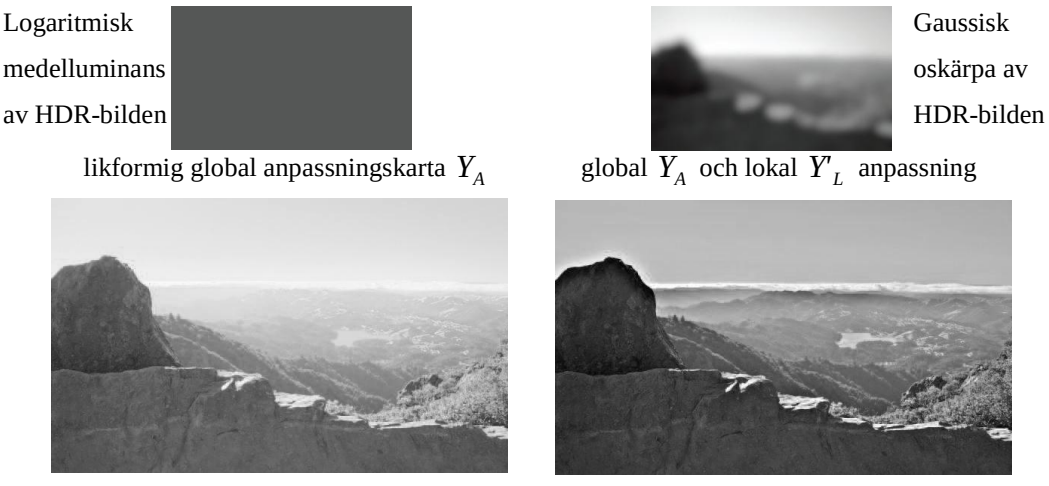
En märkbar nackdel som stöttes på med denna implementation i Far Cry var en mycket förhöjd fyllningsgrad (engelskans *fillrate*) dvs. antalet pixlar som ska renderas av grafikkortet per sekund, och den krävde också stöd för flyttalsbuffertblandning. Dessutom hade denna metod kompabilitetsproblem med FSAA (*Full Screen Anti-Aliasing*), en fullskärmskantutjämningsalgoritm. [12]

3.3 Lokal tonmappning

Globala tonmappningsoperatorer är i allmänhet enkla och effektiva, men kontrasten lider i de mörka och ljusa delarna av scenen. För att åtgärda denna brist kan överföringsfunktionen vidare utvecklas så att den beräknas lokalt på ett begränsat område av bilden. Denna mappningsmetod kallas lokal tonmappning och den kan beskrivas av ekvation 3.5 [8] där Y' betecknar den globalt anpassade luminansen i HDR-scenen. Den globalt anpassade luminansen är normaliserad som $Y' = Y / Y_A$ där Y_A är konstant för hela scenen, medan Y'_L står för den lokalt anpassade luminansen som är ett medelvärde för ett bestämt område. [8]

$$L = \frac{Y'_A}{Y'_L + 1} \tag{3.5}$$

En mappningsbild för den lokalt anpassade luminansen Y'_L kan beräknas från den luminansnormaliserade scenen med Gaussisk oskärpa, en metod för simulering av oskärpa inom bildbehandling. Effekten är åskådliggjord i Figur 2; och följden är skarpare mörka och ljusa områden. I bilden syns dock en sidoeffekt, en glödande störning eller haloeffekt (optisk fenomen, även kallad ljusgård), vid gränsen mellan riktigt ljusa och riktigt mörka områden där kontrasten är störst.



Figur 2: Jämförelse av en global och en lokal tonmappningsoperator. (egen översättning [8])

Graden av artefakten i kantområden beror på storleken av kärnan som har använts i den Gaussiska oskärpan. En större kärna orsakar starkare störning, medan en

mindre kärna minskar förutom artefakten också återgivningen av detaljer. Områden med mycket hög luminans kommer att påverkas av mörkare pixlar och ljusare pixlar kommer att ha inverkan i närheten av mycket mörka delar av scenen. Detta beror helt enkelt på att det beräknade medelvärdet saknar tillräcklig noggrannhet för varken mörka eller ljusa delar i storkontrastområdet. Det finns olika tekniker som används för att undvika denna haloeffekt, men det går lika bra att ersätta Gaussiska oskärpan med andra filter.

En effektiv lokal tonmappningsoperator introducerades av Hanli Zhao et al. [14] med speciell vikt på användning i realtidsapplikationer såsom videospel. Den är en GPU-implementation baserad på Pattanaiks operator. Operatorn upptäcker storkontrastområden och undviker att blanda intensiteten i den arean. Algoritmen kan parallelliseras i hög grad och detta ger mycket hög effektivitet på moderna grafikprocessorer som kan ha flera hundra beräkningsenheter.

HDR-bild	resolution	rutor per sekund (FPS)							
		logaritmisk	Reinhard lokal	Reinhard global	Ashikhmin lokal	Ashikhmin global	Durand	Drago	Pattanaik
spegel	346x512	259	164	228	174	234	168	129	543
minnesmärke	512x768	118	78	102	84	107	85	59	251
skrivbord	644x874	83	61	73	62	75	59	42	176
domkyrka	767x1024	60	41	52	44	54	41	30	128
kontor	2000x1312	20	8	16	9	16	2	9	39
Price Western	3272x1280	7	3	6	3	6	1	6	24

Figur 3: Jämförelse av effektiviteten som antalet rutor per sekund hos olika GPU-baserade tonmappningsoperatorer. (egen översättning [14])

Figur 3 representerar en jämförelse av effektiviteten av olika lokala och globala tonmappningsalgoritmer. Generellt kan man säga att de globala operatorerna är snabbare än de lokala, men som tidigare konstaterats ger lokala funktioner bättre resultat. Pattanaik operatorn som är lokal visar sig dock vara till och med effektivare än globala operatorer och lämpar sig för realtidsrendering även vid stora bildupplösningar. Problemet med GPU-implementationer är alfakanalsblandningen som dåligt stöds av grafikprocessorn; och det skulle behövas metoder för att göra beräkningen t.ex. i pixelskuggaren. [14]

Det har utvecklats mängder av olika tonmappningsfunktioner och dessa går att kategoriseras på många sätt. Det finns bl.a. frekvensbaserade, lutningsbaserade, kontrastbaserade och segmenteringsbaserade tonmappningsoperatorer. Från spelutvecklingssynpunkt bör metoden vara effektiv. Om den avbildade scenen inte märkbart har förbättrats av tonmappningen är det skäl att överväga om det över huvud taget lönar sig att spara HDR-data, ty efterbehandlingseffekter i sista hand bara drar ytterligare klockcykler vid slutändan av renderingen. Huruvida resultatet tilltalar åskådaren är också en smaksak och att jämföra kvaliteten hos HDR-scener kan vara subjektivt. Ofta gäller det att hitta en god balans mellan detaljrikedom och kontrast. Att välja mellan tonmappningsalgoritmer kan vara svårt också därför att de fungerar på varierande sätt beroende på hurdan bild de får som input.

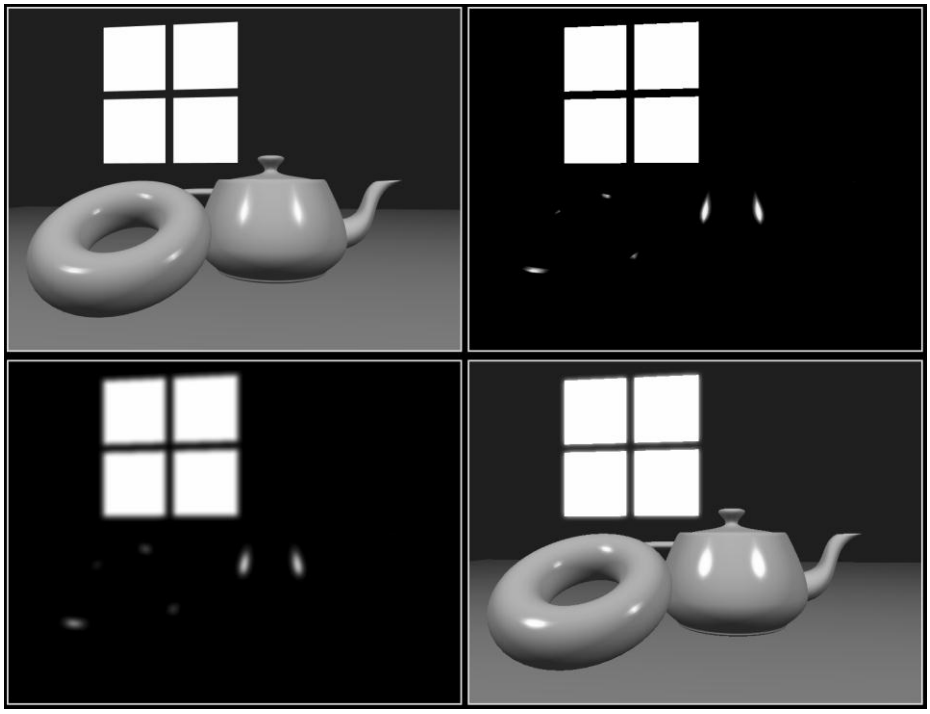
3.4 Bloomeffekt

Bloomeffekt (engelskans *bloom*, *flare*, *glow* eller *glare*) är det fenomen som uppstår då energin från en stark ljuskälla "läcker" eller "blöder" över på den sida av ett objekt som inte är direkt belyst av ljuskällan. På grund av ljusets diffraktiva beteende uppstår en så kallad Airy-mönster (optik) och ljusa delar kommer att spridas över de mörka delarna vid storkontrastområdet. I spel används denna effekt på olika glödande objekt såsom lampor, men även himlen kan betraktas som ett objekt för att enklare replikera solens storskaliga strålning. Figur 4 illustrerar denna effekt som på LDR-apparater försöker skapa en illusion av större ljushetsnivå än den i verkligheten klarar av att visa. [5][6][15]

Figur 4: En illustrativ bild av bloomeffekten längs storkontrastkanten orsakad av utstrålningen från en stark ljuskälla. [6]

Diffraktionsmönstret orsakad av utstrålningen från ljuskällan kan enkelt simuleras med hjälp av ett filter. För en konsistent och jämn effekt kan man använda Gaussisk oskärpa, som viktar pixlarna enligt en normalfördelad kurva. Utgående från hur många pass man låter Gauss-filtret modifiera bilden kan man reglera graden av oskärpa. För att effektivera processen kan filtret appliceras på ett mindre renderingsmål som är av lägre bildupplösning än den ursprungliga scenen. Vid sista passet blandas detta nerskalade textur med det vanliga renderingsobjektet i de ljusaste delarna. Detta görs i fragmentskuggaren genom att kombinera de samplade färgvärdena från bloombufferten. [5][15]

I första passet renderas scenen till en textur. Sedan utgående från ett tröskelvärde separeras en bild endast bestående av de delar som är ljusare än det valda tröskelvärdet; alternativt refererar man till förvalda objekt som skall glöda. I nästa fas appliceras Gauss-filtret på denna bild så att ljuset från de isolerade områdena aningen överskrider kanterna och jämnt tonar bort utåt. Vid sista passet kombineras den ursprungliga scenen med metabilden av de glödande objekten. De olika stegen är illustrerade i Figur 5. Ursprungliga scenen är i övre vänstra hörnet; höger om syns de separerade ljusa delarna; nere finns samma isolerade områden applicerade med Gauss-filtret i den vänstra bilden; och höger om finns resultatet av de kombinerade texturerna. [6][15]



Figur 5: Bloomeffekt implementerad med Gaussisk oskärpa i fyra pass. [6]

Det är möjligt att åstadkomma en bloomeffekt också utan flyttalstexturer enligt liknande princip, men på grund av den mindre exakta representationen av färger och ljusintensiteter lider kvaliteten. Då behöver man dessutom ett alternativt sätt för att representera ljushetsvärden; till exempel kan man implementera det i en separat alfablandningspass. Om man däremot väljer att använda stort dynamisk omfång behöver man en HDR-buffert med flera bitar för fragmentskuggaren, och 16-bitars flyttalstexturer upptar dubbelt så mycket minne och bandbredd än en representation med bara 8 bitar. [5][15]

3.5 Stjärnor och skuggstrålar

Ytan på skinande material såsom glas innehåller ytterst små repor som på grund av refraktion och spegling kan ibland orsaka synliga ljusstrimmor i stjärnform. En bra approximation av detta stjärnmönster kan fås med en riktad oskärpafilter. Genom att progressivt för varje pass ta ett sampel från ökande avstånd längs en diagonal applicerar man mindre oskärpa längre ifrån centrumet på stjärnmönstret. Detta formar en udd, och placerar man fyra sådana spetsar symmetriskt från en gemensam mittpunkt skapas ett enkelt stjärnmönster. Det kan hända att justeringar måste göras i alfakanalsblandningen för att undvika överexponering. [5]

Synliga ljusstrålar förekommer också när de passerar t.ex. moln eller damm. Sådana strålar kallas skuggstrålar (engelskans *crepuscular rays* eller *God's rays*) och för att få rätt perspektiv till denna volymetriska utbredning av ljus behövs det invecklade vektorberäkningar. Detta har tidigare gjorts färdigt av CPU:n och skickats till fragmentskuggaren istället för att låta beräkningen göras pixel för pixel. Äldre implementationer har varit ytterst fyllningsgradkrävande och först på tidigt 2000-tal kom de första realtidsalgoritmerna. Skuggstråk och liknande ljusfenomen är ett område där det ännu behövs effektivare implementationer för realtidsrendering i videospel. [15]–[17]

En metod för att simulera skuggstrålar helt av pixelskuggaren presenterades av Kenny Mitchell från spelföretaget Electronic Arts i *GPU Gems 3* (2007) [16]. För att beräkna belysningen för varje bildpunkt granskar man spridningen från

ljuskällan och avgör om strålen blockeras av någonting. Analytiska modellen för solljusutbredningen går enligt ekvation 3.6 [16]:

$$L(s,\theta) = L_0 \mathrm{e}^{-\beta_{ex} s} + \frac{1}{\beta_{ex}} E_{sun} \beta_{sc}(\theta)(1 - \mathrm{e}^{-\beta_{ex} s}), \tag{3.6}$$

där *s* står för den passerade distansen genom det emitterande mediet och θ är vinkeln mellan solen och strålen. E_{sun} står för källbelysningen från solen, β_{ex} är en utrotningskonstant orsakad av ljusabsorption och β_{sc} är en spridningsvinkel. Första termen beräknar hur mycket ljus som absorberas från åskådarens vy och andra termen tillägger det utspridda ljuset som hamnar i visningsområdet. [16]

Modellen 3.7 [16] för dämpning av källbelysningen då ljusstrålar avbryts på grund av t.ex. moln ser ut så här:

$$L(s,\theta,\phi) = (1 - D(\phi))L(s,\theta), \tag{3.7}$$

där *D(θ)* står för genomskinligheten orsakad av dämpningen från objekt som blockerar solstrålarna för utsiktsplatsen φ. I skärmplanen har man inte full volymetrisk information för att avgöra om en stråle kommer att träffas av ett objekt. Det går visserligen att uppskatta sannolikheten att ljuset täcks vid en pixel genom att addera stickprov längs en stråle av ljus i bildplanet. Utgående från proportionen mellan mätvärden som träffar blockerande objekt och sådana som inte gör det kan en blockeringsprocent räknas ut. Att någonting träffar strålen på vägen bestäms från variationer i kontrast och fungerar bäst då det emitterande mediet är ljusare än föremålen. [16]

Genom att dela samplingsbelysningen med antalet mätvärden, *n*, blir efterbehandlingen helt enkelt summerande provtagning av bilden enligt 3.8 [16]:

$$L(s,\theta,\phi) = \sum_{i=0}^{n} \frac{L(s_i,\theta_i)}{n}. \tag{3.8}$$

Slutligen parametriseras sammanräkningen med dämpningskoefficienter enligt ekvation 3.9 [16]:

$$L(s,\theta,\phi) = exponering \times \sum_{i=0}^{n} sönderfall^{\,i} \times vikt \times \frac{L(s_i,\theta_i)}{n}, \quad (3.9)$$

där *exponering* justerar graden av effekten, *vikt* bestämmer intensiteten för varje sampel och $sönderfall^{\,i}$ avger inverkan av varje sampel längs strålen från ljuskällan och p.g.a. dess exponentiella karaktär kommer varje solstråk att jämnt tona bort. Denna ekvation implementeras i pixelskuggaren; och stickproven beräknas med hjälp av texturkoordinater längs strålarna från ljuskällan. [16]

I detta kapitel gav jag exempel på några av de vanligaste HDR-effekterna som används i dagens videospel. För att få bra precision på effekterna används större dynamiskt omfång med flyttals texturer för att nå det bästa resultatet. Många ljusfenomen går dock att replikeras med förenklade modeller också utan ett högre dynamiskt område. Tonmappning måste i alla fall användas för att undvika över- eller underexponering och samtidigt kunna visa detaljer då man utgår från HDR-data. För att effektivera användningen av pixelskuggare lönar det sig att bunta ihop flera effekter till en och samma fas. Det är närmast alfablandningspasset som ställer till med problem. [15]

4. KANTUTJÄMNING

Kantutjämning (engelskans *Anti-Aliasing*) är en metod för att reducera trappstegseffekten t.ex. på kanterna av polygoner. Denna kantighet beror på att precisionen i form av antalet bildpunkter är begränsad. Vid konturerna av objekt händer det ofta att en pixel inte entydigt kan påstås vara helt innanför polygonen eller helt utanför den utan det finns gränsfall då pixeln är någonstans mittemellan och en del av den hör till vardera grannen. Genom att blanda informationen från båda delarna kan taggigheten minskas och rakare sträck åstadkommas. I detta kapitel presenterar jag några algoritmer som görs i efterbehandlingsskedet.

4.1 Supersampling

Supersampling (SSAA) går helt enkelt ut på att rendera en textur av högre bildupplösning än den slutliga scenen och att beräkna medeltalet av flera stickprov för varje pixel i den slutliga bilden. Det är intuitivt att inse att detta kommer att förbättra resultatet. Metoden är dock långt ifrån perfekt. Till att börja med kan man konstatera att den är en ytterst resurskrävande metod. För det andra, även om artefakter reduceras genom att rendera en större bild kommer de att minskas bara till en viss gräns. Teoretiskt sätt behöver man ett oändligt antal sampel för att bli av med alla tänkbara artefakter, ty gränsen av två delar kan fortfarande hamna innanför ett mindre stickprov (superpixel). [18]

Man börjar med att konstruera en kontinuerlig bild som en funktion $I(x, y)$ som samplas n gånger den slutliga bildupplösningen. Denna virtuella bild släpps sedan igenom ett lågpassfilter. Slutningsvis samplas den filtrerade bilden till en ram av skärmens storlek. För varje pixel i den resulterande bilden beräknas ett motsvarande område från den virtuella representationen. Filtret bevarar låga frekvenser men eliminerar de högre frekvenserna enligt en funktion $H(u, v)$. Utgående från signalbehandling och faltningsteori används Fouriertranform på den ursprungliga bilden, multipliceras med filterfunktionen i filterdomänen och transformeras tillbaka till bildrymden, vilket resulterar i funktionen 4.1 [18]:

$$I'(x, y) = I(x, y) * h(x, y), \tag{4.1}$$

som vidare kan diskretiseras till en tvådimensionell ekvation 4.2 [18]:

$$I'(x, y) = \sum_{i=x-k}^{x+k} \sum_{j=y-k}^{y+k} I(i, j)h(x - i, y - j) \tag{4.2}$$

Värdena (x, y) och (k, j) är diskontinuerliga och utgör ett tvådimensionellt gitter. Om skalningsfaktorn är udda fungerar funktionen utmärkt, men i det fall att jämn faktor används möter inte pixlarna i den virtuella bilden och det slutliga gittret varandra. Därför är det klokt att införa en korrigeringsfaktor och den uppdaterade ekvationen 4.3 [18] blir att se ut så här:

$$I'(i,\ j) = \sum_{p=Si-k}^{Si+k} \sum_{q=Sj-k}^{Sj+k} I(p,q)h(Si - p, Sj - q), \qquad (4.3)$$

där skalningsfaktorn *S* är lika med 2*k*+1. Filtret läggs på superpixlarna (*Si*, *Sj*) från den virtuella bilden, och värdet för den slutliga bildpunkten beräknas utgående från filtervikterna och de omgivande pixlarna. [18]

Eftersom algoritmen opererar på varenda ett sampel oberoende på hur scenen ser ut slösas beräkningsresurser till exempel om det fanns stora areor av konstant färg. Dessutom har pixelvärden innanför polygoner ofta färdigt interpolerats under renderingen. En lösning till detta problem är en teknik som kallas selektiv supersampling och är en kombination av supersampling och en annan algoritm som kallas multisampling (MSAA). Selektiv supersampling går ut på applicera supersampling bara på texturer som framkallar artefakter och multisampling längs polygonkanterna. På så sätt undviks dyr användning av supersampling i områden där samplingsfrekvensen redan är tillräcklig. [18][19]

4.2 Multisampling

En vanlig kantutjämningsalgoritm är multisampling (MSAA, *Multi Sample Anti-Aliasing*) som precis som supersampling går ut på att beräkna flera mätvärden för pixlar i bildplanet. Pixelskuggaren körs dock endast en gång för varje pixel, vilket ger en märkbar effektivitetsökning jämfört med ren supersampling. Den snabbare multisamplingsalgoritmen väljer flera stickprov under ett pass, och sampling av olika typers data görs olika ofta. Om alla positionsproven hamnar innanför fragmentet kommer samplingsvärdet för skuggaren att finnas i mitten av pixeln. Om fragmentet emellertid omfattar färre positionsprov kan samplingspunkten skiftas. Som resultat kommer bara kanterna av polygoner att jämnas ut. Både SSAA och MSAA passar tyvärr dåligt för spelkonsoler och för en relativt ny renderingsteknik som på engelska heter *Deferred Shading*. Även om multisampling körs snabbt är dess minnesförbrukning trots allt ineffektivt. Dessa kantutjämningsmetoder var standarder i videospel för över tio år och nya alternativa algoritmer har tagits i bruk först ganska nyligen. [6][20][21]

4.3 Morfologisk kantutjämning

Ett exempel på en mycket effektiv kantutjämningsalgoritm är en trefasmetod som heter morfologisk kantutjämning (MLAA, *Morphological Anti-Aliasing*). I första fasen identifieras klart avvikande bildpunkter, t.ex. pixlar av annan färg eller intensitet än grannpunkterna, men också utgående från andra data såsom djupet, normalen eller materialet. Dessa bildpunkter formar separationslinjer och innanför dem bildas grupper av liknande pixlar. Utgående från den här informationen konstrueras konturer i den andra fasen. Efter det andra skedet tillämpar man ett färgfilter på pixlarna som befinner sig längs och på vardera sidan av konturerna, och blandar deras värden. [21]

Morfologisk kantutjämning producerar en bild som är jämförbar med kvaliteten av 4X-supersampling. Metoden fungerar dessutom helt oberoende av hur scenen har renderats, och lämpar sig då även för strålföljning. Den kan arbeta på ytterst lite information, idealt endast färgvärden, och den går att köras parallellt, vilket ger extrem effektivitet på dagens grafikprocessorer som har många beräkningsenheter. Det finns dock märkbara nackdelar med MLAA; exempelvis vid pixelsmå detaljer kan det förekomma artefakter när det blir omöjligt att identifiera homogena områden. Samma problem möts också vid kanterna av bildrutan där det inte finns information av andra pixlar. Dessutom klarar MLAA inte av att jämna ut kanter i refraktioner och reflektioner på grund av att det blir svårt att avgränsa konturer på dem. [21][22]

4.4 FXAA

En annan snabb kantutjämningsalgoritm är FXAA (engelskans *Fast Approximate Anti-Aliasing*) som utvecklades av Timothy Lottes hos NVIDIA. För metoden tar det under en millisekund att processa en bild av full HD-upplösning (1920x1080) på en NVIDIA GTX 480-grafikkort. FXAA reducerar vikningseffekten i triangelkanter och artefakter som har uppkommit vid skuggning. Den kan omarbeta artefakter både av pixel- och också subpixelnivå. Kantutjämningen är

färdig i ett enda pixelskuggningspass och är designad att köras först efter konvertering till lågt dynamiskt omfång med standard färgrymd. [21][23][24]

Först görs en skalär uppskattning av luminans för skuggaren som av optimeringsskäl kan göras enbart utgående från röd och grön färg, ty helblåa artefakter sällan uppstår i spelmiljöer. Algoritmen kollar den regionala kontrasten i jämförelse med maximala lokala luminansen för att avgöra om pixlar ingår i en kant eller inte, och sålunda undviker att jämna ut dem i onödan. Bildpunkterna som har valts för utjämningen delas i horisontella och vertikala linjer. Utgående från denna indelning väljs bildpunktspar som är vinkelräta mot kanten med största kontrasten. Därefter söks ändpunkter hos kanter som hittas då ett stort hopp i medelkontrasten mellan pixelpar längs de definierade linjerna detekteras. I ändpunkterna görs 90 graders skiftning i delpixlar för att minska vikning. Inputstexturen samplas om enligt denna subpixelförskjutning och ett lågpassfilter blandas beroende på frekvensen av subpixelvikning. [21][24]

Det finns massvis med olika varianter av redan existerande algoritmer för kantutjämning med sina egna för- och nackdelar. Det är inte heller ovanligt att dagens spelmotorer stöder flera kantutjämningsalternativ. Generellt kan man säga att metoderna oftast består av identifiering av artefakter, insamling av mätvärden och ett sätt att blanda färgerna. Såsom med de flesta efterbehandlingseffekterna gäller det även för kantutjämning att överväga om det är motiverat att använda ytterligare klockcykler på finslipning av en färdigt renderad bild. Ju högre upplösning som används desto långsammare sker utjämningen av kanterna och ju högre bildupplösning man har desto mindre blir andelen av artefakter i jämförelse med antalet pixlar. Därför ger i synnerhet datorspel möjligheten att koppla på eller av denna funktionalitet. I skärmar med stor upplösning relativt arean kan nyttan vara obetydlig, om man överhuvudtaget ser skillnaden, medan till exempel med projektionsapparater förstoras bilden så mycket att taggigheten kan vara till och med störande.

5. DISKUSSION

I denna avhandling har jag presenterat principerna för stort dynamisk omfång och kantutjämning. För båda gäller att man försöker tänja på begränsningarna som hårdvaran sätter. Med HDR försöker man visa större kontraster och bibehålla detaljer som med normal rendering skulle bara tappas bort. Tonmappning är en metod för att överföra HDR-data till displayapparater med lägre dynamiskt område på ett sådant sätt att så lite information som möjligt går förlorad. Med kantutjämning emellertid försöker man visa noggrannare bilder än upplösningen på skärmen egentligen ger möjlighet till genom att sampla värden från subpixelnivå. Användningen av HDR och kantutjämning resulterar i en skarpare bild än scenen utan dem.

För att det över huvud taget skall löna sig att försöka konstruera en verklighetstrogen värld med fotorealistiskt rendering, bör implementationerna vara snabba, ty videospel är interaktiva applikationer och rutan måste uppdateras flera gånger i sekunden för att uppvisa kontinuerliga rörelser. Fragmentskuggaren är en utmärkt lösning för detta problem. Hårdvaran kan accelerera processerna genom att köra massvis med skuggare parallellt. Därför stöder flera spelmotorer någon form av skuggningspråk direkt.

Inom datorgrafik har spelindustrin i redan åratal varit vägvisaren och har gått hand i hand framåt med utvecklingen av grafikprocessorer. Bland annat inom fotografi har HDR-tekniken blivit populär efter stora framsteg inom fotorealistisk rendering i videospel. Också de alltmer populära 3D-animationerna får ibland nya tekniker från innovationer inom spelgrafik. Största skillnaden mellan 3D-film och interaktiv grafik är renderingstekniken. Bilder för animationer kan renderas i flera timmar, medan grafikmotorn i spelmotorer bör kunna generera tiotals bilder i sekunden av en dynamisk tredimensionell värld där kameran kan beroende på input förflyttas varsomhelst närsomhelst. Det är dock möjligt att även spelindustrin i framtiden går över till strålföljning från rastreringsbaserad rendering. För det krävs det dock mycket effektivare algoritmer, men några av efterbehandlingseffekterna är redan kompatibla med strålföljning och prestandan på dem ser mycket lovande ut.

Litteraturlista:

[1] R. Rouse, "What Players Want," i *Game Design – Theory & Practice*, 2. uppl., Texas, Wordware Publishing, Inc. 2005

[2] David Eberly, ”A Brief Motivation”, *3D Game Engine Design – A Practical Approach to Real-Time Computer Graphics*, Morgan Kauffman Publishers, 2000

[3] Anders Grönlund, ”Tillbaka till framtiden i 3D”, *Monitor*, 2011/12, s. 40–45

[4] Ron Fosner, ”Introduction to Shaders”, *Real-Time Shader Programming – Discovering DirectX 9.0*, Morgan Kaufmann Publishers, 2003

[5] S. St-Laurent, *Shaders for Game Programmers and Artists*, Boston, Thomson Course Technology PTR, 2004

[6] D. Wolff, *OpenGL 4.0 Shading Language Cookbook*, Birmingham, Packt Publishing, 2011

[7] CryENGINE® 1, "CryENGINE® 1 | Crytek" [online]. Tillgänglig: http://www.crytek.com/cryengine/cryengine1/overview (hämtad: 25.3.2012)

[8] K. Myszkowski *et al.* ”High Dynamic Range Video”, *Synthesis Lectures on Computer Graphics and Animation*, vol. 2, nr. 1, B. Barsky, red., Kalifornien, Morgan & Claypool Publishers, 2008

[9] Geoff Richards (2005, oktober). BrightSide DR37-P HDR display [online]. Tillgänglig: http://www.bit-tech.net/hardware/2005/10/04/brightside_hdr_edr/ (hämtad: 22.3.2012)

[10] Contrast Ratio Secrets Uncovered, "Presentation Technology Reviews" [online]. juli 2006, tillgänglig: http://www.presentationtek.com/2006/07/31/contrast-ratio-sectrets-uncovered/ (hämtad: 26.3.2012)

[11] TV Contrast Ratio Explained, "Practical Home Theater Guide" [online], oktober 2011, tillgänglig: http://www.practical-home-theater-guide.com/contrast-ratio.html (hämtad: 26.3.2012)

[12] Carsten Wenzel, "Far Cry and DirectX," i *Game Developers Conference*, San Francisco, 2005 © CRYTEK

[13] Erik Reinhard *et al.* "Photographic Tone Reproduction for Digital Images", *ACM Transactions on Graphics*, vol. 21, nr. 3:e juli, 2002, s. 269,

[14] Hanli Zhao *et al*. "Real-Time Tone Mapping for High-Resolution HDR Images," i *International Conference on Cyberworlds*, Hangzhou, 2008, s. 256–262

[15] *Post-Processing Effects*, [online]. 2008, Copyright Leadwerks Software, Tillgänglig: http://leadwerks.com Katalog: files/Tutorials/CPP Fil: Post-Processing_Effects.pdf (hämtad: 6.3.2012)

[16] Kenny Mitchell, "Volumetric Light Scattering as a Post-Process", *GPU Gems 3*, Addison Wesley, 2007, s. 275–285

[17] Ulf Assarsson, "Realistiska renderingar - spelgrafiken leder utvecklingen", Stiftelsen för Strategisk Forskning [online]. Tillgänglig: http://www.stratresearch.se/sv/Pagaende-forskning/Innovation/Forskare---Innovator/Riktigt-realistiska-renderingar---spelgrafiken-leder-utvecklingen/ (hämtad: 30.3.2012)

[18] Alan Watt och Mark Watt, *Advanced Animation and Rendering Techniques – Theory and Practice*, Addison-Wesley, ACM Press, New York, 1992, s. 119–122

[19] Emil Persson, "Selective Supersampling," i *ShaderX*^{5} *– Advanced Rendering Techniques*, Wolfgang Engel, red., Charles River Media, Boston, Massachusetts, 2006

[20] Tomas Akenine-Möller *et al.*, *Real-Time Rendering*, 3:e uppl. A K Peters, Ltd., Wellesley, Massachusetts, 2008, s.128–129

[21] Jorge Jimenez *et al.* "Filtering Approaches for Real-Time Anti-Aliasing", *The 38*th *International Conference and Exhibition on Computer Graphics and Interactive Techniques*, Vancouver, 2011

[22] Alexander Reshetov, "Morphological Antialiasing," i *Proceedings of High Performance Graphics*, 2009

[23] Timothy Lottes, *NVIDIA FXAA*, 2011, [online]. Tillgänglig: http://timothylottes.blogspot.com/2011/03/nvidia-fxaa.html (hämtad: 1.4.2012)

[24] Timothy Lottes, *FXAA*, (NVIDIA), 2009, [online]. Tillgänglig: http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf (hämtad: 1.4.2012)