CHALMERS

Ljussättning i realtid med dynamisk ambiensvolym

CHRISTIAN NILSSON
ALVAR JANSSON

Examensarbete
Civilingenjörsprogrammet för datateknik
CHALMERS TEKNISKA HÖGSKOLA
Institutionen för data- och informationsteknik
Avdelningen för datorteknik
Göteborg 2006

1 Sammanfattning

Ljussättning i realtid med dynamisk ambiensvolym

Här ges ett praktiskt exempel på hur en spatial datastruktur har använts ihop med en spherical harmonic-representation för instrålat ljus (Irradiance map). Resultatet är realtidsuppdaterad ambient ljusinformation implementerat i ett fullskaligt spel med en dynamiskt ljussatt utomhusmiljö.

2 Abstract

Real-Time lighting with dynamic ambience volume.

Here we present an implementation of real-time updated irradiance mapping based in a full-scale production game. The implementation is based on a spatial data structure combined with a spherical harmonic representation of ambient irradiance information.

3 Förord

Detta är ett examensarbete vid institutionen för data- och informationsteknik på Chalmers tekniska högskola, avdelningen för datorteknik. Arbetet utfördes under perioden september 2005 till februari 2006 vid Avalanche Studios i Stockholm, och handleddes där av Linus Blomberg. Handledare och examinator på Chalmers var Ulf Assarsson.

Kontaktinfo:

Christian Nilsson: tisten@dtek.chalmers.se
Alvar Jansson: alvar@dtek.chalmers.se

3.1 Arbetets uppdelning

Arbetet utfördes av Christian Nilsson och Alvar Jansson. Arbetet fördelades på så vis att Christian till största delen arbetade med datastrukturen och dess implementation i spelmotorn Avalanche Engine. Alvars huvudsakliga ansvar var skapandet av evalueringsrutiner och konverteringen av motorn för att kunna färgsätta objekt med ambiensen.

Generering av ambiensen via rendering av kubtexturer gjordes som en gemensam insats, där Christian arbetade med stöd för kubtexturformatet och rutiner för rendering till detta medan Alvar arbetade med att anpassa renderingsarkitekturen till en av spelarens vy oberoende uppritning.

3.2 Tack

Tack till alla på Avalanche Studios för en inspirerande arbetsmiljö, ert engagemang och för den förstklassiga utrustning vi fick låna.

http://www.avalanchestudios.se/

4 Innehållsförteckning

5 Inledning

Ljussättning i realtidsgrafik brukar inte beräknas på ett fysikaliskt korrekt sätt, utan som en kombination av olika approximationer. Det vanligaste är att man beräknar det som summan av tre termer: diffust, spekulärt och ambient ljus. Den diffusa termen är den mest fysikaliskt korrekta. Den motsvarar hur ljuskällor påverkar matta ytor, och är oberoende av betraktarens position. Spekulärtermen används för att få en illusion av blanka ytors spegling av ljuskällor.

Den ambienta termen är traditionellt en konstant färg som skall approximera hur matta ytor påverkas av bakgrundsbelysningen. Bakgrundsbelysningen kommer från ljus som studsat på närliggande objekt såväl som ljus från atmosfäriska fenomen. Att man valt att förenkla ambiensen på detta sätt är dels för att det är den minst uppenbara delen av ljuset, dels för att det inte finns något effektivt sätt att beräkna den. Approximationen ger dock bilder ett platt utseende, eftersom de enda färgskiftningar som kommer bilden till godo är de som finns i punktljuskällor eller i det ursprungliga objektets textur.

Dagens programmerbara grafikhårdvara har gett nya möjligheter att göra godtyckliga beräkningar på både vertex- och pixelnivå. Ett sätt att utnyttja detta är att skriva om delar av ljussättningsalgoritmen. Om man vill ha en mer noggrann representation av det ambienta ljuset så kan man beräkna det ljus som strålar in mot en punkt från alla riktningar. Att evaluera detta i varje synlig punkt vid varje givet tillfälle är inte praktiskt genomförbart idag, men på senare tid har det gjorts statiska scener som använder förberäknad ambiens. Detta innebär att man för ett antal punkter i en scen lagrar ljusets färg och intensitet i alla riktningar.

Scen: här en virtuell representation av ett rum eller en utomhusmiljö med både geometri och ljuskällor.

Eftersom en matt yta reflekterar ljus som strålat in från hela hemisfären ovanför ytan så blir konsekvensen att förändringar av ambiensen oftast sker med mjuka övergångar, både i tid och rum.

Avalanche Studios är särskilt intresserade av stora öppna miljöer där alla objekt är dynamiskt ljussatta och kan vara procedurellt genererade. Dessa förutsättningar omöjliggör användandet av förgenererad data, som annars är normen för ljussättning med SH.

Grunden till detta arbete är att utnyttja förutsättningarna som nämnts ovan för att skapa en belysningsmodell där ambient lågfrekvent belysning ger en illusion av global indirekt belysning.

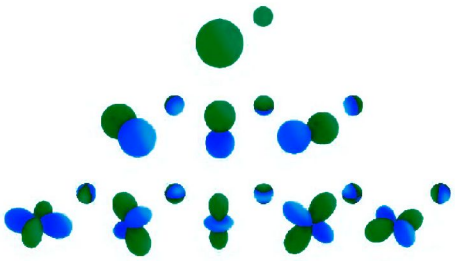
Global indirekt (eller sekundär) belysning är ett vitt begrepp inom datorgrafiken, och heter på engelska Global Illumination (GI).

5.1 Ursprunglig idé

En intressant representation av ambient ljus är spherical harmonic (SH). Det har visats att endast tredje ordningens SH behövs för att representera ambient ljus med stor noggrannhet [25]. Att sedan beräkna en färg från dessa koefficienter och en normalriktning är möjligt att göra med programmerbar grafikhårdvara.

Se *An Efficient Representation for Irradiance Environment Maps* av Ramamoorthi & Hanrahan.

Spherical harmonic är en basfunktion som kan användas för att transformera data från sfäriska koordinater till en frekvensrepresentation. Den första ordningen har bara en koefficient och representerar ett konstant värde. Andra ordningen har 3 koefficienter, och sparar linjära skiftningar över de tre kartesiska axlarna. Tredje ordningen lagrar de kvadratiska förändringarna, här krävs totalt nio koefficienter.



Grafisk representation av de tre första ordningarna av SH, grönt motsvarar positiva värden, blå negativa.

6 Tidigare arbeten inom området

Det har skrivits många papper och gjorts ett flertal experiment inom området tidigare. Vi listar här några av de mest framgångsrika exempel vi hittat, samt de arbeten som vi blivit inspirerade av.

6.1 Irradiance Volumes for Games, 2005

Det arbete som är mest likt detta heter "Irradiance Volumes for Games" och är skrivet av Chris Oat på ATI. I det används DirectX inbyggda PRT-motor för att förberäkna instrålningen till en mängd punkter i rymden. Dessa punkter interpoleras sedan, och används ihop med ambient occlusion på rörliga objekt. Intressanta punkter i detta arbete är datastrukturen som är ett octree, samt interpoleringsmetoden som används. [10]

6.2 The Irradiance Volume, 1998

Den som först visade på användningen av volymer för ljussättning var Greger m.fl. från Cornell University i sitt arbete "The Irradiance Volume". De använde ett octree med data i hörnen av kuberna. Metoden i deras papper var dock inte lämplig för realtidsanvändning, då de använde sig av raytracing. De använde sig dessutom av en matris av diskreta ljusriktningar som representation. [6]

6.3 Cube-map data structure, 2002

Ett arbete som handlar om liknande saker bär namnet "Cube-map data structure for interactive global illumination computation in dynamic diffuse environments" och är ett samarbete mellan Rafal Mantiuk, Sumanta Pattanaik och Karol Myszkowski från University of Central Florida, Technical University of Szczecin i Polen samt Max-Planck-Institut für Informatik. Här vill man approximera flera nivåer av studsande ljus med hjälp av upprepade cube-mapsamplingar. De använder sig av Irradiance volume, men har ingen typ av hierarkisk uppbyggnad som i The Irradiance Volume. [15]

6.4 Real-Time Global Illumination, 2003

Mangesh Nijasure arbetade vidare på "Cube-map data structure" tillsammans med Sumanta Pattanaik, båda från University of Central Florida, samt Vineet Goel från ATI. Det nya arbetet döptes till "Real-Time Global Illumination on GPU" och fokuserade på att hålla beräkningarna på GPUn. De lyckades visualisera ett rum i realtid med 2 ljusstudsar om inga objekt i rummet var dynamiska. De tog även hänsyn till indirekt skuggning med hjälp av z-buffern. [22]

6.5 Real-Time Computation, 2005

Gary King använder sig av Spherical harmonics för att i realtid falta en environment map, som han sedan konverterar tillbaka till texelrepresentationen för att ljussätta objekt. Arbetets fulla titel är "Real-Time Computation of Dynamic Irradiance Environment Maps". [10]

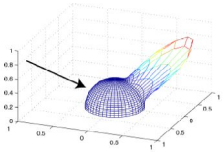
6.6 Irradiance Environment Maps, 2001

Arbetet som fick oss intresserade av ämnet är den korta klassikern "An Efficient Representation for Irradiance Environment Maps" skriven av Ravi Ramamoorthi och Pat Hanrahan på Stanford University. Här ges både matematisk bakgrund och förslag på implementation av metoder för generering och evaluering av spherical harmonics. [25]

6.7 The Gritty Details, 2003

En mycket längre och mer rigorös beskrivning av både matematiken bakom spherical harmonics och dess användningsområden inom PRT ges i "Spherical Harmonic Lighting: The Gritty Details" av Robin Green på Sony. [5]

6.8 Foundations of PRT, 2004



En liknande uppsats som också den är fokuserad på den matematiska aspekten av spherical harmonics och dess tillämpning inom PRT heter "Foundations of Precomputed Radiance Transfer" och är ett examensarbete av Jaakko Lehtinen på tekniska högskolan i Helsingfors. [13]

7 Analys

En sammanfattning av de förutsättningar som fanns inför vårt arbete återges och analyseras i detta kapitel. Planeringen av arbetet byggde på de tidiga insikter som dokumenteras här.

7.1 Önskat resultat

Med detta arbete vill vi uppnå en illusion av att omgivningens färger har ett inflytande på vilken färg ytan på ett objekt har.

TV: Den blå himmeln färgar kvarnens skugga blå.

TH: Pandan i den gröna skogen blir svagt grön i pälsen av omgivningens ljus.

Denna påverkan sker genom att ljus strålar ut från en yta eller kommer in från atmosfären, träffar en annan yta och därefter når betraktaren. Det ljus som strålar mellan ytor på detta sätt kallar vi ambiens.

7.2 Uppdelning av problemet

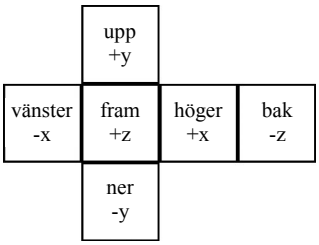
Problemet att simulera ambiens består i stort av fem delar:

1. Sampla omgivningen för att ta fram en representation av det infallande ljuset till ett antal punkter i scenen.
2. Spara ljuset i en effektiv representation och ta fram en instrålningstextur (irradiance map).
3. Lagra dessa ljusrepresentationer i en lämplig datastruktur.
4. Hämta ut ljussättningar ur denna datastruktur för positioner där objekt befinner sig.
5. Evaluera ljusrepresentationen till färger som kan renderas på ett grafikkort.

En irradiance map är en det infallande ljuset mot en punkt på en matt yta som en funktion av ytnormalen.

Dessa delar måste naturligtvis kompromissas för att passa samman med varandra och fungera i ett realtidssystem. En av de viktigaste punkterna är att det inte skall behövas någon förberäkning av ljussättningen. Med undantag för [10] görs steg 1-3 som ett förberäkningssteg (offline) i de arbeten vi studerat.

7.3 Sampling av omgivningen



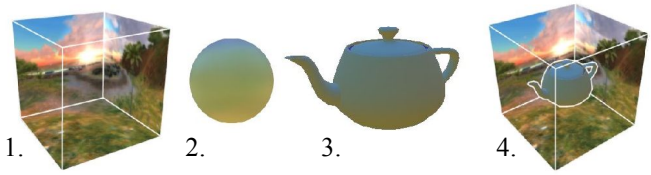
För att få en fullständig bild av hur omgivningen ser ut vid en viss punkt, måste man skapa en projektion som rastrerar all geometri som omger punkten. Med en avancerad shader skulle hela omgivningen kunna renderas på ett plan. Detta skulle dock ge stora distorsioner.

Det normala då man skall använda grafikkortet för att göra en bild av omgivningen är att projiciera omgivningen mot var och en av sidorna på en kub som omger punkten. Man bildar då en kubtextur, som består av sex mindre texturer.

7.4 Generering av ambiens

För att kunna omvandla kubtexturen till information om det instrålade ambienta ljuset i en punkt så behöver man först bestämma sig för vilken representation som ska användas för att lagra informationen.

Omgivningen (1) ger det instrålade ljuset (2), som kan appliceras på objekt (3) så att de får samma färg som om de hade befunnit sig i omgivningen från början (4).



Att använda spherical harmonic (SH) som representation för att beskriva belysningen i en punkt har blivit populärt de senaste åren. Idag finns det färdiga funktioner i DirectX för att ta fram koefficienter för SH som evaluerar ljusinverkan från punktljuskällor, hemisfäriskt ljus, riktat ljus samt riktat ljus med cirkulär area. Instrålande ljus från omgivningen beskrivet av en kubtextur (s.k. light probe) kan projicieras som en sfärisk påverkan. Dessutom finns det hjälpfunktioner för att rotera och skala en vektor av SH-koefficienter, samt addera och skalärmultiplicera dem.

För att läsa mer om SH, se Lehtinens “Foundations of PRT” [13] eller Greens “The Gritty Details” [5].

En stor fördel som SH ger är att representationen av samplingar finns i frekvensplanet, vilket gör att den nödvändiga faltningen går att utföra mycket effektivt. Det finns dessutom gott om referensmaterial att tillgå, så valet av SH som representation ter sig fördelaktigt.

Ett alternativ som använts av Valve [16] när de skapade sin Source-motor är att lagra instrålat ljus för 6 riktningar. Evalueringen görs när en bana skapats i editorn. Detta ger en snabb implementation och låga minneskrav, men det är omständligt att ta fram dessa 6 värden. Om man förberäknar värdena som i Source slipper man ta hänsyn till detta.

Ett annat sätt att lagra ambiensen är att behålla kubtexturens texelrepresentation och göra om denna till en irradiance map. King [10] hävdar att detta är lämpligt eftersom det ger snabba texturuppslag istället för vektor- och matrisoperationer. En negativ detalj är att det är jobbigt att interpolera mellan dessa och det kräver mycket minne.

Efter att man skapat en bild av omgivningen i en kubtextur och projicierat denna till SH, måste man göra om denna till en

instrålningstextur. Varje riktning i instrålningstexturen skall bestå av en cosinusviktad integral över halvsfären runt denna riktning i den ursprungliga omgivningsbilden.

Hade man använt en texelrepresentation av omgivningsbilden hade denna faltning varit mycket tung att göra. För varje element i den slutliga irradiancetexturen hade man varit tvungen att integrera över halva kubens pixlar. Med SH räcker det att multiplicera koefficienterna med konstantvärden för att uppnå samma resultat. För en mer utförlig matematisk förklaring hänvisar vi till Ramamoorthi & Hanrahans papper ”An Efficient Representation for Irradiance Environment Maps”.

Ramamoorthi & Hanrahan presenterar bevis på detta [25].

I deras papper visas det även att det maximala felet för en pixel, om man projicierar bilden av omgivningen på en sfär och approximerar denna med andra gradens SH, faltar och sedan transformerar tillbaka, jämfört med en korrekt analys, blir maximalt 9 %. Det kan även visas att medelfelet är under 3 % för alla fysiska ljusförhållanden.

För att ha möjlighet att anpassa projicieringen av koefficienter till de tidsbegränsningar som råder i en spelmotor så valde vi att implementera en egen projicieringsalgoritm från kubtextur till SH koefficienter. Med hjälp av denna går det att dela upp beräkningsbördan för projicieringen över flera skärmuppdateringar. Implementationen gjorde vi även för att få en inblick i matematiken bakom transformen, samt för att lära oss grunderna om DirectX som vi tidigare inte arbetat med.

7.5 Datastruktur

Datastrukturens huvudsakliga uppgift är att lagra ambient information med hänsyn till dess position i scenen samt att validera dess temporära giltighet. Den ambienta informationen, som består av SH koefficienter, ligger till grund för att snabbt kunna interpolera fram nya koefficienter för godtyckliga positioner i scenen. Datastrukturen är även vara till hjälp vid val av nästa position där nya koefficienter ska genereras, eftersom den har information om vart giltig data om omgivningens ambienta färg redan existerar.

Datastrukturen ska klara av att lagra ambient information i 3 dimensioner med variabel täthet. Informationen används sedan för att belysa objekt på valfri position. Bild från “Cube-map data structure” [6].

7.5.1 Alternativ

Frågan har kommit upp om det över huvud taget är nödvändigt med en datastruktur, om det inte istället hade varit lämpligare att generera en ambiens för varje objekt. Att göra så skulle inte vara helt utan förtjänst. Dels skulle man slippa interpolera mellan flera olika koefficienter, dels skulle ljuset bli korrektare eftersom man undersökte omgivningen precis runt det belysta objektet.

Det finns åtminstone två faktorer som gör användandet av generering per objekt problematiskt. Dels är det objekt som behöver nya koefficienter med tiden (t.ex. om de kan förflytta sig i en scen eller om det är ljusförändringar i ett objekts omgivning) och dels är det om antalet objekt är stort. För att exemplifiera problemen:

För förklaringar av vad LOD och billboards är så läs "Real-Time Rendering" av Akenine-Möller och Haines [1].

- Kameran färdas mot ett skogsbryn i hög hastighet. Vid övergång mellan två detaljnivåer (LOD) går man från en billboard med många träd till enskilda trädmodeller samt stenar och stubbar. Det krävs 6 bilduppdateringar för att generera en uppsättning koefficienter eftersom varje sida av kubtexturen tar lång tid att rendera. Detta diskuteras i avsnittet "Metod för stickprov av ambiens" i nästa kapitel. Varje sekund instansieras 50 nya objekt vilket kräver en bilduppdateringshastighet på 300 bilder i sekunden vilket är orimligt. Samma scenario gäller för t.ex. för städer, där antalet stolpar, markiser och dylikt snabbt gör situationen ohanterlig.
- Man färdas på en motorväg i en kuperad terräng där bergväggar och stup susar förbi, hastigheten är 110 km/h (30 m/s) och trafiken är någorlunda tät. Det syns i snitt 10 fordon per sekund på egna samt mötande körbanan sammanlagt. För att inte missa snabba färgförändringar i terrängen kan vi anta att man behöver uppdatera var 5:e meter. Denna scen skulle utan statiska objekt som ska ha eget ambient data kräva 60 nya uppsättningar av koefficienter per sekund vilket i sin tur skulle kräva en bilduppdateringshastighet på 360 bilder i sekunden.

  Ett annat problem med att ha en ambiens per objekt är att dynamiska föremål får sin korrekta belysning för sent, tidigast 6 bilduppdateringar efter att de varit vid en given position. Om man vill undvika att ljuset ändras snabbt så behöver man interpolera fram den nya färgen. På så vis får man ytterligare fördröjning samt behöver ändå interpolera, något denna lösning var tänkt att undvika. För att förhindra fördröjningen så krävs att man förutspår vart man kommer att befinna sig några bilduppdateringar in i framtiden. Detta har en inbyggd osäkerhet, vilket betyder att även argumentet att ambiensen skulle blir mer korrekt faller.

7.5.2 Motivering

Följande fördelar gjorde att beslutet föll på att använda en datastruktur att lagra koefficienter i:

- Med hjälp av en datastruktur som ger möjlighet att interpolera fram koefficienter som kombinationer av vad som finns i omgivningen så att det går att använda ett fåtal koefficienter till ett stort antal objekt.
- Koefficienter till större ytor som väggar och terräng är lämpliga att lagra i en datastruktur eftersom de då lätt kan återanvändas av objekt på dessa ytor.
- Man får mjuka färgövergångar på grund av interpolering när man förflyttar sig mellan befintliga koefficienter.
- Det går att justera tätheten så att fler koefficienter skapas nära kameran där skillnaden syns tydligast.

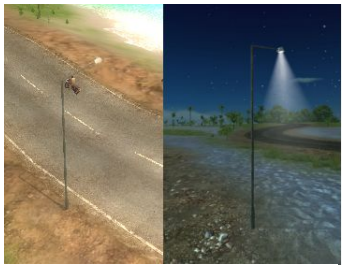
7.5.3 Specialiseringar

Ett mål för datastrukturen är att hålla en mängd koefficienter, tillräckligt många för att kunna ljussätta alla synliga objekt som skall ritas upp med uträknad ambiens. Då data saknas för en punkt, måste man snabbt generera ny. Vad man kan göra är att ställa upp en prognos för hur objekt kommer att röra sig och på så sätt försöka förutsäga punkter som är lämpliga att generera koefficienter på. En viktig aspekt att ta hänsyn till vid design av datastrukturen är vilken strategi man tänker använda för att utse dessa punkter.

Det går visserligen att göra en förflyttningsprognos som är så generell att den går att använda till alla typer av applikationer. Dock kan man få en mer korrekt prognos genom att känna till särskilda förhållanden i applikationen, och en bättre prognos ger ofta avsevärda visuella förbättringar eftersom informationen man genererar då är till bättre nytta. Det är därför viktigt att tänka igenom de begränsningar och möjligheter som den grafikmotor och spelmiljö man ska använda erbjuder, och optimera sin prognos efter dessa. Sådana optimeringar påverkar i hög grad datastrukturens design och funktion.

Några fall där man kan optimera är:

- Om markplanet är skapat av en höjdkarta så får man automatiskt en lägsta höjd där man behöver skapa koefficienter. Datastrukturen kan dra nytta av det golv som höjdkartan skapar.
- Om alla objekt befinner sig samma nivå, t.ex. precis ovanför ett markplan och inte svävar högt upp eller nere i tunnlar, då räcker det att generera ett plan av koefficienter på denna höjd. Detta gör att datastrukturen bara behöver spara koefficienter i ett 2Dgrid.



Effekten av en gatlykta som tänds och släcks skulle effektivt kunna implementeras med en extra uppsättning koefficienter.

- Vissa ljuskällor kan vara statiska i rummet, men ändå ändra sin ljusstyrka över tiden. Detta gäller för bland annat gatlyktor, som ju stängs av på dagen. Ett sätt att låta dessa påverka ambiensen är att beräkna särskilda koefficienter som bara innehåller denna ljuskällas påverkan av omgivningen. Denna särskilda ambiens kan sedan adderas till den omgivande vid de tillfällen då ljuskällan är påslagen. Detta gör att ljuskällan kan slås på och av utan att en hel ambiensberäkning måste ske. Hur mycket denna specialisering påverkar det visuella skiljer från situation till situation. Det kan göra påtaglig skillnad för en punktljuskälla i inomhusmiljö eller en gatlykta men är kanske inte ens märkbart för en signallampa t.ex. ett trafikljus. Detta fall kräver att datastrukturen kan lagra mer än en uppsättning koefficienter per position.
- Dynamiska ljuskällor samt dygnscykel gör att befintliga koefficienter behöver genereras på nytt med jämna tidsintervall. Dessutom tillkommer ett krav på att datastrukturen ska kunna interpolera mellan de befintliga och de nya koefficienterna för att få mjuka ljusändringar. Om man kan förutsäga den maximala hastigheten med vilken de relevanta ljusförhållandena ändras, kan man också ge en minsta möjliga hastighet i vilken koefficienterna behöver genereras om.
- Om kamerans rörelser är förutsägbara så går det att öka upplösningen av koefficienter på rätt ställen, t.ex. om det finns en väg som en bil kör på, så är det sannolikt att föraren följer vägen. Om det är viktigt att kunna placera koefficienterna på exakta positioner så bör man fundera på om datastrukturen ska ha förbestämda positioner där data kan lagras eller om den ska vara mer generell.

  Man bör även ta hänsyn till att det inte finns någon poäng att generera koefficienter inuti solida objekt. Det kan rent av vara något man skall undvika eftersom ljusförhållandena inuti objekt annars kan ”läcka ut” i omgivningen p.g.a. interpolering.

Steradianer är ytenheten för sfärer, totalt består en sfär av 4π steradianer.

Hur man effektivt tar hand om problemet med samplingar som blir fel nära objekt behandlas i ett papper av Samuli Laine, ”*Sampling Precomputed Volumetric Lighting”* som kommer att dyka upp i Journal of Graphics Tools.

Koefficienter blir felaktiga snabbare ju närmre ett objekt de ligger eftersom koefficienterna påverkas av hur många steradianer av sfären som täcks av objektet, och nära ett objekt ändras detta fort. Sådana snabba ändringar strider mot antagandet att förändringar av ambiensen sker långsamt, med mjuka övergångar. Genereringen av koefficienter är även känslig för objekt som kan flyttas, t.ex. fällbara träd, parkerade bilar eller luftballonger redo att lyfta. Objekt som rör på sig fortare än man genererar koefficienter bör inte alls tas med vid genereringen.

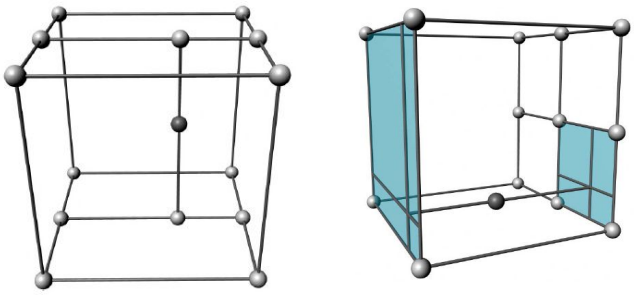
Ännu ett antagande man kan göra är att korrektheten på ambiensen blir mindre viktig ju längre bort ett objekt är. Detta i kombination med att detaljrikedomen i scenen kan skifta kraftigt mellan olika platser gör det lämpligt att datastrukturens spatiala uppbyggnad är hierarkisk. Det finns gott om exempel på hierarkiska spatiala datastrukturer, bl.a. octree, KD-tree och hierarkiska grids.

Eftersom ATI [23] i sin implementation hade använt ett octree med framgång, ansåg vi det lämpligt att använda detta som en utgångspunkt.

7.6 Interpolering

För att använda sig av informationen i datastrukturen så behöver man kunna hämta ut koefficienter till en sfärisk funktion för varje punkt i hela scenen, även de ställen där datastrukturen inte har någon data. För att hämta ut användbar data behöver man därför interpolera fram värden ur den information som finns.

Linjär interpolering mellan befintlig information ger bra resultat när samplingarna ligger tätt eller objekten som är med i samplingarna ligger långt borta och/eller är stora. Övergångarna blir långsamma och mjuka. En volym kräver trilinjär interpolering vilket fungerar bra så länge informationen är konstant.

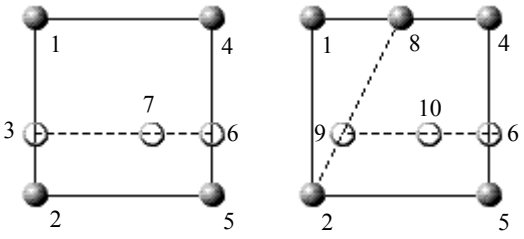


Exempel på linjärinterpolering, till vänster för en uniformt uppdelad struktur där informationen finns i hörnen. Till höger är en av sidorna delad i 4 områden och har därför 9 uppsättningar koefficienter istället för 4.

Ett problem som uppkommer då ny information läggs in eller gammal information tas bort är att det sker plötsliga färgförändringar. Sådana ändringar märks tydligt visuellt. Att detta händer när färginformationen ändras kan te sig självklart, men även om man skulle ändras information genom att interpolera fram förändringar över tid så kan linjärinterpolering ge problem om interpoleringsmönstret ändras.

Exempel på färgförändring vid ändring av interpoleringsmönstret:

Problemet med linjärinterpolering vid insättning uppkommer då interpoleringpunkterna ändras. I illustrationen till höger skapas noden som kallas 8. Även om nod 8 sätts till linjärinterpolationen mellan nod 1 och nod 4 så blir inte resultatet för nod 7 bli samma som resultatet för nod 10.



För att interpolera fram färgen i punkten 7 ovan används funktionen lerp(a,b,c) som representerar linjärinterpolationen:

$$lerp(a,b,c)=a+c\cdot(b-a) \qquad 0\leqslant c\leqslant 1$$

vilket ger följande resultat:

$$3=lerp(1,2,0.6) \quad 6=lerp(4,5,0.6) \quad 7=lerp(3,6,0.7)$$

$$7=1\cdot 0.12+2\cdot 0.18+4\cdot 0.28+5\cdot 0.42$$

När den nya punkten 8 läggs in så ändras interpoleringsmönstret och då spelar det ingen roll om färginformationen ändrats eller inte. Om vi antar att punkt 8 representerar färgen som ges vid interpolering mellan punkt 1 och 4 så får vi följande resultat:

$$8=lerp(1,4,0.5) \quad 9=lerp(8,2,0.6) \quad 10=lerp(9,6,0.625)$$

$$10=1\cdot 0.06+2\cdot 0.18+4\cdot 0.34+5\cdot 0.42$$

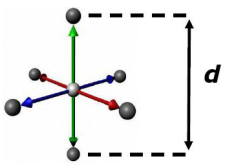
Ändringen av interpoleringsmönstret gör att färgen från punkt 4 blir mer betydelsefull och färgen i punkt 1 får mindre betydelse.

Det finns dock flera lösningar på problemet. Man kan t.ex. se till att bara infoga nya punkter på ställen som bevarar interpoleringsmönstret. Nu kvarstår dock problemet att den nya informationen troligen innehåller ny färginformation.

Problemet går att lösa genom att både beräkna färgen som det gamla interpoleringsmönstret ger och färgen som det nya mönstret ger, och sedan interpolera mellan dessa över tid. Med flera förändringar i samma område kan denna lösning bli beräkningsmässigt tung eftersom beräkningarna behöver göras för varje gång man plockar ut ett värde från det område där förändringen är.

Att beräkna gradienten och använda den för interpolering är en annan lösning som ATI [23] föreslår för att skapa korrektare övergångar. Detta ger korrektare interpolering men kräver 6 gånger fler stickprov för att skapa gradienterna. Eftersom vårt mål är att ambiensen ska genereras i realtid så sänker detta antalet samplingar till en sjättedel, vilket gör denna metod oanvändbar för oss.

ATI föreslår att man genom "Central Differencing" tar fram gradienten i x, y och z led för varje samplingspunkt och interpolerar med denna [23]. Metoden går ut på att sampla 6 gånger runt originalsamplingen och sedan beräkna derivatan längs varje axel.



$$\nabla_y = \frac{y_{+1} - y_{-1}}{d}$$

7.7 Evaluering

För varje objekt som ska renderas i en scen hämtar man ut en uppsättning SH koefficienter L_{lm} från datastrukturen. Att evaluera den sfäriska funktionen från en SH är enkelt om man har ytnormaler för objektet som skall färgsättas. Koefficienterna L_{lm} används för att skapa en matris *M* som tillsammans med objektets ytnormaler $\boldsymbol{n}$ beskriver belysningen i varje punkt på objektet. Matrisen *M* är symmetrisk och ser ut som följer:

Matrisen *M* och ekvationen $E(\boldsymbol{n})$ nedan är härledd av Ramamoorthi & Hanrahan [25].

$$M = \begin{pmatrix} c_1 L_{22} & c_1 L_{2-2} & c_1 L_{21} & c_2 L_{11} \\ c_1 L_{2-2} & -c_1 L_{22} & c_1 L_{2-1} & c_2 L_{1-1} \\ c_1 L_{21} & c_1 L_{2-1} & c_3 L_{20} & c_2 L_{10} \\ c_2 L_{11} & c_2 L_{1-1} & c_2 L_{10} & c_4 L_{00} - c_5 L_{20} \end{pmatrix}$$

Belysningen *E* för varje position med ytnormal $\boldsymbol{n}^t = (x\ y\ z\ 1)$ beräknas

$$E(\boldsymbol{n}) = \boldsymbol{n}^t M\ \boldsymbol{n}$$

En sådan beräkning behövs för varje färgkanal.

Beräkningarna sker på grafikkortet i en shader vilket ger hög effektivitet genom parallella beräkningar och snabba matris- och vektoroperationer.

Eftersom belysningen varierar med låg frekvens kan det tyckas fördelaktigt att evaluera SHn per vertex. Det finns dock modeller som har en ytstruktur bestående av en normaltextur. Denna kan växla riktning totalt mellan två texturelement.

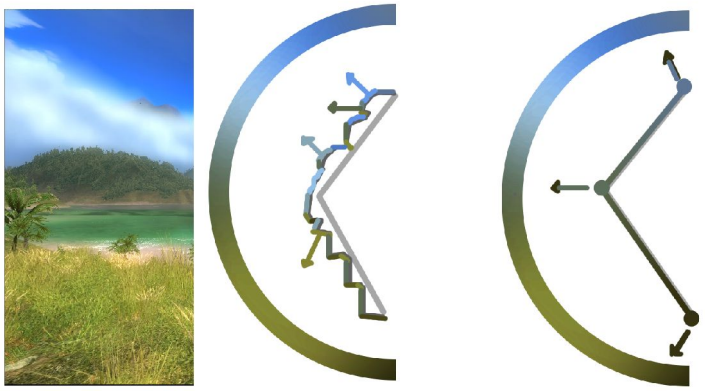
Gouraud shading: Räkna ut färgvärden för vertexar och linjärinterpolera dessa över polygonerna. (se ill. ovan)

Phong shading: Linjärinterpolera ytnormalerna över polygonen och beräkna om ljusbidraget för varje pixel.

Om man istället vill ljussätta modeller som inte har en normaltextur, så räcker det att evaluera per vertex. Detta beror på att högfrekventa komponenter varken finns i ljussättningsfunktionen eller i en sådan ytstruktur. Skillnaden kan enkelt testas genom att jämföra Gouraudshading med Phongshading av en ljuskälla som befinner sig på oändligt avstånd. En sådan ljuskälla motsvarar den högsta detaljrikedom som finns representerad i en irradiance map.

TH: Med en evaluering vid varje vertex förloras ambiensens effekt på ytstrukturen.

TV: Om man vid varje texturelement i normaltexturen gör en evaluering träder ytstrukturen fram även vid högre detaljrikedom.



Då modellen har en normaltextur blir förhållandet annorlunda, frekvensen mellan färgbytena kan bli lika hög som den hos normaltexturen. Ett exempel är en golfares vita tröja, av vilken vi ser ett tvärsnitt i figuren ovan. Denna är grovt modellerad, men har en normaltextur för att ge den finare detaljer så som skrynklor. Ovansidorna av skrynklorna blir färgade blå av himmeln, medan kanterna på dem blir gröna av träden och gräset som kantar banan. Undersidorna av skrynklorna blir gulfärgade av sanden i bunkern han står i.

Det står klart att vi måste använda oss av olika metoder för att evaluera färgerna, beroende på vilken typ av objekt som skall färgsättas i och vilken detaljnivå dessa har.

8 Metodbeskrivning

Efter att ha läst på om ett antal aspekter av problemet påbörjades arbetet med att implementera olika delar av lösningen. Det skall dock noteras att underrubrikerna nedan inte kommer i kronologisk ordning. Delar av implementationen skedde dessutom parallellt med analysen. Som plattform för implementationen startade vi med en egenutvecklad demoapplikation med få primitiver och enkla shaders. När vi sedan verifierat att grunderna fungerade så flyttade vi över utvecklingen till Avalanche Engine för att testa hur algoritmen fungerade i ett större sammanhang.

Det visade sig snart att de två största arbetsområdena inom implementationen skulle bli datastrukturen och koden för generering av kubisk sampling samt projiciering av denna till SH.

8.1 Algoritm för planering av samplingar

Det står klart att man bara kan hålla ett begränsat antal ambienser uppdaterade. För att bestämma vilka punkter som är viktigast att generera och hålla uppdaterade har en algoritm utvecklats.

Algoritmen baserar sig på information om kamerans position och hastighet, samt hur fördelningen av befintliga samplingar ser ut. I algoritmen görs antagandet att kameran, det vill säga den punkt där ambiensen spelar störst roll, har en inbyggd tröghet. Kameran bör alltså förflytta sig kontinuerligt i ungefär samma riktning under en tidsperiod av flera bilduppdateringar för att algoritmen ska ge bra resultat. Finns det redan gott om information kring kamerans estimerade framtida position så breddas det högupplösta området runt denna punkt.

8.2 Metod för stickprov av ambiens

För att göra ett stickprov av omgivningens ljusförhållande i en punkt renderas en kubtextur med kamerans centrum i denna punkt. För att skapa en bild som håller sig giltig under en längre tidsperiod så tas alla dynamiska objekt bort från scenen. Övriga objekt i världen renderas precis som vanligt.

Microsoft har gjort ett demo som följer med DirectX vilket heter HDRCubeMap [17]. Där visas bland annat visar hur man renderar en scen till en kubisk textur.

Eftersom alla statiska objekt i en scen ritas upp kan resultatet bli att det tar lika lång tid att rendera en sida av kubtexturen som det gör att rendera det som syns på skärmen. Uppdateringshastigheten kan i värsta fall bli så dålig som 1/7 av vad den var utan samplingsfunktionen. En möjlighet att minska tiden det tar att göra en sampling är att sprida ut renderingen av kubtexturen över 6 bilduppdateringar. Detta innebär visserligen att endast 1/6 så många koefficienter kan hållas uppdaterade, men ger en både mindre och jämnare beräkningsbörda under varje bilduppdatering.

De sex sidorna i kubtexturen ger data om färgerna i omgivningen trots låg upplösning.

Efter att samplingen är gjord så ska bilden projicieras på en sfär och approximeras med andra gradens spherical harmonic koefficienter. Tiden för projicieringen och transformationen är direkt beroende av antalet pixlar i samplingen. Därför är det en god idé att redan vid rendering minska antalet pixlar till det minsta acceptabla för evalueringen av koefficienter. Med kraftig reducering av upplösningen kan man även sänka kraven för det som ska renderas. Detaljnivån kan sänkas till minsta på modeller, avancerade moln kan ersättas med en ruta i rätt färg och små

detaljer kan plockas bort helt. Allt detta gör att det går snabbare att sampla, utan att färgfelet blir signifikant.

8.3 Approximering av kubisk sampling

För att generera SH av den färdigrastrerade kubtexturen måste ett antal operationer utföras på textursidorna. Det finns färdiga rutiner för detta i DirectX, men för ökad flexibilitet och möjlighet till optimering valdes att implementera en egen variant.

Utgångspunkten var att skapa en implementation som ger åtminstone lika bra utdata och är lika snabb som DirectX's D3DXSHProjectCubeMap(), samt går att bryta upp i flera delar, eller alternativt att skapa en funktion som går snabbare än DirectX's funktion på ny grafikhårdvara men eventuellt inte går att bryta upp på detta sätt.

För att projiciera en sampling i form av en kubisk textur (***P***) till SH-koefficienter ($\boldsymbol{C}_b$) där index *b* anger koefficienten (1 till 9) så behöver man vikta varje texturelement från den kubiska texturen med en projicieringsfunktion som omvandlar en kubisk sampling till en sfärisk, samt evaluera varje pixels inverkan för den aktuella koefficienten. Både koefficientens basfunktion samt projicieringsfunktionen kan försparas i en textur ($\boldsymbol{B}_b$) för att sedan användas vid evaluering. Produkterna av samplingens pixelvärden och den förgenererade viktningsfunktionens pixelvärden summeras, och divideras sedan med antalet pixlar per sida samt en normaliseringskonstant $\pi/4$ för sfärens yta. I formeln nedan har den kubiska texturen (***P***) sidan *s* texels (texturelement) och *f* anger vilken sida i texturen som avses.

$$\boldsymbol{C}_b = \frac{4}{\pi} \cdot \frac{1}{s^2} \sum_{f=1}^{6} \left(\sum_{t=1}^{s^2} \boldsymbol{P}_f(t)\, \boldsymbol{B}_{bf}(t) \right)$$

8.3.1 Försök 1, GPU-baserat

Flera bildelement kan rastereras samtidigt på en parallell GPU, eftersom de inte kan påverkas av annat än konstanter, interpolerat vertexdata samt texturer.

Dagens GPUer har stor parallell prestanda, så det skulle vara praktiskt om man kunde multiplicera en kubtextursida ($\boldsymbol{P}_f$) med en sida i basen ($\boldsymbol{B}_f$) i GPUn. Detta skulle kunna ske genom att man använder sig av en multiplikation i en pixelshader, vars resultat man för en löpande summa på. Tanken var att eftersom ingen av kubtexturerna skulle behöva lämna texturminnet på grafikkortet belastas inte heller bussen mellan internminne och grafikminne.

8.3.1.1 Design och implementation

Texturerna som skulle multipliceras var en sampling av omgivningen (***P***) samt förgenererade texturer innehållandes baserna (***B***). Dessa förgenererade texturer består både av en viktningsfunktion som projicierar kubtexturen på en sfär och en av baserna för en koefficient. Det skulle alltså behövas nio bastexturer, en för varje koefficient. Dessa lagrades i grafikminnet direkt efter sin generering.

På Xbox360 och i DirectX10 finns nya möjligheter att bland annat läsa från det bildelement man just skall skriva [8].

För den löpande summan av texturelementen behövde vi en variabel som kunde skrivas från en pixelshader och behöll sitt värde mellan olika anrop till denna. Det visade sig att det inte fanns möjlighet till detta. Det gick inte heller att skicka variabler mellan enskilda pixlar på grund av dess parallellitet.

Problemet löstes istället genom att bara multiplicera texturerna på GPUn, varpå de resulterande texturerna lästes tillbaka till internminnet för att summeras av processorn. På det sättet kunde vi i alla fall dra nytta av parallelliteten hos GPUn vid multiplikationerna.

8.3.1.2 Utvärdering

Resultatet blev att denna metod tog 10 gånger längre tid än DirectX-funktionen. En teori var att det kunde vara de nio förgenererade kubtexturerna som gjorde det långsamt, eftersom shadern var tvungen att växla mellan dessa. En omskrivning av algoritmen att packa baserna i olika färgkanaler gav dock ingen prestandaförbättring.

En bättre förklaring är att det beror på mängden data som skickas från grafikkortet till CPUn. Om den kubiska texturen har sidan s så blir mängden texturelement $6s^2 \cdot 9$ efter multiplikationen. Att så mycket data skickades mellan grafikkortet och processorn gör att prestandavinsten med multiplikationen på GPUn äts upp. Ett försök att snabba upp summeringen genom att använda D3DXFilterTexture (en funktion för att göra gradvisa nedskalningar av texturer) gjordes också, men utan märkbar skillnad i tid, vilket ledde oss till slutsatsen att funktionen troligen exekveras på CPU.

8.3.1.2.1 Negativt

Algoritmen är alltför långsam för att det skall ge någon fördel att använda den i stället för DirectX inbyggda.

8.3.1.2.2 Positivt

Metoden gav lika bra värden som DirectX, med mycket små felmarginaler, mindre än ±0.5 % skillnad. Det fungerar dessutom att dela upp arbetet i små delar för att sprida arbetsbördan.

8.3.1.2.3 Önskade förbättringar

Att kopiera mycket data mellan grafik- och systemminne verkade vara en flaskhals som måste kringgås. Det var vid detta tillfälle inte känt hur DirectX gör sina beräkningar, på CPU eller GPU. Utrymmet för förbättringar var alltså okänt.

8.3.2 Försök 2, Beräkningar i CPU

För att minska mängden data som skickas mellan grafik- och systemminne så flyttades algoritmen till CPU.

8.3.2.1 Design

Kubtexturen som beskriver omgivningen fördes nu över till CPU direkt efter den blivit renderad. Detta innebar visserligen att den tidigare nämnda flaskhalsen inte helt skulle försvinna, men mängden data som skickas minskade med en faktor nio. Nu låg dock hela beräkningsbördan på CPU, vilket vi egentligen ville undvika. Som en anpassning flyttades baserna till systemminnet vid starten av programmet.

8.3.2.2 Utvärdering

Det visade sig att minnesbussen mycket riktigt var flaskhalsen. DirectX version tar visserligen fortfarande mindre tid, speciellt när samplingarna har hög upplösning, men då kubtexturens sidor är mindre än 16 pixlar var det fortfarande mindre än en faktor 2 som skiljde. De värden som funktionen producerade var fortfarande lika bra som DirectX.

8.3.2.2.1 Negativt

Eftersom beräkningarna nu skedde helt på CPU så togs prestanda från andra viktiga funktioner i applikationen. Funktionen är fortfarande inte lika snabb som D3DXSHProjectCubeMap().

8.3.2.2.2 Positivt

Algoritmen var nu tillräckligt snabb för att kunna användas praktiskt. Det fungerade fortfarande att dela upp arbetet för en jämnare belastning på systemet.

Det blev till slut denna version som användes i applikationen.

8.4 Datastruktur och algoritm för sökning i denna

I ATIs artikel "Irradiance Volumes for Games" finns idéer för hur man ökar upplösningen adaptivt i en statisk ambiensvolym [23].

Arbetet med att skapa en datastruktur att förvara koefficienterna i började tidigt. Med inspiration från ATIs demo "Irradiance Volume Sample", som följde med DirectX SDK i augusti 2005 [23], utgick vi ifrån ett octree. Det skulle ge oss möjlighet att adaptiskt höja upplösningen nära kameran och nära marken utan att påverka upplösningen på övriga ställen.

8.4.1 Första försöket; OctTree

Utan kunskap om hur trädet skulle traverseras i slutändan, eller hur bra algoritmen som väljer vart nya koefficienter ska genereras är, så skapades en första datastruktur för att kunna utvärdera vad som egentligen spelar roll.

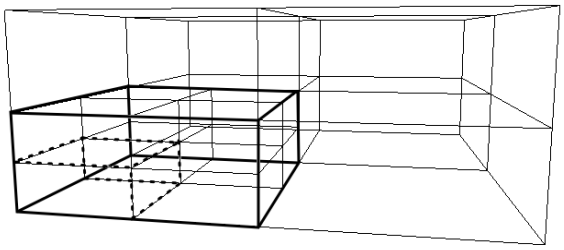
8.4.1.1 Design

Med en bottom-up ansats börjas arbetet på ett åttagrenat träd. Detta innebär att delkomponenter sätts samman när man vet att de fungerar enskilt. Trädet består av en mängd hierarkiskt ordnade noder.

Varje nod är ett rätblock med åtta barn, vilka delar upp nodens förälders volym i lika stora delar. I varje nods hörn finns det möjlighet att ha en uppsättning koefficienter. Ett hörn som delas av flera noder delar också på samma koefficientuppsättning, så att det inte innebär dubbelt arbete att generera fram koefficienter för sådana hörn.

För att kunna använda koefficienterna som lagts i hörnen behövde man veta vilket position de hämtats från i spelvärlden. Denna position användes även för att identifiera hörnen, och avgjorde därför om två hörn befann sig på samma position i världen och därför skulle dela information.

Ett åttagrenat träd med data i hörnen. Varje barn delar sin föräldranods volym symmetriskt.



Datastrukturen har ansvaret för att ge algoritmen som väljer vart nya koefficienter ska genereras information om hur bra upplösningen är. Denna information baseras på en sökning från toppnoden mot den nod som är längst ut i hierarkin och utgör den minsta existerande volym som omsluter positionen där man befinner sig. Det första hörn som saknar koefficienter i denna sökning ger den position där sampling ska ske. Är alla hörn genererade så blir lövet en nod med åtta nya löv. Bara lövet man befinner sig i får koefficienter, och detta behöver maximalt 7 nya koefficientuppsättningar då det alltid delar ett hörn med sin redan kompletta förälder.

Med denna strategi blir koefficienterna alltid mer korrekta med tiden, eftersom de hamnar närmare den position där man befinner

sig. En högre upplösning nära kameran erhålls vilket gör att ambiensen blir mest korrekt där det syns bäst. Om man förflyttar sig långsamt hinner man generera koefficienter med hög upplösning, men om man förflyttar sig snabbt så hinner trädet inte generera noder lika djupt, vilket ger mindre korrekta koefficienter när man plockar ut dem från datastrukturen.

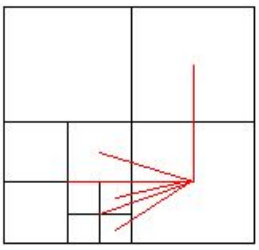
Att hämta ut information från trädet, d.v.s. att hitta koefficienterna för en viss position, sker genom att hitta ett löv för positionen och använda de närmsta koefficienterna i varje oktant runt denna position. Avståndet till dessa används för att linjärt interpolera fram vilka koefficienter som gäller i den aktuella punkten. Om det inte finns minst ett hörn åt varje riktning på sidorna av rätblocket för den yttersta noden så används förälders hörn istället, då den är garanterad att ha koefficienter åt varje riktning. Detta är sant så länge man befinner sig inom rotnodens volym och denna har samtliga åtta koefficienter. För att enkelt kunna lista ut vilka hörn som finns på väggarna i det aktuella rätblocket har alla noder en lista på de noder som delar en sida med den aktuella noden.

Givet en dynamisk värld med dag och natt, och där solen rör sig över himmeln, så kommer koefficienterna med tiden att förlora sin giltighet. Detta ställer krav på uppdatering med viss frekvens. I vårt fall handlar denna tidsperiod om cirka en minut, varefter solen har flyttat sig, eller ett moln kan ha glidit in och förmörkat den. Med en gissning på 30 bilduppdateringar per sekund (eng. fps) och en uppsättning koefficienter för ett hörn var 6:e bilduppdatering så klarar man 30 fps * 60 sekunder / 6 bilduppdateringar, totalt cirka 300 koefficienter per minut. Detta ger en övre gräns på antalet simultana hörn i trädet. För att hålla hörnen i trädet under detta antal så bör en algoritm skapas som har som mål att vid behov ta bort hörn som befinner sig långt bort från den nod som kameran är i, alternativt peka ut de noder som behöver genereras om av noderna som redan existerar.

8.4.1.2 Utvärdering

För att snabbt kunna hitta fel och utvärdera trädets beteende gjordes implementationen av detta OctTree i den egenutvecklade demoapplikationen. På grund av den stora mängden problem som påträffades så övergav vi detta träd innan det fick möjlighet att anpassas och testas i en riktig spelmotor.

8.4.1.2.1 Positiva resultat



Figur: Varje nod känner till de noder som den delar en yta med, det gör det lätt att hitta hörn på ytan när man ska interpolera.

Vid uthämtning från datastrukturen gick det snabbt att hitta vilket löv en position befann sig i, och eftersom alla löv hade en lista på sina grannar så fann man enkelt samtliga hörn på ytan av rätblocket. Att interpolera fram en koefficient från strukturen orsakade heller inga bekymmer p.g.a. att föräldern till det yttersta lövet alltid var garanterad att ha existerande koefficienter.

8.4.1.2.2 Negativa resultat

Ordningen som nya koefficienter genererades fram med var inte bra. Att börja generera koefficienter för noder med stor volym fick som konsekvens att det tog lång tid innan man nådde en bra upplösning även vid låga hastigheter, något som var speciellt uppenbart då man passerade över en yta som delade på två noder nära roten av trädet. I ett sådant fall var man tvungen att börja generera en stor mängd koefficienter som låg långt borta, och för varje löv som man delade i åtta så ökade risken att man skulle befinna sig nära ett nyligen genererat hörn. Detta kunde ge upphov till en hastig färgförändring, något man vill undvika.

Målet borde istället ha varit att hålla upplösningen hög nära kameran i alla lägen och åt alla håll, alltså vänta med att öka upplösningen längre bort. Först när alla riktningar som kameran kan röra sig åt har en bra täckning kan man generera koefficienter en bit bort. Detta förhindrar tydliga färgförändringar nära kameran.

Att sätta en övre gräns på antalet koefficienter i trädet visade sig också vara en dålig idé, då denna datastruktur ska fungera både då man har en fps över 100 och under 10. Vid tester som involverade förflyttning upptäcktes tillfällen då koefficienter som var precis intill kameran togs bort från en nod som låg spatialt nära men långt bort i trädet (t.ex. med roten som enda gemensamma förälder) bara för att en sekund senare genereras fram av den nod man kommit till.

Anledningen till att problemet uppkom var att algoritmen gallrade bort fel koefficienter, detta berodde i sin tur på kravet att koefficienter nära roten var viktigast att behålla och trädets övre gräns för antalet koefficienter var för lågt satt. Ett bättre alternativ vore att ta bort gamla koefficienter med hjälp av en tidsgräns, så att antalet befintliga hörn hela tiden var så många som maskinen klarar av. På så vis slipper man även problemet att gallra bort ”rätt” koefficienter ur trädet.

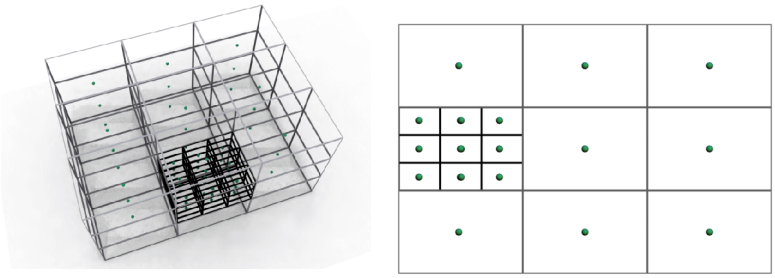
Att låta hörnens positioner anges av flyttal var ett implementationsmisstag. Följden blev att det kunde hända att två hörn som fanns på samma ställe fick en aning olika positioner. Detta orsakades av avrundningfel och resulterade i att alla hörn som borde delas mellan noder inte längre delades. Detta problem upptäcktes sent.

8.4.2 Andra försöket; SHGraph

Med lite bättre kunskaper om vad som inte fungerar, och vad som är viktigt slängdes den gamla implementationen och detta nya träd skapades. Nu var fokus på att lätt kunna traversera mellan spatialt näraliggande noder samt att ha hög upplösning kring kameran före noder långt borta får genererad information.

8.4.2.1 Design

Denna gång breddas trädet med en nod per dimension, vilket ger ett grid med totalt 27 barn för varje förälder. Tanken är att det är viktigare att ha ett brett högupplöst område runt kameran och att noder nära roten inte behöver lika hög täthet på koefficienter. Fokus ligger därför inte längre på att öka upplösningen så mycket som möjligt, nu ska en tillräckligt hög upplösning erhållas omedelbart, och först därefter kan noderna långt bort få tid att uppdateras.



Figur: Varje nivå består av 3x3x3 lådor, totalt 27 st. Informationen i mittennoden av ett grid är den samma som informationen i gridets förälder.

Koefficienterna placeras nu i centrum av varje nod, främst för att en förgrening bara ger 26 nya koefficienter, istället för 56 som koefficienter i hörnen skulle ge. Man kan dessutom prioritera vilka noder som ska genereras på fler sätt, t.ex. noden i mitten och de 8 hörnnoderna. Att ha koefficienter i centrum eliminerar också problemet med att noder delar på hörn, det är bara mittenbarnet som delar hörn med sin förälder och dessa har exakt samma position vilket gör att beräkningsfel inte uppkommer här. De 26 nya koefficienterna kan jämföras med 19 nya koefficienter i OctTree trädet per förgrening.

Detta träd har ett maximalt djup, vilket tillsammans med storleken på rotnoden ger en maximal spatial upplösning. Det maximala djupet på trädet bestäms genom att jämföra tiden det tar för datorn att generera fram koefficienter med vilken hastighet man normalt förflyttar kameran. Man bör också skapa en balans mellan maximal upplösning och i vilken radie man vill bibehålla denna. När kamerans hastighet ökar så minskas upplösningen, men den är fortfarande så hög som hastigheten tillåter nära kameran. Ökas hastigheten tar man sig bara en nivå högre i trädet och genererar nya koefficienter där.

En viktig nyhet från det åttagrenade trädet är att det är större fokus på att gå mellan noder som är grannar. Möjligheten att kunna traversera direkt mellan grannar utan att gå via föräldrar är anledningen till att strukturen döpts till graph.

Eftersom funktionen som begränsade antalet hörn det åttagrenade trädet fungerade dåligt ges nu istället koefficienterna en livstid. De har även möjlighet att begära att genereras om innan denna livstid tar slut. Detta för att förhindra att upplösningen sänks då koefficienter nära kameran försvinner. För att inte grenar som förlorat alla sina koefficienter ska ligga kvar så får de hålla reda på om de har några barn med koefficienter. Om inga koefficienter finns kvar så tas noderna bort.

Trädet får inte heller ha begränsningar i volym. Dock bör det finnas en normal storlek vilken trädet ska sträva efter att ha, så att det återgår till denna storlek när man förflyttat sig in i en ny nod. En tanke på att flyttalsprecisionen minskar när man går mot större volymer bör också finnas.

8.4.2.2 Utvärdering

Den första implementationen av SHGraph gjordes i den egenutvecklade demoapplikationen. Efter en mycket lovande första test så flyttades koden in i Avalanche Engine. Detta gav nya perspektiv, genereringen av koefficienter var nu bara en liten del av alla processorkrävande komponenter. Nya begränsningar och krav dök upp när bara en liten del av världen kunde vara laddad vid ett och samma tillfälle. Det blev även tydligt hur stora besparingar det fanns att göra när man hade en spelmotor med specifika egenskaper. I många situationer kunde tämligen exakta rörelseprognoser skapas, vilket både kunde begränsa antalet dimensioner och skapa enkla modeller för att ta fram användbara koefficienter.

8.4.2.2.1 Positiva resultat

Strategin att prioritera det högupplösta området kring de belysta modellerna gav en stor kvalitetsförbättring vid förflyttning, redan i demoapplikationen. Då man i Avalanche Engine dessutom kunde förutspå vart förflyttningarna sannolikt skulle leda, samt helt utesluta områden som inte någonsin skulle synas så förblev det högupplöst runt modeller som flyttades runt. Upplösningen var proportionell mot hastigheten.

8.4.2.2.2 Negativa resultat

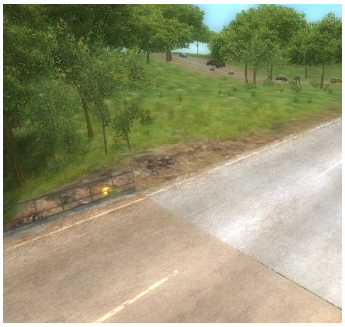
Områden nära objekt gav fortfarande tydliga problem. Att man nästan alltid befinner sig nära marken i Avalanche Engine kunde avhjälpas något genom att använda koefficienter som genererats en bit ovanför marken, men om man rörde sig nära branta sluttningar eller i täta gränder kunde snabba färgskiftningar fortfarande ske, och tydligt felaktiga färger upptäckas. Dessa problem går att lösa genom att ta hand om en stor mängd specialfall, men för att man enkelt ska kunna använda algoritmen i valfri motor krävs en generell lösning.

8.5 Interpolera fram ambiens

När ett objekt skulle belysas plockades koefficienterna som fanns närmst objektets position ut från datastrukturen. I den slutgiltiga versionen förenklades genereringen av ambiens till markplanet vilket gjorde att man bara behövde leta upp en koefficientuppsättning i varje kvadrant kring varje objekt. Dessa linjärinterpolerades sedan med avseende på sitt avstånd till objektet.

Till en början skapade denna strategi många hastiga färgförändringar eftersom objekten inte behöll sina färger eller interpolerade fram nya färger över tid. Stora färgförändringar inträffade oftast när koefficienter dök upp eller försvann, vilket gjorde att nya färger interpolerades in i datastrukturen istället för att bara läggas till direkt.

8.6 Applicera ambiensen på objekt i scenen



Bilden ovan illustrerar problemet som uppstår då alla objekt inte använder samma ambiens.

För att färgsätta objekt i scenen med de interpolerade koefficienterna behöver man evaluera spherical harmonics för varje punkt man vill färga, om man inte väljer att interpolera redan framräknade färger.

Utifrån en ytnormal och en uppsättning koefficienter så är färgvärdet enkelt att beräkna, se analysdelen. Beräkningsmässigt behövs för varje färgkanal en skalärprodukt samt en vektor-matrisoperation.

Det är viktigt att alla objekt i en scen använder samma ambienta ljus. Vår modell medför därför att det är ett krav att alla objekt har färdiga normaler. Detta gör t.ex. att normaltexturer borde användas till billboards om sådana finns.

För att skriva kod till GPUn använde vi HLSL och CG, två språk som är väldigt lika varandra. För att lära sig grunderna i shader programmering se “HLSL Workshop” [18].

Att endast vektoroperationer med fyra element per vektor ingår betyder att evalueringen går att utföra på GPU på ett effektivt sätt. Det man behöver göra är att sätta in koefficienterna i en matris, och multiplicera dem med konstanter för omvandlingen till inträlningslösningen. Insättning i matris och multiplikation med koefficienter sker i samband med interpoleringen, alltså i CPU. Denna matris sätts som pixelshaderkonstanter för objektet som skall färgsättas. Ytans normal räknas ut via normaltextureringen i pixelshadern.

Koordinatsystemet som koefficienterna är uträknade i, är det samma som scenens världskoordinater. Basen för normaltextureringen är dock tangentrymden, som för varje vertex spänns upp av vektorerna mot de två andra i samma triangel.

Spherical harmonics är komplicerade att rotera, se “Foundations of precomputed radiance transfer” [13].

I den ursprungliga pixelshadern så används normaltexturen endast ihop med en ljuskälla, solen. Man har då gjort optimeringen att transformera denna ljusvektor till tangentrymden, linjärinterpolera den transformerade vektorn över triangeln och direkt räkna ut ljussättningen. Tyvärr är spherical harmonics besvärliga att rotera på detta sätt, det är enklare att transformera ytans normal till världskoordinater. För att kunna ta fram världsnormalen för varje pixel på ytan behövs matrisen för att transformera denna från tangentbasen.

Basen för tangentrymden skickas då från vertex- till pixelshader, och linjärinterpoleras där. Tangentbasen används för att transformera normaltexturen till samma bas som spherical harmonic-koefficienterna är definierade i. Färgen kan nu räknas ut på vanligt sätt.

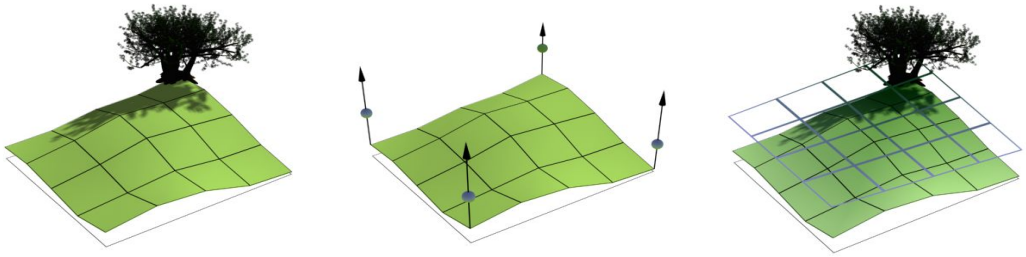
Se referens om ett unifierat ljussättningssystem från Gamasutra [12].

En bieffekt av att ta fram världsnormalen för ytorna på ett objekt är att det blir enklare och mer elegant än tidigare att lägga till fler ljuskällor och effekter, till exempel specular environment mapping.

En del typer av objekt kräver specialfall av evaluering, något som tas upp i diskussionsdelen. Den enda typen av sådan anpassning som är implementerad är för att ljussätta terräng. Terrängen består av ett i höjdled förskjutet 2D-plan. Detta plan är uppdelat i mindre fyrkanter (patches) som ligger intill varandra. Problemet med att ljussätta dessa ligger i att de skarvar som finns mellan dem blir synliga om olika ambiens används för två intilliggande patches.

En lösning skulle kunna vara att för varje hörn i terrängfyrkanten interpolera fram en ambiens, och sedan i sin tur linjärinterpolera över terrängfyrkanten. Tyvärr innebär det en mycket stor beräkningsbörda att för varje vertex i terrängen interpolera fram en hel koefficientuppsättning.

Vad som i stället implementerades var en approximering där en färg evaluerades för varje hörn i en patch. Denna färg linjärinterpolerades sedan över ytan på patchen.



1. En terrängpatch med en bit skog i ena hörnet. Observera att detta endast är en konceptskiss, det krävs många träd för att påverka ambiensen.
2. Koefficienter interpoleras fram för de 4 hörnen på terrängpatchen. Vegetationen är avlägsnad för tydlighetens skull, men påverkar ändå ambiensen. Lägg märke till att den bortre ambienskulan är grön på ovansidan. En färg evalueras för varje hörn.
3. Färgen från föregående steg linjärinterpoleras över terrängen. Marken under skogen (trädet) drar mot mörkgrönt, medan den öppna terrängen antar en blå ton från himmeln.

9 Algoritmbeskrivning

Här nedan redovisas de resultat som framkommit i form av algoritmer, körtider och minnesåtgång för några av delarna.

Visuella resultat redovisas för sig, se eget avsnitt. Hur stor del av den ursprungliga specifikationen som uppfylldes avhandlas i diskussionsdelen.

9.1 Sampling av omgivningen

För att generera samplingar av omgivningen renderas en kubtextur. För att göra detta sker följande:

1. Gå till nuvarande samplingsposition. Om ingen sådan finns eller tidigare sampling var klar förra uppdateringen så hitta en position att sampla.
2. Flytta kameran till positionen som skall samplas. Sätt dess rotation så att en ny sida av kuben genereras, samt ändra projektionen till $\pi/2$ radianer både horisontellt och vertikalt.
3. Uppdatera alla objekt som gallrats bort vid den normala uppritningen. Detta inkluderar bland annat statiska objekt, terräng, växter och moln.
4. Rendera den aktuella sidan av kubtexturen.
5. Återställ kameran till det tillstånd som den hade i steg 1.

En sida av kubtexturen genereras varje bilduppdatering.

Endast statiska objekt såsom träd, terräng och hus är med i renderingen. Detta är för att förlänga giltighetstiden för koefficienterna. Problem finns fortfarande med stadsområden, där den extra uppdateringen som sker i steg 3 inte förmår majoriteten av hus eller vägar att ritas.

9.2 Projiciering till koefficienter

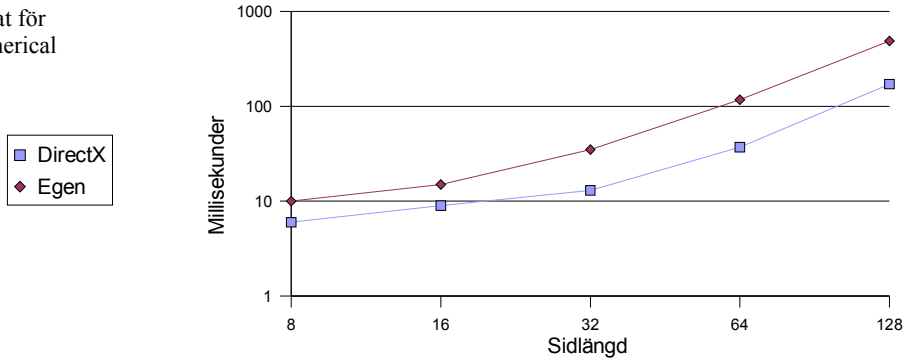
Då en kubtextur är genererad kallas funktionen för att projiciera denna på spherical harmonicform.

Algoritmen för att ta fram koefficienterna ser ut som följer:

1. Förgenerera 9 kubtexturer som projicierar en kub till en sfär, samt viktar mot en SH bas. Dessa kräver bara en kanal vardera så de packas i 3 texturer för att spara minne.
2. För över den färdiggenererade kubtexturen av omgivningen från grafikminne till systemminne.
3. Multiplicera ihop de förgenererade texturerna med omgivningens textur, summera dessa och dela med antalet pixlar för varje färg, samt med $\pi/4$ för att normalisera sfärens yta.

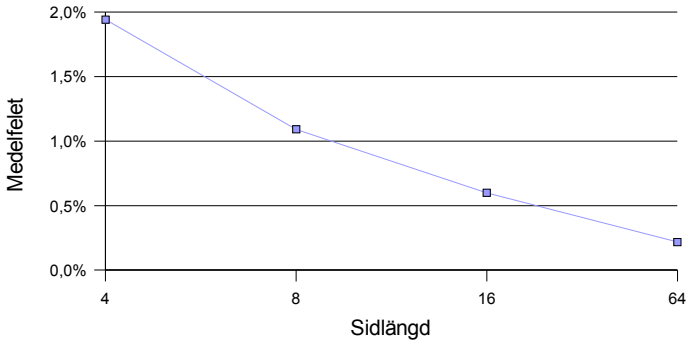
I diagrammet nedan visas prestandaresultat för vår egenutvecklade algoritm, så väl som för den som finns inbyggd i DirectX. Tider för ett antal olika storlekar för den ursprungliga kubtexturen visas för de båda algoritmerna.

Diagram 1: Resultat för projiciering till spherical harmonics.



Resultaten från de båda algoritmerna var så gott som identiska, den genomsnittliga skillnaden på felet var mindre än 0,1 %. Det kvadratiska medelvärdet (RMS) på skillnaden mellan resultatet från sampling där kubtexturen hade upplösningen 256 x 256 på varje sida, jämfört med en mindre sida visas i diagrammet nedan.

Diagram 2: Färgavvikelse i SH koefficienter som funktion av kubtexturens sidlängd, med en textur med sidan 256 som utgångpunkt.



Vi valde att använda en sidlängd på 16 pixlar för att hålla tidsåtgången låg samtidigt som felet blev acceptabelt litet.

9.3 Datastrukturen och interpolering

För att plocka ut en uppsättning koefficienter för en position i världen användes följande algoritm:

1. Hitta den närmaste koefficientuppsättning längs markplanet i varje kvadrant runt objektet. Detta ger mellan en och fyra koefficientuppsättningar.
2. Om fler än en uppsättning hittats så linjärinterpoleras koefficienterna per bas, med avseende på avståndet.
3. Om bara en uppsättning hittats så används denna.

9.4 Evaluering av Spherical Harmonics

För att färgsätta objekt med en koefficientuppsättning används följande algoritm. Här beskriven för ett objekt som använder normaltexturering.

1. Från applikationen skickas den interpolerade koefficientuppsättningen som konstanter till grafikkortets pixelshader. Detta innebär $9 \cdot 3$ skalärer, men för att spara shaderinstruktioner skickas istället färdiga matriser, vilket blir 12 vektorer med 4 skalärer i varje.
2. I vertexshader sker transformer som normalt. Dessutom skickas basen för tangentrymden till pixelshadern.
3. I pixelshader räknas världsnormalen fram för varje pixel från normaltexturen och tangentbasen. Detta kräver en matrismultiplikation.
4. Koefficienterna evalueras med hjälp av världsnormalen. Operationen är en matrismultiplikation samt en skalärprodukt för varje färgkanal.

10 Diskussion

Här nedan diskuteras hur väl det slutgiltiga resultatet överensstämmer med det ursprungliga projektförslaget. Vilka delar av projektet som lyckades väl, och vilka som skulle behöva omvärderas. Det finns även andra tekniker för att behandla ambient ljus som kan vara värda att titta på, både mer och mindre exakta än den som presenterats här. Detta kapitel inleds med en jämförelse av det slutgiltiga resultatet av arbetet med dessa tekniker.

10.1 Jämförelse mellan befintliga tekniker

Vår modell är inte den enda approximationen av ambient ljus, och inte nödvändigtvis den som är mest lämplig för alla typer av applikationer. Här jämförs några av de tekniker som används idag.

10.1.1 Klassiska modellen med konstant ambiens

I den klassiska datorgrafiken använder man vanligen grå, eller möjligen svagt tonad ambiens. Man är också noga med att välja färger på texturer samt styrkor på ljuskällor på ett sådant sätt att det inte uppstår överslag eller svarta områden.

För information om ljusförhållanden i olika väderförhållanden så se [20].

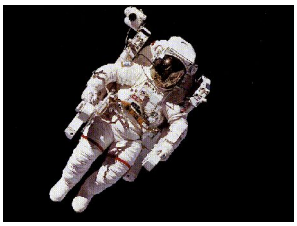
Man kan se det som att man bygger upp en värld där ögats anpassning till ljusförhållanden redan är inräknad i varje punkt. I stället för att sänka ljusnivån med sju tiopotenser då solen går ner sänker man bara ambiensen några procent, och tonar den aningen blå, för att simulera ögats ökade känslighet för höga frekvenser i sitt mörkerseende. Denna typ av ambient approximation används fortfarande flitigt, bland annat i Avalanche Engine.

Hur ögat anpassar sig till olika ljusförhållanden kan man läsa mer om i [9].

Det finns ett antal tillfällen då den klassiska modellen brister. Framför allt är det vid tillfällen då skarpa kontraster finns, vid växlingar mellan inom- och utomhusmiljöer, både på dagen och pånatten. Detta gäller både ljus- och färganpassningen. I miljöer där både ljus- och färgförhållanden är homogena är den klassiska modellen dock en bra approximation för hur ögat uppfattar omgivningen.

Ett exempel på en modell som visar hur ögat anpassar sig till varierande ljusförhållanden finns i arbetet “Time-Dependent Visual Adaptation” [24].

I rymden och på havet är konstant ambiens en bra approximation.

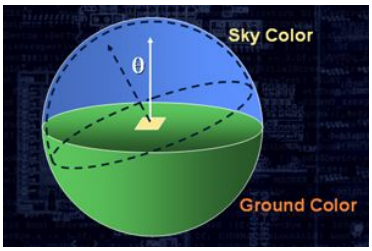


10.1.2 Enkel hemisfärisk ambiens

Philip Taylor skriver mer om hemisfärisk belysning [27].

Denna modell är tänkt för utomhusmiljöer, och tar hänsyn till att himmeln och marken dominerar. Den använder en färg för övre hemisfären och en färg för den undre. Dessa interpoleras med avseende på normalens lutning gentemot horisonten.

Hemisfärisk ambiens passar bra i flygsimulatorer.

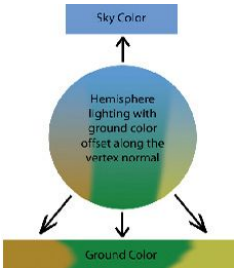


10.1.3 Hemisfärisk ambiens med radiositetstextur

Hargreaves modell är publicerad i boken ShaderX2 samt på Gamasutra [7].

En utvidgning av hemisfärisk ambiens som Shawn Hargreaves beskriver, använder sig av en textur som lagrar markens färg. Modellen användes i spelet MotoGP 2.

Racingspel kan förbättras med en högupplöst textur för marken. Bilderna är tagna från den ovan nämnda artikeln om MotoGP 2.



10.1.4 Ambiensvolym

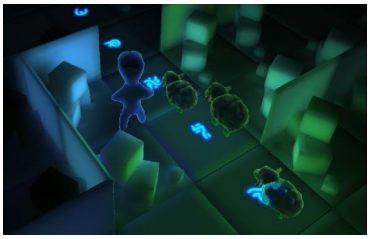
Modellen som beskrivs i detta arbete. Mest användbar i miljöer där byggnader, träd, grottor och andra objekt samverkar till en ambiens som skiftar åt alla riktningar. Det finns även versioner av denna modell som bygger på en förberäknad ambiens. Gemensamt för denna och alla tidigare nämnda modeller är att ambient occlusion inte tas hänsyn till.

För en diskussion om ambient occlusion så se nästa kapitel under rubriken “Sampling”.

Spelet Half-Life använder en förberäknad ambiensvolym för att belysa världen [16]. Modellen lämpar sig bra för scener som både har inom- och utomhusmiljöer.

10.1.5 PRT

Precomputed radiance transfer betyder att man för varje punkt på ytan av ett objekt har lagrat en överföringsfunktion mellan instrålat ljus och utstrålat. En sådan funktion beror på hela objektet den beräknas på. Till exempel tas både genomlysning och diffusa reflektioner med i beräkningen.



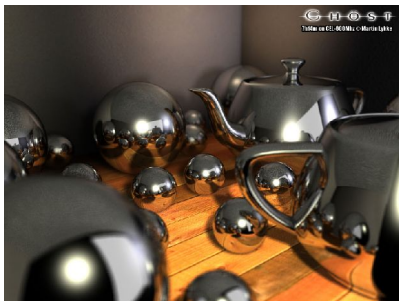
Shepherd of Dreams [26] är ett litet spel som utnyttjar PRT. Lägg märke till hur ljuset från karaktären påverkar de omgivande strukturerna.

Den här modellen tar hänsyn till ambient occlusion. Den är dock så minneskrävande och beräkningsmässigt tung att den måste förberäknas, och ändå bara kan användas för statiska objekt. Man gör dessutom även här antagandet att ljuskällor befinner sig oändligt långt bort.

10.1.6 Global illumination

I Global Illumination (GI) beräknar man ljusflödet på ett så fysiskt korrekt sätt som möjligt från ljuskällorna fram till mottagarna via interaktioner med olika material. Det finns flera metoder men alla är för långsamma för att kunna användas i realtid. Vanligast är att de är baserade på raytracing.

Global Illumination inkluderar alla belysningseffekter och strävar efter att vara fysikaliskt korrekt istället för att bygga på en approximering.



Bilden till höger tog drygt 7 timmar att rendera och kan hittas på http://www.wilddemo.dk/3dpics/toomuchchrome_info.htm

10.2 Diskussion om modellen

Vår modell har ett antal visuella tillkortakommanden som är värda att nämna. Dessa brister finns det metoder för att åtgärda, men vi har inte haft tid att utvärdera dem. Istället passar vi på att diskutera dem här.

10.2.1 Ljussättning

Modellen som används för att ljussätta objekt utgår från att det går att välja en punkt inom ett objekt sådan att det ambienta ljuset i den punkten är giltig för hela objektet. Detta håller någorlunda väl för objekt som är av samma storleksordning som, eller mindre än upplösningen hos koefficientdatastrukturen.

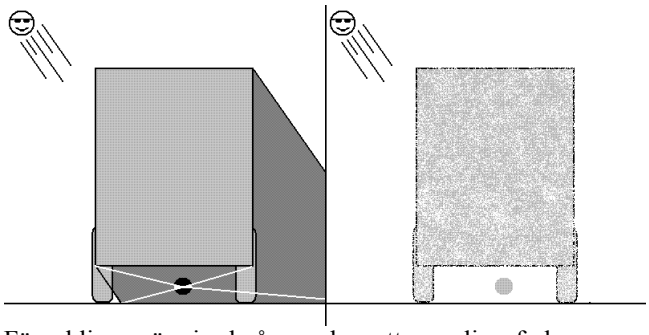
Då objekten i sig utgör en större volym än det mätbara ljuset mellan dem kommer det att uppstå fel. Det värsta exemplet är i stadsmiljö, där två stora och ganska låga hus står tätt intill varandra. Betrakta en vägg som vetter mot det andra huset. Denna vägg kommer att hämta sin ambiens från mitten av det låga huset. Denna position kommer att vara halva husets bredd genom två fel, vilket i fallet ett stort hus, kan bli ett avsevärt avstånd.

En mer korrekt modell hade delat upp huset i mindre bitar, och ljussatt dessa individuellt. Detta är i princip hur terrängen ljussätts, även om denna är betydligt lättare att dela upp i block.

10.2.2 Sampling

Ett problem som vår modell inte klarar är att ge korrekt ambiens i skuggor från dynamiska objekt, t.ex. under en lastbil.

I utrymmen utan insyn från omgivningen bildas mörka områden (Ambient occlusion). Detta är ett helt forskningsområde i sig.



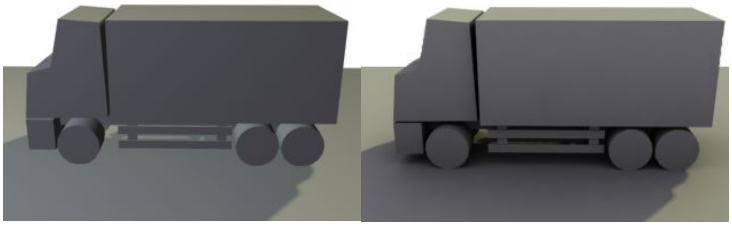
Förenklingen är gjord på grund av att samplingsfrekvensen tidsmässigt är så låg att dynamiska objekt inte tillåts vara med när man samplar ambiensen. Om dynamiska objekt inte finns med blir ljusinformationen något felaktig. Om man t.ex. samplar under en lastbil täcks nästan hela övre hemisfären av lastbilens underrede, som ligger i skugga. Nästan hela undre hemisfären består av mark som skuggas av lastbilen. Resultatet blir ett mörkt område, till skillnad från om man samplar utan lastbilen och dess skugga. Felet uppstår i områden där sekundär belysning har svårt att nå.

Att den ambienta informationen är korrekt i skuggade områden är viktigt, eftersom ytor där bara är belysta av ambient ljus.

Fenomenet där skuggade områden sänker ambiensens ljusstyrka kallas ambient occlusion. På senare tid har det gjorts ett flertal undersökningar om hur man ska lösa det i realtid.

TV: Ambient occlusion saknas i modellen, vilket ger en för ljus ton på instängda ställen.

TH: Referensbild med ambient occlusion. Mindre av omgivningens ljus förmår träffa marken under bilen.



Vanligtvis brukar man för statiska objekt rita in ambient occlusion i objektets textur, så att det ser ut att sitta ihop bättre. Till exempel skulle man ha målat insidan på en lastbils hjulhus i en mörkare färg.

Ett exempel på dynamisk ambient occlusion kan hittas i kapitel 14 i boken *GPU Gems 2* [2], där Michael Bunnell föreslår en lösning som approximerar den inom ett objekt. Objektet analyseras i realtid och mörkare ambiens används där objektet skymmer sig självt. Denna lösning klarar t.ex. dynamisk morfning mellan animationer och kan även användas för att beräkna färgen på det indirekta ljuset med hjälp av s.k. bent normal.

En annan lösning som fokuserar på ambient occlusion mellan flera olika statiska objekt heter *Ambient Occlusion Fields* [11] och är gjord av Janne Kontkanen och Samuli Laine. Här krävs offlinegenererad data för varje objekt, sedan påverkas ambiensen när objekt kommer nära varandra. En liknande lösning har även gjorts av Assarsson m.fl. vid INRIA [14].

En mycket enkel approximation för ambient occlusion mellan föremål och mark är att använda en mörk mjuk fluff under föremålet som beror på avståndet mellan föremålet och marken (blob shadows).

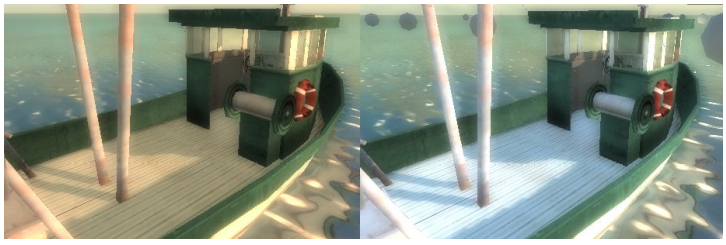
10.2.3 Ljusstyrka

För mer information om luminans från olika ljuskällor, se “Lighting Design Glossary” [20].

Det faktum att solen vid middagstid är 10^5 gånger starkare än himmeln gör att det ambienta ljuset som himmeln ger drunknar och försvinner nästan helt på en solig dag. Motsvarande gäller även för solljus reflekterat i vatten (en ambient effekt), det bör ge en så stor inverkan att vattnets i övrigt blå färger inte utgör något större tillskott. Dessa exponentiella förhållanden är svåra att få korrekta vid rendering.

TV: Grå konstant ambiens som ger båten korrekt färg i solen, men fel färg i skuggan.

TH: Beräknad ambiens som ger korrekt färg i skuggan, men blir för blå i solen. Detta beror på att exponentiella ljusförhållanden inte används. Den blå ambiensen dränks då inte av det starka gula solljuset.



HDR är en förkortning av High Dynamic Range.

Fotografier med korrekta exponentiella förhållanden kan hittas på Paul Debevecs hemsida [3]. Där finns också annan intressant forskning kring HDR.

Problemet ligger i att resultatet av en renderad scen är en 2D textur, med RGB färgvärden mellan 0 och 1. På senare tid har det blivit möjligt att rendera bilder med en teknik som kallas HDR. Här renderar man till en flyttalstextur och precis innan scenen visas på skärmen så filtreras den till för att visas med färgvärden mellan 0 och 1. En flyttalstextur använder flyttal för att representera färgvärden, vilket ger möjlighet att lagra både väldigt stora tal och mycket små tal, genom att använda flyttalens mantissa och exponent.

För en bra förklaring av termerna ljusstyrka och luminans så se Elfas faktasidor [4].

Om man använder rendering till flyttal och har belysning med fysikaliskt korrekt storleksordning så stöter man på problemet att vissa saker kommer att vara flera storleksordningar ljusare än andra. Detta går inte att återge på dagens skärmar på grund av att kontrasten är för låg. Idag har en typisk LCD skärm kontrasten 500:1, och en luminans kring 500 cd/m^2. Detta att jämföra med himmeln som på dagen har en luminans kring 8000 cd/m^2. Solen har på dagen en luminans runt $1.6 \cdot 10^9$ cd/m^2. Man måste därför simulera ögats känslighet och anpassningsförmåga så att man får en bild som upplevs som korrekt. Detta innebär att man i en renderad bild väljer ett intervall som skall vara synligt baserat på hur ljus bilden är. Man vill undvika stora områden där skärmen inte förmår återge det sanna ljusförhållandet och därför blir helt vit eller helt svart.

Samma bild ser olika ut vid olika ljuskänslighet.

För att läsa mer om hur ögat justerar sig genom kromatisk adaption och vitbalans se "Lighting Design Glossary" [19] och [21].

I fallet med ambient färgsättning får man också ta hänsyn till att ögats färgkänslighet strävar mot att normalisera sig till den genomsnittliga omgivningsfärgen (kromatisk adaption). Då adaptionen är fullständig kommer bildens färg att upplevas som neutral. Endast genom att beräkna hur olika färgers ljushet upplevs av en betraktare, kan man avgöra vilken färg omgivningsljuset verkligen har.

11 Resultat

De resultat som framkommit under arbetets slutförande sammanfattas här tillsammans med några slutsatser om hur de skulle kunna användas.

11.1 Prestanda

Den implementation vi har skrivit bygger på att man använder samma metod för rendering av omgivningsbilderna för ambiensgenerering och att rendera hela skärmbilden. Även om den bild som används för att beräkna ambiensen är flera storleksordningar mindre än skärmbilden så tar de jämförbar tid att generera. Detta får till följd att uppdateringsfrekvensen för skärmbilden halveras.

Den långsamma prestandan beror till stor del på att bilder tas från olika platser i snabb följd, något som förstör de optimeringar som bygger på att det bara finns en kamera som rör sig relativt långsamt. Culling, avancerade billboardtekniker som t.ex. molnberäkningar och streaming av data från disk, är några av de delar som far illa av ambiensgenereringen.

Datastrukturen, som är den enskilt mest komplicerade delen av arbetet, har visat sig vara snabb att använda. Den är även relativt flexibel vid ändringar av implementationen för att optimera. Minnesmässigt beror den på hur mycket information man väljer att behålla. I vår implementation i Avalanche Engine sparades i snitt runt 1000 koefficientpositioner simultant, vilket gjorde att vi använde knappt 1 MB för datastrukturen när den var fylld.

Evalueringen i shaders innebär en matris-vektormultiplikation och en skalärprodukt, vilket tillsammans blir totalt 5 skalärprodukter, för varje färgkanal. Huruvida detta är för dyrt eller inte avgörs av målplattformens prestanda.

Förutom själva belastningen på shaderinstruktionssidan tillkommer förflyttning av shadervariabler, alltså matriserna som beskriver ambiensen, till grafikkortet. Man måste dessutom skicka fler variabler mellan vertex- och pixelshadern än tidigare vid belysning med normaltextur.

11.2 Utseende

Naturligtvis kan en utseendebedömning inte bli helt objektiv. Det närmaste man kan komma är istället att avgöra om de visuella resultaten verkar rimliga eller inte.

De tester som utförts visar att föremål tar färg av sin omgivning och därför ser ut att smälta in i den. Själva modellen med irradiance istället för konstant ambient term ger en fördel i form av mer enhetlighet i ljuset vid en position till priset av en försämrad kontroll av färgsättningen. För exempel på visuell skillnad mellan vår modell och konstant ambiens, se bilaga C.

I en stor och procedurgenererad spelvärld kan det finnas en fördel i att automatisera skapandet av sekundärt ljus, för att spara tid vid skapandet av miljön. Detaljkontrollen är inte heller lika central i procedurellt genererade områden.

12 Framtida arbete

De främsta förbättringarna finns att göra inom prestandaområdet. För att metoden skall fungera i en spelmiljö utan några större försämringar av uppdateringsfrekvensen så behöver både spelmotorn i sig och algoritmerna i detta arbete utvecklas och anpassas till varandra.

12.1 Optimeringar av projektion mot SH

Mot slutet av projektet hade nya insikter vunnits om hur DirectX API fungerade. Det skulle förmodligen gå att få algoritmen att fungera helt på GPU, genom att skriva egna pixelshaders för att i flera steg summera ner resultatet från multiplikationen genom att mellanlagra i allt mindre render targets. Slutligen skulle man för varje sida i kubtexturen endast ha ett enda texturelement. Dessa skulle sedan kunna summeras ner i en textur om ett texturelement per bas som läses tillbaka till systemminnet.

Ytterligare möjligheter för att förbättra funktionen kommer i DirectX 10 [8] då man skall kunna rendera till en kubtextur i ett pass, samt använda heltalsinstruktioner till shaders som förenklar nedsampling av texturer.

12.2 Ljussättning av stora objekt

Redan i den nuvarande versionen uppstår vissa problem i områden där det är tätt mellan stora objekt, som i städer och byar. För att få en korrektare ljussättning på mycket stora objekt skulle man kunna tänka sig en lösning där man delar upp objektet i mindre delar som ljussätts individuellt. Detta är dock en intensiv operation som också kräver mycket av vertexshadern.

En annan metod för att ljussätta dessa stora objekt skulle kunna innebära att man genererade fram en ny koefficient, baserat på flera andra koefficienter, så att koefficienter i ytterkanten på objektet endast bidrar med de färger som kommer utifrån objektet. En liknande lösning finns i ATIs irradiance volume-demo för att hantera interpolering av närliggande koefficienter.

12.3 Framtagning av nya positioner för generering

Många av de optimeringar som finns i den slutgiltiga versionen bygger på att världen består av ett tvådimensionellt plan som är upphöjt och nedsänkt på olika ställen för att bilda ett landskap. I andra typer av spelvärldar, kan det vara vanligt med grottor, överhäng, stora inomhusmiljöer och liknande. Då får man antingen återgå till en mer generell version, eller hitta nya sätt att utföra optimeringarna på.

Nijasure använder djupinformation för skuggor i ambient belysning [22].

I den nuvarande versionen används inte djupinformationen i kubtexturen till annat än själva renderingen (Z-buffering). I miljöer med många väggar kan man tänka sig att dra nytta av djupinformationen för att avgöra var nya koefficienter skall tas fram och mellan vilka man skall interpolera.

Man bör ha i åtanke att skogsbeklädd mark påverkas endast i begränsad omfattning av himmelns ljus. Detta beror på att allt ljus som träffar marken har silats genom ett lövtäcke. Det direkta diffusa ljuset från solen försvinner därför och ersätts med en grön

ambiens. Detta och liknande specialfall skulle kunna användas för att snabba upp beräkningarna.

Generellt sett finns det många områden där ambiensen i stort är oförändrad. Exempel är stora skogar, stora fält eller över vatten. Om man skulle kunna identifiera sådana ställen skulle det räcka med att ha tätt mellan koefficienter i kanterna på dessa områden.

12.4 Rendering av stickprovsställen

Ett problem med den nuvarande metoden för rendering av de kubiska texturer som används för genereringen av koefficienterna är prestandan. Värst är det då det är många små och/eller högupplösta objekt som täcker synfältet. Ett sådant fall är tät skog. Stora optimeringar skulle kunna göras genom att hitta metoder för att rita en mer grovhuggen version av dessa objekt. Som det fungerar just nu så renderas alla objekt som skall finnas med i ambiensen i full upplösning. Visserligen är ritytan mycket liten, det går alltså fort att fylla den. Det som tar lång tid är istället transformeringen av alla objekt som skall vara med.

Vad som skulle behövas för att förbättra detta är antingen en metod för att förgenerera lågpolygonmodeller som ger liknande resultat vid SH-projicieringen eller att en artist som konstruerat ett objekt också har gjort en version av objektet med lägre detaljrikedom.

Om spelvärlden är procedurellt genererad borde ovanstående gå att utföra på ett ganska effektivt sätt, t.ex. genom materialegenskaper på terrängen.

12.5 Ytterligare tillägg

Om dynamiska ambiensvolymer skall användas vid produktionen av ett spel är det viktigt att grafiker snabbt kan se hur deras alster kommer att te sig i spelet. Man kan därför tänka sig en förhandsvisare där man i förväg sparat ett antal olika miljöer vid olika belysningssituationer, så att artisten kan förvissa sig om att inga skavanker syns extra bra vid något visst ljusförhållande, t.ex. att en ljus textur blir helt vit när det är starkt ambient ljus.

13 Slutsats

Den viktigaste slutsatsen är att systemet visserligen fungerar, men är alltför långsamt för att användas i sitt nuvarande skick. Förslag till optimeringar presenterades under framtida arbete.

Det visuella resultatet, en färgskiftning mot omgivningen hos dynamiska objekt, uppför sig som väntat och bidrar till helhetskänslan hos en renderad bild. Effekten kan dock uppfattas som något överdriven, beroende på att texturer och liknande inte är anpassade till systemet.

14 Referenser, litteraturförteckning

[1] Akenine-Möller, T., och Haines, E. *Real-Time Rendering*, A.K. Peters Ltd., 2nd edition, 2002, ISBN 1568811829.

[2] Bunnell, M. *Dynamic Ambient Occlusion and Indirect Lighting*, NVIDIA Corporation, 2005.

[3] Debevec, P. *Paul Debevec*, http://www.debevec.org/ , 02 feb 2006.

[4] ELFA AB, *Faktasidor från ELFA-katalogen*, 2004

[5] Green, R. *Spherical Harmonic Lighting: The Gritty Details*, Sony Computer Entertainment America, 2003.

[6] Greger, G., Shirley, P., Hubbard, P., Greenberg, D. *The irradiance volume*. IEEE Computer Graphics & Applications, 1998.

[7] Hargreaves, S. *Hemisphere Lighting With Radiosity Maps*, http://www.gamasutra.com/features/20030813/hargreaves_02.shtml , 02 feb 2006.

[8] Hoxley, J. *An Overview of Microsoft's Direct3D 10 API*, http://www.gamedev.net/reference/programming/features/d3d10overview/ , 02 feb 2006.

[9] Kaiser, P. *The Joy of Visual Perception*, http://www.yorku.ca/eye/ , 02 feb 2006.

[10] King, G. *Real-Time Computation of Dynamic Irradiance Environment Maps*. NVIDIA Corporation, 2005.

[11] Kontkanen, J., Laine, S. *Ambient Occlusion Fields*, Helsinki University of Technology, 2005.

[12] Lacroix, J. *Let There Be Light!: A Unified Lighting Technique for a New Generation of Games*, http://www.gamasutra.com/features/20050729/lacroix_pfv.htm , 02 feb 2006.

[13] Lehtinen, J. *Foundations of Precomputed Radiance Transfer*. Helsinki University of Technology, Finland, 2004.

[14] Malmer, M., Malmer, F., Assarsson, U., Holzschuch, N. *Fast Precomputed Ambient Occlusion for Proximity Shadows*, INRIA, Fankrike, 2005.

[15] Mantiuk, R., Pattanaik, S., Myszkowski, K. *Cube-map data structure for interactive global illumination computation in dynamic diffuse environments*, University of Central Florida, USA, Technical University of Szczecin, Polen, Max-Planck-Institut für Informatik, Tyskland, 2002.

[16] McTaggart, G. *Half-Life 2 / Source Shading*, Valve Software, 2004.

[17] Microsoft Corporation, *HDRCubeMap Sample*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/HDRCubeMap_Sample.asp , 02 feb 2006.

[18] Microsoft Corporation, *Hemispheric Lighting Tutorial*, http://msdn.microsoft.com/library/en-us/directx9_c/Goal_1___Hemispheric_Lighting_Tutorial.asp , 02 feb 2006.

[19] Mischler, G. *Lighting Design Glossary*: Adaptation, http://www.schorsch.com/kbase/glossary/adaptation.html , 02 feb 2006.

[20] Mischler, G. *Lighting Design Glossary*: Luminance, http://www.schorsch.com/kbase/glossary/luminance.html , 02 feb 2006.

[21] Mischler, G. *Lighting Design Glossary*: White Balance, http://www.schorsch.com/kbase/glossary/white_balance.html , 02 feb 2006.

[22] Nijasure, M., Pattanaik, S. *Real-Time Global Illumination on GPU*. University of Central Florida, USA, 2003.

[23] Oat, C. *Irradiance Volumes for Games*. ATI Research, 2005.

[24] Pattanaik, S. N., Tumblin, J., Yee, H., Greenberg, D. P. *Time-Dependent Visual Adaptation For Fast Realistic Image Display*, Cornell University, USA, 2000.

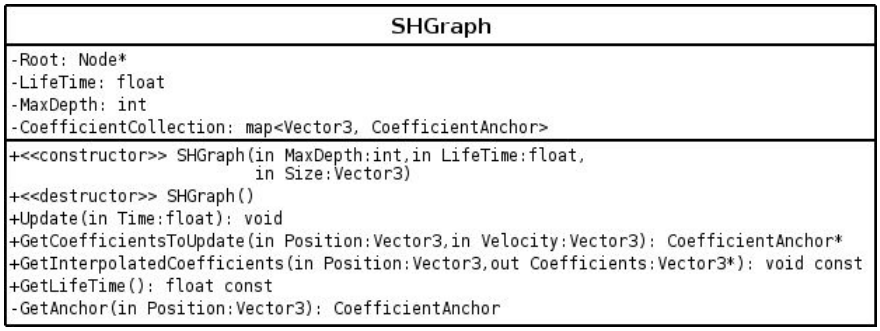
[25] Ramamoorthi, R., Hanrahan, P. *An Efficient Representation for Irradiance Environment Maps*, Stanford University, USA, 2001.

[26] StrangeBunny Team, *Shepherd of Dreams*, Texas A&M University, 2005

[27] Taylor, P. *Hemispheric Lighting Explained*, http://msdn.microsoft.com/library/en-us/directx9_c/Hemispheric_Lighting_Explained.asp , 02 feb 2006.

15 Bilaga B: Implementation av datastruktur

Datastrukturen delades upp i tre klasser. Interfacet som är tillgängligt för användaren är de publika funktionerna i klassen SHGraph. I denna klass återfinns också rotnoden för grafen samt en mappning med alla de instanser av CoefficientAnchor som skapats. CoefficientAnchor är den klass som förvarar koefficienter och som interpolerar dem långsamt över tiden när nya koefficienter genererats fram. Klassen Node bygger sedan upp grafen genom att hålla pekare en förälder, 6 grannar och 27 barn. Varje Node har också en pekare till en instans av CoefficientAnchor.



När grafen konstrueras så kalibreras den efter applikationen som den ska användas i. Den minimala frame rate som man kan förvänta sig bör styra vilket djup grafen har, då mängden genererade koefficienter per sekund är helt styrt av den frame rate man har. Har man kort sikt så kan volymen minskas. Därmed kommer också upplösningen öka. Hur lång tid en uppsättning koefficienter är giltig beror på dygnscykel, minneskrav och eventuella stora ändringar i världen.

Grafens volym ska inte sättas efter storleken på den värld som man använder. Om det behöver genereras koefficienter på en position utanför grafens befintliga volym så skapas en ny rotnod som är förälder till den gamla rotnoden. Den nya rotnoden stannar sedan kvar så länge det finns koefficienter i mer än ett av barnen, varefter den tas bort och grafen åter får den volym man ursprungligen angivit.

Funktionen som bestämmer vilken nod i trädet som skall få nya koefficienter avgör i hög grad den visuella kvalitén. Denna funktion bör välja ut noder som är optimala både med avseende på hur mycket som kommer att belysas av koefficienterna, hur sannolikt det är att kameran kommer att komma nära positionen så att ambiensens färg blir tydlig, hur stor skillnad det är mellan färgen vid en nod och de som redan är genererade i närheten o.s.v. En erfarenhet som våra test gav var att man måste kunna ljussätta allt i omgivningen för att inte något ska se underligt ut. Detta gör det lika viktigt att hålla en jämn upplösning på stora objekt som ett markplan, som att generera koefficienter tätt nära kameran. Att använda information om vart stora färgförändringar sker i en procedurellt genererad värld är en möjlighet som vi inte hunnit utvärdera, men den bör ge möjlighet att koncentrera upplösningen där det syns mest.

Hastighetsmässigt så är grafens mest kritiska funktion att snabbt kunna hitta de närmsta befintliga koefficienterna för en position. Grafen bör därför implementeras med fokus på att denna funktion ska gå så snabbt som möjligt. Sökningen startar alltid med att lokalisera den nod som befinner sig närmst den eftersökta positionen. Då denna nod känner till sina grannar så hittas koefficienter i omgivningen snabbt. Om inga giltiga koefficienter finns på det aktuella djupet så minskas upplösningen ett steg i taget, genom att nodens förälder och dess grannar undersöks. Funktionen avbryts om minst en uppsättning koefficienter hittats efter att en nivå av grannar genomsökts. Om mer än en uppsättning koefficienter har hittas så svarar funktionen med en linjärinterpolering av dessa. Maximalt används 8 uppsättningar koefficienter vid interpoleringen, en per oktant. I den slutgiltiga implementationen i Avalanche Engine användes dock inte fler än 4, då genereringen koncentrerades på markplanet.

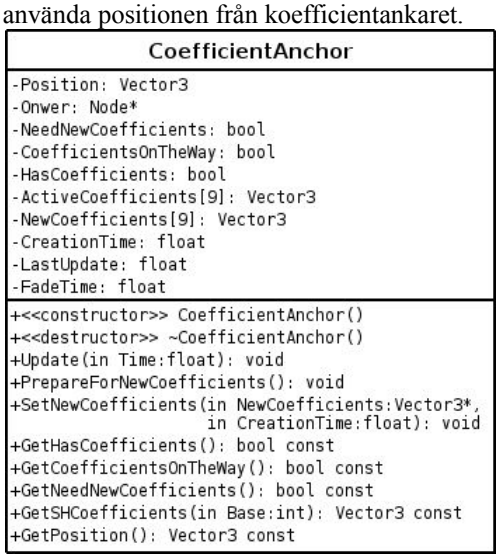
Uppdateringsfunktionens uppgift är att långsamt interpolera in nyligen genererade koefficienter. Detta görs för att undvika att nya färger ”ploppar” fram. Update ansvarar även för att plocka bort koefficienter som blivit för gamla, och att gallra bort grenar som inte längre används från grafen.

Att hålla en mappning med alla koefficienter och deras positioner i grafen var ett bekvämlighetsbeslut. Detta gör det enkelt att iterera genom alla koefficienter för att uppdatera dem. Det får dock som följd att noderna behöver kunna komma åt grafen för att erhålla en referens till sin koefficientuppsättning när de skapas. Node klassen är därför en friend klass till grafen, så att bara noderna kan skapa koefficienter.

Node
-Parent: Node* -Child[27]: Node -Neighbour[6]: Node -Coefficients: CoefficientAnchor* -Position: Vector3 -Size: Vector3 -Depth: int -ChildrenWithCoefficients: set<int>
+<<constructor>> Node(in Position:Vector3,in Size:Vector3, in Depth:int,in Parent:Node*) +<<destructor>> ~Node() +GetParent(): Node* const +GetChild(in ChildNr:int): Node* const +GetNeighbour(in Direction:int): Node* const +GetDepth(): int const +GetCenter(): Vector3 const +GetSize(): Vector3 const +HasChildren(): bool const +HasChildrenWithCoefficients(): bool const +GetChildrenWithCoefficients(): set<int>& const +GetCoefficientAnchor(): CoefficientAnchor* const +SetParent(in NewParent:Node*): void +SetChild(in ChildNr:int,in NewChild:Node*): void +SetNeighbour(in Direction:int,Neighbour:Node*): void +CreateNeighbourForChild(in RequestingChild:int,in Direction:int): Node* +Expand(): void +CoefficientsLost(): void +AddCoefficientOwningChild(in ChildNr:int): void +RemoveCoefficientOwningChild(in ChildNr:int): void

Klassen Node har som huvudsakliga uppgift att underlätta sökning av genererade koefficienter. När en nod skapas så hämtar den ett koefficientankare från grafen. Ankaret har exakt samma position som noden, och noden behåller ankaret så länge den finns kvar. För att slippa att kontrollera så att varje barn är giltigt när det besöks, så expanderas grafen alltid med 27 nya barn i taget, och alla 27 tas alltid bort samtidigt. För att inte få några ogiltiga pekare till grannar som tagits bort så används dessa pekare alltid i par, tas en nod bort försvinner pekarna åt båda håll.

Att spara ner index för de barn som har tillgång till giltiga koefficienter i ett eget ankare eller hos något av sina barn är en optimering som gjorts för att slippa söka bland övriga noder. Detta kan även användas för att gallra trädet; Om inga index är sparade i en nod så innebär det att samtliga barn kan tas bort. I implementationen i Avalanche Engine skulle man kunna göra c:a 25 kB minnesbesparing genom att räkna fram volym för noderna (m.h.a. djupet och volymen på grafen) vid förfrågan istället, samt använda positionen från koefficientankaret.



Klassen CoefficientAnchor är ansvarig för lagring av skapade koefficienter samt att hålla de koefficienter som används giltiga. För att ankare skall kunna hålla dess information giltig behöver de uppdateras kontinuerligt, så att nyskapade koefficienter långsamt kan interpoleras in och ankaret har tid att begära nya koefficienter innan de gamla blir ogiltiga. Ett ankare är passivt, d.v.s. anropar inga andra klasser, med ett undantag; om en uppsättning koefficienter blivit ogiltiga meddelar ankaret sin ägare om detta.

16 Bilaga C: Visuella resultat

Några skärmdumpar där skillnaden mellan klassisk ambiens, till vänster, och ambiensvolymen från detta arbete, till höger, kan ses.

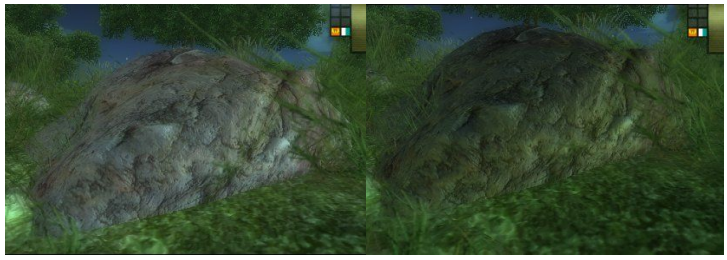
Här syns det klassiska exemplet med grönt gräs och blå himmel. Lägg märke till hur nedåtvinklade ytor reflekterar grönt ljus medan uppåtvinklade såsom karaktärens axlar blir blå på grund av infallande ljus uppifrån.

Både karaktärens kläder och de ljusa stenarna blir påverkade av att deras omgivning är grön åt i stort sett alla håll.

Marken och karaktären får en mer enhetlig belysning på grund av att de båda färgas av det molniga vädret.

Den blå omgivningen påverkar här stora delar av bilden. Om man låter alla delar av bilden påverkas kan man när bilden ritats filtrera denna för att korrigera för färgsticket.

Här syns normaltextureringen tydligt. De stenytor som är riktade nedåt antar en grön ton medan de riktade uppåt blir färgade av himmeln.



Lägg märke till hur karaktärens grå dräkt blir något färgad av att vara nära sluttningen på bilden till höger. En viss blåfärgning av den öppna ytan nära stranden kan också skönjas.

Här är effekten mer subtil. På grund av de många träden får det ljus som träffar marken en grönaktig ton på den högra bilden.