CHALMERS

Metoder för att öka realismen vid animerade 3D-förlopp

Master of Science Thesis

MARTIN REBAS

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, January 2015

Metoder för att öka realismen vid animerade 3D-förlopp

MARTIN REBAS

Examiner: ULF ASSARSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden January 2015

Metoder för att öka realismen vid animerade 3D-förlopp

Methods for enhancing the realism of animated 3D sequences

Ett examensarbete i datavetenskap vid Chalmers Tekniska Högskola
utfört på uppdrag av Carmenta AB

Utfört av:	Martin Rebas
Handledare på Carmenta:	Andreas Björnberg
Handledare på Chalmers:	Niklas Eén
Ursprunglig examinator:	David Sands
Examinator:	Ulf Assarsson

CHALMERS

carmenta
geospatial innovations

Sammanfattning

Denna rapport är resultatet av ett examensarbete utfört vid Chalmers Tekniska Högskola, på uppdrag av Carmenta AB. Målsättningen har varit att undersöka och implementera metoder för att öka realismen hos animerade 3D-förlopp. Carmenta har utvecklat ett geografiskt informationssystem med namnet SpatialAce, där kartor och höjddata kan kombineras till en tredimensionell, animerad vy av ett landskap. I samråd med handledare och andra på Carmenta bedömdes realtids-generering av skuggor vara ett lämpligt projekt, varefter en litteraturstudie över tidigare forskning genomförts, och en passande skuggalgoritm har integrerats i en testversion av SpatialAce.

Abstract

This paper is the result of a diploma project made at Chalmers University of Technology on assignment from Carmenta AB. The goal has been to examine and implement methods for increasing the realism of three-dimensional animations. Carmenta has developed a geographic information system named SpatialAce, where maps can be combined with altitude data to create a three-dimensional, animated landscape view. After consultation with Carmenta, the project's focus became real-time generation of shadows. A literature study of previous research was performed, and a suitable shadow generation algorithm was chosen and integrated into a test version of SpatialAce.

Förord

Detta examensarbete utfördes på institutionen för Datavetenskap på Chalmers Tekniska Högskola i Göteborg, ursprungligen med David Sands som examinator och Niklas Eén som handledare. På grund av belastningsskador försenades färdigställandet av rapporten med flera år och Ulf Assarsson trädde därför in som ny examinator. Arbetet utfördes på uppdrag av Carmenta AB, där Andreas Björnberg var handledare. Jag skulle vilja tacka dem för all hjälp.

Innehåll

Figurer

1 Inledning

En av Carmenta AB:s produkter är ett GIS (Geografiskt informationssystem) vid namn SpatialAce, med vilket man kan visualisera och analysera geografiska data i två eller tre dimensioner. I tre dimensioner använder SpatialAce terräng- och höjddata för att skapa landskap, som bl a innehåller vägar, floder, skog och bebyggelse. Man vill dock öka realismen ytterligare, och i samråd med min handledare beslutades det att detta lämpligtvis borde ske genom att man lägger till dynamiskt genererade skuggor till programmet.

Skuggor ger ögat viktig information om formen på och förhållandet mellan olika 3D-objekt, och gör att de uppfattas som mer solida. Det är dock inte ett trivialt problem att skapa skuggor i realtid. Att gå igenom alla polygoner för sig, och räkna ut vilka som skymmer ljuskällor för vilka, tar lång tid att räkna ut även för jämförelsevis enkla scener. Det krävs effektiva algoritmer för att minska beräkningstiden, men dessa algoritmer gör alla avkall på realismen på olika sätt, och beroende på förutsättningar och mål så kan olika algoritmer kan vara lämpliga. Av denna anledning finns det inga inbyggda funktioner för att skapa skuggor i något grafik-API, som t ex OpenGL eller Direct3D, utan man får själv avgöra vilken algoritm som passar bäst för ens syften.

Målsättningen med detta examensarbete har varit att analysera olika metoder för att generera skuggor i realtid, och att implementera en passande skuggalgoritm i SpatialAce. SpatialAce är skrivet för PC, och grafikhanteringen sker med OpenGL, vilket därför använts som API.

Rapportens första del förklarar vissa grundläggande begrepp, som behövs för att förstå fortsättningen. I delen därefter finns en översikt över olika skuggalgoritmer, och hur de fungerar. Till sist beskriver jag vilken algoritm jag valt och min implementering av den i SpatialAce, med resultat och slutsatser. Därefter följer programkod från min implementering, en mer detaljerad ordlista, samt en referensförteckning. Till sist ligger examensarbetesspecifikationen som en bilaga.

2 Grundläggande begrepp

2.1 SpatialAce

Carmenta beskriver SpatialAce som ”en avancerad verktygslåda för att bygga interaktiva geografiska applikationer”. Geografiska data om vägar, bebyggelse, markhöjd, etc kan kombineras med satellitbilder och användas för att generera ett 3d-landskap som man kan röra sig igenom. SpatialAce är programmerat i C++, medan OpenGL används för grafikhanteringen.

2.2 OpenGL

OpenGL är ett API, ett gränssnitt, som specifierar funktioner som underlättar programmering av 2D- och 3D-grafik. [Woo97]

2.3 Polygoner

En polygon är en månghörning (t ex en triangel eller en fyrhörning). Objekt i 3D-grafik är normalt uppbyggda av trianglar, eftersom det är den form som är lättast att hantera för grafikkort. Vill man rita ut objekt bestående av mer komplicerade polygoner, så måste dessa först tesselleras – konverteras till trianglar.

2.4 Texturmappning

Texturmappning innebär att man ökar detaljnivån och realismen hos 3d-grafik genom att täcka polygoners ytor med bilder. Bilder som används på detta sätt kallas texturer, och ett bildelement i en textur kallas en *texel*.

2.5 Buffertar

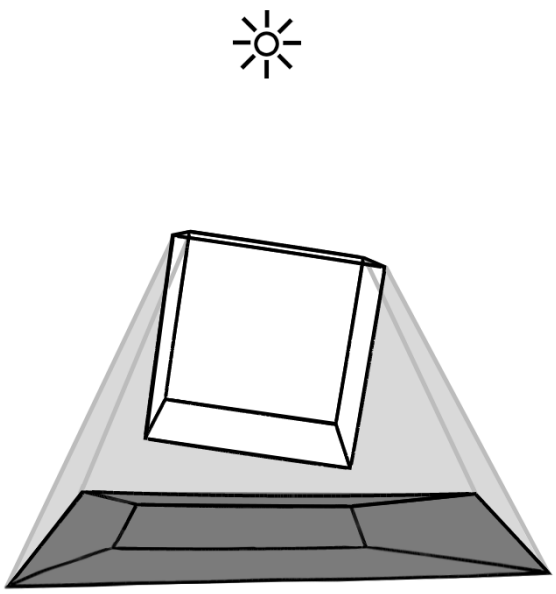
En buffert är ett minnesutrymme för att lagra bilder eller annan information som gäller pixlarna i ritytan. En färgbuffert innehåller färginformation, medan en djupbuffert för varje pixel lagrar avståndet till objektet som pixeln representerar. En stencilbuffert kan användas som schablonmall för att avgöra vilka delar av bilden som ska ritas ut. En ackumulationsbuffert kan användas för att kombinera flera bilder till en (och t ex skapa ett medelvärde av flera bilder).

En mer utförlig ordlista finns i slutet av rapporten.

3 Olika metoder för att generera skuggor

3.1 Projektionsskuggor (projection shadows)

Ett enkelt sätt att generera skuggor är att helt enkelt rita ut varje objekt två gånger, där den andra kopian färgas i den färg man vill att skuggan ska ha och projiceras ned mot ett plan med hjälp av en projektions-transformation [Blyt97]. Skuggan utgörs alltså av en flat kopia av det skuggande objektet. I och med att skuggan och planet hamnar i samma plan, och det därmed kan bli en konflikt om vilken som ska synas, så får man använda t ex *polygon offset*-funktionen i OpenGL, som flyttar det ena objektet en aning i djupled och därmed ser till att båda ritas ut.



Figur 1: En kub projiceras ned på en plan yta.

Denna metod har ett begränsat användningsområde, då man endast kan kasta skuggor på plana ytor. För att kasta en skugga på en mångfasetterat objekt måste man projicera ned ett skugg-objekt för varje fasett som skuggan faller på, och sedan beskära dessa objekt efter fasetternas kanter. I vissa specialfall krävs dock ingen beskärning – om man t ex kastar en skugga på insidan av en konvex polyeder, kommer de överflödiga delarna av skugg-objektet automatiskt att döljas, eftersom de befinner sig bakom polyedern.

Om objektet är konkavt kommer det inte att skugga sig själv. Den obelysta sidan blir inte heller automatiskt mörkare än den belysta, men den sistnämnda effekten går att approximera genom att ljussätta polygonerna baserat på vinkeln mellan ljuskällan och

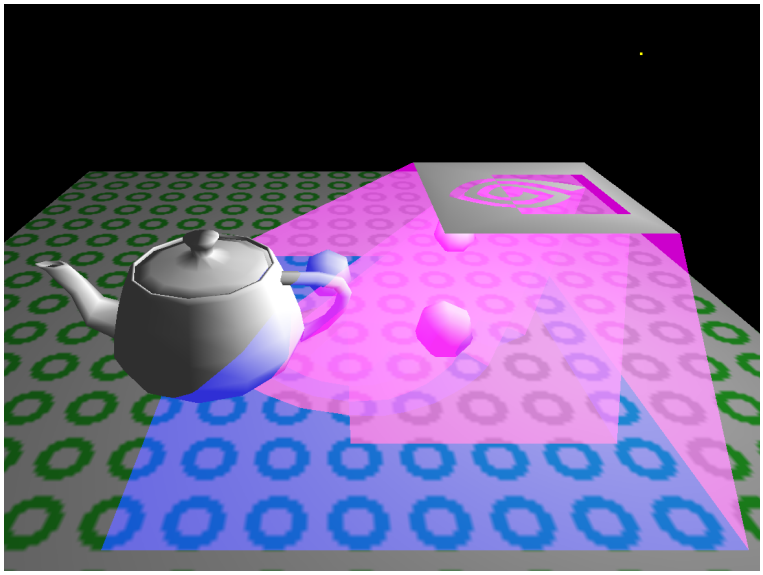
polygonernas normalvektor, så att ytor görs ljusare i den grad de är riktade mot ljuskällan, och mörka om de är bortvända.

Om ytan som skuggan faller på är flerfärgad eller texturmappad, så kan man lösa det med en *stencilbuffert;* när scenen ritas ut sätter man stencilbufferten till 1 på de pixlar som ligger i skugga. Sedan renderar man hela scenen en gång till, med svagare ljus, men denna gång fyller man endast i de områden där stencilbufferten är satt till ett. Ett annat alternativ är att skugg-objektet helt enkelt ritas ut halvgenomskinligt – detta fungerar med konvexa polyedrar, men med andra former kan olika delar av skuggan bli olika mörka.

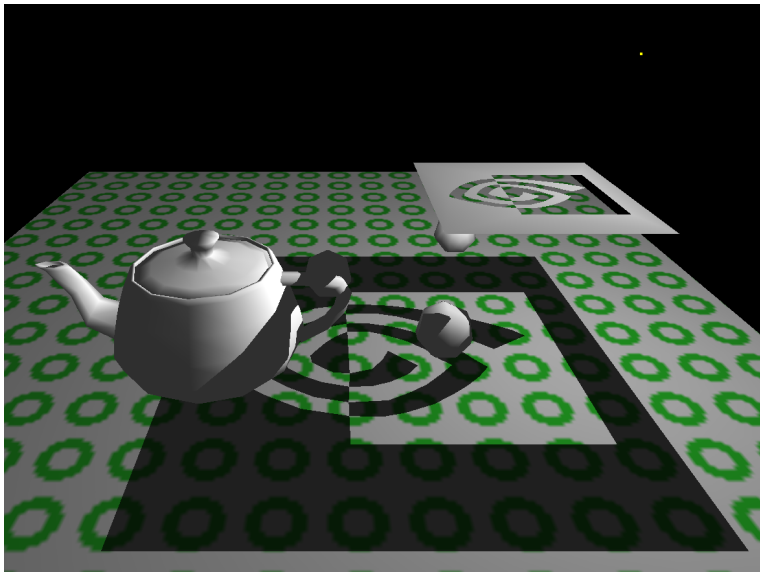
Metoden är mycket enkel att implementera, men svår att använda för annat än konvexa objekt som kastar skuggor på en plan yta. Därtill så kan beräkningskraft slösas genom att polygoner som inte tillför något till skuggan ändå ritas ut.

3.2 Skuggvolymer (shadow volumes)

Denna teknik [Crow77] går ut på att de områden i rymden som inte nås av ljuskällan – d v s hela volymen mellan det skuggande objektet och de ytor som skuggan faller på – innesluts av polygoner. Dessa polygoner syns inte själva, men med hjälp av dem kan man räkna ut vilka områden som ligger i skugga.



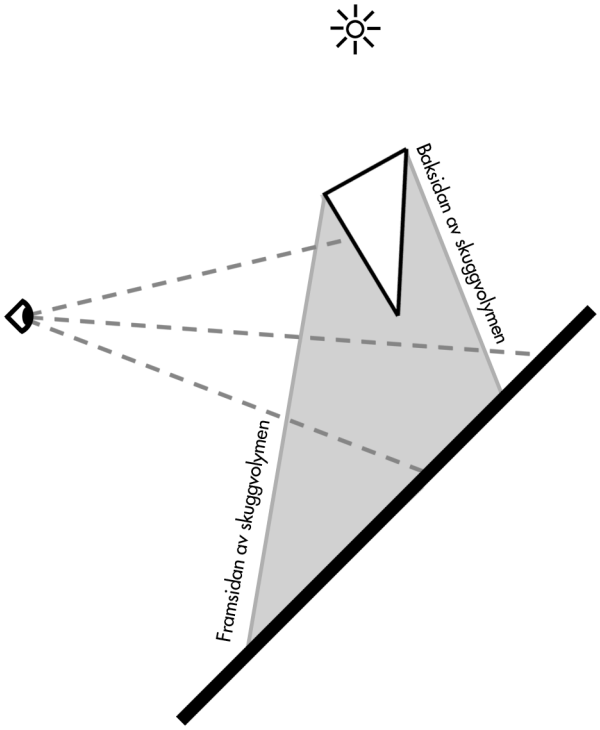
Figur 2: En visualisering av (de vanligtvis osynliga) skuggvolymerna mellan en Nvidia-symbol och andra objekt. (Nvidia Software Engineering, 1999)



Figur 3: De resulterande skuggorna. I och med att skuggvolymer endast räknats ut för symbolen, så kastar vare sig bollar eller tekanna några skuggor.

Om en stencilbuffert används, vilket är vanligt, så gör man på följande sätt:

Skuggvolymerna räknas ut genom att projicera linjer från ljuskällan genom hörnen på det skuggande objektets silhuett. Man renderar sedan de skuggvolympolygoner som ligger framför alla synliga polygoner, och samtidigt ökar man stencilbuffertens värde varje gång man ritar en polygon som hör till "framsidan" på en skuggvolym, och minskar den varje gång man ritar en polygon som hör till skuggvolymens baksida. Om man på en viss pixel ritat ut fler polygoner som hör till framsidan av skuggvolymen än som hör till baksidan, blir stencilbuffertvärdet för denna pixel mer än noll, och man vet man att den befinner sig i skugga.



Figur 4: Ljuset från en ljuskälla skyms av en polygon (den vita triangeln). Området under polygonen ligger i skugga. Till vänster finns ett öga, och tre streckade linjer har ritats ut för att visa vad ögat ser i olika riktningar. Den översta linjen går igenom framsidan på skuggvolymen, och träffar sedan triangeln; stencil-bufferten kommer alltså att vara satt till ett, och området ska vara skuggat. Nästa linje går först igenom skuggvolymens framsida, vilket ökar stencilbufferten till ett, och sedan genom dess baksida, vilket minskar stencilbufferten till noll igen; området är alltså belyst. Den tredje linjen går liksom den första endast genom framsidan, och hamnar alltså i skugga.

Skuggvolymer kan vara ett effektivt sätt att skapa skuggor från enkla objekt, och kan hantera ljuskällor som befinner sig mitt bland objekten. Rör det sig om mer komplicerade former, så kan det bli svårt att räkna ut volymerna. Därtill så måste man skapa en ny skuggvolym för varje enskilt objekt, och förändra den varje gång ljuskällan eller objekten rör sig. Om volymerna täcker en stor del av skärmen så kan själva utritandet av skuggvolym-polygonerna kräva mycket beräkningstid.

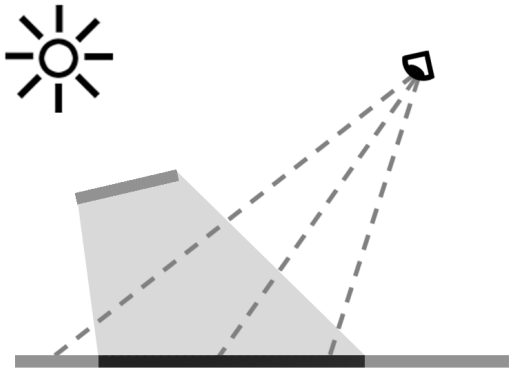
Man måste även räkna ut ifall kameran hamnat inuti en eller flera skuggvolymer, för att stencilbuffertvärdena ska bli rättvisande. Ett sätt att komma runt det är att inte utföra beräkningarna direkt på skuggvolymerna, utan på unionen mellan skuggvolymerna och vy-frustumet, vilket gör att kameran automatiskt hamnar utanför volymerna.

Skuggvolymer kan I vissa fall göras effektivare med hjälp av rymdpartitioneringsalgoritmer som BSP-träd eller Shadow Tiling. [Slater92]

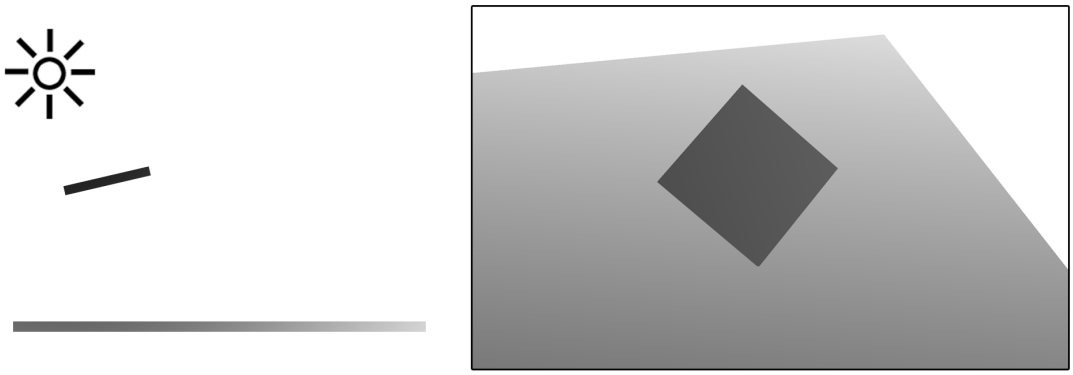
3.3 Skuggmappning (shadow maps)

Denna teknik använder sig av djupbufferten och projicerade texturer. Först renderar man en djupbuffert från ljuskällans ”synvinkel”. Denna sparas undan som en textur. Sedan så projicerar man ned djupbuffert-texturen från ljuskällan, återgår till ögats synvinkel, och för varje pixel så jämför man den projicerade texturens färg med avståndet till ljuskällan. [Will78] [Heid99]

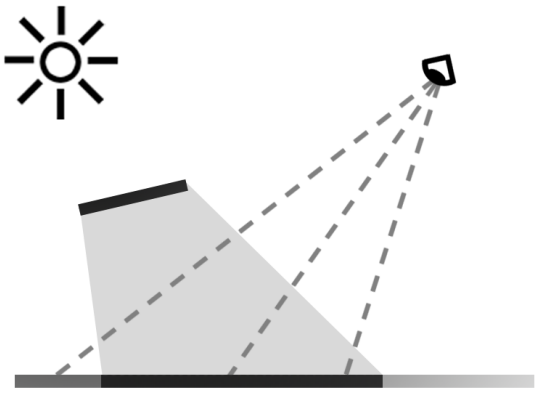
Ett exempel, för att göra det tydligare:



Figur 5: Här visas (från sidan) ett mindre plan som kastar en skugga på ett större plan. Från ögats synvinkel ligger vissa delar av det större, undre planet i skugga, andra inte.



Figur 6: Scenen renderas från ljuskällans synvinkel till en djupbuffert. Till vänster: Ett diagram från sidan där planen färgats ljusare ju längre de är från ljuskällan. Till höger: Ett exempel på hur djupbufferten, skapad från ljuskällans synvinkel, kan se ut.



Figur 7: Djupbufferten projiceras ned på planen.

Nästa steg är att projicera djupbufferten från ljuskällan, likt en filmprojektor som projicerar en bild på en duk, fast där projektionen även påverkar underliggande ytor. Som syns i figur 7 så blir den skuggade delen av det undre planet nu mörk, eftersom den mörka färgen som det mindre planet får i djupbufferten projiceras ned på de underliggande ytorna.

Alltså: Den projicerade djupbufferten gör att planen färgas ljusare i proportion till avståndet till ljuskällan, *förutom* för ytor som är bakom andra ytor; dessa får samma färg som objektet som kastar skuggan.

Allt som är kvar är att jämföra värdet som den projicerade texturen ger varje pixel med motsvarande punkts avstånd till ljuskällan, och se ifall de skiljer sig. I diagrammen ovan innebär det att jämföra planens färg i figur 6 (till vänster) med figur 7; de delar som är markant mörkare i figur 7 anses ligga i skugga.

Ifall ljuskällan är mycket långt borta (t ex solen), så kan en parallell projektion användas för djupbufferten och dess projektion, vilket motsvarar en ljuskälla på oändligt långt avstånd. I stället för att jämföra med avståndet till ljuskällan så får man göra en lämplig jämförelse mellan djupbufferten och punkternas placering i ljusets riktning.

Eftersom djupbuffertar inte har oändlig detaljrikedom och precision, så kan aliasing-problem uppstå, och orsaka oavsiktlig självskuggning, i de fall där skillnaden mellan texturfärg och avstånd ligger nära noll. Det är därför lämpligt att inte rita ut skuggor såvida inte skillnaden är ett visst värde *över* noll.

Några fördelar med metoden är att det tar lika lång tid att skugga enkla scener som komplexa, och att den hanterar självskuggning. Metoden fungerar även om scenen innehåller texturer med genomskinliga delar (som inte ska kasta skuggor). En nackdel är

att det krävs stora texturer för att skuggorna inte ska bli väldigt grovkorniga, vilket är processor- och tidskrävande. (Detta är främst ett problem om vissa objekt är mycket nära kameran, och andra långt borta). En annan nackdel är att metoden fungerar bäst när ljuset är riktat åt ett visst håll (och det därmed räcker med att rendera en djupbuffert åt just det hållet). Den fungerar illa för en ljuskälla som befinner sig mitt i scenen och lyser åt alla håll.

Skuggmappning stöds inte av OpenGL 1.1, men det finns OpenGL-tillägg som kan användas för att utföra jämförelsen mellan avstånd och projicerat djupbuffertvärde.

3.4 Skuggvolym-rekonstruktion (shadow volume reconstruction)

Denna metod är en variant av skuggvolymtekniken, där skuggvolymerna inte räknas ut direkt utifrån de polygonkoordinater man känner till, utan där man räknar ut volymerna med hjälp av en djupbuffertbild från ljuskällans synvinkel, och avancerade kantigenkänningstekniker. [McCo98]

3.5 Framåtriktad skuggmappning (forward shadow mapping)

Forward shadow mapping [Zhan98] är en variant på skuggmappningstekniken, som utvecklats av Hansong Zhang, där jämförelsen sker mellan en djupbuffert från kamerans synvinkel och en djupbuffert från ljusets synvinkel, där punkterna i den senare först transformerats till kamerans vy.

3.6 Snabba mjuka skuggor

De ovan beskrivna metoderna genererar normalt skuggor med skarpt avgränsade kanter. I verkligheten har skuggor sällan skarpa kanter, dels eftersom ljus reflekteras in i skuggor via näraliggande ytor, och dels eftersom verkliga ljuskällor inte är punktformiga, utan har en utsträckning, vilket gör att ljus från en del av ljuskällan kan nå platser som är dolda för andra delar av samma ljuskälla.

Paul S. Heckbert och Michael Herf beskriver en metod för att skapa mjuka skuggor, som kan användas i realtids-3D-grafik [Heck96] [Herf97] [Heck97]. För varje skugg-mottagare (objekt som kan få skuggor kastade på sig) genomförs följande procedur: Varje ljuskälla representeras som en samling näraliggande punktljuskällor. Man undersöker därefter vilka objekt som ligger emellan punkterna och skugg-mottagaren, och objekten projiceras till en textur som motsvarar skuggan från just denna punkt. Med hjälp av ackumulationsbufferten så kombineras texturerna från de olika punkterna; man skapar en textur som blir ett medelvärde av skuggorna från varje punkt, vilket ger den mjukare kanter (även om väldigt många punkter krävs för att de ska upplevas som helt mjuka). Till sist läggs texturerna på de objekt som ska skuggas.

Figur 8: Ett exempel på Heckbert och Herfs metod för att skapa mjuka skuggor.

Metoden ger jämförelsevis realistiska resultat, men kräver förhandsberäkningar för att skapa texturerna (för bilden ovan tog beräkningarna 13 sekunder på en SGI Crimson med RealityEngine-grafik). Scenen med skuggorna kan sedan visas i realtid i olika vinklar, men ifall ljuskällor eller objekt rör sig måste en ny, tidskrävande beräkning genomföras och nya texturer skapas.

4 Testimplementering

Denna del redovisar mina testimplementeringar med resultat och slutsatser.

4.1 Omgivning

All programmering utfördes i C++, och utvecklingsmiljön var Microsoft Visual C++ 6.0. Den datorutrustning som använts är en PentiumIII på 550 MHz, med 256 MB minne, och operativsystemet var Microsoft Windows NT 4.0. De grafikkort som använts är Nvidia GeForce 256 samt Nvidia Riva TNT2. De är båda bland de snabbare grafikkort som finns till PC-datorer, med gott stöd för hårdvaruacceleration av 3D-grafik, men utveckligen går fort, och man kan vänta sig att 3-4 gånger snabbare kort finns tillgängliga inom ett år.

Grafikhanteringen skedde med hjälp av OpenGL 1.1, vilket föll sig som ett naturligt val, eftersom det är det API som används för all grafikhantering i SpatialAce. GLUT (OpenGL Utilities Toolkit) användes för att skapa fönster, vissa 3d-objekt, samt för att hantera tangentbordsinmatning.

4.2 Val av metod

Den metod som ger mest realistiska resultat är Heckbert och Herfs. Tyvärr så kräver metoden förhandsberäkningar och är därför främst lämplig för statiska scener där man ser orörliga objekt från olika vinklar. Då simuleringarna i SpatialAce kan innehålla rörliga objekt (som bör kunna kasta och ta emot skuggor) måste andra metoder användas.

Landskapen i SpatialAce består till viss del av texturer med genomskinliga delar. Exempelvis så ritas träd ut genom att bilder på träd, med genomskinlig bakgrund, används som textur på vad som egentligen är rektangulära ytor. Rent polygonbaserade algoritmer som projektionsskuggor och skuggvolymer kan alltså inte få grenar och löv att kasta skuggor i SpatialAce, utan då får man vända sig till de djupbuffert-baserade algoritmerna, som kan fås att ta hänsyn till genomskinliga texturer.

Valet föll i första hand på vanlig skuggmappning, som är den mest etablerade, och förmodligen den mest framtidssäkra av metoderna; i jämförelse så innehåller skuggvolym-rekonstruktion en mängd undantagsfall som kan försvåra implementeringen i en stor värld där man kan röra sig fritt (som i SpatialAce). Och jämfört med framåtriktad skuggmappning är vanlig skuggmappning enklare att implementera med hårdvarustöd. Därtill så gör den snabba utvecklingen på grafikkortsfronten att eventuella prestandaskillnader mellan de djupbuffert-baserade algoritmerna blir mindre relevanta, utan den stora frågan är ifall *någon* av dem är snabb nog att skapa realtids-skuggor av tillräcklig kvalitet i SpatialAce.

4.3 Test av skuggmappning

Innan jag implementerade skuggmappningsalgoritmen i SpatialAce, skrev jag ett testprogram för att bekanta mig med algoritmen, och för att se ifall den var snabb nog för

att användas i realtid. Testprogrammet genererade ett färgglatt bergslandskap och lät en tekanna flyga igenom det (tekannan är ett standardobjekt i OpenGL, och användbart ifall man behöver ett enkelt, delvis konkavt, objekt). Nvidia hade nyligen släppt ett OpenGL-tillägg för skuggmappning, med demoprogram som demonstrerade både skuggmappning och skuggvolymer, vilket hjälpte mig att förstå hur det kunde implementeras.

4.4 Prestandatest

Test-scenen innehöll ca 10.000 texturmappade polygoner. Upplösningen var 800x600 pixlar, och de antecknade hastigheterna i tabellerna är ett genomsnitt efter 1000 frames. Det grafikkort jag hade tillgängligt klarade maximalt 512x512-pixlars texturer; 1024x1024-pixlars gick att skapa, men hela texturen lagrades inte korrekt. För praktisk användning så var alltså 512x512 pixlar maximum, men jag har ändå inkluderat de värden jag fick med en 1024-pixlarstextur. Jag gjorde samma test med och utan kontinuerlig uppdatering av djupbufferten, för att se vilka delar som tog mest tid.

Med kontinuerlig uppdatering av djupbufferten:

Texturstorlek	Frames per sekund	Sekunder per frame
64x64	12,35	0,08
128x128	10,21	0,1
256x256	6,03	0,17
512x512	2,22	0,45
1024x1024	0,83	1,21

Utan kontinuerlig uppdatering av djupbufferten:

Texturstorlek	Frames per sekund	Sekunder per frame
64x64	16,68	0,06
128x128	16,68	0,06
256x256	16,68	0,06
512x512	16,41	0,06
1024x1024	16,15	0,06

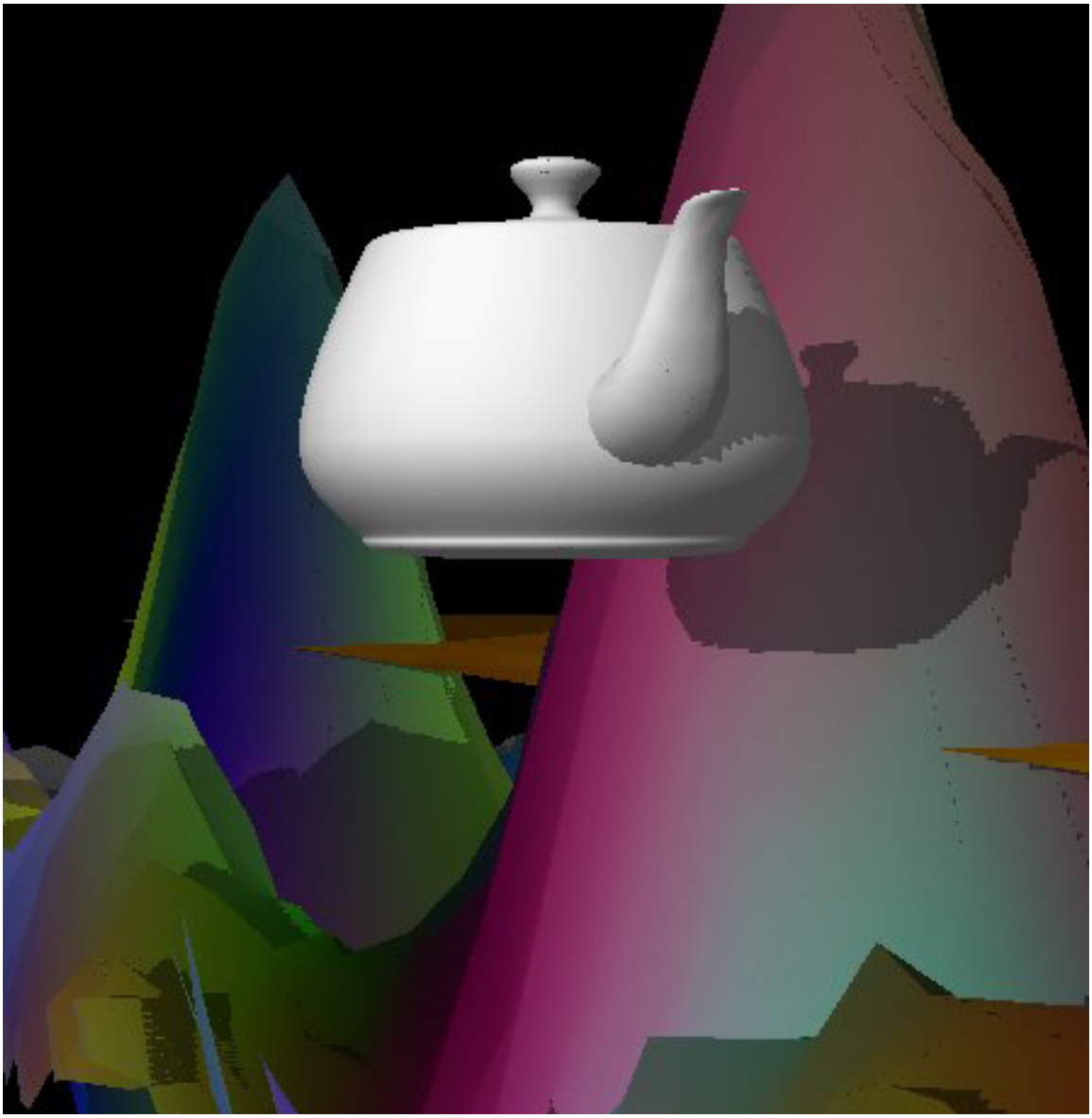
Utan skuggor:

-	58,88	0,02

Det är tydligt att hastigheten i mycket hög grad påverkas av huruvida djupbufferten uppdateras, och dess storlek. Med 512x512 pixlars texturer ser skuggorna en aning grovkorniga ut, men det är ändå en betydande ökning av realismen jämfört med inga skuggor alls. I samråd med handledare bedömdes denna algoritm vara användbar i SpatialAce; visserligen är det inte riktigt realtidshastigheter än, men som nämnts har

grafikkorten blivit 3-4 gånger snabbare varje år, och trenden ser ut att fortsätta, så det kommer förmodligen inte dröja länge innan grafikkorten hanterar skuggmappning utan problem.

Under examensarbetets gång släppte Nvidia förbättrade drivrutiner till sina kort, vilket ökade prestandan avsevärt. Särskilt märktes förbättringar när det gäller tidsåtgången för att skapa och lagra undan djupbufferten; denna tid minskade till hälften.



Figur 9: En bild från testprogrammet. Notera att pipen skuggar resten av tekannan. Bilden är en inzoomad detaljvy där man kan se skuggornas grovkornighet i jämförelse med den övriga grafikens skärpa.

5 Implementeringen i SpatialAce

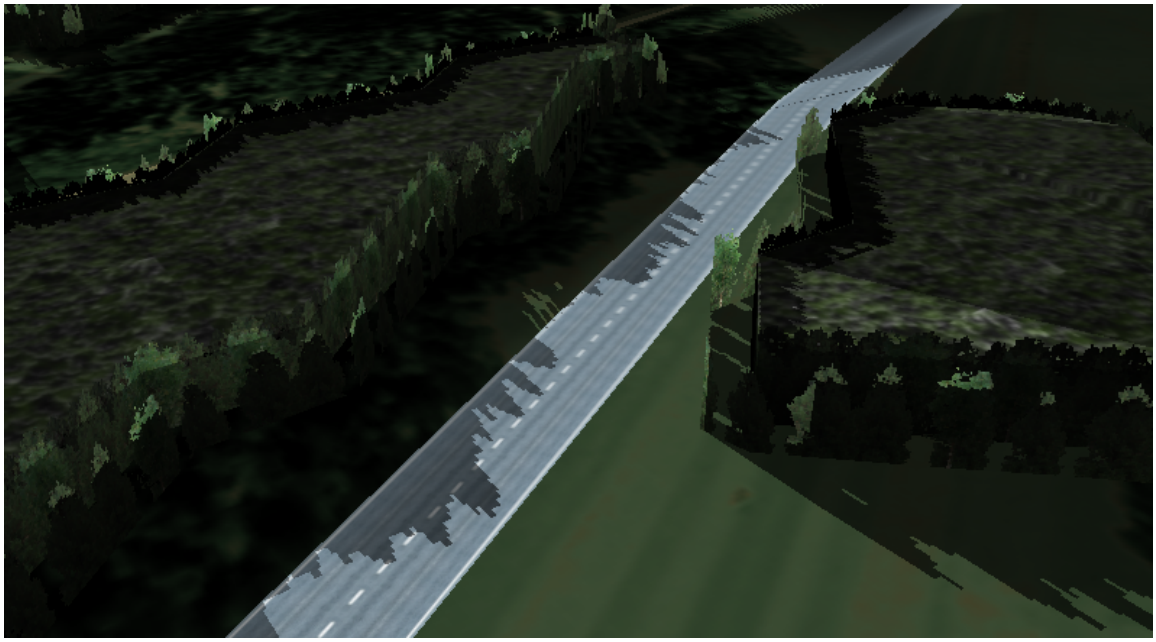
Då genereringen av skuggor påverkade den grundläggande grafikhanteringen i SpatialAce, och inte bara lade till ett objekt eller en operator, så fick algoritmen byggas in i kärnan av SpatialAces grafikhantering.

Eftersom ljuskällan var tänkt att föreställa solen så projicerades skuggorna parallellt, med ett rätblocksformat frustum. Och då landskapen kan vara mycket stora, men detaljrikedomen hos skugg-mappningen är begränsad, så valde jag att inte projicera skuggorna över hela landskapet, utan bara över ens närmaste synfält. Skugg-projektionen följer alltså betraktarens rörelser så att skuggorna alltid täcker området framför betraktaren. Efter lite experimenterande hittade jag värden för storlek och placering på projektionen som fungerade bra.

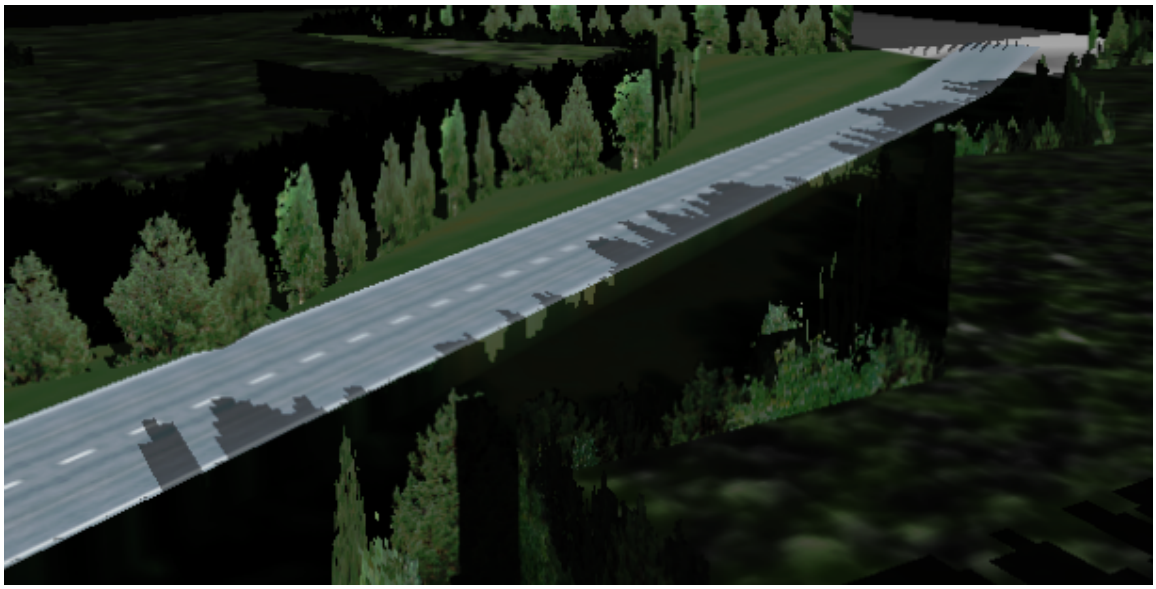
I och med att jag redan implementerat skuggmappning i mitt testprogram, så gick det relativt enkelt att skapa skuggor i SpatialAce, även om det krävdes en del modifikationer för att få skuggorna att fungera med delvis genomskinliga texturer. För projektionen använde jag en 16-bitars djupbuffert (vilket krävde två separata renderingspass samt användandet av ett OpenGL-tillägg) samt en textur på 512x512 pixlar.

6 Resultat och slutsatser

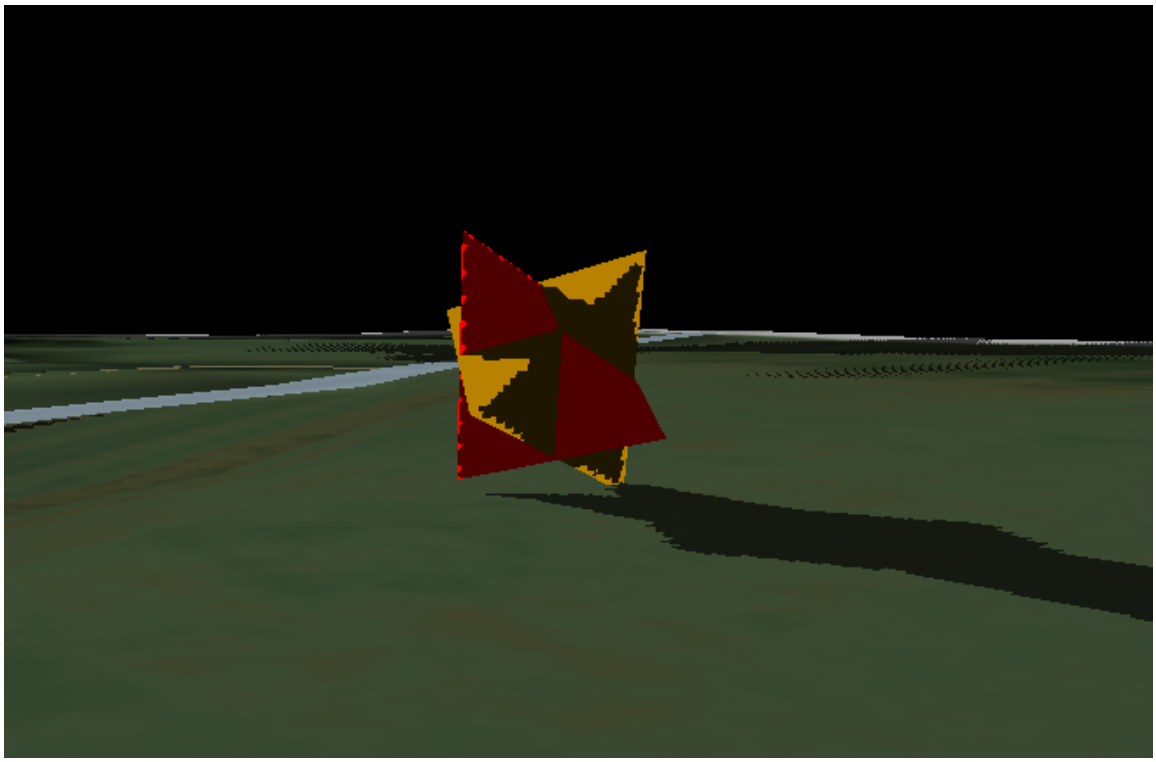
Här följer några bilder från min skuggimplementation, med kommentarer:



Figur 10: Exempel på skuggor från delvis genomskinliga texturer i SpatialAce. Skogsområden representeras i programmet av en polyeder, texturmappad i skogsfärger, omgiven av två lager av bilder på träd, med genomskinlig bakgrund, så att trädens silhuetter framträder.



Figur 11: Ännu en skärmbild från SpatialAce.



Figur 12: Skuggat realtids-roterande objekt i SpatialAce

Landskapet jag hade att jobba med innehöll inga rörliga objekt, så till min presentation på Carmenta så lade jag till en roterande figur, uppbyggd av två tetraedrar, som kunde demonstrera realtidsaspekten av skuggalgoritmen. Skuggorna ser ut som tänkt så när som på att den relativa grovkornigheten och bristande precisionen gör att självskuggningen inte når ut i kanterna. Detta kan delvis döljas genom att låta ljussättningen matcha skuggorna bättre, så att polygoner som är bortvända från ljuskällan ges samma färg som de skuggmappade skuggorna (som i exemplet med tekannan ovan).

Trots mindre skönhetsfel så förbättrar skuggorna realismen betydligt i SpatialAce; vad som främst saknas är mer detaljerade texturer och högre uppdateringsfrekvens. Med 512x512-pixlars texturer till skuggorna blev uppdateringsfrekvensen fyra frames per sekund, vilket är lågt, men borde innebära att metoden blir användbar i realtid inom ett till två år, när grafikkortens kapacitet ökat.

7 Utdrag ur kod

I denna del visas utdrag ur koden i form av utvalda funktioner som jag använde till mitt algoritm-prestandatest, samt den funktion, *showShadowed(),* som är ansvarig för skuggmappning i SpatialAce.

```
/*
------------------------------------------------------------------------

shadowtest.cpp

Draws a colourful landscape with a flying teapot, optionally with
shadowmapped shadows, while moving the camera and recording the average
number of frames per second.

main():
Initializes the drawing window and glut. Calls loadTextures(). Sets the
display callback to a function that, depending on the drawing mode,
calls one of several functions, one of which is drawEyeView().

loadTextures():
Creates a randomized mountain landscape and assigns a colourful texture
to it.

drawEyeView():
Updates and draws the scene with the help of drawScene() and
drawLandscapeTexture().

------------------------------------------------------------------------
*/

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <math.h>
#include <GL/glut.h>
#include <GL/tube.h>
#include <gl/gl.h>          // Header File For The OpenGL32 Library
#include <gl/glu.h>         // Header File For The GLu32 Library
#include <gl/glaux.h>       // Header File For The Glaux Library

#include <time.h>           // Nödvändigt för prestandamätning
#include <sys/types.h>
#include <sys/timeb.h>
#include <string.h>

#define lSize 95
GLdouble meshCoords[lSize+1][lSize+1][3];

int updates = 0; // Prestandatest
time_t ltime;
time_t ltime2;
```

```
void drawEyeView(void)
{
   glLightfv(GL_LIGHT0, GL_POSITION, lv);

   drawScene();
   drawLight();
   drawLandscapeTexture();

   updates++; // Mät uppdateringsfrekvens
}


void drawScene(void)
{
   eyeX+= cos(eyeA)/20;
   eyeY+= sin(eyeA)/20;
   eyeA+= .03;

   if(updates<1000) {
      needMatrixUpdate = 1;
      glutPostRedisplay();
   }

   glMatrixMode( GL_MODELVIEW );
   drawLandscape();
   glPushMatrix();

   glTranslatef(eyeX+cos(eyeA)*3, 0.5, eyeY+sin(eyeA)*3);
   glFrontFace(GL_CW);
   glutSolidTeapot(0.3);
   glFrontFace(GL_CCW);
   glPopMatrix();
}


void drawLandscapeTexture(void)
{
   int i, j;

   glDisable(GL_LIGHTING);
   glEnable(GL_TEXTURE_2D);

   glBlendFunc(GL_DST_COLOR, GL_ZERO);
   glEnable(GL_BLEND);
   glDepthFunc(GL_EQUAL);

   for(j=0; j<lSize-1; j++)
   {
      glBindTexture(GL_TEXTURE_2D, TO_FLOOR2);
      glColor3f(1.0, 1.0, 1.0);

      glBegin(GL_TRIANGLE_STRIP); // Start Drawing A Triangle
      glColor3f(1.0, 1.0, 1.0);

      for(i=0; i<lSize; i++)
      {
         glTexCoord2f((float)i/(lSize-1), (float)j/(lSize-1));
```

```
            glVertex3f(meshCoords[i][j][0], meshCoords[i][j][1],
            meshCoords[i][j][2]);

            glTexCoord2f((float)i/(lSize-1), ((float)j+1)/(lSize-1));
            glVertex3f(meshCoords[i][j+1][0], meshCoords[i][j+1][1],
            meshCoords[i][j+1][2]);
         }
         glEnd();
      }

      glDepthFunc(GL_LEQUAL);
      glDisable(GL_TEXTURE_2D);
      glEnable(GL_LIGHTING);
      glDisable(GL_BLEND);
   }


   void loadTextures(void)
   {
      int i, j;

      glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

      // Init mesh

      for(i=0; i<lSize; i++)
      {
         for(j=0; j<lSize; j++)
         {
            meshCoords[i][j][0] = ((GLdouble)i -
            ((GLdouble)(lSize-1))/2)*10/lSize;  // x
            meshCoords[i][j][2] = ((GLdouble)j -
            ((GLdouble)(lSize-1))/2)*10/lSize;  // y
            meshCoords[i][j][1] = (GLdouble) (-.5-myRand()*.25 +
            2*sin(((double)j*16/lSize)) * sin(((double)i*17/lSize)) *
            sin(((double)j*27/lSize)));    // z=height
         }
      }

      // Init my own image texture
      for(i=0; i<256; i++)
      {
         for(j=0; j<256; j++)
         {
            texImage[i][j][0] = (GLubyte) ((1.0 + sin((double)i/10)
            * sin((double)j/10))*120.0);
            texImage[i][j][1] = (GLubyte) ((1.0 + sin((double)i/8)
            * sin((double)j/11))*120.0);
            texImage[i][j][2] = (GLubyte) ((1.0 + sin((double)i/12)
            * sin((double)j/13))*120.0);
         }
      }

      glGenTextures(1, &texName);   // Create The Texture
      glBindTexture(GL_TEXTURE_2D, TO_FLOOR2);
      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 256, 256, 0,
      GL_RGB, GL_UNSIGNED_BYTE, texImage);
```

```
   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
}


int main(int argc, char **argv)
{
   glutInitWindowSize(800, 600);
   glutInit(&argc, argv);
   parseOptions(argc, argv);
   if (useStencil) {
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH |
      GLUT_STENCIL);
   } else {
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   }
   if (fullscreen) {
      glutGameModeString("800x600:32");
      glutEnterGameMode();
   } else {
      glutCreateWindow("Shadowmap test");
   }
   glutDisplayFunc(display);
   glutKeyboardFunc(keyboard);
   glutSpecialFunc(special);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
   glutMotionFunc(motion);
   glutVisibilityFunc(vis);

   initMenus();
   initExtensions();
   initGL();
   loadTextures();
   time(&ltime);

   glutMainLoop();
   return 0;
}
```

```
/*
-----------------------------------------------------------------------

opengldrawable.cpp

Adds 3D objects to a scene.

showShadowed():
Renders the scene with shadowmapped shadows.

-----------------------------------------------------------------------
*/

void CamOpenGLDrawable::showShadowed(void)
{
   glDisable(GL_CULL_FACE);
   glLightfv(GL_LIGHT0, GL_AMBIENT, lightDimColor);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glDisable(GL_LIGHT7);

   updateDepthMap();
   needDepthMapUpdate = 1;

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   setupEyeView();
   setView();

   glLightfv(GL_LIGHT0, GL_POSITION, lv);

   // Draw the scene with less illumination on the shadowed areas
   glLightfv(GL_LIGHT0, GL_SPECULAR, zero);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, lightDimColor);
   glLightfv(GL_LIGHT0, GL_AMBIENT, lightDimColor);

   glDepthFunc(GL_LEQUAL);
   glAlphaFunc(GL_GREATER, 0.9);
   glEnable(GL_ALPHA_TEST);
   glEnable(GL_TEXTURE_2D);
   glDisable(GL_LIGHTING);
   glColorMask(0,0,0,0);
   useTextures = 1;


   show();
   glColorMask(1,1,1,1);
   glEnable(GL_LIGHTING);
   useTextures = 0;
   glDisable(GL_TEXTURE_2D);
   glDepthFunc(GL_EQUAL);

   // Ambient light only!
   show();
```

```
    // Later passes must match depth values
    glDepthFunc(GL_EQUAL);

    // Depth map
    glActiveTextureARB(GL_TEXTURE0_ARB);
    configTexgen(0, 0); // 0,0

    // Light Z range
    glActiveTextureARB(GL_TEXTURE1_ARB);
    configTexgen(1, 0); // 1, 0

    glLightfv(GL_LIGHT0, GL_SPECULAR, lightColor);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, lightColor);

    configCombiners(1);
    if (hasRegisterCombiners) {
       glAlphaFunc(GL_GREATER, 0.0);
    } else {
       glAlphaFunc(GL_GREATER, 0.5);
    }
    glEnable(GL_ALPHA_TEST);

    show();

    if (depthMapPrecision == GL_UNSIGNED_SHORT) {
       // Re-write unshadowed regions
       configCombiners(2);
       show();
    }

    disableCombiners();

    glActiveTextureARB(GL_TEXTURE1_ARB);
    disableTexgen();
    glActiveTextureARB(GL_TEXTURE0_ARB);
    disableTexgen();

    useTextures = 1;
    glDisable(GL_LIGHTING);
    glEnable(GL_TEXTURE_2D);

    glBlendFunc(GL_DST_COLOR, GL_ZERO);
    glEnable(GL_BLEND);
    glDepthFunc(GL_EQUAL);

    show();

    glDepthFunc(GL_LEQUAL);
    glDisable(GL_BLEND);
    glDisable(GL_TEXTURE_2D);
    glEnable(GL_LIGHTING);
    useTextures = 0;

}
```

8 Ordlista

Ackumulationsbuffert	Se *Buffertar*
Aliasing	Olika sorters avrundningsartefakter eller kvantiseringsfel som uppkommer då något detaljerat fenomen ska representeras med en begränsad precision.
API	Ett API (applikationsprogrammeringsgränssnitt) är en specifikation för hur man kan använda, och kommunicera med, hårdvarufunktioner eller andra program. Vanligtvis tillhandahåller ett API olika funktioner som man kan anropa för att läsa av värden eller utföra specifika uppgifter.
Buffertar	En buffert är ett minnesutrymme för att lagra bilder eller annan information som gäller pixlarna i ritytan. En färgbuffert innehåller färginformation, medan en djupbuffert för varje pixel lagrar avståndet till objektet som pixeln tillhör. En stencil-buffert kan användas som schablonmall för att avgöra vilka delar av bilden som ska ritas ut. En ackumulationsbuffert kan användas för att kombinera flera bilder till en (och t ex skapa ett medelvärde av flera bilder).
Culling	Eliminiering av oönskade polygoner (ofta sådana polygoner som ändå inte syns, eftersom de är utanför synfältet eller döljs av andra polygoner).
Djupbuffert	Se *Buffertar*
Frame	Bildruta; en enskild bild som genereras av datorprogrammet.
Frustum	En tredimensionell region i form av en pyramid med avhuggen topp. Synfältet – de objekt man ser på skärmen – utgör normalt en sådan, och kallas då för vy-frustum.
Funktion	En funktion är en del av ett datorprogram som utför en specifik uppgift, och som kan anropas från andra delar av programmet (varefter den eventuellt returnerar ett värde).
Färgbuffert	Se *Buffertar*
GLUT	OpenGL Utilities Toolkit; en OpenGL-verktygslåda med funktioner för att hantera användargränssnitt, och skapa vissa geometriska former.

Konkav	Inåtbuktande. Ett objekt är konkavt om man kan dra en linje genom det som har mer än en en ingångs- och utgångspunkt. Exempel: En skål eller en torus.
Konvex	Utåtbuktande. Ett objekt är konvext om varje linje som kan dras genom det endast har en ingångs- och utgångspunkt. Exempel: En sfär.
Multitextur	Texturmappning med flera texturer samtidigt.
Normal	En normal är en linje som är vinkelrät mot en yta.
OpenGL	OpenGL är ett API, ett gränssnitt, med funktioner för att underlätta programmering av 2D- och 3D-grafik.
Ortogonal	Rätvinklig.
Pixel	Bildpunkt; den minsta punkt som kan ha en egen färg på bildskärmen (förkortning av ”picture element”)
Polyeder	En tredimensionell kropp vars yta begränsas av polygoner
Polygon	Månghörning (t ex en triangel eller en fyrhörning)
Projicerad textur	En textur som projiceras på scenen likt diabilder.
Rendera	Att generera/rita ut grafik
Rymdpartitionering	Att dela upp en rymd i delar, eller organisera objekten i det, t ex för att enklare kunna avgöra vilka polygoner som ska ritas ut. Ett exempel är BSP-träd, som använder en trädstruktur för att lagra vilka ytor som ligger framför respektive bakom andra ytor.
Scen	*Scenen* betecknar i denna rapport alla de polygonobjekt som ska visas.
Självskuggning	*Självskuggning* innebär att ett objekt kastar en skugga på sig själv (och inte bara på övriga objekt i scenen).
SpatialAce	Ett geografiskt informationssystem utvecklat av Carmenta AB.
Stencilbuffert	Se *buffertar*.
Tessellering	Att konvertera en polygon till mindre ytor, vanligtvis trianglar.
Tetraeder	En pyramid-formad polyeder bestående av fyra liksidiga trianglar
Texel	Bildpunkt i en textur (förkortning av TEXture

	ELement)
Texturmappning	En metod för att öka detaljnivån hos 3d-grafik genom att täcka polygoners ytor med bilder. Bilder som används på detta sätt kallas *texturer*.
Z-buffert	Annat namn för *djupbuffert*. Se *buffertar.*

9 Referensförteckning

[Blyt97] David Blythe, *Creating Shadows*, http://toolbox.sgi.com/TasteOfDT/documents/OpenGL/advanced97/node99.html, 1997

[Crow77] Franklin C. Crow, *Shadow Algorithms for Computer Graphics*, Computer Graphics (SIGGRAPH '77 Proceedings), vol. 11, no. 2 , 1997

[Heck96] Paul S. Heckbert, Michael Herf. *Fast Soft Shadows*, Visual Proceedings, SIGGRAPH 96, 1996

[Herf97] Michael Herf. *Efficient Generation of Soft Shadow Textures*, Carnegie Mellon U, 1997

[Heck97] Paul S. Heckbert, Michael Herf. *Simulating Soft Shadows With Graphics Hardware*, Carnegie Mellon U, 1997

[McCo98] Michael D. McCool, *Shadow Volume Reconstruction,* University of Waterloo*, 1998*

[Tess89] T. Tessman. *Casting Shadows on Flat Surfaces*. Iris Universe, Winter:16, 1989

[Slat92] Mel Slater, *A Comparison of Three Shadow Volume Algorithms*, The Visual Computer, 1992

[Heid99] Wolfgang Heidrich, *High-quality Shading and Lighting for Hardware-accelerated Rendering*, PhD thesis, University of Erlangen, Computer Graphics Group, 1999

[Will78] Lance Williams, *Casting Curved Shadows on Curved Surfaces,* Computer Graphics (SIGGRAPH ’78 Proceedings), August 1978

[Woo97] M. Woo, J. Neider, T.Davis, *OpenGL Programming Guide*, Second Edition, Addison-Wesley, 1997

[Zhan98] Hansong Zhang, *Forward Shadow Mapping*, Rendering Techniques '98, 1998

10 Bilaga A: kravspecifikation

Denna bilaga beskriver specifikationen av examensarbetet. Under samtal med min handledare och andra på Carmenta beslöts att mitt examensarbete skulle rikta in sig på skuggor. (Med ”dynamisk skuggning” nedan menas färgsättning som varierar beroende på vinkeln till en ljuskälla, inte att objekt kan kasta skuggor på andra objekt.)

Bakgrund

Ett av de områden Carmenta verkar inom är geografiska informationssystem. Här visualiseras geografi både i 2D och 3D.

Examensarbete

Ett tidigare utfört examensarbete har omfattat studier och testimplementationer av metoder som kan öka realismen vid 3D-visualiseringar av geografi.

De områden som studerats nära är:

1. Visualisering av naturfenomen som moln, dimma, regn och snö.
2. Texturer som täcker marken.
3. Generaliserade objekt på marken som träd och hus.
4. Dynamisk skuggning

Examensarbetet bör bygga vidare på de resultat som redan framkommit. Dels kan effektivare implementeringar studeras, men också andra typer av effekter bör undersökas. Exempel på andra effekter är:

- Solreflexer, speglingar
- Nya typer av generaliserade objekt
- Ljudeffekter
- Rörelseoskärpa
- Väderleksförhållanden

Examensarbetet bör komma fram till vilka metoder som är effektivast att använda för att öka realismen kontra kostnader i tid, pengar och beräkningskraft. Examensarbetet kan antingen inriktas mot visualisering/uppfattning eller algoritmer/implementation, och skall även undersöka metoder där man kan använda det hårdvarustöd som finns/kommer att finnas i en närliggande framtid.

Omfattning

Examensarbetet är lämpligt för 1-2 personer. Omfattning och inriktning kan i hög grad påverkas.

Tidsplan

Följande tre steg skall utföras under de tre månader som examensarbetet skall pågå:

Studie och sammanfattning av de metoder inom datorgrafik och GIS som idag finns. Tidsåtgång: ca 2 veckor.

2. Specifiering av vad som ska testimplementeras. Tidsåtgång: ca 1 vecka.

3. Testimplementering och utvärdering av de metoder som man har valt att inrikta sig på. Tidsåtgång: resten av tiden.