KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PRAKTINĖS INFORMATIKOS KATEDRA

Aras Pranckevičius

MINKŠTŲ ŠEŠĖLIŲ VAIZDAVIMAS REALIUOJU LAIKU

Magistro darbas

Vadovas
doc. dr. A. Lenkevičius

KAUNAS, 2005

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PRAKTINĖS INFORMATIKOS KATEDRA

TVIRTINU
Katedros vedėjas
doc. dr. D. Rubliauskas
2005 05

MINKŠTŲ ŠEŠĖLIŲ VAIZDAVIMAS REALIUOJU LAIKU

Informatikos mokslų magistro baigiamasis darbas

Kalbos konsultantė
Lietuvių k. katedros lekt.
dr. J. Mikelionienė
2005 05

Recenzentas
doc. dr. V. Jusas
2005 05

Vadovas
doc. dr. A. Lenkevičius
2005 05

Atliko
IFM 9/1 gr. stud.
A. Pranckevičius
2005 05

KAUNAS, 2005

KVALIFIKACINĖ KOMISIJA

Pirmininkas: Laimutis Telksnys, akademikas

Sekretorius: Stasys Maciulevičius, docentas

Nariai: Rimantas Barauskas, profesorius

Raimundas Jasinevičius, profesorius

Jonas Kazimieras Matickas, docentas

Jonas Mockus, akademikas

Rimantas Plėštys, docentas

Henrikas Pranevičius, profesorius

SUMMARY

Rendering Soft Shadows in Real-time

Shadows provide an important cue in computer graphics. In this thesis we focus on real-time soft shadow algorithms. Two new techniques are presented, both run entirely on modern graphics hardware.

Soft Shadows Using Precomputed Visibility Distance Functions renders fake soft shadows in static scenes using precomputed visibility information. The technique handles dynamic local light sources and contains special computation steps to generate smooth shadows from hard visibility functions. The resulting images are not physically accurate, nevertheless the method renders plausible images that imitate global illumination.

Soft Projected Shadows is a simple method for simulating natural shadow penumbra for projected grayscale shadow textures. Shadow blurring is performed entirely in image space and needs only a couple of special blurring passes on pixel shader 2.0 hardware. The technique treats shadow receivers as nearly planar surfaces and doesn't handle self shadowing, but executes very fast and renders plausible soft shadows. Multiple overlapping shadow casters in a single shadow map are natively supported without any performance overhead.

SANTRAUKA

Šešėlių vaizde apskaičiavimas yra svarbi kompiuterinės grafikos dalis. Šiame darbe pristatomi du nauji algoritmai šešėliams vaizduoti realiuoju laiku; abu algoritmai gali būti visiškai realizuojami šiuolaikiniuose trimačio vaizdo spartintuvuose.

Šešėlių imitacija naudojant matomo atstumo funkcijas imituoja minkštus šešėlius statinėse scenose, naudojant iš anksto apskaičiuotą matomumo informaciją. Algoritmas leidžia naudoti lokalius dinaminius šviesos šaltinius. Gaunami šešėliai nėra fiziškai teisingi, tačiau daugeliu atvejų atrodo pakankamai gerai ir imituoja globalaus apšvietimo metodais gaunamus rezultatus.

Minkšti projektuoti šešėliai – algoritmas, papildantis standartinį projektuotų šešėlių algoritmą pusšešėlio regionais. Minkštų šešėlių apskaičiavimas yra vien tik vaizdo operacija, nepriklauso nuo geometrinio vaizduojamos scenos sudėtingumo ir reikalauja tik poros vaizdo apdorojimo filtrų, kai naudojami 2.0 versijos taškų apskaičiavimo programas palaikantys trimačio vaizdo spartintuvai. Vaizdą priimantys objektai turi būti beveik plokšti ir algoritmas apskaičiuoja tik šešėlius ant šių objektų, tačiau reikalauja mažai skaičiavimo resursų ir gaunami gerai atrodantys minkšti šešėliai. Vienoje šešėlių tekstūroje gali būti vaizduojami keli šešėlį metantys objektai.

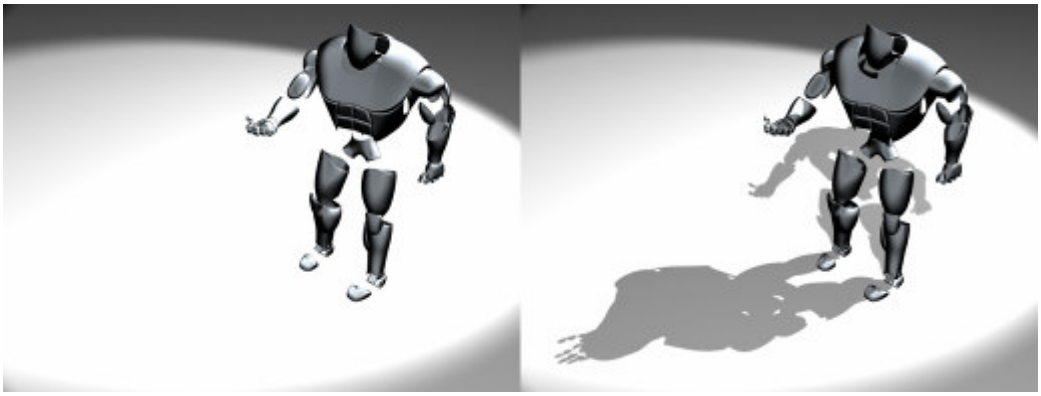
TURINYS

Lentelių sąrašas

Paveikslėlių sąrašas

ĮVADAS

Svarbi problema kompiuterinėje grafikoje, taip pat ir realaus laiko kompiuterinėje grafikoje, yra šešėlių vaizdavimas. Šešėliai vaizde leidžia geriau suvokti objektų tarpusavio išsidėstymą ir dydžius, šviesos šaltinių padėtis ir šešėlį priimančio objekto paviršių (1 pav.). Realiojo laiko kompiuterinė grafika smarkiai pažengė į priekį per pastaruosius keletą metų – jau įmanoma vaizduoti sudėtingos geometrijos modelius, paviršiaus mikronelygumus bei modeliuoti tikrų medžiagų atspindžio funkcijas; tačiau šviesos transporto ir šešėlių problema vis dar lieka svarbi.



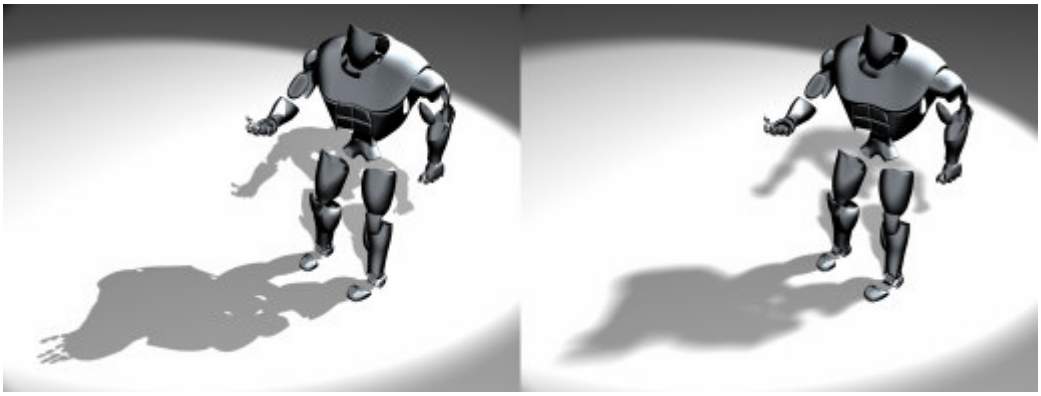
1 pav. Šešėliai vaizde padeda suvokti objektų tarpusavio padėtis

Pilnas šviesos transporto apskaičiavimas yra sudėtingas procesas, tradiciniai metodai (Monte Karlo spindulių trasavimas, energijos perdavimo modeliavimas ir kt.) nėra tinkami vaizduoti realiuoju laiku. Egzistuojantys greiti metodai (tūriniai šešėliai, šešėlių gylio tekstūros) vaizduoja šešėlius tik nuo tiesioginio apšvietimo bei turi kitų trūkumų. Neseniai pasiūlytas metodas pilnam šviesos transportui apskaičiuoti išsprendžia daugelį tradicinių problemų, tačiau vėlgi turi savų apribojimų (reikalauja statinių scenų bei netinkamas lokaliam apšvietimui).

Šiuo metu realiuoju laiku vaizduojami šešėliai dažniausiai apsiriboja *kietais* šešėliais *(hard shadows)* (2 pav.). Tokie šešėlių algoritmai šviesos šaltinius traktuoja kaip iš erdvės taško sklindančią šviesą ir nemodeliuoja šviesos šaltinio tūrio. Gaunami vaizdai pasižymi staigiais perėjimais tarp apšviestų ir neapšviestų paviršių – kiekvienas taškas arba yra apšviestas, arba yra šešėlyje. Tikrovėje taškiniai šviesos šaltiniai neegzistuoja; netgi saulė, turbūt labiausiai įprastas šviesos šaltinis, turi reikšmingą kampinį dydį ir nemeta kietų šešėlių.

Tikroviškesniu atveju, kai šviesos šaltinis turi baigtinį dydį, taškas ant šešėlį priimančio objekto gali „nematyti“ viso šviesos šaltinio (t. y. kažkuri šaltinio dalis yra užstota kitų paviršių). Tokiu atveju, taškas yra *pusšešėlio (penumbra)* regione; jis yra daugiau apšviestas nei visiško šešėlio regione esantis taškas.

Modeliuojant tokį šviesos šaltinį, gautame vaizde perėjimas tarp apšviestų ir neapšviestų paviršių nėra staigus ir šešėlis atrodo žymiai tikroviškiau (2 pav.).



2 pav. Kieti šešėliai (kairėje) ir minkšti šešėliai (dešinėje)

Realiojo laiko kompiuterinei grafikai šiuo metu praktiškai visada naudojami trimačio vaizdo spartintuvai. Pastaraisiais metais labai smarkiai išaugo tiek jų skaičiuojamoji galia, tiek ir programavimo galimybės, todėl vis daugiau kompiuterinės grafikos algoritmų yra pilnai realizuojami naudojant vien tik spartintuvą. Prognozuojama, kad trimačio vaizdo spartintuvų galia ir toliau didės greičiau nei kitų kompiuterio komponentų, todėl didesnį praktinį pritaikymą turi tie metodai ir algoritmai, kurie kuo efektyviau išnaudoja spartintuvų privalumus.

Šiame darbe pateikiami du nauji algoritmai vaizduoti minkštus šešėlius realiuoju laiku:

1. Minkštų šešėlių imitacija statinėse scenose, naudojant iš anksto apskaičiuotas matomo atstumo funkcijas. Naudojant šį algoritmą, imituojami globalaus apšvietimo procesu gaunami šešėliai.
2. Greitas minkštų šešėlių generavimo algoritmas, kai šešėlį priimantys objektai yra beveik plokšti. Šis algoritmas gaunamas papildant įprastą projektuotų šešėlių algoritmą specialiais vaizdo suliejimo filtrais; taip gaunama minkštų šešėlių aproksimacija.

Pateikiami algoritmai yra nepriklausomi, t. y. gali būti naudojami atskirai. Abu algoritmai gali būti pilnai realizuojami šiuolaikiniame trimačio vaizdo spartintuve, veikimo metu pagrindinis kompiuterio procesorius bei sisteminė atmintis yra visiškai nenaudojami.

Pirmasis pateikiamas algoritmas, „šešėlių imitacija naudojant matomo atstumo funkcijas“, buvo sukurtas 2003-2004 metais ir pirmą kartą pristatytas „Informacinės Technologijos 2004“ konferencijoje [32]; taip pat išspausdintas straipsnis knygoje *ShaderX3: Advanced Rendering with DirectX and OpenGL* [31]. Straipsnis pateikiamas 1 priede.

Antrasis pateikiamas algoritmas, „minkšti projektuoti šešėliai“, sukurtas 2005 metais ir naudojamas autoriaus darbe *Microsoft Imagine Cup 2005* kompiuterinės grafikos konkursui[1]. Taip pat priimtas straipsnis knygai *ShaderX4* [2].

Teorinėje darbo dalyje pateikiama trumpa vaizdo generavimo kompiuterinėje grafikoje apžvalga (apšvietimo modeliai, vaizdavimo lygtis, *BRDF*), toliau pristatomi ankstesni šešėlių generavimo realiuoju laiku srities darbai ir suformuluojamos kryptys ir reikalavimai naujai kuriamiems metodams ir algoritmams.

Tiriamojoje darbo dalyje atskiruose skyriuose aprašomi du sukurti algoritmai: „šešėlių imitacija naudojant matomo atstumo funkcijas“ bei „minkšti projektuoti šešėliai“. Kiekvienam iš jų pristatoma, idėja, detalus algoritmo aprašas, realizacijos detalės, pateikiami rezultatai, aptarimas bei palyginimai su egzistuojančiais algoritmais.

---

[1] Microsoft Imagine Cup 2005, Rendering Invitational: http://imagine.thespoke.net

[2] ShaderX^4: http://www.shaderx4.com . Straipsnis bus rašomas 2005 gegužę-birželį, numatoma knygos išleidimo data: 2005 gruodis.

TEORINĖ DALIS

1.1. Vaizdo generavimo kompiuterinėje grafikoje apžvalga

Kompiuterinės grafikos produktas yra galutinis kompiuteriu sugeneruotas vaizduojamos situacijos vaizdas. Vaizduojami gali būti įvairūs objektai ir įvairiais tikslais. Filmuose kompiuteriu generuojami vaizdai, kurių neįmanoma ar per brangu „gauti“ realybėje; tokie vaizdai turi būti kiek įmanoma tikroviškesni. Industrinėje vizualizacijoje generuojami vaizdai tam, kad būtų galima iš anksto pamatyti, kaip atrodys pagamintas daiktas ar pastatytas pastatas; tokie vaizdai taip pat turi būti tikroviški. Kompiuteriniame projektavime vaizdai turi aiškiai parodyti reikiamas objekto savybes (pvz., briaunas), ir nebūtinai turi atitikti realybę. Kompiuteriniuose žaidimuose vaizdas turi būti generuojamas realiuoju laiku, ir turi atitikti žaidimo meninį stilių; dažnai siekiama vaizdą generuoti kiek galima tikroviškesnį.

Visais atvejais, vaizdui sugeneruoti reikia nustatyti kur, kiek ir kokios šviesos patenka į virtualaus stebėtojo akis. Šios šviesos visuma ir yra „vaizdas“, paprastai jam sugeneruoti reikia:

1. Turėti vaizduojamos scenos aprašymą: objektų geometrinius aprašus, paviršiaus tekstūrų raštus, šviesos šaltinių aprašus ir kt.
2. Vaizduojamą sceną suprojektuoti į vaizdavimo plokštumą pagal perspektyvos dėsnius, atsižvelgiant į virtualaus stebėtojo padėtį ir orientaciją.
3. Apskaičiuoti, kaip šviesa patenka į stebėtoją: nuo kokių scenos objektų kiek ir kokios šviesos atsispindi.

Paskutinis žingsnis paprastai yra sudėtingiausias: realiame pasaulyje iš daugelio šviesos šaltinių išspinduliuojama daugybė fotonų, kurie įvairiai atsispindi nuo daugelio paviršių (priklausomai nuo paviršiaus medžiagos), ir visa tai įvyksta per labai trumpą laiką.

Šiame skyrelyje pateikiama trumpa teorinė apžvalga: pristatomi lokalūs ir globalūs apšvietimo modeliai, universali vaizdavimo lygtis ir medžiagos atspindžio funkcijos *BRDF* sąvoka. Taip pat pateikiami realiojo laiko kompiuterinėje grafikoje naudojamų apšvietimo modelių principai.

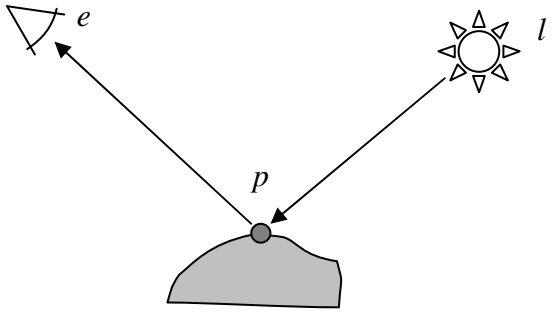
Apšvietimo modelis yra lygtys ir formulės, naudojamos taško (dažniausiai ant kažkokio objekto paviršiaus) spalvai apskaičiuoti. Šiuo metu sukurta daugybė apšvietimo modelių, kuriuos galima suskirstyti į dvi grupes: *lokalius* ir *globalius* modelius.

1.1.1. Lokalūs apšvietimo modeliai

Lokaliuose apšvietimo modeliuose taško spalva apskaičiuojama įvertinant tik taško poziciją erdvėje, paviršiaus medžiagos savybes ir scenoje esančių šviesos šaltinių savybes. Tai reiškia, kad jokie kiti scenos

taškai nėra traktuojami kaip sugeriantys ar atspindintys šviesą. Aišku, kad tai yra grubi aproksimacija (pvz., jei tarp taško ir šviesos šaltinio yra kitas nepermatomas objektas, tai nėra įvertinama), tačiau lokalūs modeliai dažnai naudojami kompiuterinėje grafikoje, nes reikalingi skaičiavimai yra nesudėtingi ir norint atvaizduoti kažkurį scenos tašką, reikia įvertinti tik šviesos šaltinius.

Lokaliam apšvietimo modeliui aprašyti imkime tašką *p*, kuris yra apšviečiamas iš kito taško *l* (t. y. šviesos šaltinio). Jei stebėtojas yra taške *e*, tai mus domina, kiek iš *l* atėjusios šviesos atsispindi nuo *p* link taško *e* (1.1 pav).



1.1 pav. Lokalaus apšvietimo modelio situacija

Taškas *p* negali atspindėti daugiau šviesos, nei kad patenka į jį, ir kadangi neegzistuoja neigiama šviesa, tai atspindėtas šviesos kiekis yra susijęs su ateinančiu šviesos kiekiu koeficientu intervale [0; 1]. Šis koeficientas priklauso nuo objekto, kurio paviršiuje yra *p*, medžiagos atspindžio funkcijos *BRDF (Bidirectional Reflectance Distribution Function)* [21, 46, 14], formuluojamos kaip nuo trijų taškų priklausanti funkcija:

$$L(\vec{e},\vec{p}) = L(\vec{p},\vec{l}) \cdot BRDF(\vec{e},\vec{p},\vec{l}) \qquad \textbf{(1.1)}$$

Funkcijos rezultatas priklauso nuo to, kaip vaizduojamos spalvos. Įprastame raudonos-žalios-mėlynos (RGB) modelyje spalva vaizduojama kaip trijų komponenčių vektorius, šiuo atveju *BRDF* reikšmė irgi yra trimatis vektorius, kurio kiekviena komponentė yra [0; 1] intervale. Šviesos kiekis, sklindantis į bet kokį tašką *a* iš bet kurio taško *b*, $L(\vec{a},\vec{b})$, tokiu atveju irgi yra trimatis vektorius, o daugyba atliekama atitinkamoms vektorių komponentėms atskirai.

BRDF galima suformuluoti ir kitaip[3] – vietoj trijų taškų funkcijos argumentuose naudoti vieną tašką ir du vektorius, nurodančius ateinančios ir atspindimos šviesos kryptis $\vec{\omega}_i$ ir $\vec{\omega}_o$:

$$L_o(\vec{p},\vec{\omega}_o) = L_i(\vec{p},\vec{\omega}_i) \cdot BRDF(\vec{p},\vec{\omega}_i,\vec{\omega}_o), \qquad \textbf{(1.2)}$$

[3] Tai yra originali *BRDF* formuluotė [21].

BRDF yra labai bendras medžiagos atspindžio aprašas, ir kompiuterinėje grafikoje (ypač realaus laiko kompiuterinėje grafikoje) retai naudojamas; vietoj jo atspindėta šviesa apskaičiuojama paprastesniais modeliais (žr. 1.1.3 skyrelį).

1.1.2. Globalūs apšvietimo modeliai

Globalūs apšvietimo modeliai įvertina visą vaizduojamą scenos informaciją tam, kad apskaičiuoti apšvietimą. Tai reiškia, kad kiekvienas vaizduojamas objektas potencialiai gali daryti įtaką kito vaizduojamo objekto spalvai. Tokios įtakos pavyzdžiai: objektas gali užstoti šviesą, atspindėti gautą šviesą link kitų objektų arba skleisti šviesą pats. Tokiu modeliu gaunamas rezultatas dažnai vadinamas *globaliu apšvietimu*. Universalus globalaus apšvietimo modelis yra *vaizdavimo lygtis (rendering equation)* [21], nurodanti kiek šviesos ateina į tašką *e* iš taško *p*:

$$L(\vec{e},\vec{p}) = v(\vec{e},\vec{p})\left(L_e(\vec{e},\vec{p}) + \int_S BRDF(\vec{e},\vec{p},\vec{s})L(\vec{p},\vec{s})d\vec{s}\right) \qquad \textbf{(1.3)}$$

Funkcija $v(\vec{e},\vec{p})$ atspindi matomumą tarp taškų *e* ir *p* – jos reikšmė yra vienetas jei taškai „mato“ vienas kitą ir nulis priešingu atveju. L_e yra iki *e* ateinanti iš taško *p* skleidžiama šviesa, ji nelygi nuliui tik tada, kai *p* yra šviesos šaltinis. *S* yra visų scenos objektų visų paviršiaus taškų aibė.

Vaizdavimo lygtis teigia: iš taško *p* į tašką *e* šviesa nepatenka, jei taškai nėra vienas kitam matomi; priešingu atveju patenkanti šviesa yra *p* skleidžiamos ir atspindimos šviesos suma. Skleidžiama šviesa yra medžiagos savybė, jos įtaka taškui *e* priklauso nuo *e* ir *p* tarpusavio geometrinės padėties. Atspindima šviesa, kylanti dėl iš vieno taško į *p* ateinančios šviesos, aprašyta skyrelyje 1.1.1, todėl visa taško *p* atspindima šviesa yra visų scenos taškų indėlių suma (integralas).

Tiesiogiai vaizdavimo lygtis negali būti išspręsta (*L* yra abiejose lygties pusėse), tačiau lygtį galima performuluoti. Užrašykime (1.3) lygtį kompaktiška forma:

$$\begin{aligned} L &= vL_e + vTL \\ <Tf>(\vec{e},\vec{p}) &= \int_S BRDF(\vec{e},\vec{p},\vec{s})f(\vec{p},\vec{s})d\vec{s} \end{aligned} \qquad \textbf{(1.4)}$$

Rekursyviai perrašydami (1.4) gauname:

$$\begin{aligned} L &= vL_e + vTL \\ &= vL_e + vT(vL_e + vTL) \\ &= vL_e + vT(vL_e + vT(vL_e + vTL)) \\ &= \sum_{n=0}^{\infty} v(Tv)^n L_e \end{aligned} \qquad \textbf{(1.5)}$$

Intuityvi (1.5) interpretacija yra tokia: kiekvieną tašką apšviečia nuo visų kitų iš jo matomų taškų 0, 1, 2, 3, ... kartų atsispindėjusi šviesa. Iš čia aiškėja esminis skirtumas tarp lokalių ir globalių apšvietimo

modelių: lokalūs modeliai įvertina tik nulį bei vieną kartą atsispindėjusią šviesą (atitinkamai: stebėtoją tiesiai iš šviesos šaltinių bei atsispindėjusi nuo objektų pasiekianti šviesa), ir neįvertina šviesos atspindžių matomumo. Globalūs apšvietimo modeliai teoriškai apskaičiuoja *visus* šviesos atspindžius bei įvertina matomumus tarp taškų.

Aišku, kad bendru atveju pilnai apskaičiuoti norimos scenos apšvietimo neįmanoma, nes vaizdavimo lygtis reikalauja begalinio iteracijų skaičiaus. Tradiciškai kompiuterinėje grafikoje naudojamos įvairios euristikos bei aproksimacijos, leidžiančios vartotojui pasirinkti tarp vaizdo kokybės ir vaizdo generavimo greičio.

1.1.3. Apšvietimo modeliai realaus laiko kompiuterinėje grafikoje

Realaus laiko kompiuterinėje grafikoje pagrindinis apribojimas yra vaizdo generavimo greitis. Paprastai „realaus laiko grafika“ reiškia, kad norimą vaizdą su turima aparatine bei programine įranga galima apskaičiuoti per mažiau nei 1/30 sekundės. Dažnai vieną vaizdą sudaro milijonas ar daugiau atskirų taškų, ir kiekvieno jų spalvą reikia apskaičiuoti.

Naudojant įprastus šiuolaikinius kompiuterius ir trimačio vaizdo spartintuvus, šiuo metu realaus laiko grafikoje dažniausiai naudojami lokalūs apšvietimo modeliai su paprastomis analitinėmis *BRDF* išraiškomis, bei kartais papildomai apskaičiuojami tiesioginiai šešėliai.

Dažniausiai naudojamos *BRDF* yra difuzinė (Lamberto) – kai paviršius ateinančią šviesą atspindi vienodai visomis kryptimis, idealaus veidrodžio *(specular)* – kai ateinanti šviesa atspindima vien tik veidrodiškai, ir *Phong* atspindžio modelis [30], aproksimuojantis blizgių *(glossy)* medžiagų *BRDF*.

1.2. Ankstesni darbai

Šiame skyriuje pateikiama trumpa šešėlių apskaičiavimo realiuoju laiku metodų ir algoritmų apžvalga. Išsamesnę modernių metodų apžvalgą galima rasti literatūroje (apžvalgai žr. [17, 35]).

1.2.1. Statinis apšvietimas

Jei vaizduojama scena ir šviesos šaltinių konfigūracija yra statinė (nejudanti), tai visų paviršių apšvietimą galima apskaičiuoti iš anksto, naudojant tradicinius metodus, aproksimuojančius vaizdavimo lygtį (1.3) (dažniausiai naudojami metodai: Monte Karlo spindulių trasavimas, energijos perdavimo modeliavimas *(radiosity)*, fotonų skleidimas *(photon mapping)* ir kt.). Šį apšvietimo skaičiavimą reikia atlikti tik vieną kartą; vaizduojant realiuoju laiku tiesiog naudojami išsaugoti skaičiavimo rezultatai. Apšvietimo reikšmės paprastai saugomos geometrijos viršūnėse *(per-vertex lighting)* arba paviršius padengiančiose apšvietimo tekstūrose *(lightmaps)*.

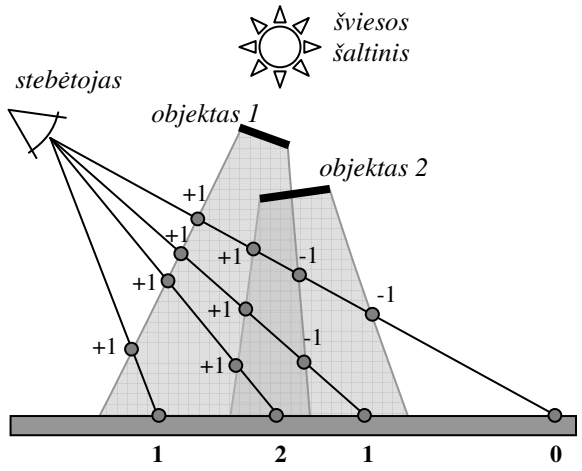
Tokio metodo privalumai: apskaičiuojamas apšvietimas nuo bet kokios šviesos šaltinių konfigūracijos, galima naudoti globalų apšvietimo modelį, vaizdavimas yra labai greitas ir gaunami vaizdai labai tikroviški.

Metodo trūkumai: galima modeliuoti tik Lamberto tipo medžiagų *BRDF* (t. y. tokias medžiagas, kurios atspindi šviesą visomis kryptimis vienodai); nei scena, nei šviesos šaltinių konfigūracija niekaip negali keistis.

Egzistuoja įvairios šio metodo modifikacijos, leidžiančios iš anksto apskaičiuoti difuzinį apšvietimą ir papildomai vaizduoti blizgius atspindžius [25], arba saugančios scenos apšvietimą nuo kiekvieno šviesos šaltinio atskirai, taip leidžiant keisti šviesos šaltinių spalvas ir/arba intensyvumą [12].

1.2.2. Tūriniai šešėliai

Geometrinis metodas šešėliams rasti, pasiūlytas Crow [10] ir vėliau pritaikytas trimačio vaizdo spartintuvams [19]. Randami šešėlius metančių objektų siluetai (šviesos šaltinio kryptimi) ir išstumiami *(extrude)* tolyn nuo šviesos šaltinio – taip suformuojamas šešėlio tūris *(shadow volume)*. Visų objektų dalys, patenkančios į šešėlio tūrį, natūraliai yra šešėlyje. Vaizdavimo metu kiekvienam taškui yra suskaičiuojama, kiek kartų spindulys tarp taško ir stebėtojo kerta šešėlio tūrius; jei nors vieno šešėlio pilnai nekerta, vadinasi, taškas yra šešėlyje (1.2 pav.). Trimačio vaizdo spartintuvuose šis skaičiavimas atliekamas naudojant kaukės *(stencil)* buferį [23].



1.2 pav. Tūrinių šešėlių apskaičiavimo iliustracija

Tūriniai šešėliai turi nemažai privalumų:

- natūraliai leidžia naudoti bekrypčius (taškinius) šviesos šaltinius,
- gaunami tiesioginio apšvietimo šešėliai vaizdavimo taško tikslumu,

- leidžia objektams mesti šešėlį ant savęs paties *(self-shadowing)*.

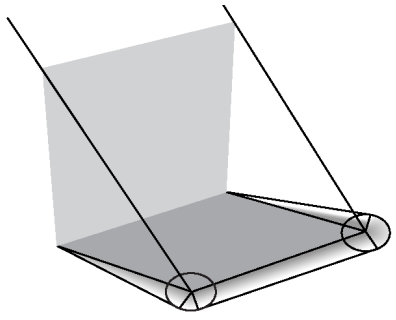
Taip pat ir keletą trūkumų:

- šešėlio formavimo laikas priklauso nuo šešėlį metančio objekto geometrinio sudėtingumo (tačiau egzistuoja algoritmai, leidžiantys naudoti supaprastintos geometrijos objektus [47]),
- šešėlio tūrio vaizdavimas sunaudoja labai daug vaizdo spartintuvo užpildymo gebos *(fillrate)*,
- neleidžia vaizduoti objektų, kurių forma aprašomas ne vien tik geometriškai (pvz., skyles objekto paviršiuje aprašo speciali tekstūra),
- bazinėje algoritmo versijoje gaunami kieti šešėliai atrodo nenatūraliai.

1.2.2.1. Tūrinių šešėlių išplėtimai minkštiems šešėliams generuoti

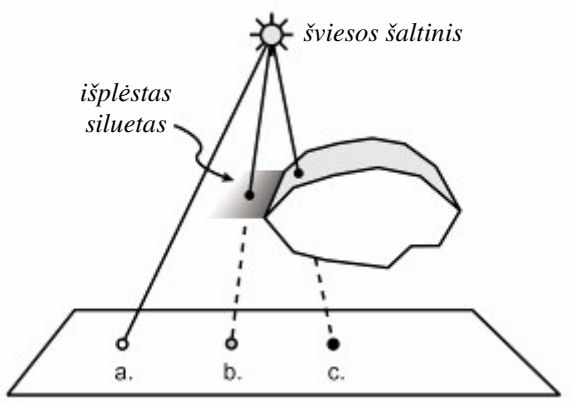
Paprasčiausias metodas gauti minkštus šešėlius naudojant tūrinių šešėlių algoritmą yra traktuoti tūrinį šviesos šaltinį kaip daugelį taškinių šviesos šaltinių. Apskaičiuojami šešėliai nuo kiekvieno taškinio šviesos šaltinio, galutinis vaizdas yra šių atskirų vaizdų vidurkis. Deja, šis metodas nelabai tinka realiojo laiko kompiuterinei grafikai, nes norint gauti gerus rezultatus kiekvieną šviesos šaltinį reikia pakeisti keliomis dešimtimis ar net keliais šimtais taškinių šaltinių; dėl to ir taip dideli tūrinių šešėlių užpildymo gebos poreikiai išauga dar kelias dešimtis kartų.

Pirmas efektyvus būdas minkštiems šešėliams apskaičiuoti pasiūlytas Haines [16]. Visiško šešėlio regionas apskaičiuojamas tradiciniu metodu, tada šešėlio siluetams sukonstruojami specialūs tūriai: viršūnėms sukonstruojami kūgio pavidalo tūriai, silueto kraštinėms sukonstruojami viršūnių kūgius jungiantys tūriai (1.3 pav.). Šešėlių siluetai randami, tūriai bei papildomi tūriai apskaičiuojami pagrindiniame kompiuterio procesoriuje. Algoritmas leidžia naudoti tik plokščius šešėlį priimančius objektus ir generuoja tik išorinį pusšešėlio regioną.

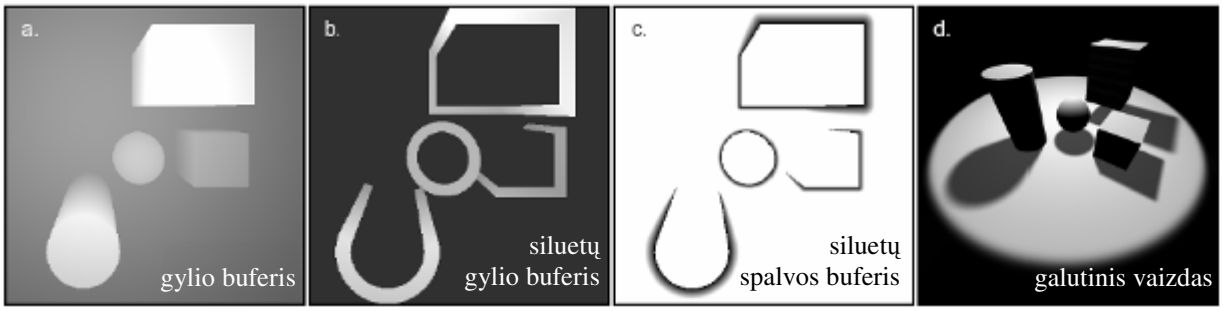


1.3 pav. Šešėlio tūrio papildymas išoriniam pusšešėliui gauti

Kitas geometrinis algoritmas, taip pat generuojantis tik išorinį pusšešėlio regioną, tačiau leidžiantis bet kokios konfigūracijos šešėlį priimančius objektus, pasiūlytas Chan ir Durand [9]. Pirma randamas šešėlį metančio objekto siluetas, kuris išplečiamas statmenai objekto paviršiui (1.4 pav.). Taip pat apskaičiuojama šešėlių gylio tekstūra ir išplėstas objekto siluetas aptekstūruojamas priklausomai nuo atstumų tarp šviesos šaltinio, silueto ir šešėlį priimančio objekto. Vaizdavimo metu pilno šešėlio regionas apskaičiuojamas pagal gylio tekstūrą, o išorinis pusšešėlio regionas gaunamas projektuojant tekstūruotą šešėlių siluetą ant priimančių paviršių (1.5 pav.). Šis išplėstų siluetų algoritmas naudoja objektų geometrinį aprašą tik siluetams rasti; likusi algoritmo dalis paremta šešėlių gylio tekstūromis (žr. 1.2.3 skyrių).



1.4 pav. Išplėsti siluetai išoriniam pusšešėliui apskaičiuoti



1.5 pav. Išplėstų siluetų algoritmas

Dar vieną geometrinį algoritmą pristatė Akenine-Möller ir Assarsson [2, 6, 5]. Kiekvienai šešėlį metančio objekto silueto kraštinei sukonstruojamas taip vadinamas „pusšešėlio pleištas“ *(penumbra wedge)* – figūra, pilnai apimanti šio silueto sukeliamą pusšešėlio tūrį. Standartiniu tūrinių šešėlių metodu apskaičiuojamas kietas šešėlis, tada vaizduojami sukonstruoti pleištai; pusšešėlis apskaičiuojamas naudojant šiuolaikinių trimačio vaizdo spartintuvų taškų apskaičiavimo programas *(pixel shaders)*. Šios programos kiekvienam pleišto taškui apskaičiuoja, kiek šviesos šaltinio yra matoma iš šio taško. Algoritmo autoriai realizavo keletą programų: sferiniams, tekstūruotiems stačiakampiams bei vienspalviams stačiakampiams šviesos šaltiniams. Algoritmas apskaičiuoja tiek išorinį, tiek vidinį

pusšešėlio regionus. Pagrindinis šio algoritmo trūkumas – kaip ir kituose tūrinių šešėlių algoritmuose, reikia rasti objektų siluetus, vaizdavimas reikalauja didelės užpildymo gebos.

1.2.3. Šešėlių gylio tekstūros

Vaizdu paremtas metodas šešėliams rasti, pasiūlytas Williams [45]. Į specialią *gylio tekstūrą (shadow map)* atvaizduojami visi objektai taip, kaip jie matomi iš šviesos šaltinio (1.6 pav.). Reikšmės tekstūros taškuose indikuoja atstumą nuo šviesos šaltinio iki artimiausio paviršiaus (kai kuriose modifikacijose saugomas atstumas, pakeltas kvadratu). Skaičiuojant galutinį vaizdą, gylio tekstūra projektuojama ant visų vaizduojamų objektų ir kiekvienam taškui lyginamas atstumas iki šviesos šaltinio su atstumu, gautu iš gylio tekstūros. Jei gylio tekstūroje atstumas mažesnis – reiškia, tarp taško ir šviesos šaltinio yra kitas nepermatomas paviršius, taigi vaizduojamas taškas yra šešėlyje. Daugelis šiuolaikinių trimačio vaizdo spartintuvų turi gylio tekstūrų formavimo funkcijas [13, 36].

1.6 pav. Šešėlių gylio tekstūra

Gylio tekstūrų privalumai:

- leidžia naudoti bet kokius objektus (nebūtinai vien gaunamus iš geometrinio aprašo),
- šešėlio formavimo laikas beveik nepriklauso nuo objektų geometrinio sudėtingumo,
- leidžia objektams mesti šešėlį ant paties savęs *(self-shadowing)*,

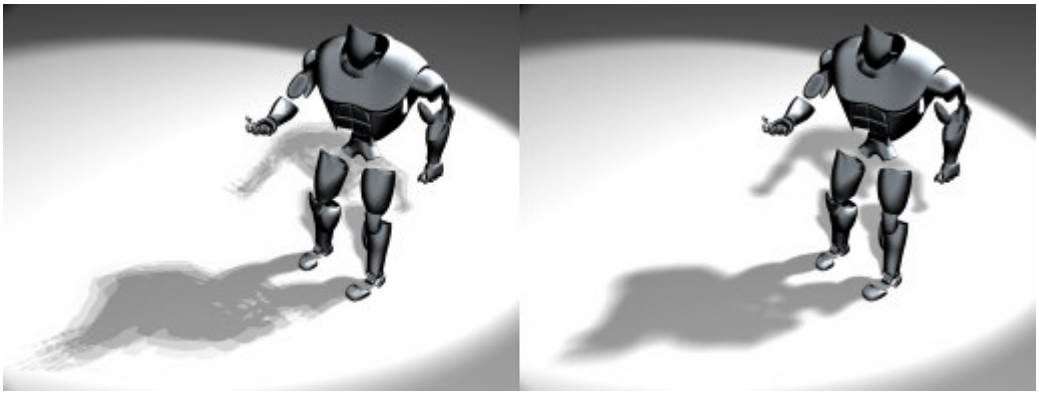
Gylio tekstūrų trūkumai:

- dėl vaizdo diskretizavimo į tekstūrą reikia imtis specialių priemonių, norint išvengti šešėliavimo klaidų,
- didėjant atstumui nuo šviesos šaltinio, prasideda šešėlių siluetų klaidos, kylančios dėl mažėjančio slankaus kablelio skaičių tikslumo bei perspektyvos,
- norint naudoti bekrypčius *(omnidirectional)* arba plataus kampo šviesos šaltinius, reikia naudoti keletą šešėlių tekstūrų (pvz., bekrypčiams šaltiniams dažnai naudojamos kubinės tekstūros, todėl vaizdavimo laikas ir reikiamos atminties kiekis išauga apie 6 kartus).

Egzistuoja įvairūs bazinio šešėlių gylio tekstūrų algoritmo patobulinimai: procentiškai artimesnis filtravimas *(percentage closer filtering)* sumažina diskretizavimo klaidas [34, 8]; perspektyvą įvertinančios gylio tekstūros *(perspective shadow maps)* [41] sumažina šešėlių klaidas, kylančias dėl perspektyvinės projekcijos.

1.2.3.1. Gylio tekstūrų išplėtimai minkštiems šešėliams generuoti

Paprasčiausias metodas gauti minkštus šešėlius naudojant šešėlių gylio tekstūras yra pakeisti tūrinį šviesos šaltinį į daugelį taškinių šviesos šaltinių [18]. Apskaičiuojami šešėliai nuo kiekvieno taškinio šviesos šaltinio, galutinis vaizdas yra šių atskirų vaizdų vidurkis. Deja, kaip ir panašus metodas tūrinių šešėlių atveju, šis taip pat nelabai tinka realiojo laiko kompiuterinei grafikai, nes norint gauti gerus rezultatus kiekvieną šviesos šaltinį reikia pakeisti keliais šimtais taškinių šaltinių (1.7 pav.).



1.7 pav. Šviesos šaltinio pakeitimas: 4 šaltiniais (kairėje) ir 1024 šaltiniais (dešinėje)

To paties metodo modifikacijos leidžia naudoti mažiau taškinių šviesos šaltinių ir taiko vaizdo apdorojimo operacijas pusšešėliams rasti [20]; saugo atstumus iki keleto artimiausių paviršių keliose gylio tekstūrose [1, 22] ir kt.

Vaizdų sąsūkos operacija paremtas minkštų šešėlių algoritmas pasiūlytas Soler ir Sillion [40]: kai šviesos šaltinis, šešėlį metantis ir šešėlį priimantis objektai visi yra lygiagrečiose plokštumose, tai gautas minkštas šešėlis yra metančio objekto ir šviesos šaltinio vaizdų sąsūkos rezultatas (1.8 pav.). Bendru atveju, kai objektai nėra lygiagrečiose plokštumose, autoriai siūlo rekursyviai dalinti šešėlius priimančių objektų aibę pagal leistinos kokybės įverčius. Šio algoritmo trūkumai: dėl priimančių objektų aibės dalijimo didėja vaizdavimui reikalingas laikas, sunku algoritmą taikyti bendru atveju (ypač kai scenoje yra ilgi šviesos krypčiai lygiagretūs paviršiai).

1.8 pav. Šviesos šaltinio (kairėje) sąsūka su objekto vaizdu (centre) gautas šešėlis (dešinėje)

Paprastas algoritmas, labai greitai apskaičiuojantis minkštus šešėlius ribotose situacijose, pasiūlytas Mitchell [26]. Šis metodas šešėlių tekstūrai pritaiko kintamo pločio suliejimo filtrą; taip gaunama minkštus šešėlius imituojanti tekstūra. Autorius naudoja Puasono *(Poisson)* disko suliejimo filtrą, kurio branduolio plotis didėja vertikalia kryptimi šešėlio tekstūros erdvėje. Toks algoritmas tinka tik tuo atveju, kai šešėlio tekstūroje yra tik vienas šešėlį metantis objektas ir jis atvaizduotas taip, kad netoli šešėlį priimančio paviršiaus esančios dalys yra tekstūros apačioje; toliau esančios dalys yra tekstūros viršuje.

Arvo ir Westelhorm minkštus šešėlius vaizduoja visų objektų geometrijos kraštinėms sukonstruodami papildomą keturkampį [4]. Pradžioje sukonstruojami nulinio ploto keturkampiai; vaizdavimo metu trimačio vaizdo spartintuvo viršūnių apskaičiavimo programa *(vertex shader)* objekto siluetui priklausančius keturkampius išplečia ir atvaizduoja į specialias vidinio ir išorinio pusšešėlio tekstūras.

Arvo ir kt. taip pat pristatė modifikuotą vaizdo užpildymo algoritmą minkštiems šešėliams gauti [3]. Pirmiausia apskaičiuojami kieti šešėliai standartiniu gylio tekstūrų metodu. Gautame vaizde kraštinių aptikimo filtru randamos šešėlių kraštinės, tada vaizdas apdorojamas modifikuotu užpildymo *(flood fill)* filtru, kuris išplečia šešėlių kraštus ir apskaičiuoja jų intensyvumą pagal atstumus tarp šešėlį priimančio, metančio objektų ir šviesos šaltinio.

Šiuo metu perspektyviausia laikoma algoritmų šaka, minkštų šešėlių gavimas iš vienos tekstūros, aprašoma atskirame skyrelyje (1.2.3.2).

1.2.3.2. Minkšti šešėliai, gaunami iš vienos gylio tekstūros

Minkštų šešėlių apskaičiavimo algoritmą, kai naudojama tik viena gylio tekstūra, pasiūlė Parker [29], vėliau jis buvo modifikuotas trimačio vaizdo spartintuvams[7]. Šiuo algoritmu apskaičiuojami šešėliai nėra fiziškai teisingi, tačiau pakankamai gerai atrodo (labai dažnai kompiuterinės grafikos tikslas yra tiesiog gerai atrodantys vaizdai).

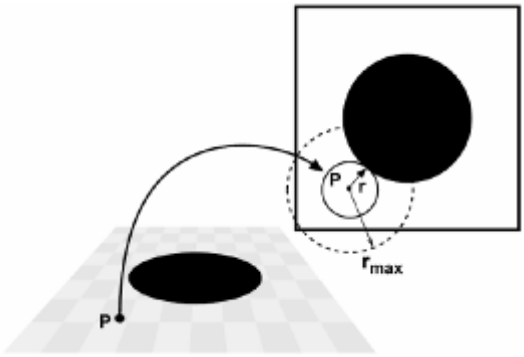
Pirmiausia apskaičiuojama standartinė gylio tekstūra, kaip kad būtų matoma iš šviesos šaltinio centro. Vaizdavimo metu kiekvienam taškui atliekama paieška gylio tekstūroje (1.9 pav.):

- jei vaizduojamas taškas yra šešėlyje, tai ieškomas artimiausias apšviestas taškas,

- jei vaizduojamas taškas yra apšviestas, tai ieškomas artimiausias taškas, kuris yra arčiau šviesos šaltinio.

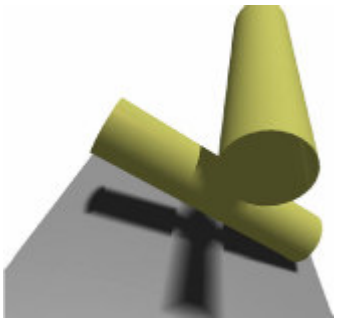
Šešėlio intensyvumas apskaičiuojamas pagal atstumus tarp vaizduojamo taško, rasto taško ir šviesos šaltinio. Gaunamo pusšešėlio regiono plotis priklauso nuo šių atstumų – gaunamas siauras pusšešėlis, kai šešėlį metantis ir priimantis paviršiai yra arti; ir platus pusšešėlis, kai jie yra toli vienas nuo kito.

Tam, kad artimiausias rastas taškas duotų teisingus rezultatus bendru atveju, autoriai papildomai naudoja objektų identifikavimo tekstūrą – tai reiškia, kad objektai negali mesti šešėlio patys ant savęs. Daugiausia skaičiavimų reikalaujanti algoritmo dalis – artimiausio taško paieška; dažnai paieškos spindulys ribojamas iki r_{max} (1.9 pav.), priklausančio nuo šviesos šaltinio dydžio ir atstumo iki apšviečiamų objektų. Kuo šis spindulys mažesnis, tuo greičiau veikia paieška.



1.9 pav. Artimiausio taško gylio tekstūroje paieška

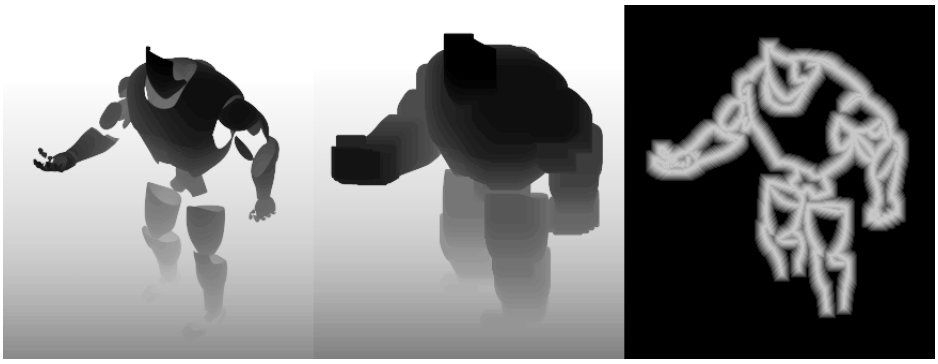
Greitesnis algoritmas pasiūlytas Kirsch [24] – kiekvienam šešėlio taškui randamas atstumas iki artimiausio apšviesto taško gylio tekstūroje. Vaizdavimo metu pagal atstumą iki apšviesto taško apskaičiuojamas vidinis pusšešėlio regionas (algoritmas negeneruoja išorinio pusšešėlio). Atstumai saugomi atskiroje *šešėlių pločio tekstūroje (shadow width map)*, kuri gaunama keletą kartų pritaikant specialų vaizdo apdorojimo filtrą. Pasiekiamas realiojo laiko grafikai pakankamas vaizdavimo greitis, tačiau galimos šešėlių vaizdavimo klaidos, esant tam tikroms scenos objektų konfigūracijoms (1.10 pav.).



1.10 pav. Šešėlių klaidos naudojant šešėlių pločio tekstūras

Valient ir de Boer naudoja kintamo pločio suliejimo filtrą minkštiems šešėliams apskaičiuoti [43]. Panašiai kaip ir Mitchell [26], autoriai naudoja Puasono disko suliejimo filtrą. Filtro branduolio plotis

priklauso nuo atstumų tarp šešėlį metančio, šešėlį priimančio paviršių ir šviesos šaltinio. Kadangi išorinio pusšešėlio regione standartinė gylio tekstūra neturi pakankamai informacijos filtro pločiui apskaičiuoti, reikalingas papildomas gylio tekstūros išplėtimo *(dilation)* ir kraštinių aptikimo žingsnis (1.11 pav.). Šis algoritmas gali būti efektyviai realizuojamas šiuolaikiniuose trimačio vaizdo spartintuvuose, tačiau pusšešėlio regionuose galimos nežymios klaidos bei šešėlio intensyvumo netolydumas (triukšmai *(noise)*, atsirandantys dėl Puasono filtro stochastinės prigimties).



1.11 pav. Gylio tekstūra išplečiama ir aptinkamos šešėlių kraštinės

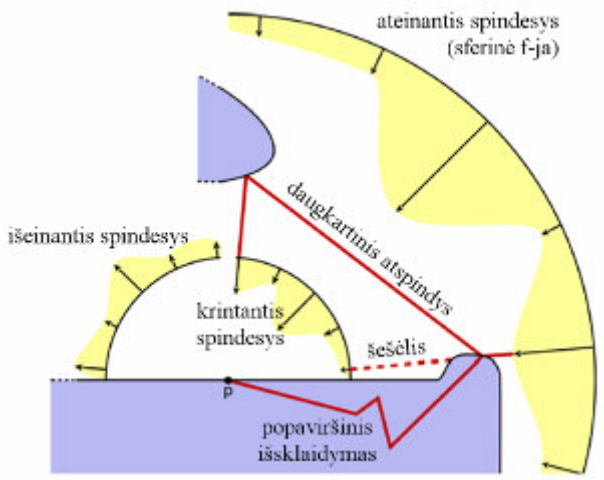
Vėliau šis algoritmas buvo išplėstas tam, kad apskaičiuotų pusšešėlių intensyvumas būtų tolydus *(smooth)* daugeliu atvejų [11]. Išorinio pusšešėlio regione šešėlio intensyvumas apskaičiuojamas iš išplėstų kraštinių tekstūros (1.11 pav. dešinėje), vidinio pusšešėlio regione papildomai naudojamas Puasono disko filtras kaip ir bazinio algoritmo atveju [43]. Galimos klaidos sumažinamos ir gaunamų pusšešėlių kokybė pagerinama papildomai naudojant *antro gylio šešėlių tekstūrą* [44] – taip tiksliau įvertinamas atstumas tarp šešėlį priimančio ir šešėlį metančio paviršių. Šiuo metu šis algoritmas laikomas vienu labiausiai tinkamų realiojo laiko kompiuterinei grafikai pagal generuojamų minkštų šešėlių kokybės ir reikiamo skaičiavimų kiekio santykį [35].

1.2.4. Apskaičiuotas šviesos perdavimas

Nauja šaka tarp realiojo laiko kompiuterinės grafikos metodų paremta iš anksto apskaičiuotomis šviesos spindesio perdavimo funkcijomis *(PRT – Precomputed Radiance Transfer)* [39]. Pagrindinė metodo idėja yra tokia: statinėse (nekintančiose) scenose galima iš anksto apskaičiuoti galimus šviesos transporto reiškinius (šviesos atsispindėjimą, sugėrimą, popaviršinį išsklaidymą *(subsurface scattering)* ir kt.), net iš anksto ir nežinant būsimos šviesos šaltinių konfigūracijos.

PRT metoduose daugelyje scenos taškų (pvz., geometrijos viršūnėse) apskaičiuojamos ir išsaugojamos funkcijos, ateinančią šviesą paverčiančios išeinančia šviesa. Pati funkcija modeliuoja daugelį šviesos perdavimo savybių: apšvietimo sumažėjimą kampu ateinančiai šviesai (*BRDF* Lamberto dalį), šešėlius dėl ribojamo matomumo, kai kuriais atvejais ir pilną medžiagos *BRDF* arba daugkartinius

šviesos atspindžius. Į kažkurį scenos paviršiaus tašką *p* krintantis spindesys *(incident radiance)* yra ateinantis spindesys, sumažintas dėl riboto matomumo ir padidintas dėl daugkartinių šviesos atspindžių (1.12 pav.). Nuo taško *p* išeinantis spindesys yra medžiagos *BRDF* ir krintančio spindesio sandauga, plius galimas popaviršinis šviesos išsklaidymas.



1.12 pav. Spindesio perdavimas taške *p*

Tiesioginio apšvietimo atveju šviesa, atsispindėjusi nuo taško *p* kryptimi *v*, išreiškiama tokiu integralu:

$$R(\vec{v}) = \int_S L(\vec{s})V(\vec{s})BRDF(\vec{v},\vec{s})d\vec{s} \qquad \textbf{(1.6)}$$

Formulėje (1.6) *R* – išeinančio spindesio funkcija, *L* – ateinančio spindesio f-ja, *V* – dvejetainė matomumo f-ja (1.13 pav.). Pagrindinė *PRT* metodo idėja yra tokia: *L* funkcija išreiškiama kokioje nors ortogonalioje bazėje:

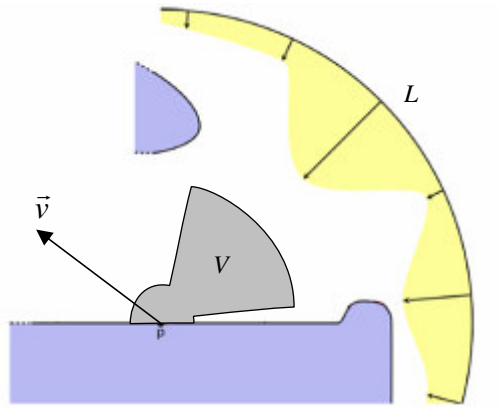
$$L(\vec{s}) \cong \sum_i l_i B_i(\vec{s}) \qquad \textbf{(1.7)}$$

Tuomet (1.6) galima pertvarkyti taip:

$$\begin{aligned} R(\vec{v}) &\cong \int_S \left(\sum_i l_i B_i(\vec{s}) \right) V(\vec{s})BRDF(\vec{v},\vec{s})d\vec{s} \\ &= \sum_i \left(l_i \int_S B_i(\vec{s})V(\vec{s})BRDF(\vec{v},\vec{s})d\vec{s} \right) \end{aligned} \qquad \textbf{(1.8)}$$

Formulėje (1.8) integralas nebepriklauso nuo ateinančios šviesos koeficientų, todėl jei *BRDF* yra Lamberto tipo arba *v* yra žinomas, integralo reikšmes galima iš anksto apskaičiuoti visiems *i*; tuomet gauname:

$$R(\vec{v}) = \sum_i l_i t_i \qquad \textbf{(1.9)}$$



1.13 pav. Nuo taško *p* atsispindėjusios šviesos apskaičiavimo schema

Dažniausiai naudojama žemo laipsnio sferinių harmonikų (SH) bazė, tačiau galima naudoti ir bet kokią kitą ortogonaliąją bazę (pvz., sferines vilneles [27]).

Lamberto tipo paviršiams paviršiaus taškuose saugomas SH vektorius *t*, vaizdavimo metu apšvietimas apskaičiuojamas iš skaliarinės sandaugos tarp saugomo vektoriaus ir ateinančios šviesos SH vektoriaus *l* (1.9 formulė). Geri rezultatai pasiekiami naudojant 4-6 eilės harmonikas (taigi 16-36 komponenčių SH vektorius).

Ne Lamberto tipo *BRDF* modeliuojamos sudėtingiau – paviršiaus taškuose saugoma SH matrica, vaizdavimo metu iš sąsūkos tarp saugomos matricos ir ateinančios šviesos SH vektoriaus *l* gaunamas išeinančios šviesos funkcijos SH vektorius. Gauta f-ja apskaičiuojama stebėtojo kryptimi $\vec{v}$ ir gaunamas galutinis apšvietimas. Naudojant 4-6 eilės harmonikas, paviršiaus taškuose reikia saugoti 256-1296 elementų matricas.

Abiem atvejais, kad sumažinti reikiamų saugoti duomenų kiekį ir paspartinti skaičiavimus, taikomas papildomas duomenų suspaudimas – visa saugomų funkcijų aibė traktuojama kaip daugiadimensinis signalas ir aproksimuojama klasterizuotu pagrindinių komponenčių analizės metodu (CPCA) [38].

PRT metodas leidžia apskaičiuoti sudėtingą šviesos perdavimo proceso rezultatą – įvertinamos sudėtingos medžiagų *BRDF*, gaunami minkšti šešėliai, daugkartiniai šviesos atspindžiai ir apšvietimas nuo pilnos sferinės šviesos funkcijos. Tačiau *PRT* turi ir trūkumų:

1. modeliuojamas tik begaliniai nutolęs apšvietimas,
2. reikalingi sudėtingi skaičiavimai ir didelė atminties apimtis, ypač nedifuzinių *BRDF* atveju,
3. kartu su *PRT* sunku panaudoti kitas vaizdavimo technologijas, pvz. mikronelygumų tekstūras *(bump maps)*.

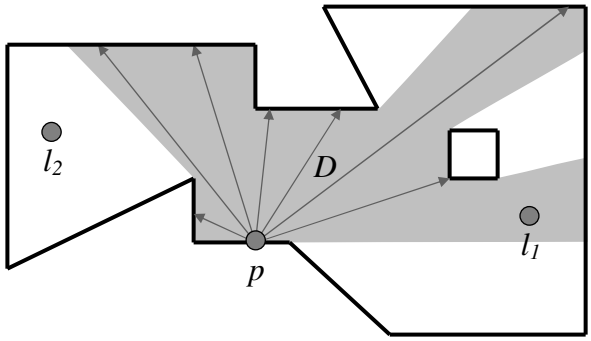
1.3. Teorinės dalies išvados ir idėjos naujiems algoritmams

1. Minkštų šešėlių vaizdavimas realiuoju laiku šiuo metu yra aktyvaus mokslinio tyrinėjimo sritis. Daugelis 1.2 skyriuje minėtų algoritmų ir metodų išrasti per pastaruosius porą metų ir vis dar yra tobulinami; nei vienas iš jų dar negali būti laikomas geriausiu.
2. *PRT* metodai yra daug žadantys, tačiau leidžia modeliuoti tik begaliniai nutolusią šviesą. Kai vaizduojamos išorės scenos, kur pagrindiniai šviesos šaltiniai paprastai būna saulė ir dangus, *PRT* metodai labai efektyvūs. Tačiau uždarose scenose šviesos šaltiniai būna arti scenos paviršių, todėl *PRT* pritaikyti sunku arba iš viso neįmanoma.
    - Šiame darbe pristatoma nauja idėja, iš dalies paremta *PRT* metodais – statinėse scenose galima iš anksto apskaičiuoti matomumo informaciją ir ją vėliau naudoti šešėliams generuoti (žr. 2.1 skyrių).
3. Dinaminėse scenose perspektyviausi atrodo šešėlių gylio tekstūromis paremti metodai ir nauji jų išplėtimai minkštiems šešėliams generuoti [3, 43, 11].
    - Šiame darbe pristatomas tarpinis algoritmas tarp paprastų Mitchell šešėlių [26] ir de Boer minkštų šešėlių algoritmo [11]. Algoritmas generuoja šešėlius tik ant beveik plokščių paviršių, tačiau reikalauja žymiai mažiau skaičiavimų nei de Boer algoritmas (žr. 2.2 skyrių).

2. TIRIAMOJI DALIS

2.1. Šešėlių vaizdavimas naudojant matomo atstumo funkcijas

Šio naujo algoritmo pagrindinė idėja tokia: jei turime statinę sceną (t. y. sceną, kurios objektų konfigūracija nesikeičia), tai daugelyje scenos taškų galime apskaičiuoti matomumo informaciją. *Matomo atstumo* funkcija *D* taškui *p* – hemisferinė funkcija, kurios reikšmė kryptimi *v* yra atstumas nuo *p* iki artimiausio nepermatomo paviršiaus (dvimatis pavyzdys 2.1 pav.). Jei kiekvienam scenos taškui turime matomo atstumo funkciją, tai vaizdavimo metu užtenka palyginti atstumą tarp *p* ir šviesos šaltinio *l* su $D\left(\vec{l} - \vec{p}\right)$ (atstumu tarp *p* ir artimiausio paviršiaus) tam, kad nuspręsti, ar taškas šešėlyje. Pavyzdžiui, 2.1 pav. atveju *p* yra apšviečiamas šviesos šaltinio l_1, tačiau yra šaltinio l_2 šešėlyje.



2.1 pav. Matomo atstumo funkcija *D* taškui *p*

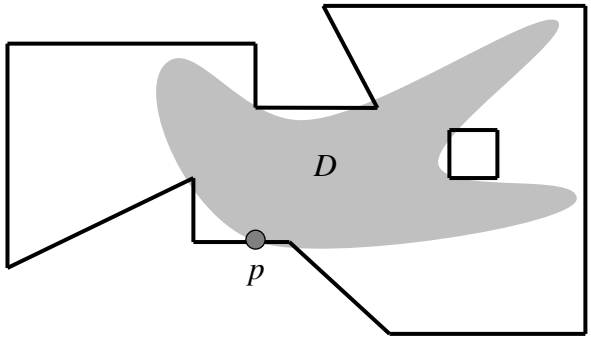
Deja, nėra praktiško būdo *tiksliai* išsaugoti matomo atstumo funkcijas *visiems* scenos taškams. Šiame darbe buvo pasirinkti du metodai reikiamam saugoti informacijos kiekiui sumažinti – matomo atstumo funkcijos saugomos daugelyje scenos taškų, maždaug tolygiai pasiskirsčiusių scenos paviršiuje, o pačios funkcijos aproksimuojamos žemos eilės sferinėmis harmonikomis (panašiai kaip *PRT* metoduose).

2.1.1. Matomo atstumo funkcijos

Šiame algoritme reikia saugoti matomo atstumo funkcijas daugelyje scenos paviršiaus taškų. Reikiamų taškų kiekis priklauso nuo scenos sudėtingumo ir norimos kokybės – kuo didesniame taškų skaičiuje saugomos funkcijos, tuo kokybiškesni šešėliai gaunami. Šiame darbe naudotoje scenoje, kurios dydis atitinka maždaug 19x19x6 metrus, matomo atstumo funkcijos saugotos kas 20 centimetrų nutolusiuose paviršiaus taškuose; iš viso reikėjo 54500 tūkstančių tokių taškų.

Jeigu scenai reikia kelių šimtų tūkstančių ar net kelių milijonų taškų matom atstumo funkcijoms saugoti, tai kiekvienai funkcijai saugoti galima skirti ne daugiau kaip keliasdešimt ar kelis šimtus baitų[4].

Sferinėms funkcijoms aproksimuoti itin gerai tinka sferinės harmonikos (SH) [15, 33, 39]. Dėl aproksimacijos prarandamos didelio dažnio dedamosios (SH atitinka Furjė transformaciją sferos paviršiuje) (2.2 pav), tačiau šis didelių dažnių praradimas ir leidžia šiame algoritme imituoti globalaus apšvietimo rezultatus (pvz., plonų objektų šešėliai pradeda nykti tolstant – toks pats rezultatas gaunamas ir globalaus apšvietimo atveju dėl daugkartinių šviesos atspindžių).



2.2 pav. SH aproksimuota matomo atstumo funkcija *D* taškui *p*

Šiame darbe buvo naudojamos antros, trečios ir penktos eilės sferinėmis harmonikomis aproksimuotos matomo atstumo funkcijos. Kiekvieną SH koeficientą saugant kaip slankaus kablelio skaičių, matomo atstumo funkcijoms saugoti reikėjo atitinkamai 16, 36 ir 100 baitų atminties (n-tosios eilės SH turi n^2 koeficientų).

Tarp taškų, kuriose saugomos matomo atstumo funkcijos, atitinkamų funkcijų koeficientai yra tiesiškai interpoliuojami. Jei vaizduojamas trikampis, kurio viršūnėse saugomos funkcijos D^a, D^b ir D^c, tai trikampio vidaus taške p, kurio baricentrinės koordinatės a, b, c, matomo atstumo funkcijos kiekvienas SH koeficientas D_i yra:

$$D_i = D_i^a a + D_i^b b + D_i^c c \qquad \textbf{(2.1)}$$

2.1.2. Matomumo apskaičiavimas ir saugojimas

Apskaičiuoti atstumo funkcijas ir jas aproksimuoti SH reikia tik vieną kartą – funkcijos priklauso tik nuo scenos geometrijos (kuri turi būti nekintanti). Kiekvienam taškui p matomo atstumo funkcijos SH koeficientai gaunami Monte Karlo integravimo būdu [15]:

[4] Jei matomo atstumo funkcijai saugoti reikia 100 baitų ir iš viso yra milijonas tokių funkcijų, tai reikiamos atminties kiekis yra apie 100 megabaitų. Pagal šiuolaikinio asmeninio kompiuterio parametrus būtų pageidautina matomumo informacijai paskirti ne daugiau keliolika megabaitų atminties.

1. Iš taško *p* trasuojama daug tolygiai sferos paviršiuje pasiskirsčiusių spindulių ir nustatomi artimiausi susidūrimai su kitais scenos paviršiais.
2. Atstumų reikšmės (kaip sferinė f-ja) projektuojamos į kiekvieną bazinę SH funkciją.

Greitam spindulių ir scenos susikirtimų tikrinimui naudojama mažai trianguliuota geometrija ir hierarchinis dengiančių gretasienių medis *(AABB tree)* [42]. Šiame darbe iš kiekvieno taško *p* buvo trasuojama 2500 spindulių; naudotai scenai matomo atstumo funkcijų skaičiavimas užtruko apie 6 minutes[5].

Atstumo funkcijos gali būti saugomos scenos trikampių viršūnėse arba tekstūrose. Pirmuoju atveju reikia tankaus trikampių tinklelio (realaus dydžio scenoms – viršūnės kas keliolika centimetrų), antruoju atveju reikia unikalaus scenos paviršiaus į plokštumą atvaizdavimo ir keleto tekstūrų SH koeficientams saugoti (paprastai tekstūros elementas yra keturmatis vektorius – taigi 5 eilės SH (25 koeficientai) reiktų 7 tekstūrų). SH koeficientai saugomi slankaus kablelio formatu (vienam koeficientui reikia 4 baitų atminties), tačiau galima juos saugoti ir mažesnio tikslumo duomenų tipuose (pvz., 2 baitų slankaus kablelio skaičiuose).

2.1.3. Vaizdavimo algoritmas

Vaizdavimas atliekamas kaip ir tradiciniuose metoduose: kiekvienam vaizduojamam taškui įvertinamas jo apšvietimas nuo kiekvieno šviesos šaltinio. Taško *p* apšvietimo nuo šaltinio *l* algoritmas:

1. Apskaičiuojama kryptis $\vec{v}$ ir atstumas d_L tarp taško ir šviesos šaltinio:

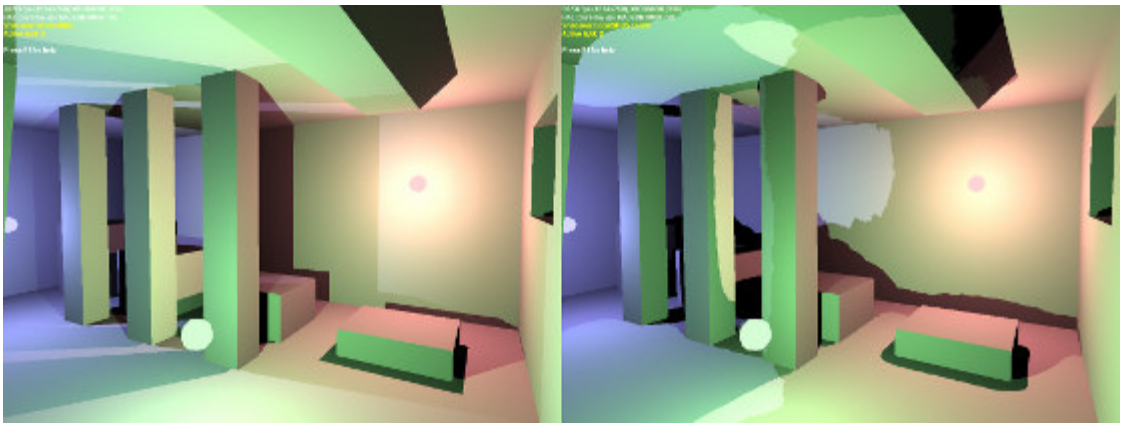
$$\vec{v} = \frac{\vec{l} - \vec{p}}{\left\|\vec{l} - \vec{p}\right\|}$$
$$d_L = \left\|\vec{l} - \vec{p}\right\| \qquad \textbf{(2.2)}$$

2. Apskaičiuojamas iš taško matomas atstumas $d_P = D(\vec{v})$. Funkcija *D* saugoma SH koeficientų pavidale, todėl $\vec{v}$ projektuojamas į SH bazines funkcijas [37] ir apskaičiuojama skaliarinė sandauga su *D* koeficientais.
3. Matomas atstumas d_P lyginamas su atstumu iki šviesos šaltinio d_L. Jei $d_P < d_L$, tai taškas yra šešėlyje.
4. Atliekami įprasti lokalaus apšvietimo modelio skaičiavimai (žr. 1.1.1 skyrių), galutinė spalva tamsinama priklausomai nuo 3 žingsnio rezultato.

[5] Naudoto kompiuterio parametrai: Intel Pentium4 3GHz procesorius, 1GB darbinės atminties (DDR 400MHz).

Šis algoritmas skirtas tik šešėliams apskaičiuoti, taigi galima naudoti bet kokią medžiagos *BRDF* ar papildomus tiesioginio apšvietimo metodus (pvz., modeliuoti mikronelygumus paviršiaus normalių tekstūromis).

Toks bazinis vaizdavimo algoritmas neduoda gerų rezultatų, nes dėl SH aproksimacijos matomo atstumo funkcijose prarandama daug aukšto dažnio detalių. Tipiškas rezultatas parodytas 2.3 pav. – kairėje šešėliai, gauti standartiniu gylio tekstūrų algoritmu; dešinėje matomo atstumo funkcijomis (naudojant 5-tos eilės SH aproksimaciją).



2.3 pav. Šešėlių gylio tekstūrų ir bazinio matomo atstumo funkcijų algoritmo rezultatų palyginimas

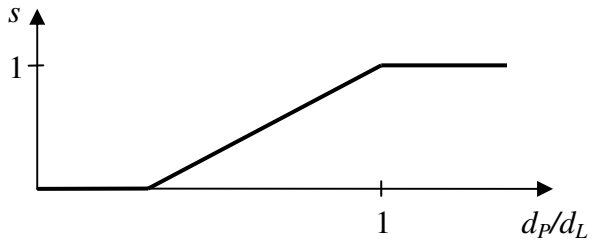
Baziniame vaizdavimo algoritme šešėlių intensyvumas s yra dvejetainis dydis:

$$\begin{aligned} s &= 1, \quad d_P \geq d_L \\ s &= 0, \quad d_P < d_L \end{aligned} \tag{2.3}$$

Galima pastebėti, kad globalaus apšvietimo metodais gaunami vaizdai dažnai yra šviesesni nei gaunami lokalaus apšvietimo metodais (daugiausia dėl daugkartinių šviesos atspindžių). Dėl to šiam algoritmui dvejetainį šešėlių intensyvumą pakeičiame sklandžiu perėjimu tarp visiško šešėlio ir visiškai apšviestų regionų:

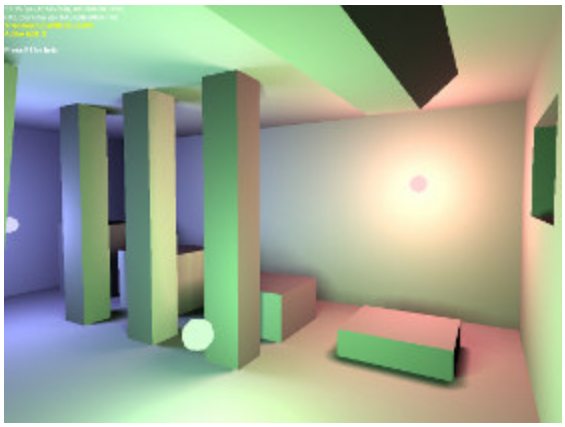
$$\begin{aligned} s &= saturate\left(\left(\frac{d_P}{d_L} - \frac{1}{3}\right) \cdot \frac{3}{2}\right) \\ saturate(x) &= \min(\max(x, 0), 1) \end{aligned} \tag{2.4}$$

Šios funkcijos grafikas parodytas 2.4 pav. Tokia funkcija pasirinkta tik todėl, kad pagal ją gaunami šešėliai atrodė geriausiai; galima naudoti ir bet kokias kitas funkcijas, kurių reikšmė lygi vienetui, kai $d_P \geq d_L$, ir reikšmė mažėja, kai d_P/d_L mažėja.



2.4 pav. Šešėlių intensyvumo funkcija

Naudojant tokią šešėlių intensyvumo funkciją, gaunami šešėliai *imituoja* globalaus apšvietimo metodais gautus rezultatus; jie nėra fiziškai teisingi, tačiau daugeliu atvejų atrodo pakankamai gerai (2.5 pav., naudota 5 eilės SH aproksimacija).



2.5 pav. Rezultatas, naudojant modifikuotą šešėlių intensyvumo f-ją

2.1.3.1. Vaizdavimo realizacija naudojant 5 eilės sferines harmonikas

Penktos eilės sferinės harmonikos turi 25 koeficientus; vaizdavimo metu reikia apskaičiuoti šiais koeficientais aprašytą funkciją šviesos šaltinio kryptimi. Pirmus devynis koeficientus (3 eilės SH) duota kryptimi galima efektyviai apskaičiuoti analitiškai [33, 37] – jei kryptis aprašoma vienetinio ilgio vektoriumi *(x,y,z)*, tai SH koeficientai *Y* šia kryptimi yra:

$$
\begin{aligned}
Y_0^0 &= \frac{1}{2\sqrt{\pi}} \\
(Y_1^1; Y_1^{-1}; Y_1^0) &= \frac{\sqrt{3}}{2\sqrt{\pi}}(-x; -y; z) \\
(Y_2^{-2}; Y_2^1; Y_2^{-1}) &= \frac{\sqrt{15}}{2\sqrt{\pi}}(xy; -xz; -yz) \\
Y_2^0 &= \frac{\sqrt{5}}{4\sqrt{\pi}}\left(3z^2 - 1\right) \\
Y_2^2 &= \frac{\sqrt{15}}{4\sqrt{\pi}}\left(x^2 - y^2\right)
\end{aligned}
\qquad \textbf{(2.5)}
$$

Pirmų devynių koeficientų reikšmės šiame darbe apskaičiuojamos trimačio vaizdo spartintuvo viršūnių apskaičiavimo programoje *(vertex shader)*, jų skaliarinė sandauga su pirmais 9 matomo atstumo funkcijos koeficientais perduodama į taškų apskaičiavimo programą *(pixel shader)*. Viršūnių apskaičiavimo programos tekstas *HLSL* programavimo kalba (*vs_1_1* profilis):

```
// viršūnės duomenys
struct VS_INPUT {
    float4 pos : POSITION;
    float3 normal : NORMAL;
    float4 shA : BLENDWEIGHT0; // pirmi 9 koeficientai
    float4 shB : BLENDWEIGHT1;
    float  shC : BLENDWEIGHT2;
    float4 shD : BLENDWEIGHT3; // likusieji 16 koeficientų
    float4 shE : BLENDWEIGHT4;
    float4 shF : BLENDWEIGHT5;
    float4 shG : BLENDWEIGHT6;
};

// duomenys, perduodami į taškų apskaičiavimo programą
struct VS_OUTPUT_25 {
    float4 pos   : POSITION;
    float  vdist : TEXCOORD0; // atstumas iš pirmų 9 koef.
    float4 sh[4] : TEXCOORD1; // likusieji 16 koeficientų
    float4 light : TEXCOORD5; // xyz - kryptis link šaltinio; w - atstumas
    float4 color : COLOR0;    // apskaičiuotas BRDF
};

float evalSH9( VS_INPUT i, float3 v ) {
    const float PI = 3.14159265;
    const float SPI = sqrt(PI);
    const float N0 = 1.0/2.0/SPI;
    const float N1 = sqrt(3)/2.0/SPI;
    const float N2 = sqrt(15)/2.0/SPI;
    const float N3 = sqrt(5)/4.0/SPI;
    const float N4 = sqrt(15)/4.0/SPI;
    float res;
    // pirmi 4 koef.
    res = dot( i.shA.wyzx,
        float4( float3(-N1,-N1,N1)*v, N0 ) );
    // sekantys 4 koef.
    float4 comb = v.xyzx * v.yzzz; // xy, yz, zz, xz
    float4 tmp2 = float4(N2, -N2, N3*3, -N2) * comb;
    tmp2.z -= N3;
    res += dot( tmp2, i.shB );
    // devintas koef.
    res += N4 * (v.x*v.x-v.y*v.y) * i.shC;
    return res;
}

// Viršūnių apskaičiavimo programa
VS_OUTPUT_25 vsMain25( VS_INPUT i ) {
    VS_OUTPUT_25 o;
    // transformuojam viršūnę, apskaičiuojam Lamberto BRDF
    o.pos = mul( i.pos, mViewProj );
    o.light = tolight( i.pos, vLightPos );
    o.color = lighting( i.normal, o.light, vLightColor );
    // atstumas iš pirmų 9 koef.
    o.vdist = evalSH9( i, o.light.xyz );
    // perduodam likusius 16 koef.
    o.sh[0] = i.shD; o.sh[1] = i.shE;
    o.sh[2] = i.shF; o.sh[3] = i.shG;
    return o;
}
```

Likusių 16 koeficientų analitinės išraiškos yra per daug sudėtingos, kad jas galima būtų apskaičiuoti realiuoju laiku kiekvienam vaizdo taškui, todėl jų bazinės SH funkcijos diskretizuojamos į kubines tekstūras. Naudojamos 64x64x6 dydžio Q8W8V8U8 formato[6] kubinės tekstūros; reikalingos keturios tokios tekstūros, nes viena tekstūra gali saugoti keturias SH bazines funkcijas. Paskutiniai 16 matomo atstumo funkcijos koeficientų iš viršūnių apskaičiavimo programos perduodami į taškų apskaičiavimo programą keturiuose 4D interpoliacijos registruose ir pilnai apskaičiuojamas atstumas d_P.

Taškų apskaičiavimo programos tekstas *HLSL* programavimo kalba (*ps_2_0* profilis; *smpSH0..smpSH3* yra bazinių funkcijų kubinės tekstūros):

```
float4 psMain25( VS_OUTPUT_25 i ) : COLOR
{
    float3 l = i.light.xyz;
    float dl = i.light.w, dv = i.vdist;
    dv += dot( texCUBE( smpSH0, l ), i.sh[0] );
    dv += dot( texCUBE( smpSH1, l ), i.sh[1] );
    dv += dot( texCUBE( smpSH2, l ), i.sh[2] );
    dv += dot( texCUBE( smpSH3, l ), i.sh[3] );
    // šešėlių intensyvumo funkcija
    float shadowFactor = saturate( (dv/dl-0.3333)*1.5 );
    //float shadowFactor = dv >= dl ? 1 : 0;
    return i.color * shadowFactor;
}
```

2.1.3.2. Vaizdavimo realizacija naudojant žemesnės eilės sferines harmonikas

Matomo atstumo funkcijas galima aproksimuoti ir žemesnės eilės sferinėmis harmonikomis, paaukojant dalį šešėlių kokybės dėl mažesnių skaičiavimo kiekio ir reikiamos atminties reikalavimų. Naudojant penktos eilės SH, vaizdavimo metu kiekvienam vaizdo taškui reikia perduoti 16 interpoliuotų koeficientų bei nuskaityti keturias kubines tekstūras; kiekvienoje viršūnėje reikia saugoti 25 SH koeficientus – tokie reikalavimai gali būti per dideli.

Jei naudojama trečios eilės SH aproksimacija, tai matomo atstumo funkcijos aprašomos devyniais koeficientais (reikiamas atminties kiekis nuo 100 baitų sumažėja iki 36 baitų) ir visi skaičiavimai gali būti atliekami viršūnių programoje. Tokiu atveju vaizdavimą galima atlikti ir senesniuose (DirectX 7/8 lygio) trimačio vaizdo spartintuvuose.

Viršūnių ir taškų apskaičiavimo programų tekstai, kai naudojama 3 eilės SH:

```
struct VS_OUTPUT_LO {
    float4 pos   : POSITION;
    float4 color : COLOR0; // apskaičiuotas BRDF
};
// Viršūnių apskaičiavimo programa
VS_OUTPUT_LO vsMain9( VS_INPUT i )
{
```

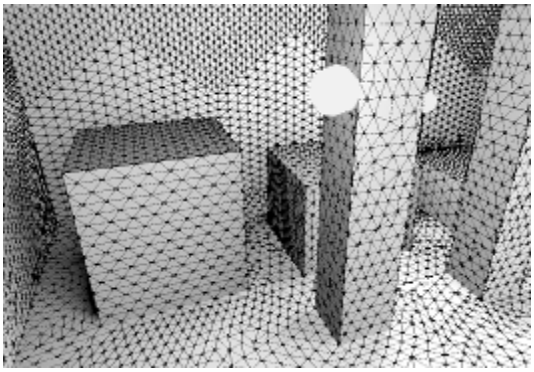
[6] Vienas tekstūros taškas sudarytas iš keturių 8 bitų komponenčių.

```
    VS_OUTPUT_LO o;
    o.pos = mul( i.pos, mViewProj );
    float4 light = tolight( i.pos, vLightPos );
    o.color = lighting( i.normal, light, vLightColor );
    // šešėlio apskaičiavimas
    float dv = evalSH9( i, light.xyz );
    float dl = light.w;
    float shadowFactor = saturate( (dv/dl-0.3333)*1.5 );
    o.color *= shadowFactor;
    return o;
}
// Taškų apskaičiavimo programa
float4 psMainLo( VS_OUTPUT_LO i ) : COLOR
{
    return i.color;
}
```

Galima aproksimuoti dar labiau ir naudoti antros eilės sferines harmonikas. Tokiu atveju matomo atstumo funkcijai saugoti tereikia 16 baitų, o viršūnių apdorojimo programa sutrumpėja iki keleto instrukcijų.

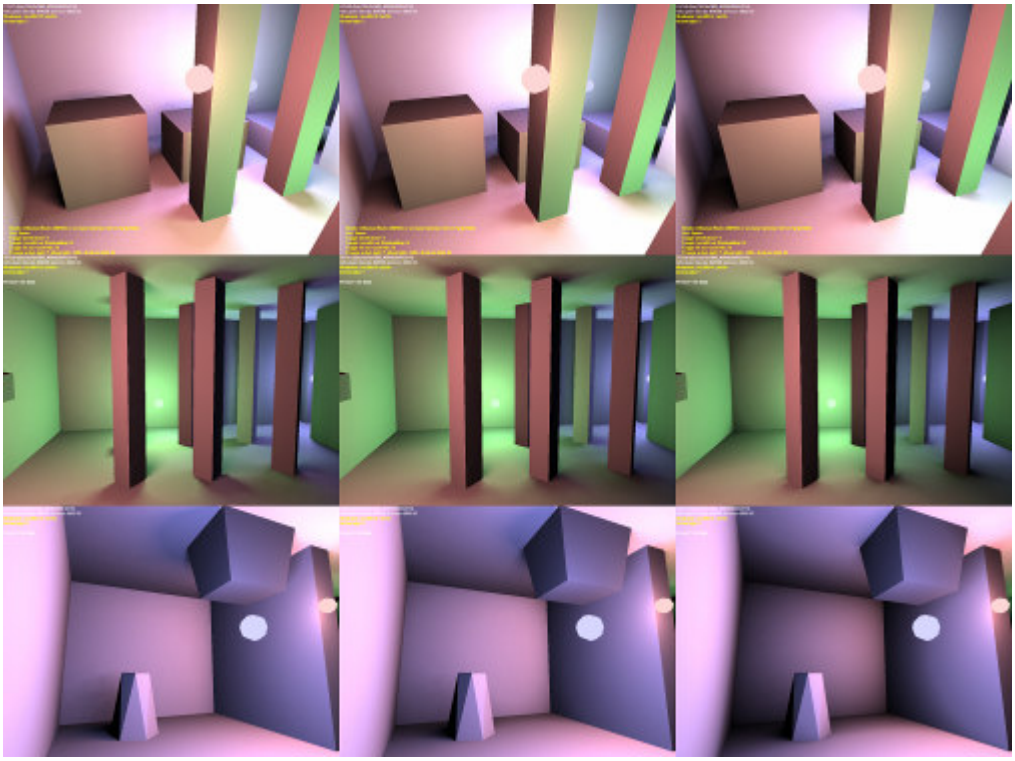
2.1.4. Rezultatai

Algoritmas realizuotas C++ programa, naudojant DirectX9.0 biblioteką trimačio vaizdo spartintuvui programuoti (programos apimtis – 5600 eilučių). Viršūnių ir taškų apskaičiavimo programos realizuotos *HLSL* programavimo kalba (apimtis – 500 eilučių); vaizdavimo metu visi skaičiavimai atliekami vien tik trimačio vaizdo spartintuve. Matomo atstumo funkcijos buvo saugomos scenos geometrijos viršūnėse maždaug kas 20 centimetrų (2.6 pav.), iš viso saugota 54500 funkcijų. Naudoti trys laisvai galintys judėti bekrypčiai šviesos šaltiniai.



2.6 pav. Matomo atstumo funkcijų saugojimo taškai

Gaunami šešėliai naudojant 5, 3 ir 2 eilės sferines harmonikas pateikiami 2.7 pav. (atitinkamai kairysis, centrinis ir dešinysis stulpeliai). Vaizdavimo spartos rezultatai pateikiami 2.1 lentelėje; pirma, antra ir trečia scenos atitinka 2.7 pav. viršutinę, vidurinę ir apatinę eilutes.



2.7 pav. Šešėliai, gaunami naudojant 5, 3 ir 2 eilės SH

Iš 2.7 pav. matosi, kad kuo žemesnės eilės SH aproksimacija naudojama, tuo labiau šešėliai praranda savo formą ir darosi sulieti. Didelio vizualinio skirtumo tarp penktos ir trečios eilės SH nesimato (tačiau skaičiavimai atliekami maždaug du kartus greičiau 3 eilės SH atveju); tuo tarpu su antros eilės SH gaunamų šešėlių kokybė jau yra prasta (nors skaičiavimai ir atliekami greičiausiai).

2.1 lentelė

Vaizdavimo sparta ir reikalingos atminties kiekis naudojant įvairios eilės SH

SH eilė	Trimačio vaizdo spartintuvas	Per sekundę apskaičiuojamų 1024x768 raiškos kadrų kiekis			Funkcijoms saugoti reikalinga atmintis, kB
		Scena 1	Scena 2	Scena 3	
5	GF6800[7]	111	102	62	5322
	R9800[8]	54	52	42	
3	GF6800	223	207	186	1916
	R9800	139	141	133	
2	GF6800	248	254	210	852
	R9800	159	161	152	

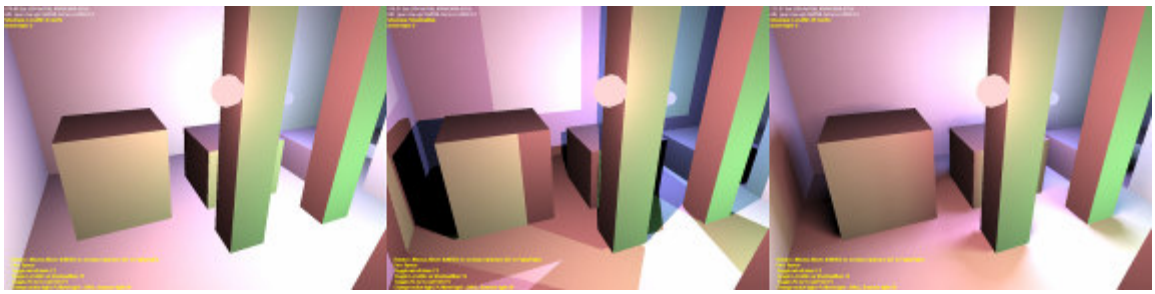
[7] nVidia GeForce 6800GT su 256MB vaizdo atminties

[8] ATI Radeon 9800Pro su 256MB vaizdo atminties

2.1.5. Algoritmo palyginimai ir aptarimas

Gaunamų rezultatų kokybės ir spartos palyginimo tikslais greta buvo realizuotas tradicinis šešėlių gylio tekstūrų algoritmas (naudojant kubines 512x512x6 dydžio R32F formato[9] tekstūras gyliui saugoti). Abu metodai skaičiavimus atlieka vien tik trimačio vaizdo spartintuve.

Gaunamų vaizdų palyginimas pateiktas 2.8 pav.: kairėje be šešėlių, centre gylio tekstūrų algoritmu gauti šešėliai, dešinėje matomo atstumo funkcijomis gauti šešėliai (naudojant 5 eilės SH). Matosi, kad gylio tekstūromis gaunami kieti šešėliai, o siūlomo algoritmo rezultatas labiau primena minkštus globalaus apšvietimo metodais gaunamus šešėlius.



2.8 pav. Vaizdas be šešėlių, su šešėlių gylio tekstūromis ir matomo atstumo f-jomis

Vaizdavimo spartos rezultatai pateikiami 2.2 lentelėje; pirma, antra ir trečia scenos atitinka 2.7 pav. viršutinę, vidurinę ir apatinę eilutes. Lyginant su šešėlių gylio tekstūromis, matomo atstumo funkcijų algoritmas yra 20-40% lėtesnis, kai naudojama 5 eilės SH aproksimacija; ir 40-50% greitesnis, kai naudojama 3 eilės SH aproksimacija.

2.2 lentelė

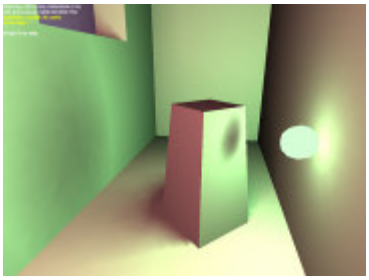
Vaizdavimo spartos palyginimas su gylio tekstūrų algoritmu

Algoritmas	Trimačio vaizdo spartintuvas	Per sekundę apskaičiuojamų 1024x768 raiškos kadrų kiekis		
		Scena 1	Scena 2	Scena 3
Atstumo f-jos, 5 eilės SH	GF6800	111	102	62
	R9800	54	52	42
Atstumo f-jos, 3 eilės SH	GF6800	223	207	186
	R9800	139	141	133
Šešėlių gylio tekstūros	GF6800	125	121	101
	R9800	79	77	74

Dėl žemos eilės SH aproksimacijos prarandama labai daug informacijos matomo atstumo funkcijose, tačiau gaunami vaizdai daugeliu atvejų atrodo pakankamai gerai. Kartais galimos šešėliavimo

[9] Vienas tekstūros taškas sudarytas iš vieno 32 bitų slankaus kablelio skaičiaus.

klaidos; dažniausiai kai šviesos šaltinis yra labai arti objekto paviršiaus (pvz., 2.9 pav. klaidinga šešėlio dėmė ant centre esančio objekto priekinės dešiniosios pusės).



2.9 pav. Galimos šešėlių klaidos

Matomo atstumo funkcijų algoritmą sunku lyginti su *PRT* metodais – *PRT* apskaičiuoja tikslesnius šešėlius (bei papildomus šviesos transporto reiškinius), tačiau netinkamas vidaus scenoms vaizduoti, kai šviesos šaltiniai yra objektų viduje. Matomo atstumo funkcijų algoritmas tik *imituoja* minkštus šešėlius, tačiau leidžia naudoti objektų viduje esančius šviesos šaltinius.

2.2. Minkšti projektuoti šešėliai

Autoriui kuriant darbą *Microsoft Imagine Cup 2005* kompiuterinės grafikos konkursui, prireikė greito algoritmo minkštiems šešėliams vaizduoti. Reikalavimai algoritmui buvo tokie: šešėliai metami nuo tam tikrų objektų ant beveik plokščių paviršių; nebūtina objektams mesti šešėlį ant savęs paties *(self shadowing)*; algoritmas turi būti paremtas šešėlių tekstūrų metodu ir nereikalauti daug skaičiavimo resursų. Reikėjo algoritmo, kuris būtų paprastesnis nei naujausi minkštų šešėlių algoritmai [3, 43, 11], tačiau būtų ne toks ribojantis kaip Mitchell paprastų minkštų šešėlių algoritmas [26].

Sukurtas algoritmas, apskaičiuojantis apytikslę minkštų šešėlių tekstūrą, kai atstumas tarp bet kokio scenos taško ir šešėlį priimančio objekto gali būti apytiksliai nustatytas analitiškai. Kai šešėlį priimantis objektas yra beveik plokščias (pvz., pastato grindys, žemės paviršius ir kt.), šis algoritmo reikalavimas tenkinamas. Algoritmas apskaičiuoja tik šešėlius ant šio priimančio objekto (t. y. visi kiti objektai tik meta šešėlius, tačiau tarpusavyje vienas kito nešešėliuoja), todėl šis metodas nėra tinkamas šešėliams apskaičiuoti bendru atveju. Kita vertus, šešėliams generuoti reikia tik poros vaizdo apdorojimo filtrų, todėl algoritmas veikia žymiai greičiau nei bendram atvejui tinkami minkštų šešėlių algoritmai.

Algoritmas yra šiek tiek sudėtingesnis nei Mitchell šešėlių algoritmas [26], tačiau leidžia į vieną šešėlių tekstūrą atvaizduoti bet kiek šešėlius metančių objektų (Mitchell algoritmui reikia naudoti atskiras tekstūras kiekvienam šešėlį metančiam objektui) bei nereikalauja jų specialiai orientuoti šešėlių tekstūroje.

2.2.1. Algoritmo idėja

Algoritmo idėja yra tokia: žinant atstumą tarp šešėlį metančio ir šešėlį priimančio objekto, galima įprastą kietų šešėlių tekstūrą sulieti kintamo pločio suliejimo filtru minkštiems šešėliams gauti. Filtro plotis turėtų tiesiškai priklausyti nuo atstumo, tuomet gaunamas pusšešėlio regionas bus platesnis, kai atstumas tarp metančio ir priimančio objektų bus didesnis.

Šį atstumą galima atvaizduoti specialioje tekstūroje tiesiog vaizduojant šešėlį metančius objektus, kiekvienam taškui apskaičiuojant atstumą iki šešėlį priimančio objekto. Tam, kad perėjimas tarp visiško šešėlio ir visiškai apšviestų vietų būtų tolydus, reikia papildomai išplėsti *(dilate)* gautą atstumo tekstūrą; tokiu principu ir veikia kai kurie minkštų šešėlių algoritmai [43, 11]. Šis išplėtimo žingsnis reikalauja nemažai skaičiavimų, nes didelio pločio vaizdo išplėtimo filtrai turi būti apskaičiuojami per keliolika iteracijų (viena iteracija išplečia vaizdą vienu tašku).

Pristatomame algoritme vaizdo išplėtimo operacija pakeičiama vaizdo suliejimo operacija – tai nėra ekvivalentus pakeitimas, tačiau gaunami šešėliai atrodo pakankamai gerai. Daugelis vaizdo suliejimo filtrų gali būti efektyviai realizuoti net ir esant dideliems filtro branduolio pločiams. Šiame darbe naudojamas Gauso suliejimo filtras, realizuotas kaip dviejų atskiriamų vienmačių Gauso filtrų pora [28]. Taip keliolika išplėtimo operacijų pakeičiama dvejomis suliejimo operacijomis.

2.2.2. Vaizdavimo algoritmas

Algoritmo rezultatas – nespalvota šešėlių intensyvumo tekstūra, kurią galima projektuoti ant šešėlį priimančio objekto standartiniu būdu. Algoritmui tekstūroje reikalingos trys spalvinės komponentės; šiame darbe naudojama A8R8G8B8 formato tekstūra[10]. Pirmoje tekstūros komponentėje (R) algoritmas apskaičiuoja standartinio kieto šešėlio kaukę; antroje (G) atstumą nuo objekto iki šešėlį priimančio paviršiaus; trečioje (B) šešėlio intensyvumo sumažėjimo dėl atstumo faktorių.

Tekstūros generavimo algoritmas:

1. Visi šešėlių tekstūros taškai nustatomi į reikšmes R=1, G=0, B=1.
2. Gylio buferio *(Z buffer)* palyginimo operacija nustatoma į „daugiau“[11] ir buferis išvalomas su nuline reikšme (0).
3. Visi šešėlį metantys objektai vaizduojami į gylio tekstūrą, taip, kad:

---

[10] Šiuolaikiniai trimačio vaizdo spartintuvai dažniausiai palaiko tik 1, 2 arba 4 spalvinių komponenčių tekstūras, todėl šiam algoritmui naudojama 4 komponenčių tekstūra (8 bitai kiekvienai komponentei).

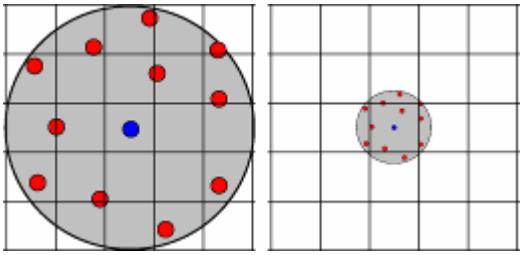
[11] Standartinė gylio buferio palyginimo operacija yra „mažiau“, t. y. galutiniame vaizde matosi artimiausi paviršiai. Nustačius į „daugiau“, gautame vaizde matysis tolimiausi paviršiai.

a. Šešėlio kaukės kanale įrašomas nulis (R=0).
b. Atstumo iki priimančio objekto kanale įrašomas apytikslis atstumas (G=[0;1]). Kai šešėlį priimantis objektas yra beveik plokščias, tai apytikslis atstumas gali būti tiesiog atstumas nuo objekto taško iki priimančio objekto plokštumos, padaugintas iš mastelio faktoriaus (kad tilptų į [0;1] intervalą).
c. Šešėlio intensyvumo faktoriaus kanale įrašomas skaičius, priklausantis nuo atstumo (B=[0;1]), vėliau šis intensyvumo faktorius naudojamas šešėlio spalvai nustatyti. Šiame darbe naudojamas B = G * 3 / 8 faktorius.

4. Gylio tekstūros atstumo kanalas (B) suliejamas *r* taškų pločio Gauso suliejimo filtru. Laisvai pasirenkama konstanta *r* apibrėžia maksimalų pusšešėlio regiono plotį. Šiame darbe naudojamas *r = 25*, suliejimas realizuojamas dvejais atskiriamais vienmačiais Gauso filtrais [28].
5. Apskaičiuojama galutinė šešėlių tekstūra, naudojant 12 taškų Puasono disko suliejimo filtrą (2.10 pav) [28]. Filtro branduolio plotis apskaičiuojamas iš 4 žingsnyje sulieto atstumo kanalo:

$$r_P = \lfloor B \cdot r/2 \rfloor \quad \textbf{(2.6)}$$

Šis žingsnis gali būti atliekamas ir tiesiogiai galutinio vaizdo formavimo metu.



2.10 pav. 12 taškų kintamo pločio Puasono suliejimo filtras

Algoritmo žingsnių rezultatai pavaizduoti 2.11 pav.: pirmos trys eilutės – trečiame žingsnyje gautos R, G, B kanalų spalvos, ketvirta eilutė – ketvirtame žingsnyje sulietas B kanalas, paskutinė eilutė – gauti minkšti šešėliai (paveikslėlyje 2-4 eilučių kontrastas padidintas).

2.11 pav. Algoritmo žingsnių rezultatai dvejose scenose

2.2.3. Algoritmo realizacija

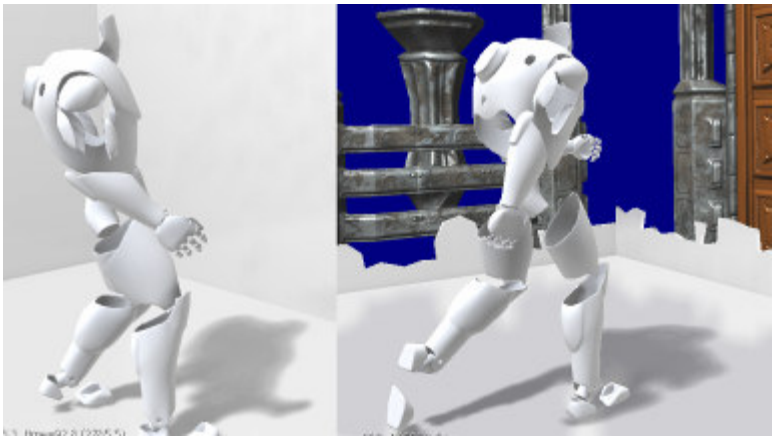
Algoritmas lengvai realizuojamas DirectX9 lygio trimačio vaizdo spartintuvuose. Šešėlį metančių objektų vaizdavimo (3 algoritmo žingsnio) skaičiavimai atliekami viršūnių apskaičiavimo programoje. Likusieji žingsniai (4 ir 5) realizuojami 2.0 versijos taškų apskaičiavimo programomis (Gauso filtras – dviem žingsniais (horizontalus ir vertikalus suliejimai), Puasono filtras – vienu žingsniu). Naudojamų viršūnių ir taškų programų tekstai *HLSL* kalba pateikti 2 priede.

Algoritmas realizuotas kaip *Imagine Cup 2005* konkursinio darbo dalis. Darbas realizuotas C++ programa, naudojant DirectX9.0 biblioteką trimačio vaizdo spartintuvui programuoti (apimtis – 54900 eilučių). Viršūnių ir taškų apskaičiavimo programos realizuotos *HLSL* programavimo kalba (apimtis – 1500 eilučių); vaizdavimo metu visi skaičiavimai atliekami vien tik trimačio vaizdo spartintuve.

2.2.4. Rezultatai

Realizacijoje naudojamos 256x256 taškų dydžio šešėlių tekstūros, vienas judantis objektas meta šešėlį ant stačiakampio gretasienio formos kambario sienų bei grindų. Algoritmo spartos analizei taip pat buvo naudojamos 512x512 ir 1024x1024 taškų dydžio šešėlių tekstūros.

Gaunami šešėliai pateikiami 2.12 pav. Vaizdavimo spartos rezultatai pateikiami 2.3 lentelėje[12]; pirma ir antra scenos atitinka 2.21 pav. kairįjį ir dešinįjį vaizdus.



2.12 pav. Algoritmu gaunami minkšti šešėliai

Nors gaunami šešėliai ir nėra fiziškai teisingi, tačiau atrodo pakankamai gerai bei turi minkštiems šešėliams būdingas savybes (pvz., pusšešėlio regionas platesnis, kai atstumas tarp šešėlį metančio ir priimančio objektų didesnis).

2.3 lentelė

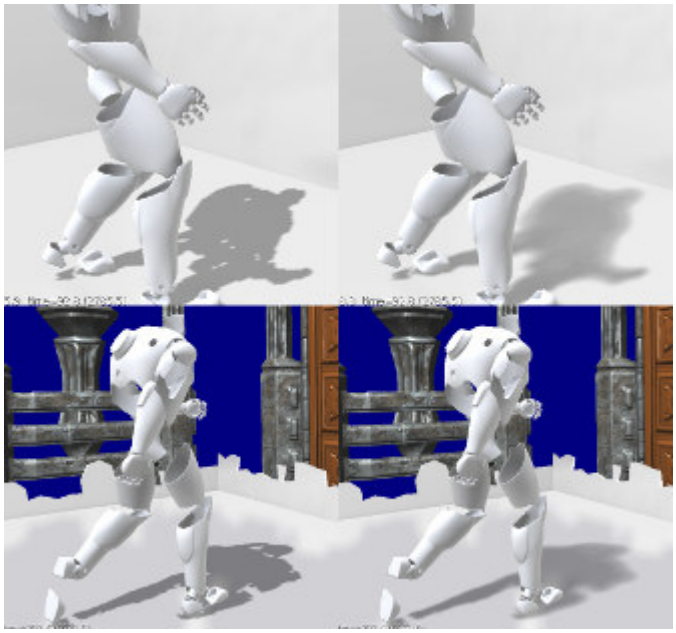
Minkštų projektuotų šešėlių algoritmo spartos rezultatai

Šešėlių tekstūros raiška	Per sekundę apskaičiuojamų 1024x768 raiškos kadrų kiekis	
	Scena 1	Scena 2
256x256	198	177
512x512	122	112
1024x1024	48	47

2.2.5. Algoritmo palyginimai ir aptarimas

Gaunamų rezultatų kokybės ir spartos palyginimo tikslais greta buvo realizuotas tradicinis projektuotų šešėlių algoritmas. Gaunami vaizdai pateikiami 2.13 pav.: kairėje įprastu metodu gauti šešėliai, dešinėje pristatomu algoritmu gauti minkšti šešėliai (abiem atvejais naudoja 256x256 raiškos šešėlių tekstūra). Matosi, kad minkšti šešėliai atrodo natūraliau bei paslepia diskretizavimo į mažos raiškos tekstūrą klaidas.

[12] Naudojant nVidia GeForce 6800GT trimačio vaizdo spartintuvą su 256MB vaizdo atminties.



2.13 pav. Kietų ir minkštų projektuotų šešėlių palyginimas

Vaizdavimo spartos palyginimas pateikiamas 2.4 lentelėje. Iš abiejų algoritmų rezultatų buvo apskaičiuotas laikas, sugaištas minkštiems šešėliams apskaičiuoti per vieną kadrą. Iš rezultatų matosi, kad šis laikas tiesiškai priklauso nuo šešėlių tekstūroje esančių taškų skaičiaus (keturis kartus padidinus taškų skaičių, laikas padidėja maždaug keturis kartus).

2.4 lentelė

Algoritmo spartos palyginimas su įprastu projektuotų šešėlių algoritmu

Šešėlių tekstūros raiška	Per sekundę apskaičiuojamų 1024x768 raiškos kadrų kiekis		Minkštiems šešėliams apskaičiuoti sugaištas laikas per kadrą, ms
	Kieti šešėliai	Minkšti šešėliai	
256x256	246	198	0,99
512x512	234	122	3,92
1024x1024	216	48	16,03

IŠVADOS

1. Darbe pristatyti du nauji algoritmai šešėliams vaizduoti. Abu algoritmai tinkami realiojo laiko kompiuterinei grafikai; visi skaičiavimai gali būti atliekami šiuolaikiniuose trimačio vaizdo spartintuvuose.

2. *Šešėlių imitacija naudojant matomo atstumo funkcijas* imituoja minkštus šešėlius statinėse scenose, naudojant iš anksto apskaičiuotą matomumo informaciją. Algoritmas imituoja globalaus apšvietimo metodais gaunamus šešėlius, kurie nėra fiziškai teisingi, tačiau daugeliu atvejų atrodo pakankamai gerai. Algoritmas leidžia naudoti lokalius dinaminius šviesos šaltinius ir veikia pakankamai greitai šiuolaikiniuose trimačio vaizdo spartintuvuose. Galimos greitesnės mažesnės kokybės šešėlius apskaičiuojančios variacijos.
3. Lyginant su standartiniu gylio tekstūrų algoritmu, *matomo atstumo funkcijų* algoritmas veikia 20-40% lėčiau, kai naudojama 5 eilės SH atstumo funkcijų aproksimacija; ir 40-50% greičiau, kai naudojama 3 eilės SH aproksimacija.

4. *Minkšti projektuoti šešėliai* papildo standartinį projektuotų šešėlių algoritmą pusšešėlio regionais. Algoritmas paprastai realizuojamas ir reikalauja mažai skaičiavimo resursų. Minkštų šešėlių apskaičiavimas yra vien tik vaizdo operacija, nepriklauso nuo geometrinio vaizduojamos scenos sudėtingumo ir reikalauja tik poros vaizdo apdorojimo filtrų, kai naudojami 2.0 versijos taškų apskaičiavimo programas palaikantys trimačio vaizdo spartintuvai. Vaizdą priimantys objektai turi būti beveik plokšti ir algoritmas apskaičiuoja tik šešėlius ant šių objektų. Vienoje šešėlių tekstūroje gali būti vaizduojami keli šešėlį metantys objektai. Algoritmas panaudotas autoriaus darbe *Imagine Cup 2005* kompiuterinės grafikos konkursui.
5. Minkštų šešėlių apskaičiavimas užtrunka apie 1 milisekundę, kai naudojamos 256x256 raiškos šešėlių tekstūros – taigi algoritmas ypač tinkamas realiojo laiko kompiuterinei grafikai. Didėjant tekstūros raiškai, skaičiavimo trukmė tiesiškai didėja priklausomai nuo taškų kiekio tekstūroje.

LITERATŪRA

1. **Agrawala M. ir kt.** Efficient Image Space Methods for Rendering Soft Shadows. *Computer Graphics / SIGGRAPH 2000: tarptautinės konferencijos pranešimų medžiaga.* 2000, 375-384 p.
2. **Akenine-Möller, T., Assarsson, U.** Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges. *Rendering Techniques 2002 (13th Eurographics Symposium on Rendering): tarptautinės konferencijos pranešimų medžiaga*. 2002, 297-306 p.
3. **Arvo, J. ir kt.** Approximate Soft Shadows with an Image-Space Flood-Fill Algorithm. *Rendering Techniques 2004 (15th Eurographics Symposium on Rendering): tarptautinės konferencijos pranešimų medžiaga*. 2004.
4. **Arvo, J., Westerholm, J.** Hardware Accelerated Soft Shadows using Penumbra Quads. *Journal of WSCG, 12(3)*. 2004.
5. **Assarsson, U. ir kt.** An Optimized Soft Shadow Volume Algorithm with Real-time Performance. *Graphics Hardware 2003: tarptautinės konferencijos pranešimų medžiaga*. 2003.
6. **Assarsson, U.**, **Akenine-Möller, T.** A Geometry-based Soft Shadow Volume Algorithm Using Graphics Hardware. *ACM Transactions on Graphics / SIGGRAPH 2003: tarptautinės konferencijos pranešimų medžiaga*. 2003.
7. **Brabec, S., Seidel, H. P.** Single Sample Soft Shadows Using Depth Maps. *Graphics Interface 2002: tarptautinės konferencijos pranešimų medžiaga*. 2002, 219-228 p.
8. **Bunnell, M., Pellacini, F.** Shadow Map Antialiasing. *GPU Gems: Programming Techniques, Tips and Tricks for Real-time Graphics.* Edited by R. Fernando. Boston, MA: Addison-Wesley, 2004, 185-192 p., ISBN 03221228324.
9. **Chan, E., Durand, F.** Rendering Fake Soft Shadows with Smoothies. *Rendering Techniques 2003 (14th Eurographics Symposium on Rendering): tarptautinės konferencijos pranešimų medžiaga*. 2003, 208-218 p.
10. **Crow, F. C.** Shadow Algorithms for Computer Graphics. *Computer Graphics / SIGGRAPH 1977: tarptautinės konferencijos pranešimų medžiaga*. 1977, 242-248 p.
11. **de Boer, W. H.** Smooth Penumbra Transitions with Shadow Maps. *Submitted to Journal of Graphics Tools.* 2004, 185-195 p.
12. **Dietrich, S.** Modern Graphics Engine Design. *Iš nVidia Developer Relations [interaktyvus]*. 2003 [žiūrėta 2005-05]. Prieiga per Internetą: http://developer.nvidia.com/object/modern_game_engine_design.html
13. **Everitt, C., Rege, A., Cebenoyan, C.** Hardware Shadow Mapping. *Iš nVidia Developer Relations [interaktyvus]*. 2001 [žiūrėta 2005-05]. Prieiga per Internetą: http://developer.nvidia.com/object/hwshadowmap_paper.html
14. **Fauerby, K., Kjær, C.** Real-time Soft Shadows in a Game Engine. *Magistro tezės, University of Aarhus, Department of Computer Science*. 2003 [žiūrėta 2005-05]. Prieiga per Internetą: http://www.peroxide.dk/papers/realtimesoftshadows.pdf
15. **Green, R.** Spherical Harmonic Lighting: the Gritty Details. *Game Developers Conference 2003: tarptautinės konferencijos pranešimų medžiaga*. 2003.
16. **Haines, E.** Soft Planar Shadows using Plateaus. *Journal of Graphics Tools, 6(1) tomas*. 2001, 19-27 p.
17. **Hasenfratz, J. M., Lapierre, M. et al.** A Survey of Real-time Soft Shadow Algorithms. *Eurographics 2003: tarptautinės konferencijos pranešimų medžiaga (State of the Art Report).* 2003 [žiūrėta 2005-05]. Prieiga per Internetą: http://www-imagis.imag.fr/Publications/2003/HLHS03
18. **Heckbert, P. S., Herf, M.** Simulating Soft Shadows with Graphics Hardware. *Techninė ataskaita CMU-CS-97-104, Carnegie Mellon University.* 1997.

19. **Heidmann, T.** Real Shadows, Real Time. *Iris Universe, 18 tomas, SGI.* 1991, 23-31 p.
20. **Heidrich, W.**, **Brabec, S., Seidel, H. P.** Soft Shadow Maps for Linear Lights High-quality. *Rendering Techniques 2000 (11th Eurographics Workshop on Rendering): tarptautinės konferencijos pranešimų medžiaga*. 2000, 269-280 p.
21. **Kajiya, J. T.** The Rendering Equation. *Computer Graphics and Interactive Techniques 1986: tarptautinės konferencijos pranešimų medžiaga*. 1986, 143-150 p.
22. **Keating, B., Max, N.** Shadow Penumbras for Complex Objects by Depth-dependent Filteting of Multilayer Depth Images. *Rendering Techniques 1999 (10th Eurographics Workshop on Rendering): tarptautinės konferencijos pranešimų medžiaga*. 1999, 205-220 p.
23. **Kilgard, M. J.** Improving Shadows and Reflections via the Stencil Buffer. *Iš nVidia Developer Relations [interaktyvus]*. 1999 [žiūrėta 2005-05]. Prieiga per Internetą: http://developer.nvidia.com/attach/6641
24. **Kirsch, F., Doellner, J.** Real-time Soft Shadows using a Single Light Sample. *Journal of WSCG, 11(1)*. 2003.
25. **Mc Taggart, G.** Half-Life 2 / Source Shading. *Game Developers Conference 2004: tarptautinės konferencijos pranešimų medžiaga.* 2004 [žiūrėta 2005-05]. Prieiga per Internetą: http://www.ati.com/developer/techpapers.html
26. **Mitchell, J. L.** Poisson Shadow Blur. *ShaderX^3: Advanced Rendering with DirectX and OpenGL.* Edited by W. Engel. Hingham, MA: Charles River Media, 2004, 403-410 p., ISBN 1-58450-357-2.
27. **Ng, R., Ramamoorthi, R., Hanrahan, P.** All-frequency Shadows using Non-linear Wavelet Lighting Approximation. *ACM Transactions on Graphics / SIGGRAPH 2003: tarptautinės konferencijos pranešimų medžiaga.* 2003.
28. **Oat, C.** Real-time 3D Scene Postprocessing. *Game Developers Conference Europe 2003: tarptautinės konferencijos pranešimų medžiaga.* 2003 [žiūrėta 2005-05]. Prieiga per Internetą: http://www.ati.com/developer/gdce/Oat-ScenePostprocessing.pdf
29. **Parker, S. ir kt.** Single Sample Soft Shadows. *Techninė ataskaita UUCS-98-019, Computer Science Department, University of Utah.* 1998.
30. **Phong, B. T.** Illumination for Computer Generated Pictures. *Communications of the ACM*, nr. 18(6), 1975, 311-317 p.
31. **Pranckevičius, A.** Fake Soft Shadows Using Precomputed Visibility Distance Functions. *ShaderX^3: Advanced Rendering with DirectX and OpenGL.* Edited by W. Engel. Hingham, MA: Charles River Media, 2004, 425-435 p., ISBN 1-58450-357-2. *Įtrauktas į 1 priedą.*
32. **Pranckevičius, A.** Minkšti dinaminiai šešėliai realiuoju laiku naudojant matomo atstumo funkcijas. *Informacinės Technologijos 2004: tarptautinės konferencijos pranešimų medžiaga.* Kaunas: Technologija, 2004, 129-134 p.
33. **Ramamoorthi, R., Hanrahan, P.** An Efficient Representation for Irradiance Environment Maps. *ACM Transactions on Graphics / SIGGRAPH 2001: tarptautinės konferencijos pranešimų medžiaga.* 2001, 497-500 p.
34. **Reeves, W. T. ir kt.** Rendering Antialiased Shadows with Depth Maps. *Computer Graphics / SIGGRAPH 1987: tarptautinės konferencijos pranešimų medžiaga.* 1987, 283-291 p.
35. **Scherzer, D.** Real-time Soft Shadows. *Eurographics 2004: tarptautinės konferencijos pranešimų medžiaga (State of the Art Report).* 2004 [žiūrėta 2005-05]. Prieiga per Internetą: http://www.cg.tuwien.ac.at/~scherzer/files/ShadowSTARelectronic.pdf
36. **Segal, M., Korobkin, C.** Fast Shadows and Lighting Effects using Texture Mapping. *Computer Graphics / SIGGRAPH 1992: tarptautinės konferencijos pranešimų medžiaga*. 1992, 249-252 p.
37. **Sloan, P. P.** Efficient Evaluation of Irradiance Environment Maps. *ShaderX^2: Shader Programming Tips and Tricks with DirectX9.* Edited by W. Engel. Plano, TX: Wordware Publishing, 2003, 226-232 p., ISBN 1-55622-988-7.

38. **Sloan, P. P. ir kt.** Clustered Principal Components for Precomputed Radiance Transfer. *ACM Transactions on Graphics / SIGGRAPH 2003: tarptautinės konferencijos pranešimų medžiaga.* 2003.
39. **Sloan, P. P., Kautz, J., Snyder, J.** Precomputed Radiance Transfer for Real-time Rendering in Dynamic, Low-frequency Lighting Environments. *ACM Transactions on Graphics / SIGGRAPH 2002: tarptautinės konferencijos pranešimų medžiaga.* 2002, 527-536 p.
40. **Soler, C., Sillion, F. X.** Fast Calculation of Soft Shadow Textures Using Convolution. *Computer Graphics / SIGGRAPH 1998: tarptautinės konferencijos pranešimų medžiaga.* 1998, 321-332 p.
41. **Stamminger, M., Drettakis, G.** Perspective shadow maps. *ACM Transactions on Graphics / SIGGRAPH 2002: tarptautinės konferencijos pranešimų medžiaga.* 2002, 557-562 p.
42. **Terdiman, P.** Memory Optimized Bounding Volume Hierarchies. *Iš autoriaus asmeninio puslapio [interaktyvus].* 2001 [žiūrėta 2005-05]. Prieiga per Internetą: http://www.codercorner.com/Opcode.pdf
43. **Valient, M., de Boer, W. H.** Fractional Disk Soft Shadows. *ShaderX^3: Advanced Rendering with DirectX and OpenGL.* Edited by W. Engel. Hingham, MA: Charles River Media, 2004, 411-424 p., ISBN 1-58450-357-2.
44. **Wang, Y., Molnar, S.** Second-Depth Shadow Mapping. *Techninė ataskaita TR94-019, University of North Carolina at Chapel Hill*. 1994.
45. **Williams, L.** Casting Curved Shadows on Curved Surfaces. *Computer Graphics / SIGGRAPH 1978: tarptautinės konferencijos pranešimų medžiaga*. 1978, 270-274 p.
46. **Wynn, C.** An Introduction to BRDF-based Lighting. *Iš nVidia Developer Relations [interaktyvus].* 2001 [žiūrėta 2005-05]. Prieiga per Internetą: http://developer.nvidia.com/object/BRDFbased_Lighting.html
47. **Zioma, R.** Reverse Extruded Shadow Volumes. *ShaderX^2: Shader Programming Tips and Tricks with DirectX9.* Edited by W. Engel. Plano, TX: Wordware Publishing, 2003, 587-595 p., ISBN 1-55622-988-7.

TERMINŲ IR SANTRUMPŲ ŽODYNAS

BRDF *(Bidirectional Reflectance Distribution Function)*	Funkcija, nusakanti kaip medžiagos paviršius atspindi šviesą. Kiekviena medžiaga gali būti nusakoma jos *BRDF*.
C++	Programavimo kalba, leidžianti programuoti objektiškai orientuotu, procedūriniu, bendruoju bei funkciniu stiliais arba jų mišiniu.
Direct3D biblioteka	Microsoft kompanijos kuriama trimatės grafikos programavimo biblioteka, skirta Windows operacinėms sistemoms.
HLSL *(High Level Shading Language)*	Aukšto lygio į C panaši programavimo kalba, skirta viršūnių ir taškų apskaičiavimo programoms rašyti. Sudedamoji Direct3D dalis.
Kubinė tekstūra *(cube map)*	Šešių dvimačių tekstūrų, atitinkančių kubo sienas, rinkinys.
Normalė	Statmenas paviršiui vienetinio ilgio vektorius.
Realiojo laiko kompiuterinė grafika *(real-time computer graphics)*	Kompiuterinės grafikos šaka, kurioje vaizdai turi būti apskaičiuojami per sekundės dalį. Dažniausiai 30 ir daugiau apskaičiuojamų vaizdų per sekundę jau vadinama realiojo laiko grafika.
SH *(spherical harmonics)*	Sferinės harmonikos – ortogonaliųjų funkcijų bazė sferos paviršiuje.
Taškų apskaičiavimo programa (*pixel shader)*	Vartotojo programuojama mikrokodo programa, kurią trimačio vaizdo spartintuvas vykdo kiekvienam generuojamo vaizdo taškui (fragmentui).
Tekstūra *(texture)*	Diskretizuota lenteline forma (vienmate, dvimate ar trimate) išreikšta funkcija. Dažnai funkcijos reikšmės yra interpretuojamos kaip paviršiaus spalvos.
Trimačio vaizdo spartintuvas *(3D accelerator)*	Kompiuterio dalis, turinti trimatės kompiuterinės grafikos užduotims pritaikytą procesorių ir greitą darbinę atmintį. Šiuolaikiniai spartintuvai turi didelį lygiagrečių skaičiavimo konvejerių skaičių, SIMD instrukcijų rinkinį, dideles programavimo galimybes ir didelę skaičiuojamąją galią.
Viršūnių apskaičiavimo programa *(vertex shader)*	Vartotojo programuojama mikrokodo programa, kurią trimačio vaizdo spartintuvas vykdo apdorodamas kiekvieno vaizduojamo trikampio kiekvieną viršūnę.

1 PRIEDAS. Straipsnis ShaderX^3 knygoje

Fake Soft Shadows Using Precomputed Visibility Distance Functions

Aras Pranckevičius

Introduction

Dynamic realtime shadowing techniques are commonly used these days. However, approaches such as shadow volumes or shadow maps model direct shadowing only and cannot produce global illumination effects or soft shadows. In recent years new techniques such as Precomputed Radiance Transfer are gaining momentum.

This chapter presents a technique that is faster to compute than PTRs and can handle some cases PRTs cannot. This algorithm renders fake soft shadows in static scenes using precomputed visibility distance functions. The technique handles dynamic local light sources and executes entirely on modern graphics hardware.

Standard shadowing techniques

Commonly used shadowing techniques are:

- Precalculated lighting, usually stored at vertices or in lightmaps. Unfortunately, dynamic geometry or lights are difficult to handle with this type of technique.
- Shadow volumes, usually using stencil buffers or destination alpha channel.
- Texture based approaches, including simple projected shadows, shadow maps (depth or ID based) and various extensions to them.
- Precomputed radiance transfer (PRT) based techniques [Sloan02]. These precompute and approximate a radiance transfer *function* at many points in the scene. However, supporting local lights, especially inside concave objects, is not easy.

Our technique

At the heart of our shadowing algorithm are precomputed visibility distances. Take a static scene and, for every point on its surface and in each possible direction, compute the distance to nearest occluder. When rendering, fetch the distance in the light's direction and compare with the distance to the light. This classifies the rendered point as being in shadow or not.

Sadly, there is no practical way to store the full visibility distance information for all points in the scene. Half of the solution is to store visibility information only at a "dense enough" resolution; the other half is to approximate and compress this visibility information.

PRT methods generally produce better results, can use real-world area light sources, and can model advanced light transport effects like interreflection and subsurface scattering [Sloan03a]. Our technique has one advantage, however – it does not require light to come from outside of the rendered object (using PRT, light must come from infinitely distant source or at least from outside of object's convex hull). Using visibility distance functions, light sources can be anywhere, much like in traditional shadowing methods. The scene depicted in this article's figures is a single closed object, and all lights are actually inside it.

The following section describes each component of the technique in detail.

Visibility distance functions

The visibility distance function for some point p is a (hemi)spherical function that for any given direction returns a scalar visibility distance value. The visibility distance in some direction is the distance to the nearest occluder.

The functions are precomputed and stored at many scene surface points: at vertices for finely tessellated geometry, or in texture maps with a unique parameterization. It does not matter exactly how the functions are stored and approximated. In an ideal world there would be no approximation at all, but there is no practical way to store and evaluate many thousands of spherical functions. So, visibility distance functions have to be compressed in such a way that enables efficient evaluation (perhaps on graphics hardware) and that does not take much memory to store.

In this article, low order spherical harmonics (SH) are used to approximate and store the functions. This is not normally considered the most suitable choice, as the visibility distance function often has large discontinuities, but SH is easy to encode and efficient to evaluate, and with some care can give plausible results.

The visibility distance function for point p is precomputed using Monte Carlo integration [Green03]:

- Trace many rays from p into the scene and check for the nearest collision. In this implementation, AABB trees are used for quick collision checks [Terdiman01] and the rays are uniformly distributed on a sphere [Green03]; each point has 2500 rays cast from it.
- Project the results onto each of the SH basis functions, obtaining an SH encoded visibility distance function. For this article, 5^{th} order SHs are used, so each function is 25 floating-point coefficients.

Other techniques for precomputation are possible, such as getting distances from a GPU-rendered cubemap depth buffer.

Rendering

Rendering is performed for each light, accumulating its contribution. Shadow calculation (which may be at the vertex or fragment level) is done for each light pass:

1. Calculate direction *(x,y,z)* and distance d_L to the light source.
2. Evaluate the visibility distance function in direction *(x,y,z)*, call this d_V. Our functions are stored in SH form, so first project *(x,y,z)* onto SH basis functions and dot the result with visibility distance SH coefficients.
3. If $d_V < d_L$, the point is in shadow (the light is beyond the occluder). The shadow modulation factor is: `shadow = dV >= dL ? 1 : 0`.
4. Perform any local lighting calculations and modulate the results with the shadow.

However, there still are several problems. A typical image is shown in Figure 2 (compare this to Figure 1, where standard shadow maps are used). The shadows are not very correct, to say at least, and there are no signs of soft shadows at all!

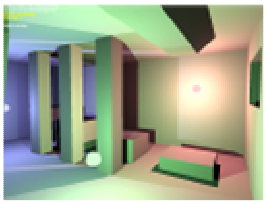


Figure 1. Scene rendered with standard shadow maps.

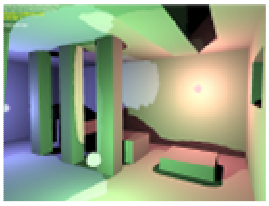


Figure 2. Scene rendered with visibility distance functions, using 5th order SH and binary shadow comparison.

This is where some faking comes in. Low order SH approximation essentially "blurs" our visibility distance functions, removing high frequency details – that is why the strange shadows are in Figure 2. We can notice that global illumination solutions usually are brighter than direct illumination solutions due to multiple light bounces, so it is not very bad to slightly brighten the shadowed areas. We do this by making the comparison of rendering step 3 smooth instead of binary:

3. Shadow modulation factor is: `shadow = saturate( (dV/dL-1.0/3.0)*1.5 ).`

In this way, shadows start to "fade in" when $d_V < d_L$, giving some appearance of soft shadows and global illumination (Figure 3), and hiding SH approximation errors. Of course, the function chosen for smooth comparison is based on experimentation only; the only requirement of this function is that the shadow modulation factor should be 1.0 when $d_V \geq d_L$, and fade to zero when the d_V / d_L ratio decreases.

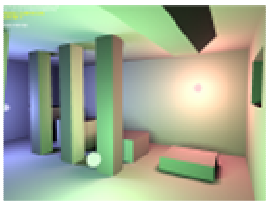


Figure 3. Scene rendered with visibility distance functions, using 5th order SH and smooth shadow comparison.

Rendering details

For reference, the scene used in all the figures is 19x19x6 meters in size and subdivided at 0.2 meters resolution, producing 57 thousand vertices and 100 thousand triangles. The whole scene is a single object. Three attenuated point lights are used, and the user is able to "fire" additional short-lived lights. Visibility distance functions are computed and stored in vertices, 25 coefficients for 5th order SH taking up 100 bytes per vertex (reduced versions are possible, using as low as 9 or 4 coefficients). Shadow maps used for comparison are 512x512 R32F format; shadowing uses one sample from the shadowmap without any filtering.

Three detail levels are implemented, one using a full 5th order SH, with evaluation happening both on vertex and pixel levels; the other two are low-quality versions using 3rd and 2nd order SHs, with all shadow calculation happening at the vertex level.

Rendering with 5th order SH

Spherical harmonics of 5th order produce 25 coefficients. We need to evaluate such a SH-encoded function in the light's direction. Up to 3rd order SH (i.e., the first 9 coefficients) can be easily and efficiently evaluated analytically in the vertex shader [Ramamoorthi01][Sloan03b]. The remaining 16 coefficients are passed to a pixel shader as four sets of 4D texture coordinates and their effect computed there.

SH basis functions for these coefficients are stored in specially made 64x64x6 cubemaps (Q8W8V8U8 format). One cubemap can hold four basis function parameters so four cubemaps are needed. Cubemap texels are computed using standard D3DX functions:

```
void WINAPI gSHCubeFiller( D3DXVECTOR4 *out,
    const D3DXVECTOR3 *uvw, const D3DXVECTOR3 *texelSize,
    void* data )
{
    float coeffs[SH_COEFFS+4]; // SH_COEFFS=25
    int offset = *(int*)data;
    D3DXVECTOR3 dir;
    D3DXVec3Normalize( &dir, uvw );
    D3DXSHEvalDirection( coeffs, 5, &dir );
    const float window = 0.75f;
    *out = D3DXVECTOR4(&coeffs[offset]) * window;
}
// later create the cubemaps
int shOffset = 9; // first 9 analytically
for( int cm = 0; cm < 4; ++cm ) {
    D3DXCreateCubeTexture( device, 64, 0, 0, D3DFMT_Q8W8V8U8,
        D3DPOOL_MANAGED, &mTexSHCubes[cm] );
    D3DXFillCubeTexture( mTexSHCubes[cm], gSHCubeFiller,
        &shOffset );
    shOffset += 4;
}
```

Vertex shader for scene rendering in HLSL for vs_1_1 profile:

```
// input vertex
struct VS_INPUT {
    float4 pos : POSITION;
    float3 normal : NORMAL;
    float4 shA : BLENDWEIGHT0; // first 9 coeffs
    float4 shB : BLENDWEIGHT1;
    float  shC : BLENDWEIGHT2;
    float4 shD : BLENDWEIGHT3; // last 16 coeffs
    float4 shE : BLENDWEIGHT4;
    float4 shF : BLENDWEIGHT5;
    float4 shG : BLENDWEIGHT6;
};

// output vertex
struct VS_OUTPUT_25 {
    float4 pos   : POSITION;
    float  vdist : TEXCOORD0; // vis.distance from 9 coeffs
    float4 sh[4] : TEXCOORD1; // last 16 coefs
    // xyz - dir to light; w - dist to light
    float4 light : TEXCOORD5;
    float4 color : COLOR0;    // vertex lighting
};

// Evaluate SH function in direction v (9 coeffs)
float evalSH9( VS_INPUT i, float3 v ) {
    const float PI = 3.14159265;
    const float SPI = sqrt(PI);
    const float N0 = 1.0/2.0/SPI;
    const float N1 = sqrt(3)/2.0/SPI;
    const float N2 = sqrt(15)/2.0/SPI;
    const float N3 = sqrt(5)/4.0/SPI;
    const float N4 = sqrt(15)/4.0/SPI;
    float res;
    // first 4 components
    res = dot( i.shA.wyzx,
        float4( float3(-N1,-N1,N1)*v, N0 ) );
    // next 4 components
    float4 comb = v.xyzx * v.yzzz; // xy, yz, zz, xz
    float4 tmp2 = float4(N2, -N2, N3*3, -N2) * comb;
    tmp2.z -= N3;
    res += dot( tmp2, i.shB );
    // 9th component
    res += N4 * (v.x*v.x-v.y*v.y) * i.shC;
    return res;
```

```
}

// Vertex shader
VS_OUTPUT_25 vsMain25( VS_INPUT i ) {
    VS_OUTPUT_25 o;
    // transform
    o.pos = mul( i.pos, mViewProj );
    // to light, distance
    o.light = tolight( i.pos, vLightPos );
    // vertex lighting
    o.color = lighting( i.normal, o.light, vLightColor );
    // evaluate first 9 coeffs
    o.vdist = evalSH9( i, o.light.xyz );
    // pass last 16 to ps
    o.sh[0] = i.shD; o.sh[1] = i.shE;
    o.sh[2] = i.shF; o.sh[3] = i.shG;
    return o;
}
```

The pixel shader continues evaluating the visibility distance for the remaining 16 coefficients. It uses four premade SH basis cubemaps (`smpSH0` to `smpSH3`), source in HLSL for ps_2_0 profile:

```
float4 psMain25( VS_OUTPUT_25 i ) : COLOR
{
    float3 l = i.light.xyz;
    float dl = i.light.w, dv = i.vdist;
    dv += dot( texCUBE( smpSH0, l ), i.sh[0] );
    dv += dot( texCUBE( smpSH1, l ), i.sh[1] );
    dv += dot( texCUBE( smpSH2, l ), i.sh[2] );
    dv += dot( texCUBE( smpSH3, l ), i.sh[3] );
    // smooth comparison
    float shadowFactor = saturate( (dv/dl-0.3333)*1.5 );
    // binary comparison
    //float shadowFactor = dv >= dl ? 1 : 0;
    return i.color * shadowFactor;
}
```

Rendering with lower order SH

Lower order SH for visibility distance approximation may be used, sacrificing some quality for lower computing and memory requirements. Interpolating a full four texture coordinate sets and sampling four cubemaps per rendered pixel is not very fast, and keeping 25 coefficients for each vertex is not a small amount of memory.

The first natural approximation is to drop the last 16 coefficients, thus keeping the 3rd order SH coefficients. This can be evaluated nicely in a vertex shader, and even the final shadow factor calculation can be done at vertex level, reducing our pixel shader to simple color output. Both now can be done on DX8 level or even older hardware:

```
struct VS_OUTPUT_LO {
    float4 pos   : POSITION;
    float4 color : COLOR0; // vertex lighting
};
// Vertex shader
VS_OUTPUT_LO vsMain9( VS_INPUT i )
{
    VS_OUTPUT_LO o;
    o.pos = mul( i.pos, mViewProj );
    float4 light = tolight( i.pos, vLightPos );
    o.color = lighting( i.normal, light, vLightColor );
    // shadowing
    float dv = evalSH9( i, light.xyz );
    float dl = light.w;
    float shadowFactor = saturate( (dv/dl-0.3333)*1.5 );
    o.color *= shadowFactor;
    return o;
}
```

```
// Pixel shader
float4 psMainLo( VS_OUTPUT_LO i ) : COLOR
{
    return i.color;
}
```

It is possible to go even further and use 2nd order SH approximation, reducing the per vertex storage to 4 coefficients and distance evaluation to just a couple of instructions.

Results

Figure 4 compares standard shadow mapping with various levels of visibility distance SH approximation (5th, 3rd and 2nd SH order). Surprisingly, rendering performance with our method is acceptable at low SH levels.

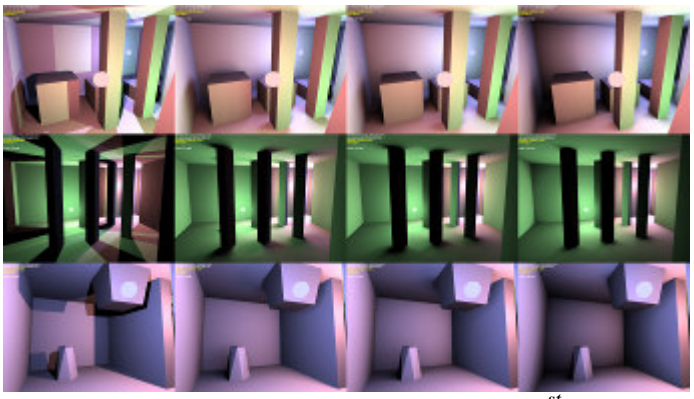


Figure 4. Comparison of shadow maps (1st column) and shadows using visibility distance functions (2nd-4th columns, using 5th, 3rd and 2nd SH order respectively). Shadows lose their form when using low order SH, but in most cases still look plausible. Top row renders at 79, 54, 139 and 159 FPS on Radeon 9800Pro at 1024x768 screen resolution.

Using SH for visibility distance functions is a gross approximation, however the resulting images somehow manage to look acceptable. There are some cases where all these approximations and fakes do fail, especially when the light is very near occluders in several directions and far from occluders in other directions, as shown in Figure 5.

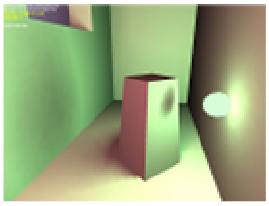


Figure 5. Shadowing failures (the ball is the light source). The most obvious failure is false shadow on the front of the box-like object.

There still are some interesting ideas to try, such as storing visibility distance functions in textures or using other approximation and compression techniques [Sloan03a].

Summary

This chapter presents a technique to render fake soft shadows in static scenes using precomputed visibility distance functions. The technique handles dynamic local light sources and executes at acceptable speeds on modern graphics hardware, with faster lower-quality fallbacks possible on low-end hardware.

References

[Green03] Green R., "Spherical harmonic lighting: the gritty details", Game Developers Conference, 2003

[Ramamoorthi01] Ramamoorthi R., Hanrahan P., "An efficient representation for irradiance environment maps", ACM Transactions on Graphics (SIGGRAPH 2001), 2001: pp. 497-500

[Sloan02] Sloan P.P. et al., "Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments", ACM Transactions on Graphics (SIGGRAPH 2002), 2002: pp. 527-536

[Sloan03a] Sloan P.P. et al., "Clustered principal components for precomputed radiance transfer", ACM Transactions on Graphics (SIGGRAPH 2003), 2003

[Sloan03b] Sloan P.P., "Efficient evaluation of irradiance environment maps" from ShaderX2, Wordware, 2003: pp. 226-232

[Terdiman01] Terdiman P., "Memory optimized bounding volume hierarchies", available online at http://www.codercorner.com/Opcode.pdf, 2001

2 PRIEDAS. Minkštų projektuotų šešėlių algoritmo programų tekstai

Pateikiami algoritmui reikalingų viršūnių ir taškų apskaičiavimo programų tekstai *HLSL* programavimo kalba. Visos programos tekstai šiame darbe nepateikiami – juos sudaro 54900 eilučių C++ tekstų ir 1500 eilučių *HLSL* tekstų; didžioji programos dalis nėra tiesiogiai susijusi su pristatomu algoritmu.

Šešėlį metančių objektų viršūnių ir taškų apskaičiavimo programos (*vs_1_1* ir *ps_1_1* profiliams):

```
SPosCol vsMain( SInput i ) {
     SPosCol o;
     float y = i.pos.y; // apytikslis atstumas iki priimančio objekto
     o.pos = ... // transformuojam viršūnę
     o.color.ra = 0;
     o.color.g = y / 3.0;
     o.color.b = y / 8.0;
     return o;
}

half4 psMain( SPosCol i ) : COLOR {
     return i.color;
}
```

Gauso suliejimo filtro taškų apskaičiavimo programos (*ps_2_0* profiliui):

```
// Gauso kreivės koeficientai
static const float COEFFS[25] = {
     0.00199005,
     0.00367145,
     0.00642218,
     0.0106512,
     0.016749,
     0.0249718,
     0.0353006,
     0.0473137,
     0.0601261,
     0.0724455,
     0.0827621,
     0.0896446,
     0.0920636,
     0.0896446,
     0.0827621,
     0.0724455,
     0.0601261,
     0.0473137,
     0.0353006,
     0.0249718,
     0.016749,
     0.0106512,
     0.00642218,
     0.00367145,
     0.00199005,
};
// Horizontalus Gauso suliejimas
half4 psMain20x( float2 uv : TEXCOORD0 ) : COLOR {
     half4 col = tex2D( smpBase, uv );
     col.g = 0;
     float texel = 1.0 / SHADOW_MAP_SIZE;
     for( int r = -12; r <= 12; r+=1 ) {
          col.g += tex2D( smpBase, uv + float2(texel*r,0) ).g * COEFFS[r+12];
     }
```

```
     return col * 1.1;
}
// Vertikalus Gauso suliejimas
half4 psMain20y( float2 uv : TEXCOORD0 ) : COLOR {
     half4 col = tex2D( smpBase, uv );
     col.g = 0;
     float texel = 1.0 / SHADOW_MAP_SIZE;
     for( int r = -12; r <= 12; r+=1 ) {
          col.g += tex2D( smpBase, uv + float2(0,texel*r) ).g * COEFFS[r+12];
     }
     return col * 1.1;
}
```

Šešėlio formavimo taškų apskaičiavimo programa (*ps_2_0* profiliui):

```
// 12 taškų Puasono disko skirstinys
#define NUM_TAPS 12
half2 filterTaps[NUM_TAPS] = {
     {-0.326212f, -0.405805f},
     {-0.840144f, -0.07358f},
     {-0.695914f,  0.457137f},
     {-0.203345f,  0.620716f},
     { 0.96234f,  -0.194983f},
     { 0.473434f, -0.480026f},
     { 0.519456f,  0.767022f},
     { 0.185461f, -0.893124f},
     { 0.507431f,  0.064425f},
     { 0.89642f,   0.412458f},
     {-0.32194f,  -0.932615f},
     {-0.791559f, -0.597705f}
};

half4 psMain20( float2 tc0 : TEXCOORD0 ) : COLOR
{
     half4 col = tex2D( smpBase, tc0 );

     // filtro plotis iš G kanalo
     half scaling = col.g + 0.02;
     half scale = 12.0/SHADOW_MAP_SIZE;

     // sumuojam filto taškus
     half colorSum = col.b;
     for( int k = 0; k < NUM_TAPS; ++k ) {
          float2 tapCoord = tc0 + filterTaps[k] * (scale * scaling);
          colorSum += tex2D( smpBase, tapCoord ).b;
     }
     // rezultatas - sumos vidurkis
     half shadow = colorSum / (NUM_TAPS+1);
     return shadow*0.4+0.6;
}
```