KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
Multimedijos Inžinerijos Katedra

Mindaugas Vinkelis

PUSIAU SKAIDRIŲ KŪNŲ APŠVIETIMO MODELIAVIMO METODAI TRIMATĖJE GRAFIKOJE

Magistro baigiamasis darbas

Vadovas: dr. S. Drąsutis
Recenzentas: doc. dr. T. Blažauskas

Kaunas 2008

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
MULTIMEDIJOS KATEDRA

Magistro baigiamasis darbas

PUSIAU SKAIDRIŲ KŪNŲ APŠVIETIMO MODELIAVIMO METODAI TRIMATĖJE GRAFIKOJE

Autorius IFM-2/1 gr. stud. Mindaugas Vinkelis .....

Vadovas doc. Sigitas Drąsutis .....

Recenzentas doc.dr. Tomas Blažauskas .....

Kaunas 2008

Summary

Light spreading in space is complicated physical phenomenon, and its simulation in computer graphics is relevant problem. This thesis focuses on partially transparent objects lighting, where light distribution may be written in BTDF[1], algorithm supports multi-colored shadows. Algorithm is real-time and can be fully implemented in modern graphics hardware.

Opacity maps for solid objects divides object into layers, in each of them is stored information about how much light is absorbed in that layer, and where light hit visible object surface. On rendering particular point we take appropriate layer (opacity map) and light absorption intensity is linearly interpolated between different distances stored in other opacity maps. View is not physically correct, in some cases, but reasonable and justifiable because of algorithm speed.

Santrauka

Šviesos sklidimas erdvėje yra sudėtingas fizinis reiškinys, ir jos modeliavimas kompiuterinėje grafikoje yra aktualus uždavinys. Šiame darbe pateikiamas apšvietimo algoritmas pusiau skaidriems, solidiems kūnams, kai šviesos sklidimas gali būti išreikštas BTDF, algoritmas pritaikomas spalvotiems šešėliams. Algoritmas yra realaus laiko, ir gali būti pilnai realizuojamas šiuolaikiniuose trimačio vaizdo spartintuvuose.

Nepermatomumo žemėlapiai solidiems kūnams suskaido vaizduojamą objektą į sluoksnius ir kiekviename sluoksnyje saugo informaciją apie tai, kiek šviesos sugeriama tame sluoksnyje, ir kokioje pozicijoje spindulys atsitrenkė į matomą objekto paviršių. Konkretaus taško piešimo metu imama informacija iš atitinkamo sluoksnio (peršviečiamumo žemėlapio) ir šviesos sugėrimo stiprumas tiesiškai interpoliuojamas tarp skirtinguose žemėlapiuose saugomų atstumų. Kai kuriais atvejais gaunamas vaizdas nėra fiziškai teisingas, tačiau patenkinamas ir pateisinamas algoritmo spartumu.

[1] Bidirectional transmittance distribution function – dvikryptė praleidimo pasiskirstymo funkcija. Aptariama plačiau teorijos apžvalgoje, atskirame skyrelyje.

Terminų ir santrumpų žodynas

OpenGL (Open Graphics Library)	Atvira grafikos biblioteka, skirta visoms operacinėms sistemoms.
Tekstūra	Lentelė (vienmatė, dvimatė, trimatė) kurioje diskretizuotos reikšmės paprastai priimamos kaip paviršiaus spalvos
Normalė	Statmenas vektorius paviršiui, kurio ilgis lygus vienetui
Realaus laiko vaizdavimas	Kompiuterinės grafikos šaka, kurioje vaizdai turi būti apskaičiuojami per sekundės dalį. Jeigu gaunamas vaizdas keičiasi per sekunde virš 24 kartų, matomas nuoseklus judesys.
Kaukės buferis	Tai papildomas buferis į kurį įrašomos reikšmės tiksliai nusakytos funkcijų. Vėliau pagal buferyje esančias reikšmes galime spręsti ar toje vietoje vaizduoti, ar ne.
BxDF (Bidirectional Distribution Function)	Funkcijų šeima, nusakanti kaip medžiagos paviršius reaguoja į šviesą.
GLSL(OpenGL Shading Language)	Aukšto lygio i C panaši programavimo kalba, skirta viršuniu ir tašku apskaiciavimo programoms rašyti OpenGL grafinei sąsajai.
Viršūnių paprogramė (vertex shader)	Kodas, kurį atlieka vaizdo korta kiekvienai viršūnei apskaičiuoti
Fragmentų paprogramė (fragment shader)	Kodas, kurį atlieka vaizdo korta kiekvienam fragmentui apskaičiuoti.
Fragmentas	Taškas rašomas į vaizdo buferį, vėliau jis gali buti perrašomas, arba suliejamas su kitu, ten pat rašomu tašku, parenkant atitinkamas suliejimo funkcijas.
Tekselis	Taško tekstūroje, diskretizuota reikšmė

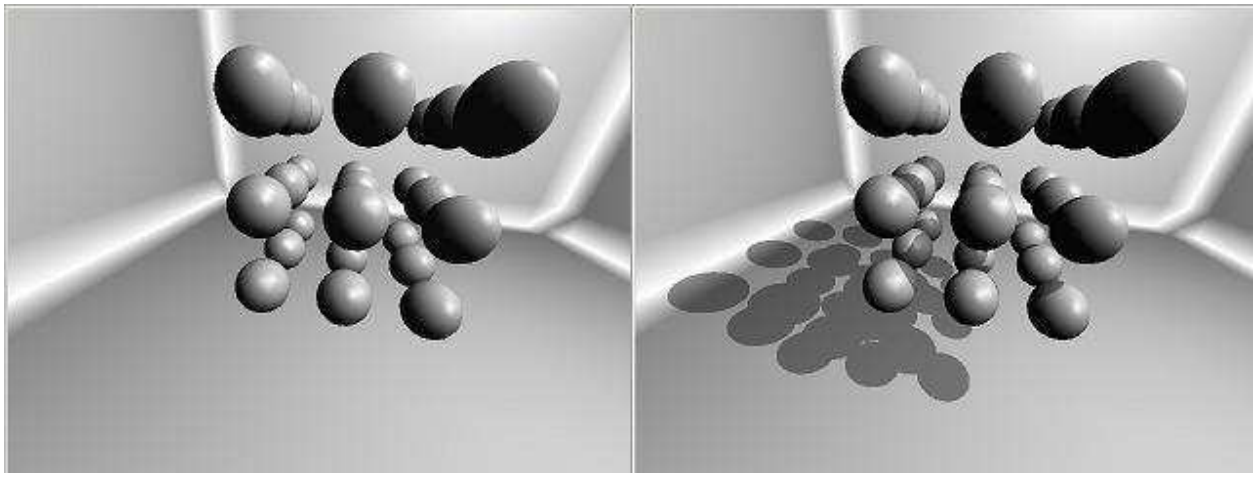
TURINYS

Paveikslėlių sąrašas

Lentelių sąrašas

Įvadas

Šešėlių vaizdavimas kompiuterinėje realaus laiko grafikoje ir, kompiuterinės grafikos programuotojams yra bene aktualiausia tema. Šešėliai padeda geriau suvokti objektų tarpusavio padėtis erdvėje, objektų šviesos atspindėjimo savybes ir jų dydžius. Įvairios medžiagos skirtingai praleidžia šviesą. Pavyzdžiui, idealiai žalias stiklas praleidžia žalią spalvą, ir sugeria visas kitas šviesos komponentes.



1.1.pav. Šešėliai padeda suvokti objektų tarpusavio padėti erdvėje

Šiuolaikinės technologijos sparčiai žengia į priekį, kartu su jomis ir atsiranda vis nauji sprendžiami uždaviniai. Atsiradus pilnai programuojamiems grafikos procesoriams (GPU) didžiausias uždavinys tapo realistiškumas. Dabar galima vaizduoti sudėtingus geometrinius paviršius, modeliuoti mikro-nelygumus, bei tikrų paviršių šviesos atspindžius, tačiau tikroviškas šviesos sklidimas ne tik erdve bet ir kūnų vidumi yra aktualus uždavinys.

Fiziškai teisingas šviesos sklidimas erdve ar kūnu yra labai sudėtingas, ir daug skaičiavimų reikalaujantis procesas. Tradiciniai metodai, spindulių trasavimas *(ray-tracing)* ar energijos perdavimas, nors ir yra fiziškai tikslesni, tačiau vis dar negali būti pilnai realizuojami realaus laiko grafikoje.

Šiuo metu realaus laiko apšvietimo modeliavimo algoritmų yra labai daug, kiekvienas jų turi savų pliusų ir trūkumų, tačiau dauguma jų pritaikyti nepermatomiems kūnams. Yra keletas

metodų, kurie įvertina šviesos sklidimą kūno vidumi. Permatomumo žemėlapių metodas (*translucent maps),* modeliuoja šviesos sklidimą kūno viduje, laikydamas, kad kūno viduje esanti medžiaga tolygiai sugeria visas šviesos komponentes, visame kūne, vienodai. Gylio šešėlių žemėlapių algoritmas *(Deep shadows map)* tinkamas permatomiems šešėliams, ir jo modifikacijos gali vaizduoti spalvotus šešėlius, tačiau jis nėra pritaikytas realaus laiko grafikai. Gylio dengiamumo žemėlapiai *(deep opacity maps)* ir jų analogai yra panašūs į čia pateiktą algoritmą, tačiau neįvertina šviesos spalvos, kuri praeina per objektą, bei matosi sluoksniavimo klaidos.

Realybėje permatomi objektai, tokie kaip vitražinis stiklas, plėvelė, ir kitos medžiagos dalinai sugeriančios šviesą, apšviestos balta spalva (visų spalvos komponenčių reikšmės maksimalios) praleis būtent tas spalvas, kurių jie nesugeria. Tokiu atveju, šviesa praėjusi pro vieną paviršių kitą paviršių apšvies visiškai kita spalva, taigi tikroviškumui padidinti būtų logiška tai įvertinti apšviečiant kiekvieną paviršių.

Realaus laiko grafika šiuo metu nebeįsivaizduojama be trimačio grafikos spartintuvo. Žinant, kad trimatės grafikos spartintuvų skaičiavimo galia didėja gerokai greičiau nei kitų kompiuterio elementų, taigi vis didesnį praktinį pritaikymą turi būtent tie metodai, kurie efektyviau išnaudoja grafinio spartintuvo teikiamas galimybes.

1.2.pav. Realaus pasaulio vaizdas, pro spalvotą stiklą.

Šiame darbe pateikiamas algoritmas modeliuoti šviesos sklidimą pusiau permatomuose vientisuose kūnuose realiu laiku.

Dengiamumo žemėlapiai vientisiems kūnams suskaido objektą į sluoksnius, kuriuose saugo informacija apie sugertos šviesos kiekį tame sluoksnyje, ir piešimo metu ta informacija paimama parenkant atitinkama spalvą konkrečiam taškui. Algoritmo idėja ta pati kaip ir dengiamumo žemėlapių *(opacity maps)* algoritmo, tik į alfa kanalą rašo informaciją apie tai, kurioje vietoje buvo kirsta matoma objekto dalis ateinančio spindulio, tokiu būdu išvengiama sluoksniuotumo klaidų.

Pateiktas algoritmas yra pilnai realizuojamas šiuolaikiniame kompiuteryje, kuris turi programuojamą trimatį grafikos spartintuvą. Algoritmas naudoja CPU (rasti kirtimo plokštumas), tačiau statinėms scenoms CPU yra naudojamas minimaliai. Sisteminė atmintis naudojama objekto struktūrai saugoti, o tekstūros saugomos video atmintyje VRAM. Darbe taip pat pateikiamas detalus algoritmo aprašymas, papildomos realizavimo detales ir patarimai, pateikiami rezultatai bei jų palyginimas su kitais egzistuojančiais panašiais algoritmais.

Teorinėje dalyje pateikiama trumpa vaizdo generavimo apžvalga, paminėti tradiciniai šviesos sklidimo modeliavimo metodai, šviesos sklidimas kūnu, pristatomi panašūs realaus laiko algoritmai modeliuojantys šviesos sklidimą, trumpai užsimenama apie galimas perspektyvas ateityje.

Teorinė dalis

Kompiuterinės grafikos generuojamų vaizdų apžvalga

Kompiuterinės grafikos vienintelė ir pagrindinė funkcija yra vaizduoti kompiuteriu sugeneruotą vaizdą. Skirtingose srityse teikiami ir skirtingi reikalavimai vaizduojamam vaizdui. Kompiuteriniuose žaidimuose pagrindinis kompiuterinės grafikos uždavinys yra viską vaizduoti realiu laiku, vaizdas turi atitikti žaidimo meninį stilių ir neprivalo būti tikroviškas, tačiau pastaruoju metu, išaugus grafinių skaičiavimų apdorojimo galimybėms vis dažniau stengiamasi sugeneruoti kuo tikroviškesnį vaizdą. Kompiuteriniame projektavime vaizdas turi parodyti tikslias savo savybes, dvimates projekcijas, matomas ir nematomas briaunas ir kitas objekto savybes. Vizualizacijose generuojami vaizdai tam, kad būtų galima iš anksto atrodys gaminamas produktas, tokie vaizdai dažnai yra tikroviški pabrėžiant objekto konstruojamas savybes. Filmuose kompiuterinė grafika turi vienintelį reikalavimą,- vaizdas privalo būti kiek įmanoma tikroviškesnis. Čia kompiuterinė grafika naudojama efektams išgauti, kurie būtų per brangūs, per pavojingi, ar tiesiog neįmanomi realizuoti realybėje.

Tai kokį mes matome realų pasaulio vaizdą yra ne kas kita kaip šviesos fotonai. Mes aplinką suvokiame taip, kaip ji reaguoja į šviesą: ar ją sugeria, ar atspindi, ar pati spinduliuoja. Jeigu mus supanti aplinka bus visiškai neapšviesta, negalėsime pasakyti, nei objektu tarpusavio padėties, nei jų fizikinių savybių, taigi tikroviškumui išgauti svarbiausia yra tikroviškas, objektų reagavimo į šviesą, modeliavimas.

Paprastai vaizdui sugeneruoti reikalingi šie parametrai:

1. Geometriniai objektų parametrai. Objektų forma, jų dydis ir tarpusavio išsidėstymas, šviesos šaltinio pozicija ir panašiai.
2. Fizikiniai parametrai. Objektų spalva, paviršiaus tekstūros, šviesos sklaidos funkcijos BSDF, šviesos šaltinių savybės.

3. Vaizduojama scena suprojektuojama į vaizdavimo plokštumą (monitoriaus ekraną, ar kitokį vaizdavimo prietaisą), pasirenkama reikalinga projekcija (2d,3d, 2d izometrinė).
4. Apskaičiuojama matoma scenos dalis pagal tai kaip šviesa apšviečia objektus, kaip objektai atspindi šviesą į kitus objektus ir kaip šviesa patenka į stebėtojo „akis“.

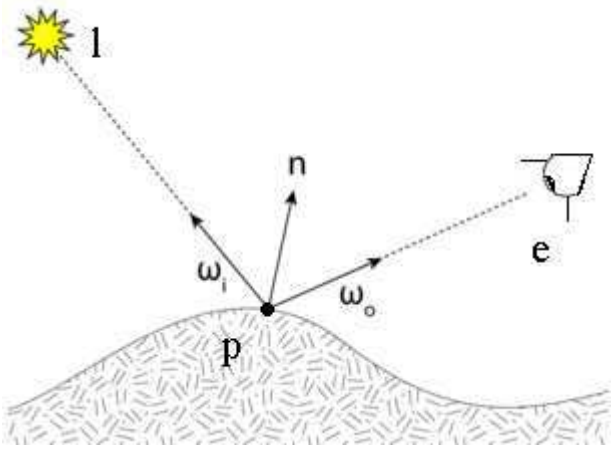
Realaus laiko ir uždelsto (offline) laiko grafika

Esminis skirtumas tarp realaus laiko grafikos, ir uždelstos kompiuterinės grafikos yra vaizdų generavimo greitis. Uždelsto laiko kompiuterinės grafikos metodai paprastai remiasi spindulių trasavimo (*ray – tracing*) idėja, kuri reikalauja labai daug skaičiavimų ir vieno vaizdo sugeneravimas gali užtrukti nuo keliu valandų, ar net iki kelių dienų. Realaus laiko grafikoje vaizdai turi būti generuojami greičiau nei per 1/30 sekundės, todėl čia naudojami kiti vaizdavimo metodai, ir technologijos.

BxDF funkcijų apžvalga

BDF (Bidirectional distribution function) dvikryptė paskirstymo funkcija. Tai labai bendra funkcija, paprastai dažniausiai grafikoje naudojamos BRDF ir BTDF, ir kartais atvejais trumpinama BDF.

BRDF (Bidirectional reflectance distribution function) – dvikryptė atspindžio paskirstymo funkcija. Tai bene labiausiai paplitusi funkcija kompiuterinėje grafikoje. Paprasčiausiam (lokaliam) apšvietimo modeliui 2.1. pav. imkime tašką $\vec{e}$, kuriame yra stebėtojo akis, imkime vektorių $\vec{n}$, kur $\vec{n}$ – normalės kryptis išeinanti iš plokštumos taško $\vec{p}$, tašką $\vec{l}$, kuriame yra šviesos šaltinis. Vektoriais $\vec{\omega}_1, \vec{\omega}_2$ pažymėkime kryptis nuo plokštumos taško $\vec{p}$ link šviesos šaltinio ir stebėtojo.



1.1.pav. Paprasčiausias lokalaus apšvietimo pavyzdys.

Atspindėtos šviesos koeficientas bus intervale [0,1]. 0 reikš, kad objektas visiškai neatspindėjo šviesos ir 1, kad pilnai atspindėjo šviesą stebėtojo link, nuo šviesos šaltinio. Taigi atspindėtos šviesos kiekis priklauso nuo vektorių $\vec{n}, \vec{\omega}_1, \vec{\omega}_2$ ir nuo medžiagos atspindžio paskirstymo funkcijos BRDF.

$$L(\vec{e}, \vec{p}) = L(\vec{p}, \vec{l}) \cdot BRDF(\vec{e}, \vec{p}, \vec{l}) \quad (1)$$

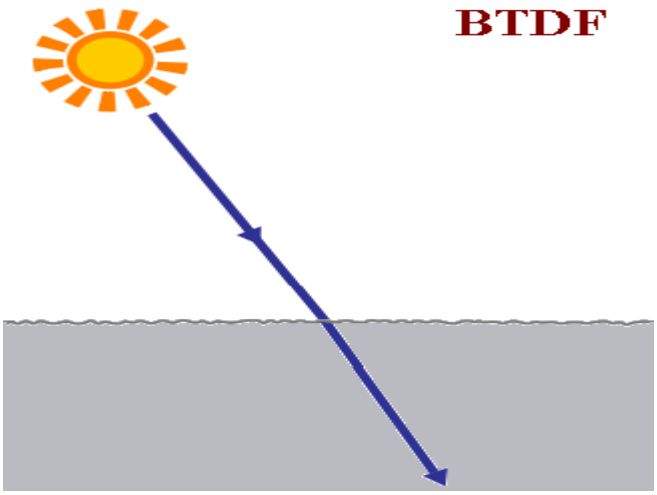
arba bendru atveju

$$f(\vec{\omega}_i, \vec{\omega}_1) = \frac{dL(\vec{\omega}_1)}{dE_i(\vec{\omega}_i)} \quad (2)$$

Kur *f* matomos šviesos intensyvumas, *L* yra atspindėta šviesa nuo paviršiaus, *E* ateinanti šviesa į paviršių.

BRDF yra bendras medžiagos atspindžio aprašas, todėl kompiuterinėje realaus laiko grafikoje dažnai naudojamos supaprastintos, analitinės BRDF išraiškos, kartu papildomai apskaičiuojant šešėlius. Pagrindinės naudojamos BRDF išraiškos yra Lamberto *(difuse),* kai paviršius ateinančią šviesą atspindi vienodai visomis kryptimis, idealaus veidrodžio *(specular),* kai šviesa atspindima veidrodžio principu, bei Phong modelis [9], apytikslis blizgių paviršiu BRDF modelis.

BTDF (Bidirectional transmittance distribution function) – dvikryptė praleidimo pasiskirstymo funkcija. Ilga laiką ji buvo nenaudojama realaus laiko kompiuterinės grafikos algoritmuose. Skirtingai nei BRDF, kuri gali būti pritaikoma visiems matomiems paviršiams, BTDF naudojama tik tiems kūnams, per kuriuos gali praeiti šviesa.

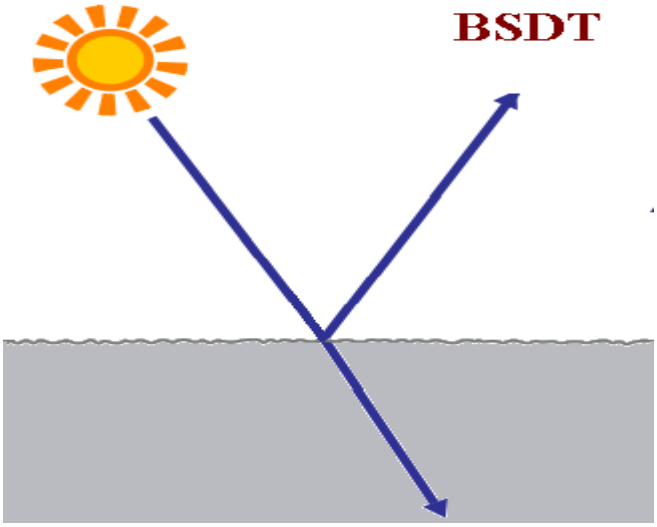


1.2.pav. BTDF apšvietimo modelis

Kadangi šviesa lūžta perėjus per skirtingų tankių medžiagas, tai paleistas spindulys į plokštumą ne tik atsispindės tame plokštumos taške, tačiau dalis šviesos pereis į kūną, ir jame dar karta atsispindėjęs išeis kitame tos pačios plokštumos taške. Tokiu atveju, jeigu kūnas visiškai nepraleidžia į jį patekusios šviesos BTDF yra nulinė.

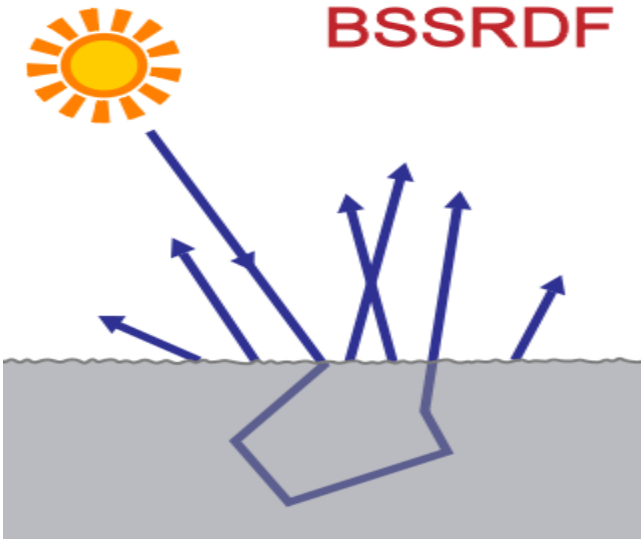
BSDF (Bidirectional scattering distribution function) – dvikryptė sklaidymo pasiskirstymo funkcija.

Tai BRDF ir BTDF funkcijų apibendrinimas. Dažniausia ji naudojama kaip apibendrinta matematinė funkcija aprašanti kaip šviesa yra pasiskirsčiusi plokštumoje. Praktikoje dažniausiai ši funkcija skaidoma į atspindžio ir praleidimo komponentes, - BRDF ir BTDF.



1.3.pav. BTDF apšvietimo modelis

BSSRDF (Bidirectional surface scattering reflectance distribution function) – dvikryptė paviršiaus sklaidymo atspindžio pasiskirstymo funkcija. Pastaruoju metu naudojamas terminas, kuris yra panašus į BSDF, tik įvertina tokį reiškinį kaip po-paviršius *(subsurfaces)* kai šviesa medžiagos viduje lūžta įvairiomis kryptimis, priklausomai nuo medžiagos vidaus savybių ir grįžta atgal.



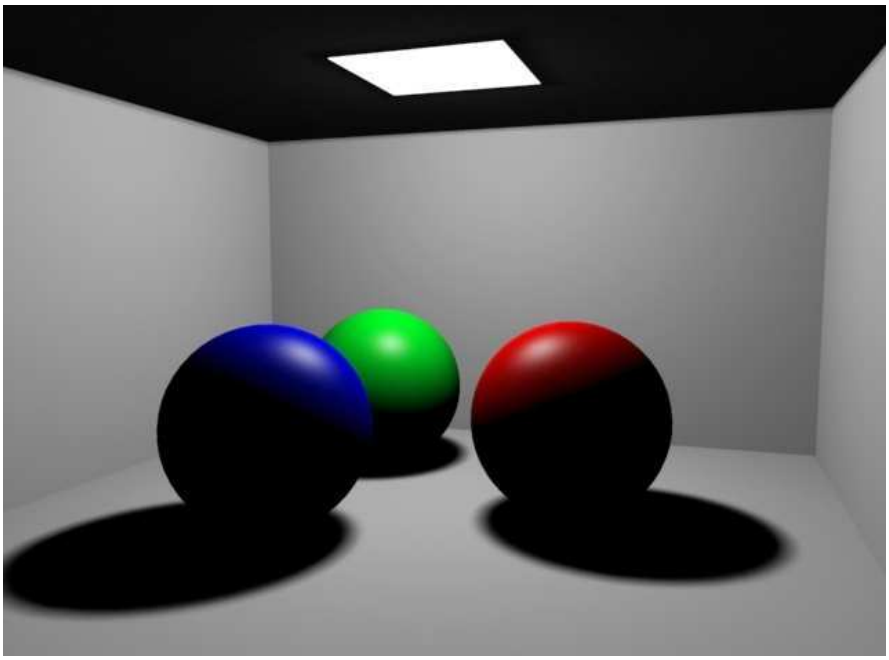
1.4.pav. BSSRDF apšvietimo modelis

Apšvietimo modeliai

Apšvietimo modeliai skirstomi i dvi grupes: lokalus ir globalus apšvietimo modelis.

Lokaliuose apšvietimo modeliuose taško spalva apskaičiuojama įvertinant tik taško, stebėtojo, šviesos šaltinių pozicijas erdvėje, bei medžiagos ir šviesos šaltinių savybes. Čia neįvertinama, kad tas pats taškas, kuriam skaičiuojama spalva, taip pat spinduliuoja šviesą. Lokaliame modelyje taškai traktuojami kaip sugeriantys šviesą, todėl remiantis fizikos dėsniais, tokie taškai turėtų būti visiškai juodi, taigi aišku, kad toks apšvietimo modelis yra labai nerealistiškas. Kadangi lokaliam apšvietimo modeliui reikia įvertinti tik šviesos šaltinius ir medžiagos savybes, tai jis yra labai greitas, ir plačiai naudojamas kompiuterinėje grafikoje, ypač žaidimų kūrimo srityje.

Detalesnis šio modelio paaiškinimas pateiktas kaip pavyzdys BRDF apžvalgoje.



1.5. pav. Lokalus apšvietimo modelis

Globalūs apšvietimo modeliai skaičiuoja taško spalvą ne tik remiantis pagrindiniais duomenimis, bet ir įvertinant visą vaizduojamos scenos informaciją. Konkrečiam taškui spalva apskaičiuojama ne tik pagal šviesos šaltinių ir to taško medžiagos savybes, bet įvertinama ir tai,

kad aplinkiniai taškai taip pat skleidžia šviesą ją atspindėdami, arba tiesiog iš savęs. Bendras apšvietimo modelis apsirašo lygtimi, kuri nurodo kiek šviesos patenka iš taško $\vec{p}$ į tašką $\vec{e}$. Taigi globaliam apšvietimui gauname tokia apšvietimo lygtį.

$$L(\vec{e},\vec{p}) = v(\vec{e},\vec{p})(L_e(\vec{e},\vec{p}) + \int_s BRDF(\vec{e},\vec{p},\vec{s}) \cdot L(\vec{p},\vec{s})d\vec{s}) \quad (3)$$

$v(\vec{e},\vec{p})$ - funkcija parodanti ar taškas $\vec{e}$ mato tašką $\vec{p}$, Jeigu mato tai gražina 1, jei nemato 0.

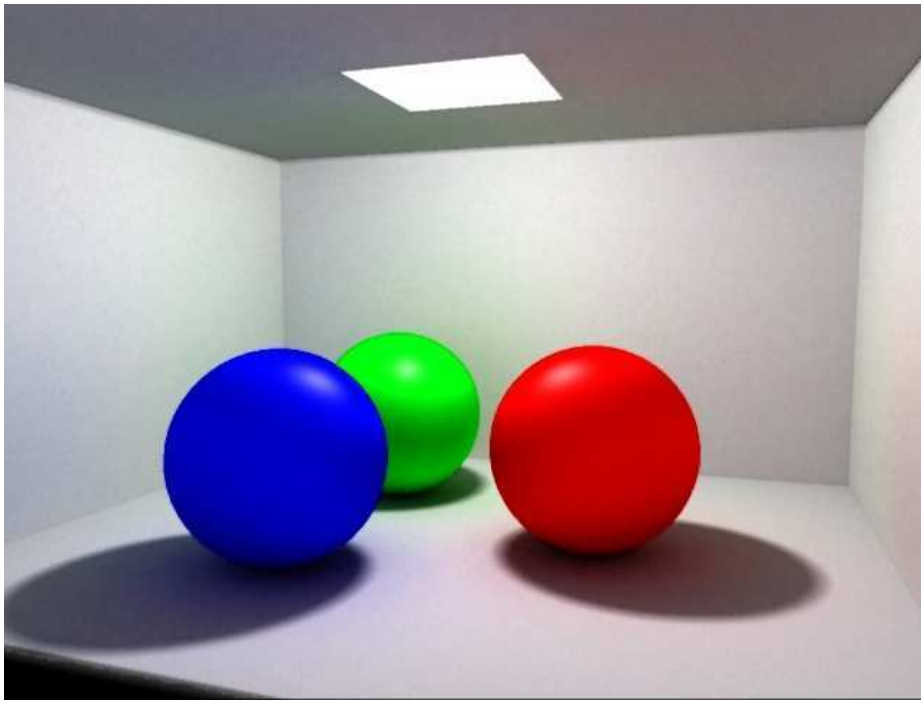
$L_e(\vec{e},\vec{p})$ - funkcija rodo kiek šviesos ateina į tašką $\vec{e}$ iš tašką $\vec{p}$, ji gražina ne nuli tuomet kai $\vec{p}$ yra šviesos šaltinis.

s - visų scenos taškų aibė.

$\int_s BRDF(\vec{e},\vec{p},\vec{s}) \cdot L(\vec{p},\vec{s})d\vec{s}$ - visų scenos taškų aibės atspindimos šviesos suma.

Vaizdavimo lygtis teigia, kad iš taško $\vec{p}$ į tašką $\vec{e}$ patenkanti šviesa yra skleidžiamos ir atspindimos šviesos suma, jei taškai yra matomi vienas kitam. Toks apšvietimo modelis vadinamas *globaliu apšvietimu (global illumination)*. Iš lygties matome esmini skirtumą tarp lokalaus ir globalaus apšvietimo modelio, kad globalus apšvietimas įvertina ne tik skleidžiamą šviesą, bet ir atspindima visų taškų šviesos sumą.

Realaus pasaulio atveju tokia lygtis turi begalinį iteracijų skaičių, kadangi abiejuose lygties pusėse yra $L(\vec{e},\vec{p})$ funkcija, taigi praktiškai apskaičiuoti tokio apšvietimo yra neįmanoma. Lokaliuose modeliuose dažnai priimta laikyti, kad visų scenos taškų atspindėtos šviesos sumos spalva yra pilka, ir lygi intensyvumui 0,2 (nulis reiškia juoda, vienetas reiškia balta). Konkretaus taško apšvietimo atveju prie gautos apšvietimo reikšmės nuo tiesioginių šviesos šaltiniu, tiesiog pridedama globalaus apšvietimo konstanta. Žinoma, tokia vaizdavimo funkcijos aproksimacija negali būti laikoma globaliu apšvietimu. Globalaus apšvietimo modeliai naudoja kitas euristikas ir aproksimacijas, dažnai pasirenkant tarp vaizdavimo kokybės ir greičio.



1.6. pav.Globalus apšvietimo modelis

Ankstesni darbai

Tradiciniai realaus laiko kompiuterinės grafikos apšvietimo metodai

Realaus laiko kompiuterinės grafikos apšvietimo metodai skirstomi į dvi grupes, statinis apšvietimas ir dinaminis apšvietimas.

Jeigu vaizduojama scena yra nejudanti, tai joje esantį apšvietimą galima vaizduoti jau iš anksto paskaičiavę naudodami tradicinius metodus, (pvz.: spindulių trasavimas *(ray tracing)*, fotonų skleidimas *(photon mapping),* ar energijos perdavimo modeliavimas *(radiosity)*). Kadangi šie metodai yra labai „brangūs“ procesoriaus laiko atžvilgiu ir netinkantys vaizduoti realiu laiku, taigi mums tereikia vieną kartą apskaičiuoti ir gautus rezultatus išsaugoti. Vaizduojant realiu

laiku naudojami tiesiog išsaugoti rezultatai. Apšvietimo reikšmės paprastai saugomos tekstūrose (*lightmaps),* arba viršūnėse (*per vertex lighting*).

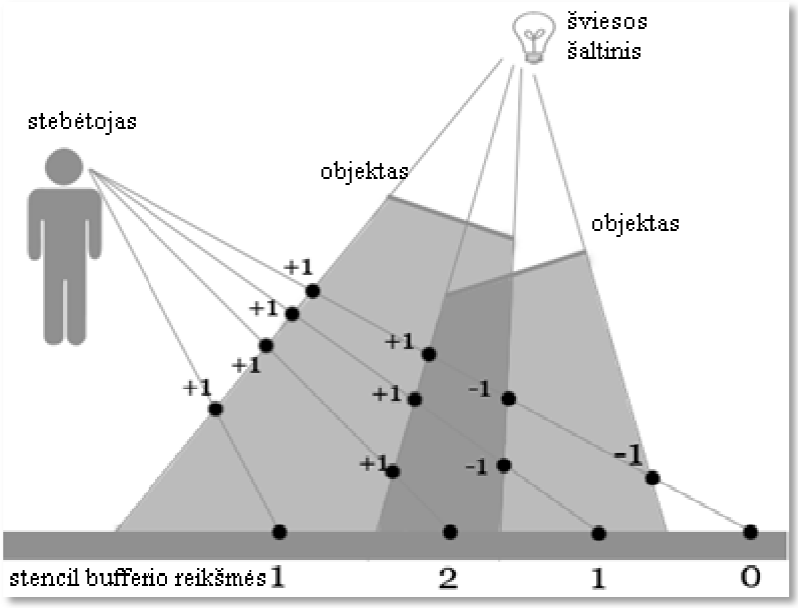
Toks apšvietimas gali būti atliekamas nuo bet kokio šviesos šaltinio bet kokiai scenai, svarbu, kad scena būtų statinė. Šie apšvietimo modeliai turi didelių privalumų, nes yra labai greitai vaizduojami, ir gaunami rezultatai yra labai tikroviški. Pagrindinis trūkumas tas, kad nei scena, nei šviesos šaltinio konfigūracija negali keistis. Egzistuoja ir kiti algoritmai paremti šiais metodais, pavyzdžiui: iš anksto apskaičiuojamas difuzinis apšvietimas ir papildomai dinamiškai vaizduojami blizgūs paviršiai.

1.7. pav. Realaus laiko statinė scena.

Dinaminis apšvietimas turi kur kas platesnį pritaikymą. Be jo neįmanoma išsiversti šiuolaikiniuose žaidimuose. Per pastaruosius keletą metų atsiradus galimybei programuoti grafikos procesorius GPU, kartu išaugo realaus laiko dinaminių apšvietimo metodų skaičius. Toliau aptariami keli tradiciniai ir nauji dinaminio apšvietimo generavimo metodai.

Tūriniai šešėliai *(shadow volumes)*

Tai kokybiški kieti šešėliai vaizduojami realiu laiku. Algoritmas generuoja šešėlius sudarydamas plokštumas. Plokštumos (šešėlio tūris) sudaroma pratempiant objekto kraštines, kurios atitinką objekto siluetą, ta pačia kryptimi kokią sudaro šviesos spindulys sklindantis iš šviesos šaltinio silueto kryptimi. Paskutinis žingsnis yra eliminavimas tų vietų kurios turi būti matomos. Vaizdavimo metu kiekvienam taškui skaičiuojama kiek kartų spindulys iš stebėtojo pozicijos kerta šešėlių tūrius, jeigu nors vieno tūrio pilnai nekerta, vadinasi taškas yra šešėlyje. Skaičiavimas atliekamas kaukės (*stencil*) buferio pagalba. Vienas iš algoritmų yra toks: Prieš piešiant šešėlius pirmiausia nupiešiama scena į gylio buferį. Sekančiu etapu piešiant šešėlio tūrį matomi paviršiai (*front face*) padidina kaukės buferio reikšmes vienetų tose vietose kur gylio testas praeina (*depth pass*) t.y. šešėlio neuždengia plokštuma, o nematomi paviršiai (*back face*) sumažina kaukės buferį vienetu, taip pat tik ten kur gylio testas praeina. Tuomet užtušuojamos vietos tose vietose kur kaukės buferio reikšmė yra nelygi nuliui. (2.8 pav.) Toks algoritmas yra vadinamas *z-pass* ir jį pirmą kartą pristatė Heidmann 1991 metais. [1]



1.8. pav. Tūrinių šešėlių apskaičiavimo iliustracija. *Z-pass* algoritmas

Tūrinių šešėlių privalumai:

1. Tinka taškiniams šviesos šaltiniams.
2. Formuojami šešėliai neturi laiptuoto šešėlio kontūro *(jagged)*.

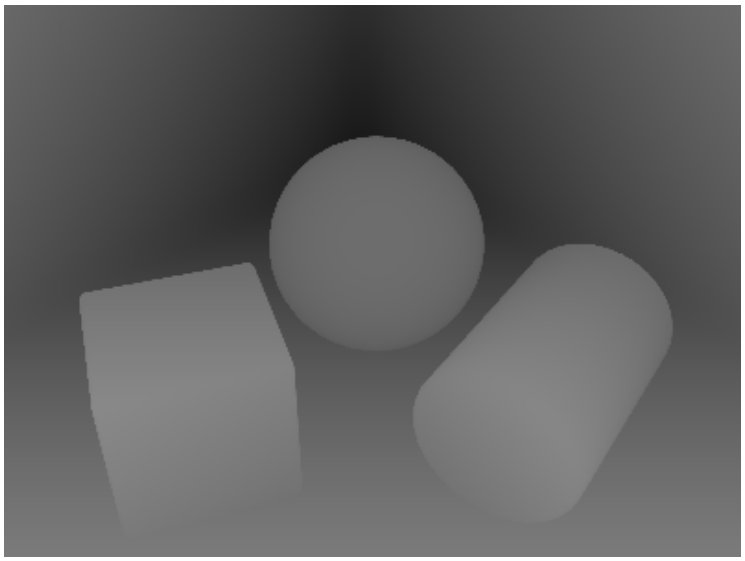
Tūrinių šešėlių trūkumai:

1. Apribojimai objekto geometrijai (modelis privalo būti sudarytas iš plokščių paviršių, ir privalo būti uždaras).
2. Reikalingas silueto sudarymas (piešti tūrius).
3. Naudoja daug grafinio procesoriaus užpildymo *fillrate (nes kiekviena tūrį reikia rasterizuoti)*.

Pirmą kartą tūriniai šešėliai buvo paminėti 1991 metais [1] tačiau įvairūs algoritmo išplėtimai pasirodė tik paskutinį dešimtmetį. 2000 metais John Carmac, [2] *id software* pagrindinis programuotojas, pasiūlė sprendimą tūriniams šešėliams, tuo atveju kai stebėtojas yra palindęs po metamu šešėliu. 2002 metais *GDC* (*Game Developers Conference*) Cass Everitt ir Mark J. Kilgard [3] apžvelgė pagrindinį tūrinių šešėlių algoritmą pateikdami daugybę patobulinimų, dvipusio kaukės buferio panaudojimą, šešėlių generavimą kryptingiems šviesos šaltiniams. Haines 2001 [4] pasiūlė pirmą efektyvų metodą minkštiems šešėliams generuoti.

Gylio tekstūrų šešėliai *(depth shadow mapping)*

Algoritmas panašus į klasikinį projektuotų šešėlių algoritmą, tik čia į tekstūrą piešiamas ne objektas, o objekto *gylio buferis* (*depth buffer*). Tuomet gylio tekstūros matrica suprojektuojama į stebėtojo koordinačių sistemą ir piešiant sceną kiekvieno taško (fragment) xyz koordinatės ir z reikšmė yra lyginama su atitinkamu *tekšelių (taškas tekstūroje)* gylio tekstūroje *(shadow map) 2.9 pav.*. Jeigu taško reikšmė didesnė už tą, kuri yra gylio tekstūroje tai taškas yra šešėlyje, priešingu atveju apšviestas. Šio algoritmo privalumas yra tas, kad ant objekto gali būti sumodeliuotas jo paties šešėlis, vykdymo laiko atžvilgiu jis labai artimas projektuotiems šešėliams.



1.9. pav. Gylio tekstūroje surašytos gylio buferio reikšmės.

Gylio tekstūrų privalumai:

1. Neturi apribojimų objekto geometrijai, nei šešėlio sudarymui nei šešėlio vaizdavimui.
2. Priešingai nei tūriniai šešėliai, gylio tekstūros praktiškai nenaudoja GPU užpildymo.

Gylio tekstūrų trūkumai:

1. Dėl gylio buferio rašymo į tekstūrą neišvengiamai buferio reikšmės turi būti diskretizuojamos, dėl to gaunami šešėliai yra laiptuoti. *(aliased)*
2. Didėjant atstumui nuo šviesos šaltinio iki šešėlio atsiranda klaidos, dėl gylio buferio prigimties sumažint tikslumą toliau esantiems objektams, ir padidinti arčiau esantiems (toks reiškinys atsiranda dėl perspektyvios projekcijos prigimties)
3. Bendras algoritmas neleidžia naudoti taškinių šviesos šaltinių, tik kryptingus. Norint naudoti taškinius šviesos šaltinius paprastai reikia kubinės tekstūros (6 tekstūros), o tai gali padidinti vaizdavimo laiką iki 6 kartų (priklausomai nuo scenos sudėtingumo).

Nors gylio tekstūros turi nemažai trūkumų, tačiau šiuo metu jos yra labai plačiai naudojamos. Yra įvairių metodų sumažinančių diskretizavimo klaidas *(percentage closer filtering, ar variante shadow maps)*, įvertinančias perspektyvą gylio tekstūros *(perspective shadow maps)* ir k.t.

Visi šie metodai ir jų panašūs analogai remiasi supaprastintomis BRDF, tai gerai jeigu objektų paviršiai yra kieti, ir atspindi visą šviesą, tačiau realybėje dauguma paviršių dalį šviesos sugeria. Toliau aptariami metodai įvertinantys BSDF ir BSSRDF funkcijas, kuriais remiantis buvo sugalvotas čia pasiūlytas algoritmas.

Šiuolaikiniai realaus laiko kompiuterinės grafikos apšvietimo metodai.

Šiuolaikinės kompiuterinės grafikos galimybės per pastarąjį dešimtmetį labai išaugo. 1991 metais pasirodė pirmas realaus laiko šešėlių algoritmas [1,2,3] sugebantis mesti šešėlį ant bet kokios formos objektų ir ant pačio savęs, jis naudojo tais pačiais metais atsiradusį kaukės (stencil) buferį. Atsiradus gylio tekstūroms, prasidėjo tekstūromis paremta šešėlių karta.

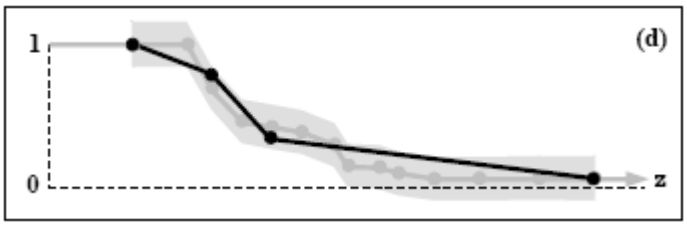
Šiuo metu yra daug įvairių apšvietimo metodų, tiek statinėms scenoms tiek dinaminėms. Pasiūlytas ne vienas tūrių šešėlių metodo praplėtimas, leidžiantis vaizduoti švelnius šešėlius. Taip pat pasiūlyti gylio tekstūrų metodais remti algoritmai gebantys vaizduoti švelnius šešėlius, bei visiškai nauji metodai naudojami dalinai permatomiems kūnams, tokiems kaip rūkas, dūmai, plaukai ir vientisiems kūnams. Toliau aptarsime pagrindinius apšvietimo metodus dalinai permatomiems kūnams vaizduoti, jais remiantis buvo sugalvotas šiame darbe pasiūlytas metodas.

Gilūs šešėlio žemėlapiai *(deep shadow maps)*

Priešingai nei tradiciniai gylio tekstūrų žemėlapiai, kurie laiko vieną gylio reikšmę kiekviename taške, gilūs šešėlio žemėlapiai [5] (toliau DSM) laiko suglaustą matomumo funkciją kiekviename taške, ir visuose galimuose gyliuose. Algoritmas vykdomas dviem žingsniais: pirma

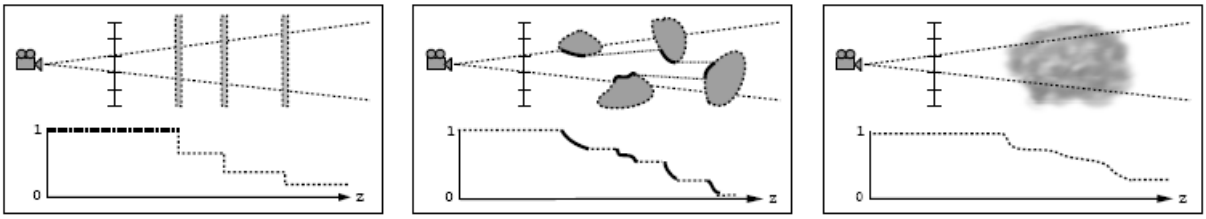
sudaromi žemėlapiai, piešimo metu skaitoma iš jų informacija (suglaudinta matomumo funkcija), ir konkrečiam taškui priskiriama konkreti spalva.

Matomumo funkcijos apskaičiuojamos paleidžiant spindulį iš šviesos šaltinio link scenos, kiekvienam pikseliui, ir trasuojama tol kol spindulio intensyvumas tampa nuliu. Vėliau ji yra suspaudžiama pasirenkant maksimalią galimos paklaidos reikšmę, ir naudojant godų *(greedy)* algoritmą, sumažinamas funkcijos kontrolinių taškų kiekis.



1.10. pav. Matomumo funkcijos suspaudimas

Šviesiai pilka spalva pažymėtos realios reikšmės gautos trasuojant spindulį, stora šviesi linija yra maksimali galima paklaida, ryškiai juoda linija yra suspausta matomumo funkcija pagal godų algoritmą.



1.11. pav. Matomumo funkcijos

a) spindulio stiprumas yra mažinamas kiekviena kartą kai praeinama per pusiau skaidrų paviršių, b) spindulį blokuoja dengiami paviršiai, tačiau jie uždengia tik dalį pikselio zonos, c) spindulys eina pro dūmus, ir silpnėja labiau tiesiškai

DSM turi daug privalumų:

1. Yra iš anksto filtruoti *(pre-filtered)*, kas leidžia vykdytis greitesnius tekstūrų ieškojimus *(lookup)* ir naudoja daug mažiau atminties, negu paprasti gylio tekstūrų žemėlapiai panašios kokybės.
2. Palaiko šešėlius iš dalinai permatomų paviršių, ir tūrinių objektų, tokių kaip rūkas.
3. Leidžia realizuoti veiksmo suliejimo efektą *(motion blur)* be papildomų sąnaudų.

4. Turi galimybę generuoti spalvotos šešėlius išnaudojant tik 2 kartus daugiau atminties.
5. Gaunami vaizdo rezultatai yra globaliai teisingi.

Didžiausias DSM trūkumas yra tas, kad jis negali būti realizuojamas realiu laiku, dėl spindulių trasavimo sudarant matomumo funkcijas, kurios reikalauja daug skaičiavimų.

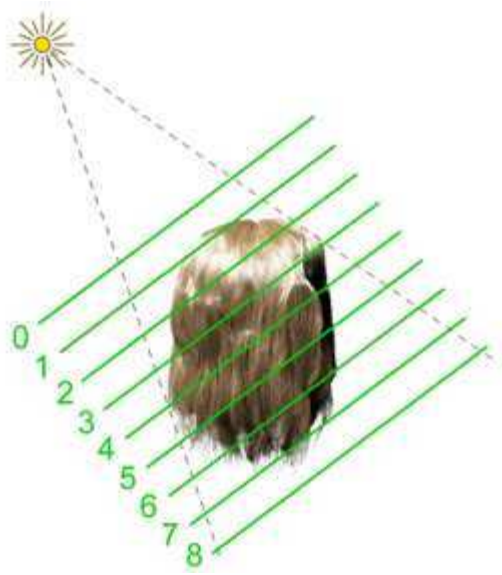
1.12. pav. Pavaizduota plaukų sruoga su DSM (kairėje) ir be DSM.

Nors šis metodas nėra realaus laiko, tačiau juo remiantis buvo sukurta keletas supaprastintų šio metodo versijų realaus laiko grafikai.

Dengiamumo žemėlapiai *(Opacity shadow maps)*

Dengiamumo žemėlapių metodas [6] yra paprastesnė gilių šešėlio žemėlapių versija ir buvo pirmiausia sukurtas dėl plaukų vaizdavimo realiu metu, tačiau gali būti pritaikytas ir dūmų, rūko ar kitų panašių pusiau permatomų objektų vaizdavimui.

Pirmiausia jis apskaičiuoja į kiek plokštumų-dalių (dengiamumo žemėlapių) reikia padalinti vaizduojamą objektą. Tos plokštumos yra statmenos šviesos šaltinio krypčiai ir yra nustatomos pagal atstumą nuo šviesos šaltinio. Tuomet dengiamumo žemėlapiai skaičiuojami piešiant objektą iš šviesos šaltinio pozicijos. Kiekvienam dengiamumo žemėlapiui reikalingas atskiras piešimas (*pass*), nukerpant (clipping) tą objekto geometriją, kuri nepriklauso konkrečiam dengiamumo žemėlapiui ir į žemėlapį surašant alfa reikšmes.



1.13. pav. Objektas suskaldomas į sluoksnius

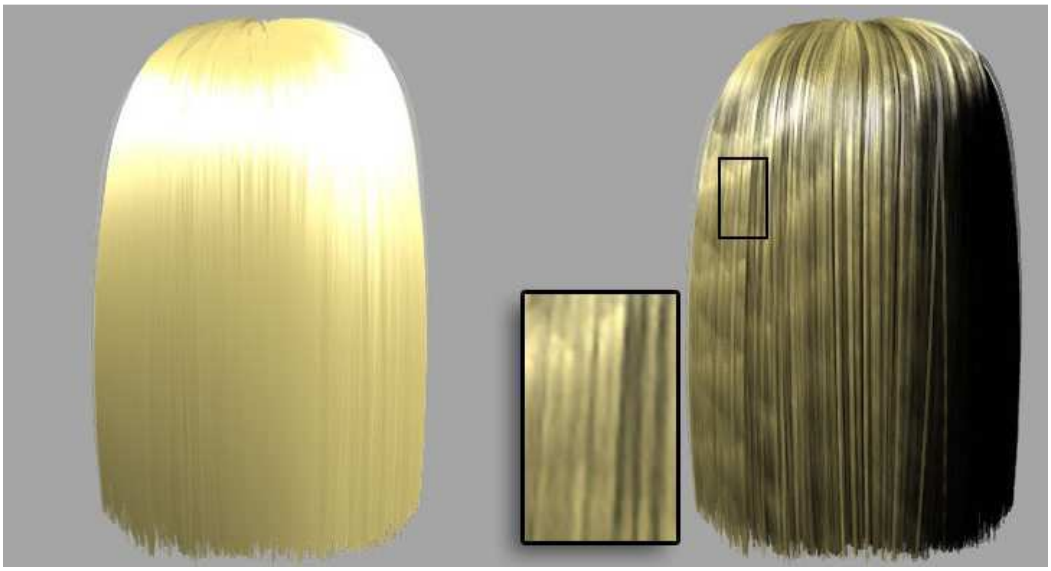
Objekto tankumas kiekviename taške apskaičiuojamas naudojant papildomą suliejimą. Dengiamumo žemėlapiai yra piešiami pradedant nuo to kuri yra arčiausiai šaltinio link, ir reikšmės nuo praeitos žemėlapio yra sukaupiamos (surašomos i *accumulation* buferį) kitam žemėlapiui. Kai visi sluoksniai yra nupiešti, šis dengiamumo žemėlapis gali būti panaudojamas surasti praleistos šviesos kiekį bet kuriame taške naudojant tiesioginį reikšmių interpoliavimą tarp reikšmių paimtų iš gretimų sluoksnių.

Metodo privalumai:

1. Lankstus algoritmas,- galima nesunkiai pasirinkti tarp kokybės ir greičio nustatant sluoksnių skaičių.
2. Pakankamai lengvai realizuojamas.
3. Vienas iš ne daugelio tokio pobūdžio realaus laiko metodų.

Metodo trūkumai:

1. Kadangi sluoksniai, kuriuose saugoma šviesos pralaidumo informacija, yra plokšti, o vaizduojamas objektas retai išlaiko tokią struktūrą, tai ieškant konkrečiam taškui šviesos pralaidumo neišvengiamai atsiranda sluoksniavimo klaidos.
2. Metodas labiau pritaikytas smulkiems pusiau permatomiems kūnams.



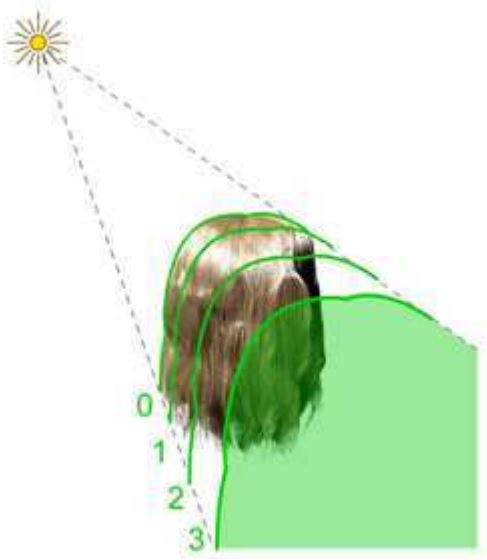
1.14. pav. Vaizdas naudojant dengiamumo žemėlapius (16 sluoksnių) (dešinėje), ir be šešėlių (kairėje).

Nors galima pasirinkti tarp greičio ir kokybės, tačiau vis tiek nebus galima išgauti DSM metodu vaizduojamos kokybės. Kadangi dengiamumo reikšmės yra sluoksniuotos, ir nepaisant sluoksnių reikšmių interpoliavimo taškui, vis tiek matosi sluoksniavimo klaidos. Šios klaidos lieka mažiau matomos, kai naudojamas labai didelis sluoksnių skaičius.

Gilūs dengiamumo žemėlapiai *(deep opacity maps)*

Gilūs dengiamumo žemėlapiai [7] ir jo pirmtakas yra labiau pritaikyti smulkiems paviršiams: plaukams, dūmams, rūkui. DSM metode pastebėjome, kad šviesa praeinanti per

dūmus silpnėjo beveik tiesiška funkcija. Taigi čia pristatomas metodas praplečia pagrindinį dengiamumo žemėlapių modelį naudojant gylio žemėlapį *(depth map)*, kad sudarytų tokią sluoksnių formą, kokios yra gylio buferis 2.15 pav.



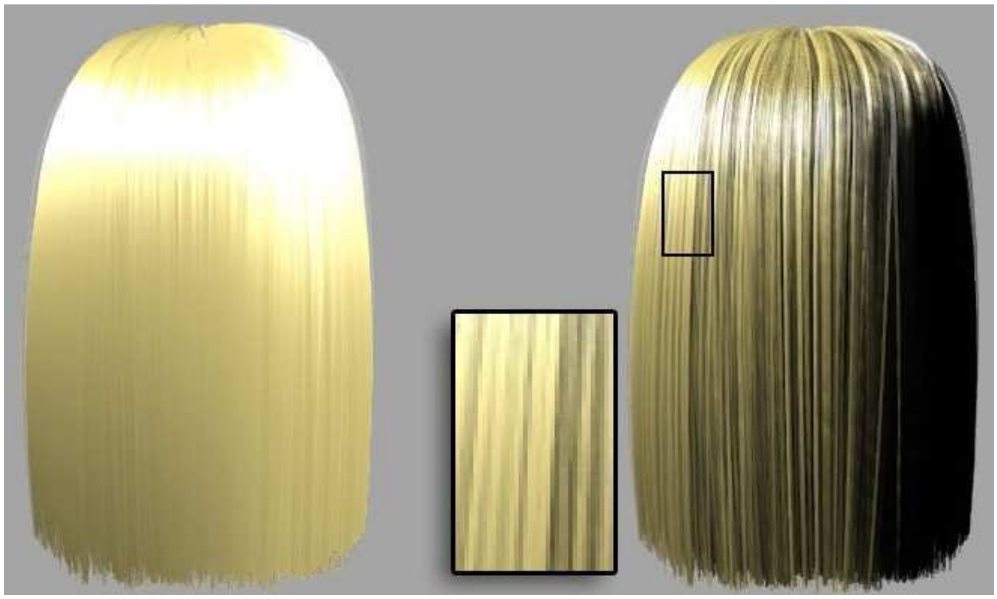
1.15. pav. Objektas suskaidomas į sluoksnius išlaikant objekto kontūrą.

Toks būdas leidžia reikšmes interpoliuoti tiesiškai tarp objekto tūrio, o ne tarp atskirų sluoksnių, kas panaikina standartinių dengiamumo žemėlapių sluoksniavimo klaidas, ir reikalauja kur kas mažiau sluoksnių norint pasiekti aukštos kokybės šešėlių apskaičiavimus. Kiekvienas sluoksnis įgyja objekto formą žiūrint iš šviesos šaltinio.

Šio metodo privalumai:

1. Sluoksnių formos prisitaikymas prie objekto kontūrų, pašalina vaizdavimo netikslumus filtruojant tarp sluoksnių.
2. Reikalauja žymiai mažiau sluoksnių, dėl to gali realizuotas taip, kad pirmas piešimas įrašant į sluoksnius, ir paskutinis piešimas, gali būti įvykdytas per vieną kartą.

Metodo trūkumas tas, kad jis mažiau pritaikomas objektams, kurie savyje turi daug tuščių ertmių, kadangi tiesiškai interpoliuojamos reikšmės yra tik tokios struktūros, kokios yra objekto kontūras.



1.16. pav. Vaizdas naudojant gilius dengiamumo žemėlapius (3 sluoksnių) (dešinėje), ir be šešėlių (kairėje).

Teorinės dalies išvados

Minkšti šešėliai, BSSRDF ir BSDF paremti algoritmai šiuo metu turi labai didelį susidomėjimą mokslininkų tarpe. Kadangi šie algoritmai dar nauji, tai kiekvienas jų yra vis dar tobulinami, ir labai sunku įvertinti jų gerumą, nes nė vienas jų nėra universalūs.

Nors palyginti neseniai atsirado gylio tekstūros, tačiau galima pastebėti, kad būtent jomis paremti nauji algoritmai išplėstiems minkštiems šešėliams, ir pusiau permatomiems kūnams atrodo perspektyviausi. Tačiau praktiškai nėra algoritmų apskaičiuojančių permatomus šešėlius vientisiems, tiesioginę šviesa, praleidžiantiems kūnams, taigi šiame darbe bus pristatomas algoritmas, tokiems kūnams apšviesti.

Tiriamoji dalis

Dengiamumo žemėlapiai vientisiems kūnams.

Reziumė

Šis algoritmas turi panašumų su dengiamumo žemėlapių *(opacity shadow maps)* ir su gylio dengiamumo žemėlapių *(deep opacity maps)* metodais. Skirtingai gylio dengiamumo žemėlapių metodas, šis metodas nesistengia išlaikyti objekto kontūro, nes dažniausiai vientisus objektai turi chaotiška sluoksnių pasiskirstymą, taigi nėra pagrindo išlaikyti objekto kontūrą, tačiau algoritmas naudoja alfa kanalą, kuriame pasižymi atstumą kada spindulys kerta plokštumą tame sluoksnyje. Į sluoksnių dengiamumo žemėlapius saugomos visų spalvų komponentes. Sluoksniai yra pasiskirstę tolygiu atstumu vienas nuo kito, kadangi metodas daugiau taikytas vientisiems, šviesą praleidžiantiems kūnams.

Dengiamumo sluoksnių generavimas

Jeigu paleistume spindulį iš šviesos šaltinio į objektą, kiekvieną kartą susidūręs su objekto paviršiumi jis susilpnėtų. Mūsų tikslas turint atskirame sluoksnyje saugomą informaciją apie jame susilpnėjusią šviesą, nustatyti koks yra šviesos intensyvumas konkrečiame sluoksnyje.

Pirmiausia reikia įvertinti kertamų plokštumų pasiskirstymo dėsningumą tarp sluoksnių. Tarkime turime du objektus, vienas sudarytas iš smulkių detalių (tarkim dūmai ar debesys). Kitas objektas, stambus objektas, turintis žymiai mažiau persidengiančių sluoksnių. Nesunku suprasti, kad kuo tankesnis objektas, tuo tiesiškiau jame silpnėja šviesa tarp sluoksnių, todėl interpoliuojant tarp sluoksnių esamas reikšmes svarbu, kad jos būtų interpoliuojamos tiesiškai, būtent toks ir buvo tikslas gilių dengiamumo žemėlapių metodo. Tačiau jeigu objektas stambus, tai reikšmės paprastai tarp sluoksnių pasiskirsto chaotiškai, dėl to nėra prasmės išlaikyti sluoksnių kontūrą pagal gylio buferį.

Prieš pradedant piešimą į sluoksnius turi būti išjungtas gylio buferis, ir įjungtas suliejimas *(blend)* su tokiais parametrais, kad prie esamos vaizdo buferyje *(frame buffer)* reikšmės,pridėtų nauja reikšmę. Tam atlikti naudojama *glBlendFunc(GL_ONE,GL_ONE)*, Mus domina tik matomi paviršiai, nes šviesos sugėrimas neturi padidėti išėjus šviesai iš objekto, dėl to turime įjungti galinių paviršių šalinimą *(glCullFace).*

Dengiamumo sluoksnai piešiami iš šviesos šaltinio pozicijos ir juose rašoma sulaikytos šviesos kiekis kiekviename sluoksnyje ir į alfą kanalą įrašome atstumą nuo plokštumos iki šviesos šaltinio. Alfa kanalas reikalingas, kad galėtume sužinoti kada šviesa atsitrenkė į paviršių ir užstojo šviesą. Žinodami tikslų atstumą iki plokštumos, galėsime tiksliau interpoliuoti sluoksniuose esamas reikšmes. Skirtingai nei dengiamumo žemėlapiuose, ar gylio dengiamumo žemėlapiuose, rašoma informacija būtent tik į tą sluoksnį, kuriame taškas ir priklauso, nereikia rašyti prieš jį esančių sluoksnių, nes gali išsigadinti informacija. (4)

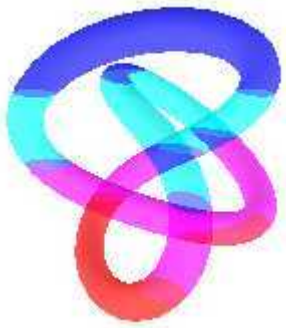
Įrodymas: (4)

Tarkime yra 4 sluoksniai, juos kerta plokštumos, ir paleistas šviesos spindulys sluoksniuose susilpnėja atitinkamai po 0.5, 0.4, 0.3, 0,4 praeidamas per kiekvieną sluoksnį.

Gylio dengiamumo žemėlapiuose ir dengiamumo žemėlapiuose sluoksniuose gausime tokias reikšmes: S1=0.5; S2=S1+0.4=0.9; S3=S2+0.3=1.0 (reikšmės gali būti [0..1] intervale), S4=S3+0.4=1.0. Kaip matome jau trečiame sluoksnyje praradome dalį informacijos, nekalbant apie sekančius po jo sluoksnius, taigi nebegalime atkurti informacijos atgal, ir tinkamai interpoliuoti reikšmes.

Čia rašomas algoritmas į sluoksnį saugo tik tam sluoksniui reikalingas reikšmes. Todėl jei norime sužinoti kiek šviesa susilpnėjo atėjusi iki 4 sluoksnio, tiesiog susumuojame visus sluoksnius iki jo.

Priklausomai nuo realizavimo ir sluoksnių skaičiaus, piešiama gali būti vieną kartą, ir apskaičiuojami sluoksniai viršūnių ir fragmentų paprogramėse, arba piešiama tiek kartų, kiek yra sluoksnių nukertant plokštumas naudojant standartinę OpenGL funkcija *glClipPlane.*



3.1.pav. Objektas suskaidytas į keturis sluoksnius, kiekvienas jų pažymėtas skirtinga spalva

3.2.pav.Sluoksniuose esančios šviesos pralaidumo reikšmės, (dešinėje pirmas sluoksnis, kairėje ketvirtas).

Dengiamumo sluoksnių piešimas

Priklausomai nuo algoritmo realizavimo ir sluoksnių skaičiaus, piešiama vieną kartą arba daugiau ir turint dengiamumo sluoksnius taško spalva apskaičiuojama fragmentu paprogramėje *(fragment shader)*. Šis piešimas nereikalauja jokių papildomų nustatymų išskyrus, kelių konstantų perdavimą grafinėm paprogramėm, ir yra pilnai apskaičiuojamas grafinėje plokštėje.

Taškas bus šešėlyje jeigu, visų sluoksniuose esamų reikšmių ilgių suma didesnė už 0. (4)

$$f(\vec{p}) = \sum_{n} length(clr(\vec{p})_i) \qquad (4)$$

$\vec{p}$ konkretus taškas, n – sluoksnių skaičius, $clr(\vec{p})_i$ - spalva į-tame sluoksnyje, taške $\vec{p}$, *length* – funkcija gražinanti vektoriaus ilgį.

Viršūnių paprogramė apskaičiuoja paprastą difuzinę šviesos dedamąją (Lamberto BRDF) ir prideda ambient šviesos konstantą, paskaičiuoja projektuotas tekstūrų koordinates, dėl teisingų reikšmių paieškų tarp dengiamumo sluoksnių, taip pat paskaičiuojamas atstumas nuo taško, iki šviesos šaltinio plokštumos, kad būtų galima nustatyti kuri sluoksnį pasirinkti, ir kaip interpoliuoti gautas reikšmes.

Fragmentų paprogramėje pirmiausia nustatoma ar fragmentas yra šešėlyje (4), jeigu fragmentas nėra šešėlyje tai priskiriama difuzinės ir ambient spalvos reikšmė, kuri buvo paskaičiuota viršūnių paprogramėje. Jeigu fragmentas yra šešėlyje, tai pažiūrime kokioje vietoje jis yra tarp artimos ir tolimos objekto atkirtimo plokštumos, pagal tai galima nuspręsti kokiame sluoksnyje yra fragmentas.

Žinodami kokiame sluoksnyje yra fragmentas, žiūrime ar to sluoksnio alfa reikšmė nelygi 0. Jeigu alfa > 0 pridedame prieš jį esančių sluoksnių reikšmes ir interpoliuojame esamą sluoksnį pagal skirtumą, fragmento koordinatės ir nuskaitytos alfa reikšmės, padaugintą iš tiek, kiek išviso yra sluoksnių. Jeigu alfa = 0, grįžtame į aukštesnį sluoksnį, ir atliekame tuos pačius veiksmus, kaip su prieš tai buvusiu sluoksniu. Vykdome tol, kol randame alfa > 0, arba kol yra sluoksnių.

Galiausiai atimame rastą reikšmę iš plokštumos taško natūralios spalvos, nes ši reikšmė yra sugertos šviesos intensyvumas.

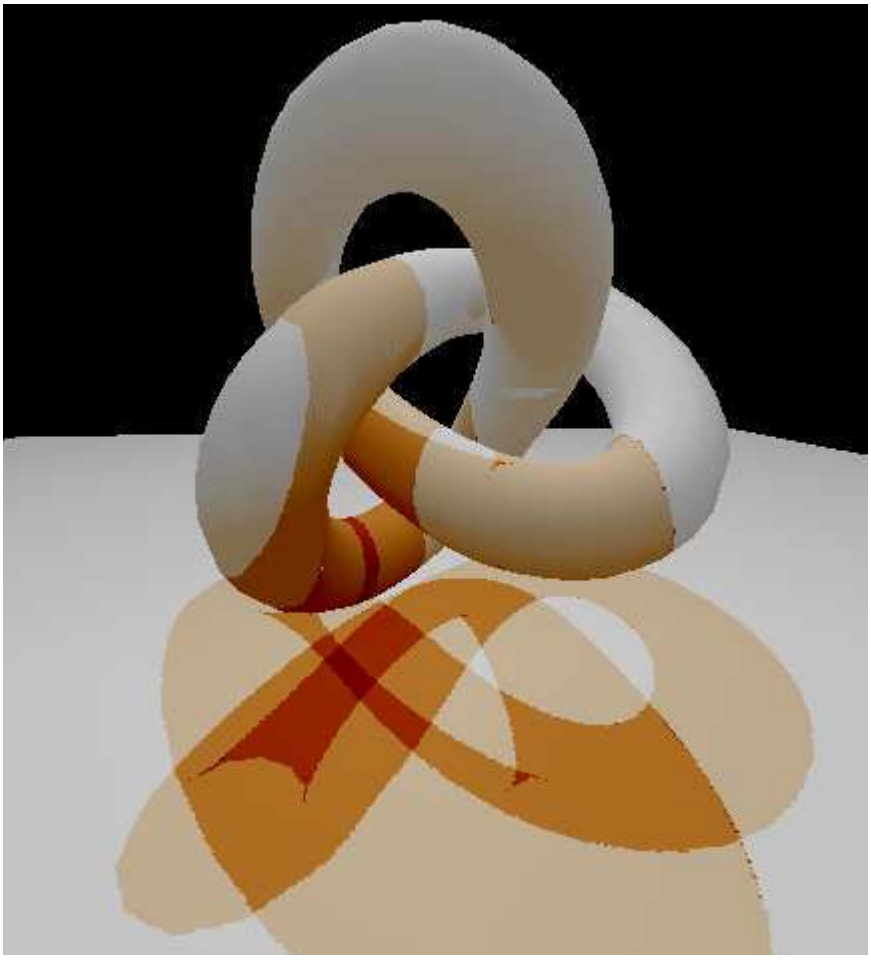
Fragmentų paprogramės pseudo kodas, kai yra keturi dengiamumo žemėlapių sluoksniai:

1. Nuskaitom visas reikšmes R_i iš visų sluoksnių N taške $\vec{p}$.
2. Susirandame *interpolate,* jis nusako taško $\vec{p}$ atstumą.
3. *Jeigu* ($\sum_N length(R_i) > 0$)
    a. Randame kuriame sluoksnyje esame,- *layer.*
    b. *Ciklas (layer >= i >=0)*
        i. *Jeigu* ($R_i.alfa \neq 0$)
            1. $spalva = R_i * min((interpolate - R_i.alfa) * N, 1.0);$

               $spalva += \sum_i Rj$
    c. Iš objekto spalvos atimame *spalva*
4. Taškas nėra šešėlyje nieko skaičiuoti nereikia

Reiktu paminėti, kad fragmentų paprogramės neleidžia naudoti vartotojų sukurtų vektorinių masyvų, tik tekstūrų koordinačių masyvus, taigi realus paprogramės kodas bus gerokai didesnis.

Antrame žingsnyje *interpolate* kintamasis turi būti paskaičiuojamas lygiai taip pat, kaip sluoksnių generavimo etape buvo paskaičiuojama alfa reikšmė, kadangi mes lyginame atstumus tarp *interpolate* ir *alfa* reikšmės sluoksnyje, kad galėtume teisingiau interpoliuoti šviesos susilpnėjimą sluoksnyje. *Interpolate* reikšmė visame yra intervale [0..1], kad maksimaliai būtų išnaudojami 8 alfa bitai tekstūroje. Funkcija $min((interpolate - R_i.alfa) * N, 1.0)$ yra interpoliavimo funkcija. $(interpolate - R_i.alfa)$ yra atstumų skirtumas tarp taško $\vec{p}$ ir artimiausios prieš šį tašką plokštumos. Kadangi objektas suskaidytas į N sluoksnių ir yra intervale [0..1], tai vieno sluoksnio intervalas yra lygus $1/N$. Kiekvienas sluoksnis saugo praleistos šviesos kiekį tik savajame sluoksnyje, taigi norint tiesiškai interpoliuoti tarp sluoksnio i ir $i+1$ gautą atstumų skirtumą turime padauginti iš N, tokiu būdu sluoksnyje interpoliacijos koeficientas tampa lygus 1. Algoritmas nenumato kada spindulys išeina iš objekto, t.y. kerta galinę plokštumą, tai reikia numatyti, kad spindulys gali kirsti plokštumą i-tajame sluoksnyje, ir dar kartą kirsti $i+j$; $j>0;$ tokiu atveju imant skirtumą tarp *interpolate ir* $R_i.alfa$, gali gautis j kartų didesnis atstumas, negu galimas tarp sluoksnių $(1/N)$, taigi naudojame funkcija *min,* kad nesusilpnintume per daug šviesos spindulio. Apskaičiavus spalvą i-tajame sluoksnyje, prie gautos spalvos pridedame prieš i-tajį sluoksnį esančių sluoksnių spalvas. Kintamasis *spalva* rodo, kiek būtent buvo sugerta šviesos, taigi iš taško $\vec{p}$ spalvos atimame kintamąjį *spalva.*



3.3.Objektas praleidžiantis auksinę spalvą interpoliuojamos reikšmės pagal alfa reikšmę, 4 sluoksniai.

Algoritmo realizacija

Algoritmo realizacijoje yra naudojama OpenGL biblioteka. Algoritmo abu žingsniai piešimai yra atliekami vieną kartą, naudojami 4 sluoksniai. Kadangi aplikacija veikia ne viso ekrano *(full-screen)* režime, tai alfa bitai, pagrindiniame kadro buferyje *(framebuffer),* yra išjungti ir norint kopijuoti į tekstūra duomenis reikia susikurti kadro buferio objektus *(framebuffer object),* be to tai leidžia padidinti ir tekstūrų dydį.

Pirmam piešimui (į sluoksnius) pasinaudojama *framebuffer object* ir vienu metu rašoma į 4-turis aktyvius kadro buferio objektus, prie kuriu yra pririštos 4 sluoksnių tekstūros. Reiktų atkreipti dėmesį į

tai, kad tekstūrų tolimas *(magnification)* ir artimas *(minification)* filtrai [8] turi būti taškinis *(point sampling)*. Kitu atveju vienos ir kitos reikšmės gali pasikartoti keliuose sluoksniuose, dėl ko, skaičiuojant atsiranda nepageidaujami netikslumai. (5.1. paveikslėlis prieduose.)

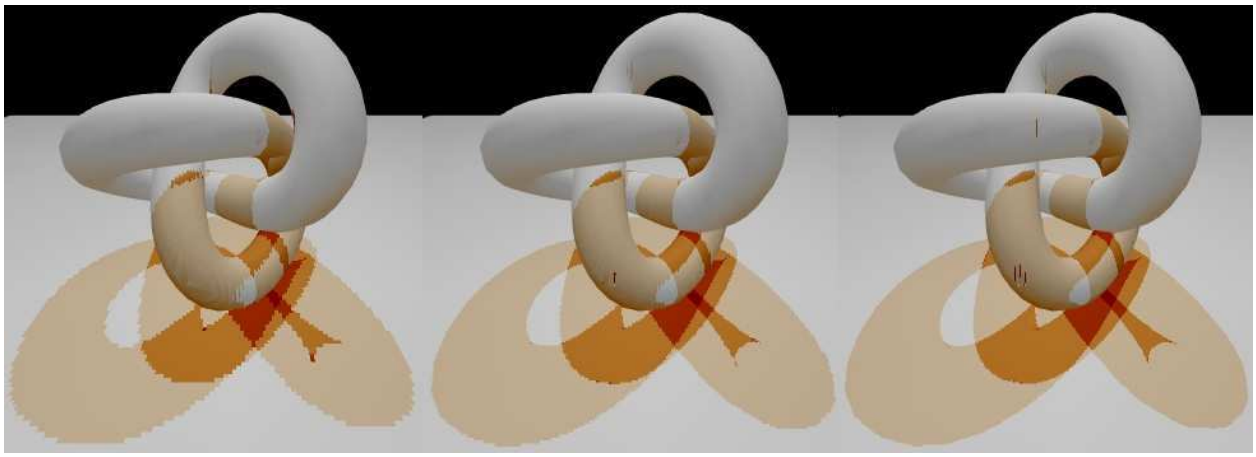
Viršūnių paprogramė perduoda spalvą, kuria objektas sugeria, ir apskaičiuoja objekto koordinates, kurios yra intervale [0..1] per visą objektą. Fragmentų paprogramė pagal perduotą atstumą atsirenka kuriam sluoksniui priklauso tas taškas ir į jį įrašo spalvą ir atstumą į alfą buferį. Piešimo metų įjungtas suliejimas.

Antram piešimui (galutinis piešimas) pirmiausia sutvarkoma tekstūrų matrica. Kadangi pirma kartą objektas buvo piešiamas iš šviesos šaltinio pozicijos, tai tekstūrų matrica iš šviesos šaltinio erdvės reikia perkeisti į pasaulio erdvę (eye space) tai atliekama dauginant projekcijos matricą iš pasaulio matricos (modelview) ir iš invertuotos šviesos šaltinio matricos. Tokiu būdu gauname tekstūrų koordinates pasaulio erdvėje. Įjungiame visas tekstūras (multitexturing), priskiriame nesikeičiančius kintamuosius (uniform) vaizdo paprogramėms ir piešiame visą sceną.

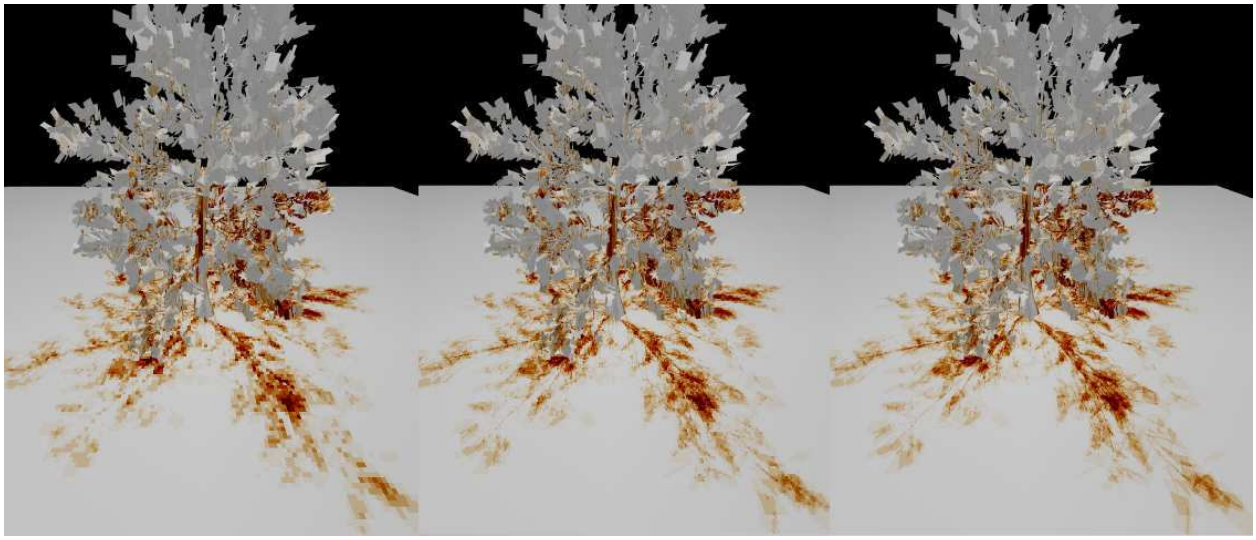
Viršūnių paprogramė apskaičiuoja atstumą, tokiu pat būdu, kaip ir pirmame žingsnyje, kad galėtume tarpusavyje sulyginti rezultatus. Apskaičiuojamas paprastas Lamberto difuzinis apšvietimas, bei paskaičiuojamos tekstūrų koordinatės. Fragmentų paprogramė pagal anksčiau aptartą algoritmą paskaičiuoja galutinio fragmento tašką.

Rezultatai

Realizacijoje naudojami 4-turi sluoksniai 1024x1024 dydžio, R8G8B8A8 formato, tekstūrų. Rezultatuose pateikiamos dvi scenos: vieną sudaro 2912 trikampiai, kitą 25646. Algoritmo spartos ir kokybės analizėj taip pat buvo parinktos 256x256 ir 512x512 dydžio to pačio formato tekstūros. Žemiau pateikti rezultatai 3.4.pav. esant skirtingų dydžių tekstūroms, iš kairės į dešinę atitinkamai naudojama 256x256, 512x512, ir 1024x1024 dydžio tekstūros.



3.4.Scena sudaro 2912 trikampiai, kairėje pusėje 256x256 dydžio tekstūra, per vidurį 512x512, dešinėje 1024x1024.



3.5.Scena sudaro 25646 trikampiai, kairėje pusėje 256x256 dydžio tekstūra, per vidurį 512x512, dešinėje 1024x1024

Dengiamumo žemėlapių vientisiems kūnams spartos rezultatai. 1. Lentelė

Sluoksnių tekstūrų raiška	Kadrų per sekundę skaičius 640*480 rezoliucija	
	Scena 1 (2916 trikampiai)	Scena2 (25646)
256x256	86	14
512x512	81	14
1024x1024	66	14

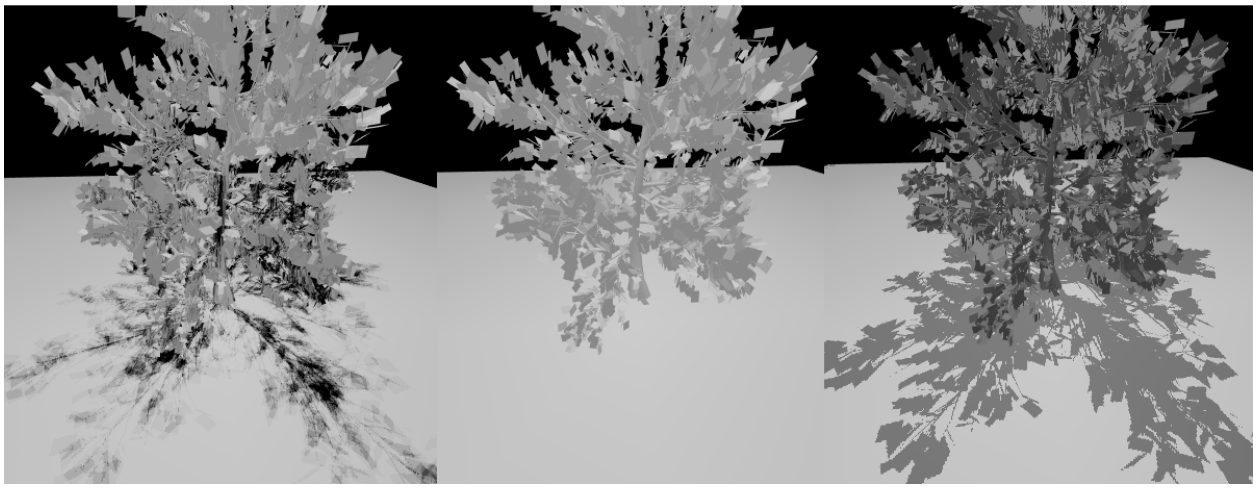
Lentele. 1

Esant mažam trikampių skaičiui įtakos turėjo tekstūrų dydis ,stabdė fragmentų operacijos, tačiau esant dideliam trikampių skaičiui pradėjo stabdyti viršūnių operacijos.

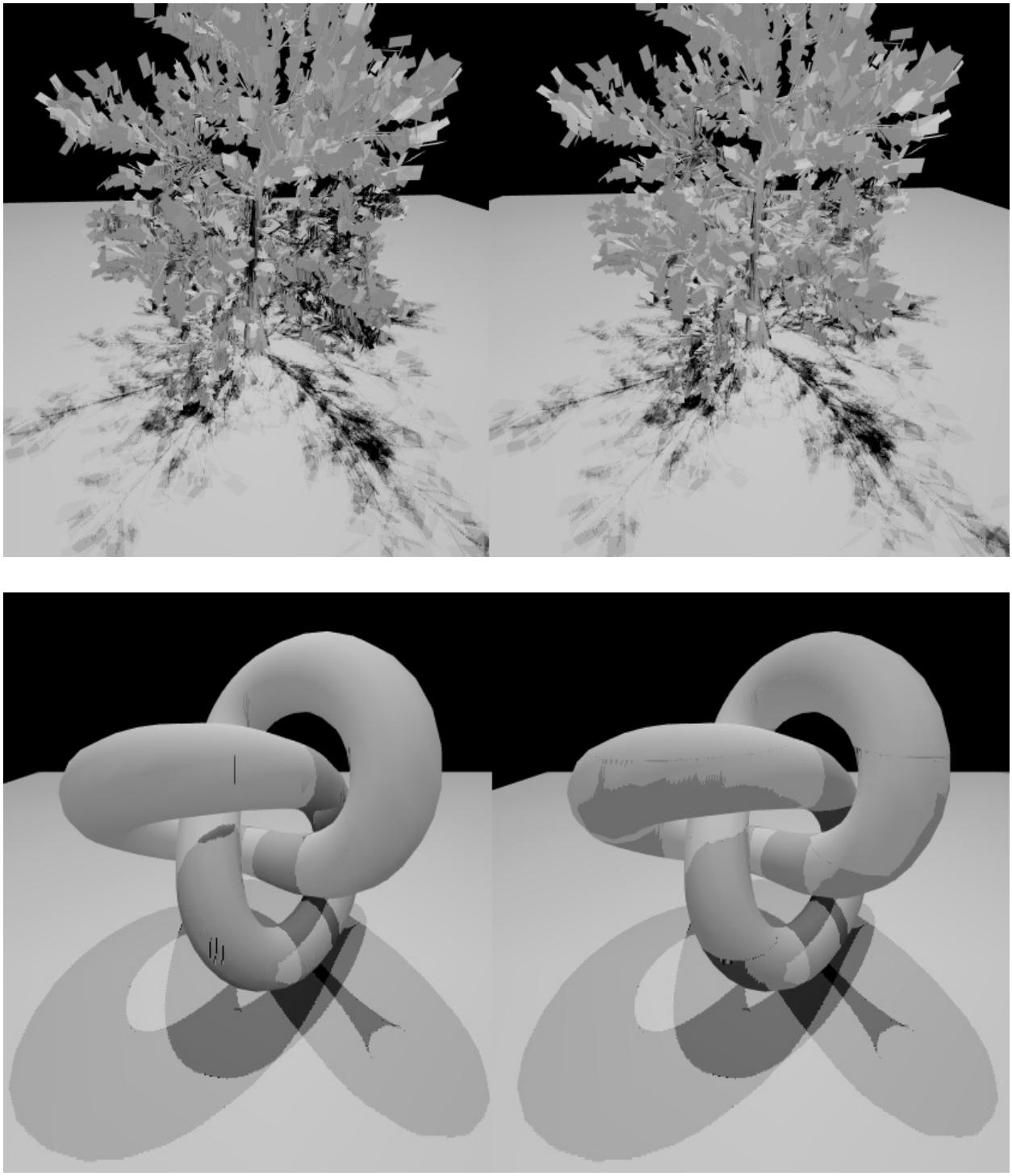
Algoritmo aptarimas

Algoritmo įvertinimui buvo realizuotas tradicinis gylio tekstūros šešėlių algoritmas, ir dengiamumo žemėlapių algoritmas. Gauti vaizdai pateikiami 3.6, 3.7. paveikslėliuose.

Tradicinių gylio tekstūros žemėlapių apšvietimas labai apribotas, visose vietose šešėlio stiprumas vienodas, dėl to vaizdas atrodo nenatūraliai lyginant su dengiamumo žemėlapiais vientisiems kūnams. 3.7. paveikslėlyje žiūrint į čia aprašyta metodą, ir dengiamumo žemėlapių metodą, gaunamas vaizdas labai panašus, ypač pirmuose sluoksniuose, tačiau apatiniai sluoksniai atrodo šiek tiek „švelniau“ dengiamumo žemėlapiuose. Tačiau ten kur objektas vientisus, ir šešėlių tikroviškumą galima lengvai įvertinti akimis čia aptartas metodas atrodo žymiai geriau, nors irgi yra klaidų.



3.6.Nuo kairės į dešinę: dengiamumo žemėlapiai vientisiems kūnams, vaizdas tik su difuziniu Lamberto apšvietimu, gylio tekstūrų žemėlapiai.



3.7.Nuo kairės į dešinę: dengiamumo žemėlapiai vientisiems kūnams, dengiamumo žemėlapiai

Pažiūrėjus į 3.4. paveikslėli pastebime įdomų dalyką. Esant aukštai rezoliucijai atsiranda nepageidaujami brūkšneliai. Taip nutinka dėl diskretizavimo, rašant kadro buferio reikšmes į tekstūrą.

Tokio tipo „juodų“ brūkšnelių galima išvengti pakeičiant tekstūrų filtrus, tačiau tuomet atsiras sluoksnių persidengimo klaidos (4.1.pav. prieduose). Norint visiškai išvengti šių klaidų reiktų pertvarkyti tekstūras užpildant realiomis reikšmėmis tas zonas, kuriose spalva lygi 0, ir aplinkui yra bent du tekseliai [8].

Išvados

1. Darbe pasiūlytas realaus laiko metodas, vaizduoti pusiau peršviečiamus šešėlius vientisiems kūnams. Kadangi šešėliams saugoti imamos trys spalvų komponentės, algoritmas lengvai pritaikomas spalvotiems šešėliams vaizduoti.
2. Pateiktas algoritmas, esant ribotam sluoksnių skaičiui, gali pavaizduoti galutinį vaizdą per keletą piešimų. Tai įgyvendinama naudojant vaizdo buferio objektus.
3. Įrodyta, kad į konkretų sluoksnį geriau rašyti tik jam priklausančią informaciją, nepridedant prieš jį esančiuose sluoksniuose esamų reikšmių.
4. *Dengiamumo žemėlapiai stambiems kūnams* iš dalies išsprendžia sluoksniuotumo klaidas (reikšmėms interpoliuoti naudojant alfa kanalą), tačiau ne taip tiksliai interpoliuoja jeigu tarp vieno sluoksnio pasitaiko kelios ir daugiau matomos plokštumos.
5. Algoritmas pagrindinius skaičiavimus atlieka naudodamas vaizdo paprogrames. Perkėlus statinę geometriją į video atmintį (VRAM), CPU būtų praktiškai nenaudojamas.

Literatūra

[1]. **T. Heidmann**, 1991. *Real shadows real time*. IRIS Universe, 18, 28–31.

[2]. **J. Carmac,** http://www.rasterise.com/CarmackOnShadowVolumes.txt (žiūrėta 2008.05)

[3]. **C. Everitt ir M. J. Kilgard,** skaidrės iš GDC 2002, http://developer.nvidia.com/object/robust_shadow_volumes.html (žiūrėta 2008.05)

[4]. **E. Haines,** Soft Planar Shadows using Plateaus. *Journal of Graphics Tools, 6(1) tomas*. 2001, 19-27 p.

[5]. **T. Lokovic ir E. Veach,** *SIGGRAPH 2000 Proceedings* (August 2000), Addison-Wesley. http://graphics.stanford.edu/papers/deepshadows/ (žiūrėta 2008.05)

[6]. **T.-Y. KIM, ir U. NEUMANN** Opacity shadow maps. In *12th Eurographics Workshop on Rendering Techniques* (2001), pp. 177–182.

[7]. **C. Yuksel ir J. Keyser**, „Deep Opacity Maps“ *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2008), 2008*

[8] **M.Vinkelis,** *Tekstūrų filtrai.* 2004. VDU konferencija.

[9] **B. T. Phong,** *Illumination for Computer Generated Pictures*. Communications of the ACM

[10] **H. Zhang,** *Forward shadow mapping, Rendering Techniques '98*, Vol. 9, pp. 131-138, Springer-Verlag, 1998.

[11] **J. Foley, A. Van Dam, S. K. Feiner, ir J. F. Hughes,** *Computer graphics, principles and practice, Second Edition,* Addison-Wesley, July, 1995

[12] **A. S. Glassner,** *An introduction to ray tracing*, Academic Press, 1993

[13] **J. Kajiya ir B. P. Herzen,** *Ray tracing volume densities,* SIGGRAPH Proceedings, Vol. 18, pp. 165-174, 1984.

[14] **W. T. Reeves, D. H. Salesin, ir R. L. Cook,** *Rendering antialiased shadows with depth maps,* SIGGRAPH Proceedings, Vol. 21, pp. 283-291, 1987.

Priedai

Paprogramių tekstai naudoti algoritmui.

Programa realizuota su *OpenGL* grafine biblioteka, todėl bus pateikiamos *GLSL* kalba parašytos vaizdo spartintuvo programos, - paprogramės.

Pirmas piešimas: informacijos rašymas į dengiamumo žemėlapius. Prieš piešiant reikia prijungti frame buffer objektą, prie kurio prisektos 4 tekstūros, ir nustatyti 4 piešiamus spalvų buferius *GL_COLOR_ATTACHMENT0_EXT+i; i=0..4;*, su *glDrawBuffers* komanda. Taip pat išjungti gylio testą ir įjungti suliejimą.

Viršūnių paprogramė (pirmam piešimui): priskiria spalvą, ir apskaičiuoja viršūnės atstumą objekto intervale.

```
uniform float ClipNear;
uniform float ClipFar;
uniform vec3 LightDir;
varying float vertOffset;

void main()
{
	vec4 worldPos = gl_ModelViewMatrix * gl_Vertex;
	vec3 worldPos3 = (vec3(worldPos)) / worldPos.w;
	vec3 offset = worldPos3 - gl_LightSource[0].position.xyz;
	float Depth = -dot(offset, normalize(gl_NormalMatrix *
LightDir));
	vertOffset = length(LightDir) + Depth - ClipNear;
	vertOffset = 1.0-vertOffset/(ClipFar-ClipNear);

	gl_Position = ftransform();
	gl_ClipVertex = gl_ModelViewMatrix * gl_Vertex;

	gl_FrontColor = gl_Color;
}
```

Fragmentų paprogramė (pirmam piešimui): pagal paskaičiuota atstumą įrašo spalvą į atitinkamą buferį, ir į alfą kanalą išsaugo atstumą.

```
varying float vertOffset;
void main()
{
  //0-virsune, 1-apacia
	gl_FragData[1]=vec4(0.0);
	gl_FragData[2]=vec4(0.0);
	gl_FragData[3]=vec4(0.0);
	if (vertOffset <= 0.25) {
```

```
            gl_FragData[0]=vec4(gl_Color.rgb,vertOffset);
      } else if (vertOffset <= 0.5) {
            gl_FragData[1]=vec4(gl_Color.rgb,vertOffset);
      } else if (vertOffset <= 0.75) {
            gl_FragData[2]=vec4(gl_Color.rgb,vertOffset);
      } else if (vertOffset <= 1.0) {
            gl_FragData[3]=vec4(gl_Color.rgb,vertOffset);
      }
}
```

Antras piešimas: labai mažai pasiruošimų, tereikia sutvarkyti tekstūrų matricą (aprašyta algoritmo realizacijoje), ir priskirti reikiamus *uniform* tipo kintamuosius.

Viršūnės paprogramė (antram piešimui): apskaičiuoja Lamberto difuzinį apšvietimą, ir randa viršūnės atstumą, tokiu pat principu kaip ir pirmame piešime.

```
//varyingai perduodami fragment shaderiui
varying vec4 projCoord;
varying float vertOffset;

uniform float ClipNear;
uniform float ClipFar;
uniform float DepthNear;
uniform float DepthFar;
uniform vec3 LightDir;


// ambient ir diffuse faktoriai
const float Af = 1.0 / 1.5;
const float Df = 1.0 / 3.0;

void main()
{
  //virsunes pozicija world erdvej
  vec4 worldPos = gl_ModelViewMatrix * gl_Vertex;
  vec3 worldPos3 = (vec3(worldPos)) / worldPos.w;
  vec3 lightVec = vec3(gl_LightSource[0].position) - worldPos3;
  //normalizuotas vektorius is virsunes i sviesos saltini
  lightVec = normalize(lightVec);
  vec3 normal = normalize(gl_NormalMatrix * gl_Normal);
  //dotproduktas normales ir ligthVec (
  float difIntensity = max(0.0, dot(normal, lightVec));
  //galutinis spalvos faktorius
  float factor = min(1.0, Af + difIntensity * Df);

  //is sviesos saltinio i akis suprojektuotos matricos
  //ir konkretaus pasaulio tasko sandauga
  projCoord = gl_TextureMatrix[0] * worldPos;

  //atstumas tarp tasko ir sveiso saltinio
  vec3 offset = worldPos3 - gl_LightSource[0].position.xyz;
```

```
  //atstumas nuo tasko iki svies. salt. plokstumos (plokstumos normale
- lightDir)
  float Depth = -dot(offset, normalize(gl_NormalMatrix * LightDir));
    //siaip tai cia reiktu imti lightPosition, bet
gl_lightsource.position paveiktas modelview matricos
  vertOffset = length(LightDir) + Depth - ClipNear;
  vertOffset = vertOffset/(ClipFar-ClipNear);
  gl_Position = ftransform();
  gl_FrontColor = vec4(factor*gl_Color.rgb, gl_Color.a);
}
```

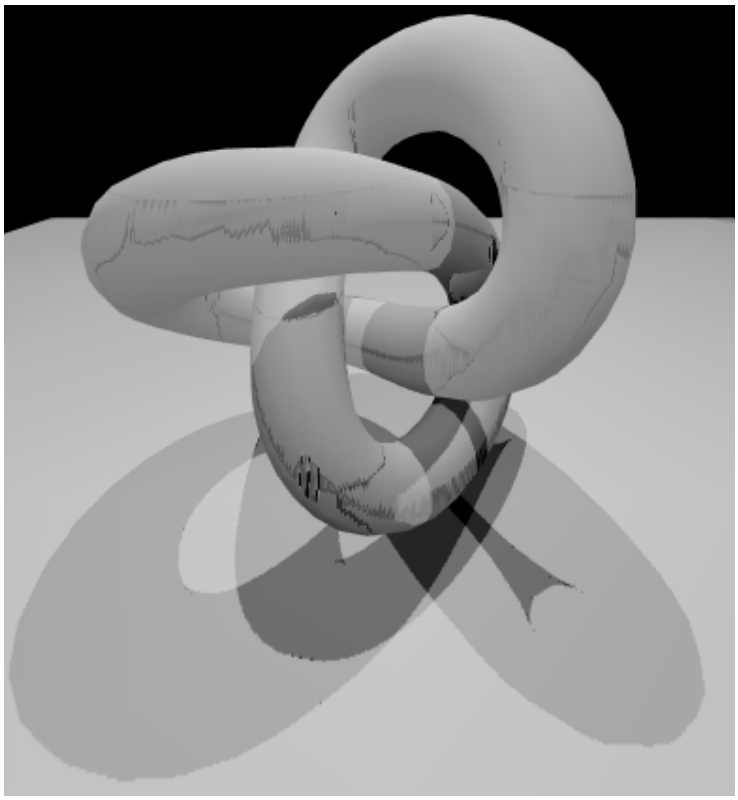
Fragmentų paprogramė (antram piešimui): aiskaičiuoja galutinę taško spalvą, algoritmas aptartas algoritmo aprašyme.

```
uniform sampler2D ShadowTex;
uniform sampler2D OpacityTex1;
uniform sampler2D OpacityTex2;
uniform sampler2D OpacityTex3;
uniform sampler2D OpacityTex4;

uniform int OPACITY_MAPS;

varying float vertOffset;
varying vec4 projCoord;
const float intClr=4.0;
const float intTctl=1.0;

void main()
{
  vec4 tc1 = texture2DProj(OpacityTex1, projCoord);
  tc1.rgb=tc1.rgb*intTctl;
  vec4 tc2 = texture2DProj(OpacityTex2, projCoord);
  tc2.rgb=tc2.rgb*intTctl;
  vec4 tc3 = texture2DProj(OpacityTex3, projCoord);
  tc3.rgb=tc3.rgb*intTctl;
  vec4 tc4 = texture2DProj(OpacityTex4, projCoord);
  tc4.rgb=tc4.rgb*intTctl;
  float interpolate = 1.0-vertOffset;
  vec4 clr=vec4(0.0,0.0,0.0,0.0);
  float i;
  if (length(tc1+tc2+tc3+tc4) > 0.0) {
      if (interpolate < 0.25) {
      clr=tc1*min((interpolate-mod(tc1.a,0.26))*intClr,1.0);
    } else if (interpolate < 0.5) {
           if (tc2.a != 0.0) {
                clr=tc1+tc2*min((interpolate-
mod(tc2.a,0.51))*intClr,1.0);
           } else
      clr=tc1*min((interpolate-tc1.a)*intClr,1.0);
    } else if (interpolate < 0.75) {
           if (tc3.a != 0.0) {
```

```
      clr=tc1+tc2+tc3*min((interpolate-mod(tc3.a,0.76))*intClr,1.0);
           } else
           clr=tc1+tc2*min((interpolate-mod(tc2.a,0.76))*intClr,1.0);
    } else if (interpolate < 1.0) {
           if (tc4.a != 0.0) {
      clr=tc1+tc2+tc3+tc4*min((interpolate-
mod(tc4.a,1.01))*intClr,1.0);
           } else
           clr=tc1+tc2+tc3*min((interpolate-
mod(tc3.a,1.01))*intClr,1.0);
    } else
      clr=tc1+tc2+tc3+tc4;
     }
  clr=gl_Color-smoothstep(vec4(0.0),vec4(1.0),clr);
  gl_FragColor = vec4(clr.rgb,1.0);
}
```

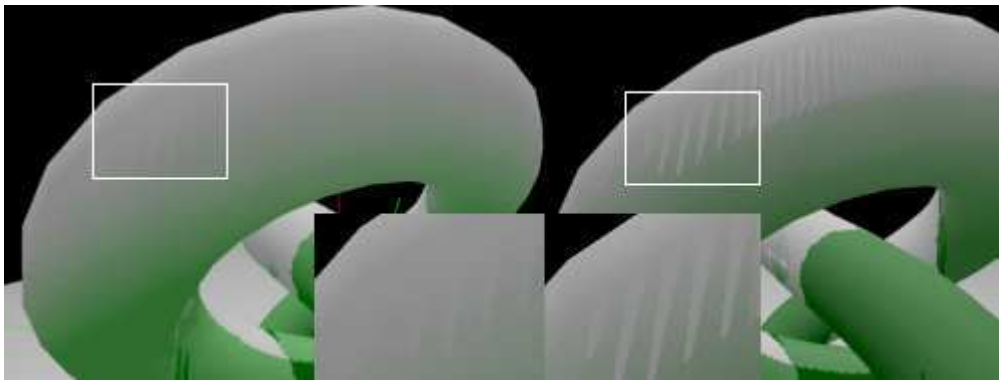
Įdomūs paveikslėliai

Pakeistos tekstūrų filtravimo būdas i bi-linearų filtrą. Matomi persidengimai dėl filtravimo klaidų, kai sluoksniu kontūruose sutampa reikšmės.



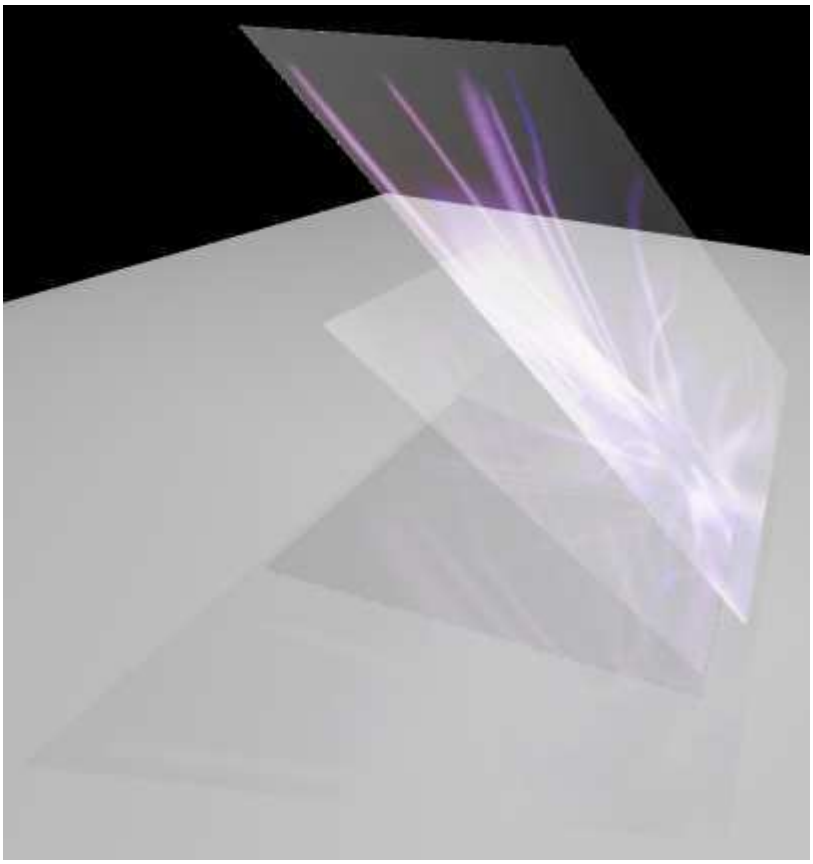
4.1.pav.Rezultatai su bi-liniariu tekstūrų filtru.

Smoothstep() funkcija. Parenka švelnų perėjimą. Funkcija priima tris argumentus, du kraštus (kampus), link kuriu turi švelniai pereiti perduodant trečia parametrą, reikšmę kuria norima sušvelninti.



4.2. pav. Iliustracija kairėje, naudoja smoothstep(), ir tiesiog paprastai atima prasklidusios šviesos kiekį (dešinėje)

Vitražinio stiklo simuliacija. Ant antro stiklo lakšto truputį matosi per pirmą lakštą praėjusi šviesa. Antras stiklas yra visiškame po pirmo stiklo šešėliu, ir dėl to yra tamsesnis.



4.3. pav. Piešimas su suliejimu, papildomai paskaičiuojant atspindį