KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
MULTIMEDIJOS INŽINERIJOS KATEDRA

Mindaugas Brasas

Šešėlių tūrių ir šešėlių planų metodų realizacija ir tyrimas

Magistro darbas

Darbo vadovas
doc. dr. A. Lenkevičius

Recenzentas
prof. Vacius Jusas

Kaunas
2010

TURINYS

PRATARMĖ 4
ĮVADAS 5
1. TEORINĖ DALIS 7
1.1. Šešėliai kompiuterinėje grafikoje apžvalga 7
1.1.1. Šviesos šaltiniai 8
1.1.2. Šešėliavimas 11
1.1.3. Šešėliai 14
1.2. Realaus laiko šešėlių technikos 17
1.2.1. Šešėlių tūrio algoritmas 17
1.2.2. Šešėlių plano algoritmas 20
1.2.3. Silueto plano algoritmas 25
1.2.4. Efektyvus hibridinis šešėlių vaizdavimo algoritmas 29
1.3. Skyriaus išvados 32
2. TIRIAMOJI DALIS 33
2.1. Šešėlių tūrių realizacija ir jos problemos. 33
2.1.1. Realizacija 33
2.1.2. Testavimas ir išvados 42
2.2. Šešėlių plano realizacija ir jos problemos 45
2.2.1. Realizacija 45
2.2.2. Testavimas ir išvados. 50
2.3. Metodų palyginimas 54
IŠVADOS 57
LITERATŪRA 58
Shadow volumes and shadow mapping realization and analysis summary 59
TERMINŲ IR SANTRUMPŲ ŽODYNAS 60
1 PRIEDAS. Šešėlių tūrius realizuojančios programos kodas 61
2 PRIEDAS. Šešėlių planus realizuojančios programos kodas 76

Lentelių sąrašas

1.2.1. lentelė Šešėlių metodų našumo palyginimai 28

2.2.1. lentelė Šešėlių planų greičio priklausomybė nuo šešėlių raiškos 52

2.3.1. lentelė Šešėlių planų ir šešėlių tūrių greičių palyginimas 54

Paveikslėlių sąrašas

1. pav. Pavaizduotos paprasto tiesioginio bei *radiosity* apšvietimo algoritmų scenos 28

1.1. pav. Apšviestos ir neapšviestos trimatės figūros 7

1.1.1. pav. Žibintinis šviesos šaltinis 10

1.1.2. pav. Warn‘o modelio situacija 10

1.1.3. pav. Sfera skirtingu padengimu atvaizduota naudojant plokščiąjį šešėliavimą 11

1.1.4. pav. Sfera skirtingu padengimu atvaizduota naudojant interpoliuotą šešėliavimą 11

1.1.5. pav. Skenavimo linijos technika 12

1.1.6. pav. Vaizduojamas išlenktas paviršius ir trikampis, kuris aproksimuoja dalį šio paviršiaus 13

1.1.7. pav. Idealaus atspindžio apskaičiavimas 15

1.1.8. pav. Šešėlis ant objekto 15

1.2.1. pav. Kadras iš žaidimo DOOM 3. Geras šabloninių šešėlių tūrių pavyzdys 17

1.2.2. pav. Šešėlio tūrio perteikimas: skaičiuojant įeinančius ir išeinančius spindulius iš žiūrėjimo taško 18

1.2.3. pav. Šviesos spinduliai išvesti pagal šviesą dengiančio kūno kontūrus 20

1.2.4. pav. Šešėlio tūrio dangčiai. 20

1.2.5. pav. Šešėlių plano algoritmo vizualizacija 21

1.2.6. pav. Poveikis dėl paklaidos kontrasto. 22

1.2.7. pav. Šešėlio plano *aliasing‘o* formalizacija 23

1.2.8. pav. Procentiškai priartėjančio filtravimo principas 24

1.2.9. pav. Procentiškai priartėjančio filtravimo poveikis 24

1.2.10. pav. Perspektyvinių šešėlių planai 25

1.2.11. pav. Visos skirtingos kombinacijos gylio patikrinimo rezultatų ir šešėliavimo konfigūracijų. 27

1.2.12. pav. Testuojamos scenos 28

1.2.13. pav. Efektyvaus hibridinio šešėlių vaizdavimo algoritmo apibendrinimas 30

1.2.14. pav. 512 x 512 šešėlių planas ir aprašytas hibridinis algoritmas su 256 x 256 šešėlių planu 30

1.2.15. pav. Palyginimas pav. kokybės naudojant šešėlių planus, hibridinį algoritmą ir šešėlių tūrius 31

1.2.16. pav. Našumo palyginimas šešėlių planų, hibridinio algoritmo ir šešėlių tūrių 31

2.1.1. pav. Pradinė scena. 33
2.1.2. pav. Tūrio braižymas 34
2.1.3. pav. Po pirmo perėjimo 3
2.1.4. pav. Po antro perėjimo 37
2.1.5. pav. Scena su mūsų šešėliu 37
2.1.6. pav. Šviesos šaltiniui atsidūrus kitoj poligono pusėj, šešėlis dingsta 38
2.1.7. pav. Poligono viršūnės apjungiamos prieš laikrodžio rodyklę brėžiant viršutinį dangtį 38
2.1.8. pav. Koordinačių transformacija pastatant poligoną mums į tinkamą padėtį. 41
2.1.9. pav. Scena kai šviesos šaltinis toli. Dešinėj pusėj nupiešti tūrių kontūra 42
2.1.10. pav. Scena kai šviesos šaltinis prie pat. Dešinėj pusėj nupiešti tūrių kontūrai 42
2.1.11. pav. Tūrinių šešėlių kontūrai iš arti 43
2.1.12. pav. Šviesos šaltinis vamzdyje. Ši šešėliams sunki scena – ne kliūtis tūriniams šešėliams 43
2.1.13. pav. Šešėliai nepriklauso nuo nupieštų poligonų, bet tik nuo pažymėtų tūrių 44
2.2.1. pav. Funkcija glPolygonOffset panaudojimas. 47
2.2.2. pav. Šviesos regos lauko įtaka šešėliams 48
2.2.3. pav. Kai šviesos šaltinis tinkamoj padėty. 51
2.2.4. pav. Kai šviesos šaltinis yra toliau pradeda matytis artefaktai 51
2.2.5. pav. Kai šviesos šaltinis yra per arti šešėlius metančių objektų 52
2.2.6. pav. Šešėlių kokybės priklausomybė nuo šešėlių tekstūrų raiškos 53
2.3.1. pav. Šešėlių planų ir šešėlių tūrių palyginimas 55
2.3.2. pav. Šešėlių tūrių ir šešėlių planų palyginimas, kai šviesos šaltinis arti. 56
2.3.3. pav. Šešėlių tūrių ir šešėlių planų palyginimas, kai šviesos šaltinis toli 56

PRATARMĖ

Šiame darbe pristatysime du alternatyvius, vienus iš pačių populiariausių, realaus laiko šešėlių generavimo metodus. Tai šešėlių planų (angl. *shadow maps*) ir šešėlių tūrių (angl. *shadow volumes*) metodai. Kadangi šie metodai turi įvairių skirtingų privalumų ir trūkumų, yra nuolat tobulinami, kartais kuriami ir iš jų ir hibridiniai algoritmai, kurie perimtų kuo daugiau šių dviejų technikų gerųjų savybių. Todėl šiame darbe pamėginsime nustatyti ir įvertinti, kuriais atvejais yra vienas geresnis už kitą.

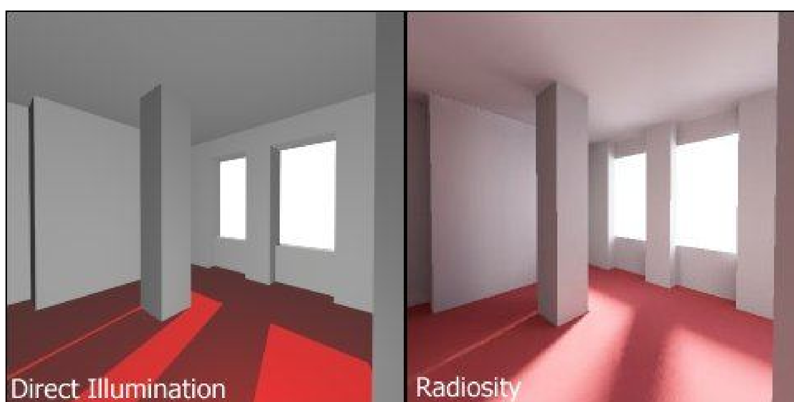
ĮVADAS

Paviršiaus išvaizdos apibūdinamas yra paveikslo sintezės šerdyje, ir sukurti įtikinamus paveikslus su kompiuteriu išlieka sudėtingas procesas, reikalaujantis pažangių specialių įgūdžių. Tikrovėje, išvaizda kontroliuojama sudėtinga šviesos sąveika (elektromagnetinis spinduliavimas ateina iš įvairiausių šaltinių) ir medžiagomis sudarančiomis objektus scenoje, su sudėtingais išsklaidymo efektais, kurie išsidėstę per visą šviesos kelią.

Fizikiniai dėsniai valdantys šį procesą buvo senai žinomi šviesos bangų ilgių skalėje. Ir dabar, kad būtų galima sukurti apskaičiuojamą modelį, kuris galėtų būti naudojamas kompiutery, reikalingas diskretus modelis.

Dabar kompiuteriai aprūpinti su galingais grafikos agregatais, atliekantys masiškai didelius lygiagrečius skaičiavimus viršūnių ir pikselių lygyje. Sudėtingi skaičiavimai reikalingi tinkamam vaizdavimui tapo prieinami realaus laiko taikymuose. Tuo pačiu tapo prieinamas ir realaus laiko šešėlių generavimas. Todėl viena iš kompiuterinės grafikos problemų, tapo realaus laiko šešėliai, nes be šešėlių mes niekad nesukursime realaus vaizdo.

Šiuo metu yra prikurta daugybė realaus laiko šešėlių generavimo algoritmų, ir jie nuolat tobulinami, kuriami nauji – bandoma išgauti kuo tikroviškesnį apšvietimą, kuo mažiau apkraunant kompiuterių resursus.



1. pav. Šiame paveiksle matome pavaizduotas paprasto tiesioginio bei *radiosity* apšvietimo algoritmų scenas. Kaip matome, kaip vien dėl tinkamo apšvietimo/šešėlių gauname žymiai tikroviškesnę sceną. Paveikslas paimtas iš: http://en.wikipedia.org/wiki/Radiosity_(3D_computer_graphics)

Šešėlių planai (angl. *shadow maps*) ir šešėlių tūriai (angl. *shadow volumes*) yra dvi iš populiariausių technikų naudojamų realaus laiko šešėlių generavimui. Kuri geresnė, vienareikšmiškai negalima pasakyti – abi turi savo privalumų bei trūkumų. Aišku šios technikos nuolat tobulinamos, remiantis jomis išrandami nauji, kartais hibridiniai algoritmai, norint išgauti tikroviškesnius, greitesnius šešėlių generavimo būdus.

Šiame darbe būtent ir pristatysime šešėlių planų (angl. *shadow maps*) ir šešėlių tūrių (angl. *shadow volumes*) algoritmus, bei jų praktiškas realizacijas.

Teorinėje darbo dalyje pateikiama trumpa vaizdo generavimo kompiuterinėje grafikoje apžvalga, toliau trumpai pristatomi planiniai bei tūriniai šešėliai ir jų pagrindai. Čia žinoma apžvelgsime ir du hibridinius algoritmus. Tai silueto plano (angl. *shadow silhouette maps*) ir efektyvus hibridinis šešėlių vaizdavimo (angl. *an efficient hybrid shadow rendering*) algoritmas.

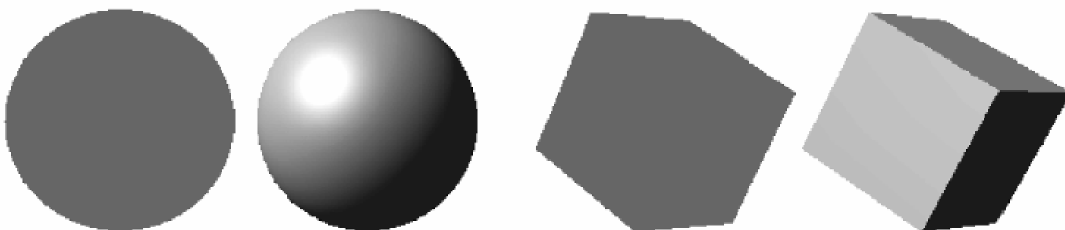
Tiriamojoje darbo dalyje pateiksime praktiškas šešėlių plano bei šešėlių tūrio algoritmų realizacijas bei jos detales, pateiksime rezultatus bei šių metodų tarpusavio palyginimą. Pilnai veikiančių programų kodai bus pateikti priede.

1. TEORINĖ DALIS

Šioje dalyje apžvelgsime pagrindinius kompiuterinė grafikos aspektus susijusius su apšvietimu: kas šviesos šaltiniai; kas yra šešėliavimas ir šešėliai, ir kuo jie skiriasi. Toliau apžvelgsime du pagrindinius realaus laiko šešėlių generavimo metodus – tai Šešėlių planai ir šešėlių tūriai, bei pora jų hibridinių algoritmų.

1.1. Šešėliai kompiuterinėje grafikoje apžvalga

Šviesos atspindžiai, šešėlių kritimas reikalingas vaizduoti trijų dimensijų scenai dvimatyje vaizduoklyje [2, 4, 10]. Šviesos bei šešėlių metimas (angl. *projection*) yra tam tikras tekstūrų dengimas (angl. *mapping*) iš tridimensinės erdvės į plokštumą. T. y., šviesos bei šešėlių metimas tiesiog apibūdina kur taškas ar objektas turi būti nupieštas ant metimo plokštumos. Visų pirma reiktų nustatyti matomus objektus, kurie turi būti nupiešti ir ant kurių metama šviesa ir šešėliai, o kurie slepiasi už kitų objektai. Informacijos kur objektas turėtų būti nupieštas ant metimo plokštumos, t. y. kuriuos pikselius dengia objektas, nėra pakankamas tikroviškos 3D scenos atvaizdavimui. Pav. 1.1. vaizduoja šviesos bei šešėlių metimą ant pilkos sferos bei kubo. Pirmame variante sferos bei kubo pikseliams tiesiog nustatoma pilka spalva. Taip nuspalvinus vienoda spalva geometriniai objektai praranda beveik visą informaciją apie jų tridimensinę struktūrą. Sfera tampa pilku apskritimu, o kubas – šešiakampiu.



1.1. pav. Apšviestos ir neapšviestos trimatės figūros [2]

Pridėjus apšvietimą bei atspindžius suteikiami skirtingi šviesos efektai trimačio objekto paviršiams bei nevienodai metami šešėliai. Šitaip, net ir plokšti paveikslai suteikia tridimensinį įspūdį kaip matoma 1.1 paveiksle. Šešėliavimas (angl. *shading*) nurodo kaip perteikti objekto paviršių su apšvietimu ir atspindžiais.

Pagal teorinį požiūrį, apskaičiavimai šešėliavimui turėtų būti padaryti kiekvienam šviesos bangos ilgiui individualiai. Kadangi tai neįmanoma, skaičiavimai visad bus apriboti trijų pirminių spalvų: raudonos, žalios ir mėlynos, kad būtų galima nustatyt RGB reikšmes atvaizdavimui.

1.1.1. Šviesos šaltiniai

Prie informacijos apie objektus ir žiūrovą, apibūdinant 3D sceną ,turi būti nurodoma informacija ir apie scenos apšvietimą [2, 4, 10]. Vienas šviesos šaltinis ar keletas šviesos šaltinių gali prisidėti prie scenos apšvietimo. Daugumoje atvejų , šviesos šaltiniai teikia baltą ar „pilką" šviesą, t. y., balta šviesa, kuri nuturi pilno intensyvumo. Bet šviesos taip pat būna ir spalvotos, pvz. scenoje gali būti raudona ar oranžinės spalvos šviesa einanti ryte nuo saulės. Šviesos šaltinio spalva bei intensyvumas yra apibrėžiami tinkamomis RGB reikšmėmis.

Pati paprasčiausia šviesos forma – tai aplinkos šviesa. Aplinkos šviesa neateina iš specifinio šviesos šaltinio ir neturi krypties. Ji atvaizduoja šviesą, kuri daugiau ar mažiau yra aplinkui scenoje, atsirandanti iš daugumos šviesos atspindžių ant įvairių paviršių. Kambary su lempa ant stalo, nebus visiškai tamsu po stalu, nors lempa ir negali peršviesti kiaurai stalo. Šviesa atspindėta nuo stalo paviršiaus, po to nuo sienų, patenka po stalu. Žinoma šviesos patekusios po stalu intensyvumas bus mažesnis, bet jos intensyvumas apytikriai vis dar bus toks pat kaip ir visur, neateinantis iš specifinės krypties. Aplinkos šviesa yra supaprastinimas apšvietimo skaičiavimui. Iš teorinės pusės, nieko nėra panašaus kaip aplinkos šviesa. Teisingas būdas nustatyti aplinkos šviesą būtų, susekti visus šviesos atspindžius. Bet tai nepaprastai padidintų skaičiavimo kiekius, todėl šis variantas po kol kas nėra labai tinkamas realaus laiko kompiuteriniai grafikai.

Aplinkos šviesai užtenka nurodyti jos spalvą. Kryptinis šviesos šaltinis prie spalvos dar turi kryptį. Šviesos spinduliai nuo kryptinio šviesos šaltinio yra lygiagretūs. Kryptinės šviesos yra naudojamos modeliuose su begalo toli esančiu šviesos šaltiniu, pvz. saulės šviesa.

Lempa modeliuojama kaip taškinis šviesos šaltinis. Taškinis šviesos šaltinis turi poziciją ir šviesos spinduliai sklinda visomis kryptimis. Intensyvumas šviesos mažėja didinant atstumą. Šis efektas vadinamas nusilpimu. Sekantis argumentas rodo, kad šviesos intensyvumas mažėja kvadratiškai su atstumu iki šviesos šaltinio. Jei taškinis šviesos šaltinis yra sferos centre, kurios spindulys yra r, tada pilna šviesos energija bus išdalinta lygiai visam vidiniam sferos paviršiui. Jei sferą pakeistume didesne sfera su spinduliu R, tai pilna šviesos energija nesikeičia. Bet ji tiesiog paskirstyta po didesne sferą. Dviejų sferų paviršių santykis yra:

$$\frac{4\pi r^2}{4\pi R^2} = \left(\frac{r}{R}\right)^2 . \quad (1.1.)$$

Santykiui r/R = ½ kiekvienas didesnės sferos vidinio paviršiaus taškas gauna tik ketvirtadalį energijos, lyginant su mažesnės sferos taškais.

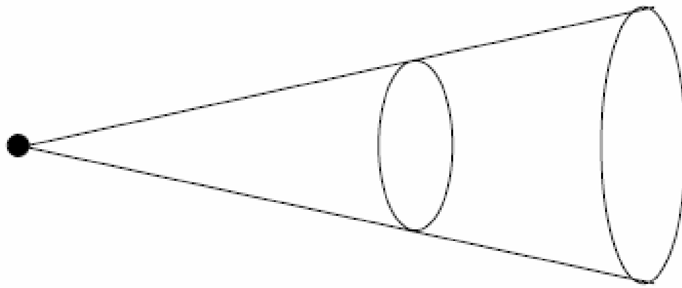
Teorinis nusilpimo modelis būtų: padauginti šviesos intensyvumą nuo šaltinio taško iš $1/d^2$, kai jis atsitrenks į objekto paviršių atstumu d. Intensyvumas nukris labai greitai su atstumu, tai intensyvumo skirtumai didesniems atstumams bus beveik nepastebimi. Kaip bebūtų, objektams esantiems labai arti šviesos šaltinio, atsiras radikalūs pokyčiai. Intensyvumas galėtų būti savavališkai didelis ir tęstis iki begalybės, kai paviršius yra tiesiai prieš šviesos šaltinį. Kad būtų išvengta šių efektų, intensyvumo sumažėjimas nuo nusilpnėjimo yra modeliuojamas pagrindiniu kvadratiniu polinomu daliklyje šioje formoje:

$$f_{\mathrm{att}} = \min\left\{\frac{1}{c_1 + c_2 d + c_3 d^2}, 1\right\} \quad (1.2.)$$

Kur konstantos c_1, c_2, c_3 gali būti pasirenkamos individualiai, kiekvienam taškiniui šviesos šaltiniui. d yra atstumas objekto iki šaltinio. Ši formulė garantuoja, kad intensyvumas niekad negali viršyti 1. Konstantos taip pat gali būti nustatytos, kad silpnėjimo efektas būtų pakenčiamesnis, nei paprasta $1/d^2$ forma.

Kitas paplitęs šviesos šaltinis būtų žibintai. Skirtumas nuo taškinio šviesos šaltinio yra tas, kad žibintai turi kryptį, kuria išskleidžia šviesą kūgio forma. Žibintai yra apibūdinami spalva, vieta, kryptim bei kampiniu apribojimu, kuris apibūdina šviesos kūgio išplėtimą.

Nusilpimas žibintui apskaičiuojamas pagrinde pagal 1.2. form., tuo pačiu būdu kaip ir taškiniams šviesos šaltiniams. Kvadratinis intensyvumo sumažėjimas su didėjančiu atstumu gali būti išvestas iš figūros 1.1.1. paveiksle, kur galima matyti, kad pilna energija nuo žibinto yra paskirstyta po apskritimą, kurio spindulio augimas yra tiesiškai priklausomas nuo atstumo. Be to, paviršius auga kvadratiškai nuo atstumo.

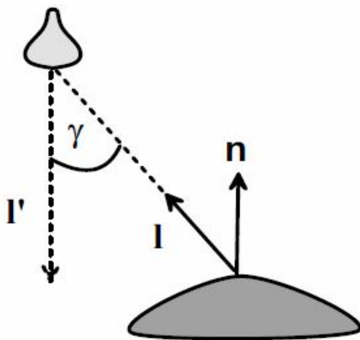


1.1.1. pav. Žibintinis šviesos šaltinis [2]

Labiau realistiniam žibinto modeliui, turėtų būti įvertinama, kad intensyvumas mažesnis šviesos kūgio kraštuose, nei centre. Warn‘o modelyje, parametras p naudojamas kontroliavimui, kaip greitai intensyvumas nuo centro mažėja. Tarkim paviršiaus taškas yra apšviestas žibinto. Tegul **l** būna vektorius, kuris veda nuo šaltinio taško iki apšviečiamo taško, ir tegul $\mathbf{l}_S$ yra kūgio ašis rodanti šviesos kryptimi. Tada šviesos intensyvumas paviršiaus taške yra apskaičiuojamas pagal formulę:

$$I = I_S \cdot f_{\text{att}} \cdot (\cos\gamma)^p = I_S \cdot f_{\text{att}} \cdot \left(-\mathbf{l}_S^\top \cdot \mathbf{l}\right)^p . \quad (1.3.)$$

I_S yra žibinto šviesos intensyvumas, f_{att} yra nuo atstumo priklausomas faktorius silpnėjimui iš 1.2. form. ir γ yra kampas tarp **l** ir $\mathbf{l}_S$. p reikšmė kontroliuoja kaip šviesa yra fokusuota. Kai p=0 žibintas elgiasi tokiu pačiu principu, kaip ir taškinis šviesos šaltinis. Kuo p didesnis, tuo daugiau šviesos intensyvumas sukoncentruotas į kūgio ašį. Kuo mažesnis p, tuo daugiau sukoncentruotas į kraštus. 1.1.2. pav. iliustruoja Warn‘o modelio situaciją:

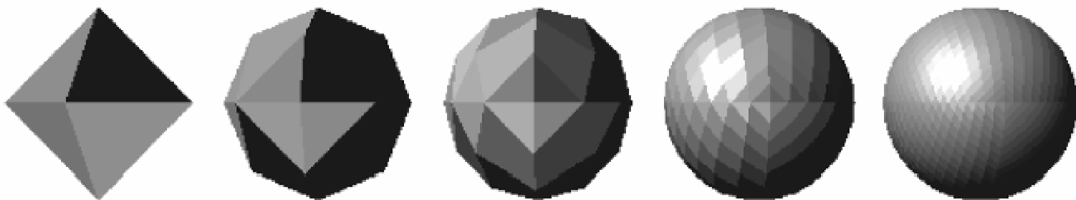


1.1.2. pav. Warn‘o modelio situacija [2]

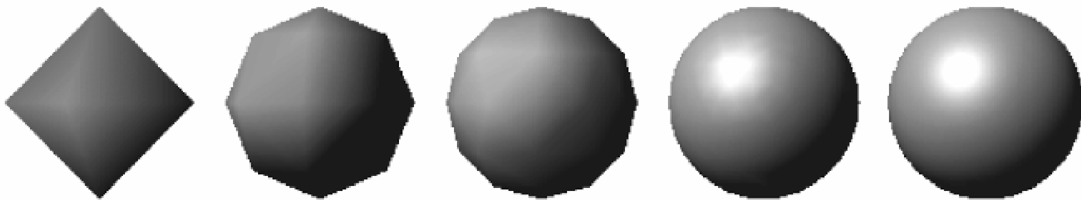
Reikėtų žinot, kad intensyvumas I_S 1.3. lygybėje gali turėti skirtingas reikšmes kiekvienai spalvai, ar bent jau kiekvienai pirminiai spalvai: raudonai, žaliai bei mėlynai.

1.1.2. Šešėliavimas

Kad būtų galima apskaičiuoti pikseliui teisingą spalvą ant metimo (šviesos, šešėlio) plokštumos, nebūtinai užtenka nustatyti vien, kuris objekto paviršius yra matomas piksely, bet ir kuriame taške metimas pro pikselį susiduria su plokštuma [2, 4, 10]. Kubinėm laisvos formos paviršiams tai reikštų lygybės sistemą. Kaip bebūtų, šviesos atspindžiai nėra apskaičiuojami tiesiogiai laisvos formos paviršiam, bet apytiksliai su poligonais. Ką tai reiškia normaliniams vektoriams nukreiptiems į paviršių? Labai paprasta išeitis būtų ignoruoti originalius normalinius vektorius į laisvos formos paviršius ir naudoti normalinius vektorius į plokščius poligonus.



1.1.3. pav. Sfera skirtingu padengimu atvaizduota naudojant plokščiąjį šešėliavimą (angl. *flat shading*)



1.1.4. pav. Sfera skirtingu padengimu atvaizduota naudojant interpoliuotą šešėliavimą [2]

Konstanta ar plokščias šešėliavimas supaprastina šią idėją dar labiau. Poligonui, spalva nustatoma tik pagal vieną pikselį, į kurį nukreiptas vienas normalinis vektorius. Visi kiti pikseliai esantys tame pačiame poligone nudažoma ta pačia spalva, kuri nustatoma pagal krentančią šviesą. Šis būdas yra tinkamas nuo šių prielaidų:

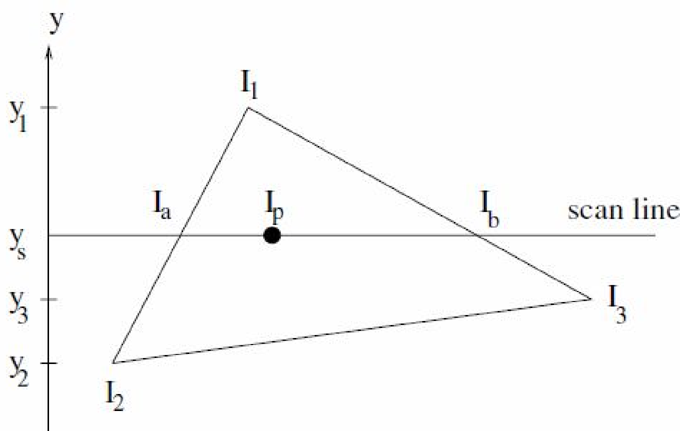
- Šviesos šaltinis yra begalo nutolęs, t.y. tinka tik kryptinėms šviesoms.
- Stebėtojas yra begalo nutolęs.
- Poligonas atstovauja tikrą objekto paviršių ir nėra tiesiog lankstyto paviršiaus aproksimacija.
- Nėra jokių veidrodinių atspindžių.

Su šiom prielaidom, šešėliavimas gali būti apskaičiuojamas greitu ir paprastu būdu, bet tai neduos realistinį paveikslą. Net ir smulkiausiai 1.1.3. pav. padengta sfera aiškiai rodo savo trikampius, tuo tarpu 1.1.4. pav. net ir stambesni trikampiai beveik nesimato.

Reikia begalo smulkių trikampių padengimo, kad būtų išvengtas trikampių matomumo efektas naudojant plokščiąjį šešėliavimą. Šios problemos priežastis yra žmogaus įgimta matymo sistema, kurioje automatiškai yra sustiprinamas kontrastas, t. y., kraštai, todėl ir labai mažos briaunos yra lengvai pastebimos.

Todėl vietoj plokščiojo šešėliavimo daugiau naudojamas yra interpoliuotas šešėliavimas (angl. *interpolated shading*). Interpoliuotam šešėliavimui reikalinga nustatyti normalinius vektorius poligono ar trikampio viršūnėse. Čia paaiškinimui naudosime trikampius. Normaliniai vektoriai trijose trikampio viršūnėse gali būti skirtingi interpoliaciniame šešėliavime, kai trikampis turėtų aproksimuoti dalį išlenkto paviršiaus.

Kai išlankstytas paviršius yra aproksimuotas pagal trikampius ir tinkami normaliniai vektoriai yra specifikuoti trikampių viršūnėse, Gouraud‘o šešėliavimas apskaičiuoja spalvą kiekvienoje trikampio viršūnėje pagal atitinkamus normalius vektorius. Šešėliavimas kitų taškų trikampyje yra paremtas spalvos interpoliavimui gautam iš trijų viršūnių.



1.1.5. pav. Skenavimo linijos technika [2]

Efektyvi schema intensyvumo skaičiavimui trikampyje naudoja skenavimo linijos techniką. Skenavimo linijai y_S intensyvumai I_a ir I_b trikampio kraštuose yra paskaičiuojami, kur skenavimo linija kerta trikampį. Reikšmės gaunamos pagal nusveriančia interpoliacija tarp atitinkamų trikampio kraštinių viršūnių. Intensyvumas keičiasi tiesiškai pagal skenavimo liniją su pradine I_a reikšme ir galutine I_b reikšme. 1.1.5. paveikslas iliustruoja šį principą. Intensyvumas yra paskaičiuojami pagal šias lygybes:

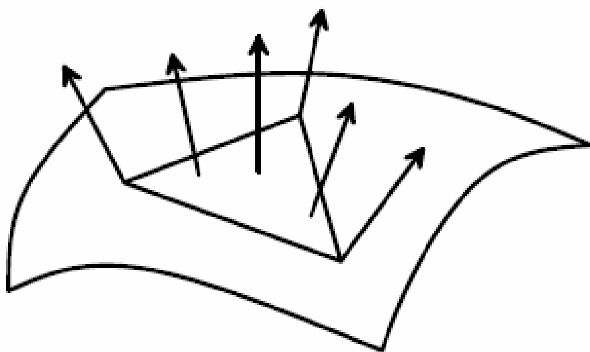
$$I_a = I_1 - (I_1 - I_2)\frac{y_1 - y_s}{y_1 - y_2}, \quad (1.4.)$$

$$I_b = I_1 - (I_1 - I_3)\frac{y_1 - y_s}{y_1 - y_3}, \quad (1.5.)$$

$$I_p = I_b - (I_b - I_a)\frac{x_b - x_p}{x_b - x_a}. \quad (1.6.)$$

Apskaičiuot spalvos intensyvumai yra sveiko skaičiaus reikšmės nuo 0 iki 255. Dažniausiai intensyvumai ant vieno trikampio labai nesiskiria, todėl absoliutus nuožulnumas, tiesinės intensyvumo kreivės palei skenavimo linija, bus mažas. Šiuo atveju, vidurio taško (angl. *midpoint*) algoritmas galėtų būti naudojamas, kad nustatyti atskiras intensyvumo reikšmes.

Nepageidaujamas matomų trikampių ir briaunų efektas pataisytas ir su Gouraud'o šešėliavimu. Nepaisant to, dėl tiesinės interpoliacinės schemos Gouraud'o šešėliavimui, maksimalus ir minimalus intensyvumas visada bus viršūnėse. Tai gali privesti prie matomų atsikišusių briaunų bei viršūnių. Phong'o šešėliavimas taip pat paremtas interpoliacija. Bet vietoj spalvos intensyvumo interpoliavimo trikampio viršūnėse, viršūnėse interpoliuojami normaliniai vektoriai, kad būtų galima apskaičiuot spalvų intensyvumus kituose taškuose. Šiuo būdu, minimalus ir maksimalus intensyvumas gali susidaryti ir trikampio viduj, priklausomai nuo normalinių vektorių viršūnėse ir nuo krypties iš kur šviesa ateina. 1.1.6. paveikslas vaizduoja išlenktą paviršių ir trikampį, kuris aproksimuoja dalį paviršiaus. Normaliniai vektoriai trikampio viršūnėse yra normaliniai vektoriai į paviršių šiušė taškuose. Trikampio viduj, normaliniai vektoriai yra išgaubtos kombinacijos šių vektorių.



1.1.6. pav. Vaizduojamas išlenktas paviršius ir trikampis, kuris aproksimuoja dalį šio paviršiaus [2]

Iš skaičiavimo pusės, Phong'o šešėliavimas reikalauja daugiau laiko nei Gouraud'o. Gouraud'o technikoje, sudėtingi apšvietimo skaičiavimai įtraukiant šviesos šaltinius bei atspindžius turi būti atlikti

tik trikampio viršūnėms. Visas likęs trikampis šešėliuojamas pagal paprastą skenavimo linijos techniką, atliekant paprastus skaičiavimus. Phong‘o šešėliavimui, po normalinių vektorių interpoliavimo, vis dar turi būti atliktas skaičiavimas kiekvienam pikseliui atvaizduojant apšvietimą.

Gouraud‘o ir Phong‘o šešėliavimai duoda gerą aproksimaciją lenktų paviršių šešėliavimui. Iš teorinės pusės, būtų geriau gauti normalinį vektorių taškui esančiam trikampy, tiesiai iš atitinkamo taško ant lenkto paviršiaus, kad nustatyti spalvą atitinkamo pikselio. Tai reikštų, kad neužtenka vien nustatytų normalinių vektorių pasirinktiems taškams – viršūnėms, bet reikalinga ir informacija apie originalų išlenktą paviršių, kuri būtina šešėliavimui. Bet tai būtų nepriimtina iš skaičiuojamosios požiūrio pusės.

1.1.3. Šešėliai

„Šešėlio metimas“ nėra veikli materija, bet tiesiog šviesos trūkumas nuo šviesos šaltinio, kai šviesa nepasiekia objekto paviršiaus be šešėlio ant jo [2, 4, 10]. Apšvietimo lygybė įtraukiant ir šešėlius tampa:

$$\begin{aligned} I &= I_{\text{self_emission}} + I_{\text{ambient_light}} \cdot k_a \\ &+ \sum_j S_j \cdot I_j \cdot f_{\text{att}} \cdot g_{\text{cone}} \cdot \left(k_d \cdot (\mathbf{n}^\top \cdot \mathbf{l}_j) + k_{\text{sr}} \cdot (\mathbf{r}_j^\top \cdot \mathbf{v})^n \right) \end{aligned}$$

(1.7.)

Intensyvumas I 1.7-oje formulėje neturi viršyti vieneto, jei ir viršija, tai jis tiesiog prilyginamas vienetui.

S_j = 1, kai šviesa nuo šaltinio j pasiekia paviršių;

S_j = 0, kai šviesa nuo šaltinio j nepasiekia paviršiaus (šešėlis);

$I_{self-emision}$ – paties kūno šviesos intensyvumas;

$I_{ambient_light}$ – aplinkos šviesos intensyvumas;

k_a – paviršiaus aplinkos šviesos atspindžio koeficientas;

I_j – j šaltinio intensyvumas;

k_d – atspindžio išsklaidymo koeficientas;

f_{att} – intensyvumo sumažėjimas priklausantis nuo atstumo iki šaltinio, f_{att} = 1, kai šviesos šaltinis yra kryptinis;

g_{cone} – intensyvumo sumažėjimas priklausantis nuo šviesos kūgio centro link kraštų, g_{cone}= 1, kai šviesos šaltinis yra kryptinis;

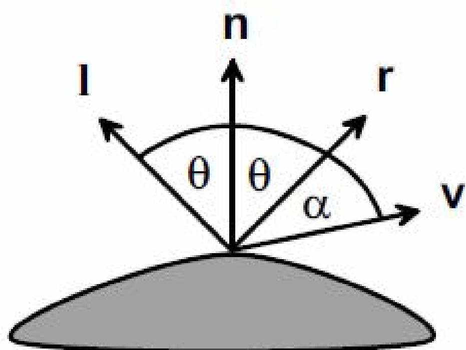
k_{sr} – veidrodinio atspindžio koeficientas;

l – žymi vektorių, kuris nurodo iš kur šviesa pasiekia tašką (1.1.7. pav.);

n – normalinis paviršiaus vektorius tame taške, kampai su vektoriais **l** ir **r** yra vienodi;

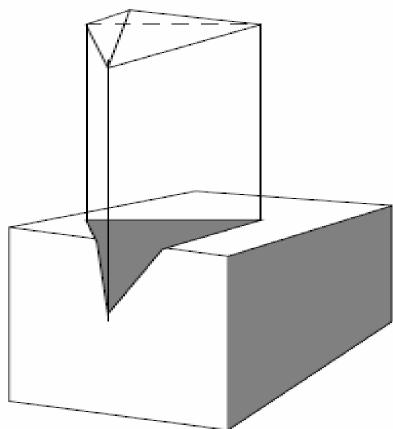
r – vektorius žymi idealų veidrodinį atspindį;

v – šis vektorius rodo į stebėtojo tašką.



1.1.7. pav. Idealaus atspindžio apskaičiavimas [2]

Kada šviesos šaltinio šviesa pasiekia paviršių, o kada ji blokuojama objekto, kada ir susidaro šešėlis? Tam naudojami matomų paviršių nustatymo metodai, t. y. metodai nustatantys, kurie objektai matomi scenoje iš stebėjimo taško, o kuriuos užstoja kiti objektai. Šešėlių nustatymo problema ta pati, tiesiog čia vietoj stebėtojo imamas šviesos šaltinis. Kai nustatoma, kad paviršius matomas iš šviesos šaltinio j taško, tada $S_j = 1$ ir šiam šaltiniui nėra šešėlio toje vietoje. Kai nustatoma, kad paviršius nėra matomas iš šviesos šaltinio j taško, tada $S_j = 0$ ir šešėlis yra metamas ant objekto. 1.1.8-ame paveiksle parodytas šešėlis krentantis nuo tetraedro ant kubo. Šešėlis nereiškia, kad paviršius bus juodas. Aplinkos šviesa vis dar bus atspindima. Ir jeigu yra daugiau nei vienas šviesos šaltinis scenoje, paviršius gali būti blokuojamas nuo vieno, bet ne nuo visų.



1.1.8. pav. Šešėlis ant objekto [2]

Ryšys tarp šešėlio ir matomumo nustatymo yra realizuojamas dviejų-praėjimų z-buferio (angl. *two-pass zbuffer)* arba dviejų-praėjimų gylio buferio (angl. *two-pass depth buffer*) algoritmu. Pirmajame variante, naudojamas standartinis z-buferio algoritmas su kelėta modifikacijų. Stebėtojas pakeičiamas šviesos šaltiniu. Kryptiniam šviesos šaltiniui naudojama lygiagreti projekcija priešinga šviesos sklidimo pusei. Taškiniams šviesos šaltiniams ir žibintams naudojamas perspektyvinė projekcija, kur perspektyvos centras yra šviesos šaltinio taškas. Visuose atvejuose projekcija pataisoma į lygiagrečią projekciją į x/y paviršių, pagal tinkamą transformaciją T_L. Šiame pirmame praėjime dviejų-praėjimų z-buferio algoritme tik z-buferio reikšmės Z_L yra įvedamos. Kadro (angl. *frame*) buferio ir jo paskaičiavimų nereikia. Antras algoritmo perėjimas yra identiškas z-buferio algoritmas stebėtojui su toliau išvardintom modifikacijom.

Transformacija T_L paverčianti perspektyvinę projekciją, kur stebėtojas yra perspektyvos centras, į lygiagrečią projekciją į x/y plokštumą kaip visada reikalinga. Stebėtojas z-bufery Z_V irgi naudojamas kaip įprasta antrame algoritmo perėjime. Bet prieš tai kol projekcija įvedama į kadro buferį stebėtojui, yra atliekamas apšvietimo testas, kad patikrintų ar paviršius yra apšviestas tikrinamos šviesos. Jei taško ant paviršiaus koordinatės (x_V, y_V, z_V) yra projektuojamos tai transformacija

$$\begin{pmatrix} x_L \\ y_L \\ z_L \end{pmatrix} = T_L \cdot T_V^{-1} \cdot \begin{pmatrix} x_V \\ y_V \\ z_V \end{pmatrix} \quad (1.8.)$$

duoda koordinates to pačio taško iš šviesos šaltinio stebėjimo taško. T_V^{-1} yra inversinė transformacija, t. y., inversinė matrica T_V. Reikšmė z_L palyginama su įrašais Z_L z-bufery šviesos šaltiniui pozicijoje (x_L, y_L). Jei reikšmė Z_L z-bufery įvedama mažesnė nei z_L toje pozicijoje, tada turėtų būti objektas tarp šviesos šaltinio ir tikrinamo paviršiaus, ir šis paviršius turėtų negauti šviesos iš tikrinamo šaltinio. Paviršius yra šešėly ir faktorius S_j 1.7-toje lygybėje turėtų būti nustatytas nuliui. Jei scenoje yra daugiau kaip vienas šaltinis, pirmasis algoritmo praėjimas turėtų būti atliktas kiekvienam šviesos šaltiniui. Antrajame praėjime nustatoma ar paviršius gauna šviesos iš atitinkamų šaltinių ir pagal tai parenkami faktoriaus S_j reikšmės.

1.2. Realaus laiko šešėlių technikos

1.2.1. Šešėlių tūrio algoritmas

Šešėlių tūrio *(angl. shadow volume)* algoritmas yra geometrija paremtas šešėlių algoritmas, kuriam reikia informacijos apie poligonų tinklų jungimą, kad būtų galima efektyviai apskaičiuoti kiekvieną šešėlį metančio objekto siluetą [1]. Tai yra ir per pikselinis algoritmas, t. y., patikrinantis kiekvieną paduodamą fragmentą (duomenys atskiro pikselio, reikalingi jo atvaizdavimui) ar šis randasi šešėly. Ši operacija gali būti pagreitinta naudojant grafinę techninę priemonę (šablono buferį). Pseudo kode algoritmas atrodo taip:

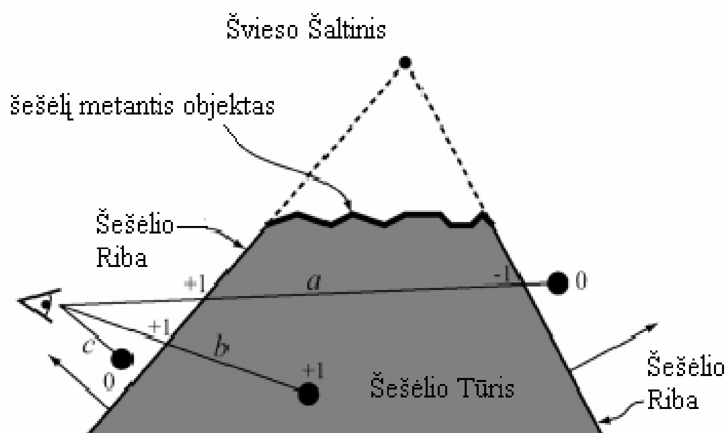
```
procedure SHADOWVOLUMERENDERING
for visiems taškais išreikštiems fragmentams do
    nupiešti fragmentą su aplinkos ir emisiniu apšvietimu;
    atnaujinti Z-buferį   ;
end for
COMPUTEFRAGMENTSINSHADOW
for visiems taškais išreikštiems fragmentams do
    if not INSHADOW(fragmentas) then
        nupiešti fragmentą su difuziniu ir atspindinčiu apšvietimu;
    end if
end for
```

1.2.1. pav. Kadras iš žaidimo DOOM 3. Geras šabloninių šešėlių tūrių pavyzdys. Pav. paimtas iš http://www.gamespot.com/pc/action/doom3/

Suprantama, pagrindinė problema yra nustatyti ar paduodamas fragmentas randasi šešėly ar ne. Šią procedūrą nesunkiai atlieka kiekviena grafikos techninė priemonė su šablono buferiu. Po kiekvieno fragmento su išskiriančiu ir aplinkos apšvietimu pateikimo, turėtų būti įterpta:

```
procedure COMPUTEFRAGMENTSINSHADOW (kai Z-tinkamas)
for visi objektai metantys šešėlį do
    suskaičiuoti potencialius silueto kraštus(PSE) poligoninio modelio;
    suskaičiuoti šešėlio tūrio poligonus nuo šviesos šaltinio ir PSE;
end for

for visiems priekiniams šešėlio ribos poligonam iš žiūrėjimo taško do
    if Z-buferio testas praeina then
       padidinti šablono buferio reikšmę;
    end if
end for

for visiems esantiems už šešėlio ribos poligonam iš žiūrėjimo taško do
    if Z-buferio testas praeina then
        pamažinti šablono buferio reikšmę;
    end if
end for
```

Dabar įvertinti dviejų reikšmių (taip arba ne) funkciją INSHADOW (panaudota procedūroje SHADOWVOLUMERENDERING) kiekvienam fragmentui, tiesiog užklausiama šablono buferio reikšmė. Jei reikšmė šablono bufery po COMPUTEFRAGMENTSINSHADOW procedūros yra didesnė nei nulis, fragmentas yra šešėly ir neturi būti nupieštas antroje generavimo dalyje. Kad būtų lengviau suprast geometriškai, pateiktas 1.2.2. pav.

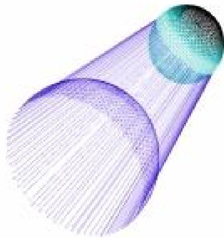


1.2.2. pav. Šešėlio tūrio perteikimas: skaičiuojant įeinančius ir išeinančius spindulius iš žiūrėjimo taško, kad nustatyt ar matomas fragmentas šešėly [1]

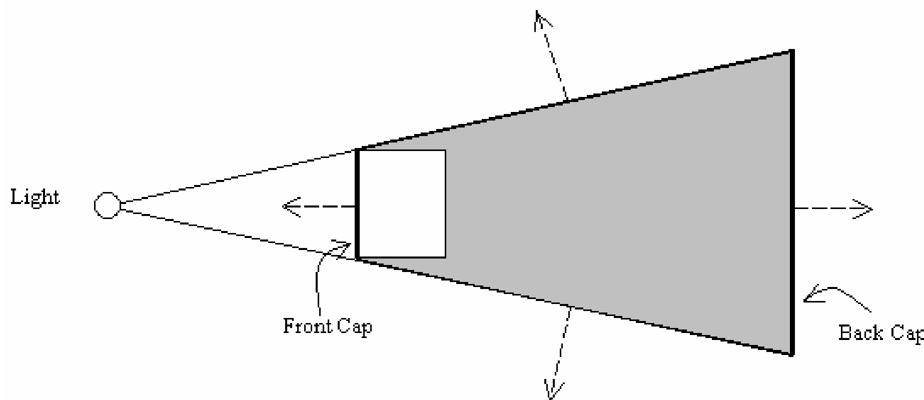
Algoritmas, kaip sakoma, kenčia nuo trūkumų, kurie jį daro nepraktišką tam tikrom situacijom, ypač jei specialiose situacijose nėra elgiamasi su juo tinkamai. Visų pirma, algoritmas veikia tik tada, kai žiūrėjimo taškas yra už šešėlio ribų, kitaip šablono skaičiavimas yra atvirkščias. Tai gali būti ištaisyta tikrinant šią situaciją, invertuojant šešėlių tikrinimą ir sužymint šablono buferį į 2N-1 (kur N yra šablono buferio tikslumas), bet kai šablono buferis laiko tik nepažymėtas reikšmes ir sumažėjimas nuo nulinės reikšmės, gali vėl būti priežastis neteisingo šešėlio. Labiau tinkamas sprendimas būtų pasiūlytas Everitt'o ir Kilgard'o 2002 m. Visų pirma, vietoj to, kad skaičiuojant šablono reikšmes sustiprinant priešais stovinčias šešėlio ribas ir sumažinant užnugary stovinčias šešėlio ribas Z-tinkamas versijoje, visas procesas modifikuojamas skaičiuojant nuo begalybės, vietoj to, kad nuo žiūrėjimo taško, taip vadinamame Z-netinkamas versijoje:

```
procedure COMPUTEFRAGMENTSINSHADOW (kai Z-tinkamas)
for visi objektai metantys šešėlį do
    suskaičiuoti potencialius silueto kraštus(PSE) poligoninio modelio;
    suskaičiuoti šešėlio tūrio poligonus nuo šviesos šaltinio ir PSE;
end for

for visiems priekiniams šešėlio ribos poligonam iš žiūrėjimo taško do
    if Z-buferio testas nepraeina then
       pamažinti šablono buferio reikšmę;
    end if
end for

for visiems esantiems už šešėlio ribos poligonam iš žiūrėjimo taško do
    if Z-buferio testas nepraeina then
        padidinti šablono buferio reikšmę;
    end if
end for
```

Dvi reprezentacijos (Z-tinkamas ir Z-netinkamas) yra visiškai ekvivalentiški ir skaičiuoja tą pačią reikšmę, ir nekenčia nuo problemos paminėtos anksčiau: žiūrėjimas šešėly nebėra ypatingas atvejis. Kita problema susijusi su šabloninių šešėlių tūriais, kada tūrinį šešėlį kertančioji plokštuma tęsiasi iki begalybės (1.2.3. pav.). Bet tūrinis šešėlis negali tęstis iki begalybės, nes taip gali būti perpildytas buferis. Išeitį pasiūlė Everitt'as ir Kilgard'as. Naudojant Z-netinkamas metodą ir nustatant kirpimo plokštumą į begalybę, bereikia nustatyti tūrinio šešėlio dangčius (1.2.4. pav.). Tam nustatomi objekto į kurį šviečia šaltinis kontūrai, t.y. viršutinis dangtis, kuris užstoja šviesą, toliau galima nustatyti apatinį dangtį, nes jo kontūro forma identiška viršutiniajam. Ši visa technika pavadinta tvirtų šabloninių šešėlių tūriais. Šabloninio šešėlio tūrio algoritmas duoda tik sunkius šešėlius ir, taip pat, tinkamas tik idealiems taškiniams šviesos šaltiniams, be pusšešėlių tipiškų ploto šviesos šaltiniams.



1.2.3. pav. Šviesos spinduliai išvesti pagal šviesą dengiančio kūno kontūrus [11]



1.2.4. pav. Šešėlio tūrio dangčiai. Angl. *light* - šviesa; angl. *front cap* – viršutinis dangtis; back cap – apatinis dangtis [11]

1.2.2. Šešėlių plano algoritmas

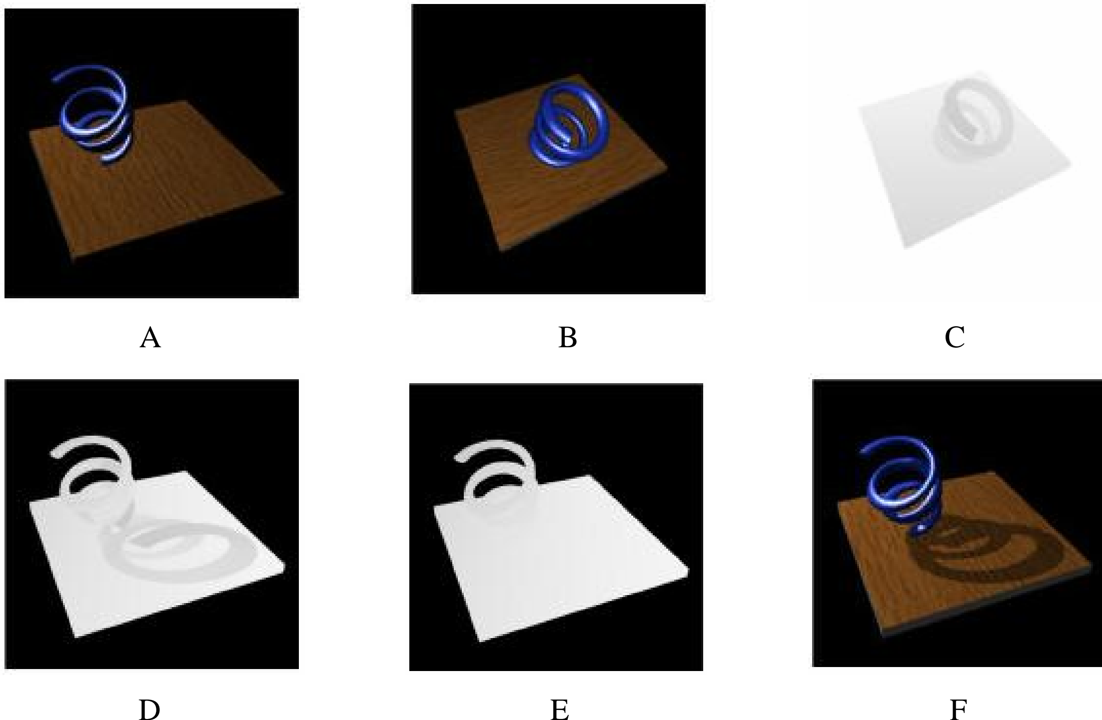
Šešėlių plano (angl. *shadow mapping*) yra visiškai paveikslo erdvės algoritmas, tai reiškia, žinojimas apie scenos geometriją nėra reikalingas, kad būtų atlikti būtini skaičiavimai. Kadangi šis algoritmas naudoja diskrečius bandinius, jis turi susidoroti įvairiais *aliasing*'o artefaktais, ir tai pagrindinis šios technikos trūkumas [1]. Bet šie trūkumai dalinai apeinami.

Šis algoritmas kaip ir šešėlio tūrio, atlieka šešėlio skaičiavimus kiekvienam pikseliui, kad nustatyt ar pikselis turi išsklaidantį ir/ar atspindintį komponentą ar visai neturi. Dvejų perėjimų algoritmas pseudo kode:

```
procedure SHADOWMAPPING

Render depth buffer (Z-buffer) from lights point of view,
resulting in a shadow map or depth map;
Now, render scene from the eye's point of view;

for all rasterized fragments do
```

```
    Determine fragment's xyz position relative to the light;
    That is transform each fragment's xyz into the light's coordinate system;
    A = depth map(x,y);
    B = z-value of fragment's xyz light position;
    if A < B then
        fragment is shadowed;
    else
        fragment is lit;
    end if
end for
```

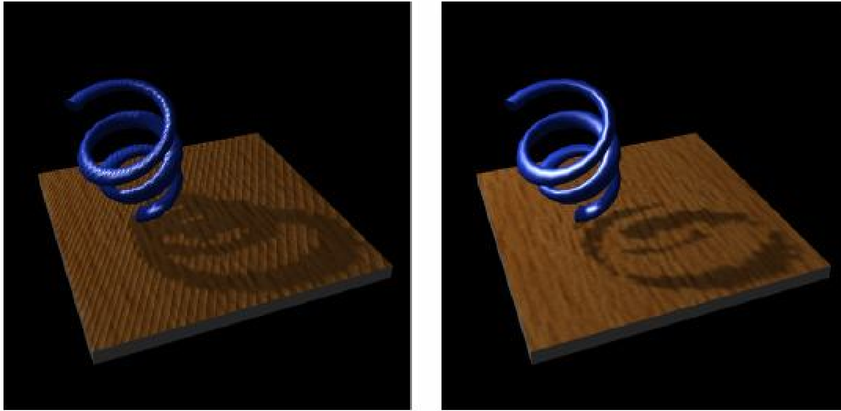
Išsami vizualizacija algoritmo pateikta demonstracinėje scenoje su paaiškinimais 1.2.5. paveikslėly.



1.2.5. pav. Šešėlių plano algoritmo vizualizacija. A: bešešėlinė scena iš akies regėjimo lauko; B: Scena matoma iš taškinio šviesos šaltinio; C: šešėlio planas (gylio planas) sukonstruotas iš šviesos šaltinio regos lauko; D: šešėlio planas iš akies regos lauko; E: šviesos plokštuminis nuotolis sudarytas iš akies regos lauko; F: šešėliuota scena atlikus gylio patikrinimą tarp D ir E [1]

Pirmoji problema būtų klaidingas savasis šešėliavimas: transformuojant tašką iš paviršiaus iš akies žiūrėjimo taško į šviesos koordinačių sistemą, A ir B turėtų būti idealiai lygūs viršui nurodytame algoritme. Tačiau dėl Z-buferio nustatymo panašu, kad bus A ≠B ir transformuotas taškas bus žemiau arba aukščiau paviršiaus. Pataisyti šį trūkumą, paklaidos reikšmė yra atimama, užtikrinti, kad klaidingas savasis šešėliavimas būtų panaikintas. Tam paprasčiausiai atimama konstantinė paklaida iš taškų z-reikšmių, po to kai jie būna transformuoti į šviesos erdvę, tai gali šiek tiek pastumti šešėlio liniją.

Paklaidos būtinybė mažėja su šešėlio planavimo tikslumu ir tendencija turėtų būti link aukštesnio paklaidos nustatymo (1.2.6. pav.).



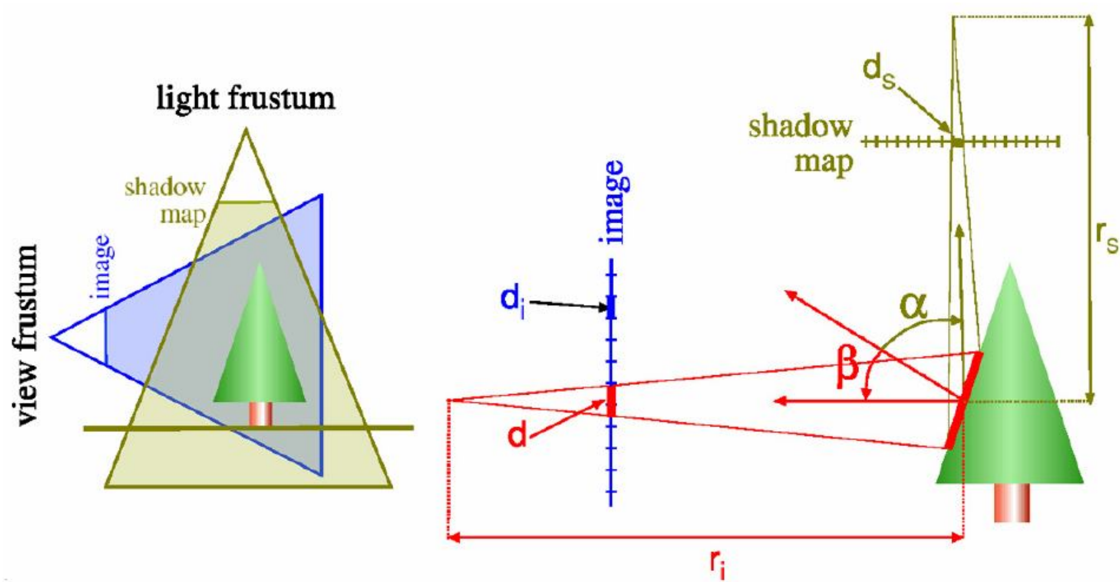
1.2.6. pav. Poveikis dėl paklaidos kontrasto. Kairysis pav. rodo kada per mažai paklaidos; viskas pradeda šešėliuoti patys save. Dešinysis naudoja per daug paklaidos, šešėlis pasislinkęs per toli [1]

Kad atlikti gylio patikrinimą, gylio buferis turi būti skaitomas atbulai ir prieinamas su transformuotom taškinio fragmento koordinatėm. Fragmento šviesos pozicija gali būti sugeneruota naudojant akies-tiesinį tekstūrų koordinačių generavimą, kuris paremtas projektuojamu tekstūravimu (angl. *projective texturing*). Visa procedūra kopijavimo z-buferio į tekstūrą, fragmento transformavimas į šviesos xuz koordinates, priėjimas prie tekstūrų gylio ir šešėlio patikrinimo atlikimas gali būti suplanuotas į techninį prietaisą naudojant egzistuojančius OpenGL plėtiniais (prietaisas taip pat turi palaikyti šiuos plėtinius).

Kaip paminėta anksčiau, *aliasing'o* artefaktai gali pastebimai pakeisti šešėlių kokybę. Šie artefaktai (1.2.9. pav. dešinėje ir 1.2.10. pav. antras iš kairės) priklauso po-atrankiniam šešėlio planavimui (angl. *shadow map undersampling*), tai reiškia kad šešėlio planavimo tekselis (angl. *texel*) planuojamas daugiau nei vienam kadro buferio pikseliui. Aiški formalizacija skaitoma taip (1.2.7. pav.): kiekvienas pikselis šešėlio plane, kurio dydis $d_S \times d_S$ planuojamas į pikselio vietą, kurios dydis (apytiksliai) d, galutiniame vaizde.

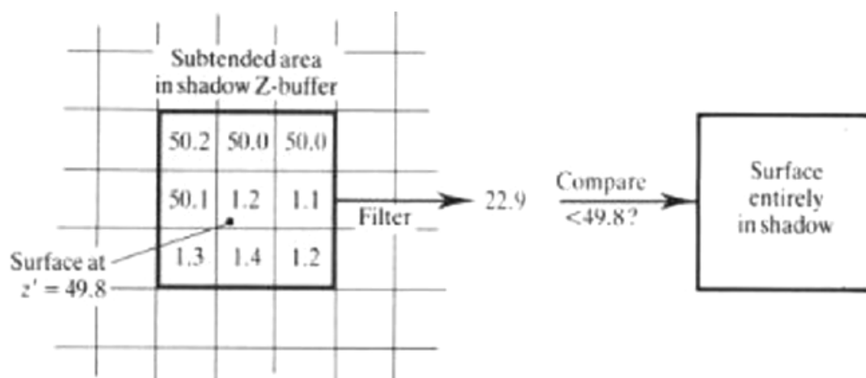
$$d = d_s \frac{r_s}{r_i} \frac{\cos \beta}{\cos \alpha} \quad (2.1.)$$

Po-atrankinis būna tada, kai d yra platesnis nei paveikslo pikselis d_i. Šešėlio plano kreipimasis *aliasing'as* gali būti padalintas į dvi nepriklausomas dalis, perspektyvos kreipimasis, priklausantis nuo sąlygos $d_S r_S / r_i$ ir projekcijos kreipimasis, priklausantis nuo cos β/cos α.

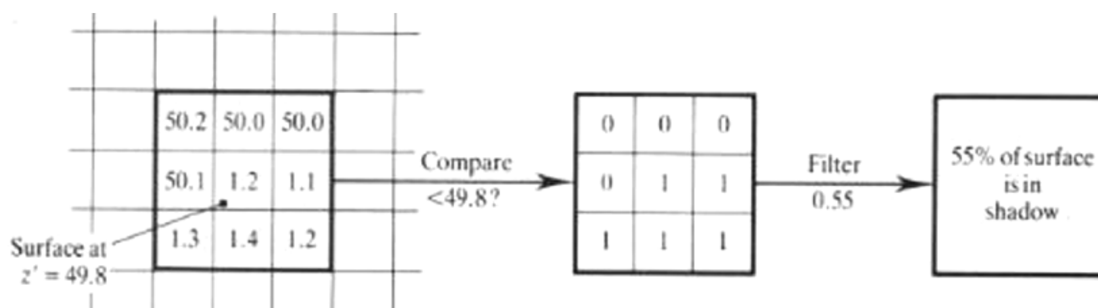


1.2.7. pav. Šešėlio plano *aliasing'o* formalizacija. angl. *view frustrum* – vaizdo kūgis; angl. *light frustum* – šviesos kūgis; angl. *image* – paveikslas; angl. *shadow ma*p – šešėlio planas [1]

Vienas sprendimas kreipimosi problemai, procentiškai artimas filtravimas (angl. *percentage closer filtering*), buvo pasiūlytas Reeves et. al. Pagrinde, gylio plano (angl. *depth map*) reikšmės negali būti sumaišytos, nes tai gali privesti prie netinkamų pikselių. Procentiškai artimas filtravimas suvidurkina lyginamus rezultatus filtro branduolio ribose, tai jei pavyzdžiui, mes dirbame su 3x3 branduoliu aplink apskaičiuotą fragmentą, kaip parodyta 1.2.8. paveiksle, su parodytais rezultatais, tada pikselis turi būti 55% šešėly (rezultatas parodytas 1.2.9. pav. kairėje).

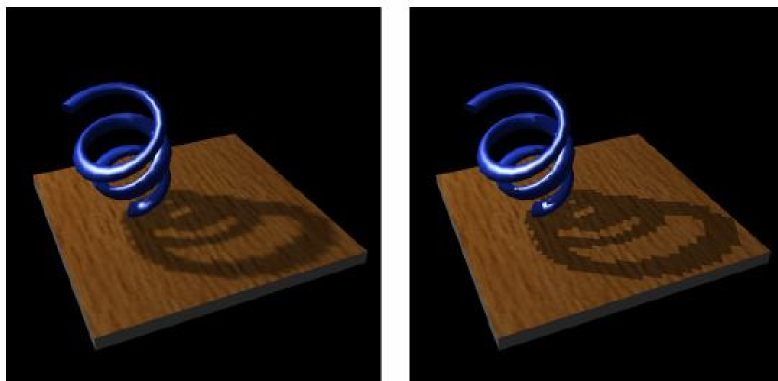


(a)



(b)

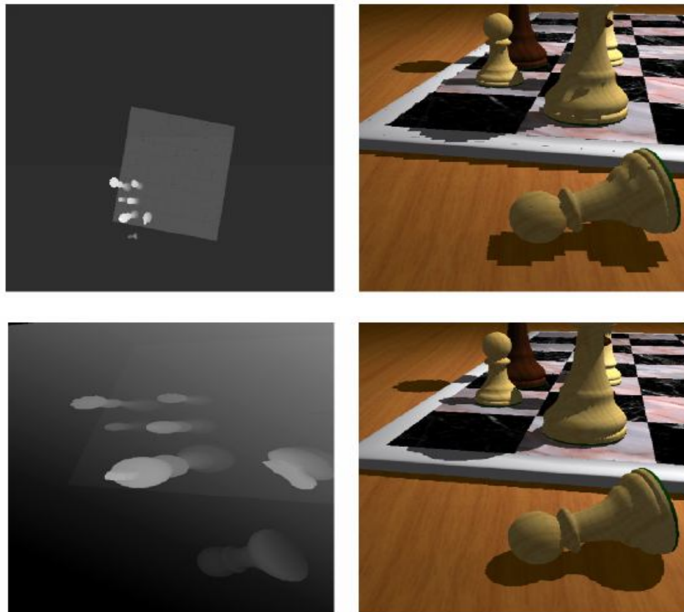
1.2.8. pav. Procentiškai priartėjančio filtravimo principas [1]



1.2.9. pav. Procentiškai priartėjančio filtravimo poveikis (šešėlio plano rezoliucija 128x128). Dešinysis pav. su filtravimu, kairysis pav. be filtravimo [1]

Stamminger'as ir Drettakis'as pristatė kitą sprendimą, labai pagerinantį perspektyvinį kreipimasį: perspektyviniai šešėlių planai (angl. *perspective shadow maps*). Pagrindinė idėja yra atlikti šešėlio plano skaičiavimus ir šešėlio patikrinimą normalizuoto įtaiso koordinačių erdvėje po perspektyvinės transformacijos. Tikslas čia yra išlaikyti trupmeną r_S / r_i artimą konstantai. Po perspektyvinėje erdvėje galutinis paveikslas yra ortogonalus vaizdas į kubo visumą. Pagrindiniai principai ir rezultatai dviejų

metodų parodyti 1.2.10. paveiksle. Šiame paveiksle pavaizduotas atvejis, kada perspektyvinio šešėlių planavimo technika duoda gerus rezultatus. Atvejai, kuriuose perspektyvinis šešėlių planas konverguoja į standartinę šešėlių plano uniforma, gali būti lengvai sukonstruotas.



1.2.10. pav. Perspektyvinių šešėlių planai. Viršutinė eilė: naudojant standartinę šešėlių plano uniforma šviesos erdvėje. Apatinė eilė: naudojant perspektyvinį šešėlių planą po-perspektyvinėje transformacijos erdvėje (iškarpos erdvėje (angl. *clip space*)) [1]

1.2.3. Silueto plano algoritmas

Šešėlių planai (angl. *shadow maps*) ir šešėlių tūriai (angl. *shadow volumes*) yra dvi populiarios technikos naudojamos realaus laiko šešėlių generavimui. Šešėlių planai yra lankstūs ir efektyvūs, bet jie labai linkę į aliasing'ą (dėl to grublėti šešėlių kraštai). Šešėlių tūriai yra tikslūs, bet reikalauja didelio pikselių generavimo dažnio (angl. *fillrate*), dėl to lėtai pateikia sudėtingas scenas. Pasiekt abiejų: tikslumo ir greičio yra gana sunki užduotis realaus laiko šešėliavimo algoritmams. Pagrinde, hibridiniai algoritmai iš šešėlių planų ir šešėlių tūrių, bando pagerinti šešėlių tūrių kokybės bei šešėlių planų našumo santykį, t. y., pasiekti šešėlių generavimo našumą kaip šešėlių planų algoritmo, neprarandant tūrių algoritmo šešėlių kokybės.

Silueto plano algoritmas (angl. *shadow silhouette maps*) algoritmas paremtas sekimu, kurį šešėlių planas atlieka gerai daugumoje paveikslo vietų, bet kenčia nuo nepageidaujamo aliasing'o šešėlių

kraštuose [8]. Šioje technikoje standartinių šešėlių planai papildomi informacija apie šešėlio krašto poziciją siluetų plane, kuri panaudojama šešėlių kraštų kokybei pagerinti.

Šis algoritmas turi du lygius: pirmame lygyje scena perteikta iš šviesos šaltinio regos taško, kad sugeneruoti gylio planą ir siluetų planą. Antrame lygyje scena pateikiama iš žiūrimo taško perspektyvos ir yra nustatomas šešėlis. Kadangi gylio planas ir tradicinis šešėlio planas realizuojami taip pat, čia bus aptarta tik siluetų plano generavimas ir jo naudojimas šešėlių nustatyme.

Kadangi siluetų plano tikslas teikti informaciją apie šešėlio kraštus, todėl, visų pirma, šioje informacijoje šešėlio kraštas turi būti uždaras, visų antra, ją turi būti lengva patalpinti į tekstūrą.

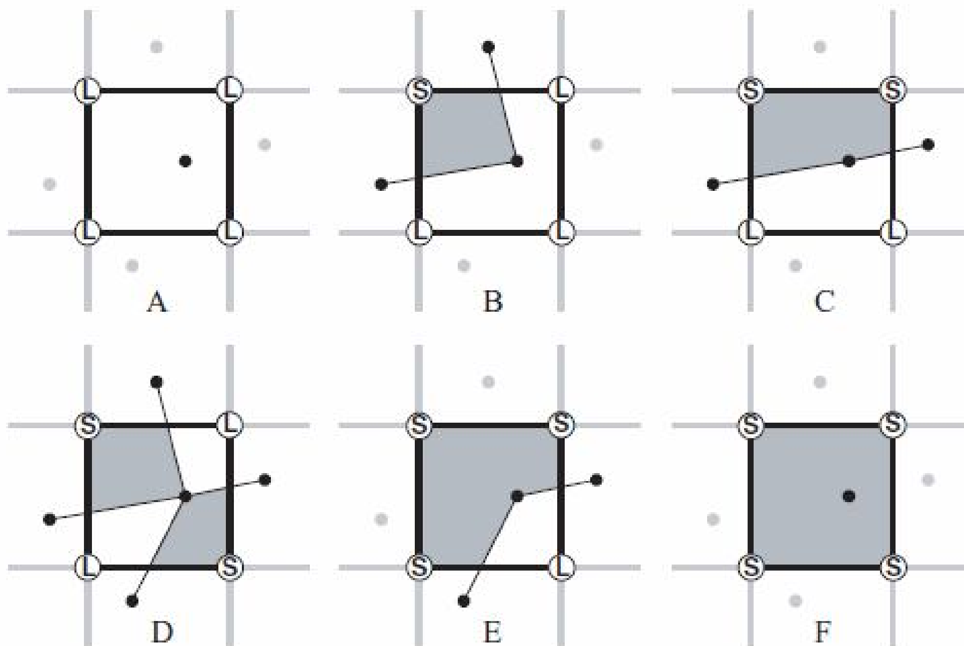
Šešėlio kraštas gali būti apytiksliai nustatytas pagal linijos segmentų seriją. Tam naudojama dvigubo kontūravimo algoritmas ir duomenų apie siluetą laikymas tekstūroje. Tada rekonstruojamas linijinis kontūras antrame algoritmo lygyje jungiant gretimus taškus. Be to, silueto planas yra tekstūra, kurios tekselis reprezentuoja taško (x, y) koordinatę, kuris guli ant objekto silueto. Tekseliai, per kuriuos neina jokie siluetai, yra žymimi kaip tušti. Kiekviename tekselyje tegali būti vienas silueto taškas; jei per tekselį eina daugiau kaip vienas siluetas, tai tik paskutinis įrašytas taškas bus laikomas.

Siluetų plano generavimui, geometrijos siluetų kraštai visų pirma turi būti identifikuojami naudojant, bet kurią šešėlių tūrių techniką. Šie kraštai sutaškuojami kaip linijos segmentai; taškavimo algoritmas turi sugeneruoti fragmentą kiekvienam tekseliui susikertančiam su linijos segmentu. Išėjęs fragmentų rinkinys yra 4-ryšis, sąlyga reikalinga jungti gretimiems taškams siluetų plane. Fragmento programa turi išrinkti tašką laikymui siluetų plane, kuris yra ant linijos segmento ir einamo tekselio viduj. Kadangi svarbu, kad tik matomi siluetai yra laikomi plane, yra palyginami silueto gylis su gylio planu, kad išmest siluetus, kurie yra pasislėpę nuo šviesos regos lauko. Šiame algoritme, gylio planas pakeičiamas nuo siluetų plano per puse pikselio kiekviena kryptimi, kad gylio planas reprezentuotų gylį kiekvieno tekselio kampuose siluetų plane. Kai pirma lygis pabaigtas, gylio ir silueto planai gali būti panaudoti šešėlių rekonstrukcijai iš žiūrovo taško.

Kad nustatyti ar taškas scenoje yra šešėlyje, pradžioj jis projektuojamas šviesos erdvėje. Einamojo fragmento gylis palyginamas su keturiais arčiausiai esančiais šešėlių gylio bandiniais. Ir jei jie visi sutinka, kad objektas yra apšviestas ar šešėly, ši sritis neturi silueto ribos einančios per jį ir fragmentas šešėliuojamas atitinkamai. Tai panašu į standartinį šešėlio gylio patikrinimą. Jei gylio patikrinimas nesutinka, kaip bebūtų, šešėlio riba turi praeiti per šį tekselį. Šiuo atveju, yra naudojamas siluetų planas, kad aproksimuoti teisingą šešėlio kraštą.

Šešėlio kraštai, pagal apibrėžimą, skiria šviesos ir šešėlio sritis. Be to, jie taip pat skiria sritis išlaikyto ir neišlaikyto gylio testo (patikrinimo). Duotas šios technikos kvadrato tekselis su gylio patikrinimu kiekviename kampe, stebima, kad šešėlis krašte turi kirsti kvadrato šonus su skirtingais

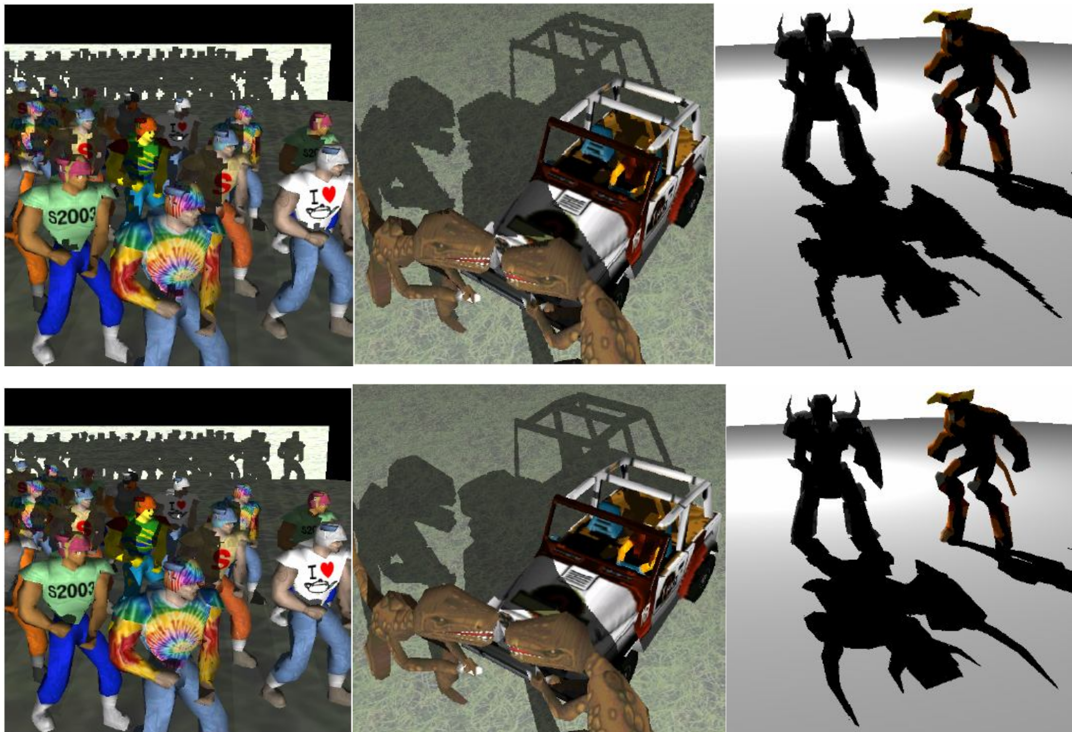
patikrinimo rezultatais. Jungiant einamąjį silueto tašką su atitinkamais kaimynais, galima sugeneruoti linijų segmentus, kurie aproksimuos šešėlio kraštą tikrinamajame tekselyje. Šis aproksimuotas silueto kontūras skiria tekselį į šešėliuotą ir apšviestą sritį. Taigi rezultatas atitinkamo kampo gylio patikrinimo pasakys kaip šešėliuoti fragmentus siluetų ribų pusėj. 1.2.11. paveikslas parodo visas skirtingas kombinacijas gylio patikrinimo rezultatų ir šešėliavimo konfigūracijų.



1.2.11. pav. Visos skirtingos kombinacijos gylio patikrinimo rezultatų ir šešėliavimo konfigūracijų. Gylio patikrinimo rezultatas kiekviename kampe yra išreikštas per L ar S, žymintčius apšviestą ir šešėliuotą atitinkamai. (A) visi kampai apšviesti, (B) vienas kampas šešėliuotas, (C, D) du kampai šešėliuoti, (F) visi kampai šešėliuoti [8]

Kadangi gylis tikrinamas diskrečiais intervalais ir su nebegaliniu tikslumu, gali kilti nesutarimų gylio patikrinime, kas gali privesti šešėliuotoją (angl. *shader*) patikėt, kad toj vietoj yra silueto kraštas, kai iš tikrųjų nėra. Ši problema išspręsta dedant tašką kiekvieno tekselio vidury. Jei joks taškas negali būti rastas teksely, algoritmas priima, kad jis yra centre.

Ši technika gerai tinkama grafiniams prietaisams, nors dar ir yra trūkumų šio metodo palaikyme, tačiau buvo gauti geri veikimo rezultatai (1.2.12. pav.ir 1.2.1. lent.).



1.2.12. pav. Testuojamos scenos (viršui standartinis šešėlių planas (angl. *shadow map*), apačioj – siluetų planas (angl. *siluette map*)). Scenų pavadinimai: Crowd, Jeep, Castle [8]

1.2.1. lentelė Šešėlių metodų našumo palyginimai [8]

Scenos informacija				Šešėlių planai	Šešėlių tūriai	Siluetų planai
Pavadinimas	Trikampiai	Silueto kraštai	Scenos užpildymas	Kadrai p/s	Kadrai p/s	Kadrai p/s
CROWD	14,836	4,800	148.9k	11.5	9.2	9.5
JEEP	1,732	729	298.3k	75	72	58
CASTLE	1,204	466	268.5k	97	95	80

1.2.4. Efektyvus hibridinis šešėlių vaizdavimo algoritmas

Efektyvus hibridinis šešėlių vaizdavimo algoritme (angl. *an efficient hybrid shadow rendering algorithm*) priimama, kad šviesos blokuotojai yra poligonai, gerai funkcionuojantys, uždari ir daugeriopi; šios ypatybės užtikrina tvirtą šešėlių tūrių pritaikymą [9].

Apibendrinimas šio sprendimo parodytas 1.2.13. paveiksle. Iš pradžių sukuriamas eilinis šešėlio planas, kuris padeda nustatyti šešėlio silueto pikselius. Tada naudojamas šešėlių tūriai, kad suskaičiuoti tikslius šešėlius tik ant silueto pikselių. Pagrindinė prielaida yra ta, kad silueto pikselių skaičius yra maža dalelė visų šešėlio poligono pikselių skaičiaus, ir techninis prietaisas palaiko mechanizmą efektyviai išmetamų pikselių, kurie neguli ant silueto. Algoritmo žingsniai yra:

1. Sukurti šešėlių planą. Kamera pastatoma į šviesos šaltinio tašką, kad sugeneruot artimiausia gylio reikšmes į buferį, kaip parodyta 1.2.13. a figūroj. Kadangi čia tereikia šešėlio plano tereikia aproksimuoti šešėlio siluetą, galima naudoti žemos rezoliucijos šešėlių planą, kad tausot tekstūrų atmintį ir pagreitinti šešėlių plano gaminimą. Mainai yra tokie, kad žemos rezoliucijos šešėlių planai gali praleisti mažas detales ir dažniausiai padidinti pikselių skaičių, kurie klasifikuojami kaip silueto pikseliai.

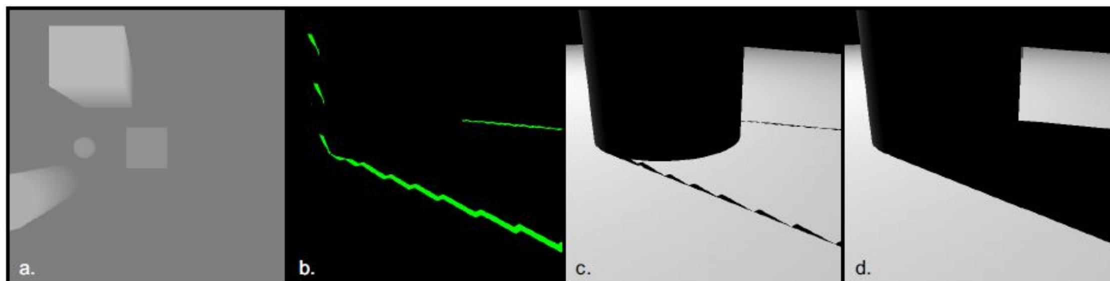
2. Identifikuoti šešėlio silueto pikselius galutiniame paveiksle. Tam vaizduojama scena iš žiūrėjimo taško ir panaudojama *Sen et al.* technika, kad surast silueto pikselius. Kiekvienas bandinys transformuojamas į šviesos erdvę ir palyginamas gyliu su artimiausiais keturiais gylio bandiniais iš šešėlio plano. Jei palyginimo rezultatai nesutampa, tai klasifikuojame bandinį kaip silueto pikselį (parodyta 1.2.13. b paveiksle). Kitaip, bandinys yra nesiluetinis pikselis ir šešėliuojamas pagal gylio palyginimo rezultatą.

Šiame žingsnyje, čia taip pat atliekamas standartinis z-buferizavimas, kuris palieka artimiausias gylio reikšmes iš žiūrėjimo taško gylio bufery. Tai paruošia gylio buferį šešėlių tūrių paišymui kitame žingsnyje.

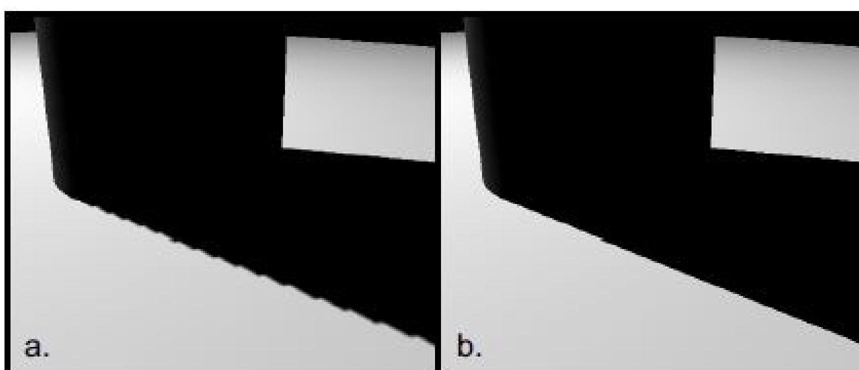
3. Nupiešti šešėlių tūrius. Šablono šešėlio tūrio algoritmas dirba padidindamas arba sumažindamas šablono buferį, pagal tai ar pikseliai šešėlio tūrio poligonų praeina ar nepraeina gylio testą. Tam sekama z-netinkamas (angl. *z-fail*) metodika apibūdinta Everitt'o ir Kilgard'o, dėl savo tvirtumo. Pagrindinis skirtumas šiame taikyme yra sutaškuoti šešėlio poligono pikselius ir atnaujinti šablono buferį tiktai kadro buferio (angl. *framebuffer*) adresuose laikančius silueto pikselius.

Šio žingsnio gale, šablono buferis turės ne-nulius pikseliams, kurie yra šešėly; nuliai bus pikseliams, kurie nėra šešėliuoti arba nėra silueto pikseliai. Pavyzdžiui, juodo šešėlio kraštai 1 c figūroje parodo sritis, kur šablono buferis laiko ne-nulius.

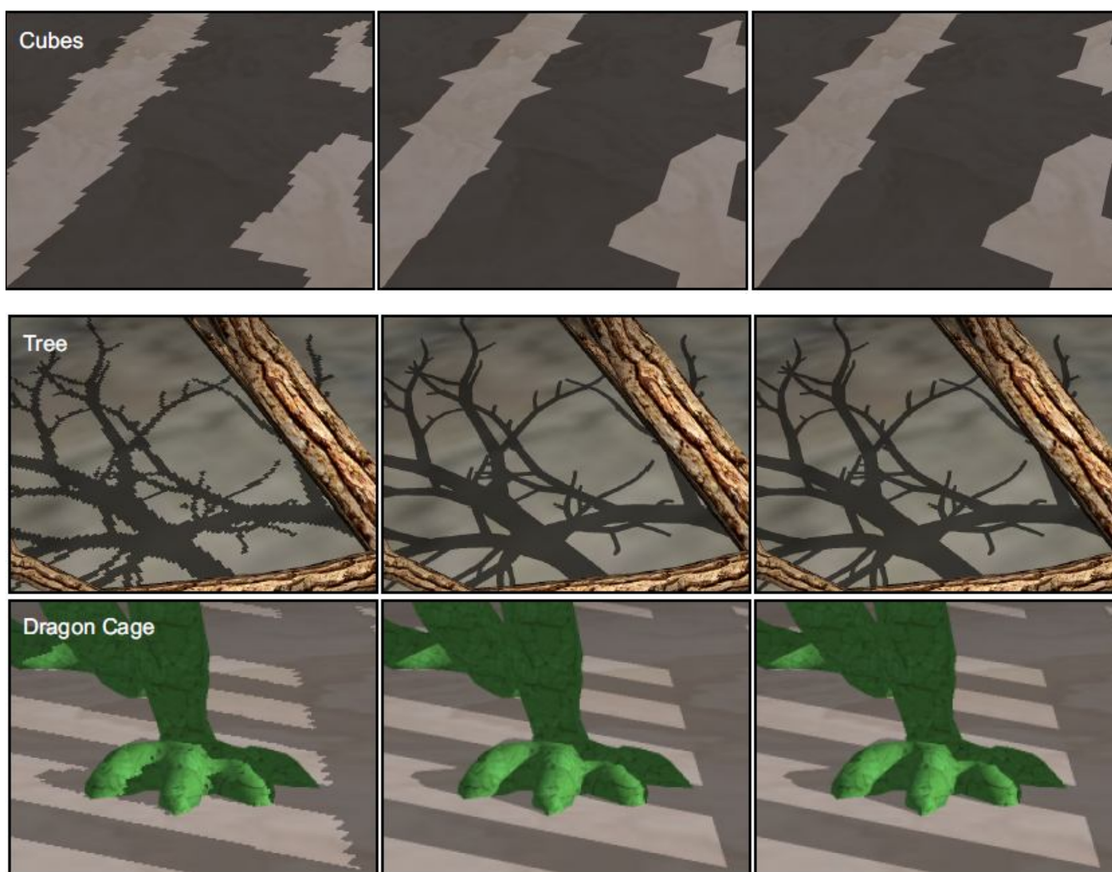
4. Suskaičiuoti šešėlius. Scena yra paišoma ir šešėliuojama ties pikseliais, kurių šablonų reikšmės lygios nuliui, tuo būdu išvengiant paveiksle šešėliuotų sričių.



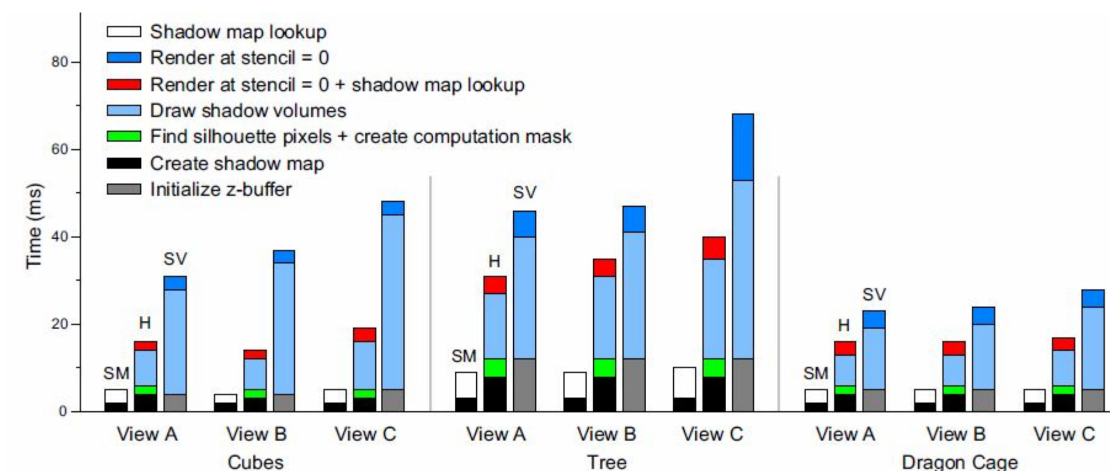
1.2.13. pav. Efektyvaus hibridinio šešėlių vaizdavimo algoritmo apibendrinimas. Pirmiausiai naudojamas šešėlių planas (a), kad identifikuoti pikselius paveiksle, kurie guli arti prie šešėlio silueto. Šie pikseliai, matomi iš žiūrėjimo taško, yra nuspalvoti žaliai b paveiksle. Toliau, generuojami šešėlių tūriai tik tuose pikseliuose, kad išgaut tikslius šešėlių kraštus (c). Toliau naudojamas šešėlio planas šešėlių apskaičiavimui visur kitur, ir galutinis rezultatas parodytas d paveiksle [9]



1.2.14. pav. (a) 512 x 512 šešėlių planas ir (b) aprašytas hibridinis algoritmas su 256 x 256 šešėlių planu. Naudojant žemesnes rezoliucijos šešėlių planą, šiame atvejyje yra priimtina, nes šešėlių tūriai rekonstruoja šešėlio siluetą [9]



1.2.15. pav. Palyginimas paveikslų kokybės naudojant šešėlių planus (kairysis stulpelis), hibridinį algoritmą (centre) ir šešėlių tūrius (dešinysis stulpelis). 1024 x 1024 rezoliucija buvo naudojama dešiniame ir centriniame atvejyje [9]



1.2.16. pav. Našumo palyginimas. Vertikalūs stulpeliai žymi kadro generavimo laiką milisekundėmis (ms) šešėlių planui (SM), hibridiniam algoritmui (H) ir šešėlių tūrių algoritmui (SV). Spalvotoji dalis žymi kiek laiko sugaištama kiekvienai algoritmo daliai [9]

1.3. Skyriaus išvados

- Šešėlių planai (angl. *shadow maps*) ir šešėlių tūriai (angl. *shadow volumes*) yra dvi populiarios technikos naudojamos realaus laiko šešėlių generavimui.
- Šešėlių palnai yra lankstūs ir efektyvūs, bet jie labai linkę į aliasing‘ą (dėl to grublėti šešėlių kraštai). Šešėlių tūriai yra tikslūs, bet reikalauja didelio pikselių generavimo dažnio (angl. *fillrate*), dėl to lėtai pateikia sudėtingas scenas. Pasiekt abiejų tikslumo ir greičio yra gana sunki užduotis realaus laiko šešėliavimo algoritmams.
- Pagrinde, hibridiniai algoritmai iš šešėlių planų ir šešėlių tūrių, bando pagerinti šešėlių tūrių kokybės bei šešėlių planų našumo santykį, t. y., pasiekti šešėlių generavimo našumą kaip šešėlių planų algoritmo, neprarandant tūrių algoritmo šešėlių kokybės.

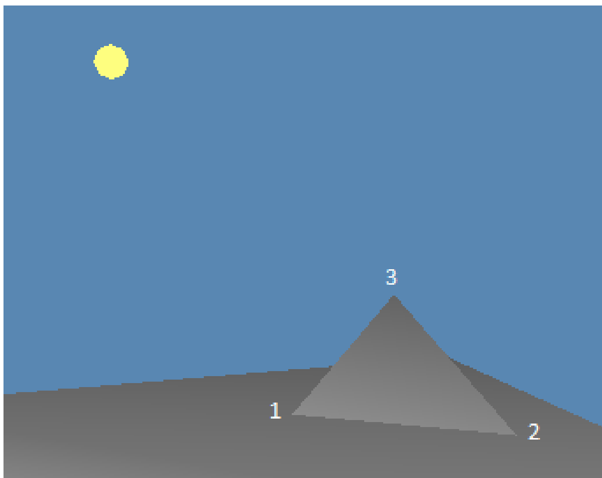
2. TIRIAMOJI DALIS

2.1. Šešėlių tūrių realizacija ir jos problemos

Šiame skyriuje aprašysime kaip praktiškai galima realizuoti. Aišku čia bus aprašomi tik pagrindai. Realizacijai panaudosime C++ kalbą, bei OpenGL bibliotekos *gl.h* ir *glu.h* modulius. Kompiliatorių naudosime MS Visual Studio 2008.

2.1.1. Realizacija

Pradžiai tarkime, kad mūsų sceną sudaro vienas šešėlio metėjas tik iš vieno trikampio poligono ir paviršius, ant kurio šešėlis krenta, bei žinoma šviesos šaltinis (2.1.1 pav.). Tarkime kad šešėlio metėjęs yra atskiras objektas-poligonas sudarytas iš viršūnių, kurios išdėstytos tinkama tvarka, kad tinkamai apibrėžtų jį, bei jam priskirto sąrašo numerio.



2.1.1. pav. Pradinė scena. Skaičiai rodo viršūnių numerius

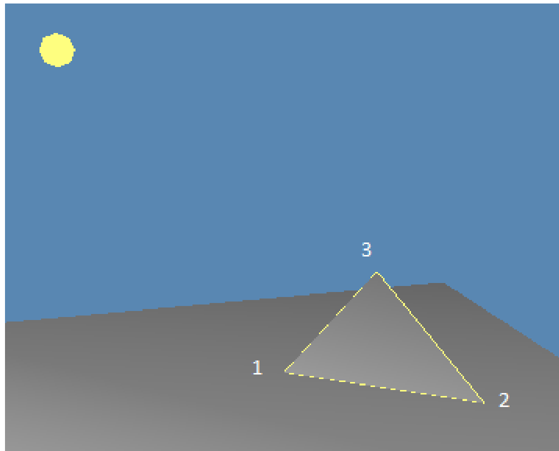
Dabar, kad galėtume piešt šešėlį, visų pirma mums reikia išskirti tūrį, į kurį papuolęs pikselis atsiduria mūsų poligono šešėlyje. Tam į pagalbą pasikviečiame OpenGL biblioteką: `glNewList` – sukuriamas naujas sąrašas, kuriame saugosime tūrį, ir tam sąrašui priskiriame numerį, pagal kurį žinosime, kad tai būtent to šešėlio metėjo sąrašas:

```
glNewList( sąrašo_nr, GL_COMPILE );
{
	// apibrėžiame tūrį
}
glEndList();
```

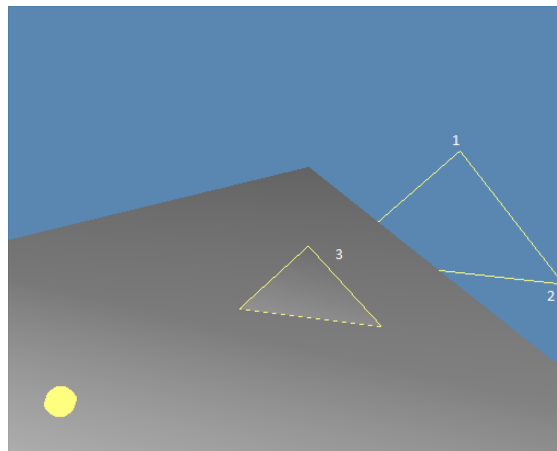
Mūsų tūriui iškirti reikės 5-penkių poligonų (viršui, apačiai ir trejų šonams), nes tūris turi būti visiškai uždaras. Dangčių žymėjimui panaudosime komandas `glVertex3f` ir `glVertex4f`. Tarkim, kad (x1, y1, z1), (x2, y2, z2), (x3, y3, z3) yra atitinkamų poligono taškų koordinatės, o (xs, ys, zs) – šviesos šaltinio koordinatės. Be to, paminėsime, kad dagčiai turi būti nubrėžti atitinkama tvarka jungiant viršūnes, kad būtų galima nustatyti, kuri dangčio pusė atsukta į vidų, kuri į išorę.

Pirma pažymėsime viršutinį dangtį, kuris yra tiesiog šešėlio metėjas. Žymėsime prieš laikrodžio rodyklę žiūrint iš šviesos šaltinio taško (2.1.2. (a) pav.):

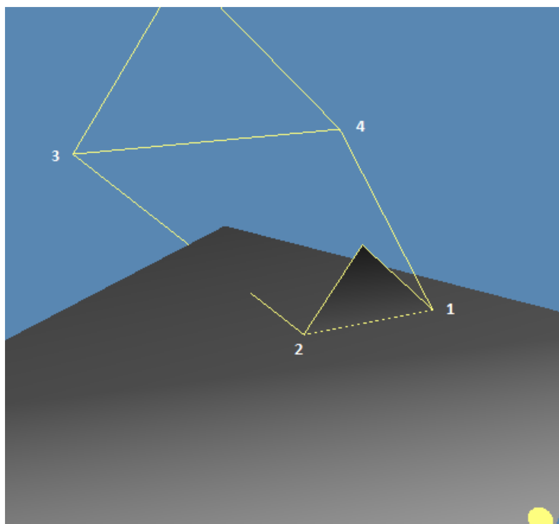
```
glBegin( GL_TRIANGLES ); // GL_TRIANGLES – nes brėžiame trikampį
{
     glVertex3f( x1, y1, z1 );
     glVertex3f( x2, y2, z3 );
     glVertex3f( x3, y3, z3 );
}
glEnd();
```



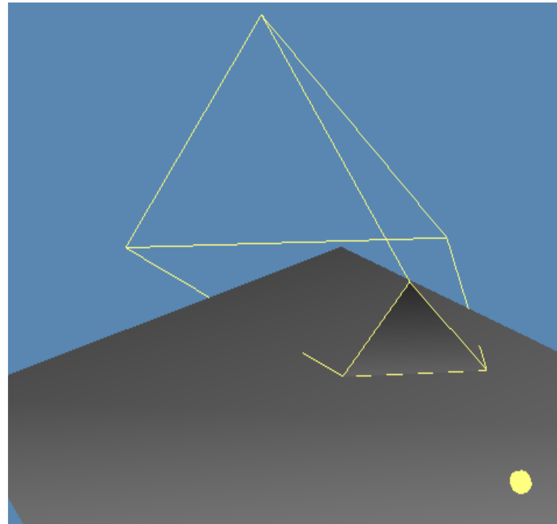
(a) Viršutinis dangtis



(b) Apatinis dangtis



(c) Šoninis dangtis



(d) Visi dangčiai uždengti

2.1.2. pav. Tūrio braižymas. Skaičiai rodo kokia tvarka apibrėžta tūrio siena

Apatinį dangtį reikia brėžti begalybėje, t. y. nubrėžti toliau už bet kokį objektą, ant kurio galėtų krist šešėlis (2.1.2. (b) pav.). Tai leis padaryti komanda `glVertex4f`, bet reikia nepamiršti, kad reikia brėžti atvirkščia tvarka žiūrint nuo šviesos šaltinio, nei viršutinysis, kad apatinis dangtis būtų atsisukęs į tūrio vidų. Be to iš viršutinio dangčio koordinačių reikia atimti šviesos šaltinio koordinates, kad komanda `glVertex4f` į tinkamą pusę nukeltų dangtį:

```
glBegin( GL_TRIANGLES ); // GL_TRIANGLES – nes brėžiame trikampį
{
      glVertex4f( x3 - xs, y3 - ys, z3 - zs, 0.0f);
      glVertex4f( x2 - xs, y2 - ys, z2 - zs, 0.0f);
      glVertex4f( x1 - xs, y1 - ys, z1 - zs, 0.0f);
}
glEnd();
```

Kai apatinis ir viršutinis dangčiai uždėti, reikia uždėti šoninius dangčius, kaip matome iš mūsų pavyzdžio, mums reikės trijų. Bet juos taip pat reikia nubrėžti tinkama tvarka, kad būtų atsisukę į vidų. Reikėtų brėžti taip, kad kraštas sueinantis su viršutinio dangčio kraštu būtų brėžiamas pagal laikrodžio rodyklę, žiūrint iš šviesos taško, o sueinantis su apatinio dangčio kraštu – atvirkščiai (2.1.2. (c) pav.):

```
glBegin( GL_QUADS ); // GL_QUADS – nes keturkampis
{
      glVertex3f( x2, y2, z2 );
      glVertex3f( x1, y1, z1 );
      glVertex4f( x1 - xs, y1 - ys, z1 - zs, 0.0f );
      glVertex4f( x2 - xs, y2 - ys, z2 - zs, 0.0f );
}
glEnd();
```

Analogiškai uždedami likę du šoniniai dangčiai ir gauname visiškai uždarą tūrį (2.1.2. (d) pav.).

Dabar kai turime šešėlio metėjo apibrėžtą turį, galime piešti patį šešėlį. Prieš paišant nauja sceną pradžiai reikia išvalyti spalvų, gylio bei šabloninį buferius:

```
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );
```

Tada sugeneruojame savo sceną be šviesos ir šešėlių, kad pradžiai užpildyti spalvų ir gylio buferius. Tada prieš pildant šabloninį buferį, kuris bus reikalingas mūsų šešėliams, išjungiame spalvų ir gylio buferių pildymą:

```
glColorMask( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE );// spalvų buferio pildymas
                                                       // sustabdomas
glDepthMask( GL_FALSE ); //gylio buferio pildymas sustabdomas
```

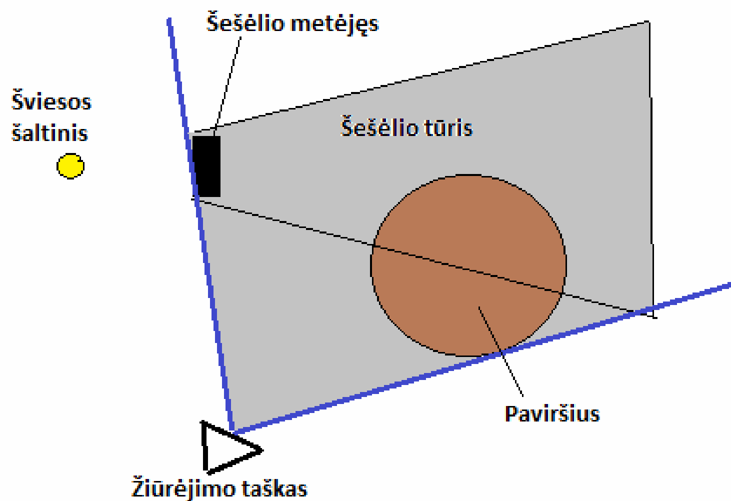
Dabar įjungiame išorinės/vidinės sienos pasirinkimą bei šabloninio buferio palyginimą:

```
        glEnable(GL_CULL_FACE);
        glEnable(GL_STENCIL_TEST);
```

Toliau, kad mūsų šešėlis būtų taisyklingas, generuosime šešėlio tūrį dviem perėjimais. Pirmas perėjimas būtų:

```
glStencilFunc( GL_ALWAYS, 0, ~0 );
glStencilOp( GL_KEEP, GL_INCR, GL_KEEP ); //didinama pikselio šabloninė reišmė
glCullFace( GL_FRONT ); // iširpti išore atsuktas sienas
glCallList( sąrašo_nr ); // iškviesti mūsų šešėlio metėjo sąrašą
```

Šis kodo fragmentas viršuje atrenka išore atsuktas iš žiūrėjimo taško tūrio sienas ir tik nubrėžia vidum į žiūrovą atsuktas sienos. Su `glCallList` nurodome savo sąrašo numerį, kuriame saugomos, mūsų tūrio sienos. Šablono operacijos (`glStencilOp`) viršuj pagrinde reiškia tai, kad kai gylio testas paveda (t. y. poligono pikselis yra toliau nei pikselis, kuris jau laikomas jo vietoj ant ekrano), mes padidiname pikselio šabloninę vertę. Kad būtų lengviau suprast, pažiūrėkit į paveikslą apačioje:

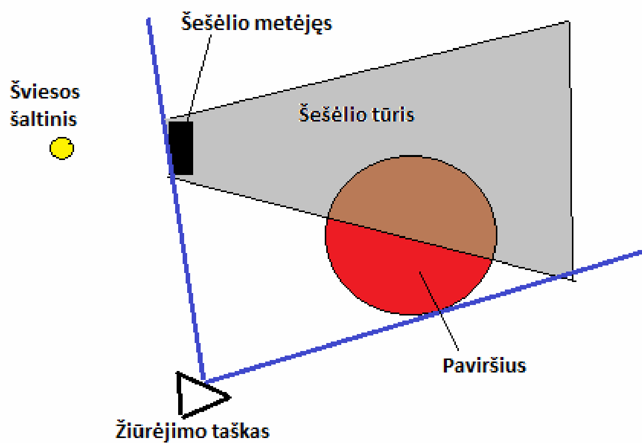


2.1.3. pav. Po pirmo perėjimo [12]

2.1.3 paveiksle pilka spalva rodo, kurių pikselių šabloninė reikšmė yra padidinama, t. y. pikseliai atsidūrę tarp akies ir tūrio vidinių sienų. Tada antras perėjimas būtų toks:

```
glStencilOp( GL_KEEP, GL_DECR, GL_KEEP ); //mažinama pikselio šabloninė reikšmė
glCullFace( GL_BACK ); // iškirpti vidum atsuktas sienas
glCallList( sąrašo_nr ); // iškviesti mūsų šešėlio metėjo sąrašą
```

Šis kodo fragmentas nurodo, kad pikselių esančių tarp akies ir priekiu atsuktų sienų pikselių šablonines reikšmes pamažinti, tokiu būdų tik pikseliai esantys tūrio viduj turi didžiausias reikšmes. Mūsų 2.1.3 paveikslas po antro perėjimo atrodytų atrodytu taip (2.1.4 pav.):



2.1.4. pav. Po antro perėjimo [12]

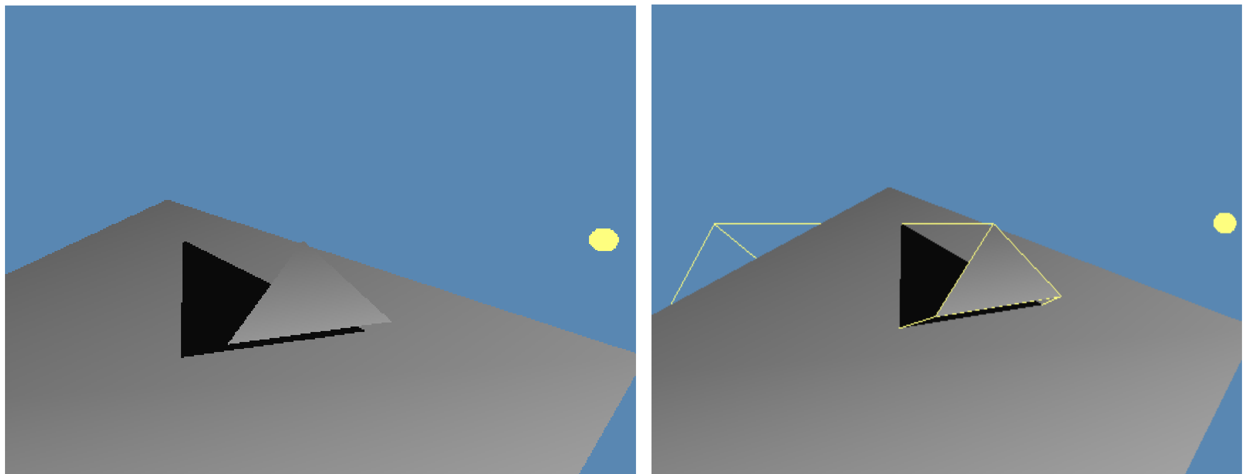
Dabar kai šabloninės pikselių reikšmės nustatytos atstatome pradines reikšmes:

```
glDepthMask( GL_TRUE ); // vėl įjungti gylio buferį
glDepthFunc( GL_LEQUAL );
glColorMask( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE ); // vėl įjungt spalvų buferį
glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP ); // šabloninių reikšmių nebekeisti
glDisable( GL_CULL_FACE ); // nebeišrinkti jokių sienų
glStencilFunc( GL_EQUAL, 0, ~0 );
```

Dabar vėl sugeneruojame savo sceną, tik jau dabar joje bus mūsų šešėliai. O po scenos galutinio generavimo išjungiam šabloninio buferio palyginimą:

```
glDisable( GL_STENCIL_TEST );
```

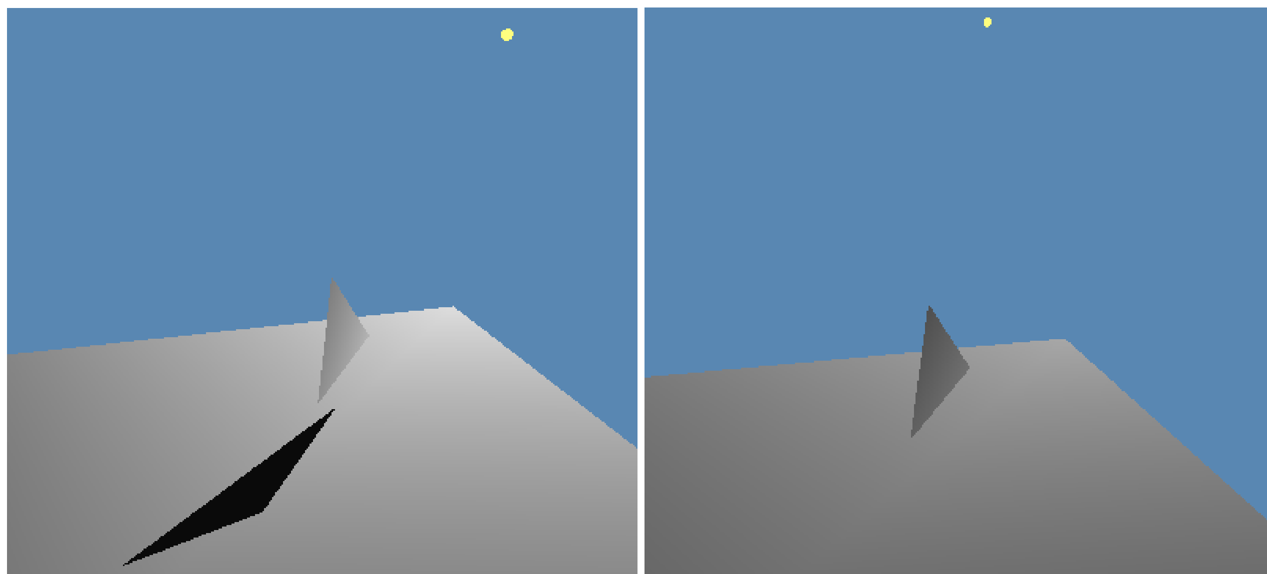
Ir pagaliau gauname sceną su šešėliu:



2.1.5. pav. Scena su mūsų šešėliu (dešinėj pusėj parodyti tūrio sienų kontūrai)

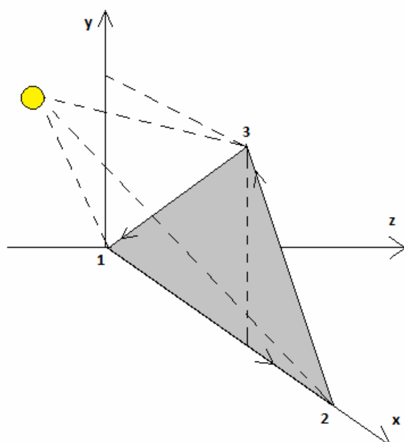
Iš pradžių atrodytų kaip ir viskas tvarkoj, bet šviesos šaltiniui atsidūrus kitoj mūsų šešėlio metėjo pusėj, šešėlis dingsta (2.1.6. pav.). Kodėl taip atsitiko? Todėl, kad mūsų šviesos šaltiniui perėjus į kitą pusę

mūsų tūris „išsiverčia“, t. y. kur buvo vidinės sienos pusės tapo išorinėm, o išorinės – vidinėm, nes kaip minėjome dagčiai turi būti apibrėžti tam tikra tvarka (viršutinis prieš laikrodžio rodyklę, apatinis – pagal,



2.1.6. pav. Šviesos šaltiniui atsidūrus kitoj poligono pusėj, šešėlis dingsta

žiūrint iš šviesos šaltinio taško), ir mūsų dviejų perėjimų algoritmas tampa nebetinkamas. Kad to išvengti mums kažkaip reikia visų pirma nustatyti, kurioj mūsų poligono plokštumos pusėj yra šviesos šaltinis. Tam reikalui panaudojome trigonometriją. Tarkim mūsų poligono pirmos ir antros viršūnės kraštinė yra ant x ašies, t. y. $x1<x2$, $y1=0$, $y2=0$, $z1=0$, $z2=0$ ((x1, y1, z1), (x2, y2, z2), (x3, y3, z3) yra atitinkamų poligono viršūnių koordinatės), o trečioji poligono viršūnė yra xy plokštumoje, t. y. $z3=0$, be to, $y3>0$, tada mums būtų nesunku nustatyti, kurioj pusėj yra šviesa, o svarbu svarbiausia, kokia tvarka apjungti viršūnes paišant dangčius (2.1.7. pav.).



2.1.7. pav. Poligono viršūnės apjungiamos prieš laikrodžio rodyklę brėžiant viršutinį dangtį

Kai poligonas xy plokštumoj, bei viršūnės išsidėsčiusios tokia tvarka kaip 2.1.7 pav., tada tiesiog pažiūrim į šviesos šaltinio z koordinatę, jei z<0, tai viršūnes piešdami viršutinį dangtį apjungiam tokia tvarka: 1, 2, 3, o jei z>0, apjungiame atvirkščia tvarka: 3, 2, 1. Problema ta, kad poligonas gali būti bet kurioje vietoje ir, bet kaip pasviręs. Tačiau mes galime sužinoti, kiek ir apie kokią ašį reikia pasukt poligono viršūnes, kad mūsų poligonas atsidurtų mums patogioje padėtyje (kaip 2.1.7 pav.), ir lygiai tokius pat pasukimus atlikti su šviesos šaltinio tašku, kad santykis tarp šviesos šaltinio ir poligono koordinačių išliktų toks pat. Tam pritaikėme tokį algoritmą:

1. 1-mos viršūnės koordinates pastatyti į sistemos pradžią. T. y. pirmos viršūnės koordinatės bus V1(0 , 0, 0), tada antros viršūnės koordinatės transformuojamos taip - V2(x2-x1, y2-y1, z2-z1), trečiosios - V2(x3-x1, y3-y1, z3-z1), o šviesos šaltinio VS(xs-x1, ys-y1, zs-z1) (1.2.8. (b) pav.).
2. Pasukam viršūnes aplink z ašį (į z koordinates dabar nekreipiam dėmesio) tiek, kad V2 atsidurtų x plokštumoje, t. y. y2=0 (2.1.8. (c) pav.). Matematiškai V2, V3, VS koordinačių transformacija būtų:

   $r2_{xy} = \sqrt{x2^2 + y2^2}$; $r3_{xy} = \sqrt{x3^2 + y3^2}$; $rs_{xy} = \sqrt{xs^2 + ys^2}$;

   $r2_{xy}$, $r3_{xy}$ ir rs_{xy} – vektorių išvestų iš atskaitos sistemos pradžios link atitinkamų taškų ilgiai xy plokštumoje.

   $x2 = r2_{xy}$; $x3 = r3_{xy} * \cos(\beta 3_{xy} - \beta 2_{xy})$; $xs = rs_{xy} * \cos(\beta s_{xy} - \beta 2_{xy})$;

   $y2 = 0$; $y3 = r3_{xy} * \sin(\beta 3_{xy} - \beta 2_{xy})$; $ys = rs_{xy} * \sin(\beta s_{xy} - \beta 2_{xy})$;

   $\beta 2_{xy}$ - kampas tarp vektoriaus $\vec{r2}_{xy}$ ir 0x ašies xy plokštumoje, $\beta 3_{xy}$ ir βs_{xy} irgi taip pat atitinkamai.
3. Dabar, kad pirmos ir antros viršūnių kraštinė atsidurtų 0x ašyje mums bereikia taškus apsukti aplink y ašį (į y koordinates dabar nekreipiam dėmesio) (2.1.8. (d) pav.). Matematiškai V2, V3, VS koordinačių transformacija būtų:

   $r2_{xz} = \sqrt{x2^2 + z2^2}$; $r3_{xz} = \sqrt{x3^2 + z3^2}$; $rs_{xz} = \sqrt{xs^2 + zs^2}$;

   $r2_{xz}$, $r3_{xz}$ ir rs_{xz} – vektorių išvestų iš atskaitos sistemos pradžios link atitinkamų taškų ilgiai xz plokštumoje.

   $x2 = r2_{xz}$; $x3 = r3_{xz} * \cos(\beta 3_{xz} - \beta 2_{xz})$; $xs = rs_{xz} * \cos(\beta s_{xz} - \beta 2_{xz})$;

   $z2 = 0$; $z3 = r3_{xz} * \sin(\beta 3_{xz} - \beta 2_{xz})$; $zs = rs_{xz} * \sin(\beta s_{xz} - \beta 2_{xz})$;

$\beta 2_{xz}$ – kampas tarp vektoriaus $\vec{r2}_{xz}$ ir 0x ašies xz plokštumoje, $\beta 3_{xz}$ ir βs_{xz} irgi taip pat atitinkamai.

4. Dabar mums beliko apsukti 3-čia viršūnę aplink x ašį (į x koordinates dabar nekreipiam dėmesio), kad mūsų poligonas atsidurtų reikiamoje padėtyje, kaip 2.1.7 paveiksle, aišku nepamiršti tokia pat transformacija atlikti ir su šviesos šaltinio koordinatėm (2.1.8. (e) pav.).

   $r3_{yz} = \sqrt{y3^2 + z3^2}$; $rs_{yz} = \sqrt{ys^2 + zs^2}$;

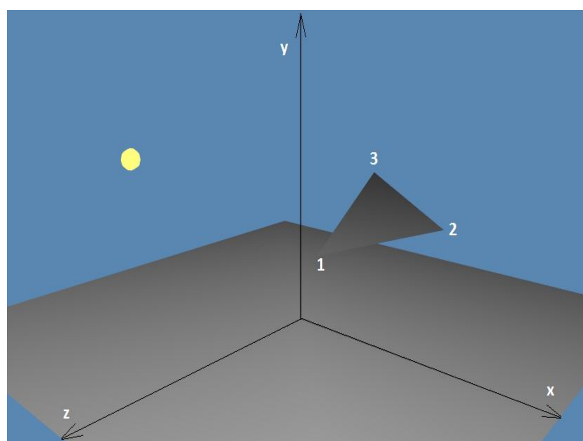
   rs_{yz} – vektoriaus išvesto iš atskaitos sistemos pradžios link šviesos šaltinio taško ilgis yz plokštumoje, $r3_{yz}$ – atitinkamai iš trečios viršūnės.

   $y3 = r3_{yz}$; $ys = rs_{yz} * \cos(\beta s_{yz} - \beta 3_{yz})$;

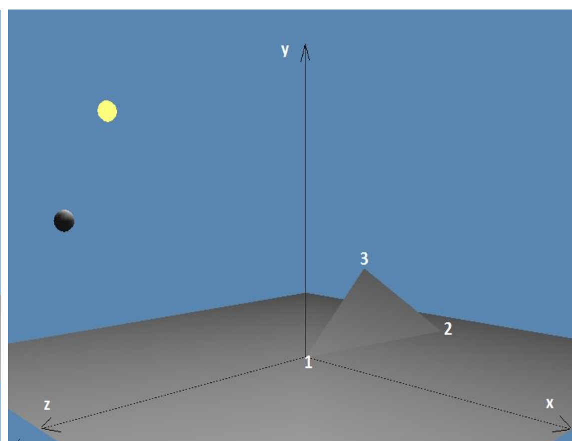
   $z3 = 0$; $zs = rs_{yz} * \sin(\beta s_{yz} - \beta 3_{yz})$;

   $\beta 3_{xz}$ – kampas tarp vektoriaus $\vec{r3}_{yz}$ ir 0z ašies yz plokštumoje, βs_{xz} - atitinkamai.

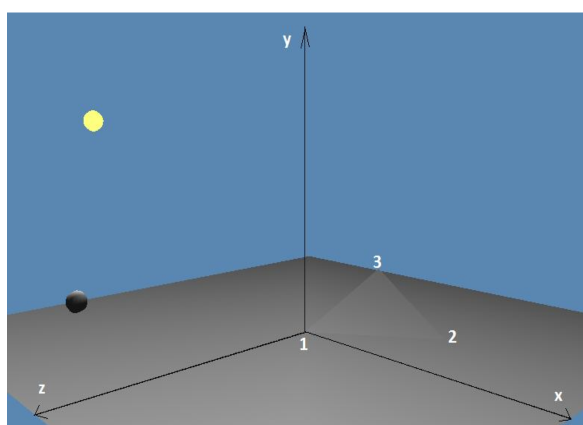
5. Dabar kai tinkamai transformavome koordinates, mums belieka pažiūrėt į šviesos šaltinio z koordinatės reikšmę zs, jei zs > 0, tai viršūnes apjungiame paišant viršutinį dangtį tokia tvarka: 1, 2 ir 3; jei zs < 0 apjungiame atvirkščia tvarka: 3, 2, 1.



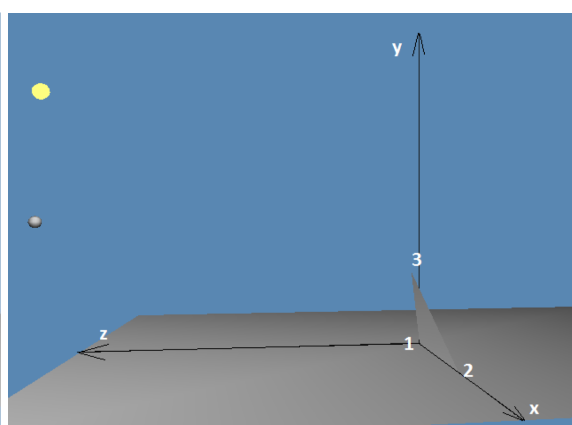
(a) Pradinė padėtis



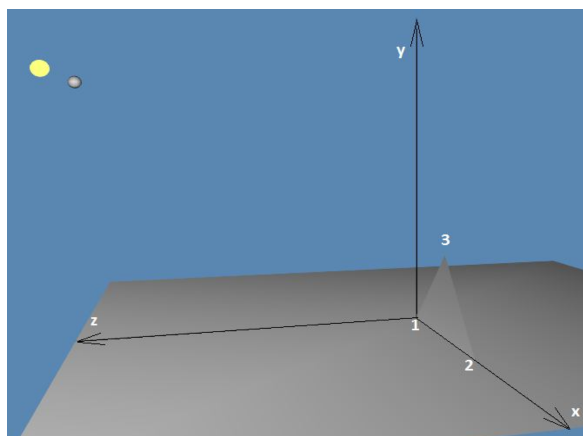
(b) Po pirmo žingsnio



(c) Po antro žingsnio



(d) Po trečio žingsnio



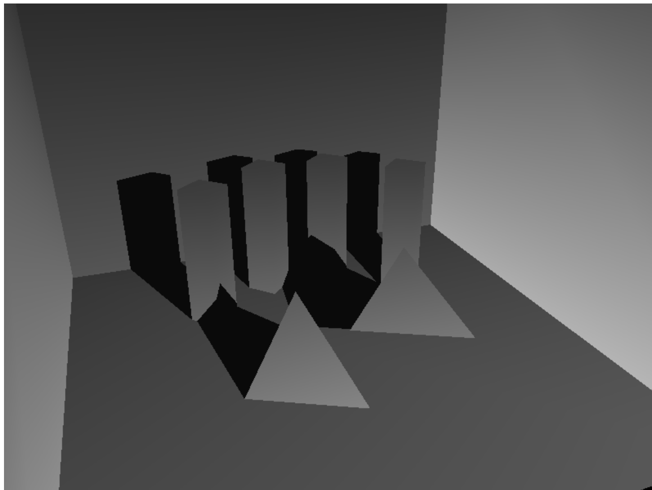
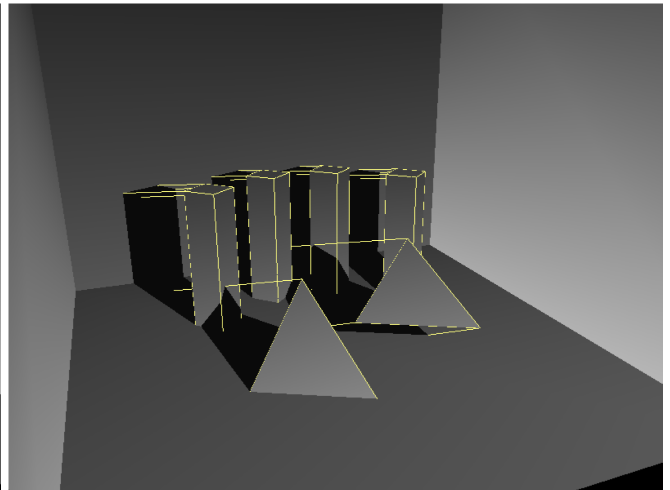
(e) Po ketvirto žingsnio

2.1.8. pav. Koordinačių transformacija pastatant poligoną mums į tinkamą padėtį. Geltonas ritulys – šviesos šaltinis, pilkas ritulys – šviesos šaltinio koordinatės po tam tikros transformacijos. Pilko rutulio ir poligono santykis (e) paveiksle lygiai toks pat, kaip ir (a) pradiniame paveiksle tarp šviesos šaltinio ir poligono pradinėje padėtyje

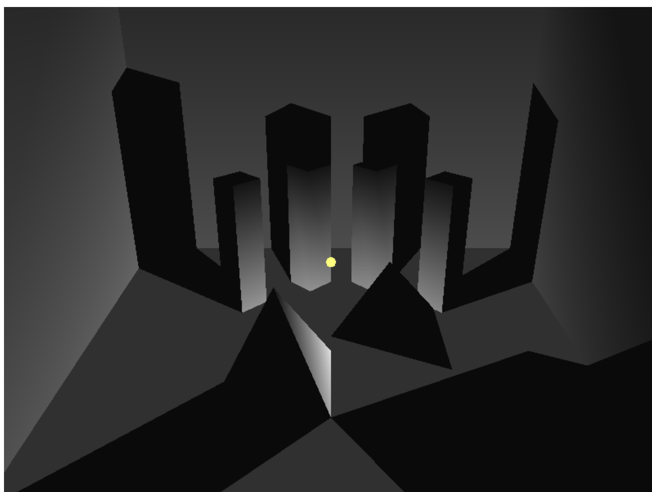
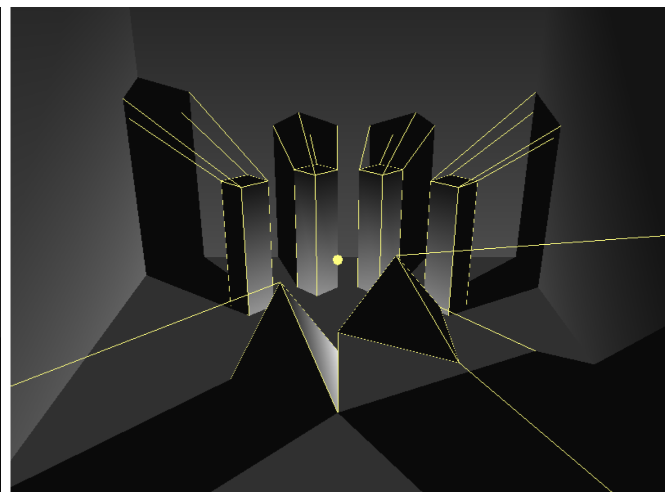
Dabar kai mūsų poligonas tinkamai metą šešėlį, galime dėlioti sceną iš daugybės poligonų. Mūsų realizuotoje programoje – kiekvienas poligonas yra atskiras objektas, nepriklausomas šešėlio metėjęs, t. y. kiekvienam poligonui visa mūsų aprašyta procedūra atliekama atskirai.

2.1.2. Testavimas ir išvados

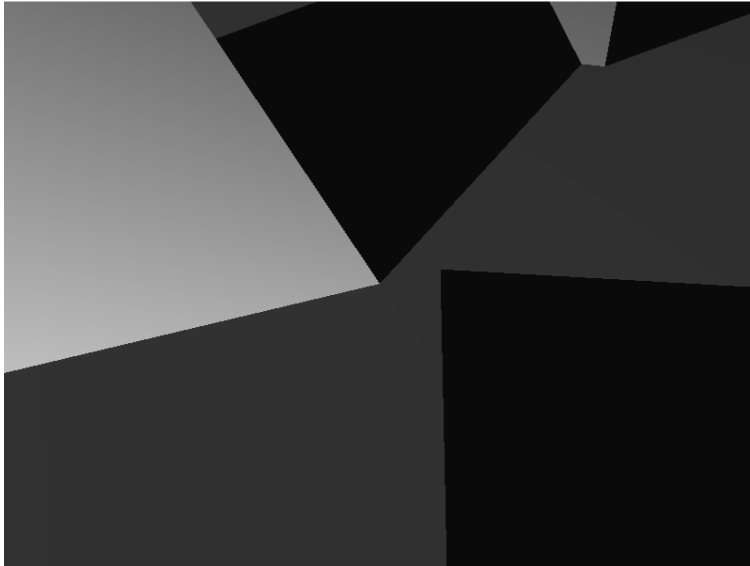
Tūrinių šešėlių kokybė yra palyginti gera, nesvarbu ar šviesos šaltinis būtų (2.1.9. pav.) toli ar prie pat objekto (2.1.10. pav.) šešėlio kontūras labai tikslus, be jokių iškraipymų ir nedantytas (be *aliasing'o*) (2.1.11. pav.). Be to tūriniai šešėliai be jokių problemų susidoroja su realaus laiko šešėliams palyginti sunkia scena, kai šviesos šaltinis yra objekto viduje (2.1.12. pav.).

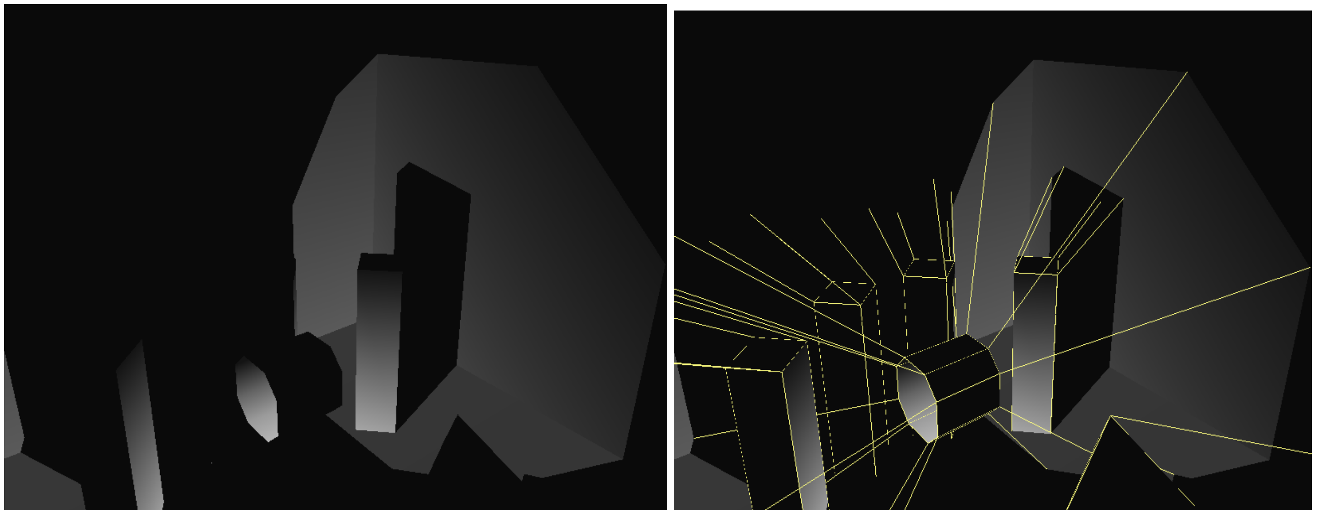
2.1.9. pav. Scena kai šviesos šaltinis toli. Dešinėj pusėj nupiešti tūrių kontūrai

2.1.10. pav. Scena kai šviesos šaltinis prie pat. Dešinėj pusėj nupiešti tūrių kontūrai



2.1.11. pav. Tūrinių šešėlių kontūrai iš arti

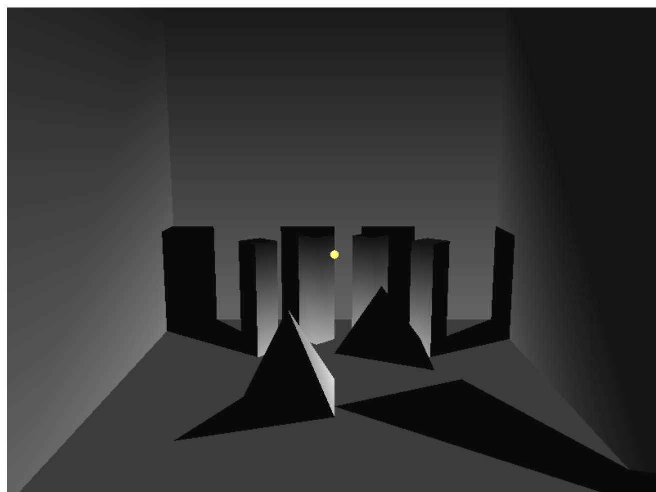


2.1.12. pav. Šviesos šaltinis vamzdyje. Ši šešėliams sunki scena – ne kliūtis tūriniams šešėliams. Dešinėj pusėj nupiešti tūrių kontūrai

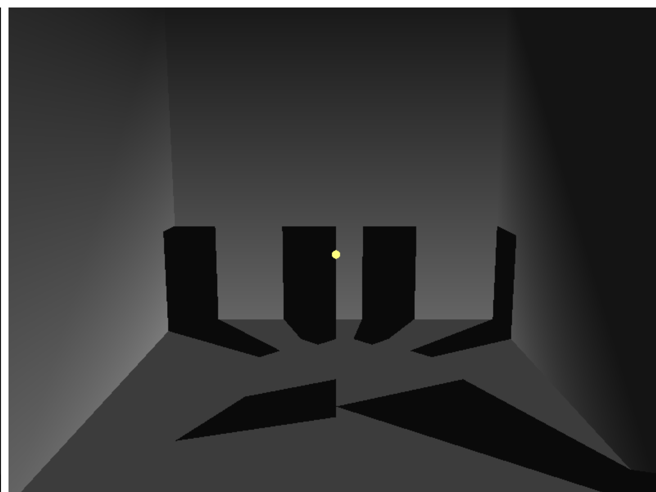
Labai tikslūs šešėlių kontūrai, gali būti netik tūrinių šešėlių privalumas, bet, galima sakyti, ir trūkumas, nes realiuose vaizduose šešėlių kontūrai būna paprastai neryškūs. Tai puikiai iliustruoja 1. paveikslas (žr. 5 pusl.), kaip matyti scena su neryškiais šešėlių kontūrais atrodo žymiai realesnė. Šiai problemai spręsti, žinoma, yra prikurta įvairių minkštų šešėlių algoritmų, tačiau jie dar labiau, savaime suprantama, sulėtina, palyginti ir taip lėtokai generuojamą tūrinių šešėlių sceną.

Kita, galima sakyti, problema būtų, kad tūriniai šešėliai tiesiogiai nepriklauso nuo scenos poligonų, t. y. jei objektas pakeitė koordinates, jo šešėlis liks toj pačioj vietoj jei į tai neatsižvelgsime

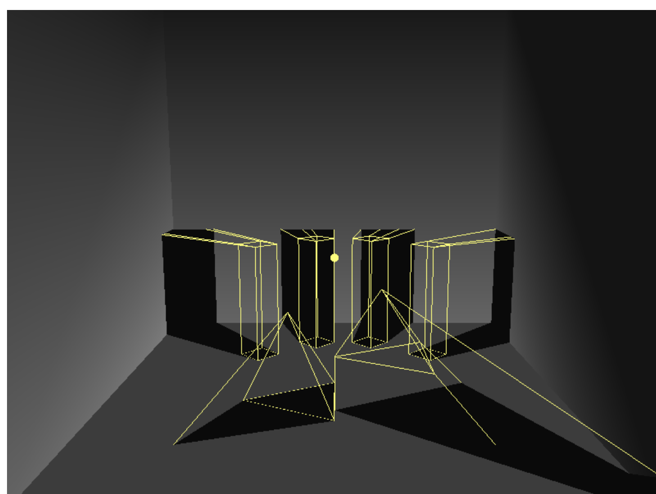
(2.1.13. pav.). Tai gali sukelti papildomų problemų, ypač jei kūnas ne tik keičia savo padėtį scenoje, bet ir formą (pvz. animacija).



(a) Pilna scena



(b) Scena tik su šešėliais



(c) Scena su nubrėžtais tūriais

2.1.13. pav. Šešėliai nepriklauso nuo nupieštų poligonų, bet tik nuo pažymėtų tūrių

2.2. Šešėlių plano realizacija ir jos problemos

Šiame skyriuje aprašysime kaip praktiškai galima realizuoti šešėlių planus. Aišku čia bus aprašomi tik pagrindai. Realizacijai panaudosime C++ kalbą, bei OpenGL bibliotekos *gl.h, glu.h, glext.h* ir *wglext.h* modulius. Kompiliatorių naudosime MS Visual Studio 2008.

2.2.1. Realizacija

Šešėlių plano technika pagrinde naudoja gylio tekstūrą, kad išsiaiškintų ar pikselio z reikšmė pastato jį į šešėlio dengiamą vietą ar ne. Tai įtraukia du perėjimus. Pirmasis būtų panaudoti p-buferį, kad sukurt gylio tekstūrą iš šviesos šaltinio žiūrėjimo taško. Antrame perėjime būtų sugeneruoti 3D turinį ir atlikti gylio palyginimą kiekvienam pikseliui su gylio tekstūra. Jei pikselio z reikšmė yra didesnė nei gylio tekstūros, tai reiškia pikselis yra toliau nuo šviesos šaltinio, t. y. jis yra šešėly. Ir atvirkščiai, jei pikselio z reikšmė mažesnė – tai pikselis arčiau šviesos šaltinio, nei koks nors objektas metantis šešėlį iš kurio buvo sudaryti gylio tekstūra.

Pradžioj p-buferio organizavimui panaudosime tokią struktūrą:

```
struct PBuferis
{
    HPBUFFERARB hPBuferis;// Panaudoti p-buferyje.
    HDC         hDC;      // Panaudoti prietaiso kontekste.
    HGLRC       hRC;      // Panaudoti grafinio vaizdavimo lango kontekste.
    int         Plotis;   // P-buferio plotis.
    int         Aukstis;  //.
};
```

Toliau nustatome p-buferio pradines reikšmes, kad vėliau jį galėtume pildyti. P-buferio paruošimo metodas atrodytų taip:

```
void ParuostiPBuferi()
{
    PBuferis.hPBuferis = NULL; // Pradžioj p-buferis tuščias.
    PBuferis.Plotis   = 1024; // Kuo didesnis p-buferio dydis (Plotis*Aukštis)
    PBuferis.Aukštis  = 1024; // tuo šešėliai bus tikslesni, bet lėtesni.
```

Nustatome minimalius pikselio formato reikalavimus mūsų p-buferiui. P-buferis yra kaip kadrų buferis, jis gali turėti gylio buferį susijusį su juo:

```
int pf_atributai[] =
{
    WGL_SUPPORT_OPENGL_ARB, TRUE,        // P-buferis bus panaudotas su OpenGL.
    WGL_DRAW_TO_PBUFFER_ARB, TRUE,       // Leisti generavimą į p-buferį.
    WGL_BIND_TO_TEXTURE_DEPTH_NV, TRUE, // Reikalauti gylio tekstūros.
    WGL_BIND_TO_TEXTURE_RGBA_ARB, TRUE, // P-buferis bus panaudotas kaip tekstūra.
    WGL_DOUBLE_BUFFER_ARB, FALSE,        // Mums nereikia dvigubo buferiavimo.
```

```
    0                                   // Nulis nutraukia sąrašą.
};
int pixelFormat;
unsigned int count = 0;

if( !wglChoosePixelFormatARB( g_hDC, pf_atributai, NULL, 1, &pixelFormat, &count ) )
{
   MessageBox(NULL,"pbuferio kūrimo klaida: nepavyko parinkti pikselių formato.",
                           "ERROR",MB_OK|MB_ICONEXCLAMATION);
   exit(-1);
}
```

Nustatome kelėta p-buferio atributų, kad jį galėtume naudoti kaip 2D RGBA tekstūrų objektą:

```
int pb_atributai[] =
{
   WGL_DEPTH_TEXTURE_FORMAT_NV, WGL_TEXTURE_DEPTH_COMPONENT_NV,// Mums reikia
                                                  //sugeneruot į gylio tekstūrą.
   WGL_TEXTURE_FORMAT_ARB, WGL_TEXTURE_RGBA_ARB, // mūsų buferis turės RGBA tekstūrų
                                                 // formatą.
   WGL_TEXTURE_TARGET_ARB, WGL_TEXTURE_2D_ARB,   // Tekstūrų objektas bus
                                                 // GL_TEXTURE_2D.
   0                                             // Nulis nutraukia sąrašą.
};
```

Priskiriame mūsų p-buferiui reikšmes:

```
  PBuferis.hPBuferis = wglCreatePbufferARB( hDC, pixelFormat, PBuferis.Plotis,
                                                  PBuferis.Aukstis, pb_atributai );
  PBuferis.hDC      = wglGetPbufferDCARB(PBuferis.hPBuferis );
  PBuferis.hRC      = wglCreateContext(PBuferis.hDC );

} // ParuostiPBuferi() metodo pabaiga.
```

Kai p-buferis paruoštas galime pradėti gylio tekstūrų kūrimą. Tam panaudosime metodą SukurtiGyliotekstura():

```
void SukurtiGylioTekstura()
{
```

Pradžioj padarom p-buferio kontekstą einamuoju:

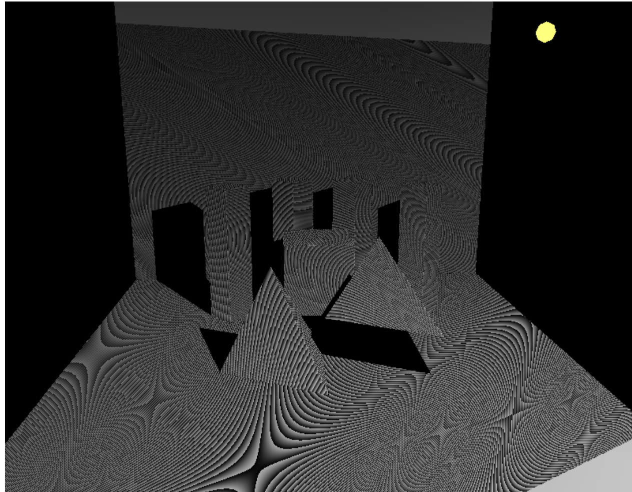
```
   if( wglMakeCurrent(PBuferis.hDC, PBuferis.hRC) == FALSE )
       {
      MessageBox(NULL,"Neišėjo padaryt p-buferio konteksto einamuoju!",
                         "ERROR",MB_OK|MB_ICONEXCLAMATION);
      exit(-1);
     }
```

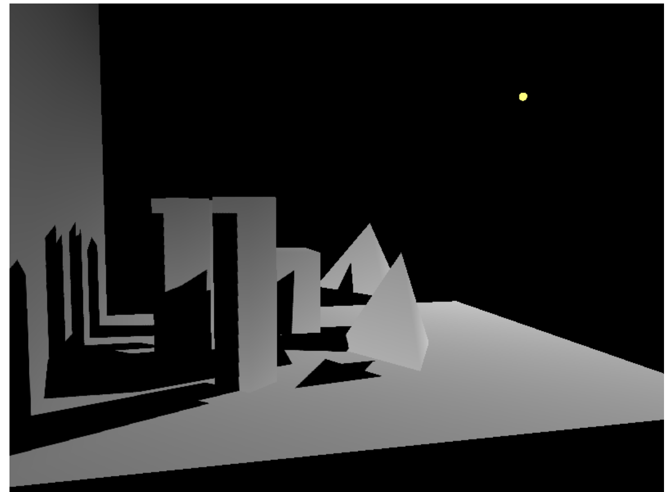
Išvalome gylio ir spalvų buferius:

```
      glClear( GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT );
```

Su komanda glPolygonOffset nustatome skalę ir vienetus gylio reikšmių skaičiavimui, kad atitrauktų šešėlio pikselius toliau nuo poligonų ir išvengtume siūlių (*stitching*) efekto, 2.2.1. (a) paveikslas iliustruoja, kai ši komanda nėra įtraukta. Bet negalima parinkti ir per didelės skalės reikšmės, nes tada šešėliai bus atitraukti nuo kampų kaip 2.2.1. (b) paveiksle.

```
glPolygonOffset( 2.0f, 4.0f );
glEnable( GL_POLYGON_OFFSET_FILL ); // Įjungiame glPolygonOffset funkciją.
```

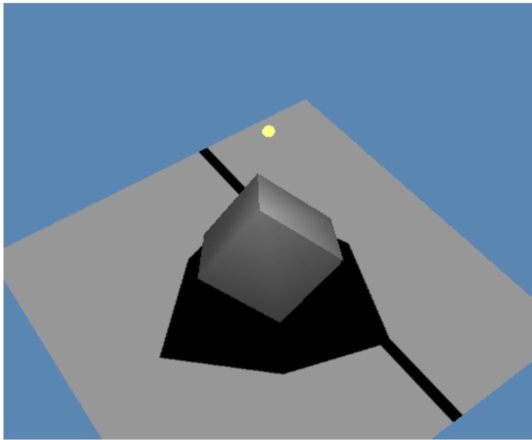

(a)


(b)

2.2.1. pav. Kai funkcija glPolygonOffset nėra panaudota arba skalės reikšmė per maža (kairėje). Ir kai reikšmė paimta per didelė (dešinėje)

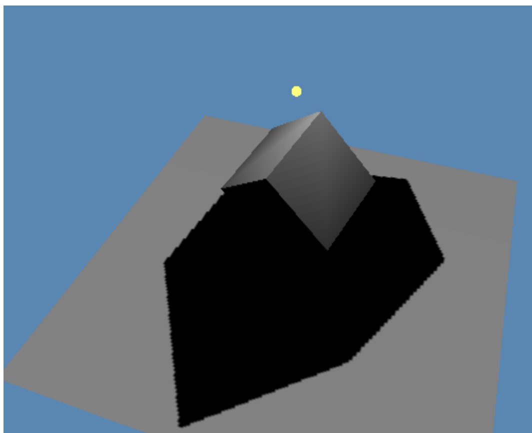
Kadangi, kaip minėjome, gylio tekstūros sudaromos iš šviesos žiūrėjimo taško. Su gluPerspective nustatome žiūrėjimo perspektyvą, o su glViewport mūsų p-buferio didį. Aišku kuo paimsime didesnę perspektyvą, tuo daugiau scenos įtrauksime į p-buferį, bet tuo pačiu pablogės mūsų šešėlių kokybė, nes gylio tekstūra turės padengti didesnį plotą, t. y. viena gylio tekstūra apims daugiau pikselių. Tai puikiai iliustruoja 2.2.2. paveikslas. 2.2.2. (a) matome kaip kubo iš dviejų kampų šešėlis išsitempė, taip įvyko dėl to, kad, kaip matome 2.2.2. (a) pav., kubas neįtilpo į šviesos regos lauką. Tačiau padidinus šviesos šaltinio regos lauką, kubas metą teisingai šešėlį, netgi šviesos šaltiniui būnant arčiau nei prieš tai (2.2.2. (c) pav.), nes kaip matome iš 2.2.2. (d) paveikslo jis tilpo į regos lauką. Tačiau šešėlio kokybė pablogėjo, taip sakant, pradėjo matytis *aliasing*'as.

```
glLoadIdentity();
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
gluPerspective( 95.0f, 1.5f, 0.5f, 100.0f); //Žiūrėjimo perspektyvos nustatymas.
glViewport( 0, 0, PBuferio.Plotis, PBuferio.Aukštis);
glMatrixMode( GL_MODELVIEW );
glMultMatrixf( lightsLookAtMatrix); //Žiūrėti iš šviesos šaltinio.

GeneruotiScena(); // Generuojam savo scenos polygonus.
glDisable( GL_POLYGON_OFFSET_FILL ); //Išjungiame glPolygonOffset funkciją.
```



(a) Iš siauresnio regos lauko šešis

(b) Gylio tekstūra



(c) Iš platesnio regos lauko šešis

(d) Gylio tekstūra

2.2.2. pav. Šviesos regos lauko įtaka šešėliams. (a) – siauresnio regos lauko šešėlis, (c) – platesnio, (b) ir (d) atitinkamai jų gylio tekstūros iš šviesos šaltinio regos lauko

Dabar kai užpildėme p-buferį, padarome vaizdavimo sąsajos kontekstą vėl aktyvų vietoj p-buferio konteksto:

```
    if( wglMakeCurrent( g_hDC, g_hRC ) == FALSE )
    {
            MessageBox(NULL,"Neišėjo lango kontekstą padaryt einamuoju!",
                  "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
    }
} // SukurtiGylioTekstura() metodo pabaiga.
```

Dabar, kai turime nustatę gylio tekstūras, galime piešti galutinę sceną su šešėliais. Tarkim mūsų galutinės scenos generavimo metodas bus GeneruotiGalutineScena():

```
void GeneruotiGalutineScena()
{
```

Pradžiai sukūriame šviesos žiūrėjimo matricą, reikalingą gylio tekstūros generavimui:

```
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    gluLookAt( Sviesos_x_koordinate, Sviesos_y_koordinate, Sviesos_z_koordinate,
                                            // Žiūrėt iš švisos pozicijos
               0.0f, 1.0f, 0.0f,   // Nurodom taško koordinates į kurį žūrės šviesa.
               0.0f, 0.1f, 0.0f ); // žiūrėjimo vektorius.
    glGetFloatv( GL_MODELVIEW_MATRIX, lightsLookAtMatrix);
```

Kai žiūrėjimo matrica paruošta, galime vykdyti jau ankščiau aprašytą gylio tekstūros kūrimo metodą:

```
SukurtiGylioTekstura();

glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT ); // Išvalome spalvų bei gylio
                                                      // buferius.
```

Pastatome žiūrėjimo tašką iš kur žiūrės žiūrovas (o ne šviesos šaltinis):

```
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    glTranslatef( 0.0f, -2.0f, -12.0f );// žiūrėjimo taško koordinatės
```

Toliau skaičiuosime glTexGen matricą, kurią panaudosime tekstūrų koordinavimui ir planinių šešėlių projektavimui į mūsų sceną. S, T, R, Q - žiūrėjimo piramidės ribojamos plokštumos:

```
    float S_plokstuma[] = { 1.0f, 0.0f, 0.0f, 0.0f };
    float T_plokstuma[] = { 0.0f, 1.0f, 0.0f, 0.0f };
    float R_plokstuma[] = { 0.0f, 0.0f, 1.0f, 0.0f };
    float Q_plokstuma[] = { 0.0f, 0.0f, 0.0f, 1.0f };
    glTexGenfv( GL_S, GL_EYE_PLANE, S_plokstuma );
    glTexGenfv( GL_T, GL_EYE_PLANE, T_plokstuma );
    glTexGenfv( GL_R, GL_EYE_PLANE, R_plokstuma );
    glTexGenfv( GL_Q, GL_EYE_PLANE, Q_plokstuma );
    glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
    glTexGeni( GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
    glTexGeni( GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
    glTexGeni( GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
    glEnable( GL_TEXTURE_GEN_S );
    glEnable( GL_TEXTURE_GEN_T );
    glEnable( GL_TEXTURE_GEN_R );
    glEnable( GL_TEXTURE_GEN_Q );
```

Nustatome gylio tekstūrų proktavimą:

```
    glMatrixMode( GL_TEXTURE );
    glLoadIdentity();
    glTranslatef( 0.5f, 0.5f, 0.5f );     // Ofsetas/nukrypimas
    glScalef( 0.5f, 0.5f, 0.5f );         // Paklaida

    gluPerspective(95.0f, 1.5f, 0.5f, 100.0f);// Švieoso žiūrėjimo
                              // piramidė/perspektyva, nustatome tokią pat kaip ir
                              // gylio tekstūro kūrime(SukurtiGylioTekstura()).
    glMultMatrixf( lightsLookAtMatrix );  // Šviesos matrica.
```

Surišame tekstūravimą su gylio tekstūrą, kad jį galėtumę panaudot kaip šešėlių planus:

```
glEnable( GL_TEXTURE_2D ); // Įjungiam tekstūras.
glGenTextures( 1,&depthTexture );
glBindTexture( GL_TEXTURE_2D, depthTexture ); // Sūrišam tekstūras su
                                              /// gylio tekstūrom.

if( wglBindTexImageARB(PBuferis.hPBuferis, WGL_DEPTH_COMPONENT_NV ) == FALSE )
{
        MessageBox(NULL,"Negalėjo surišti su p-buferiu tekstūrų generavimui!",
              "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
}

GeneruotiScena(); // Generuojam savo sceną, bet jau dabar su šešėliais.
```

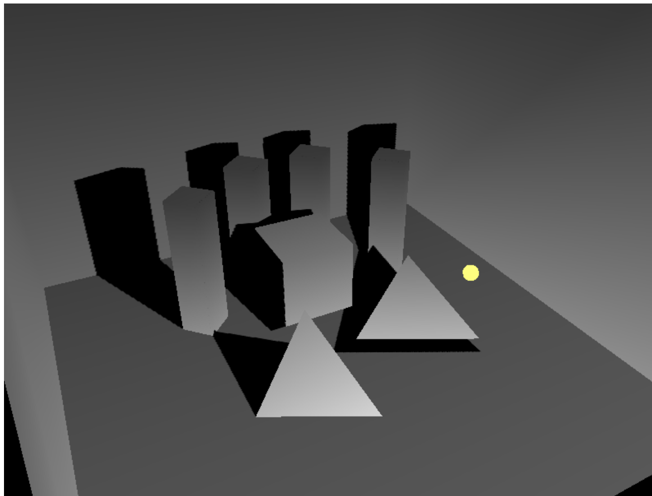
Sugeneravus gautinę sceną atstatome pradines būsenas:

```
if(wglReleaseTexImageARB(PBuferis.hPBuffer, WGL_DEPTH_COMPONENT_NV )== FALSE)
{
        MessageBox(NULL,"Negalėjo paleist p-buferio nuo tekstūrų generavimo!",
              "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
}

glDisable( GL_TEXTURE_2D );
glDisable( GL_TEXTURE_GEN_S );
glDisable( GL_TEXTURE_GEN_T );
glDisable( GL_TEXTURE_GEN_R );
glDisable( GL_TEXTURE_GEN_Q );
```

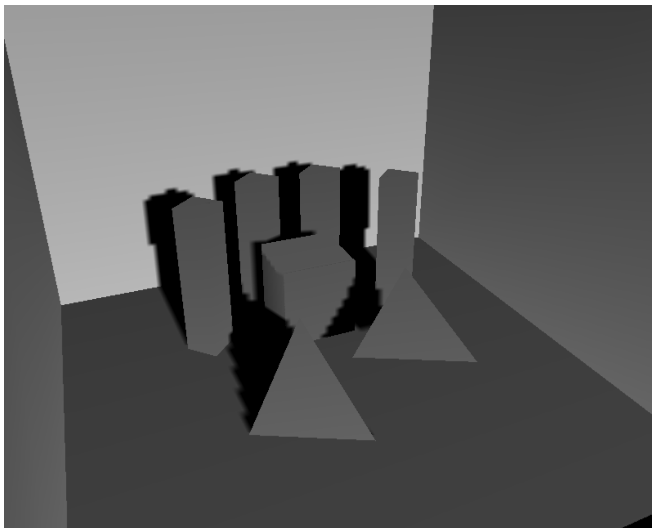
2.2.2. Testavimas ir išvados

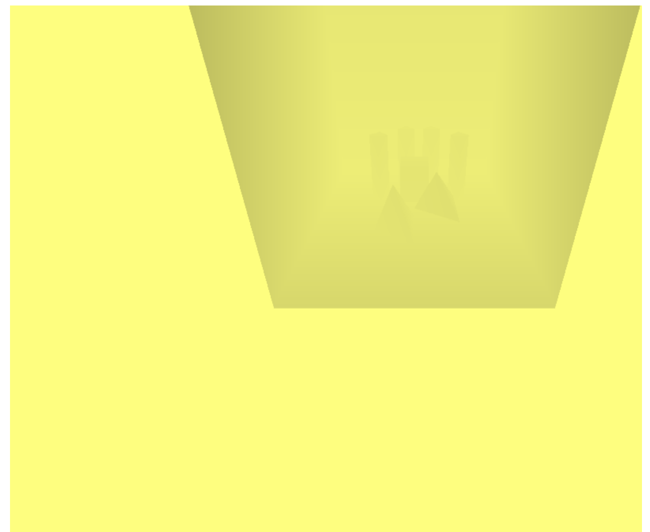
Šešėlių planai, kai šviesos šaltinis tinkamoj padėty (nei per arti, nei per toli šešėlį metančio objekto) mūsų scenoje duoda tikslius, kokybiškus šešėlius esant 1024x1024 šešėlių tekstūros raiškai (2.2.3. pav.). Tačiau šviesos šaltiniui atsidūrus toliau pradeda matytis artefaktai (dantuotumas) ir šešėlių kontūrai palieka nebetiesūs ir iškraipyti, esant tai pačiai 1024x1024 šešėlių tekstūros raiškai (2.2.4. pav.). Aišku tai galima pataisyti padidinus šešėlio tekstūrų raišką, bet kuo didesnę raišką nustatysime, tuo šešėlių planų metodas dirbs lėčiau.

Kita šešėlių planų problema – tai, kad šviesos šaltiniui atsidūrus per arti šešėlius metančių objektų arba parinkus netinkamas taško koordinates, į kurį šviesa nukreipta žiūrėti, mūsų scena nebetelpa į šviesos regos piramidę ir to pasakoje netilpę objektai meta neteisingus šešėlius (2.2.5. pav.).

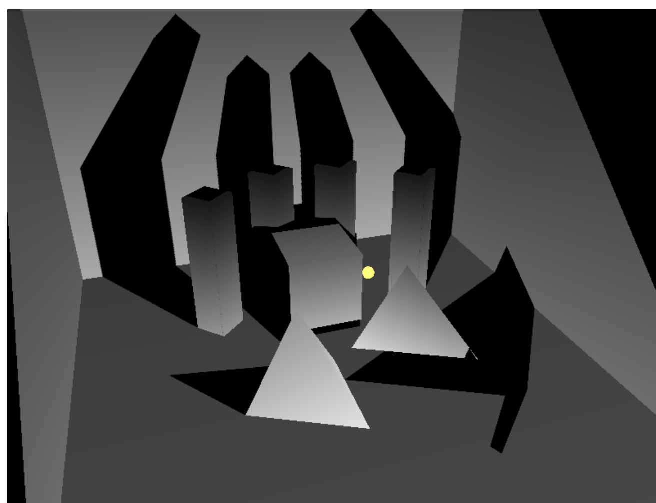


2.2.3. pav. Kai šviesos šaltinis tinkamoj padėty, šešėlių planai mūsų scenoje duoda tikslius, kokybiškus šešėlius esant 1024x1024 šešėlių tekstūros raiškai. Dešinėje pusėje pavaizduota gylio tekstūra





2.2.4. pav. Kai šviesos šaltinis yra toliau pradeda matytis artefaktai (dantuotumas) ir šešėlių kontūrai palieka nebetiesūs ir iškraipyti, esant tai pačiai 1024x1024 šešėlių tekstūros raiškai. Dešinėje pusėje pavaizduota gylio tekstūra



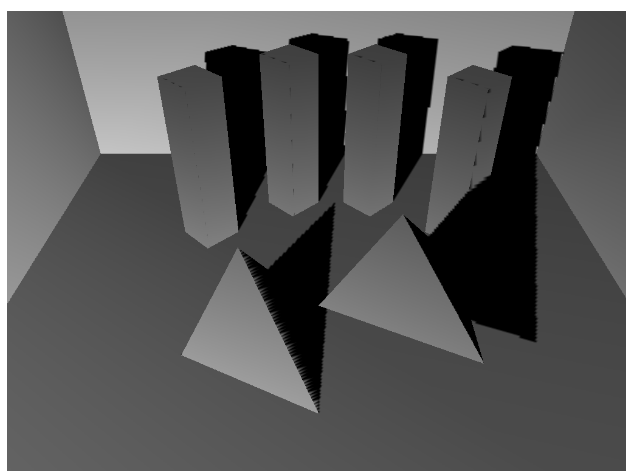
2.2.5. pav. Kai šviesos šaltinis yra per arti šešėlius metančių objektų arba parinkus netinkamas taško koordinates, į kurį šviesa nukreipta žiūrėti, mūsų scena nebetelpa į šviesos regos piramidę ir to pasakoje netilpę objektai meta neteisingus šešėlius. Dešinėje pusėje pavaizduota gylio tekstūra

2.2.1. lentelė Šešėlių planų greičio priklausomybė nuo šešėlių raiškos (1024x768 rez., 32 bitų spalvos)

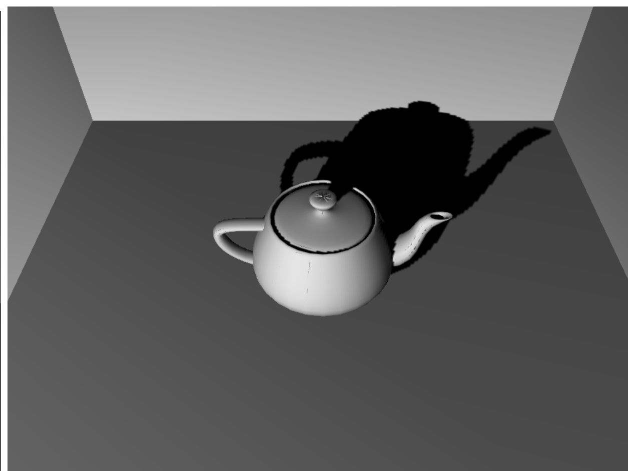
Šešėlių tekstūrų raiška	Kadrai per sekundę (vidurkis)	
	Scena 1	Scena 2
128x128	1512 kadr./s	294 kadr./s
256x256	1514 kadr./s	293 kadr./s
512x512	1482 kadr./s	289 kadr./s
1024x1024	1329 kadr./s	286 kadr./s
2048x2048	1172 kadr./s	280 kadr./s
4096x4096	484 kadr./s	261 kadr./s
8192x8192	126 kadr./s	127 kadr./s

Mūsų testavimo sistema: GeForce GTX260 vaizdo plokštė, AMD Athlon 64 X2 5000+ procesorius, Asus M2N-X pagrindinė plokštė, 2GB DDR2 800MHz operatyvinė atmintis.

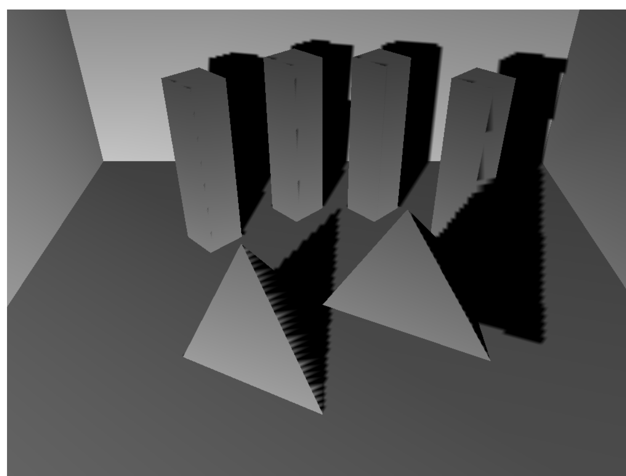
Kaip iš 2.2.1. lentelės matome, kad su šiuolaikiniais vaizdo spartintuvais šešėlių planų greitis palyginti nedaug priklauso nuo šešėlių tekstūrų raiškos. Pavyzdžiui pagal mūsų atliktus testus padidinus raišką 8 kartus, kadrų generavimas sulėtėja tik kažkur 7-9 procentus, o mažesnės kaip 512x512 raiškos praktiškai neapsimoka imti, nes greičio praktiškai nepadidinama, o kokybė šešėlių būna palyginti labai prasta (2.2.6. pav.). Nors esant raiškai 4096x4096 pirmoje scenoje kadrų skaičius nukrito kelis kartus, o antroje scenoje stiprus sulėtėjimas pasijautė esant 8192x8192 raiškai.



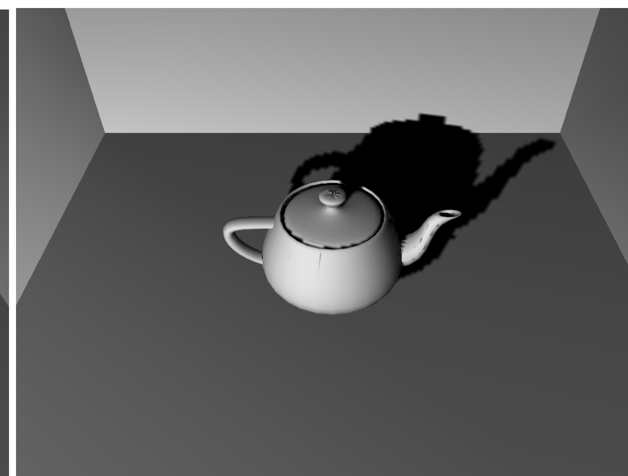
(a) Scena 1, 1024x1024



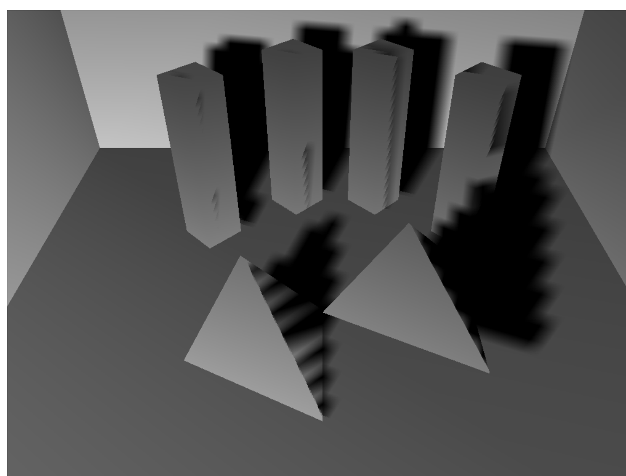
(b) Scena 2, 1024x1024



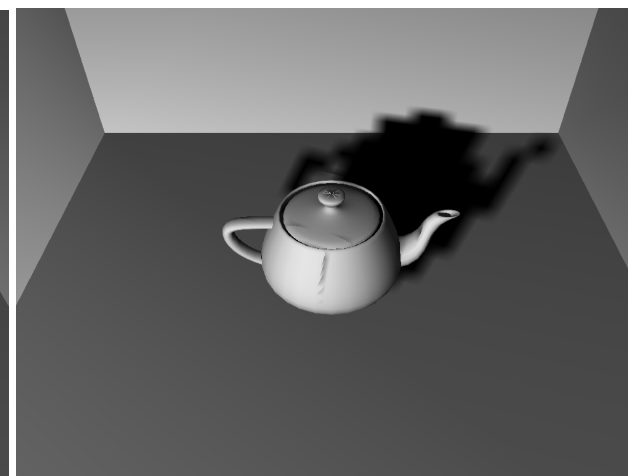
(c) Scena 1, 512x512



(d) Scena 2, 512x512



(e) Scena 1, 128x128



(f) Scena 2, 128x128

2.2.6. pav. Šešėlių kokybės priklausomybė nuo šešėlių tekstūrų raiškos (raiškos dydis nurodytas po paveikslėliais)

2.3. Metodų palyginimas

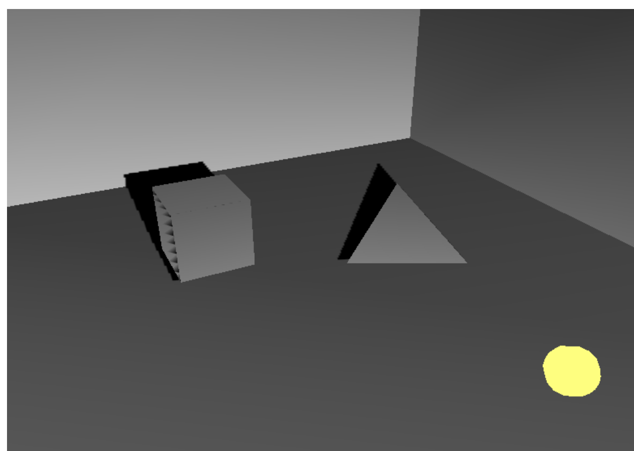
Pagal mūsų testavimo duomenis šešėlių tūriai žymiai duoda kokybiškesnius šešėlius įvairiuose šviesos šaltini padėtyse. Pavyzdžiui šviesos šaltiniu esant prie pat šešėliuotojų (2.3.2. (c) pav.) šešėlių tūriai duoda tikslius šešėlius, tuo tarpu šešėlių planų scenoje tetraedras nemeta šešėlių teisingai, nes kaip iš 2.3.2. (b) paveikslo matosi, tetraedras pilnai netelpa į šviesos šaltinio regos lauką. Šviesos šaltiniui atsidūrus toli (tarkim apie100 kartų toliau nuo scenos centro nei šaltinis 2.3.2. pav.) pradeda matytis šešėlių planų scenoje pradeda matytis artefaktai (dantuotumas) esant ir palyginti aukštai 4096x4096 šešėlių raiškai (2.3.3. (a) pav.). Tuo tarpu šešėlių tūriai, esant šaltiniui taip pat toli, duoda tokius pat tikslius šešėlius kaip ir šaltiniui esant arti (2.3.3. (b) pav.).

Tačiau šešėlių tūrių greičiui poligonų skaičius daro žymiai didesnę įtaką, nei šešėlių planų, t. y. sudėtingesnėse scenose šešėlių tūriai sulėtėja daugiau nei šešėlių planai (2.3.1. lent.). Pavyzdžiui 2.3.1. pav. trečioje scenoje šešėlių planai (2.3.1. (e)), esant ir 1024x0124 šešėlių raiškai, atrodo beveik taip pat gerai kaip ir šešėlių tūriai (2.3.1. (f), bet scena generuojama beveik tris kartus greičiau.

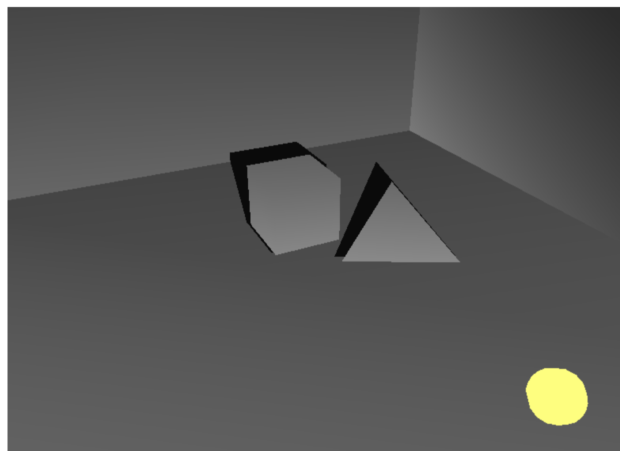
2.3.1. lentelė Šešėlių planų ir šešėlių tūrių greičių palyginimas (1024x768 rez., 32 bitų spalvos). Nurodytas vidutinis kadrų kiekis per sekundę (2.3.1. pav.)

	Šešėlių tūriai	Šešėlių planai (1024x1024)	Šešėlių planai (2048x2048)	Šešėlių planai (4096x4096)
1 Scena (14 poligonų)	1732 kadr./s	1723 kadr./s	1541 kadr./s	510 kadr./s
2 Scena (32 poligonai)	865 kadr./s	1592 kadr./s	1489 kadr./s	483 kadr./s
3 Scena (69 poligonai)	554 kadr./s	1520 kadr./s	1410 kadr./s	441 kadr./s

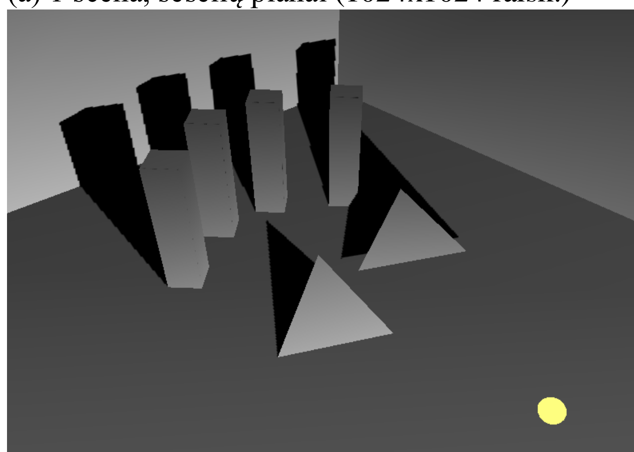
Mūsų testavimo sistema: GeForce GTX260 vaizdo plokštė, AMD Athlon 64 X2 5000+ procesorius, Asus M2N-X pagrindinė plokštė, 2GB DDR2 800MHz operatyvinė atmintis.



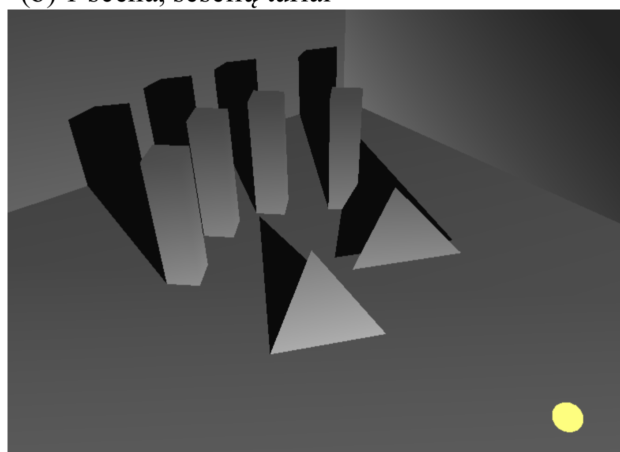
(a) 1 scena, šešėlių planai (1024x1024 raišk.)



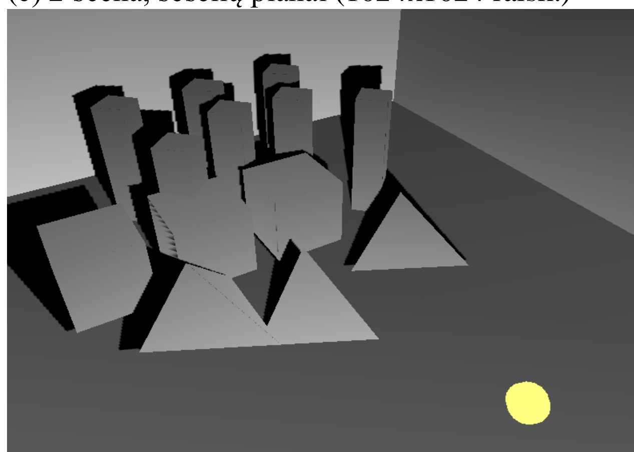
(b) 1 scena, šešėlių tūriai



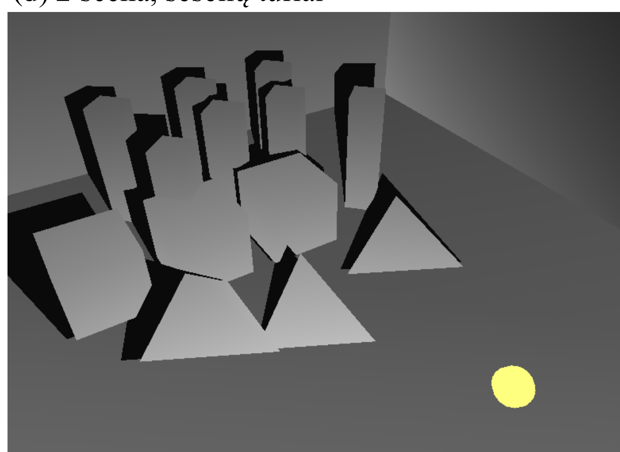
(c) 2 scena, šešėlių planai (1024x1024 raišk.)



(d) 2 scena, šešėlių tūriai

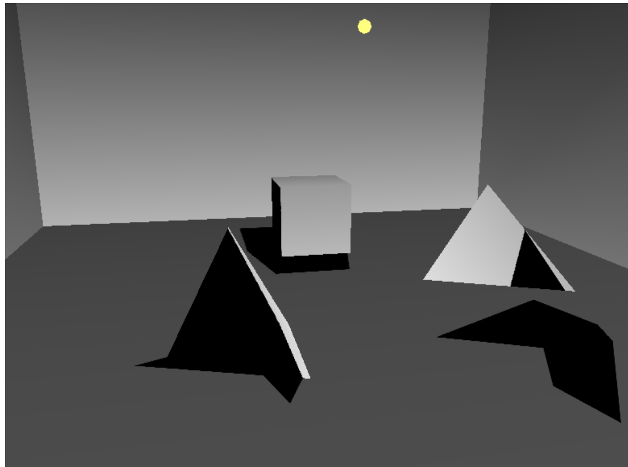


(e) 3 scena, šešėlių planai (1024x1024 raišk.)



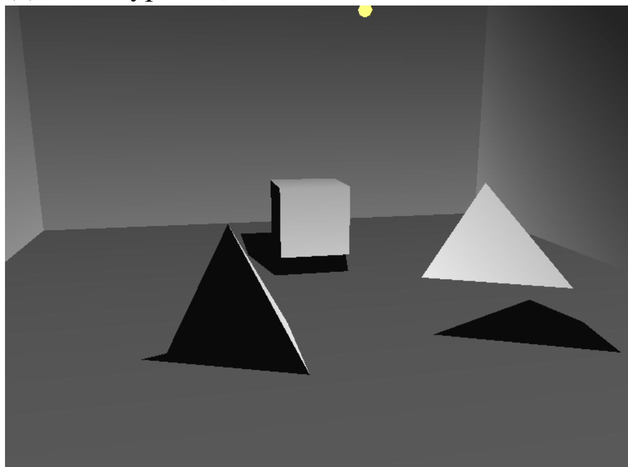
(f) 3 scena, šešėlių tūriai

2.3.1. pav. Šešėlių planų ir šešėlių tūrių palyginimas

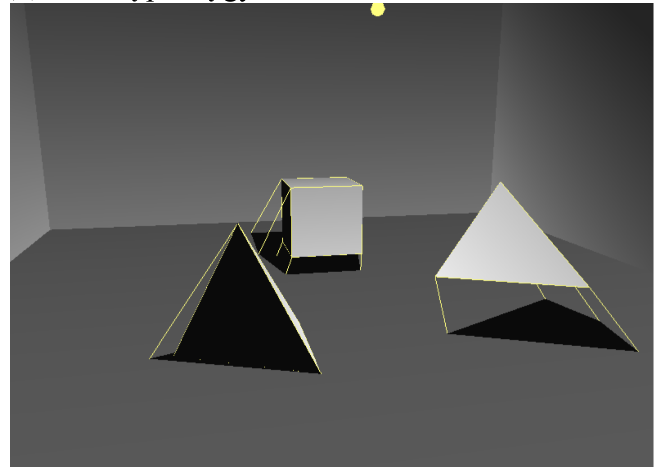


(a) Šešėlių planai, kai šviesos šaltinis arti

(b) Šešėlių planų gylio tekstūra

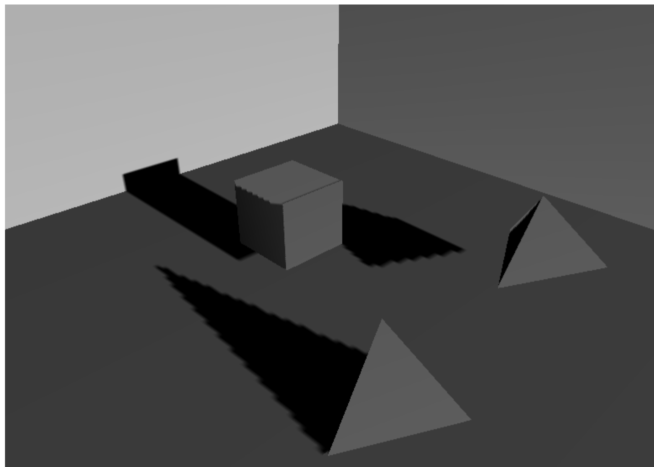


(c) Šešėlių tūriai, kai šviesos šaltinis arti

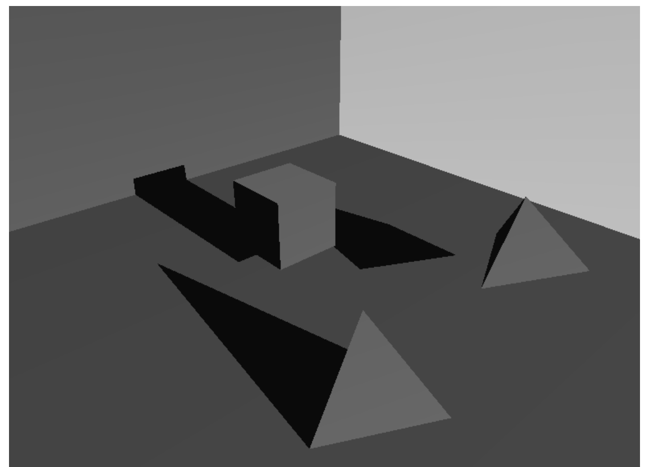


(d) Pažymėti šešėlių tūriai

2.3.2. pav. Šešėlių tūrių ir šešėlių planų palyginimas, kai šviesos šaltinis arti



(a) Šešėlių planai 4096x4096



(b) Šešėlių tūriai

2.3.3. pav. Šešėlių tūrių ir šešėlių planų palyginimas, kai šviesos šaltinis toli

IŠVADOS

1. Darbe pristatytos bei praktiškai realizuotos dvi iš populiariausių realaus laiko šešėlių vaizdavimo technikos: tūriniai šešėliai bei šešėlių planai.
2. Atlikti kokybiniai bei kiekybiniai minėtų šešėlių technikų testavimai bei jų tarpusavio palyginimai.
3. Šešėlių tūriai duoda kokybiškus, visiškai tiesiais kontūrais šešėlius, net ir palyginti šešėliams sunkiose užduotyse (šviesos šaltiniui esant arti, toli šešėlį metančio objekto ar net jo viduj).
4. Šešėlių planai tik tam tikrais atvejais duoda kokybiškus šešėlius. Kai šviesos šaltinis per arti, t. y. visa scena netelpa į šviesos regos lauką ir nepatekusioje dalyje šešėlių visa nėra, arba jie neteisingai ištampomi. Be to šešėlių planų kraštai kenčia nuo *aliasing*'o (dantuotumo).
5. Nors šešėlių tūriai duoda tikslesnius šešėlius nei šešėlių planai, bet poligonų skaičius scenoje jiems turi žymiai didesnę greičio įtaką, nei šešėlių planų greičiui. Todėl sudėtingose scenose poligonų atžvilgiu šešėlių planų technika gali dirbti daug kartų greičiau nei šešėlių tūriai.

LITERATŪRA

[1] Andrew V. Nealen. Shadow Mapping and Shadow Volumes: Recent Developments in Real-Time Shadow Rendering. University of British Columbia. 2002 m.

[2] Frank Klawonn. Introduction to Computer Graphics Using Java 2D and 3D. Springer-Verlag London Limited. 2008 m.

[3] Eric Lengyel. Mathematics for 3D Game programing and Compiuter Graphics (Second Edition). 2004 m.

[4] Samuel R. Buss. 3-D Computer Graphics - A Mathematical Introduction With OpenGL. 2003 m.

[5] Kelly Dempski ir Emmanuel Viale. Advanced Lighting And Materials With Shaders. 2005 m.

[6] Sebastien St-Laurent. Shaders for Game Programming and Artists. 2004 m.

[7] Michael D. McCool. Shadow Volume Reconstruction from Depth Maps. University of Waterloo. 2000 m.

[8] Pradeep Sen, Mike Cammarano, Pat Hanrahan. Shadow Silhouette Maps. Stanford University. 2003 m.

[9] Eric Chan Fr'edo Durand. An Efficient Hybrid Shadow Rendering Algorithm. Massachusetts Institute of Technology.

[10] Morgan Kaufmann. Advanced Graphics Programming Using OpenGL (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling). 2005 m.

[11] Hun Yen Kwoon. The Theory of Stencil Shadow Volumes. 2002 m.

[12] Josh Beam. Tutorial - Stenciled Shadow Volumes in OpenGL. 2008 m.

[13] Eric Lengyel. Mathematics for 3D Game Programming & Computer Graphics. Charles River Media. 2002 m.

[14] Tomas Moller, Eric Haines. Realtime Rendering, 2nd Edition. A K Peters Ltd. 2002 m.

[15] Wolfgang F. Engel, Amir Geva and Andre LaMothe. Beginning Direct3D Game Programming. Prima Publishing. 2001 m.

[16] Wolfgang F. Engel. Direct3D ShaderX Vertex and Pixel Shader Tips and Tricks. Wordware Publishing Inc. 2002 m.

Shadow volumes and shadow mapping realization and analysis

SUMMARY

In this work we introduce to two alternative, most popular real time shadow rendering techniques. This is shadow map (or mapping) and shadow volumes techniques. Because those two methods have deferent advantages and weaknesses, there are always in development, and sometimes hybrid algorithms are created, which would absorb more good features from them. So in this work we tried to determine and rate in which cases one is better than another.

TERMINŲ IR SANTRUMPŲ ŽODYNAS

Pikselis (*pixel*)	mažiausia informacijos dalis paveiksle
Šešėliuotojas (*shader*)	kompiuterinėje grafikoje yra programinės įrangos instrukcijų rinkinys, visų pirma skirtas vaizdavimo efektams apskaičiuoti vaizdo kortoje našumui ir kokybe pagerinti.
Fragmentas (*fragment*)	duomenys reikalingi sugeneruoti vieną pikselį paveiksle kadrų bufery.
Kadrų buferis (*frame buffer*)	video išėjimo prietaisas, kuris išveda vaizdą iš atminties buferio, laikančio pilno kadro informaciją, į monitorių.
Aliasing'as	skaitmeninio paveikslo atstatyme, jei rezoliucija maža, tai atstatytas paveikslas skirsis nuo originalaus, t. y. matysis taip vadinamas *aliasing*'as („kvadratėliai“).
Gylio buferis (*depth buffer* arba *z-buffer*)	buferis, kuriame laikoma sugeneruoto pikselio gylis (z koordinatė).
RGBA (Red Green Blue Alpha)	Tai paprastas raudonos, žalios ir mėlynos (RGB) spalvų modelis su papildoma informacija.
OpenGL (Open Graphics Library)	Kompiuterinės grafikos biblioteka.
Poligonas	Plokščia uždara figūra, gauta apjungiant taškus tiesiom linijom.
Vekselis (*texel*)	Tekstūros elementas ar tekstūros pikselis.

1 PRIEDAS.

Šešėlių tūrius realizuojančios programos kodas

Failas **shadow_volume.cpp**:

```
//-----------------------------------------------------------------------------
//  Valdymo mygtukai: Up    - sviesos saltinis juda i prieki
//                    Down  - sviesos saltinis juda atgal
//                    Left  - sviesos saltinis juda i kaire
//                    Right - sviesos saltinis juda i desine
//                    Enter - sviesos saltinis juda i virsu
//                    Backspace - sviesos saltinis juda zemyn
//                    Left Mouse Button  - sukineti kamera
//                    F1 - rodyti turiu konturus
//                    F2 - priartinti kamera prie sukinejimo centro
//                              F3 - atitolinti kamera nuo sukinejimo centro
//
//  Poligonai ivedami is failo "duom.txt"
//-----------------------------------------------------------------------------

#define STRICT
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include "geometry.h"
#include "resource.h"
#include <math.h>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <omp.h>
#include <string>
const double PI = 3.141592;
using namespace std;
ifstream f("duom.txt");
//-----------------------------------------------------------------------------
// GLOBALS
//-----------------------------------------------------------------------------
HWND   g_hWnd = NULL;
HDC    g_hDC  = NULL;
HGLRC  g_hRC  = NULL;
DEVMODE g_oldDevMode;

// Vaizdo sukinejimui
float  g_fSpinX_L =   0.0f;
float  g_fSpinY_L = -10.0f;
float ziurejimo_z = -18.0;

bool  RenderShadowVolume = false;
float AmountOfExtrusion  = 1.0f;
float lightPosition[] = { 0.0f, 3.0f, 0.0f, 1.0f }; // sviesos pozicija pradine

struct Vertex
{
      float x, y, z;
};
```

```
Vertex V0[] = {{ 0, 0, 0 },{0, 0, 0 },{ 0, 0, 0 },{ 0, 0, 0 }};
int psk = 0;

float shadowCasterNormal[] = { 0.0f, 1.0f, 0.0f};

struct ShadowCaster0 //seselio metejas
{
    Vertex *verts;        // virsunes
    float  *normal;       // pavirsiaus apsvietimo reiksmes
    int     numVerts;     // virsuniu skaicius
    int     shadowVolume; // seselio turio saraso numeris
};

ShadowCaster0 shadowCaster[100];

//-----------------------------------------------------------------------------
// PROTOTYPES
//-----------------------------------------------------------------------------
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow);
LRESULT CALLBACK WindowProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
void init(void);
void render(void);
void shutDown(void);
void buildShadowVolume(ShadowCaster0 caster[100], float lightPosit[3], float ext,
int h);

//-----------------------------------------------------------------------------
//Sasajos lango paruosimas
//-----------------------------------------------------------------------------
int WINAPI WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR     lpCmdLine,
                   int       nCmdShow )
{
    WNDCLASSEX winClass;
    MSG        uMsg;

    memset(&uMsg,0,sizeof(uMsg));

    winClass.lpszClassName = "MY_WINDOWS_CLASS";
    winClass.cbSize        = sizeof(WNDCLASSEX);
    winClass.style         = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
    winClass.lpfnWndProc   = WindowProc;
    winClass.hInstance     = hInstance;
    winClass.hIcon         = LoadIcon(hInstance, (LPCTSTR)IDI_OPENGL_ICON);
    winClass.hIconSm       = LoadIcon(hInstance, (LPCTSTR)IDI_OPENGL_ICON);
    winClass.hCursor       = LoadCursor(NULL, IDC_ARROW);
    winClass.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
    winClass.lpszMenuName  = NULL;
    winClass.cbClsExtra    = 0;
    winClass.cbWndExtra    = 0;

    if( !RegisterClassEx(&winClass) )
        return E_FAIL;

    g_hWnd = CreateWindowEx( NULL, "MY_WINDOWS_CLASS",
        "OpenGL - Shadow Volume Using The Z-Fail Technique",
```

```
        WS_OVERLAPPEDWINDOW | WS_VISIBLE,
        0, 0, 800, 600, NULL, NULL, hInstance, NULL );
    if( g_hWnd == NULL )
        return E_FAIL;

    ShowWindow( g_hWnd, nCmdShow );
    UpdateWindow( g_hWnd );

    init();

    while( uMsg.message != WM_QUIT )
    {
        if( PeekMessage( &uMsg, NULL, 0, 0, PM_REMOVE ) )
        {
            TranslateMessage( &uMsg );
            DispatchMessage( &uMsg );
        }
        else
            {
            render();
            }
    }

    shutDown();

    UnregisterClass( "MY_WINDOWS_CLASS", winClass.hInstance );

    return uMsg.wParam;
}

//-----------------------------------------------------------------------------
// Valdymo nustatymas
//-----------------------------------------------------------------------------
LRESULT CALLBACK WindowProc( HWND   hWnd,
                             UINT   msg,
                             WPARAM wParam,
                             LPARAM lParam )
{
    static POINT ptLastMousePosit_L;
    static POINT ptCurrentMousePosit_L;
    static bool  bMousing_L;

    static POINT ptLastMousePosit_R;
    static POINT ptCurrentMousePosit_R;
    static bool  bMousing_R;

    switch( msg )
    {
    case WM_KEYDOWN:
        {
            switch( wParam )
            {
            case VK_ESCAPE:
                PostQuitMessage(0);
                break;

            case VK_F1:
                RenderShadowVolume = !RenderShadowVolume;
                break;
```

```
        case VK_F2:
            ziurejimo_z += 0.2f;
            break;

        case VK_F3:
            ziurejimo_z -= 0.2f;
            break;

        case 38: // Up Arrow Key
            lightPosition[2] -= 0.1f;
            break;

        case 40: // Down Arrow Key
            lightPosition[2] += 0.1f;
            break;

        case 37: // Left Arrow Key
            lightPosition[0] -= 0.1f;
            break;

        case 39: // Right Arrow Key
            lightPosition[0] += 0.1f;
            break;

              case VK_BACK: //Backspace
            lightPosition[1] -= 0.1f;
            break;

        case VK_RETURN: // Enter
            lightPosition[1] += 0.1f;
            break;
        }
    }
    break;

case WM_LBUTTONDOWN:
    {
        ptLastMousePosit_L.x = ptCurrentMousePosit_L.x = LOWORD (lParam);
        ptLastMousePosit_L.y = ptCurrentMousePosit_L.y = HIWORD (lParam);
        bMousing_L = true;
    }
    break;

case WM_LBUTTONUP:
    {
        bMousing_L = false;
    }
    break;

case WM_RBUTTONDOWN:
    {
        ptLastMousePosit_R.x = ptCurrentMousePosit_R.x = LOWORD (lParam);
        ptLastMousePosit_R.y = ptCurrentMousePosit_R.y = HIWORD (lParam);
        bMousing_R = true;
    }
    break;

case WM_RBUTTONUP:
```

```
        {
            bMousing_R = false;
        }
        break;

    case WM_MOUSEMOVE:
        {
            ptCurrentMousePosit_L.x = LOWORD (lParam);
            ptCurrentMousePosit_L.y = HIWORD (lParam);
            ptCurrentMousePosit_R.x = LOWORD (lParam);
            ptCurrentMousePosit_R.y = HIWORD (lParam);

            if( bMousing_L )
            {
                g_fSpinX_L -= (ptCurrentMousePosit_L.x - ptLastMousePosit_L.x);
                g_fSpinY_L -= (ptCurrentMousePosit_L.y - ptLastMousePosit_L.y);
            }

            ptLastMousePosit_L.x = ptCurrentMousePosit_L.x;
            ptLastMousePosit_L.y = ptCurrentMousePosit_L.y;
            ptLastMousePosit_R.x = ptCurrentMousePosit_R.x;
            ptLastMousePosit_R.y = ptCurrentMousePosit_R.y;
        }
        break;

    case WM_SIZE:
        {
            int nWidth  = LOWORD(lParam);
            int nHeight = HIWORD(lParam);
            glViewport(0, 0, nWidth, nHeight);

            glMatrixMode( GL_PROJECTION );
            glLoadIdentity();
            gluPerspective( 45.0, (GLdouble)nWidth / (GLdouble)nHeight, 0.1, 100.0);
        }
        break;

    case WM_CLOSE:
        {
            PostQuitMessage(0);
        }
        break;

    case WM_DESTROY:
        {
            PostQuitMessage(0);
        }
        break;

    default:
        {
            return DefWindowProc( hWnd, msg, wParam, lParam );
        }
        break;
    }

    return 0;
}
```

```
//---------------------------------------------------------------------------------
// Apibrezia sesekiu turius
//---------------------------------------------------------------------------------
void buildShadowVolume( ShadowCaster0 caster[100], float lightPosit[3], float ext,
int h )
{
float x1, y1, z1;
float x2, y2, z2;
float x0, y0, z0;
float xs, ys, zs;

xs = lightPosition[0]-shadowCaster[h].verts[0].x;
ys = lightPosition[1]-shadowCaster[h].verts[0].y;
zs = lightPosition[2]-shadowCaster[h].verts[0].z;

x1=shadowCaster[h].verts[1].x-shadowCaster[h].verts[0].x;
y1=shadowCaster[h].verts[1].y-shadowCaster[h].verts[0].y;
z1=shadowCaster[h].verts[1].z-shadowCaster[h].verts[0].z;

x2=shadowCaster[h].verts[2].x-shadowCaster[h].verts[0].x;
y2=shadowCaster[h].verts[2].y-shadowCaster[h].verts[0].y;
z2=shadowCaster[h].verts[2].z-shadowCaster[h].verts[0].z;

x0=0;
y0=0;
z0=0;

double ryx1 = sqrt(x1*x1+y1*y1);
double ryx2 = sqrt(x2*x2+y2*y2);
double ryxs = sqrt(xs*xs+ys*ys);
double dal1, dal2, dals;
double beta1 =0;
double beta2 =0;
double betas =0;

if(y1>0 && x1>0){ dal1 = y1/x1; beta1 = atanf(dal1);}
if(y1<0 && x1>0){ dal1 = x1/y1*(-1); beta1 = atanf(dal1)+1.5*PI;}
if(y1<0 && x1<0){ dal1 = y1/x1;  beta1 = atanf(dal1)+PI;}
if(y1>0 && x1<0){ dal1 = x1/y1*(-1); beta1 = atanf(dal1)+PI/2;}
if(y1==0 && x1>0){ beta1=0;}
if(y1>0 && x1==0){ beta1=PI/2;}
if(y1==0 && x1<0){ beta1=PI;}
if(y1<0 && x1==0){ beta1=1.5*PI;}

if(y2>0 && x2>0){ dal2 = y2/x2; beta2 = atanf(dal2);}
if(y2<0 && x2>0){ dal2 = x2/y2*(-1); beta2 = atanf(dal2)+1.5*PI;}
if(y2<0 && x2<0){ dal2 = y2/x2;  beta2 = atanf(dal2)+PI;}
if(y2>0 && x2<0){ dal2 = x2/y2*(-1); beta2 = atanf(dal2)+PI/2;}
if(y2==0 && x2>0){ beta2=0;}
if(y2>0 && x2==0){ beta2=PI/2;}
if(y2==0 && x2<0){ beta2=PI;}
if(y2<0 && x2==0){ beta2=1.5*PI;}

if(ys>0 && xs>0){ dals = ys/xs; betas = atanf(dals);}
if(ys<0 && xs>0){ dals = xs/ys*(-1); betas = atanf(dals)+1.5*PI;}
if(ys<0 && xs<0){ dals = ys/xs;  betas = atanf(dals)+PI;}
if(ys>0 && xs<0){ dals = xs/ys*(-1); betas = atanf(dals)+PI/2;}
if(ys==0 && xs>0){ betas=0;}
```

```
if(ys>0 && xs==0){ betas=PI/2;}
if(ys==0 && xs<0){ betas=PI;}
if(ys<0 && xs==0){ betas=1.5*PI;}

      x1=ryx1;
      y1=0;
      x2=ryx2*cosf(beta2-beta1);
      y2=ryx2*sinf(beta2-beta1);

      xs=ryxs*cosf(betas-beta1);
      ys=ryxs*sinf(betas-beta1);

double rzx1 = sqrt(x1*x1+z1*z1);
double rzx2 = sqrt(x2*x2+z2*z2);
double rzxs = sqrt(xs*xs+zs*zs);

double zdal1, zdal2, zdals;
double zbeta1 =0;
double zbeta2 =0;
double zbetas =0;

if(z1>0 && x1>0){ zdal1 = z1/x1; zbeta1 = atanf(zdal1);}
if(z1<0 && x1>0){ zdal1 = x1/z1*(-1); zbeta1 = atanf(zdal1)+1.5*PI;}
if(z1<0 && x1<0){ zdal1 = z1/x1;  zbeta1 = atanf(zdal1)+PI;}
if(z1>0 && x1<0){ zdal1 = x1/z1*(-1); zbeta1 = atanf(zdal1)+PI/2;}
if(z1==0 && x1>0){ zbeta1=0;}
if(z1>0 && x1==0){ zbeta1=PI/2;}
if(z1==0 && x1<0){ zbeta1=PI;}
if(z1<0 && x1==0){ zbeta1=1.5*PI;}


if(z2>0 && x2>0){ zdal2 = z2/x2; zbeta2 = atanf(zdal2);}
if(z2<0 && x2>0){ zdal2 = x2/z2*(-1); zbeta2 = atanf(zdal2)+1.5*PI;}
if(z2<0 && x2<0){ zdal2 = z2/x2;  zbeta2 = atanf(zdal2)+PI;}
if(z2>0 && x2<0){ zdal2 = x2/z2*(-1); zbeta2 = atanf(zdal2)+PI/2;}
if(z2==0 && x2>0){ zbeta2=0;}
if(z2>0 && x2==0){ zbeta2=PI/2;}
if(z2==0 && x2<0){ zbeta2=PI;}
if(z2<0 && x2==0){ zbeta2=1.5*PI;}

if(zs>0 && xs>0){ zdals = zs/xs; zbetas = atanf(zdals);}
if(zs<0 && xs>0){ zdals = xs/zs*(-1); zbetas = atanf(zdals)+1.5*PI;}
if(zs<0 && xs<0){ zdals = zs/xs;  zbetas = atanf(zdals)+PI;}
if(zs>0 && xs<0){ zdals = xs/zs*(-1); zbetas = atanf(zdals)+PI/2;}
if(zs==0 && xs>0){ zbetas=0;}
if(zs>0 && xs==0){ zbetas=PI/2;}
if(zs==0 && xs<0){ zbetas=PI;}
if(zs<0 && xs==0){ zbetas=1.5*PI;}

z1=0;
x1=rzx1;

x2=rzx2*cosf(zbeta2-zbeta1);
z2=rzx2*sinf(zbeta2-zbeta1);

xs=rzxs*cosf(zbetas-zbeta1);
zs=rzxs*sinf(zbetas-zbeta1);

double rzy2 = sqrt(y2*y2+z2*z2);
```

```
double rzys = sqrt(ys*ys+zs*zs);
double zydal2;
double zydals;
double ybeta2 = 0;
double ybetas = 0;

if(z2>0 && y2>0){ zydal2 = z2/y2; ybeta2 = atanf(zydal2);}
if(z2>0 && y2<0){ zydal2 = y2/z2*(-1); ybeta2 = atanf(zydal2)+PI/2;}
if(z2<0 && y2<0){ zydal2 = z2/y2;  ybeta2 = atanf(zydal2)+PI;}
if(z2<0 && y2>0){ zydal2 = y2/z2*(-1); ybeta2 = atanf(zydal2)+1.5*PI;}
if(z2==0 && y2>0){ ybeta2=0;}
if(z2>0 && y2==0){ ybeta2=PI/2;}
if(z2==0 && y2<0){ ybeta2=PI;}
if(z2<0 && y2==0){ ybeta2=1.5*PI;}

if(zs>0 && ys>0){ zydals = zs/ys; ybetas = atanf(zydals);}
if(zs>0 && ys<0){ zydals = ys/zs*(-1); ybetas = atanf(zydals)+PI/2;}
if(zs<0 && ys<0){ zydals = zs/ys;  ybetas = atanf(zydals)+PI;}
if(zs<0 && ys>0){ zydals = ys/zs*(-1); ybetas = atanf(zydals)+1.5*PI;}
if(zs==0 && ys>0){ ybetas=0;}
if(zs>0 && ys==0){ ybetas=PI/2;}
if(zs==0 && ys<0){ ybetas=PI;}
if(zs<0 && ys==0){ ybetas=1.5*PI;}

y2=rzy2;
z2=0;

ys=rzys*cosf(ybetas-ybeta2);
zs=rzys*sinf(ybetas-ybeta2);

//****************************************************************

if( shadowCaster[h].shadowVolume != -1 )
        glDeleteLists( shadowCaster[h].shadowVolume, 0 );


    shadowCaster[h].shadowVolume = glGenLists(1);

// jei poligonas is 3 tasku
if(shadowCaster[h].numVerts==3)
{
            double xx0= shadowCaster[h].verts[0].x;
                  double yy0= shadowCaster[h].verts[0].y;
                  double zz0= shadowCaster[h].verts[0].z;

                  double xxs0= shadowCaster[h].verts[0].x-lightPosit[0];
                  double yys0= shadowCaster[h].verts[0].y-lightPosit[1];
                  double zzs0= shadowCaster[h].verts[0].z-lightPosit[2];

                  double xx1= shadowCaster[h].verts[1].x;
                  double yy1= shadowCaster[h].verts[1].y;
                  double zz1= shadowCaster[h].verts[1].z;

                  double xxs1= shadowCaster[h].verts[1].x-lightPosit[0];
                  double yys1= shadowCaster[h].verts[1].y-lightPosit[1];
                  double zzs1= shadowCaster[h].verts[1].z-lightPosit[2];

                  double xx2= shadowCaster[h].verts[2].x;
                  double yy2= shadowCaster[h].verts[2].y;
```

```
            double zz2= shadowCaster[h].verts[2].z;

            double xxs2= shadowCaster[h].verts[2].x-lightPosit[0];
            double yys2= shadowCaster[h].verts[2].y-lightPosit[1];
            double zzs2= shadowCaster[h].verts[2].z-lightPosit[2];
if(zs<0)
{
  double tx1 = xx0; xx0 = xx2; xx2 = tx1;
  double ty1 = yy0; yy0 = yy2; yy2 = ty1;
      double tz1 = zz0; zz0 = zz2; zz2 = tz1;
      double tx2 = xxs0; xxs0 = xxs2; xxs2 = tx2;
  double ty2 = yys0; yys0 = yys2; yys2 = ty2;
      double tz2 = zzs0; zzs0 = zzs2; zzs2 = tz2;
}
      glNewList( shadowCaster[h].shadowVolume, GL_COMPILE );
      {

            glDisable( GL_LIGHTING );

            glBegin( GL_TRIANGLES );
            {
                  glVertex3f( xx0, yy0, zz0 );
                  glVertex3f( xx1, yy1, zz1 );
                  glVertex3f( xx2, yy2, zz2 );

                  glVertex4f( xxs2, yys2, zzs2, 0.0f);
                  glVertex4f( xxs1, yys1, zzs1, 0.0f);
                  glVertex4f( xxs0, yys0, zzs0, 0.0f);

            }
            glEnd();

            glBegin( GL_QUADS );
            {

                        glVertex3f( xx1, yy1, zz1 );
                        glVertex3f( xx0, yy0, zz0 );
                        glVertex4f( xxs0, yys0, zzs0, 0.0f );
                        glVertex4f( xxs1, yys1, zzs1, 0.0f );
            }
            glEnd();

            glBegin( GL_QUADS );
            {
                        glVertex3f( xx2, yy2, zz2 );
                        glVertex3f( xx1, yy1, zz1 );
                        glVertex4f( xxs1, yys1, zzs1, 0.0f );
                        glVertex4f( xxs2, yys2, zzs2, 0.0f );
            }
            glEnd();

            glBegin( GL_QUADS );
            {
                        glVertex3f( xx0, yy0, zz0 );
                        glVertex3f( xx2, yy2, zz2 );
                        glVertex4f( xxs2, yys2, zzs2, 0.0f );
                        glVertex4f( xxs0, yys0, zzs0, 0.0f );
            }
            glEnd();
```

```
                    glEnable( GL_LIGHTING );
                }
                glEndList();
}
// jei polygonas is 4 virsuniu
if(shadowCaster[h].numVerts==4)
{
        if(zs<0)
        {
          double tx1 = shadowCaster[h].verts[0].x;
                shadowCaster[h].verts[0].x = shadowCaster[h].verts[3].x;
                shadowCaster[h].verts[3].x = tx1;
                double ty1 = shadowCaster[h].verts[0].y;
                shadowCaster[h].verts[0].y = shadowCaster[h].verts[3].y;
                shadowCaster[h].verts[3].y = ty1;
          double tz1 = shadowCaster[h].verts[0].z;
                shadowCaster[h].verts[0].z = shadowCaster[h].verts[3].z;
                shadowCaster[h].verts[3].z = tz1;

                double tx2 = shadowCaster[h].verts[1].x;
                shadowCaster[h].verts[1].x = shadowCaster[h].verts[2].x;
                shadowCaster[h].verts[2].x = tx2;
                double ty2 = shadowCaster[h].verts[1].y;
                shadowCaster[h].verts[1].y = shadowCaster[h].verts[2].y;
                shadowCaster[h].verts[2].y = ty2;
          double tz2 = shadowCaster[h].verts[1].z;
                shadowCaster[h].verts[1].z = shadowCaster[h].verts[2].z;
                shadowCaster[h].verts[2].z = tz2;
        }
                double xx0= shadowCaster[h].verts[0].x;
                double yy0= shadowCaster[h].verts[0].y;
                double zz0= shadowCaster[h].verts[0].z;

                double xxs0= shadowCaster[h].verts[0].x-lightPosit[0];
                double yys0= shadowCaster[h].verts[0].y-lightPosit[1];
                double zzs0= shadowCaster[h].verts[0].z-lightPosit[2];

                double xx1= shadowCaster[h].verts[1].x;
                double yy1= shadowCaster[h].verts[1].y;
                double zz1= shadowCaster[h].verts[1].z;

                double xxs1= shadowCaster[h].verts[1].x-lightPosit[0];
                double yys1= shadowCaster[h].verts[1].y-lightPosit[1];
                double zzs1= shadowCaster[h].verts[1].z-lightPosit[2];

                double xx2= shadowCaster[h].verts[2].x;
                double yy2= shadowCaster[h].verts[2].y;
                double zz2= shadowCaster[h].verts[2].z;

                double xxs2= shadowCaster[h].verts[2].x-lightPosit[0];
                double yys2= shadowCaster[h].verts[2].y-lightPosit[1];
                double zzs2= shadowCaster[h].verts[2].z-lightPosit[2];

                double xx3= shadowCaster[h].verts[3].x;
                double yy3= shadowCaster[h].verts[3].y;
                double zz3= shadowCaster[h].verts[3].z;

                double xxs3= shadowCaster[h].verts[3].x-lightPosit[0];
```

```
            double yys3= shadowCaster[h].verts[3].y-lightPosit[1];
            double zzs3= shadowCaster[h].verts[3].z-lightPosit[2];


    glNewList( shadowCaster[h].shadowVolume, GL_COMPILE );
    {
        glDisable( GL_LIGHTING );

        glBegin( GL_QUADS );
        {

                              glVertex3f( xx1, yy1, zz1 );
                              glVertex3f( xx0, yy0, zz0 );
                              glVertex4f( xxs0, yys0, zzs0, 0.0f );
                              glVertex4f( xxs1, yys1, zzs1, 0.0f );


                              glVertex3f( xx2, yy2, zz2 );
                              glVertex3f( xx1, yy1, zz1 );
                              glVertex4f( xxs1, yys1, zzs1, 0.0f );
                              glVertex4f( xxs2, yys2, zzs2, 0.0f );

                              glVertex3f( xx3, yy3, zz3 );
                              glVertex3f( xx2, yy2, zz2 );
                              glVertex4f( xxs2, yys2, zzs2, 0.0f );
                              glVertex4f( xxs3, yys3, zzs3, 0.0f );


                              glVertex3f( xx0, yy0, zz0 );
                              glVertex3f( xx3, yy3, zz3 );
                              glVertex4f( xxs3, yys3, zzs3, 0.0f );
                              glVertex4f( xxs0, yys0, zzs0, 0.0f );


                  // Virsutinis dangtis
            glVertex3f( xx0, yy0, zz0 );
            glVertex3f( xx1, yy1, zz1 );
            glVertex3f( xx2, yy2, zz2 );
            glVertex3f( xx3, yy3, zz3 );


            // Apatinis dangtis
            glVertex4f( xxs3, yys3, zzs3, 0.0f);
            glVertex4f( xxs2, yys2, zzs2, 0.0f);
            glVertex4f( xxs1, yys1, zzs1, 0.0f);
            glVertex4f( xxs0, yys0, zzs0, 0.0f);
        }
        glEnd();

        glEnable( GL_LIGHTING );
    }
    glEndList();
}

}

//------------------------------------------------------------------------------
// Pradiniai nustatymai pries scenos generavima
//------------------------------------------------------------------------------
```

```
void init( void )
{


    GLuint PixelFormat;

    PIXELFORMATDESCRIPTOR pfd;
    memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));

    pfd.nSize        = sizeof(PIXELFORMATDESCRIPTOR);
    pfd.nVersion     = 1;
    pfd.dwFlags      = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
    pfd.iPixelType   = PFD_TYPE_RGBA;
    pfd.cColorBits   = 32; // 32 bit spalvos
    pfd.cDepthBits   = 24;
    pfd.cAlphaBits   = 8;
    pfd.cStencilBits = 8;  // Sabloninio buferio bitai

    g_hDC = GetDC( g_hWnd );
    PixelFormat = ChoosePixelFormat( g_hDC, &pfd );
    SetPixelFormat( g_hDC, PixelFormat, &pfd);
    g_hRC = wglCreateContext( g_hDC );
      wglMakeCurrent( g_hDC, g_hRC );


    SelectObject (g_hDC, GetStockObject(SYSTEM_FONT));


    //glClearColor( 0.35f, 0.53f, 0.7f, 1.0f );
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 45.0f, 800.0f / 600.0f, 1.0f, -1.0f);

    // ijungti viena OpenGL sviesos saltini
    float lightAmbient[] = {0.2f, 0.2f, 0.2f, 1.0f};
    float lightDiffuse[] = {1.0, 1.0, 1.0, 1.0};
    float lightSpecular[] = {1.0, 1.0, 1.0, 1.0};
    glLightfv(GL_LIGHT0, GL_DIFFUSE, lightDiffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, lightSpecular);
    glLightfv(GL_LIGHT0, GL_AMBIENT, lightAmbient);


    // Seselio metejo nustatymas...

    // virsunes skaityt is failo "duom.txt"
      float v1[100], v2[100], v3[100];
      int k=0, n=0;
      f>>psk;

      for(int i=0; i<psk; i++)
      { shadowCaster[i].verts = V0;}

      for(int j=0; j<psk; j++)
      {
            f>>k;
```

```
            for (int i = 0; i < k; i++)
            {
                  f>>v1[n]>>v2[n]>>v3[n];
                  shadowCaster[j].verts[i].x=v1[n];
                  shadowCaster[j].verts[i].y=v2[n];
                  shadowCaster[j].verts[i].z=v3[n];
                  n++;
            }
            shadowCaster[j].numVerts=k;
            shadowCaster[j].normal = shadowCasterNormal;
            shadowCaster[j].shadowVolume = -1;
      }
    f.close();
}

//-----------------------------------------------------------------------------
// shutDown()
//-----------------------------------------------------------------------------
void shutDown( void )
{
    if( g_hRC != NULL )
    {
        wglMakeCurrent( NULL, NULL );
        wglDeleteContext( g_hRC );
        g_hRC = NULL;
    }

    if( g_hDC != NULL )
    {
        ReleaseDC( g_hWnd, g_hDC );
        g_hDC = NULL;
    }
}

//-----------------------------------------------------------------------------
// generuoja scenos poligonus
//-----------------------------------------------------------------------------
void renderScene( void )
{

    //ziurejimo taskas
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    glTranslatef( 0.0f, -3.0f, ziurejimo_z );
    glRotatef( -g_fSpinY_L, 1.0f, 0.0f, 0.0f );
    glRotatef( -g_fSpinX_L, 0.0f, 1.0f, 0.0f );


    glDisable( GL_LIGHTING );

    glPushMatrix();
    {
        // padet sviesa...
        glLightfv( GL_LIGHT0, GL_POSITION, lightPosition );
            // nupiest svieso saltini kaip sfera
        glTranslatef( lightPosition[0], lightPosition[1], lightPosition[2] );
        glColor3f(1.0f, 1.0f, 0.5f);
        renderSolidSphere( 0.1, 8, 8 );
    }
```

```
    glPopMatrix();

    glEnable( GL_LIGHTING );

    // Seseliu meteju poligonu generavimas

    glPushMatrix();
    {

            for(int h=0; h<psk; h++)
            {
                glBegin( GL_POLYGON );
                  {
                        glNormal3fv( shadowCaster[h].normal );

                        for( int i = 0; i < shadowCaster[h].numVerts; ++i )
                        {
                              glVertex3f( shadowCaster[h].verts[i].x,
                                    shadowCaster[h].verts[i].y,
                                    shadowCaster[h].verts[i].z );
                        }
                  }
                  glEnd();
             }

    }
    glPopMatrix();
}

//-----------------------------------------------------------------------------
// generuot galutine scena
//-----------------------------------------------------------------------------
void render( void )
{

    // pagal seselio metejo virsunes ir svieso kordinates
      // nustame seseliu turius

      for(int h=0; h<psk; h++)
      {
         buildShadowVolume( &shadowCaster[100], lightPosition, AmountOfExtrusion, h
);
      }


    // Buferiu isvalymas
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );

    // Is naujo uzpildome buferius sugenerave scenos poligonus
    glDisable(GL_LIGHT0);
    renderScene();

    glEnable(GL_LIGHT0);


    // Sukurt sabloninius seselius

    glColorMask( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE );
    glEnable( GL_CULL_FACE );
```

```
    glEnable( GL_STENCIL_TEST );
    glDepthMask( GL_FALSE );

    // Render the shadow volume and increment the stencil every where a front
    // facing polygon is rendered.


    //kai gylio testas paveda padidiname pikselio šabloninę vertę
      glStencilFunc( GL_ALWAYS, 0, ~0 );
    glStencilOp( GL_KEEP, GL_INCR, GL_KEEP ); //didinama pikselio sablonine reišme
    glCullFace( GL_FRONT ); // iširpti isore atsuktas sienas
      for(int h=0; h<psk; h++)
      {
            glCallList( shadowCaster[h].shadowVolume );// iskviesti musu seselio
meteju sarasus
      }
    // pikseliu esanciu tarp akies ir priekiu atsuktu sienu pikseliu sablonines
      // reiksmes pamazinti

    glStencilOp( GL_KEEP, GL_DECR, GL_KEEP ); // mazinama pikselio sabloninė reisme
kai gylio testas paveda
    glCullFace( GL_BACK ); // iskirpti vidum atsuktas sienas iskviesti musu seselio
meteju sarasus
    for(int h=0; h<psk; h++)
      {
            glCallList( shadowCaster[h].shadowVolume );// iskviesti musu seselio
meteju sarasus
      }
    // Atstome pradines reiksmes
    glDepthMask( GL_TRUE );
    glDepthFunc( GL_LEQUAL );
    glColorMask( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE );
    glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP );
    glCullFace( GL_BACK );
    glDisable( GL_CULL_FACE );

    glStencilFunc( GL_EQUAL, 0, ~0 );
    glEnable( GL_LIGHT0 );
    renderScene();
    glDepthFunc( GL_LESS );
    glDisable( GL_STENCIL_TEST );

      // rodyti turiu konturas kai paspaudzia F1
    if( RenderShadowVolume )
    {
        glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
            for(int h=0; h<psk; h++)
            {
                glCallList( shadowCaster[h].shadowVolume );
            }
        glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
    }
    SwapBuffers( g_hDC );
}
```

2 PRIEDAS.

Šešėlių planus realizuojančios programos kodas

Failas **shadow_mapping.cpp**:

```
//--------------------------------------------------------------------------------
//  Valdymo mygtukai: Up    - sviesos saltinis juda i prieki
//                    Down  - sviesos saltinis juda atgal
//                    Left  - sviesos saltinis juda i kaire
//                    Right - sviesos saltinis juda i desine
//                    Enter - sviesos saltinis juda i virsu
//                    Backspace - sviesos saltinis juda zemyn
//                    Left Mouse Button  - sukineti kamera
//                    F1 - rodyti gylio tekstura
//                    F2 - priartinti kamera prie sukinejimo centro
//                    F3 - atitolinti kamera nuo sukinejimo centro
//
//  Poligonai ivedami is failo "duom.txt"
//--------------------------------------------------------------------------------

#define STRICT
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include "geometry.h"
#include "resource.h"
#include <iostream>
#include <iomanip>
#include <fstream>
#include <omp.h>
#include <string>
using namespace std;
ifstream f("duom.txt");
//-------------------------------------------------------------------------------
// Funkciju pointer'iai OpenGl pletiniams
//-------------------------------------------------------------------------------
#include "glext.h"
//#include <GL/glext.h>
#include "wglext.h"
//#include <GL/wglext.h>

// WGL_ARB_extensions_string
PFNWGLGETEXTENSIONSSTRINGARBPROC wglGetExtensionsStringARB = NULL;
// WGL_ARB_pbuffer
PFNWGLCREATEPBUFFERARBPROC       wglCreatePbufferARB       = NULL;
PFNWGLGETPBUFFERDCARBPROC        wglGetPbufferDCARB        = NULL;
PFNWGLRELEASEPBUFFERDCARBPROC    wglReleasePbufferDCARB    = NULL;
PFNWGLDESTROYPBUFFERARBPROC      wglDestroyPbufferARB      = NULL;
// WGL_ARB_pixel_format
PFNWGLCHOOSEPIXELFORMATARBPROC   wglChoosePixelFormatARB   = NULL;
// WGL_ARB_render_texture
PFNWGLBINDTEXIMAGEARBPROC        wglBindTexImageARB        = NULL;
PFNWGLRELEASETEXIMAGEARBPROC     wglReleaseTexImageARB     = NULL;

struct Vertex
{
      float x, y, z;
};
```

```
struct ShadowCaster0 // seselio metejas
{
    Vertex *verts;        // virsunes
      float  *normal;       // pavirsiaus apsvietimo reiksmes
    int     numVerts;     // virsuniu skaicius
};

Vertex V0[] = {{ 0, 0, 0 },{0, 0, 0 },{ 0, 0, 0 },{ 0, 0, 0 }};
int psk = 0;

ShadowCaster0 shadowCaster[100];
float shadowCasterNormal[] = { 0.0f, 1.0f, 0.0f};

//-----------------------------------------------------------------------------
// Globalus
//-----------------------------------------------------------------------------
HWND   g_hWnd = NULL;
HDC      g_hDC  = NULL;
HGLRC  g_hRC  = NULL;
GLuint g_depthTexture = -1;
DEVMODE g_oldDevMode;

//vaizdo sukinejimui
float g_fSpinX_L =  0.0f;
float g_fSpinY_L = -10.0f;
float ziurejimo_z = -18.0;

float lightsLookAtMatrix[16]; // sviesos ziurejimo matrica
float lightPosition[] = { 0.0f, 5.0f, 10.0f, 1.0f };
bool RenderDepthTexture = false;

// This little struct will help to organize our p-buffer's data
struct PBUFFER
{
    HPBUFFERARB hPBuffer; // Panaudoti p-buferyje
    HDC         hDC;      // Panaudoti prietaiso kontekste
    HGLRC       hRC;      // Panaudoti grafinio vaizdavimo lango kontekste
    int         nWidth;   // P-buferio plotis
    int         nHeight;  // P-buferio aukštis
};

PBUFFER PBuferis;

const int PBUFFER_WIDTH  = 2048;
const int PBUFFER_HEIGHT = 2048;

//-----------------------------------------------------------------------------
// PROTOTYPES
//-----------------------------------------------------------------------------
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,
                           LPSTR lpCmdLine, int nCmdShow);
LRESULT CALLBACK WindowProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
void init(void);
void shutDown(void);
void initExtensions(void);
void initPbuffer(void);
void render(void);
void renderScene(void);
void createDepthTexture(void);
```

```
void displayDepthTexture(void);

//-----------------------------------------------------------------------------
// Sasajos lango paruosimas
//-----------------------------------------------------------------------------
int WINAPI WinMain(     HINSTANCE hInstance,
                              HINSTANCE hPrevInstance,
                              LPSTR     lpCmdLine,
                              int       nCmdShow )
{
      WNDCLASSEX winClass;
      MSG        uMsg;

    memset(&uMsg,0,sizeof(uMsg));

      winClass.lpszClassName = "MY_WINDOWS_CLASS";
      winClass.cbSize        = sizeof(WNDCLASSEX);
      winClass.style         = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
      winClass.lpfnWndProc   = WindowProc;
      winClass.hInstance     = hInstance;
    winClass.hIcon             = LoadIcon(hInstance, (LPCTSTR)IDI_OPENGL_ICON);
    winClass.hIconSm       = LoadIcon(hInstance, (LPCTSTR)IDI_OPENGL_ICON);
      winClass.hCursor       = LoadCursor(NULL, IDC_ARROW);
      winClass.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
      winClass.lpszMenuName  = NULL;
      winClass.cbClsExtra    = 0;
      winClass.cbWndExtra    = 0;

      if( !RegisterClassEx(&winClass) )
            return E_FAIL;

      g_hWnd = CreateWindowEx( NULL, "MY_WINDOWS_CLASS",
                             "OpenGL - Shadow Mapping",
                                         WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                                                0, 0, 800, 600, NULL, NULL,
hInstance, NULL );

      if( g_hWnd == NULL )
            return E_FAIL;

    ShowWindow( g_hWnd, nCmdShow );
    UpdateWindow( g_hWnd );

      init();

      while( uMsg.message != WM_QUIT )
      {
            if( PeekMessage( &uMsg, NULL, 0, 0, PM_REMOVE ) )
            {
                  TranslateMessage( &uMsg );
                  DispatchMessage( &uMsg );
            }
        else
                render();
      }

      shutDown();

    UnregisterClass( "MY_WINDOWS_CLASS", winClass.hInstance );
```

```
      return uMsg.wParam;
}

//-----------------------------------------------------------------------------
// Valdymo nustatymas
//-----------------------------------------------------------------------------
LRESULT CALLBACK WindowProc( HWND   hWnd,
                                           UINT   msg,
                                           WPARAM wParam,
                                           LPARAM lParam )
{
      static POINT ptLastMousePosit_L;
      static POINT ptCurrentMousePosit_L;
      static bool  bMousing_L;

      static POINT ptLastMousePosit_R;
      static POINT ptCurrentMousePosit_R;
      static bool  bMousing_R;

    switch( msg )
      {
        case WM_KEYDOWN:
            {
                  switch( wParam )
                  {
                        case VK_ESCAPE:
                              PostQuitMessage(0);
                              break;

                case VK_F1:
                    RenderDepthTexture = !RenderDepthTexture;
                            break;

                        case VK_F2:
                              ziurejimo_z += 0.2f;
                         break;

                        case VK_F3:
                              ziurejimo_z -= 0.2f;
                      break;

                case 38: // Up Arrow Key
                              lightPosition[2] -= 0.1f;
                              break;

                        case 40: // Down Arrow Key
                              lightPosition[2] += 0.1f;
                              break;

                        case 37: // Left Arrow Key
                              lightPosition[0] -= 0.1f;
                              break;

                        case 39: // Right Arrow Key
                              lightPosition[0] += 0.1f;
                              break;
                        case VK_BACK: //Backspace
                              lightPosition[1] -= 0.1f;
```

```
                break;

                        case VK_RETURN: // Enter
                              lightPosition[1] += 0.1f;
                break;

                  }
            }
        break;

            case WM_LBUTTONDOWN:
            {
                  ptLastMousePosit_L.x = ptCurrentMousePosit_L.x = LOWORD (lParam);
            ptLastMousePosit_L.y = ptCurrentMousePosit_L.y = HIWORD (lParam);
                  bMousing_L = true;
            }
            break;

            case WM_LBUTTONUP:
            {
                  bMousing_L = false;
            }
            break;

            case WM_RBUTTONDOWN:
            {
                  ptLastMousePosit_R.x = ptCurrentMousePosit_R.x = LOWORD (lParam);
            ptLastMousePosit_R.y = ptCurrentMousePosit_R.y = HIWORD (lParam);
                  bMousing_R = true;
            }
            break;

            case WM_RBUTTONUP:
            {
                  bMousing_R = false;
            }
            break;

            case WM_MOUSEMOVE:
            {
                  ptCurrentMousePosit_L.x = LOWORD (lParam);
                  ptCurrentMousePosit_L.y = HIWORD (lParam);
                  ptCurrentMousePosit_R.x = LOWORD (lParam);
                  ptCurrentMousePosit_R.y = HIWORD (lParam);

                  if( bMousing_L )
                  {
                        g_fSpinX_L -= (ptCurrentMousePosit_L.x -
ptLastMousePosit_L.x);
                        g_fSpinY_L -= (ptCurrentMousePosit_L.y -
ptLastMousePosit_L.y);
                  }



                  ptLastMousePosit_L.x = ptCurrentMousePosit_L.x;
            ptLastMousePosit_L.y = ptCurrentMousePosit_L.y;
                  ptLastMousePosit_R.x = ptCurrentMousePosit_R.x;
            ptLastMousePosit_R.y = ptCurrentMousePosit_R.y;
```

```
            }
            break;

            case WM_SIZE:
            {
                  int nWidth  = LOWORD(lParam);
                  int nHeight = HIWORD(lParam);
                  glViewport(0, 0, nWidth, nHeight);

                  glMatrixMode( GL_PROJECTION );
                  glLoadIdentity();
                  gluPerspective( 45.0, (GLdouble)nWidth / (GLdouble)nHeight, 0.1,
100.0);
            }
            break;

            case WM_CLOSE:
            {
                  PostQuitMessage(0);
            }
            break;

        case WM_DESTROY:
            {
            PostQuitMessage(0);
            }
        break;

            default:
            {
                  return DefWindowProc( hWnd, msg, wParam, lParam );
            }
            break;
      }

      return 0;
}

//-----------------------------------------------------------------------------
// Pletiniu nustatymai
//-----------------------------------------------------------------------------
void initExtensions( void )
{

    // wglGetExtensionsStringARB

    wglGetExtensionsStringARB =
(PFNWGLGETEXTENSIONSSTRINGARBPROC)wglGetProcAddress("wglGetExtensionsStringARB");
    char *wgl_ext = NULL;

    if( wglGetExtensionsStringARB )
        wgl_ext = (char*)wglGetExtensionsStringARB( wglGetCurrentDC() );
    else
    {
        MessageBox(NULL,"Unable to get address for wglGetExtensionsStringARB!",
            "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
    }
```

```
    // WGL_ARB_pbuffer

    if( strstr( wgl_ext, "WGL_ARB_pbuffer" ) == NULL )
    {
        MessageBox(NULL,"WGL_ARB_pbuffer extension was not found",
            "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
    }
    else
    {
        wglCreatePbufferARB    =
(PFNWGLCREATEPBUFFERARBPROC)wglGetProcAddress("wglCreatePbufferARB");
        wglGetPbufferDCARB     =
(PFNWGLGETPBUFFERDCARBPROC)wglGetProcAddress("wglGetPbufferDCARB");
        wglReleasePbufferDCARB =
(PFNWGLRELEASEPBUFFERDCARBPROC)wglGetProcAddress("wglReleasePbufferDCARB");
        wglDestroyPbufferARB   =
(PFNWGLDESTROYPBUFFERARBPROC)wglGetProcAddress("wglDestroyPbufferARB");

        if( !wglCreatePbufferARB || !wglGetPbufferDCARB ||
            !wglReleasePbufferDCARB || !wglDestroyPbufferARB )
        {
            MessageBox(NULL,"One or more WGL_ARB_pbuffer functions were not found",
                "ERROR",MB_OK|MB_ICONEXCLAMATION);
            exit(-1);
        }
    }

    // WGL_ARB_pixel_format

    if( strstr( wgl_ext, "WGL_ARB_pixel_format" ) == NULL )
    {
        MessageBox(NULL,"WGL_ARB_pixel_format extension was not found",
            "ERROR",MB_OK|MB_ICONEXCLAMATION);
        return;
    }
    else
    {
        wglChoosePixelFormatARB  =
(PFNWGLCHOOSEPIXELFORMATARBPROC)wglGetProcAddress("wglChoosePixelFormatARB");

        if( !wglCreatePbufferARB || !wglGetPbufferDCARB )
        {
            MessageBox(NULL,"One or more WGL_ARB_pixel_format functions were not
found",
                "ERROR",MB_OK|MB_ICONEXCLAMATION);
            exit(-1);
        }
    }

    // WGL_ARB_render_texture

    if( strstr( wgl_ext, "WGL_ARB_render_texture" ) == NULL )
    {
        MessageBox(NULL,"WGL_ARB_render_texture extension was not found",
            "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
    }
    else
```

```
    {
        wglBindTexImageARB    =
(PFNWGLBINDTEXIMAGEARBPROC)wglGetProcAddress("wglBindTexImageARB");
        wglReleaseTexImageARB =
(PFNWGLRELEASETEXIMAGEARBPROC)wglGetProcAddress("wglReleaseTexImageARB");

        if( !wglBindTexImageARB || !wglReleaseTexImageARB )
        {
            MessageBox(NULL,"One or more WGL_ARB_render_texture functions were not
found",
                "ERROR",MB_OK|MB_ICONEXCLAMATION);
            exit(-1);
        }
    }
}

//-----------------------------------------------------------------------------
// sukurti p-buferi gylio teksturu generavimui
//-----------------------------------------------------------------------------
void initPbuffer( void )
{
    PBuferis.hPBuffer = NULL;
    PBuferis.nWidth   = PBUFFER_WIDTH;
    PBuferis.nHeight  = PBUFFER_HEIGHT;

    // Nustatome minimalius pikselio formato reikalavimus musu p-buferiui.
      // P-buferis yra kaip kadrų buferis, jis gali turėti gylio buferį susijusį su
juo

    int pf_attr[] =
    {
        WGL_SUPPORT_OPENGL_ARB, TRUE,       // P-buferis bus panaudotas su OpenGL
        WGL_DRAW_TO_PBUFFER_ARB, TRUE,      // Leisti generavimą i p-buferi
            WGL_BIND_TO_TEXTURE_DEPTH_NV, TRUE, // Reikalauti gylio teksturos
        WGL_BIND_TO_TEXTURE_RGBA_ARB, TRUE, // P-buferis bus panaudotas kaip
tekstura
        WGL_DOUBLE_BUFFER_ARB, FALSE,       // Mums nereikia dvigubo buferiavimo
        0                                   // Nulis nutraukia sarasa
    };

    unsigned int count = 0;
    int pixelFormat;

    if( !wglChoosePixelFormatARB( g_hDC, pf_attr, NULL, 1, &pixelFormat, &count ) )
    {
        MessageBox(NULL,"pbuffer creation error:  wglChoosePixelFormatARB()
failed.",
                "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
    }

    if( count <= 0 )
    {
        MessageBox(NULL,"pbuffer creation error:  Couldn't find a suitable pixel
format.",
                "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
    }
```

```
    // Nustatome keleta p-buferio atributu, kad ji galetume naudoti kaip 2D RGBA
teksturu objekta

    int pb_attr[] =
    {
        WGL_DEPTH_TEXTURE_FORMAT_NV, WGL_TEXTURE_DEPTH_COMPONENT_NV,//Mums reikia
sugeneruot i gylio tekstura
        WGL_TEXTURE_FORMAT_ARB, WGL_TEXTURE_RGBA_ARB,               // musu buferis
tures RGBA teksturu formata
        WGL_TEXTURE_TARGET_ARB, WGL_TEXTURE_2D_ARB,                 // Teksturu
objektas bus GL_TEXTURE_2D
        0                                                           // Nulis
nutraukia sarasa
    };

    // sukurt p-buferi...

    PBuferis.hPBuffer = wglCreatePbufferARB( g_hDC, pixelFormat, PBuferis.nWidth,
PBuferis.nHeight, pb_attr );
    PBuferis.hDC      = wglGetPbufferDCARB( PBuferis.hPBuffer );
    PBuferis.hRC      = wglCreateContext( PBuferis.hDC );

    if( !PBuferis.hPBuffer )
    {
        MessageBox(NULL,"pbuffer creation error: wglCreatePbufferARB() failed!",
                  "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
    }
}

//-----------------------------------------------------------------------------
// Pradiniai nustatymai pries scenos generavima
//-----------------------------------------------------------------------------
void init( void )
{

      GLuint PixelFormat;


      PIXELFORMATDESCRIPTOR pfd;
      memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));

    pfd.nSize      = sizeof(PIXELFORMATDESCRIPTOR);
    pfd.nVersion   = 1;
    pfd.dwFlags    = PFD_DRAW_TO_WINDOW |PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
    pfd.iPixelType = PFD_TYPE_RGBA;
    pfd.cColorBits = 32;
    pfd.cDepthBits = 24;
      g_hDC = GetDC( g_hWnd );
      PixelFormat = ChoosePixelFormat( g_hDC, &pfd );
      SetPixelFormat( g_hDC, PixelFormat, &pfd);
      g_hRC = wglCreateContext( g_hDC );
      wglMakeCurrent( g_hDC, g_hRC );

    glEnable( GL_LIGHTING );
      glEnable( GL_TEXTURE_2D );
      glEnable( GL_DEPTH_TEST );

      glMatrixMode( GL_PROJECTION );
```

```
      glLoadIdentity();
      gluPerspective( 45.0f, 800.0 / 600.0f, 0.1f, 100.0f);

      // ijungti aplinkos apsvietima

      GLfloat ambient_lightModel[] = { 0.5f, 0.5f, 0.5f, 1.0f };
      glLightModelfv( GL_LIGHT_MODEL_AMBIENT, ambient_lightModel );

      // nustatyti taskini sviesos saltini...

    glEnable( GL_LIGHT0 );
    GLfloat diffuse_light[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    GLfloat linearAttenuation_light[] = { 0.0f };
      glLightfv( GL_LIGHT0, GL_DIFFUSE, diffuse_light );
      glLightfv( GL_LIGHT0, GL_LINEAR_ATTENUATION , linearAttenuation_light );

    // Inicializuoja p-buferi dabar, kad turetumem tinkama
      // konteksta, kuri galetume panaudot p-buferio kurimo procesu

    initExtensions();
    initPbuffer();

    // Inicializuojame keleta grafiniu busenu p-buferio generavimui

    if( wglMakeCurrent( PBuferis.hDC, PBuferis.hRC) == FALSE )
    {
            MessageBox(NULL,"Could not make the p-buffer's context current!",
                  "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
    }

    glEnable( GL_LIGHTING );
    glEnable( GL_LIGHT0 );
    glEnable( GL_DEPTH_TEST );

    glLightModelfv( GL_LIGHT_MODEL_AMBIENT, ambient_lightModel );

    if( wglMakeCurrent( g_hDC, g_hRC ) == FALSE )
    {
            MessageBox(NULL,"Could not make the window's context current!",
                  "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
    }

    // Sukurt gylio tekstura

    glGenTextures( 1,&g_depthTexture );
    glBindTexture( GL_TEXTURE_2D, g_depthTexture );

    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_SGIX, GL_TRUE );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_OPERATOR_SGIX,
GL_TEXTURE_LEQUAL_R_SGIX );

    // Seselio meteju paruosimas
```

```
      for(int i=0; i<psk; i++)
      { shadowCaster[i].verts = V0;}

      //virsunes skaityt is failo
      float v1[100], v2[100], v3[100];
      int k=0, n=0;
      f>>psk;
      for(int j=0; j<psk; j++)
      {
            f>>k;
            for (int i = 0; i < k; i++)
            {
                  f>>v1[n]>>v2[n]>>v3[n];
                  shadowCaster[j].verts[i].x=v1[n];
                  shadowCaster[j].verts[i].y=v2[n];
                  shadowCaster[j].verts[i].z=v3[n];
                  n++;
            }
            shadowCaster[j].numVerts=k;
        shadowCaster[j].normal = shadowCasterNormal;
      }
    f.close();

}

//-----------------------------------------------------------------------------
// shutDown()
//-----------------------------------------------------------------------------
void shutDown( void )
{
    glDeleteTextures( 1, &g_depthTexture );

      if( g_hRC != NULL )
      {
            wglMakeCurrent( NULL, NULL );
            wglDeleteContext( g_hRC );
            g_hRC = NULL;
      }

      if( g_hDC != NULL )
      {
            ReleaseDC( g_hWnd, g_hDC );
            g_hDC = NULL;
      }

      // Nepamirsti isvalyt p-buferio baigus darba

      if( PBuferis.hRC != NULL )
      {
            wglMakeCurrent( PBuferis.hDC, PBuferis.hRC );
            wglDeleteContext( PBuferis.hRC );
            wglReleasePbufferDCARB( PBuferis.hPBuffer, PBuferis.hDC );
            wglDestroyPbufferARB( PBuferis.hPBuffer );
            PBuferis.hRC = NULL;
      }

      if( PBuferis.hDC != NULL )
      {
            ReleaseDC( g_hWnd, PBuferis.hDC );
```

```
            PBuferis.hDC = NULL;
      }
}

//-----------------------------------------------------------------------------
// Parodyti gylio tekstura
//-----------------------------------------------------------------------------
void displayDepthTexture( void )
{

    glDisable( GL_LIGHTING );

    glViewport( 0, 0, 800, 600 );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluOrtho2D( -1.0, 1.0, -1.0, 1.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();

    glMatrixMode( GL_TEXTURE );
    glLoadIdentity();
    glEnable( GL_TEXTURE_2D );
    glBindTexture( GL_TEXTURE_2D, g_depthTexture );

    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_SGIX, GL_FALSE );


    if( wglBindTexImageARB( PBuferis.hPBuffer, WGL_DEPTH_COMPONENT_NV ) == FALSE )
    {
            MessageBox(NULL,"Could not bind p-buffer to render texture!",
                  "ERROR",MB_OK|MB_ICONEXCLAMATION);
            exit(-1);
      }

    glBegin( GL_QUADS );
    {
        glTexCoord2f( 0.0f, 0.0f );
        glVertex3f( -1.0f, -1.0f, 0.0f );

        glTexCoord2f( 0.0f, 1.0f );
        glVertex3f( -1.0f, 1.0f, 0.0f );

        glTexCoord2f( 1.0f, 1.0f );
        glVertex3f( 1.0f, 1.0f, 0.0f );

        glTexCoord2f( 1.0f, 0.0f );
        glVertex3f( 1.0f, -1.0f, 0.0f );
    }
    glEnd();

    glDisable( GL_TEXTURE_2D );

    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_SGIX, GL_TRUE );

    if( wglReleaseTexImageARB( PBuferis.hPBuffer, WGL_DEPTH_COMPONENT_NV ) == FALSE
)
    {
            MessageBox(NULL,"Could not release p-buffer from render texture!",
                  "ERROR",MB_OK|MB_ICONEXCLAMATION);
```

```
            exit(-1);
      }
}

//------------------------------------------------------------------------------
// Sukuria gylio tekstura
//------------------------------------------------------------------------------
void createDepthTexture( void )
{
    // padarom p-buferio kontekstą einamuoju
    if( wglMakeCurrent( PBuferis.hDC, PBuferis.hRC) == FALSE )
    {
            MessageBox(NULL,"Could not make the p-buffer's context current!",
                  "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
    }

    glClear( GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT );

    glPolygonOffset( 3.0f, 0.0f );
    glEnable( GL_POLYGON_OFFSET_FILL );

      glLoadIdentity();
    glMatrixMode( GL_PROJECTION );

      //gylio teksturos kurimo perspekyva
      glLoadIdentity();
      gluPerspective( 95.0f, 1.5f, 0.5f, 100.0f);
    glViewport( 0, 0, PBUFFER_WIDTH, PBUFFER_HEIGHT );
    glMatrixMode( GL_MODELVIEW );
    glMultMatrixf( lightsLookAtMatrix);

    renderScene();

    glDisable( GL_POLYGON_OFFSET_FILL );

    // Padarome lango konteksta einamuoju vel
    if( wglMakeCurrent( g_hDC, g_hRC ) == FALSE )
    {
            MessageBox(NULL,"Could not make the window's context current!",
                  "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
    }
}

//------------------------------------------------------------------------------
// Scenos poligonu generavimas
//------------------------------------------------------------------------------
void renderScene( void )
{
    glMatrixMode( GL_MODELVIEW );

    glPushMatrix();
    {

            for(int h=0; h<psk; h++)
            {
                glBegin( GL_POLYGON );
                  {
```

```
                        glNormal3fv( shadowCaster[h].normal );
                        for( int i = 0; i < shadowCaster[h].numVerts; ++i )
                        {
                              glVertex3f( shadowCaster[h].verts[i].x,
                                    shadowCaster[h].verts[i].y,
                                    shadowCaster[h].verts[i].z );
                        }
                  }
                  glEnd();
            }
    }
    glPopMatrix();

}

//-----------------------------------------------------------------------------
// Name: Galutines scenos generavimas
//-----------------------------------------------------------------------------
void render( void )
{
    // Pradziai sukuriame sviesos ziūrejimo matrica, reikalinga gylio teksturos
generavimui

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();

    gluLookAt( lightPosition[0], lightPosition[1], lightPosition[2], // koordinates
               0.0f, 1.0f, 0.0f,   // nurodom tasko koordinates i kuri ziures sviesa
               0.0f, 0.1f, 0.0f ); // ziurejimo vektorius
    glGetFloatv( GL_MODELVIEW_MATRIX, lightsLookAtMatrix );

    // sukurt gylio tekstura...

    createDepthTexture();

    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    // Padedam ziurejimo taska

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    glTranslatef( 0.0f, -1.0f, ziurejimo_z );
    glRotatef( -g_fSpinY_L, 1.0f, 0.0f, 0.0f );
    glRotatef( -g_fSpinX_L, 0.0f, 1.0f, 0.0f );

    // generuojame sviesos pozicija kaip sfera...

    glDisable( GL_LIGHTING );

    glPushMatrix();
    {
        glLightfv( GL_LIGHT0, GL_POSITION, lightPosition );
        glTranslatef( lightPosition[0], lightPosition[1], lightPosition[2] );
        glColor3f(1.0f, 1.0f, 0.5f);
        renderSolidSphere( 0.1, 8, 8 );
    }
    glPopMatrix();

    glEnable( GL_LIGHTING );
```

```
// Nustatome OpenGl busena gylio palyginimui naudojant gylio tekstura

glEnable( GL_LIGHTING );

float x[] = { 1.0f, 0.0f, 0.0f, 0.0f };
float y[] = { 0.0f, 1.0f, 0.0f, 0.0f };
float z[] = { 0.0f, 0.0f, 1.0f, 0.0f };
float w[] = { 0.0f, 0.0f, 0.0f, 1.0f };
glTexGenfv( GL_S, GL_EYE_PLANE, x );
glTexGenfv( GL_T, GL_EYE_PLANE, y );
glTexGenfv( GL_R, GL_EYE_PLANE, z );
glTexGenfv( GL_Q, GL_EYE_PLANE, w );

glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
glTexGeni( GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
glTexGeni( GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
glTexGeni( GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );

glEnable( GL_TEXTURE_GEN_S );
glEnable( GL_TEXTURE_GEN_T );
glEnable( GL_TEXTURE_GEN_R );
glEnable( GL_TEXTURE_GEN_Q );

// nustatyti gylio teksturu projekcija
glMatrixMode( GL_TEXTURE );
glLoadIdentity();
glTranslatef( 0.5f, 0.5f, 0.5f );              // kontrastas/Offset
glScalef( 0.5f, 0.5f, 0.5f );                  // Paklaida/Bias
gluPerspective( 95.0f, 1.5f, 0.5f, 100.0f);    // sviesos ziurejimo kugis
glMultMatrixf( lightsLookAtMatrix );           // Light matrix

// Surist teksuras su gylio tekstura, kad galetume naudot kaip seseliu planus...

glEnable( GL_TEXTURE_2D );
glBindTexture( GL_TEXTURE_2D, g_depthTexture );

if( wglBindTexImageARB( PBuferis.hPBuffer, WGL_DEPTH_COMPONENT_NV ) == FALSE )
{
        MessageBox(NULL,"Could not bind p-buffer to render texture!",
              "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
  }

renderScene();

if( wglReleaseTexImageARB( PBuferis.hPBuffer, WGL_DEPTH_COMPONENT_NV ) == FALSE
)
{
        MessageBox(NULL,"Could not release p-buffer from render texture!",
              "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
  }

// Atstomai kai kurias pradines busenas, kito kadro generavimui

glDisable( GL_TEXTURE_2D );
glDisable( GL_TEXTURE_GEN_S );
glDisable( GL_TEXTURE_GEN_T );
```

```
    glDisable( GL_TEXTURE_GEN_R );
    glDisable( GL_TEXTURE_GEN_Q );

    if( RenderDepthTexture == true )
        displayDepthTexture(); // rodyti gylio tekstura

    SwapBuffers( g_hDC );
}
```