VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

# Nematomų linijų ir paviršių nustatymas

## Hidden-line-surface-removal Algorithms

Baigiamasis bakalauro darbas

Atliko: 4 kurso, 2 grupės studentas
Ernestas Mitkus (parašas)

Darbo vadovas: Doc. A. Lenkevičius (parašas)

Recenzentas: Prof. R. Vaicekauskas (parašas)

Vilnius – 2017

# Turinys

# Įvadas

Kompiuterinė grafika šiais laikais yra neatsiejama nuo darbo kompiuteriu. Ji yra naudojama ir piešiant ant ekrano paprastas aplikacijas, tokias kaip skaičiuoklė, teksto redaguotojas ar nuotraukų galerija, ir atliekant sudėtingesnius darbus, kaip kompiuterinių žaidimų atvaizdavimas.

Prieš kelias dešimtis metų, atsirandant pirmiesiems kompiuteriniams žaidimams, kūrėjams buvo keliamas didžiulis įšūkis kuriant žaidimus. Kompiuteriai turėdavo labai mažai atminties, todėl kūrėjams tekdavo spręsti galvosūkius, kaip atvaizduoti žaidimus sunaudojant kuo mažiau kompiuterio resursų.

Šiais laikais, kompiuterio atmintis nėra tokia didelė problema. Vaizdo plokštės yra nepriklausomos nuo procesoriaus, todėl galima kurti didelės raiškos, spalvingus žaidimus. Tačiau, kuriant didesnės raiškos 3D žaidimus gali iškilti gero spartaus veikimo (angl. *performance*) problema, kai kompiuteris turi nupiešti daug modelių ant ekrano per ypač mažą laiką. Norint atvaizduoti žaidimo kadrą ekrane, reikia įvykdyti visus reikalingus loginius veiksmus pasauliui, žaidėjui ir kitiems žaidime esantiems objektams ir juos nupiešti. Šiais laikais dažniausiai yra naudojami 60 FPS (frames per second, liet. *kadrai per sekundę*), taigi kiekvienam kadrui yra skiriama apie 0.016 sekundės.

Esant komplikuotoms scenoms - turint daug objektų, turint sudėtingus objektų tinklelius, turint daug skaičiavimų tarp atvaizdavimo iteracijų, turint daug vidinių skaičiavimų kiekvienai viršūnei, pikseliui, tą pasiekti gali būti sudėtinga. Šiai problemai išspręsti yra naudojami nematomų linijų ir paviršių nustatymo algoritmai.

Šiame darbe bus apžvelgiami keli klasikiniai nematomų linijų ir paviršių nustatymo algoritmai, sukuriama programa ir atliekami veikimo greičio tyrimai trims populiariems trimačio pasaulio nematomų paviršių nustatymo algoritmams: galinių plokštumų nustatymo, vaizduojamojo tūrio ir Z-buferio metodai.

# Darbo tikslas

Šio darbo tikslas: atlikti klasikinių nematomų linijų ir paviršių nustatymo algoritmų analitinę analizę, sukurti eksperimentinio tyrimo programinę įrangą ir ištirti trijų populiarių, kompiuterinėje grafikoje naudojamų nematomų paviršių nustatymo algoritmų - galinių plokštumų, vaizduojamojo tūrio ir Z-buferio - našumą.

# Darbo uždaviniai

Darbo uždaviniai:

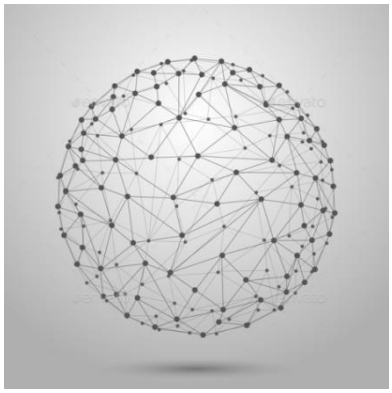
1. Atlikti klasikinių nematomų linijų ir paviršių nustatymo algoritmų analitinę analizę.
2. Atrinkti optimalią trimačio pasaulio atvaizdavimo biblioteką ir išsiaiškinti jos panaudojimo galimybes eksperimentiniam tyrimui atlikti.
3. Sukurti programinę įrangą tiriamų nematomų linijų ir paviršių nustatymo algoritmų eksperimentiniam tyrimui.
4. Sukurti eksperimentiniams tyrimams reikalingas scenas.

5. Atlikti eksperimentinių tyrimų rezultatų analizę.
6. Ištirti įvairių faktorių poveikį realizuoto algoritmo našumui siekiant pagerinti jo kiekybines bei kokybines charakteristikas su tiriama GPU architektūra

# 1. Nematomų linijų ir paviršių nustatymo metodų analitinė apžvalga

## 1.1. Nematomų linijų nustatymas

Kietieji objektai kompiuterinėje grafikoje dažniausiai būna sumodeliuojami briaunainių pagalba. Briaunainio siena yra daugiakampis apribotas tiesėmis, vadinamomis briaunomis. Lenkti paviršiai dažniausiai yra apytiksliai sukuriami daugiakampių rinkiniu (1.1 pav.). Permatomų objektų linijoms nupiešti kompiuterio aplikacija turi sugebėti atskirti kurios linijų dalys yra paslėptos to pačio ar kitų pasaulyje esančių objektų. Ši problema yra žinoma kaip nematomų linijų šalinimas (angl. *hidden line removal*).



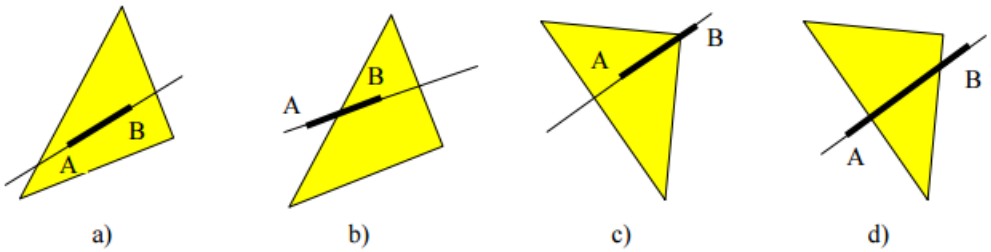
1.1 pav. Sfera išreikšta trikampiais

### 1.1.1. Robertso algoritmas

Pirmąjį sprendimą šiai problemai pasiūlė L. G. Roberts 1963 metais knygoje „Machine perception of three-dimentional solids“ [Rob63]. Algoritmas gali būti suskaidytas į tris dalis, kad būtų galima paruošti liniją piešimui:

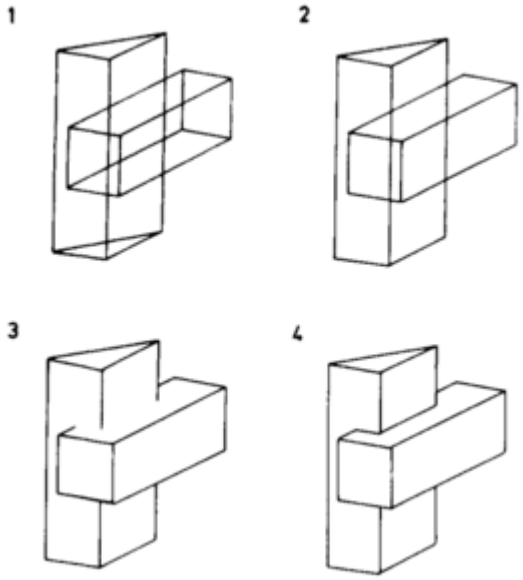
**Pirmasis žingsnis.** Atmetamos visos sienos ar briaunos, kurių abi nustatančios sienos yra nematomos.

**Antrasis žingsnis.** Tikrinamas kiekvienos linijos susiklojimas su visomis priekinėmis briaunainio sienomis. Jeigu briaunainio siena iš dalies dengia liniją (1.2 pav. b, c, d), ta linija yra skaidoma į kelias dalis, iš kurių daugiausiai dvi būna matomos. Išskaidytos linijos yra vėl tikrinamos iš naujo. Jeigu briaunainio siena visiškai uždengia liniją (1.2 pav. a), tokiu atveju linija yra atmetama.




1.2 pav. Linijų matomumo nustatymas

**Trečiasis žingsnis.** Nustatomos naujos briaunos formuojančios atkarpas tais atvejais, kai kūnai kerta vienas kitą.



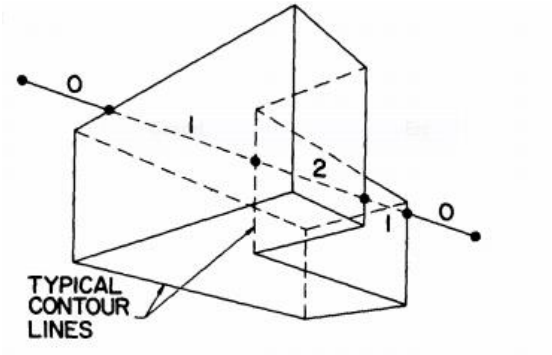

1.3 pav. Linijos, po kiekvieno Robertso algoritmo žingsnių

**Algoritmo trūkumai:** Veikia tik su iškilaisiais briaunainiais

**Algoritmo sudėtingumas:** $O(n^2)$, kai n - bendras sienų skaičius. Patikrinimų skaičių galima gerokai sumažinti, naudojantis vaizdo plokštumos suskaidymu, kai ekranas yra suskaidomas į mažesnius langelius, ir kiekvienam langeliui $A_{ij}$ sudaromas visų priekinių sienų, kurių projekcijose yra šis langelis, sąrašas.

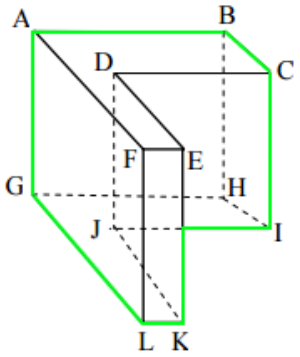
### 1.1.2. Appel algoritmas

Daugiau objektų turinčiame pasaulyje šalinant dengiamąsias linijas naudojant Robertso algoritmą užtruks ypač ilgai. Po kelių metų, nuo Robertso algoritmo išleidimo, 1967 m., A. Appel išleido knygą „The notion of quantitative invisibility and the machine rendering of solids“, kurioje aprašė paprastesnį algoritmą nematomų linijų nustatymą. Šiam algoritmui kūnų ribose išskiriama kontūrinių linijų aibė ir joms skaičiuojamas kiekybinis nematomumas (angl. *quantitative invisibility*). Taško kiekybinis nematomumas yra projektuojant tą tašką, dengiančių taškų skaičius. Kitaip sakant, jis parodo, kiek taškų, linijų ar plokštumų tą tašką dengia (1.4 pav.). Linijos kiekybiniui nematomumui esant lygiui 0, ji yra piešiama ant ekrano [App67].

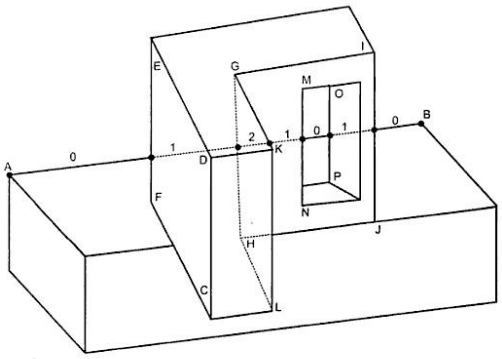



1.4 pav. Kiekybinis nematomumas

Algoritmas:
Šiam algoritmui kūnų ribose išskiriama kontūrinių briaunų aibė. 1.5 pav. esančios figūros kontūrinių briaunų aibė gaunama iš laužtės ABCIJDEKLG. Kiekviena briauna iš aibės yra suskaidoma į dalis per tuos taškus, kuriuose projektuojant į vaizdo plokštumą ji yra uždengiama kokia nors kita briauna iš tos aibės, kuri eina briaunos dengimo taške arčiau vaizdo plokštumos. Taip yra gaunama aibė briaunų, kurių kiekviena turės vieną kiekybinį nematomumą.




1.5 pav. Briaunainio kontūras



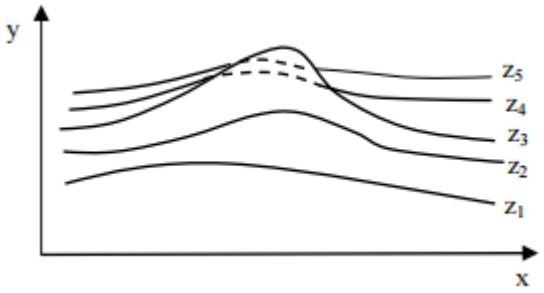

1.6 pav. Kiekybinio nematomumo pavyzdys

**Algoritmo sudėtingumas:** $O(\sqrt{n})$, kai n - bendras sienų skaičius. Patikrinimų skaičių galima gerokai sumažinti, jeigu naudojamasi vaizdo plokštumos suskaidymu.

### 1.1.3. Plaukiojančio horizonto metodas

Plaukiojančio horizonto metodas (angl. *Floating Horizon algorithm*) dažniausiai yra naudojamas paviršių, kuriuos galima išreikšti per funkciją $F(x, y, z) = 0$, nematomoms kontūrinėms linijoms lygiagrečiose projekcijose pašalinti. Jo pagrindinė idėja yra paversti trimatį uždavinį į dvimatį, kertant pirminį paviršių kertančiųjų lygiagrečiųjų plokštumų su pastoviomis x, y arba z koordinatėmis [Hed].

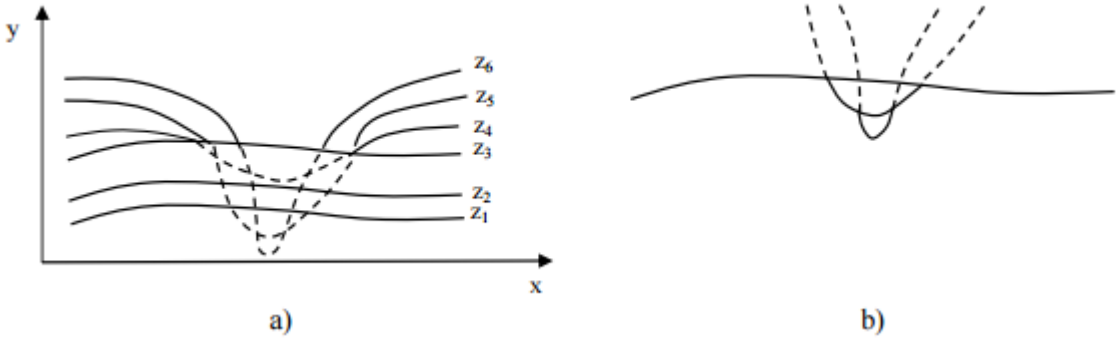
Algoritmas:
Algoritmas yra formuluojamas taip: Jeigu kurioje nors x reikšmėje y koordinatės reikšmė kreivėje yra didesnė nei y reikšmė kurios nors prieš tai buvusios kreivės toje pačioje x reikšmėje, tada kreivė yra matoma, kitu atveju kreivė yra nematoma.




1.7 pav. Kreivių projekcijos į plokštumą z=0

Nematomi segmentai pavaizduoti brūkšnine linija 1.7 pav. Šis algoritmas yra gan lengvai įgyvendinamas, norint laikyti visas maksimalias y reikšmes galima naudoti masyvą, kurio dydis priklausys nuo visų tikrinamų x reikšmių.

Šio algoritmo problema atsiranda, kai kuri nors kreivė yra žemiau pačios pirmos kreivės. Kadangi tos kreivės y reikšmė bus mažesnė nei prieš tai buvusios - ji nebus rodoma. Šiai problemai išspręsti yra naudojamas antras masyvas, kuris veikia atvirkščiai nuo pirmojo. Yra laikomos minimalios y reikšmės.




1.8 pav. Probleminė situacija eilinei kreivei atsiradus žemiau pačios pirmosios kreivės (a) ir algoritmo modifikacija patikslintas sprendimas (b)

**Algoritmo pranašumai:** Šis metodas tinka visiems išreikštinėmis dviejų kintamųjų funkcijomis aprašytiems paviršiams ir gerai išnaudoja briaunų koherentiškumą.

**Algoritmo trūkumai:** Jeigu funkcija turi labai aštrius lūžius, tai pateiktas algoritmas gali duoti nekorektiškus rezultatus.

## 1.2. Nematomų plokštumų nustatymas

Nepermatomi objektai erdvėje gali pilnai ar dalinai užstoti kitus objektus, kurie yra toliau nuo žiūrimojo taško. Kompiuterinėje grafikoje, šiuos nematomus paviršius turime pašalinti, kad gautumėme realistišką vaizdą. Ši problema yra vadinama nematomų plokštumų šalinimu (angl. *hidden surface removal*).

### 1.2.1. Z-buferio metodas

Vienas dažniausiai naudojamų metodų nematomų plokštumų nustatymui yra vadinamas Z-buferio metodu (Z-buffering). Šio metodo idėja yra testuoti kiekvieno paviršiaus nutolimą nuo žiūrimojo taško, kad pavyktų nustatyti artimiausią paviršių. Z-buferio metodas yra taikomas kiekvienam paviršiaus pikseliui. Jeigu [x, y] koordinatėse randamas mažesnį gylį turintis pikselis nei prieš tai buvęs, tada [x, y] koordinatėse yra piešiamas ta mažesnį gylį turinčio pikselio spalva [FN10] [Tut].

Kad galėtumėm perrašyti tolimesnius daugiakampius artimesniais, reikalingi du buferiai, kurie laikytų einamąsias ekrano pikselių spalvas ir žinotų kokio gylio pikselis buvo kurioje nors koordinatėje. Šie buferiai atitinkamai yra vadinami vaizdo atnaujinimo buferiu (angl. *frame buffer*) ir gylio buferiu (angl. *depth buffer*). Gylio buferyje dažniausiai z koordinatės yra normalizuojamos iki [0; 1] intervalo, kur 0 reikštų, jog gylis yra ant tolimosios atkirtimo plokštumos (angl. *far plane*, žiūrėti 1.3.1 skyrių), o 1 reikštų, kad gylis yra ant gretimosios atkirtimo plokštumos (angl. *near plane*).

Z-buferiui implementuoti reikalinga nemažai atminties. 1920x1080 rezoliucijai palaikyti reikalinga 1920 x 1080 x 32 bitai atminties vaizdo atnaujinimo buferiui ir tiek pat gylio buferiui. Tai yra virš 2 milijonų pikselių ekrane ir vos ne po 8 megabaitai atminties vienam buferiui. Dauguma šiuolaikinių vaizdo plokščių realizuotą z-buferį pačiame aparate, dažnai ir su aparatine „rastrizacija“ (vaizdo transformavimą iš koordinatinio pateikimo į rastrinį). Norint sumažinti atminties sąnaudas, galima uždavinį spręsti naudojant vienos eilutės dydžio Z-buferį. Tiriamieji paviršiai analizuojami po vieną nuskaitymo eilutę, rezultatai išsaugomi ir tada tiriama sekanti nuskaitymo eilutė.

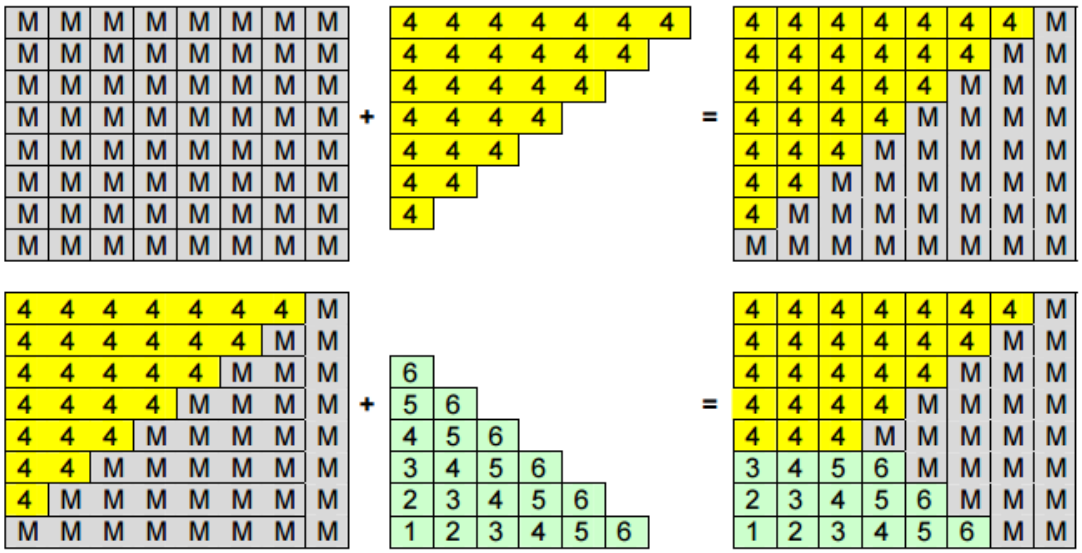
Gylio buferio algoritmo realizacijos schema:

**Pirmasis žingsnis.** Gylio buferio reikšmės nustatomos į mažiausią reikšmę – 0. Vaizdo atnaujinimo buferio reikšmės nustatomos į fono spalvą.

```
GylioBuferis[x, y] = 0
VaizdoAtnaujinimoBuferis[x, y] = FonoSpalva
```

**Antrasis žingsnis.** Kiekvienas daugiakampis yra apdorojamas paeiliui. Kiekvienam atvaizduojamam (x, y) daugiakampio pikselio pozicijai apskaičiuojamas gylis Z.

```
Jeigu Z > GylioBuferis[x, y]
      GylioBuferis[x, y] = Z
      VaizdoAtnaujinimoBuferis[x, y] = DaugiakampioPaviršiausSpalva[x, y]
```



1.9 pav. Z-bufferio (gylio buferio) pavyzdys nenormalizuotose koordinatėse

**Algoritmo pranašumai:** Paprastas, efektyvus. Lengvai realizuojamas aparatūrinėmis priemonėmis. Daugiakampių nagrinėjimo tvarka neturi įtakos galutiniam rezultatui. Šešėliavimą galima atlikti proceso pabaigoje

**Algoritmo trūkumai:** Reikia daug atminties, realizacijai reikalingi 2 buferiai – gylio ir vaizdo atnaujinimo. Galimos kvantavimo paklaidos. Sudėtinga analizuoti skaidrius daugiakampius. Skaitymo, redagavimo, įrašymo operacijoms vidiniuose cikluose atlikti reikalinga sparti atmintinė.

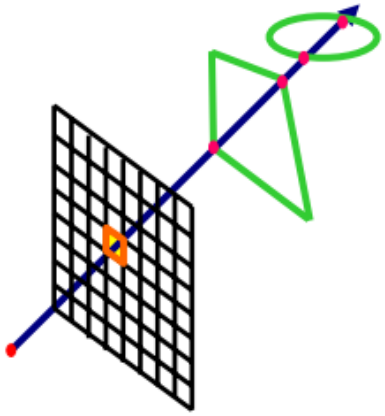
**Algoritmo sudėtingumas:** $O(n)$, kur n – bendrasis sienų skaičius.

### 1.2.2. Spindulių trasavimo metodas

Gamtoje, šviesos šaltinis skleidžia spindulius, kurie keliauja erdvėje iki kol pasiekia paviršių, kuris jį sustabdo. Atsitrenkus spinduliui į paviršių įvyksta 3 dalykai: šviesos sugėrimas, atspindėjimas ir šviesos lūžimas. Paviršius šviesos spindulį gali atspindėti į vieną ar kelias puses. Jis taip pat gali sugerti dalį šviesos, taip sumažindamas šviesos intensyvumą. Jei paviršius yra nors kiek permatomas, dalis šviesos lūžta į patį paviršių galimai apšviečiant paviršius esančius už jo pačio, ar pakeičiant šviesos spalvą [Mac15] [Van07].

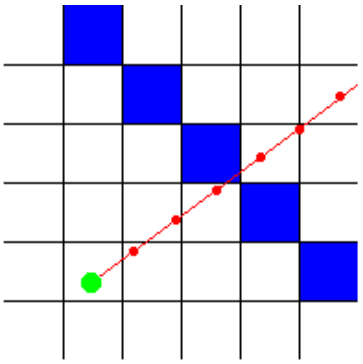
Spindulių trasavimo (angl. *ray tracing*) idėją pristatė Rene Descartes 1637 metais, kurio principas būtų kaip šviesos spinduliai veikia pasaulyje. Nematomų plokštumų nustatymo problemai reikalinga yra tik matomumo nustatymo dalis (angl. *ray casting*), tad yra nepaisoma spindulių atspindžiai ir lūžiai.

Pirmąjį spindulių trasavimo algoritmą pristatė A. Appel 1968 metais. Šio algoritmo principas yra skleisti spindulius iš stebėtojo pozicijos (akies) per kiekvieną vaizdo plokštumos tašką (pikselį) ir rasti artimiausia objektą blokuojantį spindulio kelią. Šį algoritmą galima optimizuoti naudojant pavyzdžiui gaubiančiųjų apvalkalų (angl. *bounding volume*) metodą.

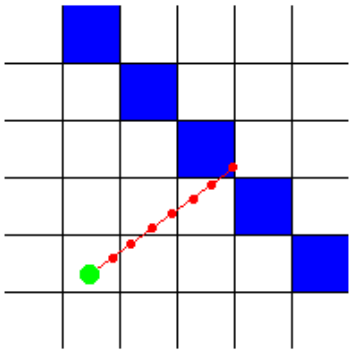


1.10 pav. Spindulių trasavimo metodas paviršių matomumui nustatyti

Algoritmas:
Norint rasti pirmąjį objektą sustabdantį spindulį, turime pradėti nuo kameros pozicijos (žiūrimojo taško) ir „šauti“ spindulį per kurį nors vaizdo plokštumos tašką (pikselį). Pradiniame taške yra patikrinama, ar taškas yra viduje objekto. Jeigu spindulys dar neatsitrenkė į objektą, pridedame prie spindulio ilgio kokią nors reikšmę, ir tikriname iš naujo. Vienas spindulys atlieka kelis patikrinimus, kas nustatytą ilgį. Žmogus gali akimirksniu pasakyti, kur spindulys atsitrenkia į sieną, tačiau kompiuteriui tai padaryti su viena formule yra neįmanoma, kadangi kompiuteris gali patikrinti tik baigtinį skaičių variantų. Jeigu spindulio tikrinimo ilgis bus per didelis, yra galimybė, kad tikrinimo taškai peršoks objektą ir mes jo neaptiksime (1.11 pav.). Esant kuo mažesniam tikrinimo ilgiui, bus mažesnė tikimybė praleisti objektą, tačiau atliekant daugiau patikrinimų algoritmas užtruks ilgiau.



1.11 pav. Spindulio trasavimo metodas neranda objekto tikrinimo ilgiui esant per dideliam



1.12 pav. Spindulio trasavimo metodas su mažu tikrinimo ilgiu

**Algoritmo pranašumai:** Paprastas. Galima suderinti matomumo nustatymą su pikselio spalvos apskaičiavimu.

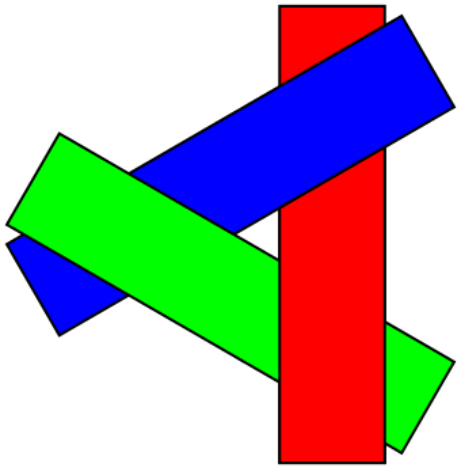
**Algoritmo trūkumai:** Nagrinėjami visi scenos pikseliai užuot nagrinėjus daugiakampių pikselius, nepanaudojamas koherentiškumas, smulki diskretizacija, lėtumas.

**Algoritmo sudėtingumas:** $O(C \log n)$, kur C – bendrasis pikselių skaičius ekrane, o n – bendrasis sienų skaičius. Yra metodų, kurių taikymo sąnaudos nepriklauso nuo objektų skaičiaus.

### 1.2.3. Tapytojo algoritmas

Dauguma dailininkų tapydami pirmiausia tapo tolimąsias piešinio dalis, tokias kaip peizažas, dangus. Tada ant viršaus yra nutapomi objektai, esantys arčiau stebėtojo. Iš to kilo tapytojo algoritmo (angl. *painter's algorithm*) idėja. Šis algoritmas yra vienas iš paprasčiausių nematomų paviršių problemos sprendimų kompiuterinėje grafikoje. Jo principas yra surūšiuotus pagal gylį objektus piešti vienas ant kito, taip sukuriant norimą vaizdą [Owe98b].

Kuriant 2D aplikacijas ar žaidimus be įsivaizduojamos trečios dimensijos (be z koordinatės) šis algoritmas teoriškai neturi jokių trūkumų. Pirmiausia yra nupiešiami paveikslėliai esantys scenos gale, tada nupiešiami kiti objektai vizualiai esantys arčiau matymo taško.
3D kompiuterinėje grafikoje šis algoritmas tampa šiek tiek sudėtingesnis ir atsiranda trūkumų. Vienas iš trūkumų yra objektų persidengimas. 1.13 pavyzdyje matomi trys stačiakampiai, tačiau kad ir kaip jie būtų surūšiuoti, šio vaizdo tapytojo algoritmas negalėtų atkurti. Šiai problemai išspręsti yra naudojamas paviršių skaidymas.



1.13 pav. Tapytojo algoritmo problema

Tapytojo algoritmo procesas:

**Pirmasis žingsnis.** Daugiakampiai surikiuojami z koordinatės mažėjimo tvarka nuo tolimiausio iki artimiausio.

**Antrasis žingsnis.** Patikrinti, ar daugiakampis yra priekyje kurio nors kito daugiakampio. Jei ne, jį galima piešti. Jeigu taip – abu daugiakampius suskaidyti, surūšiuoti ir pakartoti antrąjį žingsnį.

**Algoritmo pranašumai:** Labai paprastas ir lengvai įgyvendina objektų permatomumą.

**Algoritmo trūkumai:** Netinka karkasiniams paviršiams. Daugiakampių rūšiavimas ir daugiakampių skaidymas norint išspręsti persidengiančių objektų problemą.

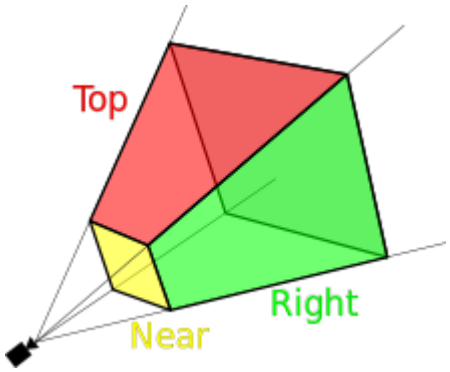
## 1.3. Optimizavimo metodai

Pritaikyti nematomų linijų ir paviršių algoritmus visam pasauliui užtruktų labai ilgai, kadangi objektų teoriškai gali būti be galo daug. Šį objektų skaičių galima stipriai sumažinti panaudojant paprastus, sparčiai vykdomus algoritmus, kurie nustato objektus nesančius kameros matymo lauke.

### 1.3.1. Vaizduojamojo tūrio atmetimas

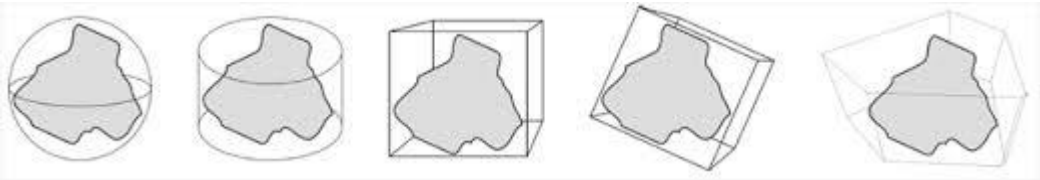
3D grafikoje geometrinių figūrų atmetimas tampa sudėtingesniu procesu. Tiek 3D pasaulyje, tiek realybėje asmuo gali matyti tik tai, kas yra jo matymo lauke, neįmanoma matyti objektų esančių už nugaros. Šis matymo laukas kompiuterinėje grafikoje yra vadinamas vaizduojamuojamu tūriu (angl. *view frustum*). Šią paprastą techniką naudoja kiekvienas rimtas 3D variklis. 3D grafikoje vaizduojamasis tūris yra erdvės regionas modeliuojamąjame pasaulyje, kuris gali atsirasti ekrane; tai yra menamosios kameros matymo laukas (1.14 pav.). Vaizduojamasis tūris yra dažniausiai gaunamas sutrumpinant matymo piramidės lygiagrečias plokštumas – matymo piramidė. Ši forma realybėje susidarytų, jeigu akis ar kamera turėtų stačiakampio peržiūras, kurias dažniausiai naudoja kompiuterinės grafikos [FN10] [Tut].

Pati šio regiono forma gali skirtis, priklausomai nuo to, kokio tipo kameros lęšis yra simuliuojamas. Dažniausiai tai yra nupjautinė piramidė. Plokštumos nukertančios piramidę yra vadinamos gretimąja plokštuma (angl. *near plane*) ir tolimąja plokštuma (angl. *far plane*). Objektai esantys arčiau kameros nei gretimoji plokštuma arba toliau nei tolimoji plokštuma yra nepiešiami.




1.14 pav. Vaizduojamasis tūris (angl. view frustum)

Vaizduojamojo tūrio atmetimu (angl. *view frustum culling*) yra vadinamas objektų, esančių visiškai už nupjautinės piramidės matymo ribų, atmetimo iš piešimo procesu. Piešti tokius objektus būtų laiko švaistymas, kadangi jie yra nematomi vartotojui. Šiam procesui dar labiau paspartinti dažniausiai objektai yra naudojamas gaubiančiųjų apvalkalų metodas, kuris leidžia atlikti matematinius skaičiavimus atmetimui ne su sudėtingu objektu, o jo apvalkalu, kuris būna daug paprastesnis (1.15 pav.).

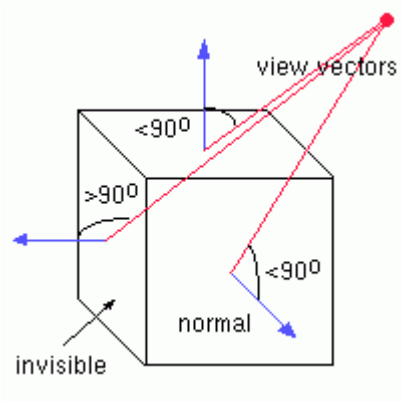


1.15 pav. Gaubiančiojo apvalkalo metodo pavyzdžiai

### 1.3.2. Galinių plokštumų atmetimas

Galinių plokštumų atmetimas (angl. *backface culling*) yra procesas, kuris atmeta visas plokštumas, kurios yra nematomos iš kurio nors taško erdvėje. Teoriškai, šiuo metodu, yra atmetami pusė objektų paviršių. Suskaičiuojama plokštumos normalės vektoriaus ir kameros žiūrėjimo krypties vektoriaus (angl. *view vector*) skaliarinė sandauga (angl. *dot product*). Jeigu skaliarinė sandauga yra teigiama, ta plokštuma yra matoma ir turėtų būti piešiama (1.16 pav.) [FN10] [Owe98a].

$$viewVector \cdot normal > 0$$

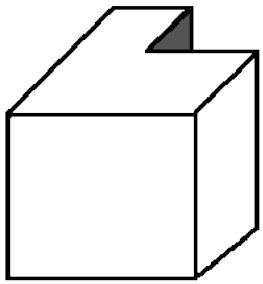



1.16 pav. Galinių plokštumų atmetimas

Matymo taškui esant tvarkingai sumodeliuoto objekto viduje, pats objektas bus nematomas, kadangi visos objekto plokštumų normalės bus nukreiptos nuo matymo taško. Naudojant neuždarus objektų modelius, galima pamatyti šios technikos veikimą nesant pačio objekto viduje (1.17 pav.).

1.17 pav. Neuždaro objekto galinių plokštumų atmetimas (kairėje be galinių plokštumų atmetimo, dešinėje - su)

Tačiau, šis metodas turi trūkumą. Galinių plokštumų atmetimas veikia teisingai tik iškiliesiems briaunainiams. Neiškilieji briaunainiai gali turėti plokštumų, kurios teoriškai yra matomos, tačiau praktiškai yra užstojamos to pačio objekto ir pilnai ar dalinai nematomos (1.18 pav.).



1.18 pav. Galinių plokštumų atmetimo problema

## 1.4. Skyriaus išvados

Susipažinus su keliais klasikiniais algoritmais, matome, kad nematomų linijų nustatymo algoritmai yra sudėtingesni, bei sunkiau įgyvendimai, nei nematomų paviršių nustatymo algoritmai. Literatūros apie juos taip pat yra žymiai sunkiau rasti.

Dengiamųjų paviršių nustatymo algoritmai gali būti panaudoti ir dengiamųjų linijų nustatymui, tačiau ne atvirkščiai. Jeigu paviršius yra nematomas, tai jį supančios briaunos taip pat bus nematomos. Šis teiginys leidžia paspartinti dengiamųjų linijų nustatymo algoritmus, pašalinant dengiamųjų paviršių briaunas prieš atliekant algoritmus.

Pritaikyti šiuos algoritmus visam pasauliui užtruktų labai ilgai, kadangi objektų teoriškai gali būti be galo daug. Šį objektų skaičių galima labai sumažinti panaudojant paprastus, sparčiai vykdomus algoritmus, kurie nustato objektus ar jų paviršius, kurie yra nematomi kamerai.

# 2. Tyrime naudojamos aplikacijos sprendimai bei implementacijos detalės

## 2.1. Grafikos atvaizdavimo sąsaja

Norint atvaizduoti objektus ekrane reikalinga pagalba iš kokios nors bibliotekos, gebančios palengvinti objektų atvaizdavimo darbą ir gebančios tiesiogiai bendrauti su vaizdo plokšte. Šiuo metu dvi populiariausios bibliotekos yra OpenGL ir DirectX.

Tyrimo metu nuspręsta naudoti OpenGL dėl įvairių platformų palaikymo ir patirties dirbant su šia biblioteka. Tyrimo metu nuspręsta naudoti Javos platformos objektinę kalbą – Groovy. Kadangi OpenGL yra parašyta C kalba, tenka naudoti OpenGL sąsajas Javos platformai – LWJGL.

### 2.1.1. VAO ir VBO

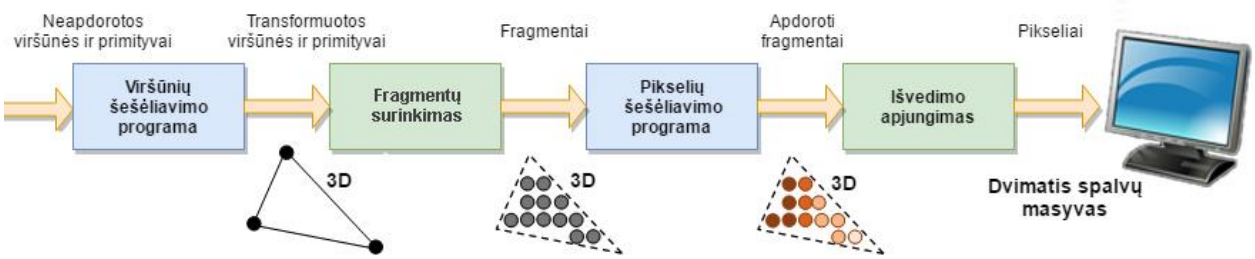
Su OpenGL 3.0 versija buvo pristatyta nauja piešimo strategija naudojant VAO (angl. *Vertex Array Objects*) ir VBO (angl. *Vertex Buffer Objects*). Duomenys yra saugomi specialiuose sąrašuose, kuriuos užkrauname į spartaus veikimo grafinę atmintį, tad nesikeičiantys duomenys gali būti ypač sparčiai perskaitomi ir apdorojami lygiagrečiai.

VAO (angl. *Vertex Array Objects*) – viršūnių masyvo objektai. Tai yra OpenGL objektas, skirtas saugoti viršūnes ir visus joms reikalingus duomenis. Šie duomenys yra saugomi per specialiai tam skirtus buferių objektus.

VBO (angl. *Vertex Buffer Objects*) – viršūnių buferio objektai. Tai yra buferio objektas skirtas saugoti viršūnėms reikalingus duomenis (tokius kaip tekstūrų koordinatės, normalių vektoriai). Atminties valdymo dalis sukurs buferių objektus geriausioje atminties vietoje pagal vartotojo užuominas (kam skirtas buferis ir ar jis žada keistis). Pagal tai bus optimizuojama buferio vieta ir jis bus saugomas vienoje iš 3 vietų: sistemos atmintyje, AGP atmintyje (angl. *Accelerated Graphics Port*, atmintis kuria dalijasi centrinis procesorius ir grafinis procesorius) arba grafinėje atmintyje.

### 2.1.2. OpenGL atvaizdavimo procesas

Norint atvaizduoti objektą su OpenGL, reikia sukompiliuoti ir užkrauti šešėliavimo programas į grafinę atmintį, užkrauti objekto geometrinį aprašymą į grafinę atmintį ir nusiųsti užklausą objekto atvaizdavimui. Objekto viršūnės ir primityvai yra nusiunčiami į viršūnių šešėliavimo programą. Ten jos yra apdorojamos, dažniausiai transformuojamos pagal kamerą ir apskaičiuojamos kitos reikšmės, reikalingos kiekvieno iš pikselių spalvoms apskaičiuoti. Visa transformuota informacija yra suskaidoma į fragmentus (išrenkami visi pikseliai esantys trikampyje) ir nusiunčiami į pikselių (fragmentų) šešėliavimo programą, kur kiekvienam taškui yra priskiriama spalva. Apskaičiavus visas objektų plokštumų taškų spalvas, jos yra apjungiamos, piešiant pikselius vienas ant kito (tapytojo algoritmu), pritaikomas Z-buferio metodas, atkreipiant dėmesį į dalinai matomus pikselius [Chu12].



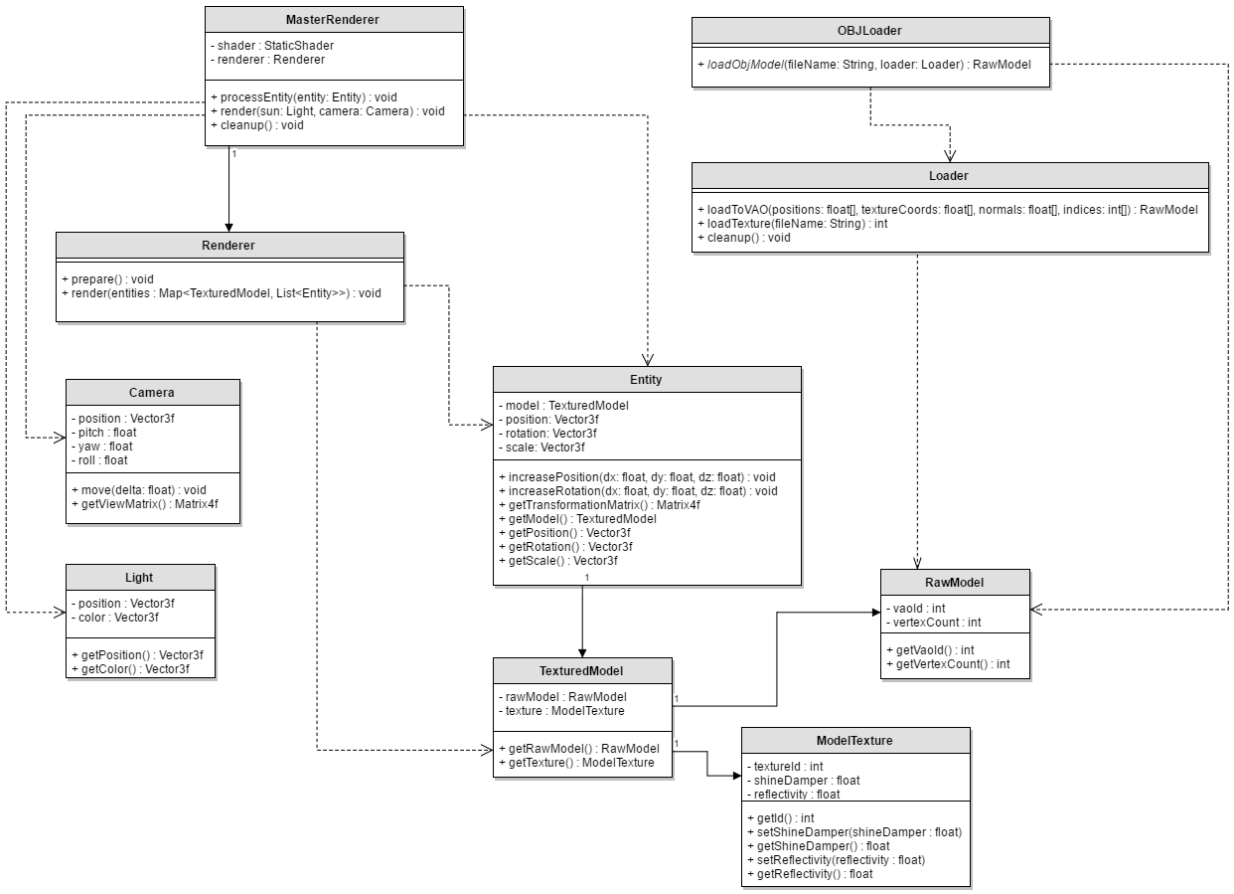

2.1 pav. OpenGL paprasčiausias atvaizdavimo procesas naudojant šešėliavimo programas

OpenGL visos šešėliavimo kalbos yra rašomos GLSL kalba. GLSL (OpenGL šešėliavimo kalba, angl. *OpenGL Shading Language*) yra aukšto lygio šešėliavimo kalba paremta C kalbos sintakse. Ją sukūrė OpenGL ARB (OpenGL Architecture Review Board) norėdami suteikti programuotojams tiesiogią grafinio atvaizdavimo proceso kontrolę. Anksčiau programuotojai galėjo naudotis ARB assemblerio kalba. GLSL versijos tobulėjo šalia OpenGL API. Nuo OpenGL 3.3 versijos buvo sutapatintos OpenGL ir GLSL versijos lengvesniam atsekamumui.

Naudojant GLSL dažniausiai yra aprašomos viršūnių ir pikselių (fragmentų) šešėliavimo programos (taip pat dar yra geometrijos ir mozaikos šešėliavimo programos). Kiekviena iš šešėliavimo programų turi input, output ir uniform (kintamuosius) parametrus. Input parametrai yra gaunami iš prieš tai esančios programos, output parametrai yra perduodami sekančiai programai, uniform parametrai yra pakraunami į programos atmintį tiesiai iš aplikacijos prieš kiekvieną piešimą.

## 2.2. Siūloma tyrimo aplikacijos sistemos architektūra

Tyrimui atlikti suprojektuota programinė įranga, leidžianti modeliuoti objektus trimatėje erdvėje. Siekiant pagyvinti scenas panaudojamos kelios pasaulio esybės kaip kamera ir šviesos šaltinis. Tyrimo metu pasiūlyta naudoti OBJ tipo failus objektų modelių geometrijos aprašymui. Aplikacija turės sugebėti nuskaityti šiuos failus ir juos paversti į atvaizduojamą objektą mūsų kode. Šiems objektams atvaizduoti yra reikalinga klasė, galinti užregistruoti visus kadre esančius objektus ir vienu iškvietimu viską nupiešti ekrane.
Kiekvienas objektas turi savo tinklelį (angl. *mesh*), tekstūrą, poziciją, posūkį, mastelį (angl. *scale*). Tačiau keli objektai gali naudoti tą patį tinklelį ir tą pačią arba kitą tekstūrą. Pozicija, posūkis ir mastelis yra unikalūs kiekvienam objektui.

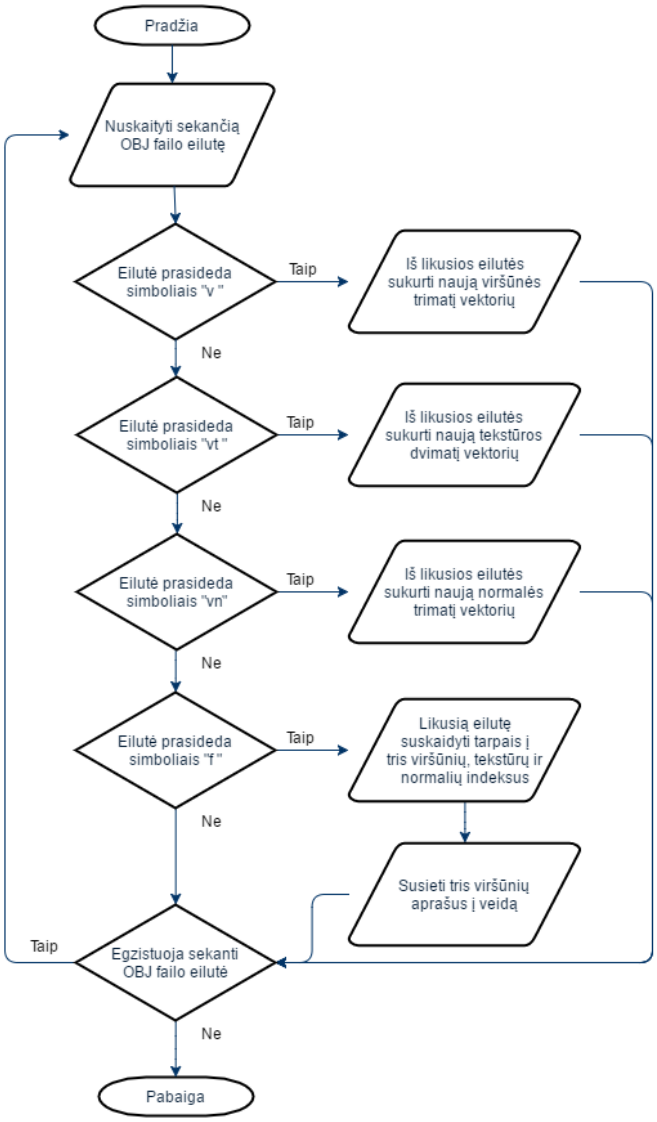



2.2 pav. Sistemos architektūros UML diagrama

Tyrimo metu pastebėta, kad tekstūras reikia aktyvuoti prieš pradedant atvaizdavimą. Jeigu kiekvienai esybės atvaizdavimo procesas būtų: aktyvuojama tekstūra, atvaizduojamas objektas, deaktyvuojama tekstūra, atvaizdavimas atliktų daug nereikalingų tekstūros aktyvacijų, atvaizduojant didelį sąrašą tą pačią tekstūrą naudojančių esybių. Atvaizdavimui optimizuoti visos esybės yra registruojamos į atvaizdavimo sąrašą ir surenkamos pagal jų naudojamas tekstūras. Šiuo būdu yra sutaupomas tekstūrų aktyvavimui praleidžiamas laikas.

### 2.2.1. OBJ failas

OBJ failas aprašo objekto geometriją. Išnagrinėjus šį failą galima atvaizduoti objektą trimatėje erdvėje pritaikius jam tekstūrą. Jo struktūrą sudaro objekto apibrėžimų sąrašai: viršūnių koordinatės, tekstūrų koordinatės, normalių vektoriai ir veidų/plokštumų aprašai. Kiekvienoje eilutėje pirmas žodis nusako ką ta eilutė aprašo, po jo tarpais atskirtos vektorių reikšmės. v – viršūnė, vt – tekstūros koordinatė, vn – normalės vektorius, f – veido aprašas. Jo pavertimo į atvaizduojamą objektą kode procesą galima pamatyti schemoje (2.3 pav.). OBJ failo pavyzdį galima rasti priede P.7.




**2.3 pav. OBJ failo nagrinėjimo schema**

### 2.2.2. Šešėliavimo programos

Tyrimo metu pasiūlyta scenose atvaizduoti objektus kartu su šviesos šaltiniu, apšviečiančiu objektą ir kamera, kurios pozicija būtų kontroliuojama. Kiekvienas objektas turės aplinkos apšvietimą (angl. *ambient lighting*) - veidas bus paryškintas arba patamsintas pagal šviesos vektoriaus ir to veido normalės vektoriaus skaliarinę sandaugą (angl. *dot product*). Taip pat, kiekvienas objektas turės veidrodinį apšvietimą (angl. *specular lighting*), kuris leis objektams suteikti blizgėjimo iliuziją. Šis blizgesys yra apskaičiuojamas pagal atspindimos šviesos nuo veido paviršiaus vektoriaus ir kameros matymo vektoriaus skaliarinės sandaugos. Objektai, atvaizduojami be šių apšvietimo apskaičiavimų, atrodytų labai paprasti [Bli77] [Dal13] [Ros06] [Sch93].

Kaip stipriai pasikeičia vaizdas naudojant apšvietimą galima pamatyti (2.4 pav.). Šių šešėliavimo programų GLSL implementacijos kodą galite rasti prieduose P.3 ir P.4.

2.4 pav. Kairysis objektas atvaizduojamas be apšvietimo, vidurinis - su aplinkos apšvietimu, dešinysis - su aplinkos ir veidrodiniu apšvietimu

# 3. Nematomų paviršių nustatymo algoritmų eksperimentinis tyrimas

## 3.1. Algoritmų panaudojimas aplikacijoje

OpenGL pati turi implementavus tam tikrus nematomų paviršių nustatymo algoritmus, kurie veikia ant grafinio procesoriaus ir leidžia juos įjungti iškvietus tam tikras funkcijas. OpenGL numatytai piešia objektus nurodyta eilės tvarka, tad nieko papildomai nenurodžius yra naudojamas tapytojo algoritmas [Khr15] [Ope].

**Galinių plokštumų atmetimas (backface culling)**

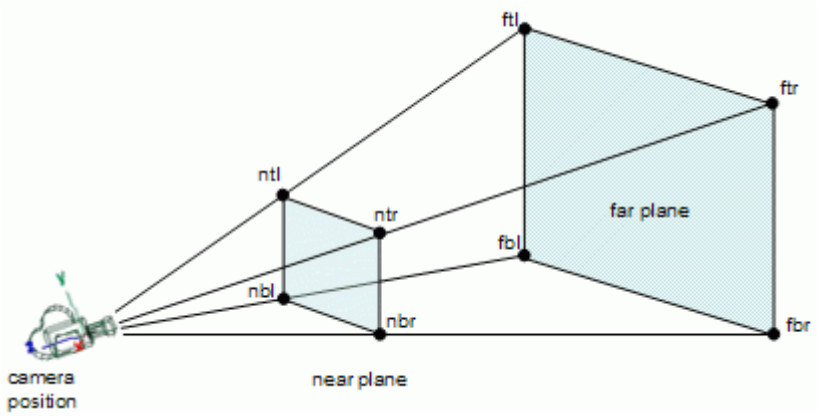
```
GL11.glEnable(GL11.GL_CULL_FACE)
GL11.glCullFace(GL11.GL_BACK)
```

**Z-buferio metodas**

```
GL11.glEnable(GL11.GL_DEPTH_TEST)
GL11.glClear(GL11.GL_DEPTH_BUFFER_BIT)
```

## 3.2. Vaizduojamojo tūrio algoritmo implementacija

OpenGL grafinėje kortoje automatiškai yra nepiešiami pikseliai, kurie yra už ekrano ribų, tačiau tai geriausiu atveju tik šiek tiek paspartina programos veikimą, kadangi visi scenos objektai vistiek praeina atvaizdavimo procesą (skaičiavimai šešėliavimo programose). To neužtenka norint paspartinti programos veikimą.

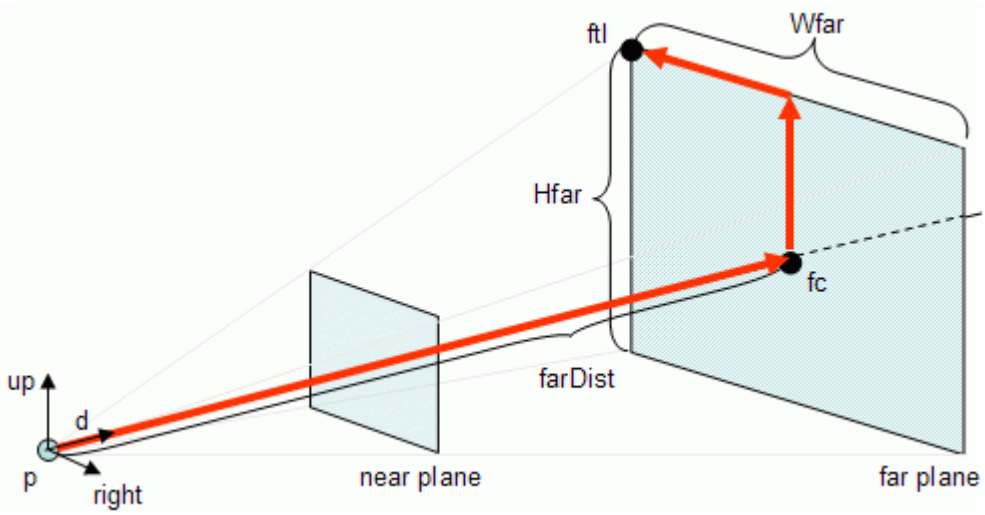



3.1 pav. Vaizduojamojo tūrio kampų apskaičiavimas

Norint naudoti šį metodą, pirmiausia reikia apskaičiuoti patį vaizduojamąjį tūrį (3.1 pav.). Jį apskaičiuoti reikia kelių parametrų:

- **p** – kameros pozicija
- **d** – normalizuotas kameros matymo vektorius
- **nearDist** – atstumas nuo kameros iki vaizduojamojo tūrio šalimosios sienos
- **farDist** – atstumas nuo kameros iki vaizduojamojo tūrio tolimosios sienos
- **Hnear, Wnear** – vaizduojamojo tūrio šalimosios sienos aukštis ir plotis
- **Hfar, Wfar** – vaizduojamojo tūrio tolimosios sienos aukštis ir plotis
- **up, right** – normalizuoti vektoriai, rodantys į viršų ir į dešinę nuo kameros

Turint šiuos parametrus galima apskaičiuoti visus 8 vaizduojamojo tūrio kampus (3.2 pav.). Šių kampų apskaičiavimo kodą galima rasti priede P.1 [Liga].




3.2 pav. Vaizduojamojo tūrio tolimosios kairės sienos kampo pozicijos apskaičiavimas

Turint visus 8 vaizduojamojo tūrio kampus galima apskaičiuoti sienų plokštumų formules. Plokštumos formulė yra aprašoma kaip:

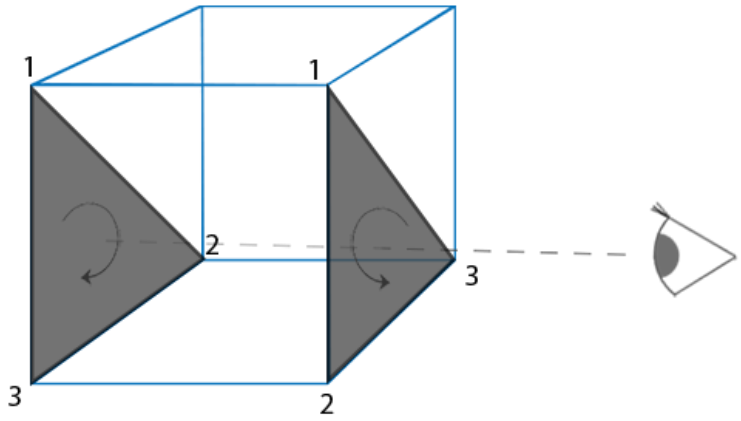
$$Ax + By + Cz + D = 0$$

Norint paversti sieną į plokštumos formulę, paimamos betkurios trys viršūnės laikantis prieš laikrodžio rodyklę tvarkos (3.3 pav.). Nuo vienos iš jų (α, tai yra trimatės viršūnės koordinatės)

suskaičiuojami vektoriai iki kitų dviejų viršūnių (β ir γ) ir jiems atliekame skaliarinę sandaugą, gauname trimatį vektorių *v*. Tai yra nenormalizuota, plokštumos formulė kertanti koordinačių erdvės tašką {0; 0; 0}. Apskaičiuojami visi nenormalizuoti plokštumos kintamieji:

$$A = v.x$$
$$B = v.y$$
$$C = v.z$$
$$D = -(v.x * \alpha.x + v.y * \alpha.y + v.z * \alpha.z)$$

Norint šiuos kintamuosius įstatyti į plokštumos formulę, juos reikia normalizuoti. Tai yra atliekama, padalinant kiekvieną kintamąjį iš didžiausios reikšmės tarp šių kintamųjų. Šio algoritmo implementaciją galima rasti priede P.2 [Daw] [Ligb].




**3.3 pav. Sienų viršūnės išdėstytos prieš laikrodžio rodyklę tvarką yra laikomos priekinėmis**

Šiame tyrime vaizduojamojo tūrio atmetimo sprendimui buvo tikrinama du šio metodo panaudojimo variantai. Pirmas – vaizduojamojo tūrio atmetimas geometrijos šešėliavimo programoje. Antras – vaizduojamojo tūrio atmetimas panaudojant gaubiančiojo apvalkalo metodą prieš paduodant objektus atvaizdavimui grafikos kortai.

## 3.3. Vaizduojamojo tūrio atmetimas geometrijos šešėliavimo programoje

Paprasčiausias OpenGL atvaizdavimo procesas naudojant šešėliavimo programas vyksta perduodant objekto viršūnes viršūnių šešėliavimo programai, atlikus skaičiavimus rezultatas yra perduodamas pikselių šešėliavimo programai, kur nustatoma pikselio spalva ir perduodamas rezultatas grafinei kortai.

Geometrijos šešėliavimo programa yra leidžiama tarp viršūnių ir pikselių šešėliavimo programų. Ši programa leidžia manipuliuoti objektų veidus. Geometrijos šešėliavimo programa gauna veido apskaičiuotas viršūnes ir jai priskirtas reikšmes ir rezultatui paduoda neribotą skaičių veidų. Pavyzdžiui, jeigu veidas būtų vienas pikselis erdvėje, geometrijos šešėliavimo programa galėtų kiekvienam iš šių pikselių sugeneruoti kubą, kurio centras būtų tame taške. Dažniausias šios programos panaudojimas yra kuriant dalelių (angl. *particle*) mechanizmą, tarkim fejerverkų sprogimo, dulkių efektą.

Šiame tyrime trikampiai yra laikomi objektų veidais. Taigi geometrijos šešėliavimo programa gauna 3 viršūnes ir rezultatui pikselių šešėliavimo programai paduoda kokį nors veidų skaičių.

Šiuo atveju, veidai yra perduodami neredaguoti – transformacijos, reikšmės nepakitusios. Tačiau, perduodami yra tik tie veidai, kurių nors viena viršūnė patenka į vaizduojamąjį tūrį.

Šio sprendimo būdo implementacija galima rasti priede P.5.

## 3.4. Vaizduojamojo tūrio atmetimas panaudojant gaubiančiojo apvalkalo metodą

Kadangi kiekvieno objekto veidai gali būti atmetami tik geometrijos šešėliavimo programoje, jų skaičiavimai vistiek yra atliekami viršūnių šešėliavimo programoje. Objekto modeliui turint daug viršūnių, šis procesas gali užtrukti nemažai laiko. Idealiausia būtų grafinei kortai net neliepti apskaičiuoti tų objektų, kurie nepatenka į vaizduojamąjį tūrį.

Taigi, kiekvienam modeliui priskirta paprasčiausia gaubianti sfera, kuri yra skirta tikrinimui ar yra už vaizduojamojo tūrio ribų. Tai reiškia, kad tikrinami yra patys objektai, o ne jų veidai. Tikrinamų detalių skaičius sumažėja n kartų, kur n yra to objekto veidų skaičius. Vienintelis minusas, kad šį tikrinimą reikia atlikti programos kode, o ne šešėliavimo programoje. Tai reiškia, kad skaičiavimas yra atliekamas ant procesoriaus, o ne grafinės kortos.

Norint pasiekti dar idealesnio varianto, galima atrinkti tuos objektus, kurių gaubiantysis apvalkalas kertasi su vaizduojamuoju tūriu ir jiems atlikti tikrinimą geometrijos šešėliavimo programoje. Tačiau, tyrimo metu buvo nuspręsta, kad tai daryti yra neverta, kadangi geometrijos šešėliavimo programa būtų kviečiama kiekvienam veidui, net ir tiems objektams, kurie yra vaizduojamojo tūrio viduje. Atkreipiant dėmesį, kiek dažniausiai objektų būna vaizduojamojo tūrio viduje ir kiek kertasi su juo, programos veikimas tikriausiai šiek tiek paspartėtų, o gal net ir iš vis nepaspartėtų ar net sulėtėtų dėl papildomos šešėliavimo programos kvietimo kiekvienam veidui.

Šio sprendimo būdo implementacija galima rasti priede P.6.

## 3.5. Testo procesas

Norint, kad testai vyktų keičiantis matomoms objekto plokštumoms, testuose visi objektai yra sukami aplink Y koordinatės ašį 60 laipsnių per sekundę greičiu. Kiekvienam testui skirta po 20 minučių su 2 minutėm apšilimo. Aplikacijos lango rezoliucija: 1280x720 (~920 tūkstančių pikselių).

Pirmo tyrimo metu lyginami testų variantai su skirtingais vaizduojamojo tūrio algoritmo įgyvendinimais (1 variantas – vaizduojamojo tūrio atmetimas geometrijos šešėliavimo programoje, 2 variantas – vaizduojamojo tūrio atmetimas panaudojant gaubiančiojo apvalkalo metodą). Tyrimo analizėje nagrinėjamas 2 testo variantas, kadangi jis kiekvienoje sudėtingesnėje scenoje veikė greičiau nei naudojant vaizduojamojo tūrio atmetimo algoritmą geometrijos šešėliavimo programoje. Pagrindinis tyrimų faktorius - aplikacijos veikimo spartumas pagal FPS (liet. *kadrai per sekundę*) naudojant skirtingus algoritmų junginius.

Naudojami algoritmų trumpiniai tyrimo lentelėse:

- **GP algoritmas** – Galinių plokštumų algoritmas
- **Z algoritmas** – Z buferio algoritmas
- **VT algoritmas** – Vaizduojamojo tūrio algoritmas

Naudojant šiuos trumpinius kartu, reiškia šių algoritmų panaudojimą kartu.

Pastebėjimas: Galinių plokštumų atmetimo algoritmas ir vaizduojamojo tūrio yra naudojami norint paspartinti aplikacijos veikimo laiką, kadangi jie pašalina iš piešimo ir taip teoriškai nematomus paviršius. Z buferio algoritmas sutvarko scenos atvaizdavimą kurioje nors koordinatėje piešdamas tik tą pikselį, kuris yra arčiausiai kameros. Tai reiškia, kad jis yra reikalingas norint teisingai atvaizduoti sceną; iš jo nėra tikimąsi paspartinti programos veikimo laiką.

**Kompiuterio specifikacija:**

- Procesorius: Intel® Core™ i5-6300U CPU @ 2.40GHz
- CPU: 4 cores
- RAM: 16GB
- Vaizdo korta: Intel HD Graphics 520
- Vaizdo atmintis: 256 MB
- Operacinė Sistema: Linux Ubuntu x86_64

## 3.6. Tyrimai ir gauti rezultatai

### 3.6.1. Scena „Kubas“

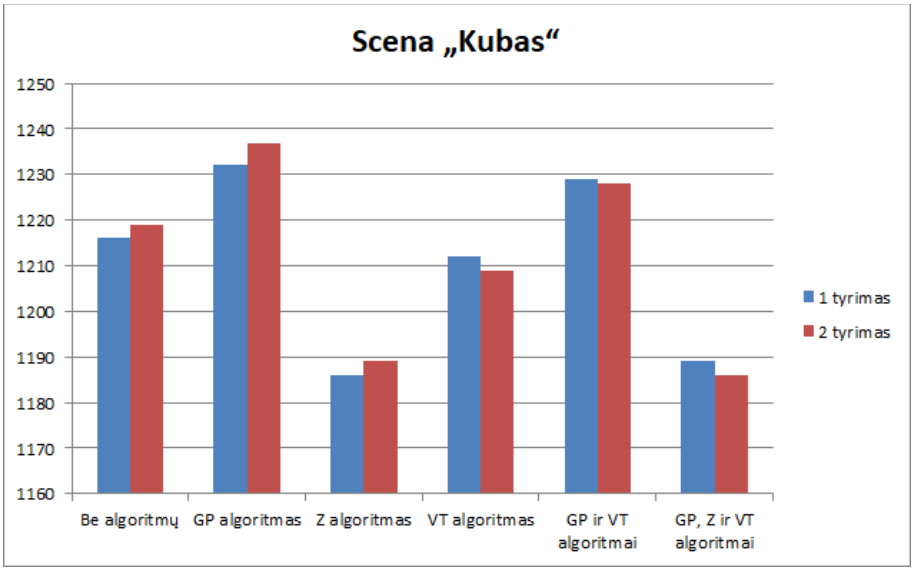
Šioje scenoje yra atvaizduojamas vienas paprastas kubas. Jis yra piešiamas koordinatėse {0; -1; -7}, tai reiškia, kad kubas yra horizontaliai centre ir vertikaliai šiek tiek žemiau centro. Dėl to, bet kuriuo metu, minimaliai matome 2 kubo sienas, maksimaliai – 3 kubo sienas. Tai reiškia, kad galinių plokštumų atmetimas dažniausiai pašalins 50% kubo sienų (3 sienos), o kubui atsisukus tiesiai į kamerą – 66% kubo sienų (4 sienos). Scenos nuotraukas galima rasti priede P.8.

**Kubo objekto geometrijos aprašymas:**

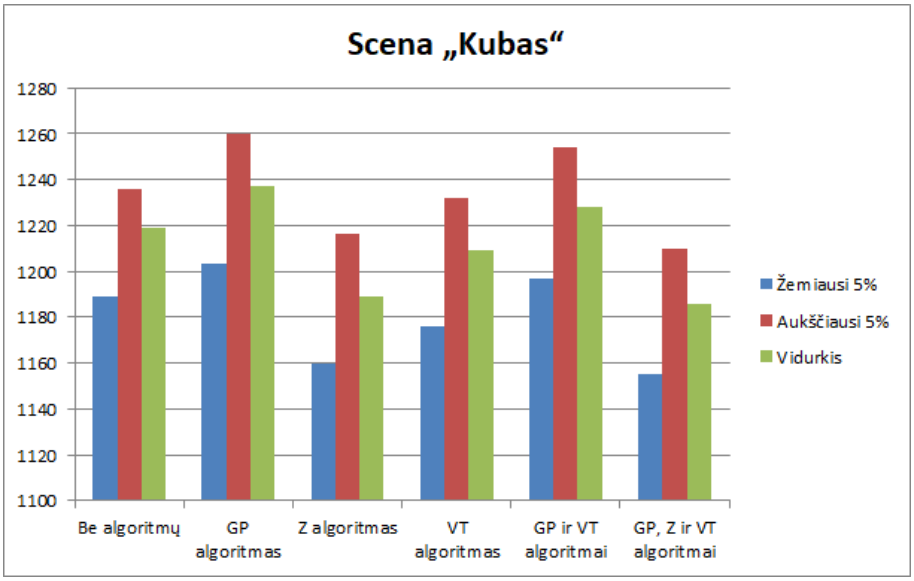
- Viršūnių skaičius: 8
- Veidų skaičius: 12

**Scenos aprašymas:**

- Atvaizduojami kubai: 1
- Viršūnių suma scenoje: 8
- Veidų suma scenoje: 12
- Tyrimo metu matomų kubų skaičius: 1




3.4 pav. Scenos „Kubas“ 1 ir 2 testo variantų tyrimo rezultatai




3.5 pav. Scenos „Kubas“ 2 testo varianto tyrimo rezultatai

**Tyrimo analizė:**

Iš pirmo tyrimo matome, kad ši scena veikia greičiau naudojant gaubiančiojo apvalkalo metodą, išskyrus tada, kai yra įjungtas vaizduojamojo tūrio (VT) algoritmas. Tiek galinių plokštumų (GP) metodas, tiek Z buferio metodas veikia greičiau pašalinus geometrijos šešėliavimo programą - atvaizdavimo procesas įvyksta šiek tiek greičiau.

Naudojant vaizduojamojo tūrio algoritmą, atvaizdavimas vyksta greičiau kai yra naudojama geometrijos šešėliavimo programa. Kadangi vienu metu daugiausiai galime matyti tik 3 sienas, geometrijos šešėliavimo programa veikia daugiausiai tik 3 kartus kiekvieną kadrą. Šis skaičiavimas įvyksta greičiau, nei naudojant gaubiančiojo apvalkalo metodą (kuris atlieka tą patį skaičiavimą, tik 1 kartą per kadrą) todėl, nes geometrijos šešėliavimo programa viską skaičiuoja ant grafinės kortos, kol gaubiantysis apvalkalas yra tikrinamas ant procesoriaus.

Iš antro tyrimo matome, kad naudojant galinių plokštumų atmetimo algoritmą, atvaizdavimo procesas trunka mažiau. Naudojant Z buferio algoritmą atvaizdavimo procesas užtrunka ilgiau,

tačiau to ir yra tikimąsi, kadangi tai nėra optimizavimo metodas. Vaizduojamojo tūrio algoritmas atvaizdavimo procesą sulėtina, tačiau to ir buvo tikimąsi, kadangi scenoje egzistuoja tik vienas objektas, kuris yra visada vaizduojamojo tūrio viduje.

Atvaizduojant objektus, kurie visada yra aplink kameros matymo centrą yra neverta naudoti vaizduojamojo tūrio algoritmą. Galinių plokštumų algoritmą vistiek verta naudoti, nes ir tokiai mažai apkrautai scenai.

### 3.6.2. Scena „Prekystaliai“

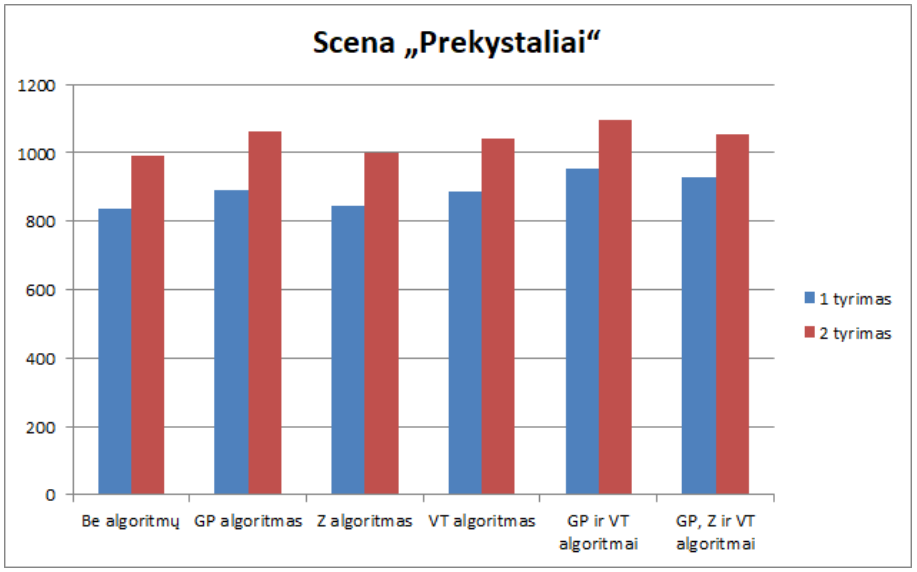
Šioje scenoje yra atvaizduojami keli prekystaliai. Jie yra išdėstyti aplink kamerą ir vienas prekystalis papildomai yra atvaizduojamas tiesiai prieš kamerą. Scenos nuotraukas galima rasti priede P.9.

**Prekystalio objekto geometrijos aprašymas:**

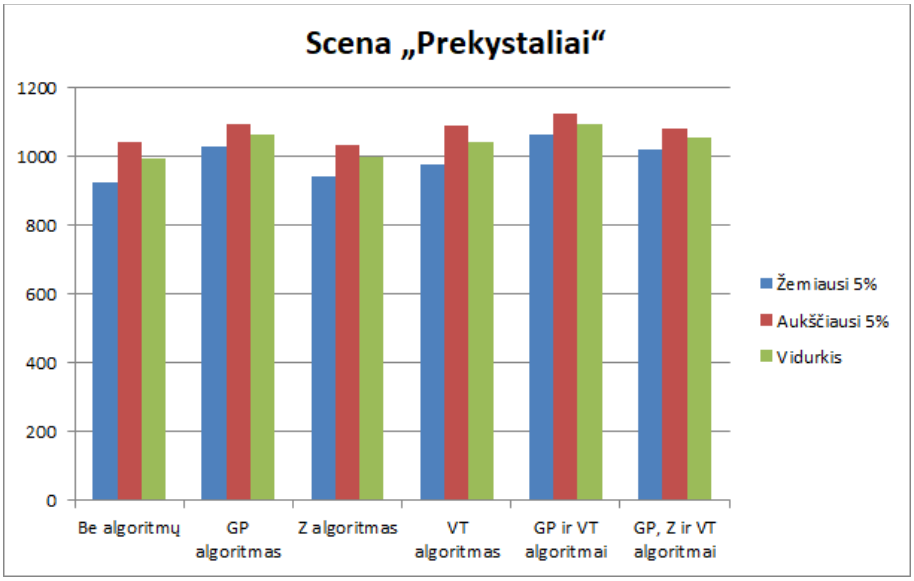
- Viršūnių skaičius: 509
- Veidų skaičius: 626

**Scenos aprašymas:**

- Atvaizduojami prekystaliai: 31
- Viršūnių suma scenoje: 15 779
- Veidų suma scenoje: 19 406
- Tyrimo metu matomų prekystalių skaičius: ~4.7




3.6 pav. Scenos „Prekystaliai“ 1 ir 2 testo variantų tyrimo rezultatai




3.7 pav. Scenos „Prekystaliai“ 2 testo varianto tyrimo rezultatai

**Tyrimo analizė:**

Iš pirmo tyrimo matome, kad ši scena visada veikia greičiau atliekant vaizduojamojo tūrio skaičiavimus procesoriuje ir nenaudojant geometrijos šešėliavimo programos. Pašalinus geometrijos šešėliavimo programą atvaizdavimo procesas įvyksta greičiau. Kompiuterio procesorius greičiau patikrina 31 sferą su vaizduojamuoju tūriu, nei grafinė korta patikrina 19.4 tūkstančius veidų.

Iš antro tyrimo matome, kad naudojant galinių plokštumų (GP) atmetimo algoritmą, atvaizdavimo procesas trunka mažiau. Naudojant Z buferio algoritmą atvaizdavimo procesas vidutiniškai trunka mažiau. Tačiau šis skirtumas labai mažas, galima netgi pastebėti, kad 5% greičiausių veikimų vistiek yra lėtesni nei nenaudojant Z buferio. Taip pat, lyginant du paskutinius tyrimus matome, kad Z buferis sulėtina procesą. Vaizduojamojo tūrio (VT) algoritmas atvaizdavimo procesą paspartino.

### 3.6.3. Scena „Medžiai“

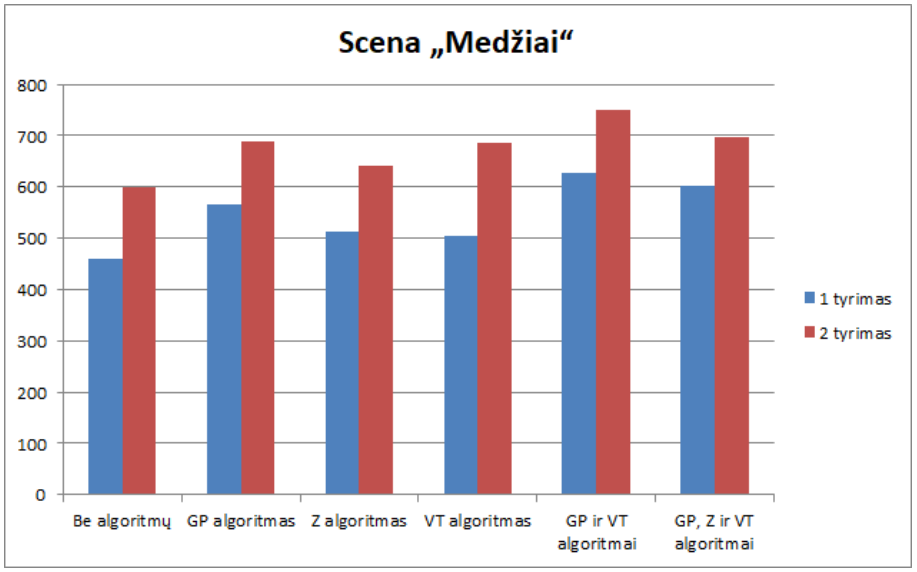
Šioje scenoje yra atvaizduojami nedaug veidų turintys medžiai. Jie yra išbarstyti visoje scenoje aplink kamerą, norint kad scena atrodytų kaip miškas. Scenos nuotraukas galima rasti priede P.10.

**Medžio objekto geometrijos aprašymas:**

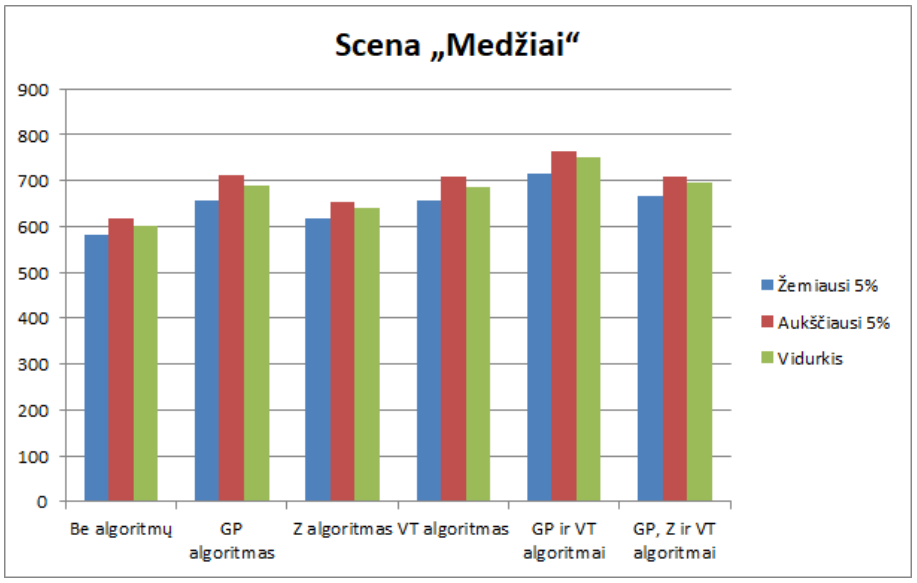
- Viršūnių skaičius: 224
- Veidų skaičius: 432

**Scenos aprašymas:**

- Atvaizduojami medžiai: 135
- Viršūnių suma scenoje: 30 240
- Veidų suma scenoje: 58 320
- Tyrimo metu matomų medžių skaičius: ~11.5




3.8 pav. Scenos „Medžiai“ 1 ir 2 testo variantų tyrimo rezultatai




3.9 pav. Scenos „Medžiai“ 2 testo varianto tyrimo rezultatai

**Tyrimo analizė:**

Iš pirmo tyrimo matome, kad ši scena visada veikia greičiau atliekant vaizduojamojo tūrio skaičiavimus procesoriuje ir nenaudojant geometrijos šešėliavimo programos. Pašalinus geometrijos šešėliavimo programą atvaizdavimo procesas įvyksta greičiau. Kompiuterio procesorius greičiau patikrina 135 sferas su vaizduojamuoju tūriu, nei grafinė korta patikrina 58.3 tūkstančius veidų.

Iš antro tyrimo matome, kad naudojant galinių plokštumų (GP) atmetimo algoritmą, atvaizdavimo procesas trunka mažiau. Naudojant Z buferio algoritmą atvaizdavimo procesas paspartėjo nei nenaudojant jokio algoritmo. Tačiau įjungus galinių plokštumų ir vaizduojamojo

tūrio (VT) algoritmus, Z buferis programos veikimą sulėtino. Vaizduojamojo tūrio algoritmas atvaizdavimo procesą paspartino.

### 3.6.4. Scena „Drakonai“

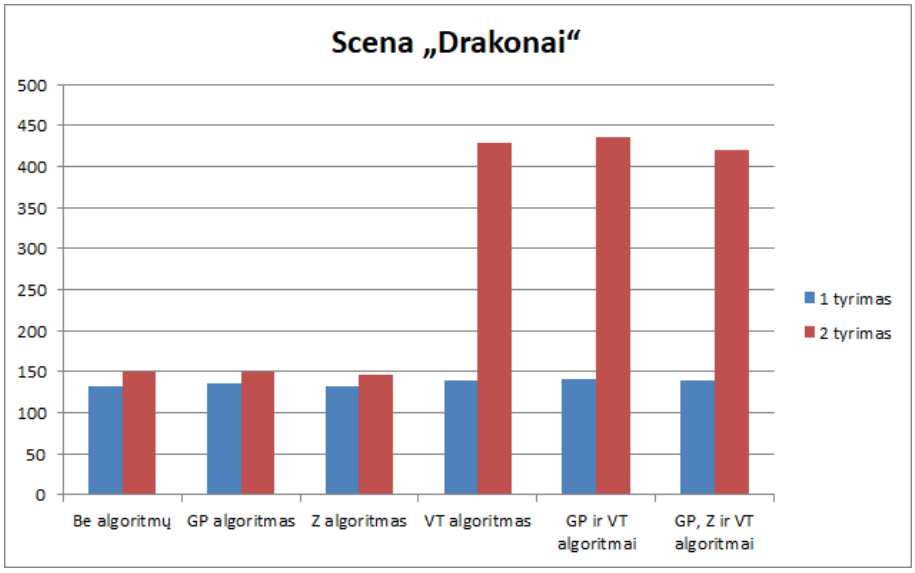
Šioje scenoje yra keturi labai detalizuoti drakonai. Jie visi yra išdėstyti 25 vienetų atstumu nuo kameros priekyje, kairėje, dešinėje ir už kameros. Scenos nuotraukas galima rasti priede P.11.

**Medžio objekto geometrijos aprašymas:**

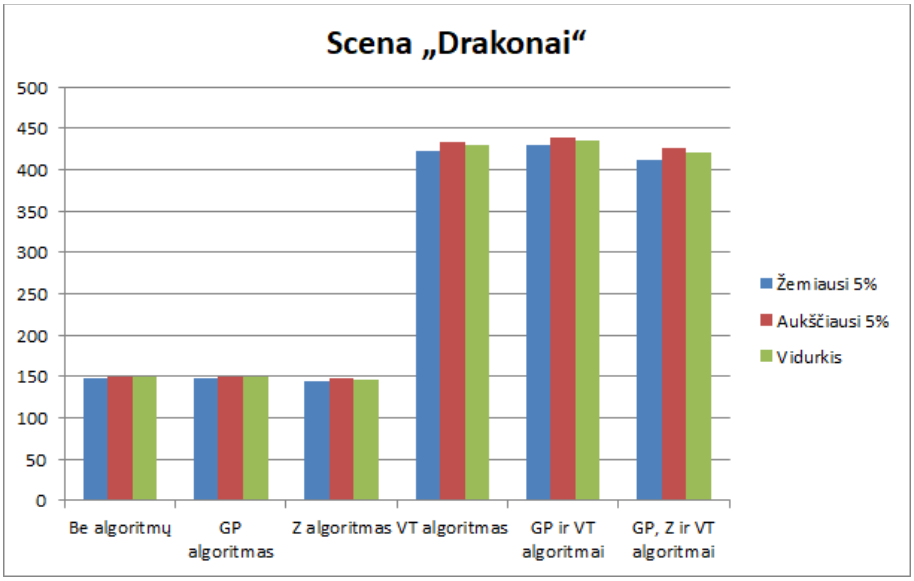
- Viršūnių skaičius: 50 000
- Veidų skaičius: 100 000

**Scenos aprašymas:**

- Atvaizduojami drakonai: 135
- Viršūnių suma scenoje: 200 000
- Veidų suma scenoje: 400 000
- Tyrimo metu matomų drakonų skaičius: 1




3.10 pav. Scenos „Drakonai“ 1 ir 2 testo variantų tyrimo rezultatai




3.11 pav. Scenos „Drakonai“ 2 testo varianto tyrimo rezultatai

**Tyrimo analizė:**

Iš pirmo tyrimo matome, kad ši scena visada veikia greičiau atliekant vaizduojamojo tūrio skaičiavimus procesoriuje ir nenaudojant geometrijos šešėliavimo programos. Pašalinus geometrijos šešėliavimo programą atvaizdavimo procesas įvyksta greičiau. Taip pat, galima pastebėti drastiškai pakitusį programos veikimo laiką įjungus vaizduojamojo tūrio (VT) algoritmą. Kompiuterio procesorius žymiai greičiau patikrina 4 sferas su vaizduojamuoju tūriu, nei grafinė korta patikrina 400 tūkstančių veidų.

Iš antro tyrimo matome, kad naudojant galinių plokštumų (GP) atmetimo algoritmą, atvaizdavimo proceso trukmė šiek tiek paspartėja (labiau matosi kartu įjungus vaizduojamojo tūrio algoritmą (VT). Naudojant Z buferio algoritmą atvaizdavimo procesas šiek tiek sulėtėjo. Vaizduojamojo tūrio algoritmas atvaizdavimo procesą labai stipriai paspartino. Į grafinę kortą apdorojimui nukeliaudavo 4 kartus mažiau objekto veidų.

# Išvados

Atlikus klasikinių nematomų linijų ir paviršių nustatymo algoritmų analitinę analizę pastebėta, kad nematomų linijų nustatymo algoritmai yra sudėtingesni, bei sunkiau įgyvendinami, nei nematomų paviršių nustatymo algoritmai. Dengiamųjų paviršių nustatymo algoritmai gali būti panaudoti ir dengiamųjų linijų nustatymui, tačiau ne atvirkščiai. Jeigu paviršius yra nematomas, tai jį supančios briaunos taip pat bus nematomos.

Išnagrinėjus šiuo metu populiariausias atvaizdavimo bibliotekas, tyrimo metu nuspręsta naudoti OpenGL biblioteką. Nuspręsta aplikaciją rašyti Groovy kalba su LWJGL OpenGL sąsaja. Sukūrus aplikaciją, gebančią atvaizduoti trimačius objektus ekrane, buvo sukurtos keliose scenos su skirtingais objektais ir atlikti jiems testai.

Suprojektavus ir naudojantis eksperimentinio tyrimo sistemos architektūra buvo sukurta aplikacija, gebanti atvaizduoti objektus trimatėje erdvėje, aktyvuoti/deaktyvuoti nematomų paviršių algoritmus, valdyti kamerą scenoje ir tirti programos veikimo laiką.

Tyrimui atlikti buvo sukurtos keturios scenos su įvairiais objektų modelių sudėtingumais. Kubas – paprasčiausia scena, kurioje atvaizduojamas vienas paprastas šešiasienis kubas turintis 12 plokštumų. Medžiai ir prekystaliai, vidutinio sudėtingumo scenos pasižyminčios nedaug viršūnių ir plokštumų turinčiais objektų modeliais, tačiau atvaizduojant nemažą kiekį objektų. Drakonai – sudėtinga scena, atvaizduojanti keturis didelio sudėtingumo objektus, kiekvienas turintis po 50 tūkstančių viršūnių ir 100 tūkstančių plokštumų.

Kiekvienai scenai atlikti po 2 tyrimų variantai. Kiekvienam tyrimui buvo atlikti 6 skirtingi testai su skirtingom įjungtų nematomų paviršių algoritmų kombinacijom. Pirmo tyrimo metu lyginami testų variantai su skirtingais vaizduojamojo tūrio algoritmo įgyvendinimais (1 variantas – vaizduojamojo tūrio atmetimas geometrijos šešėliavimo programoje, 2 variantas – vaizduojamojo tūrio atmetimas panaudojant gaubiančiojo apvalkalo metodą). Detaliau nagrinėjamas 2 testo variantas, kadangi jis kiekvienoje sudėtingesnėje scenoje veikė greičiau nei naudojant vaizduojamojo tūrio atmetimo algoritmą geometrijos šešėliavimo programoje. Pagrindinis tyrimų faktorius - aplikacijos veikimo spartumas pagal FPS (liet. *kadrai per sekundę*) naudojant skirtingus algoritmų junginius.

Tyrimų metu galinių plokštumų metodas visada paspartindavo programos veikimą. Teoriškai, jis pašalina apie 50% nematomų plokštumų, todėl atvaizduojant sudėtingesnius modelius, šis algoritmas tampa efektyvesnis. Šis algoritmas yra labai nesunkiai įgyvendinamas, o daugumoje vaizdo plokščių jis yra jau implementuotas, tai reikia jį tiesiog įjungti. Vaizduojamojo tūrio metodas su gaubiančiųjų apvalkalų metodu yra tikrai vertas dėmesio ir laiko implementuoti, ypač apkrautose scenose, kur yra daug objektų aplink kamerą. Jeigu visi objektai yra atvaizduojami tik prieš kamerą (dažniausiai taip būna 3D galvosūkių žaidimuose), šio metodo galima net nenaudoti. Kadangi, norint atvaizduoti sceną taisyklingai reikalingas koks nors už plokštumų gylius atsakingas metodas, tenka naudoti Z buferio metodą. Jis yra labai paprastas, veikia ant grafinio procesoriaus, OpenGL įjungiamas su pora kodo eilučių ir programuotojui pačiam nereikia gilintis į tai, kaip jis veikia.

# Summary

Nowadays, computer graphics are widely used starting from the simplest calculator or text editor and finishing with high-definition video game rendering. While it‘s not a common problem with the simpler usages, there might be drastic performance issues while rendering complex 3D scenes or video games, because of the huge amount of vertices and polygons graphics cards have to render. Usually, the games run on 60 FPS(frames per second) mode, so each scene has to be prepared and rendered in less than 0.016 second. With the massive number of primitives graphics card has to process, some measures have to be taken to reduce that number before each render call. This is where the hidden line and surface removal algorithms come to play.

Hidden line and surface removal algorithms determine which vertices, lines or faces are not visible from the camera viewport and are removed from the rendering process, thus considerably improving the application‘s performance. There are a number of hidden line and surface removal algorithms, so this paper focuses on testing application‘s performance changes by using 3 popular hidden surface removal algorithms: backface culling, view-frustum culling and Z-buffer methods.

For this testing a custom application is written using Groovy and LWJGL(lightweight java game library), which is a binding for OpenGL library. Four different scenes are constructed and they are tested against 2 different test versions, with the difference of view-frustum implementation. Each test version is run with 6 different modes giving each of them 2 minutes of warmup time and 20 minutes of testing time. During that time, current FPS value is logged each second, and after all the tests a report is generated showing all the FPS values during the test, the average FPS value, the maximum and minimum 5% of the FPS values.

After the tests it is confirmed, that it is almost always better to implement a CPU side view-frustum culling using bounding boxes on all objects. It is incredibly taxing to use it inside geometry shader, as it not only adds an additional layer the primitives must go through, but it tests each and every so far not removed face of all objects, which takes a lot of time and usually becomes counter-productive. However, since most objects are quite complex, they have a lot of faces, so having one bounding box or sphere around the object makes it faster to test against the view-frustum planes.

After the tests it is noted, that backface culling always improves the application‘s performance. In theory, it should remove about 50% of the objects faces which are facing away from the camera and since it is easy to activate, it should be used in every application.

GPU‘s do not have an inbuilt view-frustum algorithm, so the developer must implement it himself, however it is worth the time implementing it if the application renders a full scene with many objects outside the camera‘s viewport. If it is a rendering program, which renders only the objects in front of the camera, then the view-frustum culling should not be activated as it would only decrease the performance.

Z-buffer method is a depth fixing method, and should be used despite it‘s performance changes. It is implemented in the GPU, so it‘s easy to activate it and use it. In the tests the Z-buffer method did decrease the performance, but it was quite minimal and did not make any difference from the user‘s perspective.

# Šaltiniai


[App67] Arthur Appel. *The notion of quantitative invisibility and the machine rendering of solids*. International Business Machines Corporation, Yorktown Heights, New York, 1967

[Bli77] James F. Blinn. *Models of light reflection for computer synthesized pictures*. Univerity of Utah, Salt Lake City, USA, 1977.

[Chu12] Chua Hock-Chuan. *3D Graphics with OpenGL. Basic Theory*. Nanyang Technology University, Singapore 2012, [žiūrėta 2017-05-23] URL: https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

[Dal13] Tom Dalling. *Modern OpenGL 07 – More Lighting: Ambient, Specular, Attenuation, Gamma*. Tom Dalling blog, 2013, [žiūrėta 2017-05-23] URL: http://www.tomdalling.com/blog/modern-opengl/07-more-lighting-ambient-specular-attenuation-gamma/

[Daw] Paul Dawkins. Paul's Online Math Notes. Calculus III – Equations of Planes, [žiūrėta 2017-05-23] URL: http://tutorial.math.lamar.edu/Classes/CalcIII/EqnsOfPlanes.aspx

[FN10] Oskar Forsslund, Jan Nordberg. *Hidden Surface Removal and Hyper Z*. Bachelor of Science Thesis, Stockholm, Sweden, 2010, [žiūrėta 2017-05-23] URL: http://www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/rapport/forsslund_oskar_OCH_nordberg_jan_K10058.pdf

[Hed] Dr Shriram Hegde. *Visualisation Concepts. Visible Lines and Surfaces*. Applied Mechanics Department, [žiūrėta 2017-05-23] URL: http://web.iitd.ac.in/~hegde/cad/lecture/L36_visibility.pdf

[Khr15] Khronos. *Depth Test*. Khronos, OpenGL documentation, [žiūrėta 2017-05-23] URL: https://www.khronos.org/opengl/wiki/Depth_Test

[Liga] Lighthouse3d.com. *Tutorials » View Frustum Culling » Geometric Approach – Extracting the Planes*, [žiūrėta 2017-05-23] URL: http://www.lighthouse3d.com/tutorials/view-frustum-culling/geometric-approach-extracting-the-planes/

[Ligb] Lighthouse3d.com. *Tutorials » View Frustum Culling » Geometric Approach – Testing Points and Spheres*, [žiūrėta 2017-05-23] URL: http://www.lighthouse3d.com/tutorials/view-frustum-culling/geometric-approach-testing-points-and-spheres/

[Mac15] Jon Macey. *Ray-tracing and other Rendering Approaches*. National Center for Computer Animation webite. 2015, [žiūrėta 2017-05-23] URL: https://nccastaff.bournemouth.ac.uk/jmacey/CGF/slides/RayTracing4up.pdf

[Ope] OpenGL. *12 The Depth Buffer*. OpenGL archives, [žiūrėta 2017-05-23] URL: https://www.opengl.org/archives/resources/faq/technical/depthbuffer.htm

[Owe98a] G. Scott Owen. *Back Face Removal*. The Association for Computing Machinery. 1998, [žiūrėta 2017-05-23] URL: https://www.siggraph.org/education/materials/HyperGraph/scanline/visibility/backface.htm

[Owe98b] G. Scott Owen. *Visible Surface Determination: Painter's Algorithm*. The Association for Computing Machinery. 1998, [žiūrėta 2017-05-23] URL: https://www.siggraph.org/education/materials/HyperGraph/scanline/visibility/painter.htm

[Rob63] Lawrence Gilman Roberts. *Machine Perception of Three-Dimensional Solids*. Massachusetts Institute of Technology, 1963, p. 62-69.

[Ros06] Randi J. Rost. *OpenGL Shading Language, Second Edition*. Addison Wesley Professional, 2006

[Sch93] Christophe Schlick. *A Customizable Reflectance Model for Everyday Rendering*. Fourth Eurographics Workshop on Rendering, Paris, France, 1993.

[Tut] Tutorials Point. *Visible Surface Detection*, [žiūrėta 2017-05-23] URL: http://www.tutorialspoint.com/computer_graphics/visible_surface_detection.htm

[Van07] Lode Vandevenne. *Lode's Computer Graphics Tutorial*. 2007, [žiūrėta 2017-05-23] URL: http://lodev.org/cgtutor/raycasting.html

# Priedai

## P.1. Vaizduojamojo tūrio kampų apskaičiavimo algoritmas

```
static List<Vector3f> calculateFrustumPlanesPoints(Camera camera) {
    Vector3f direction = camera.viewDirection
    Vector3f position = camera.position
    Vector3f up = camera.up
    Vector3f right = camera.right

    float nearDist = Renderer.NEAR_PLANE
    float farDist = Renderer.FAR_PLANE

    float Hnear = 4f
    float Wnear = 7f
    float Hfar = 394f
    float Wfar = 700f

    use(Vectors) {
        Vector3f farCenter = position + direction * farDist
        Vector3f nearCenter = position + direction * nearDist
        Vector3f ftl, ftr, fbl, fbr, ntl, ntr, nbl, nbr

        ftl = farCenter + (up * (Hfar / 2)) - (right * (Wfar / 2))
        ftr = farCenter + (up * (Hfar / 2)) + (right * (Wfar / 2))
        fbl = farCenter - (up * (Hfar / 2)) - (right * (Wfar / 2))
        fbr = farCenter - (up * (Hfar / 2)) + (right * (Wfar / 2))

        ntl = nearCenter + (up * (Hnear / 2)) - (right * (Wnear / 2))
        ntr = nearCenter + (up * (Hnear / 2)) + (right * (Wnear / 2))
        nbl = nearCenter - (up * (Hnear / 2)) - (right * (Wnear / 2))
        nbr = nearCenter - (up * (Hnear / 2)) + (right * (Wnear / 2))

        return [ftl, ftr, fbl, fbr, ntl, ntr, nbl, nbr]
    }
}
```

## P.2. Vaizduojamojo tūrio sienos pavertimas į plokštumos formulę

```
static List<Vector4f> extractPlanes(List<FrustumPlane> frustumPlanes) {
    use(Vectors) {
        frustumPlanes.collect { plane ->
            def vec1 = plane.b - plane.a
            def vec2 = plane.c - plane.a

            def crossProduct = Vector3f.cross(vec1, vec2, null)
            def vec = new Vector4f(
                crossProduct.x,
                crossProduct.y,
                crossProduct.z,
                - (crossProduct.x * plane.a.x + crossProduct.y * plane.a.y +
crossProduct.z * plane.a.z) as float
            )
            vec.normalise(null)
        }
    }
}
```

## P.3. Viršūnių šešėliavimo programos kodas

```
#version 150

in vec3 position;
in vec2 textureCoords;
in vec3 normal;

out VertexData {
  vec3 pass_position;
  vec2 pass_textureCoords;
  vec3 surfaceNormal;
  vec3 toLightVector;
  vec3 toCameraVector;
} VertexOut;

uniform mat4 transformationMatrix;
uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;

uniform vec3 lightPosition;

void main(void) {
    vec4 worldPosition = transformationMatrix * vec4(position, 1.0);
    gl_Position = projectionMatrix * viewMatrix * worldPosition;
    VertexOut.pass_textureCoords = textureCoords;

    VertexOut.surfaceNormal = (transformationMatrix * vec4(normal, 0.0)).xyz;
    VertexOut.toLightVector = lightPosition - worldPosition.xyz;

    VertexOut.toCameraVector = (inverse(viewMatrix) * vec4(0.0, 0.0, 0.0,
1.0)).xyz - worldPosition.xyz;
    VertexOut.pass_position = worldPosition.xyz;
}
```

## P.4. Pikselių šešėliavimo programos kodas

```
#version 150

in VertexData {
  vec3 pass_position;
  vec2 pass_textureCoords;
  vec3 surfaceNormal;
  vec3 toLightVector;
  vec3 toCameraVector;
} VertexIn;

out vec4 out_Color;

uniform sampler2D textureSampler;
uniform vec3 lightColor;
uniform float shineDamper;
uniform float reflectivity;

void main(void) {
    vec3 unitNormal = normalize(VertexIn.surfaceNormal);
    vec3 unitLightVector = normalize(VertexIn.toLightVector);

    float nDot1 = dot(unitNormal, unitLightVector);
    float brightness = max(nDot1, 0.2);
    vec3 diffuse = brightness * lightColor;

    vec3 unitCameraVector = normalize(VertexIn.toCameraVector);
    vec3 lightDirection = -unitLightVector;
    vec3 reflectedLightDirection = reflect(lightDirection, unitNormal);

    float specularFactor = dot(reflectedLightDirection, unitCameraVector);
    specularFactor = max(specularFactor, 0.0);
    float dampedFactor = pow(specularFactor, shineDamper);
    vec3 finalSpecular = dampedFactor * reflectivity * lightColor;

    out_Color = vec4(diffuse, 1.0) * texture(textureSampler,
VertexIn.pass_textureCoords) + vec4(finalSpecular, 1.0);
}
```

## P.5. Geometrijos šešėliavimo programos kodas

```
#version 150

layout (triangles) in;
layout (triangle_strip, max_vertices = 3) out;

in VertexData {
  vec3 pass_position;
  vec2 pass_textureCoords;
  vec3 surfaceNormal;
  vec3 toLightVector;
  vec3 toCameraVector;
} VertexIn[3];

out VertexData {
  vec2 pass_textureCoords;
  vec3 surfaceNormal;
  vec3 toLightVector;
  vec3 toCameraVector;
} VertexOut;

uniform vec4 frustumPlanes[6]; // right left bottom top far near
uniform bool useFrustumCulling;

float distanceFromPlane(in vec4 plane, in vec3 p) {
    return dot(plane.xyz, p) + plane.w;
}

bool pointInFrustum(in vec3 p) {
    for (int i = 0; i < 6; i++) {
        vec4 plane = frustumPlanes[i];
        if (distanceFromPlane(plane, p) <= 0) {
            return false;
        }
    }
    return true;
}

bool foundVertexInsideFrustum() {
    for (int i = 0; i < gl_in.length(); i++) {
        vec3 position = VertexIn[i].pass_position;
        if (pointInFrustum(position)) {
            return true;
        }
    }
    return false;
}

void main() {
    if (useFrustumCulling && !foundVertexInsideFrustum()) {
      return;
    }

    for (int i = 0; i < gl_in.length(); i++) {
        gl_Position = gl_in[i].gl_Position;
        VertexOut.pass_textureCoords = VertexIn[i].pass_textureCoords;
        VertexOut.surfaceNormal = VertexIn[i].surfaceNormal;
        VertexOut.toLightVector = VertexIn[i].toLightVector;
        VertexOut.toCameraVector = VertexIn[i].toCameraVector;

        EmitVertex();
    }
    EndPrimitive();
}
```

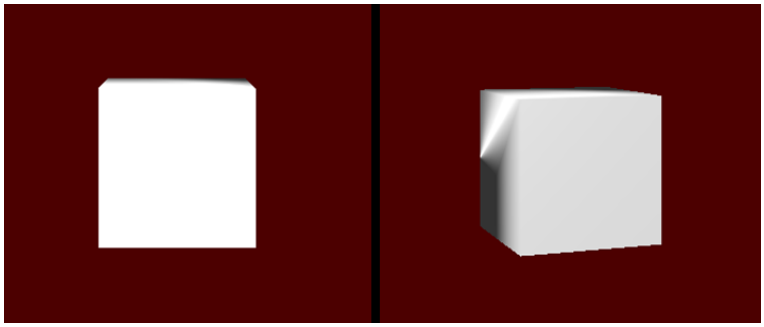
## P.6. Vaizduojamojo tūrio atmetimas panaudojant gaubiančiojo apvalkalo metodą

```
private static float distanceFromPlane(Vector4f plane, Vector3f point) {
    // dot product + plane distance from origin
    (plane.x * point.x + plane.y * point.y + plane.z * point.z + plane.w) as float
}

static boolean sphereInsideFrustum(Vector3f spherePosition, float sphereRadius,
List<Vector4f> frustumPlanes) {

    for (def plane: frustumPlanes) {
        float distance = distanceFromPlane(plane, spherePosition)
        if (distance < -sphereRadius) {
            return false
        }
    }
    return true
}
```

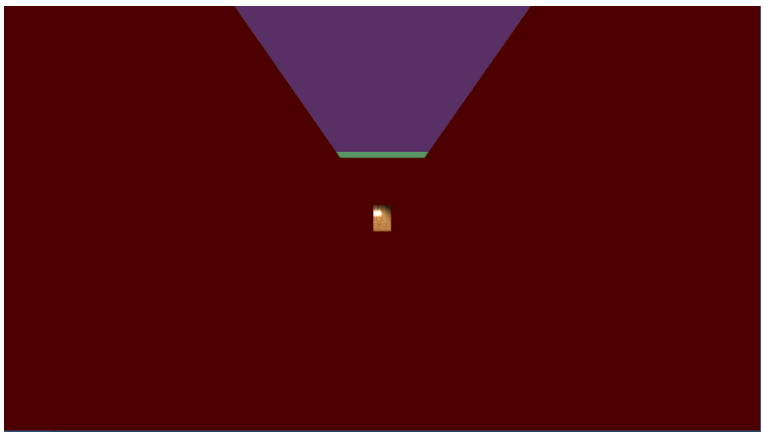
## P.7. Pavyzdinis OBJ failas

```
# Kubo OBJ failas

v  -0.5  -0.5  -0.5
v  -0.5  -0.5   0.5
v  -0.5   0.5  -0.5
v  -0.5   0.5   0.5
v   0.5  -0.5  -0.5
v   0.5  -0.5   0.5
v   0.5   0.5  -0.5
v   0.5   0.5   0.5

vt 0.0 0.0
vt 0.0 1.0
vt 1.0 1.0
vt 1.0 0.0

vn  0.0  0.0  1.0
vn  0.0  0.0 -1.0
vn  0.0  1.0  0.0
vn  0.0 -1.0  0.0
vn  1.0  0.0  0.0
vn -1.0  0.0  0.0

f  1/1/2  7/3/2  5/4/2
f  1/1/2  3/2/2  7/3/2
f  1/4/6  4/2/6  3/3/6
f  1/4/6  2/1/6  4/2/6
f  3/1/3  8/3/3  7/4/3
f  3/1/3  4/2/3  8/3/3
f  5/1/5  7/2/5  8/3/5
f  5/1/5  8/3/5  6/4/5
f  1/1/4  5/2/4  6/3/4
f  1/1/4  6/3/4  2/4/4
f  2/4/1  6/1/1  8/2/1
f  2/4/1  8/2/1  4/3/1
```

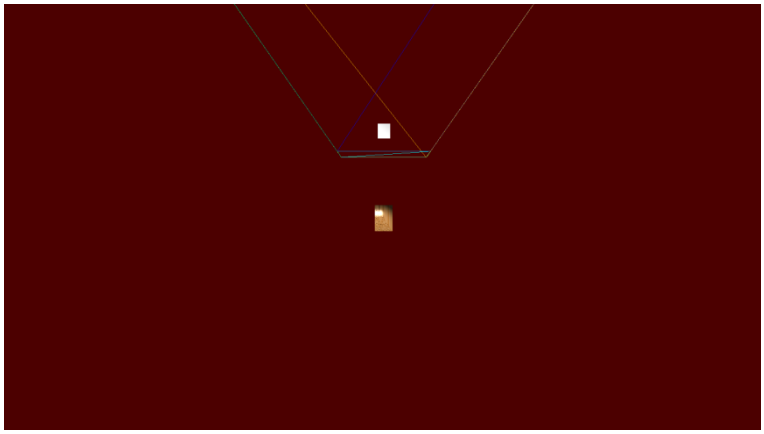
## P.8. Scenos „Kubas“ nuotraukos



Scenos „Kubas“ vaizdas iš kameros (kairėje - pradinė scena, dešinėje - pasisukęs kubas scenoje)



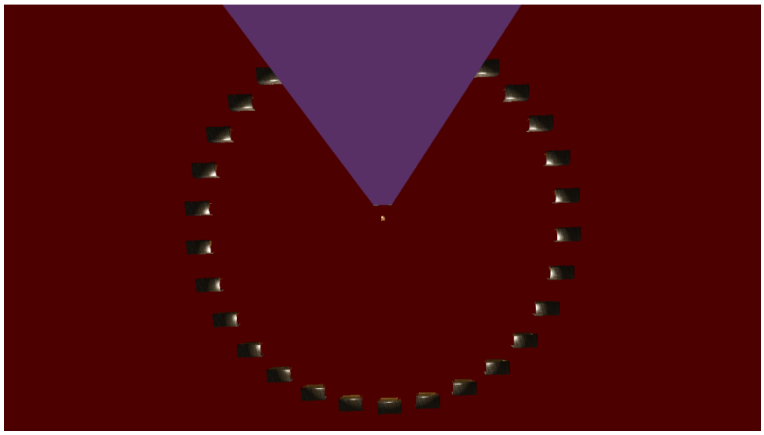
Scenos „Kubas“ vaizdas iš viršaus įjungus vaizduojamojo tūrio rodymą (pats algoritmas nejjungtas)



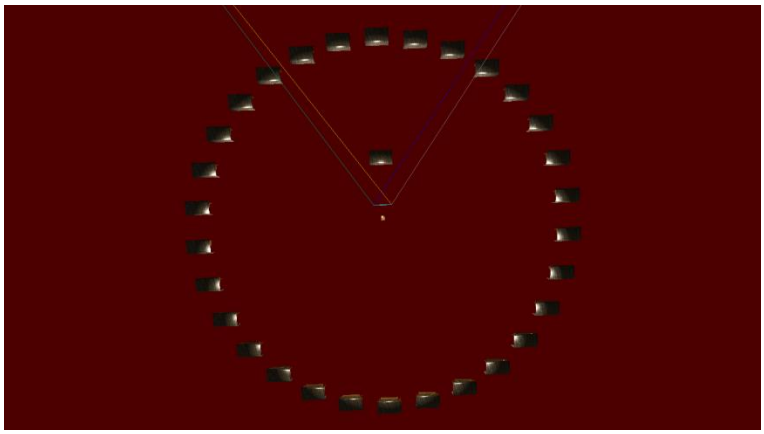
Scenos „Kubas“ vaizdas iš viršaus įjungus vaizduojamojo tūrio linijinį rodymą (pats algoritmas nejjungtas)

## P.9. Scenos „Prekystaliai“ nuotraukos

Scenos „Prekystaliai“ vaizdas iš kameros

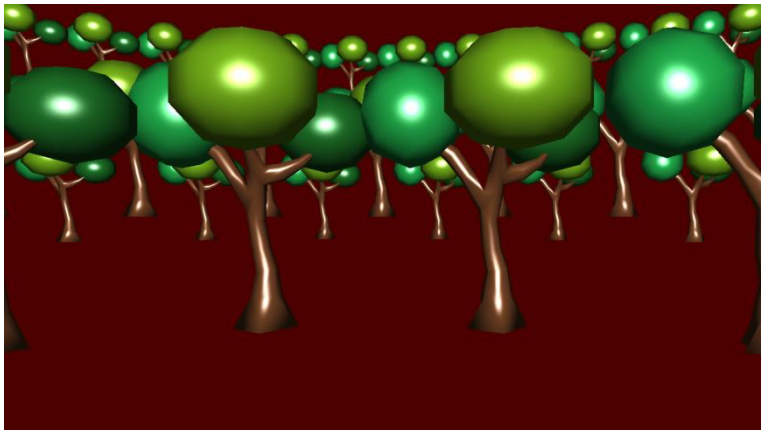


Scenos „Prekystalis“ vaizdas iš viršaus įjungus vaizduojamojo tūrio rodymą (pats algoritmas nejjungtas)

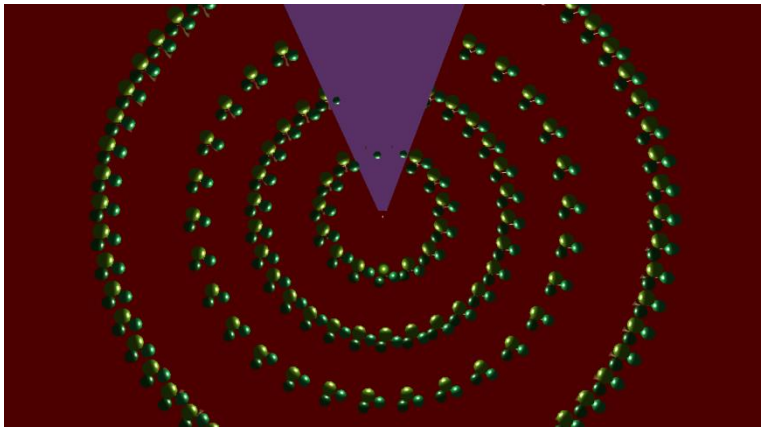


Scenos „Prekystalis“ vaizdas iš viršaus įjungus linijinį vaizduojamojo tūrio rodymą (pats algoritmas nejjungtas)

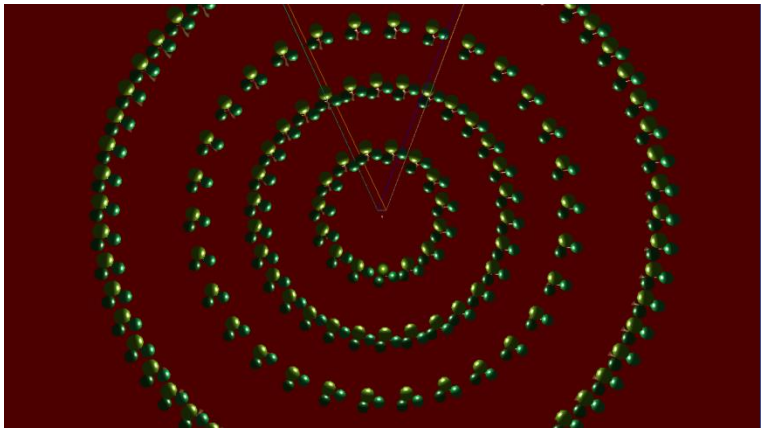
## P.10. Scenos „Medžiai“ nuotraukos



Scenos „Medžiai“ vaizdas iš kameros

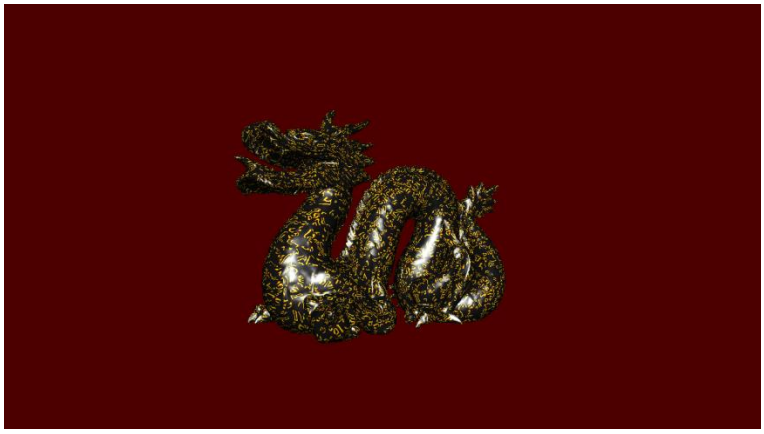


Scenos „Medžiai“ vaizdas iš viršaus įjungus vaizduojamojo tūrio rodymą (pats algoritmas nejjungtas)

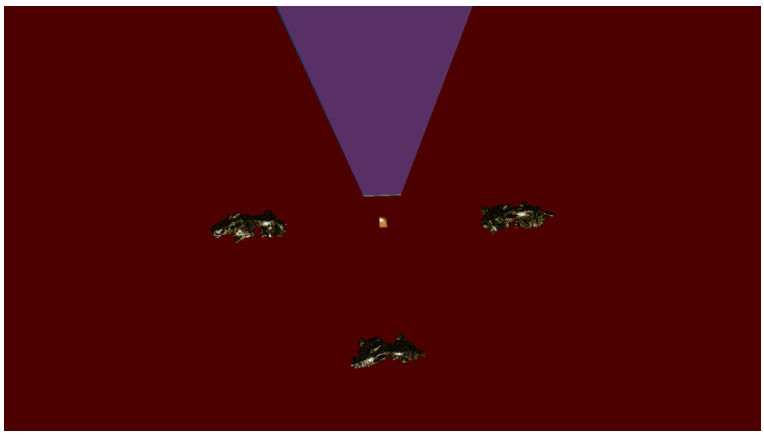


Scenos „Medžiai“ vaizdas iš viršaus įjungus linijinį vaizduojamojo tūrio rodymą (pats algoritmas nejjungtas)

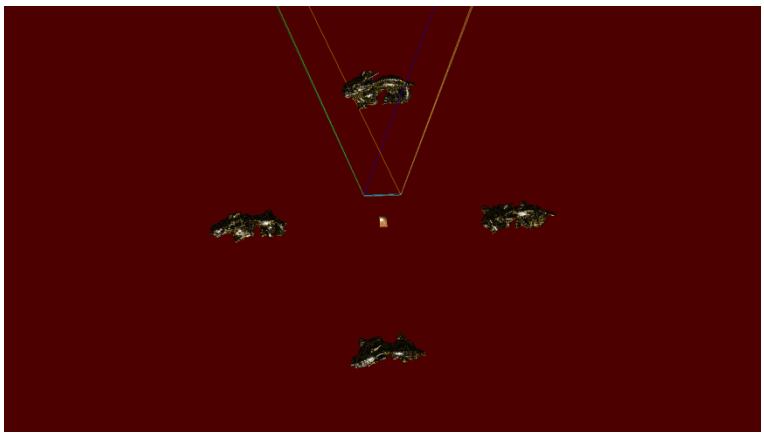
## P.11. Scenos „Drakonai“ nuotraukos



Scenos „Drakonai“ vaizdas iš kameros



Scenos „Drakonai“ vaizdas iš viršaus įjungus vaizduojamojo tūrio rodymą (pats algoritmas nejjungtas)



Scenos „Drakonai“ vaizdas iš viršaus įjungus linijinį vaizduojamojo tūrio rodymą (pats algoritmas nejjungtas)

## P.12. Resursai

Tyrime naudojamos programos kodas:
https://github.com/ErnestasMitkus/projbaka

Internetinės nuorodos į darbe panaudotas iliustracijas:

1.1 pav. Sfera išreikšta trikampiais URL: http://thumb10.shutterstock.com/thumb_large/868231/331877837/stock-vector-wireframe-d-mesh-polygonal-vector-sphere-network-line-design-sphere-dot-and-structure-331877837.jpg

1.2 pav. Linijų matomumo nustatymas URL: http://imgur.com/82zENsc

1.3 pav. Linijos, po kiekvieno Robertso algoritmo žingsnių URL: http://imgur.com/7jzOlpB

1.4 pav. Kiekybinis nematomumas URL: http://imgur.com/LjNa3AW

1.5 pav. Briaunainio kontūras URL: http://imgur.com/g1P0Dro

1.6 pav. Kiekybinio nematomumo pavyzdys URL: http://imgur.com/9cvI80p

1.7 pav. Kreivių projekcijos į plokštumą z=0 URL: http://imgur.com/hcLkczq

1.8 pav. Probleminė situacija eilinei kreivei atsiradus žemiau pačios pirmosios kreivės (a) ir URL: http://imgur.com/QWXrDHC

1.9 pav. Z-bufferio (gylio buferio) pavyzdys nenormalizuotose koordinatėse URL: http://imgur.com/0d5g5s2

1.10 pav. Spindulių trasavimo metodas paviršių matomumui nustatyti URL: http://imgur.com/utjhSR9

1.11 pav. Spindulio trasavimo metodas neranda objekto tikrinimo ilgiui esant per dideliam URL: http://lodev.org/cgtutor/images/raycastmiss.gif

1.12 pav. Spindulio trasavimo metodas su mažu tikrinimo ilgiu URL: http://lodev.org/cgtutor/images/raycastmiss2.gif

1.13 pav. Tapytojo algoritmo problema URL: https://upload.wikimedia.org/wikipedia/commons/thumb/7/78/Painters_problem.svg/220px-Painters_problem.svg.png

1.14 pav. Vaizduojamasis tūris (angl. view frustum) URL: https://upload.wikimedia.org/wikipedia/commons/thumb/0/02/ViewFrustum.svg/220px-ViewFrustum.svg.png

1.15 pav. Gaubiančiojo apvalkalo metodo pavyzdžiai URL: http://www.inf.ufrgs.br/~dlmtavares/collisiondetection.png

1.16 pav. Galinių plokštumų atmetimas URL: http://imgur.com/KyBhbC6

1.17 pav. Neuždaro objekto galinių plokštumų atmetimas (kairėje be galinių plokštumų atmetimo, dešinėje - su) URL: http://revolvy-cdn.revolvyllc.netdna-cdn.com/images/cache/b6/47/7a/b6477a14db3029d1c6119039c32a3afb.png

1.18 pav. Galinių plokštumų atmetimo problema URL: http://imgur.com/5YqeJQJ

2.1 pav. OpenGL paprasčiausias atvaizdavimo procesas naudojant šešėliavimo programas URL: http://imgur.com/0aIZnSm

2.2 pav. Sistemos architektūros UML diagrama URL: http://imgur.com/pbptU5G

2.3 pav. OBJ failo nagrinėjimo schema URL: http://imgur.com/Ny9mlDk

2.4 pav. Kairysis objektas atvaizduojamas be apšvietimo, vidurinis - su aplinkos apšvietimu, dešinysis - su aplinkos ir veidrodiniu apšvietimu URL: http://imgur.com/3iZmGTq

3.1 pav. Vaizduojamojo tūrio kampų apskaičiavimas URL: http://imgur.com/M9OJOq4

3.2 pav. Vaizduojamojo tūrio tolimosios kairės sienos kampo pozicijos apskaičiavimas URL: http://imgur.com/UL0MHRQ

3.3 pav. Sienų viršūnės išdėstytos prieš laikrodžio rodyklę tvarką yra laikomos priekinėmis URL: http://imgur.com/u5NPwjx

3.4 pav. Scenos „Kubas“ 1 ir 2 testo variantų tyrimo rezultatai URL: http://imgur.com/k9agW6b

3.5 pav. Scenos „Kubas“ 2 testo varianto tyrimo rezultatai URL: http://imgur.com/2FlpCcI

3.6 pav. Scenos „Prekystaliai“ 1 ir 2 testo variantų tyrimo rezultatai URL: http://imgur.com/cBv3viS

3.7 pav. Scenos „Prekystaliai“ 2 testo varianto tyrimo rezultatai URL: http://imgur.com/wPYXyjb

3.8 pav. Scenos „Medžiai“ 1 ir 2 testo variantų tyrimo rezultatai URL: http://imgur.com/Whc3oin

3.9 pav. Scenos „Medžiai“ 2 testo varianto tyrimo rezultatai URL: http://imgur.com/BviIWsn

3.10 pav. Scenos „Drakonai“ 1 ir 2 testo variantų tyrimo rezultatai URL: http://imgur.com/G3GuxYZ

3.11 pav. Scenos „Drakonai“ 2 testo varianto tyrimo rezultatai URL: http://imgur.com/KeiXM71

Scenos „Kubas“ vaizdas iš kameros (kairėje - pradinė scena, dešinėje - pasisukęs kubas scenoje) URL: http://imgur.com/R4UdzMS

Scenos „Kubas“ vaizdas iš viršaus įjungus vaizduojamojo tūrio rodymą (pats algoritmas neįjungtas) URL: http://imgur.com/TvxvyVn

Scenos „Kubas“ vaizdas iš viršaus įjungus vaizduojamojo tūrio linijinį rodymą (pats algoritmas neįjungtas) URL: http://imgur.com/4z2GWfw

Scenos „Prekystaliai“ vaizdas iš kameros URL: http://imgur.com/lF8jdY5

Scenos „Prekystalis“ vaizdas iš viršaus įjungus vaizduojamojo tūrio rodymą (pats algoritmas neįjungtas) URL: http://imgur.com/Qta69S9

Scenos „Prekystalis“ vaizdas iš viršaus įjungus linijinį vaizduojamojo tūrio rodymą (pats algoritmas neįjungtas) URL: http://imgur.com/viSqLXs

Scenos „Medžiai“ vaizdas iš kameros URL: http://imgur.com/xrcissV

Scenos „Medžiai“ vaizdas iš viršaus įjungus vaizduojamojo tūrio rodymą (pats algoritmas neįjungtas) URL: http://imgur.com/Y2HU9xb

Scenos „Medžiai“ vaizdas iš viršaus įjungus linijinį vaizduojamojo tūrio rodymą (pats algoritmas neįjungtas) URL: http://imgur.com/qcgsL1x

Scenos „Drakonai“ vaizdas iš kameros URL: http://imgur.com/Pa8A012

Scenos „Drakonai“ vaizdas iš viršaus įjungus vaizduojamojo tūrio rodymą (pats algoritmas neįjungtas) URL: http://imgur.com/kiQDa6c

Scenos „Drakonai“ vaizdas iš viršaus įjungus linijinį vaizduojamojo tūrio rodymą (pats algoritmas neįjungtas) URL: http://imgur.com/J1SYJoO