VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Uždengtų objektų atkirtimas
Occlusion Culling

Magistro baigiamasis darbas

Atliko: Edgaras Pavlišinas

Darbo vadovas: Doc. dr. Antanas Lenkevičius

Recenzentas: Doc. dr. Rimvydas Krasauskas

Vilnius 2016

Santrauka

Uždengtų objektų atkirtimas yra būdas nustatyti, kurie scenos objektai nėra matomi, ir jų neatvaizduoti. Kadangi šie objektai nėra matomi, tai jie neturės jokios įtakos galutiniam vaizdui, tačiau naudos kompiuterio resursus.

Egzistuoja keletas skirtingų metodų nematomiems objektams nustatyti, tačiau visi šie metodai yra paremti ta pačia idėja, skiriasi tik detalės – naudojamas procesorius arba grafinis procesorius, ar naudojamos hierarchinės struktūros, ar naudojame praeito kadro rezultatus, ir kt. Šiame darbe pasiūlysime mišrų metodą, kuris apjungia procesorių ir grafinį procesorių bei naudoja dalį praeito kadro rezultatų. Pasiūlytas metodas buvo įgyvendintas ir jo greitaveika palyginta su kitų metodų. Darbe pristatomi gauti eksperimentinio tyrimo rezultatai ir pateikiamos išvados.

Raktiniai žodžiai: uždengtų objektų atkirtimas, gylio žemėlapis, atvaizdavimo spartinimas, atvaizdavimo optimizacija.

Summary

Occlusion culling is a method to determine which scene objects are not visible and discarding them. Because these objects are not visible they do not contribute to the final image and only waste computing resources.

There are several methods to determine occluded objects, however all these methods use the same idea. Difference between these methods is in implementation – whether computation is performed on CPU or GPU, whether hierarchical trees are used, whether we compute and use results right now or use results from the previous frame, etc. This thesis proposes a new hybrid occlusion culling method, which combines both, CPU and GPU, and uses some information from the previous frame. Proposed method has been implemented and performance results were compared with some other methods. Obtained results are presented along with conclusion.

Keywords: occlusion culling, depth buffer, improving rendering time, rendering optimization.

Turinys

Įvadas

Šiuolaikinės kompiuterinės technologijos sparčiai vystosi, bet didėjant kompiuterių trimatės grafikos atvaizdavimo spartai taip pat didėja reikalavimai ir scenos dydžiui, ir atvaizdavimo kokybei. Tačiau reikalavimai didėja greičiau negu technologijos, todėl didelis dėmesys yra skiriamas atvaizdavimo spartinimui ir optimizacijai. Vienas iš pagrindinių principų yra neatvaizduoti to, kas nebus matoma ekrane. Remiantis [Har11; Eng13] šaltiniais šiuo metu stacionarūs kompiuteriai kiekvieną kadrą gali apdoroti vos kelis tūkstančius atvaizdavimo komandų, o kompiuterinių žaidimų konsolės – kelis kartus daugiau. Taip pat sužinojome, kad atvaizdavimo komanda efektyviai naudotų GPU resursus, reikia atvaizduoti pakankamai daug primityvų, tam kad GPU dirbtų ilgiau negu užtrunka atvaizdavimo komandos apdorojimas CPU pusėje. Taigi jeigu matomumo testai bus per daug paprasti GPU resursai bus naudojami neefektyviai. Iš šaltinio [Ari15] taip pat sužinome, kad naudojant naujausią grafinio programavimo sąsajos versiją (DirectX 12, Mantle, Vulkan) apdorojamų atvaizdavimo komandų skaičius per sekundę padidėja kelis kartus. Tai parodo, kad yra neišnaudojama didelė gausybė GPU resursų, nes daug laiko sugaištama pačioms atvaizdavimo komandoms apdoroti. Taigi yra galimybė gauti labai didelę naudą sumažinant atvaizdavimo komandų skaičių, kas ir yra šio magistrinio darbo tikslas.

Pats primityviausias metodas nustato kurie objektai yra už matomos scenos dalies ribų ir tų objektų neatvaizduoja. Kartais to pakanka, tačiau neretai siekiama aptikti ir tuos objektus, kurie yra uždengti kitų objektų. Šitos problemos sprendimas yra labai sudėtingas ir, kaip matyti iš analizuojamų atkirtimo metodų, bei tiriamo algoritmo, yra prilyginamas tiesioginiam objektų atvaizdavimui. Darbe pristatyti metodai pasitelkia įvairiausias optimizacijas testuojamų objektų skaičiui sumažinti bei CPU ir GPU sąveikai pagerinti.

Dauguma išanalizuotų metodų objektų matomumui nustatyti naudoja GPU, tačiau tai sunaudoja resursus kuriuos mes bandome taupyti, todėl naivus šių metodų panaudojimas yra nepraktiškas ir nenaudingas. Metodams tobulėjant buvo pradėtas hierarchinių struktūrų panaudojimas testuojamų objektų skaičiui mažinti, bei pradėti naudoti statistiniai metodai matomumo užklausų skaičiui mažinti. Po tokių patobulinimų uždengtų objektų nustatymo metodai tapo praktiškai pritaikomi. Didėjant CPU pajėgumams ir siekiant dar labiau sumažinti GPU panaudojimą uždengtų objektų nustatymo algoritmas buvo pilnai perkeltas į CPU.

Šio darbo tikslas yra pasiūlyti ir realizuoti naują uždengtų objektų nustatymo ir atkirtimo metodą bei atlikti eksperimentinį tyrimą lyginant rezultatus su jau egzistuojančių metodų efektyvumo rodikliais.

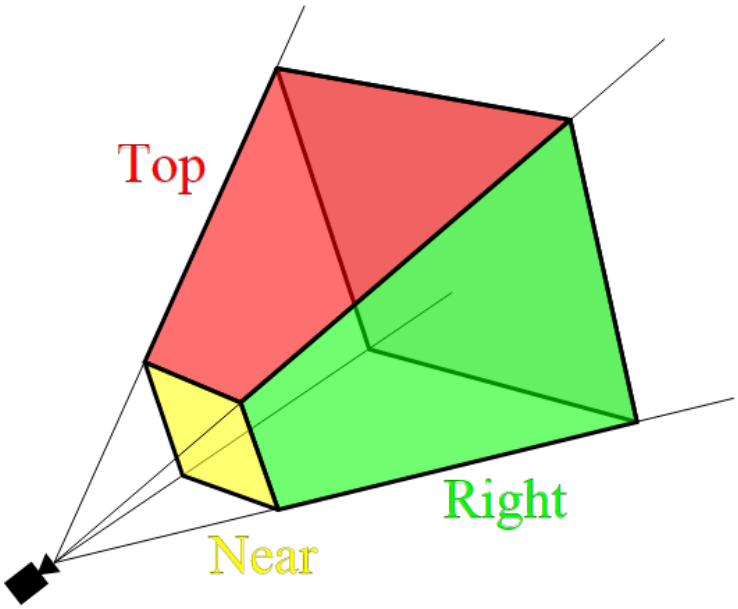
Darbo uždaviniai:

- išanalizuoti egzistuojančius uždengtų objektų nustatymo metodus, išryškinant jų privalumus ir trūkumus, bei optimizavimo galimybės;
- pasiūlyti naują uždengtų objektų nustatymo metodą;
- realizuoti pasiūlytą metodą programiškai;
- atlikti eksperimentinį tyrimą;
- palyginti rezultatus su kitais metodas.

1 Analitinė dalis

1.1 Vaizduojamasis tūris

Vaizduojamasis tūris (angl. viewing frustum; 1 pav.) yra nupjautinės piramidės formos regionas nusakantis kuri scenos dalis yra matoma. Tai yra erdvės dalis, kuri yra atkirsta 6 plokštumų. Šis tūris nurodo kas bus atvaizduojama ekrane, taigi, objektai esantys už vaizduojamojo tūrio ribų nebus tiesiogiai matomi. [Len11]



1 pav. Vaizduojamasis tūris
(paimta iš http://en.wikipedia.org/wiki/Viewing_frustum)

Kadangi objektai esantys už vaizduojamojo tūrio ribų nebus matomi, tai nėra jokios prasmės juos atvaizduoti. Neretai mes turime galimybę matyti tik mažą scenos dalį, taigi nustatydami nematomus objektus ir jų neatvaizduodami mes turėsime galimybę sutaupyti daug kompiuterio resursų. Tokia procedūra yra vadinama vaizduojamojo tūrio atkirtimu (angl. frustum culling). Šis atkirtimo metodas yra priskiriamas objekto erdvės metodams (angl. object space), tai reiškia, kad skaičiavimai yra atliekami objekto, ar pasaulio, erdvėje. [AHH08]

1.1.1 Vaizduojamojo tūrio atkirtimas

Kompiuterinės grafikos programose vaizduojamasis tūris yra apibrėžiamas [4x4] dydžio matrica. Šiame darbe yra nagrinėjamas Villi Miettinen pasiūlytas algoritmas. [SJ02]

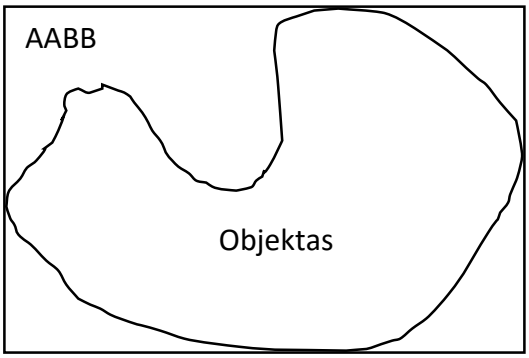
Šis algoritmas atkirtimą vykdo plokštumų pagalba. Taigi norint pasinaudoti šiuo algoritmu pirmiausia reikia rasti plokštumų normalinius vektorius. Turint matricą M reikia rasti plokštumas N_1, N_2, N_3, N_4, N_5, N_6. Remiantis [Kir04] tai padaryti nėra sudėtinga.

$$M = \begin{pmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{pmatrix}$$

$$N_1 = (M_{41} + M_{11}, M_{42} + M_{12}, M_{43} + M_{13}, M_{44} + M_{14})$$
$$N_2 = (M_{41} - M_{11}, M_{42} - M_{12}, M_{43} - M_{13}, M_{44} - M_{14})$$
$$N_3 = (M_{41} + M_{21}, M_{42} + M_{22}, M_{43} + M_{23}, M_{44} + M_{24})$$
$$N_4 = (M_{41} - M_{21}, M_{42} - M_{22}, M_{43} - M_{23}, M_{44} - M_{24})$$
$$N_5 = (M_{41} + M_{31}, M_{42} + M_{32}, M_{43} + M_{33}, M_{44} + M_{34})$$
$$N_6 = (M_{41} - M_{31}, M_{42} - M_{32}, M_{43} - M_{33}, M_{44} - M_{34})$$

Tokio algoritmo panaudojimas turi prasmę tik tuomet, kai atkirtimą galima įvykdyti greičiau negu atvaizdavimą. Tačiau bandant atkirsti tikslias objektų geometrijas bus sunaudota labai daug laiko. Šios problemos sprendimas yra paprastas – supaprastinti objektų geometrijas iki primityvių geometrinių objektų – gaubiančių apvalkalų, ir atkirtinėti apvalkalus. [Len07]

Nagrinėjamas algoritmas naudoja pačius paprasčiausius apvalkalus – pagal ašis išlygintas gaubiančias dėžes (angl. axis aligned bounding box (AABB); toliau – AABB; 2 pav.).



2 pav. Objekto AABB

AABB apskaičiavimas yra labai paprastas – surandame visų viršūnių minimalias ir maksimalias koordinates. Gautos reikšmės atitiks priešingas AABB viršūnes. Viena iš galimų algoritmo realizacijų pavaizduota žemiau (1 alg.). [Eri04]

```
// vertices - viršūnių aibė
// num_vertices - viršūnių skaičius
// v3min ir v3max - AABB viršūnės
vec3 v3min = vertices[0];
vec3 v3max = vertices[0];
for(int i = 1; i < num_vertices; i++) {
	v3min.x = min(v3min.x, vertices[i].x);
	v3min.y = min(v3min.y, vertices[i].y);
	v3min.z = min(v3min.z, vertices[i].z);
	v3max.x = max(v3max.x, vertices[i].x);
	v3max.y = max(v3max.y, vertices[i].y);
	v3max.z = max(v3max.z, vertices[i].z);
}
```

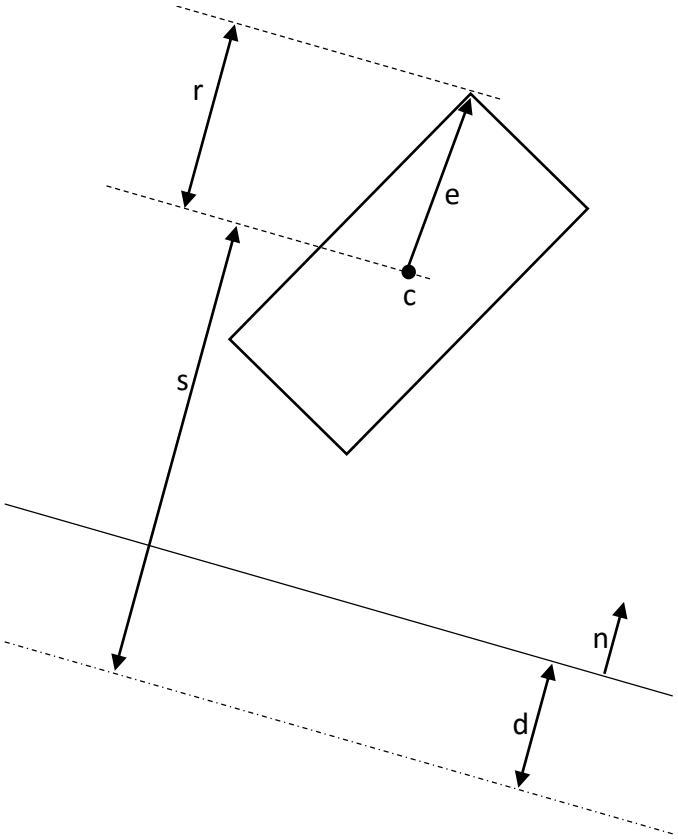
1 alg. AABB radimas (C++)

Tačiau analizuojamas algoritmas reikalauja kitokių AABB parametrų – centro ir pusįstrižainės, kuriuos yra lengva rasti. (2 alg.) [AHH08; Eri04]

```
vec3 center = (min + max) * 0.5f;
vec3 extent = (max – min) * 0.5f;
```

2 alg. Centro ir pusįstrižainės radimas (C++)

Apskaičiavus visus reikalingus parametrus galima nustatyti ar AABB pilnai yra neigiamoje plokštumos pusėje, t.y. už vaizduojamojo tūrio ribų. Turėdami pusįstrižainę *e* ir plokštumos normalės vektorių n pagal formulę $r = e \cdot |n|$ apskaičiuojame atstumą nuo AABB centro iki AABB krašto lygiagrečiai plokštumos normalės vektoriui. Tuomet pasinaudodami formule $s = c \cdot n + d$ apskaičiuojame atstumą nuo AABB centro *c* iki plokštumos, kuris bus neigiamas jeigu centras yra neigiamoje pusėje. Jeigu *r* ir *s* sumos reikšmė yra neigiama, tuomet AABB yra neigiamoje plokštumos pusėje ir objektas nėra matomas. Tačiau jeigu sumos reikšmė yra teigiama, tai darome prielaidą, kad objektas yra teigiamoje plokštumos pusėje. Tas atvejis kai plokštuma yra kertama nėra atskirai nagrinėjamas. 3 pav pavaizduoja skaičiavimų geometrinį vaizdą. [Eri04]



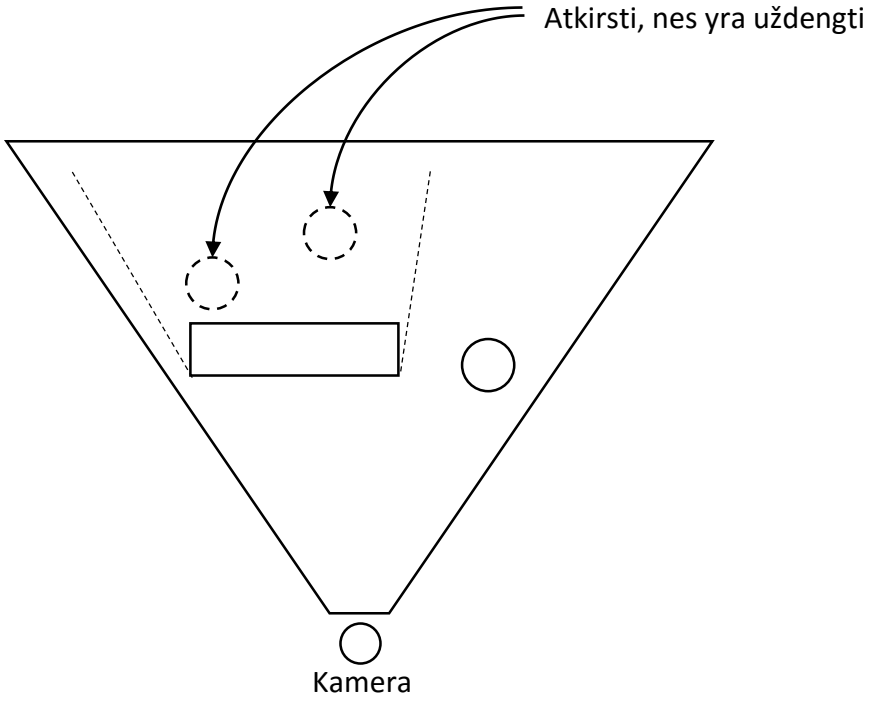
3 pav. Atkirtimo algoritmo geometrinis vaizdas

Tai buvo tik vienos plokštumos atkirtimas. Analogiškus skaičiavimus atliekame su kitomis 5-mis plokštumomis ir jeigu bent vienu atveju AABB buvo neigiamoje plokštumos pusėje, tai objektas tikrai nėra matomas. Bet jeigu nė su viena plokštuma negauname tokio rezultato, tai objektai yra potencialiai matomas (tikslesni skaičiavimai nėra atliekami). [Eri04]

Įprastai vaizduojamojo tūrio atkirtimas yra naudojamas kaip pirminė optimizacija su tikslu kuo greičiau atkirsti nematomus objektus. Tuomet uždengtų objektų atkirtimo metu bus mažiau objektų, su kuriais atliekami skaičiavimai.

1.2 Uždengtų objektų atkirtimo metodai

Uždengtų objektų atkirtimas (angl. occlusion culling) turi tik 1 tikslą – nustatyti kurie objektai yra pilnai uždengti kitų objektų iš dabartinio žiūrėjimo taško ir juos atskirsti (4 pav.). Šiam tikslui pasiekti yra siūlomi ne vienas metodas, tačiau jie visi yra paremti ta pačia idėja.



4 pav. Uždengtų objektų atkirtimas (vaizdas iš viršaus)

1.2.1 Potencialiai matomos aibės metodas

Potencialiai matomos aibės metodas (angl. potentially visible set; toliau – PVS) pirmą kartą buvo paminėtas dar 1990 metais. Autorius siūlė padalinti trimatę erdvę į regionus ir kiekvienam regionui nustatyti kurie objektai yra matomi. Tai galima atlikti programos kūrimo metu ir tik išsaugoti rezultatus. Scenos atvaizdavimo metu atvaizduojame visus objektus matomus iš to regiono. Nesunku suprasti, kad tokiu būdu bus atvaizduota nemažai objektų, kurie nėra matomi žiūrint iš dabartino taško. Toks metodas yra vadinamas konservatyviu. Priklausomai nuo skirtingų aplinkybių tokio metodo gali nepakakti, todėl jų egzistuoja ir daugiau: [CCSD03]

- Konservatyvus (angl. conservative) metodas pervertina matomumą, ir gali bandyti atvaizduoti daug nematomu objektų, tačiau nedaro priešingų klaidų – jeigu objektas yra matomas tai jis tikrai bus atvaizduotas;
- Agresyvus (angl. aggressive) metodas pervertina nematomumą, ir kai kurie mažesni arba beveik pilnai uždengti objektai gali būti neatvaizduoti, tačiau praktikoje netgi mažos paklaidos yra stipriai pastebimos;
- Apytiksliai (angl. approximate) metodai daro abiejų tipų klaidas, tačiau su mažesne tikimybe;

- Tikslūs (angl. exact) metodai duoda tikslius rezultatus, tačiau tam gali prireikti per daug resursų.

1.2.2 Naivus uždengtų objektų atkirtimo metodas

Taigi kaip galime nustatyti uždengtus objektus? Pasirodo, kad galima pasinaudoti GPU atvaizdavimu ir nuskaityti kiek pikselių buvo atvaizduota. Taigi jeigu atvaizduotų pikselių skaičius yra nulis tai objektas buvo pilnai uždengtas. Toks metodas yra priskiriamas vaizdo erdvės metodams (angl. image space), nes yra dirbama ne su pačiais objektais, o su jų projekcijomis.

Naivus uždengtų objektų atkirtimo metodas yra labai tiesioginis – užklausos pagalba sužinome kiek pikselių tam tikras objektas būtų atvaizdavęs ekrane, ir nusprendžiame, ar mums reikia atvaizduoti tą objektą, t.y. atliekame matomumo testą. Algoritmas:

1. Sukuriame užklausą;
2. Išjungiame atvaizdavimą ekrane;
3. Atvaizduojame objektą;
4. Įjungiame atvaizdavimą ekrane;
5. Iš užklausos gauname pikselių skaičių kuris būtų atvaizduotas;
6. Jeigu pikselius skaičiaus daugiau negu 0 (arba koks nors kitas pasirinktas skaičius);
    a. Atvaizduojame objektą;

Tačiau čia slypi problema – kadangi CPU ir GPU yra atskiri prietaisai, kurie dirba lygiagrečiai, tai kiekvieną kartą, norėdami sužinoti matomų pikselių skaičių, priversime šiuos prietaisus sinchronizuotis ir užtruksime daug laiko laukdami rezultatų, arba GPU neturės darbo ir švaistys laiką belaukiant sekančios komandos. [Sek04]

1.2.3 Pagerintas naivus metodas

Šiai problemai spręsti yra siūlomas pagerintas metodas, kuris tik dalinai išsprendžia šią problemą. Kad nelaukti rezultatų kiekvienam objektui atskirai, tai galima sukurti ir panaudoti daug užklausų vienu metu:

1. Sukuriame n užklausų;
2. Išjungiame atvaizdavimą ekrane;
3. Atvaizduojame n objektų kiekvienam panaudodami po 1 užklausą;

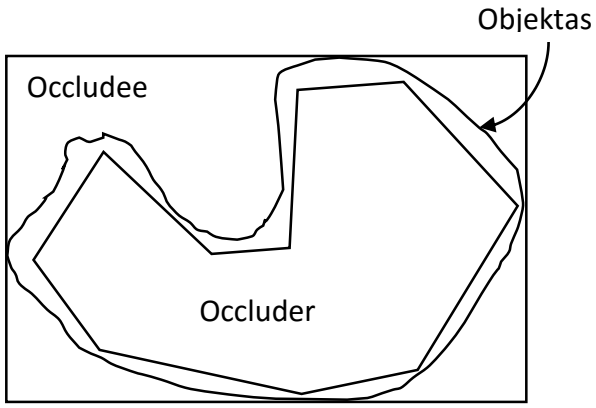
4. Įjungiame atvaizdavimą ekrane;
5. Iš kiekvienos užklausos gauname pikselių skaičių kuris būtų atvaizduotas;
6. Jeigu pikselius skaičiaus daugiau negu 0 (arba koks nors kitas pasirinktas skaičius);
    a. Atvaizduojame objektą;

Tokiu būdu buvo išspręsta lygiagretaus darbo problema, tačiau rezultatai nedaug pagerės. Atlikinėdami matomumo testus su tiksliomis objektų geometrijomis priversime GPU atlikti labai daug darbo ir sunaudosime daug resursų, gal net tiek pat, kiek reikėtų objektams atvaizduoti tiesiogiai. [Sek04; WB05]

1.2.4 Uždangos ir matomumo testo geometrijos

Kaip buvo minėta anksčiau, dabar GPU sugaišta per daug laiko atlikinėjant testus. Problema yra tame, kad yra naudojama tiksli objektų geometrija. Tai duoda didžiausią tikslumą, bet sunaudoja per daug resursų – butų efektyviau atvaizduoti objektus be jokių testų. Taigi yra siūloma kiekvienam objektui sukurti po dvi papildomas geometrijas (5 pav.):

- Pirmoji pilnai aprėpia visą objekto geometriją ir yra nemažesnė negu pats objektas (gali būti kad ir AABB); ji bus naudojama matomumo testui atlikti (angl. occludee);
- Antroji yra pilnai objekto viduje ir yra nedidesnė negu pats objektas; ši geometrija bus naudojama kaip uždanga (angl. occluder);



5 pav. Occludee ir occluder pavyzdys

Abi šios geometrijos turėtų būti labai paprastos, taigi jų dydžiai gali daug skirtis nuo objekto geometrijos. Paprastumo reikia tam, kad sumažinti geometrijos sudėtingumą ir pagreitinti skaičiavimus, net ir tuo atveju jeigu dėl to sumažės tikslumas. Algoritmo pakeitimai bus minimalūs: pradžioje atvaizduojame visas uždangos geometrijas, o testuodami naudojame matomumo testo geometrijas, o ne pačio objekto geometriją. [Sek04; WB05]

Taip pat yra siūlomos ir papildomos optimizacijos:

- Jeigu objektas yra mažas, tai jis tikriausiai nieko neuždengs, taigi nėra prasmės turėti uždangos geometriją;
- Jeigu objektas yra labai paprastas, tai matomumo testas gali kainuoti beveik tiek pat kiek ir tiesioginis objekto atvaizdavimas, taigi galima nenaudoti matomumo testo geometrijos;
- Jeigu objektas yra labai didelis, tai tikriausiai jis bus matomas beveik visą laiką, taigi galima nenaudoti matomumo testo geometrijos, arba naudoti labai paprastą;
- Galimos ir kitokios prielaidos.

Buvo išspręsta problema su per dideliu resursų panaudojimu, tačiau vis dar egzistuoja dvi problemos, kurios yra panašios į prieš tai buvusias:

1. Paduodame GPU labai paprastas komandas kurios yra įvykdomos greičiau, negu užtrunka pačios komandos padavimas, taigi GPU lauks naujų komandų kol mes galvojame ką atvaizduoti ekrane;
2. CPU yra daug greitesnis ir mes prašysime rezultato kol jo dar nėra, tokiu būdu sinchronizuodami CPU ir GPU.

1.2.5 Uždelsto matomumo testo metodas

Abi problemas galima išspręsti uždelstų testų pagalba. Idėja labai paprasta – atliekame testus dabar, o rezultatus nuskaitysime sekantį kadrą. Taigi darysime sprendimą, ar reikia atvaizduoti objektą, pagal praeito kadro rezultatus. Kitaip sakant, yra daroma prielaida, kad jeigu objektas buvo matomas praeitą kadrą, tai jis tikriausiai yra matomas ir dabartinį kartą. Analogiškai, jeigu objektas nebuvo matomas praeitą kadrą, tai jis tikriausiai nebus matomas ir dabartinį kadrą. Tačiau tai sudaro dvi naujas problemas:

1. Objektai, kurie dabartiniame kadre tapo nematomais bus atvaizduoti, nes jie buvo matomi praeitame kadre;
2. Objektai, kurie dabartiniame kadre tapo matomais nebus atvaizduoti.

Tačiau praktika rodo, kad abi šios problemos yra beveik nereikšmingos. Papildomai atvaizduotas vienas kitas objektas daro mažą įtaką greitaveikai, o staigiai atsiradęs objektas ekrane bus sunkiai pastebimas jeigu jis nėra didelis, arba jeigu yra atvaizduojama daug kadrų per sekundę. Algoritmas nedaug pasikeičia, taigi juo pasinaudoti yra lengva:

1. Gauname rezultatą iš praeito kadro matomumo testo;
2. Jeigu objektas buvo matomas, tai atvaizduojame objektą kartu darydami ir matomumo testą;
3. Jeigu objektas nebuvo matomas, tai darome matomumo testą panaudodami matomumo testo geometriją;

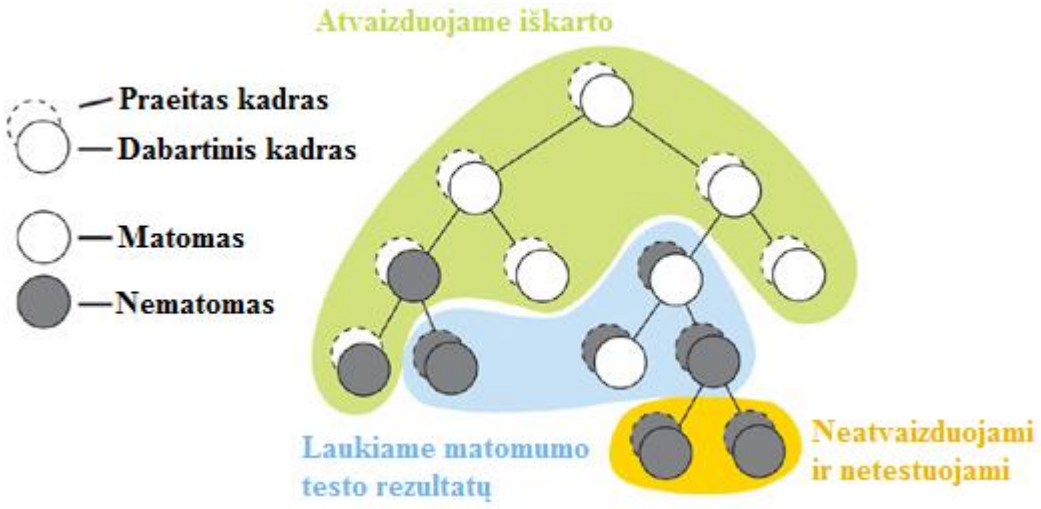
Nesunku pastebėti, kad nėra naudojama uždangos geometrija, vietoje jos yra naudojama tikroji objekto geometrija. Taigi darbo apimtis yra šiek tiek sumažinama. [Sek04; WB05]

1.2.6 Hierarchinis uždengtų objektų atkirtimo metodas

Buvo keli skirtingi pasiūlymai kaip optimizuoti atkirtimo algoritmą, kuriuose buvo spręstos skirtingos problemos, bet viena problema vis dar išliko – užklausų skaičius. Net jeigu užklausos ir yra labai paprastos, jos reikalauja nemažai CPU resursų, o atlikinėdami testus anksčiau pristatytais algoritmais, apie pusė visų atvaizdavimo komandų bus matomumo testai, taigi pačiam scenos atvaizdavimui lieka tik pusė visų turimų resursų, kas nėra priimtina. Hierarchinis uždengtų objektų atkirtimo metodas (angl. Coherent Hierarchical Culling arba CHC) bando spręsti tik užklausų skaičiaus problemą. [WB05; BWPP04]

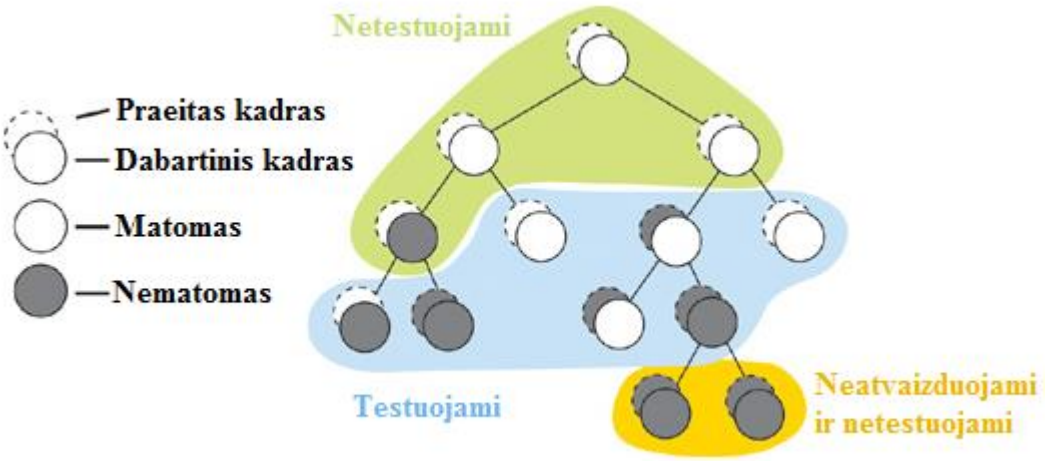
Šio metodo idėja yra šiek tiek panaši į uždelsto matomumo testo metodą – bandome spėti ar objektas yra matomas. Tačiau dabar mes sudedame scenos objektus į kokią nors hierarchinę struktūrą, pvz.: k-d medį (angl. k-d tree), grupuodami šalia esančius objektus į didesnes grupes ir sudarydami toms grupėms matomumo testo geometriją. Kadangi dabar objektai yra sugrupuoti, tai galime atlikinėti matomumo testus iškarto visai grupei vienu metu, taigi jeigu visa grupė nėra matoma, tai ir individualūs objektai esantys toje grupėje taip pat nebus matomi. Žalioje grupėje esantys objektai buvo matomi praeitą kadrą, taigi juos iškarto atvaizduojame, bet jie galėjo tapti

nematomais dabartiniame kadre, taigi atvaizduodami kartu atliekame matomumo testą. Kaip papildoma optimizacija yra siūloma atlikinėti matomumo testus tik kas N kadrų, o ne kiekvieną kartą. Mėlynoje grupėje objektai nebuvo matomi, bet galbūt tapo matomais dabartiniame kadre, taigi atliekame matomumo testą ir laukiame rezultatų. Geltonos grupės objektus net nereikės testuoti, nes jų tėvas nėra matomas. (6 pav.)



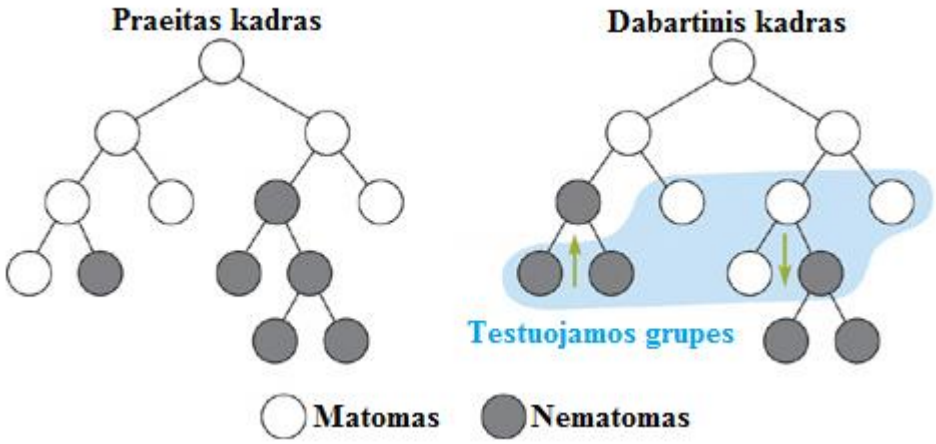
6 pav. k-d medžio atvaizdavimas (paimta iš [WB05])

Sugrupuoti nematomi objektai reikalaus mažiau testų, taigi sutaupysime resursus. Bet norisi atlikti mažiau testų ir su matomais objektais. Pasirodo tai padaryti yra labai paprasta. Mums net nereikia testuoti daugumos matomų objektų, nes matomumą galima nustatyti iš medžio vaikų – jeigu bent vienas vaikas yra matomas, tai ir visa grupė (tėvas) yra matomas, taigi šios grupės nereikia testuoti. Testuojami bus tik vaikai. Galima galvoti ir atvirkščiai – jeigu visi lapai nėra matomi, tai ir tėvas nėra matomas. Taigi mums užtenka testuoti tik lapus, ir tas grupes kurios yra nematomos, bet turi matomą tėvą (7 pav.). Dabar matomumo testų skaičius tiesiogiai priklauso nuo matomų objektų skaičiaus. [WB05; BWPP04]



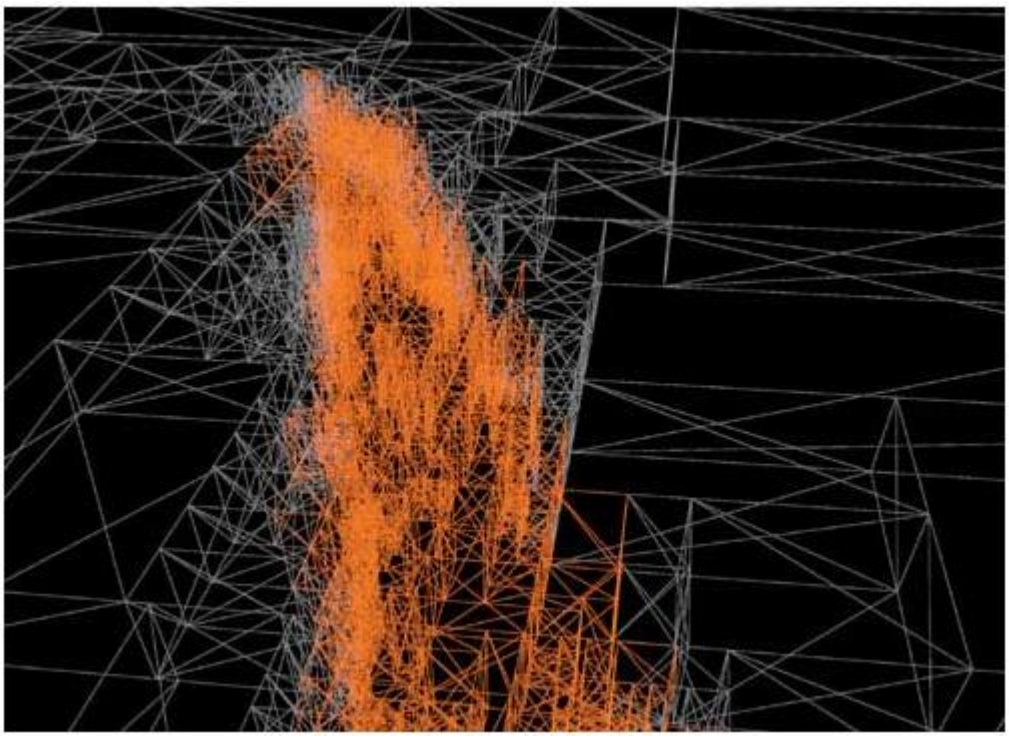
7 pav. Matomumo testo reikalavimai skirtingais atvejais (paimta iš [WB05])

Taigi nustatę, kad visi tėvai yra nematomi, pažymime tėvą kaip nematomą, o nustatę, kad tėvas yra matomas testuojame jo vaikus. Tokiu būdu judėdami į viršų ir į apačią sumažinsime užklausų skaičių ir tikrai atvaizduosime tai kad yra matoma (8 pav.). [WB05; BWPP04]



8 pav. Tėvai tampa nematomais arba matomais (paimta iš [WB05])

Eksperimentiniai tyrimai patvirtina laukiamus rezultatus: matomi objektai yra testuojami individualiai, o toliau esančios objektų grupės išlieka netestuotomis. Galima pastebėti, kad kuo toliau nematoma grupė yra nuo matomų objektų, tuo ji yra didesnė – buvo atkirsta anksčiau, taigi buvo sutaupyta daugiau resursų (9 pav.).



9 pav. Matomi objektai pažymėti oranžine spalva (paimta iš [WB05])

1.2.7 Pagerintas hierarchinis uždengtų objektų atkirtimo metodas

Praeitame skyriuje pristatytas CHC metodas sėkmingai sumažino naudojamų užklausų skaičių, tačiau jis vis tiek išliko maždaug lygus matomų objektų skaičiui, kas yra daug daugiau negu norėtume sunaudoti. Tačiau vystantis technologijoms ir didėjant GPU galingumui užklausų skaičius tapo labai aktualia problema. Paprastos geometrijos atvaizdavimas tampa vis greitesnis ir pigesnis, taigi vis dažniau pasitaiko tie atvejai, kai objektų atvaizdavimas be jokių matomumo testų būna pigesnis negu su matomumo testu. Šiai problemai išspręsti buvo pasiūlytas pagerintas hierarchinis uždengtų objektų atkirtimo metodas (angl. CHC++). Pagerintas algoritmas yra paremtas paprasta idėja – matomumo testų apjungimu. [MBW08]

Pirmiausia pradedame nuo CHC metodo pakeitimų. CHC metodas iškarto atvaizduodavo matomus ir nematomus objektus, tačiau CHC++ metode jų iškarto neatvaizduosime, o sudėsime į atskiras aibes: nematomus objektus dėsime į nematomų objektų aibę, o matomus – į matomų objektų aibę.

Nematomi objektai programos vykdymo metu yra dinamiškai apjungiami į didesnes grupes ir matomumo testas yra atliekamas iškarto visai grupei. Jeigu visa grupė buvo nematoma tai ir individualūs objektai yra nematomi, taigi buvo sutaupytos kelios užklausos. Tačiau jeigu grupė tapo

matoma, tuomet tenka testuoti kiekvieną objektą individualiai. Kad atrinkti kuriuos objektus galima apjungti į grupes buvo panaudoti statistiniai metodai. Kadangi tikslas yra apjungti nematomus objektus į didelę nematomą grupę, tai reikia parinkti tokiu objektus, kurie turi didelę tikimybę išlikti nematomais. Straipsnio autorius siūlo naudoti formulę

$$p_{keep}(i) \approx 0.99 - 0.7e^{-i}$$

tikimybei, kad objektas išliks nematomu, apskaičiuoti, kur i – kadrų skaičius, kurį objektas išliko nematomas. Tuomet apsibrėžiame nesėkmės tikimybės formulę objektų aibei, šiuo atveju nesėkmė – bent vienas objektas tapo matomas:

$$p_{fail}(M) = 1 - \prod_{\forall N \in M} p_{keep}(i_N)$$

kur M – objektų aibė, i_N – objekto N kadrų skaičius kurį jis išliko nematomas. Dabar galime apskaičiuoti vidutinį užklausų skaičių:

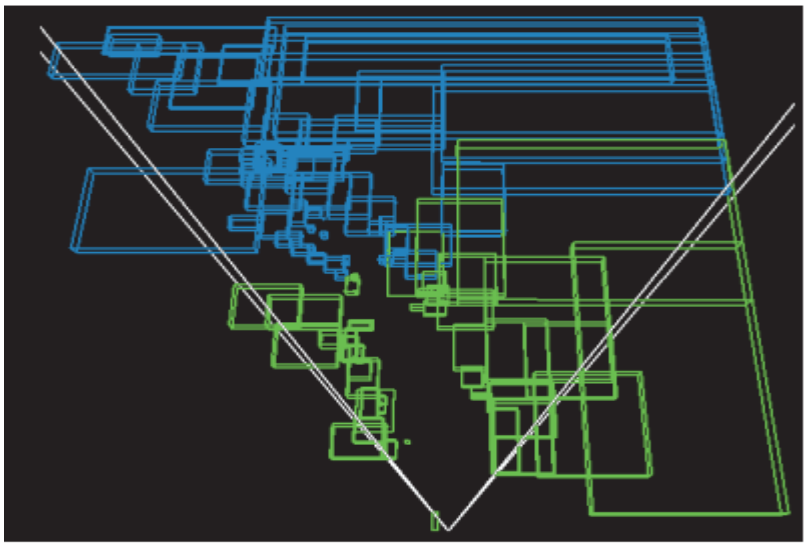
$$C(M) = 1 + p_{fail}(M) * |M|$$

kur 1 yra grupės užklausa, o $p_{fail}(M) * |M|$ yra vidutinis užklausų skaičius dėl klaidingų spėjimų.

Objektų parinkimui apjungti į grupes naudojamas godus algoritmas (angl. greedy algorithm), kad maksimizuoti iš apjungtos užklausos gautą naudą. Pirmiausia išrikiuojame visus nematomos aibės objektus $p_{keep}(i_N)$ mažėjimo tvarka. Tuomet paeiliui imame objektus ir dedame juos į aibę M ir skaičiuojame santykį tarp aibės dydžio ir vidutinės užklausų kainos, kitaip sakant apjungtos užklausos nauda. Nauda yra skaičiuojama pagal formulę

$$V(M) = \frac{|M|}{C(M)}$$

Pasirodo, kad tokiu būdu galime rasti $V(M)$ maksimumą. Gauta nematomų objektų aibė M ir bus ta objektų aibė, kurią reikia apjungti į vieną užklausą. Dabar atliekame matomumo testą ir pašaliname aibės M elementus iš nematomų objektų aibės. Šitą algoritmą kartojame kol nematomų objektų aibė lieka tuščia. Vaizdavimo taškui nejudant po kiek laiko visi nematomi objektai bus įtraukti į vieną grupę (10 pav.).



10 pav. Matomų ir nematomų objektų grupės pavaizduotos skirtingomis spalvomis (paimta iš [MBW08])

CHC++ taip pat sprendžia ir kitą problemą. Kadangi CHC siūlo atlikinėti matomų objektų testus kas N kadrų, tai gali susidaryti tokia situacija, kurios metu didelė dalis objektų bus testuojama tame pačiame kadre, taigi kažkurį kadrą bus atlikta daug daugiau darbo ir vaizdas gali trūkčioti. Šiai problemai spręsti yra siūlomas primityvus sprendimas: atsitiktinai pasirinkti skaičių N_0 kuris nurodo kadrų skaičių iki pirmojo matomumo testo, o sekantys testai bus atliekami įprastai kas N kadrų. N_0 rėžis priklauso nuo scenos, aparatūros, ir kitų parametrų, tačiau autoriaus eksperimentai parodo, jog pasirenkant $5 \leq N_o \leq 10$ dauguma atvejų bus gauti neblogi rezultatai. [MBW08]

CHC++ metodas yra tinkamas taikyti praktiškai, tačiau tam tikros problemos liko neišspręstos:

- Atvaizdavimo komandų skaičius. Nors sunaudojamas atvaizdavimo komandų skaičius ir sumažės, bet visgi norisi nešvaistyti atvaizdavimo komandų testams atlikinėti. Jeigu objektas yra nematomas, tai kodėl jis naudoja resursus?;
- Iki šiol buvo minimas tik tipinis atvejis, kai matomumo testo rezultatus gauname sekantį kadrą, tačiau tai ne visada yra tiesa. Stacionarūs kompiuteriai su keliomis GPU gali atvaizduoti kelis kadrus vienu metu, taigi užklausų rezultatai bus uždelsti ne vieną, o keletą kadrų, o bandydami nuskaityti rezultatus tą patį kadrą priversime CPU laukti kol GPU baigs darbą, tokiu būdu panaikinsime kelių GPU pranašumus. Ar šios problemos sprendimas yra įmanomas, kai naudojame GPU testams atlikinėti?

1.2.8 Programinis uždengtų objektų atkirtimo metodas

Visi iki šiol analizuoti atkirtimo metodai naudojo GPU atkirtimui spartinti, todėl didžioji tyrimų ir metodų gerinimų dalis koncentravosi ties matomumo testų skaičiaus mažinimu, užklausų skaičiaus mažinimu, mažesniu CPU ir GPU resursų suvartojimų, sinchronizacijos problemos sprendimais. Tačiau didėjant CPU galingumui pradėjo kilti naujos idėjos. Kas jeigu bandytume atlikti uždengtų objektų atkirtimą nenaudodami GPU? Tuomet nebūtų sinchronizacijos problemos, o be sinchronizacijos užklausų skaičius neturėtų įtakos, taigi CHC++ algoritmo problemos būtų iškarto išspręstos. Pasirodo, jog tai pasiekti yra įmanoma, tačiau tenka paaukoti šiek tiek tikslumo.

Intel korporacijos pasiūlytas metodas yra beveik identiškas į naivų metodą naudojant uždangos ir matomumo testo geometrijas: [Int12]

1. Programiškai atvaizduojame uždangos geometriją į 2D gylio žemėlapį;
2. Atskirai testuojame kiekvieną matomumo testo geometrijas (Intel naudojo AABB);
    a. Jeigu bent vienas pikselis yra matomas, tai iškarto sustabdome šio objekto testavimą ir jį atvaizduojame.

Buvo pritaikytos ir kelios optimizacijos: [Int12]

- Jeigu uždangos geometrija ekrane bus maža, tai ji tikriausiai nieko neuždegs, todėl ji nėra atvaizduojama;
- Jeigu matomumo testo geometrija ekrane bus labai maža (pvz. keli pikseliai) tai testo neatliekame ir objekto neatvaizduojame net jei ir jis yra matomas; tačiau priklausomai nuo reikalavimų šiuos objektus galima atvaizduoti visada.

Guerrilla Games ir DICE įmonės savo programinėje įrangoje taip pat naudoja programinį uždengtų objektų atkirtimo metodą, bet su keliais pakeitimais. Pagrindinis jų – naudojamas mažesnės rezoliucijos 2D gylio žemėlapis, tokiu būdu sumažinant sunaudojamų resursų kiekį. Mažesnės rezoliucijos gylio žemėlapis gali klaidingai nustatyti objektą kaip nematomą, tačiau tai įvyksta tik tuomet, kai objekto dydis ekrane yra tik keli pikseliai. [Gue11; Col11]

Nors programinis uždengtų objektų nustatymo metodas ir išsprendžia anksčiau minėtas CHC++ metodo problemas, tačiau jis atneša kitokias problemas:

- Reikia turėti atskiras uždangos ir matomumo testo geometrijas. Tai ne tik padidins atminties reikalavimus, bet ir reikalaus daug papildomo laiko, kad šias geometrijas sukurti. Nors automatinis generavimas ir yra įmanomas, tačiau jis nebus optimalus, ir nėra pritaikomas visais atvejais;
- Nors CPU greitaveika sparčiai auga, tačiau ji vis dar stipriai atsilieka nuo GPU greitaveikos, taigi ne optimali algoritmo realizacija, arba per daug detalios geometrijos turės labai didelę įtaką bendrai programos greitaveikai, taigi reikės skirti daug laiko tinkamų geometrijų parinkimui.

1.2.9 Mišrus uždengtų objektų atkirtimo metodas

Pagrindinės CHC++ ir kitų metodų problemos yra atvaizdavimo komandų naudojimas nematomų objektų testams atlikinėti – tai sunaudoja labai daug procesoriaus resursų ir nuskaitant rezultatus atsiranda sinchronizacijos tarp CPU ir GPU problema. Programinis uždengtų objektų atkirtimo metodas šias problemas išsprendžia, kadangi testavimas yra atliekamas programiškai, tačiau tuomet yra naudojami procesoriaus resursai testams atlikinėti, kas gali pareikalauti daug resursų. Siūlomo metodo idėja yra apjungti šias dvi idėjas – GPU įprastu darbo režimu sudaro gylio žemėlapį, taigi galime jį panaudoti, o patį testavimą atliksime CPU pagalba. Taigi šiame darbe tirsime tokį metodą:

1. Nuskaitome praeito kadro gylio žemėlapį ir jį naudojame kaip jau atvaizduotą uždangos geometriją (žr. 1.2.8 skyrių);
2. Programiškai testuojame objektų matomumo testo geometrijas.

Tokiu būdu bus pašalinta matomumo testavimo metu atsirandanti sinchronizavimo problema, o uždangos geometrijos gylio žemėlapį gausime beveik nemokamai, taigi galėsime pašalinti pirmąjį programinio atkirtimo metodo žingsnį, kurio metu yra sudaromas gylio žemėlapis.

1.3 Analitinės dalies išvados

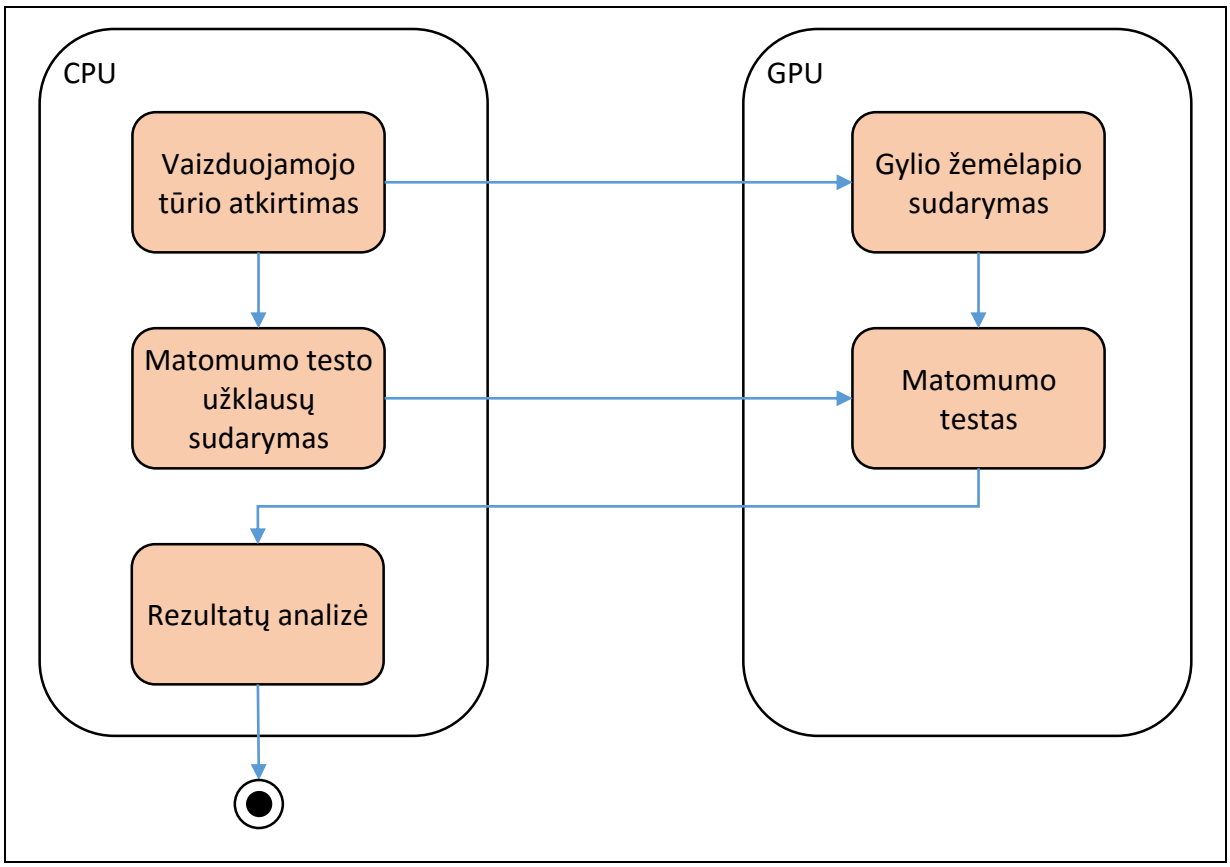
1. Atlikus literatūros analizę nustatyta, kad nematomų objektų nustatymas prasideda nuo vaizduojamojo tūrio atkirtimo metodo, kuris yra pats paprasčiausias metodas, tačiau jis atkerta tik tuos objektus, kurie nepatenka į matomumo zoną, bet dėl didelio šio metodo greičio jis yra naudojamas kaip optimizacija tikriems uždengtų objektų atkirtimo metodams. Priklausomai nuo projekto reikalavimo šio vieno metodo gali pakakti tikslui pasiekti;
2. Tolimesnė literatūros analizė parodo, jog sudėtingesni metodai nustato ar tam tikras objektas yra uždengtas kitų objektų, taigi yra atkertama dar daugiau objektų. Šių metodų grupė evoliucionavo nuo pačio paprasčiausio metodo. Didėjant reikalavimams pradinis metodas pradėjo tobulėti mažindamas grafinio procesoriaus darbą. Tai buvo pasiekta pritaikant hierarchines struktūras bei statistinius modelius;
3. Analizuojant naujausius literatūros šaltinius nustatyta, kad tobulėjant technologijoms atsirado galimybė atsisakyti grafinio procesoriaus panaudojimo ir metodas buvo realizuotas programiškai. Toks žingsnis išsprendė visas egzistuojančias problemas kituose metoduose, tačiau reikalavo labai daug procesoriaus resursų;
4. Remiantis gautomis išvadomis buvo nuspręsta sumažinti procesoriaus resursų panaudojimą, taigi pasiūlėme metodą, kuris paima iš grafinio spartintuvo dalį praeito kadro informacijos ir panaudoja uždengtų objektų nustatymui.

2 Projektinė dalis

Šiame skyriuje yra pristatomos atkirtimo metodų veikimo diagramos ir parodoma kaip tai evoliucionavo į šio darbo siūlomą metodą. Kiekvienos modifikacijos tikslas yra išspręsti kurią nors iš greitaveikos problemų.

2.1 Pradinis atkirtimo metodas

Pradiniu metodu buvo paimtas naivus uždengtų objektų atkirtimo metodas, kuris tiesiogiai įgyvendina uždengtų objektų atkirtimo idėją. 11 pav diagramoje yra parodyta šio metodo veiksmų seka.



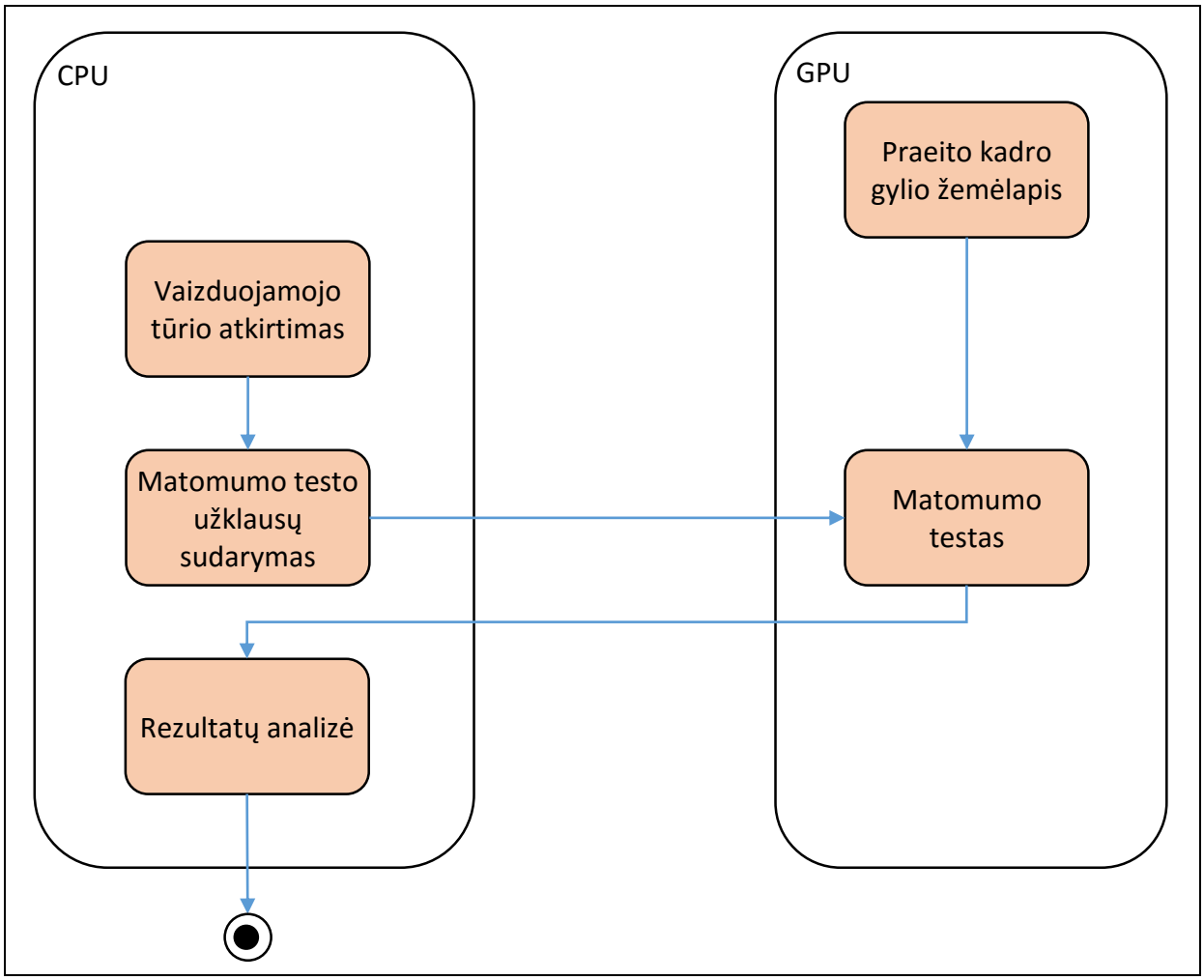
11 pav. Naivaus uždengtų objektų atkirtimo metodo veikimo diagrama

Toks naivus metodas turi nemažai trūkumų:

- Atkirtimo metu objektai bus atvaizduojami 2 kartus: sudarant gylio žemėlapį ir atliekant matomumo testus;
- Matomumo testo rezultatai bus nuskaitomi CPU pusėje, o tam reikia sinchronizacijos tarp CPU ir GPU, bei papildomo laiko perduodant duomenis.

2.2 Metodo modifikacijos

Pirmosios modifikacijos tikslas yra panaikinti gylio žemėlapio sudarymo žingsnį (12 pav.). Gylio žemėlapis yra būtinas norint pasinaudoti šiuo atkirtimo metodu, bet mes nenorime turėti papildomo žingsnio, taigi sprendimas yra gana paprastas – pasinaudosime praeito kadro gylio žemėlapiu. Praeito kadro gylio žemėlapis bus sudarytas scenos atvaizdavimo metu, taigi pasiekiame mūsų tikslą beveik nemokamai.

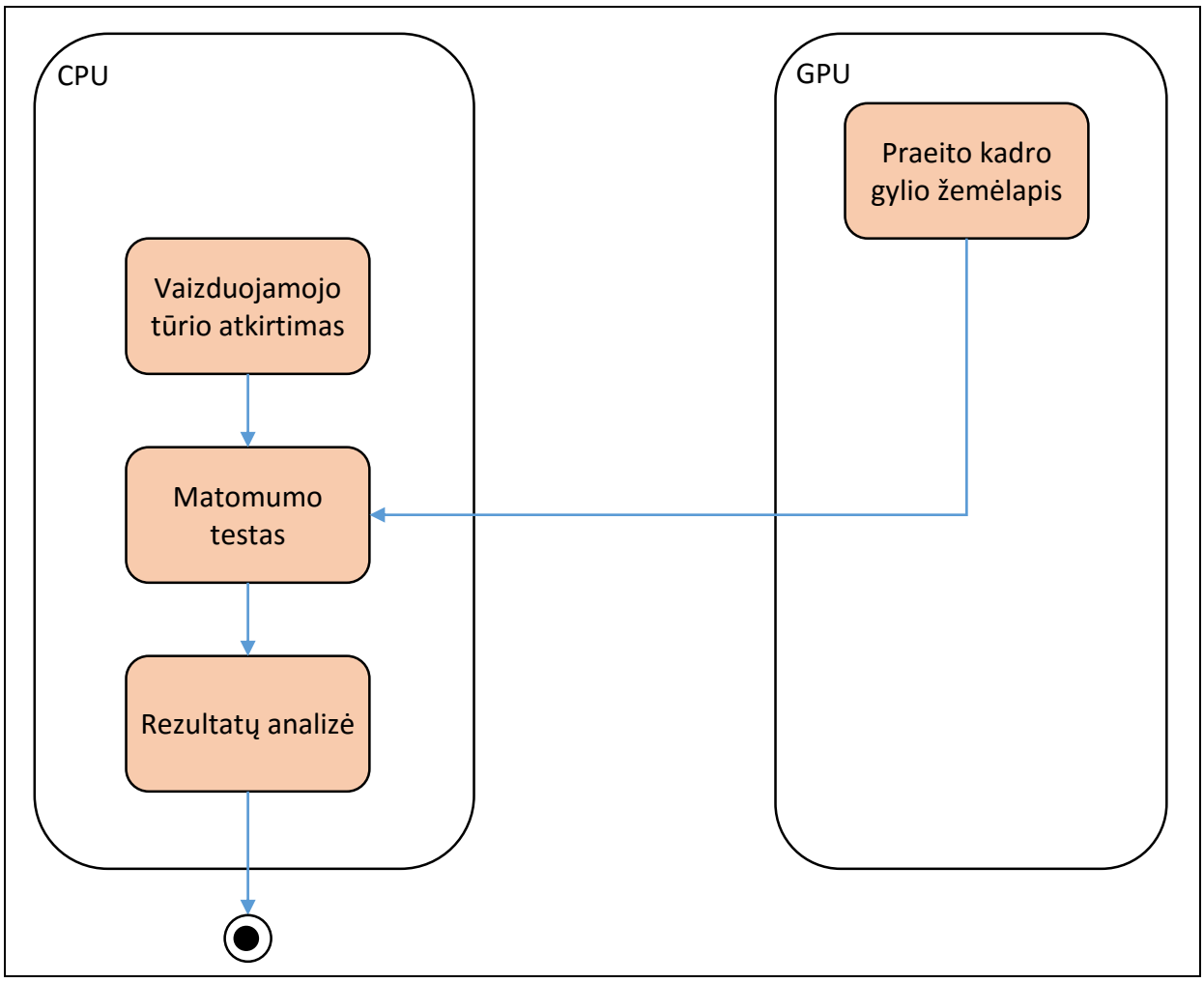


12 pav. Gylio žemėlapio sudarymo žingsnio panaikinimas

Tokia modifikacija sutaupo resursus, tačiau turi savų trūkumų:

- Kadangi naudojame praeito kadro gylio žemėlapį, tai negalėsime gauti korektiškų rezultatų, jeigu atlikinėsime matomumo testus naujame kadre, taigi atliekant matomumo testus naudosime praeito kadro duomenis: objektų transformacijas, kameros poziciją ir kt.;
- Naudojant praeito kadro duomenis atsiranda naujas trūkumas – praeitame kadre kuris nors objektas yra matomas, o naujame kadre jau nėra matomas, tačiau matomumo testas rezultatas pasakys, kad objektas yra matomas. Taigi papildomai bus atvaizduoti keli objektai, tačiau didelės įtakos greitaveikai tai neturės;
- Yra galimas ir atvirkščias variantas – praeitame kadre objektas nebuvo matomas, o naujame kadre jau yra matomas, tačiau matomumo testo rezultatas – nematomas. Taigi objektai ekrane atsiras vienu kadru vėliau, negu turėtų. Priklausomai nuo to, ar tarp kadrų įvyksta mažas ar didelis pokytis, tai gali būti mažai pastebima, taigi neretai šį trūkumą galima ignoruoti.

Ankstesniame skyriuje buvo paminėtas programinis uždengtų objektų atkirtimo metodas, kurio idėja yra atlikti visus skaičiavimus pasinaudojant CPU. Kadangi procesorius nėra specializuotas darbui su grafika, taigi visi skaičiavimai užtrunka daug daugiau laiko. Tačiau pasinaudojus programinio atkirtimo metodo idėja galima atlikinėti matomumo testus programiškai, t.y. be GPU pagalbos. Tokiu būdu sutaupysime resursus, kuriuos naudojame užklausų sudarymui ir laukdami rezultatų iš GPU. Norint įgyvendinti šią idėją reikia turėti gylio žemėlapį CPU pusėje. Taigi sekančios modifikacijos (13 pav.) tikslas yra perkelti matomumo testus į CPU pusę.



13 pav. Matomumo testo perkėlimas į CPU

Tačiau ir ši modifikacija nėra be trūkumų:

1. Reikia kopijuoti gylio žemėlapį iš GPU į CPU, o tam reikalinga sinchronizacija tarp CPU ir GPU, bei bus sugaištas laikas perduodant duomenis;
2. CPU rastrizacija yra lėtesnė už GPU rastrizaciją, taigi užtruks daugiau laiko.

Pasirodo, kad (1) trūkumas šios modifikacijos atveju yra dalinai išspręstas – kadangi yra naudojamas praeito kadro gylio žemėlapis, tai galima pradėti duomenų kopijavimą iškarto, kai gylio žemėlapis yra pilnai sudarytas. Tuo metu, kai atlikinėsime veiksmus, kurie nekeičia gylio žemėlapio, sistema kopijuos duomenis, o kai pasieksime sekančio kadro uždengtų objektų atkirtimo žingsnį gylio žemėlapis jau bus nukopijuotas, taigi nereikės laukti kol CPU ir GPU bus sinchronizuoti. Duomenų perdavimas vis tiek naudos sistemos resursus, bet kadangi tai vyksta lygiagrečiai mes to nepastebėsime.

Antrasis trūkumas taip pat gali būti dalinai išspręstas – sumažinus gylio žemėlapio rezoliuciją 2 arba 4 kartus atitinkamai pikselių skaičius sumažės 4 ir 16 kartų. Taigi perėjus nuo 1 pikselio tikslumo iki 16 pikselių tikslumo teoriškai reikės atlikti 16 kartų mažiau skaičiavimų, tokiu būdu išspręsime (2) trūkumą. Priklausomai nuo to, kaip mažinsime gylio žemėlapį, matomumo testai rezultatai nurodys mažiau arba daugiau matomų objektų, negu yra iš tikrųjų. Atvaizdavus daugiau objektų negu yra iš tikro yra matomi iššvaistysime šiek tiek resursų, o neatvaizdavus – nematysime mažos objekto dalelės, atitinkamai iki 3 ($2x2-1$) ir iki 15 ($4x4-1$) pikselių. Įprastai tokia paklaida yra labai sunkiai pastebima, taigi ją galima ignoruoti. Tačiau jeigu bandytume mažinti rezoliuciją dar labiau, šios paklaidos taptų labiau matomos, taigi reikėtų pakeisti gylio žemėlapio mažinimo metodą ir rezultate atvaizduosime daugiau objektų negu iš tikrųjų yra matoma. Priklausomai nuo to, kokiomis sąlygomis atkirtimas yra naudojamas, galima pasirinkti norimą tikslumą ir paklaidą – tai kompromisas tarp vaizdo korektiškumo ir greitaveikos.

2.3 Modifikuoto metodo žingsnių realizavimo detalės

Tiriamas atkirtimo metodas buvo realizuotas C++ programavimo kalba ir buvo panaudota DirectX 11 grafinio programavimo sąsają, taigi toliau pateikiami kodo fragmentai yra skirti C++ kalbai.

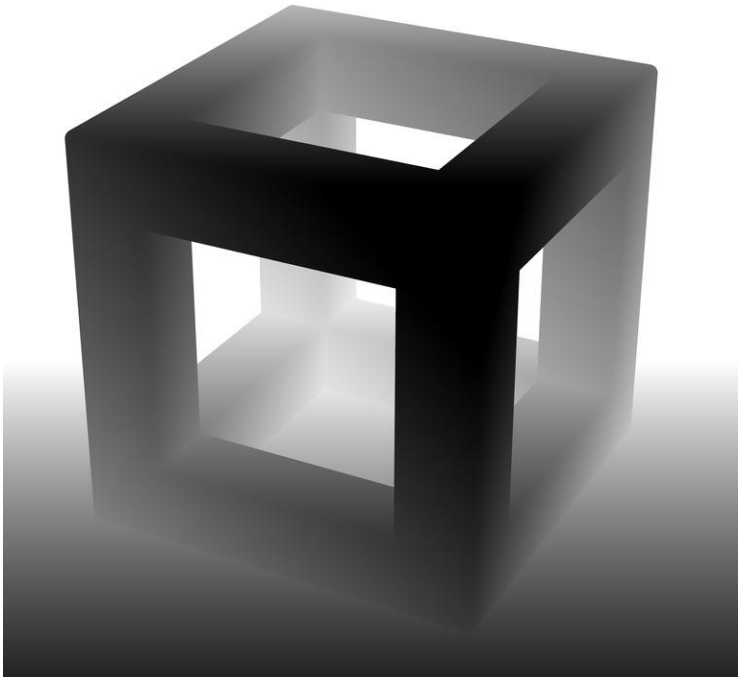
2.3.1 Vaizduojamojo tūrio atkirtimas

Kadangi objektai nepatenkantys į vaizduojamąjį tūri negali būti matomi, tai jiems atlikinėti matomumo testus nėra prasmės. Uždengtų objektų atkirtimo atveju vaizduojamojo tūrio atkirtimo metodo pritaikymas yra skirtas algoritmo optimizacijai, o ne atvaizdavimo spartinimui. Vaizduojamojo tūrio atkirtimas buvo nagrinėtas ankstesniame skyriuje (1.1.1).

2.3.2 Gylio žemėlapis

Gylio žemėlapis (14 pav.) nusako kiekvieno pikselio gylį – kaip toli tas pikselis yra nutolęs nuo žiūrėjimo taško. Gylio žemėlapio reikšmių aibė yra [0.0; 1.0], kur reikšmė 1.0 reiškia, kad taškas yra nutolęs didžiausią leidžiamą atstumą, o reikšmė 0.0 reiškia, kad objektas yra nutolęs mažiausią leidžiamą atstumą. Atstumų apribojimai yra nurodomi, nes kompiuteris negali atvaizduoti begalinės erdvės. Taigi nurodžius per didelius atstumus gausime mažą gylio žemėlapio tikslumą, o nurodę per mažus atstumus matysime nepakankamai toli. [Mic12a]

14 pav vaizde tamsesni pikseliai yra arčiau 0.0 reikšmės, taigi jie yra arčiau, o baltesni regionai yra toliau nuo žiūrėjimo taško, taigi tos reikšmės artėja prie 1.0.



14 pav. Gylio žemėlapis

Objektų atvaizdavimo metu kiekvieno naujo pikselio gylis yra lyginamas su seno pikselio gyliu, jeigu naujas pikselis yra arčiau už senąjį, tuomet senasis pikselis yra išmetamas, o naujasis išsaugomas, ir atvirkščiai – jeigu senasis pikselis yra arčiau už naują, tai naujas pikselis yra išmetamas. Iškarto pastebime, kad priklausomai nuo situacijos turėsime išmesti labai daug pikselių – daug atliktų skaičiavimų bus nepanaudoti. Tai ypač aktualu kai vienas objektas yra pilnai uždengiamas kito objekto. Tokiu atveju uždengtas objektas neduoda visiškai jokios naudos, bet naudoja GPU ir CPU resursus. Šią problemą turėtų spręsti nematomų objektų nustatymo metodai, o šiuo atveju – pasiūlytas algoritmas.

2.3.3 Praeito kadro gylio žemėlapio nuskaitymas

Gylio žemėlapio tekstūros sukūrimo metu (3 alg.) yra nustatomi keli šiam žingsniui aktualūs parametrai:

- Naudojimo būdas, šiuo atveju nustatyta D3D11_USAGE_DEFAULT, kas reiškia, kad GPU gali ir rašyti į tekstūra ir skaityti iš jos;
- CPU naudojimo būdas, šiuo atveju nustatyta 0, kas reiškia, kad CPU nenaudos tekstūros duomenų.

```
D3D11_TEXTURE2D_DESC depth_stencil_texture_desc = {};
depth_stencil_texture_desc.Width = width;
depth_stencil_texture_desc.Height = height;
depth_stencil_texture_desc.MipLevels = 1;
depth_stencil_texture_desc.ArraySize = 1;
depth_stencil_texture_desc.Format = DXGI_FORMAT_D32_FLOAT;
depth_stencil_texture_desc.SampleDesc.Count = 1;
depth_stencil_texture_desc.SampleDesc.Quality = 0;
depth_stencil_texture_desc.Usage = D3D11_USAGE_DEFAULT;
depth_stencil_texture_desc.BindFlags = D3D11_BIND_DEPTH_STENCIL;
depth_stencil_texture_desc.CPUAccessFlags = 0;
depth_stencil_texture_desc.MiscFlags = 0;
device->CreateTexture2D(&depth_stencil_texture_desc, nullptr,
&depth_stencil_texture);
```

3 alg. Gylio žemėlapio tekstūros sukūrimas (C++, DirectX 11)

Sukūrus gylio žemėlapio tekstūrą su tokiais parametrais bus pasiekta geriausia greitaveika atvaizdavimo metu, tačiau buvo nurodyta, kad CPU nenaudos tekstūros, taigi nėra įmanoma jos nuskaityti. Pirmasis bandymas spręsti šią problemą būtų nustatyti D3D11_CPU_ACCESS_READ reikšmę, tačiau šioje vietoje yra tam tikri apribojimai – jeigu CPU nori skaityti tekstūros duomenis, tuomet GPU negali nei rašyti nei skaityti iš tokios tekstūros, vienintelės leidžiamos operacijos yra kopijavimas [Mic12b].

Taigi norint nuskaityti gylio žemėlapį privalome sukurti antrą tekstūrą, kurią CPU gali skaityti, ir į ją nukopijuoti GPU naudojamą tekstūrą (4 alg.). Tereikia nustatyti kitokius GPU ir CPU naudojimo būdus, o kiti parametrai išlieka tokie patys.

```
depth_stencil_staging_texture_desc.Usage = D3D11_USAGE_STAGING;
depth_stencil_staging_texture_desc.BindFlags = 0;
depth_stencil_staging_texture_desc.CPUAccessFlags = D3D11_CPU_ACCESS_READ;
```

4 alg. Parametrai antrai tekstūrai

Tekstūros kopijavimas iš vienos į kitą taip pat yra paprastas:

```
context->CopyResource(depth_stencil_staging_texture, depth_stencil_texture);
```

Po šitos funkcijos kvietimo sistema nurodys GPU, kad reikia nukopijuoti tekstūrą. Kopijavimas bus atliekamas lygiagrečiai, taigi nuskaityti tekstūros iškarto negalėsime. Kaip anksčiau buvo minėta kopijavimą reikia pradėti kuo anksčiau, kad sistema turėtų pakankamai laiko kopijavo veiksmui atlikti. Paprašius duomenų anksčiau nei kopijavimas bus pabaigtas privers CPU laukti kol kopijavimas bus baigtas – CPU ir GPU sinchronizuosis.

Tačiau duomenų paėmimas iš tekstūros yra šiek tiek sudėtingesnis. Pirmiausia paprašome prieigos prie duomenų nurodydami, kad juos norime skaityti:

```
D3D11_MAPPED_SUBRESOURCE mapped_depth_stencil = {};
context->Map(depth_stencil_staging_texture, 0,
             D3D11_MAP_READ, 0, &mapped_depth_stencil);
```

Dabar duomenis galima pasiekti per `mapped_depth_stencil.pData` rodyklę. Tačiau čia reikia būti atidiems – dėl tam tikrų programinių ar techninės įrangos sprendimų kiekviena tekstūros eilutė gali turėti daugiau duomenų negu tikimės, taigi neteisingai juos kopijuodami gausime klaidingus rezultatus. Dvimatės tekstūros atveju, mus domina tik parametras `mapped_depth_stencil.RowPitch`, kuris nurodo eilutės dydį baitais. Įprastai mes tikimės, kad eilutės dydis baitais bus lygus pikseliui skaičiui padaugintam iš pikselio dydžio baitais (`width * sizeof(float)`), tačiau eilutės pabaigoje gali būti papildomų nenaudojamų baitų, kurie išlygiuoja sekančios eilutės adresą kompiuterio atmintyje. [Mic12c]

Taigi norint taisyklingai nukopijuoti duomenis privalėsime kopijuoti po vieną eilutę – nukopijuojame `width * sizeof(float)` baitų į mums reikiamą vietą, o `mapped_depth_stencil.pData` rodyklę paslenkame per `mapped_depth_stencil.RowPitch` baitų (5 alg.).

```
float* depth_values_row = depth_values;
uint8_t* mapped_depth_row = (uint8_t*)mapped_depth_stencil.pData;
for(int row = 0; row < height; ++row) {
	memcpy(depth_values_row, mapped_depth_row, width * sizeof(float));
	depth_values_row += width;
	mapped_depth_row += mapped_depth_stencil.RowPitch;
}
```

5 alg. Tekstūros kopijavimas (C++, DirectX 11)

Svarbu neužmiršti atlaisvinti prieigą prie duomenų:

```
context->Unmap(depth_stencil_staging_texture, 0);
```

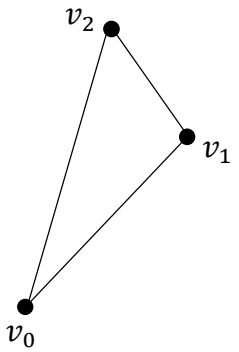
Turėdami gylio žemėlapį galime pradėti atlikinėti matomumo testus.

2.3.4 Trikampių rastrizacija

Kaip buvo minėta anksčiau, matomumo testas susiveda prie objekto atvaizdavimo. Atvaizduodami objektą lyginsime gaunamas gylio reikšmes su reikme esančia gylio tekstūroje. Jeigu bent vienas pikselis bus arčiau negu gylio žemėlapio tekstūroje, tai reikš, kad objektas yra matomas, ir jį reikia atvaizduoti, priešingu atveju turėsime patikrinti visus pikselius, kad galėtume teigti, jog objektas nėra matomas.

Kompiuterinės grafikos programose dauguma atvaizduojamų objektų yra sudaryti tik iš trikampių, taigi norėdami atlikti matomumo testus tereikia mokėti atvaizduoti trikampius.

Kaip visiems yra žinoma, trikampiai yra sudaryti iš 3 viršūnių v_0, v_1, v_2 ir 3 kraštinių v_0v_1, v_1v_2, v_2v_1 (15 pav.).



15 pav. Trikampis

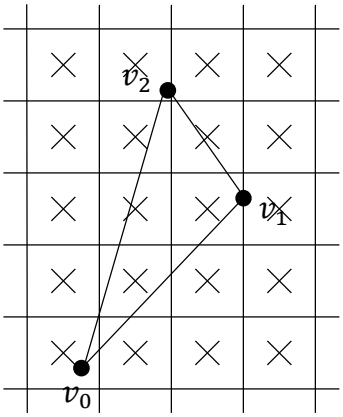
Trikampis dalija plokštumą į dvi dalis:

1. Dalis esanti trikampio viduje;
2. Dalis esanti už trikampio ribų.

Taigi norėdami rastrizuoti trikampį renkamės aibę taškų ir tikriname ar jie yra trikampio viduje:

- Jeigu yra, tai atvaizduojame tašką;
- Jeigu nėra, tai nieko nedarome.

Kadangi trikampius rastrizuosime į dvimatę tekstūrą, tai taškai bus parenkami fiksuotais intervalais, kurie atitiks tekstūros pikselius (16 pav.).



16 pav. Trikampis tekstūroje (X – pikselio centras)

Žinoma nereikia naiviai tikrinti visus tekstūros pikselius – trikampis gali būti labai mažas. Taigi iškarto vertėtų susimažinti tikrinamų pikselių skaičių aibę iki arti esančių. Paprasčiausias būdas būtų susirasti minimalias ir maksimalias koordinates visų 3 viršūnių – trikampiui priklausantys pikseliai gali būti tik šiame stačiakampyje. Bet kadangi trikampis gali būti didesnis už tekstūrą, tai reikia apkirpti stačiakampį iki tekstūros ribų.

Tai kaip visgi nustatyti, ar pikselis yra trikampio viduje?

Tarkime turime kraštinę v_0v_1. Jeigu tai būtų tiesė, einanti per viršūnes v_0v_1, tai ji dalintų plokštumą į dvi dalis: kairę (dar vadinama teigiama) ir dešinę (neigiama) puses. Jeigu tiesė būtų horizontali, tai būtų sudėtinga pasakyti kur yra kairė, o kur yra dešinė pusė. Galime įsivaizduoti taip – tarkime stovime taške v_0 ir einame taško v_1 kryptimi. Tuomet tiesės kairė pusė bus mūsų kairėje. [Gie13]

Taigi, turėdami trikampį $v_0v_1v_2$, bet koks taškas p bus trikampio viduje, jeigu jis bus teigiamose kraštinių v_0v_1, v_1v_2, v_2v_0 pusėse. Pastebime, kad jeigu trikampio viršūnės būtų atvirkščia tvarka ($v_2v_1v_0$), tai trikampio vidus būtų visų trijų kraštinių neigiamoje pusėje – tie patys taškai jau nebebus trikampio viduje. Taigi viršūnių tvarka taip pat yra svarbi. GPU rastrizacija yra atliekama tokiu pačiu principu. [Gie13]

Turint taškus a, b, c tereikia suskaičiuoti matricos determinantą ir patikrinti rezultato ženklą: jeigu rezultatas teigiamas, tai taškas c yra teigiamoje plokštumos pusėje, jeigu neigiamas – tai neigiamoje, rezultatas lygus 0 reikštų, kad taškas c yra ant tieses, kuri eina per taškus ab:

$$rez = \begin{vmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ 1 & 1 & 1 \end{vmatrix} = \begin{vmatrix} a_x & b_x - a_x & c_x - a_x \\ a_y & b_y - a_y & c_y - a_y \\ 1 & 0 & 0 \end{vmatrix} = \begin{vmatrix} b_x - a_x & c_x - a_x \\ b_y - a_y & c_y - a_y \end{vmatrix}$$

Pasirodo, kad įstate visas 3 trikampio viršūnes į aukščiau esančią matricą ir suskaičiave determinantą rezultate gausime trikampio plotą padaugintą iš 2., tai bus svarbu tolesniuose skaičiavimuose. [Gie13, Wei16]

Dabar žinome kaip nustatyti ar tekstūros pikseliai yra trikampio viduje ir šito pakanka, kad rastrizuoti individualius trikampius, tačiau prisiminkime ko mums reikia – įsitikinti, kad trikampis yra uždengtas kitų objektų. Kad tai galėtume nustatyti mums reikia žinoti atstumą iki kiekvieno trikampio pikselio. Pikseliai esantys trikampio viršūnėse bus nutolę tokiu pat atstumus kaip ir viršūnės, o kaip dėl pikselių esančių trikampio viduje? Pasirodo, kad tereikia tiesiškai interpoliuoti tarp trijų viršūnių. Belieka rasti interpoliacijos koeficientus. Šie koeficientai taip pat yra vadinami baricentrinėmis koordinatėmis kurias galima rasti pasinaudojus aukščiau esančia matrica. Normalizavus gautas reikšmes gausime normalizuotas baricentrines koordinates. Pasinaudojus gautomis reikšmėmis interpoliuojame tarp viršūnių gylio reikšmių ir gauname gylio reikšmę tam tikrame trikampio taške.

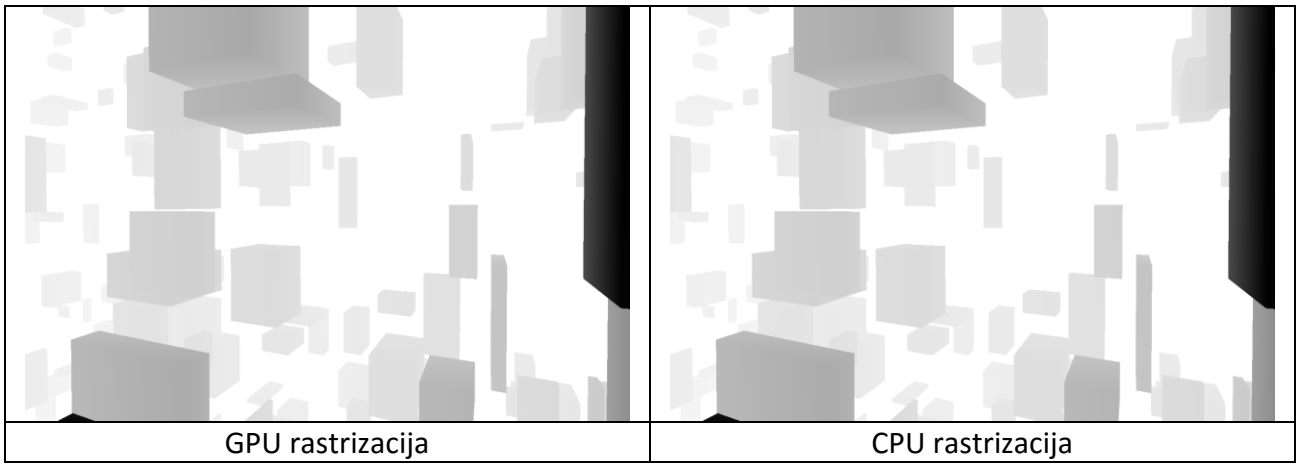
```
// determinanto skaičiavimo funkcija
int orient2d(const XMINT2& a, const XMINT2& b, const XMINT2& c) {
       return (b.x - a.x)*(c.y - a.y) - (b.y - a.y)*(c.x - a.x);
}

// normalizuotų baricentrinių koordinačių radimas
int w12 = orient2d(v1, v2, p);
int w20 = orient2d(v2, v0, p);
int w01 = orient2d(v0, v1, p);
float area2 = orient2d(v0, v1, v2);
float l0 = (float)w12 / area2;
float l1 = (float)w20 / area2;
float l2 = (float)w01 / area2;

// gylio reikšmės interpoliacija
float d = z[0] * l0 + z[1] * l1 + z[2] * l2;
```

6 alg. Algoritmas interpoliuotai gylio reikšmei surasti (C++)

Reikia taip pat prisiminti, kad gylio žemėlapio reikšmės yra tarp 0.0 ir 1.0, taigi norėdami gauti teisingus rezultatus turime apsiriboti tik šiuo rėžiu. Reikšmės nepatenkančios į šitą rėžį nusako, kad pikselis yra arba už kameros arba per daug toli nuo jos. Tuomet belieka patikrinti ar gauta reikšmė yra mažesnė už senąją, t.y. ar pikselis yra arčiau negu senasis. Tai reikštų, kad pikselis yra matomas, taigi bent vienas objekto pikselis bus matomas, taigi objektas yra matomas. Kadangi tai jau nustatėme galime išeiti iš funkcijos. Šiuo atveju gražiname reikšmę, kuri reiškia, kad trikampis nebuvo uždengtas:

```
if(d >= 0.0f && d <= 1.0f)
        if(depth_buffer[p.y * width + p.x] > d)
                return false;
```

Verta paminėti, kad jeigu vietoj testavimo tiesiog išsaugotume mažesnę reikšmę, tai atliktume tą patį, ką atlieka GPU rastrizavimas, ir turėtume beveik identiškus rezultatus (17 pav.).



GPU rastrizacija	CPU rastrizacija

17 pav. GPU ir CPU rastrizavimo palyginimas

Tačiau rezultatai nebūtų visiškai identiški. GPU naudoja griežtas taisykles tam, kad turint nesikertančius trikampius, bet turinčius sutampančias kraštines, kiekvienas pikselis būtų priskirtas tik vienam iš tų trikampių. CPU atveju tokių taisyklių nenaudojame, todėl tais atvejais, kai trikampių kraštinės eina tiesiai per pikselio centrą, pikselis pateks į abu trikampius. Taigi blogiausiu atveju visi trikampiai padidėja 1 pikseliu ir nustatysime daugiau matomų pikselių negu yra iš tikrųjų – tai yra leistina paklaida, nes galutinis vaizdas išlieka toks pat.

2.3.5 Rezultatų analizė

Šiame žingsnyje surenkame rezultatus gautus iš matomumo testų, kurie nurodo ar objektas yra matomas ar ne, ir pagal tai objektą atvaizduojame arba ne.

2.4 Projektinės dalies išvados

1. Siekiant sukurti pasiūlyto metodo algoritmą šioje dalyje buvo pristatytos uždengtų objektų atkirtimo metodų veikimo diagramos bei parodyta kokios modifikacijos buvo atliktos sprendžiant vieną ar kitą greitaveikos problemų. Bandydami spręsti metodų trūkumus pasiekėme savo tikslą – sukūrėme siūlomo uždengtų objektų atkirtimo metodo algoritmą;
2. Siekiant realizuoti pasiūlytą metodą buvo išanalizuota ir pateikta reikalinga informacija kiekvienam pasiūlyto metodo žingsniui realizuoti.

3 Eksperimentinio tyrimo dalis

Pasiūlytas metodas buvo realizuotas C++ programavimo kalba naudojant DirectX 11 grafinio programavimo sąsają. Realizavimui buvo panaudotas Intel korporacijos sukurtas programinio uždengtų objektų atkirtimo pavyzdys, kuris realizuoja 1.2.8 skyriuje aprašytą metodą. Šis pavyzdys yra labai gerai optimizuotas, taigi yra tinkamas eksperimentiniam tyrimui. Pavyzdys buvo modifikuotas ir realizuotas pasiūlytas metodas.

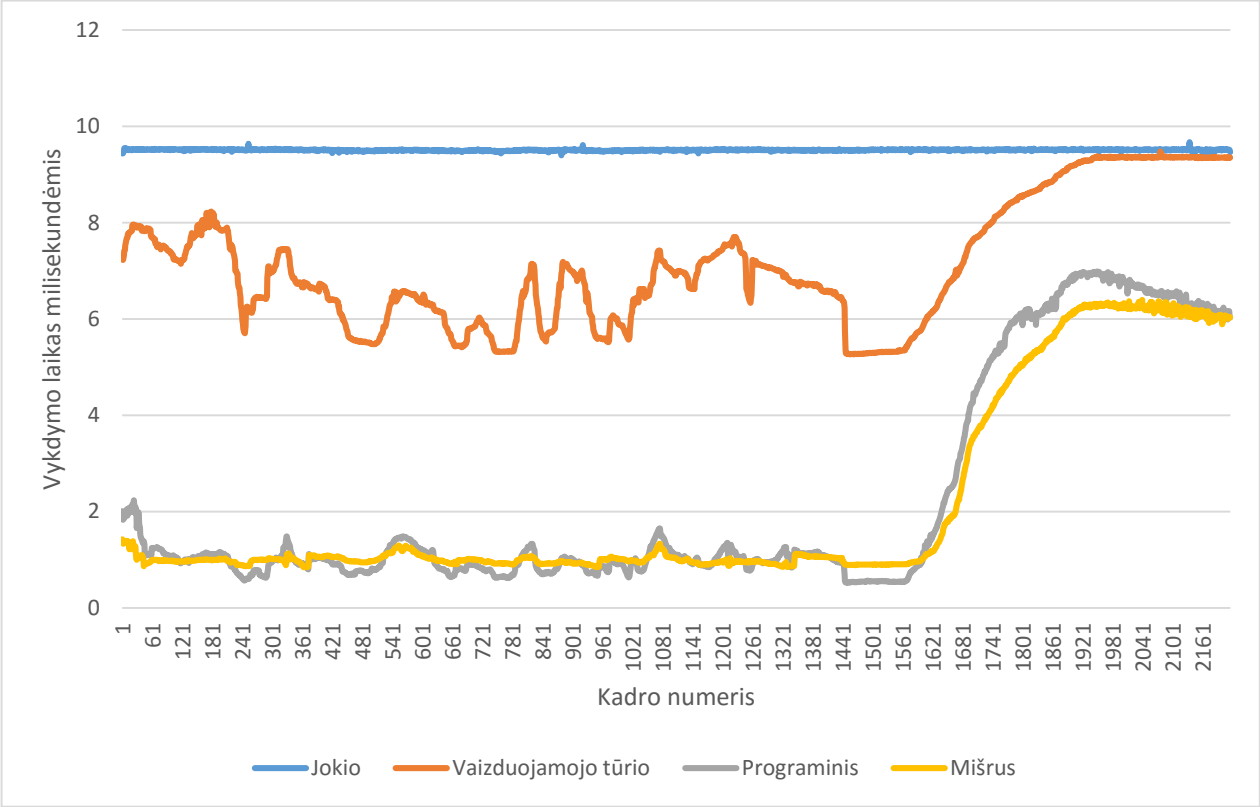
Eksperimentinis tyrimas buvo atliktas 4 skirtingais scenarijais:

1. Nenaudojant jokių atkirtimo metodų;
2. Naudojant vaizduojamojo tūrio atkirtimo metodą;
3. Naudojant programinį uždengtų objektų atkirtimo metodą;
4. Naudojant pasiūlytą mišrų uždengtų objektų atkirtimo metodą;

Eksperimentiniam tyrimui atlikti buvo parinkta 2216 skirtingų fiksuotų kameros padėčių. Tyrimo duomenys kiekvienam scenarijui buvo renkami tokiu būdu:

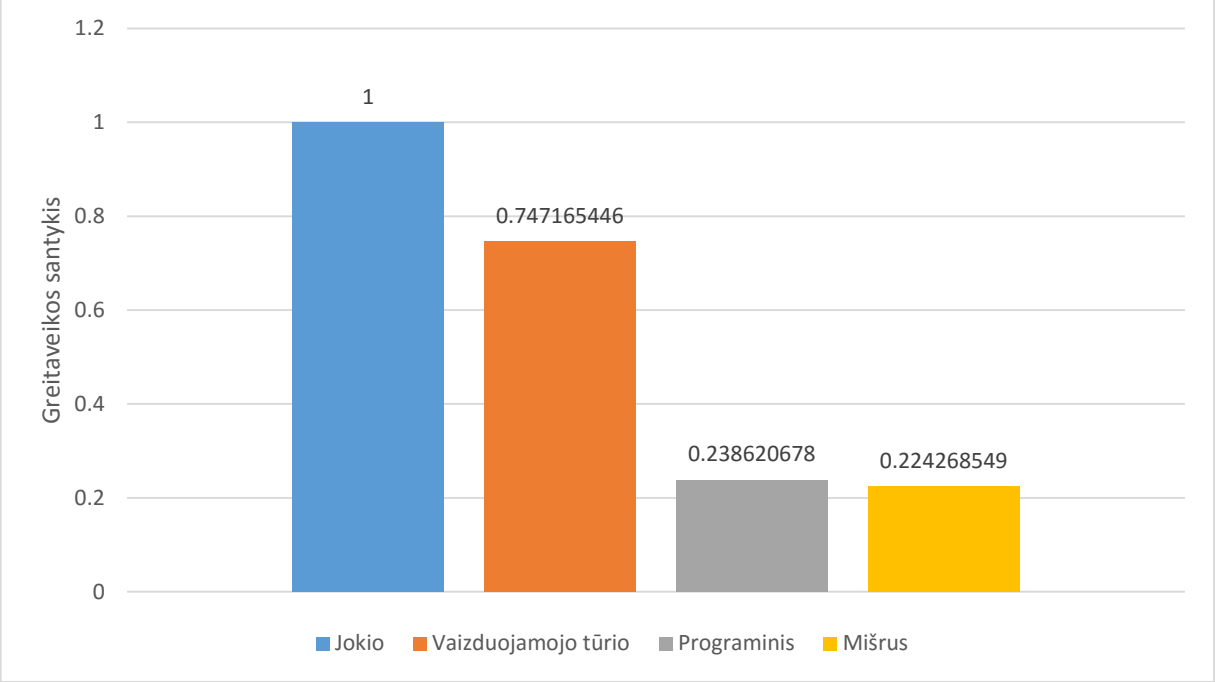
1. Kiekvienai kameros padėčiai atvaizduota 200 kadrų ir išmatuotas vykdymo laikas;
2. 100 blogiausiu rezultatų išmesti – daroma prielaida, kad fone dirbančios programos darė jiems įtaką;
3. Likusiems 100 rezultatų suskaičiavome vidurkį;
4. Surinkus visus 2216 vidurkių sudaroma greitaveikos diagrama;

Atlikus eksperimentinius tyrimus kiekvienu scenarijaus atveju ir apjungus visų keturių scenarijų rezultatus į vieną grafiką (18 pav.) galime palyginti gautus rezultatus:



18 pav. Sujungta greitaveikos diagrama

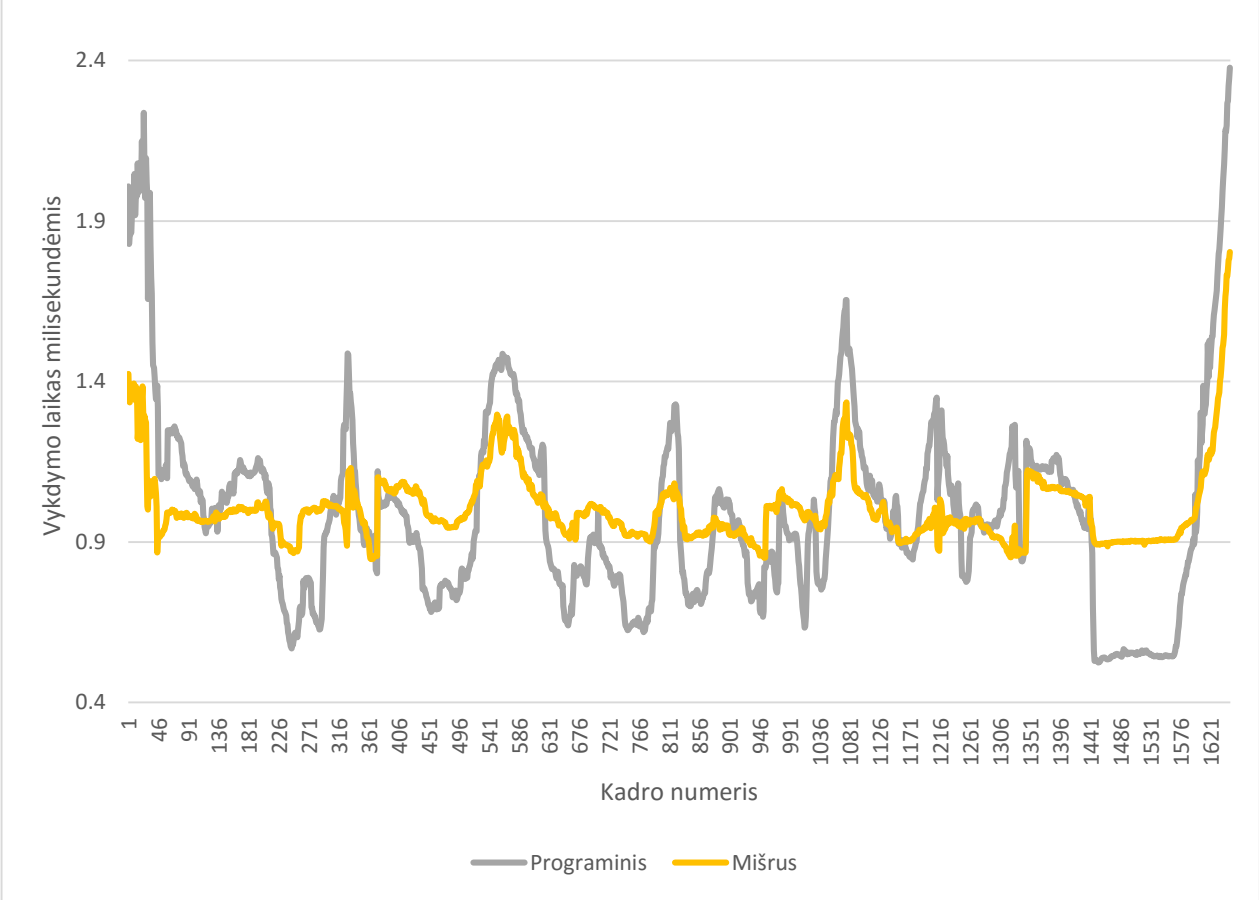
Iš greitaveikos diagramos matome, kad nenaudojant jokių atkirtimo metodų vykdymo laikas buvo labai didelis, ko ir galėjome tikėtis. Vaizduojamojo tūrio atkirtimo metodas šiek tiek pagerino greitaveiką – vidutiniškai pagerėjo 25% (žr. 19 pav.). Programinis ir mišrus metodai sutaupė virš 75% resursų ir vidutiniškai mišrus metodas buvo tik vienu procentiniu punktu pranašesnis.



19 pav. Greitaveikos palyginimas

Kadangi vaizduojamojo tūrio algoritmas yra labai lengvai realizuojamas, tai priklausomai nuo to, kur bus naudojamas vaizduojamojo tūrio atkirtimo metodas, tokių rezultatų gali pakakti. Svarbu neužmiršti, kad gaunami rezultatai yra labai priklausomi nuo duomenų, taigi jeigu scenoje nebus jokių didelių objektų, kurie gali uždengti kitus objektus, tuomet uždengtų objektų atkirtimo metodai bus beverčiai.

Kita vertus, programinis uždengtų objektų atkirtimo metodas ir mišrus uždengtų objektų atkirtimo metodas davė gana panašius rezultatus. 20 pav grafike šių metodų rezultatai yra parodyti iš arčiau.



20 pav. Sujungta greitaveikos diagrama

Žiūrint pirmus 1650 kadrų iš arčiau matome koreliaciją tarp gautų rezultatų, kuri, taip pat yra matoma ir vaizduojamojo tūrio atkirtimo atveju, kas nėra nejprasta. Beveik viso eksperimento metu pasiūlytas mišrus uždengtų objektų atkirtimo metodas rodė panašius rezultatus, priklausomai nuo kameros padėties tai vienas, tai kitas metodas pasirodydavo geriau. Mišrus metodas galėjo prarasti savo pranašumą dėl to, kad iš GPU paimtas gylio žemėlapis yra detalesnis, negu tas, kuris yra sukuriamas programinio metodo metu, todėl reikėjo testuoti daugiau objektų, kas sunaudojo papildomus resursus. Eksperimentinio tyrimo metu mišrus metodas, nors ir ne visada buvo pranašesnis, tačiau davė stabilesnius rezultatus negu programinis metodas (1 len.).

Metodas	**Dispersija**
Programinis	0.091305
Mišrus	0.0127826

1 len. Gautų rezultatų dispersijos

4 Rezultatai ir išvados

1. Atlikus literatūros analizę, išsiaiškinome, kad kompiuterinės grafikos trimačių vaizdų atvaizdavimo realiame laike spartai turi įtakos ir vaizduojamojo tūrio atvaizdavimo sparta. Norint paspartinti atvaizdavimą reikia optimizuoti atvaizdavimo procesą. Kompiuterinės grafikos programos, kaip ir žmogaus akis, atvaizduoja tik dalį visos scenos, o didelė dalis lieka nematoma. Objektai, esantys už vaizduojamojo tūrio ribų, nebus matomi, tačiau jų atvaizdavimas vis tiek naudoja resursus. Be to, vieni objektai gali būti uždengti kitų objektų, taigi jie taip pat bus nematomi, bet naudos kompiuterio resursus. Taigi norint paspartinti kompiuterinės grafikos programą reikia nustatyti ir atkirsti nematomus objektus.
2. Remiantis išanalizuota literatūra buvo pristatyti jau egzistuojantys metodai, taip pat pristatyta jų vystymosi istorija bei kiekvieno jų privalumai ir trūkumai.
3. Remiantis egzistuojančių metodų privalumų ir trūkumų pasiūlytas mišrus uždengtų objektų nustatymo metodas – naudojame gylio žemėlapį iš GPU, o testus atliekame programiškai.
4. Remiantis projektinės dalies analize realizuotas mišrus uždengtų objektų nustatymo metodas ir atliktas eksperimentinis tyrimas, o rezultatai palyginti su programinio metodo rezultatais, su vaizduojamojo tūrio atkirtimo metodo rezultatais, ir kai nenaudojamas joks metodas.
5. Atlikus atkirtimo metodų eksperimentinį tyrimą nustatyta:
    - Blogiausius rezultatus gavome tuo atveju, kai nenaudojome jokio atkirtimo metodo, ko ir galima buvo tikėtis;
    - Maždaug 25% geresnius rezultatus parodė vaizduojamojo tūrio atkirtimo metodas;
    - Programinis ir mišrus metodai parodė geriausius rezultatus, kurie buvo labai panašūs. Abu metodai sutaupė virš 75% resursų, tačiau mišraus metodo rezultatai rodo, kad jis yra tik 1% geresnis už programinį metodą. Tai yra gana nereikšmingas skirtumas.
    - Mišraus metodo greitaveika lyginant su programiniu metodu buvo labiau stabilesnė – mažesni greitaveikos svyravimai, taigi toks metodas yra labiau priimtinas naudojimui, negu programinis, kuris turės labai didelius greitaveikos skirtumus.

Literatūros sąrašas

[AHH08] Tomas Akenine-Möller, Eric Haines, Naty Hoffman. Real-Time Rendering, Third Edition. A.K. Peters Ltd.. 2008.

[Ari15] Fahad Arif. DirectX 12 Boosts Draw Calls By 330% On 3 Year Old GTX 670. 2015. [žiūrėta 2016-05-10]. Prieiga internetu: http://wccftech.com/directx-12-draw-calls-330-3-year-gtx-670/

[BWPP04] Jirí Bittner, Michael Wimmer, Harald Piringer, Werner Purgathofer. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. 2004. [žiūrėta 2016-05-10]. Prieiga internetu: http://www.cg.tuwien.ac.at/research/vr/chcull/bittner-eg04-chcull.pdf

[CCSD03] Daniel Cohen-Or, Yiorgos L. Chrysanthou, Cla´udio T. Silva, Fre´do Durand. A Survey of Visibility for Walkthrough Applications. 2003. [žiūrėta 2016-05-10]. Prieiga internetu: http://vgc.poly.edu/~csilva/papers/tvcg2003bcr.pdf

[Col11] Daniel Collin (DICE). Culling the Battlefield: Data Oriented Design in Practice. 2011. [žiūrėta 2016-05-10]. Prieiga internetu: http://www.frostbite.com/wp-content/uploads/2013/05/CullingTheBattlefield.pdf

[Eng13] Wolfgang Engel. GPU Pro 4: Advanced Rendering Techniques. CRC Press. 2013.

[Eri04] Christer Ericson. Real-Time Collision Detection, Volume 1. CRC Press, 2004.

[Gie13] Fabian Giesen. The barycentric conspiracy. 2013. [žiūrėta 2016-05-10]. Prieiga internetu: https://fgiesen.wordpress.com/2013/02/06/the-barycentric-conspirac/

[Gue11] Guerrilla Games. Practical Occlusion Culling in Killzone 3. 2011. [žiūrėta 2015-06-29]. Prieiga internetu: http://www.guerrilla-games.com/presentations/Siggraph2011_MichalValient_OcclusionInKillzone3.pdf

[Har11] Ben Hardwidge. The DirectX Performance Overhead. 2011. [žiūrėta 2016-05-10]. Prieiga internetu: http://www.bit-tech.net/hardware/graphics/2011/03/16/farewell-to-directx/2

[Int12] Intel Corporation. Software Occlusion Culling. 2012. [žiūrėta 2016-05-10]. Prieiga internetu: https://software.intel.com/sites/default/files/softwareocclusionculling.pdf

[Kir04] Andrew Kirmse. Game Programming Gems 4. Charles River Media. 2004.

[Len07] A. Lenkevičius. Grafika ir vizualizacija: mokymo medžiaga. Vilnius 2007. 194 p.

[Len11] Eric Lengyel. Mathematics for 3D Game Programming and Computer Graphics. Cengage Learning. 2011.

[MBW08] Oliver Mattausch, Jirí Bittner, Michael Wimmer. CHC++: Coherent Hierarchical Culling Revisited. 2008. [žiūrėta 2016-05-10]. Prieiga internetu: http://www-sop.inria.fr/reves/CrossmodPublic/uploads/files/EG08_CHC.pdf

[Mic12a] Microsoft. What Is a Depth Buffer?. 2012. [žiūrėta 2016-05-10]. Prieiga internetu : https://msdn.microsoft.com/en-us/library/bb976071

[Mic12b] Microsoft. D3D11_USAGE enumeration. 2012. [žiūrėta 2016-05-10]. Prieiga internetu: https://msdn.microsoft.com/en-us/library/ff476259

[Mic12c] Microsoft. D3D11_MAPPED_SUBRESOURCE structure. 2012. [žiūrėta 2016-05-10]. Prieiga internetu: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476182

[Sek04] Dean Sekulic. Efficient Occlusion Culling. 2004. [žiūrėta 2016-05-10]. Prieiga internetu: http://http.developer.nvidia.com/GPUGems/gpugems_ch29.html

[SJ02] Daniel Sykora, Josef Jelinek. Efficient View Frustum Culling. 2002. [žiūrėta 2016-05-10]. Prieiga internetu: http://www.cescg.org/CESCG-2002/DSykoraJJelinek/paper.pdf

[WB05] Michael Wimmer, Jirí Bittner. Hardware Occlusion Queries Made Useful. 2005. [žiūrėta 2016-05-10]. Prieiga internetu: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter06.html