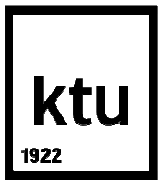


KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

Tadas Baskutis

TRIMAČIŲ OBJEKTŲ SANKIRTŲ NUSTATYMAS, NAUDOJANT CUDA

Baigiamasis magistro projektas

Vadovas

Lekt. dr. Kęstutis Jankauskas

KAUNAS, 2015

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

TRIMAČIŲ OBJEKTŲ SANKIRTŲ NUSTATYMAS, NAUDOJANT CUDA

Baigiamasis magistro projektas

Informatika (621I10003)

Vadovas

(parašas) Lekt. dr. Kęstutis Jankauskas

(data)

Recenzentas

(parašas)

(data)

Projektą atliko

(parašas) Tadas Baskutis

(data)

KAUNAS, 2015

KAUNO TECHNOLOGIJOS UNIVERSITETAS

Informatikos fakultetas

(Fakultetas)

Tadas Baskutis

(Studento vardas,pavardė)

Informatika (621I10003)

(Studijų programos pavadinimas, kodas)

Baigiamojo projekto „Trimačių objektų sankirtos nustatymas, naudojant CUDA“

AKADEMINIO SĄŽININGUMO DEKLARACIJA

20 15 m. gegužės 22 d.

Kaunas

Patvirtinu, kad mano, **Tado Baskučio**, baigiamasis projektas tema „Trimačių objektų sankirtų nustatymas, naudojant CUDA“ yra parašytas visiškai savarankiškai ir visi pateikti duomenys ar tyrimų rezultatai yra teisingi ir gauti sąžiningai.Šiame darbe nei viena dalis nėra plagijuota nuo jokių spausdintinių ar internetinių šaltinių, visos kitų šaltinių tiesioginės ir netiesioginės citatos nurodytos literatūros nuorodose. Įstatymų nenumatytų piniginių sumų už šį darbą niekam nesu mokėjęs.

Aš suprantu, kad išaiškėjus nesąžiningumo faktui, man bus taikomos nuobaudos, remiantis Kauno technologijos universitete galiojančia tvarka.

(vardą ir pavardę įrašyti ranka) *(parašas)*

Baskutis T. Trimačių objektų sankirtos nustatymas, naudojant CUDA. *Magistro* baigiamasis projektas / vadovas lekt. dr. Kęstutis Jankauskas; Kauno technologijos universitetas, Informatikos fakultetas.

Kaunas, 2015. 51 p.

Trimačių objektų sankirtų nustatymas, naudojant CUDA

SANTRAUKA

Objektų sankirtos nustatymas – aktualus uždavinys kompiuterinės grafikos, fizikinių dalelių sąveikos, bei robotikos uždaviniuose. Tikslus sankirtos tarp objektų nustatymas yra itin imlus skaičiavimams uždavinys, ypač kai objektai yra sudaryti iš didelės konstrukcinių elementų aibės ar pačių tarpusavyje sąveikaujančių objektų skaičius yra didelis. Todėl tyrimai, kaip paspartinti sankirtų nustatymą yra vykdomi ir iki šių dienų. Įžvelgiant tendenciją, kad CPU skaičiavimo pajėgumai dažnai yra nepakankami sudėtingų sankirtos nustatymo uždavinių sprendimui realiuoju laiku, vis daugiau dėmesio yra skiriama lygiagretiesiems skaičiavimams grafiniuose procesoriuose.

Šiame darbe yra tiriamas trikampių sankirtos nustatymo algoritmų efektyvumas, tarpusavyje lyginami erdvės skaidymo metodai, jų greitaveika, bei atminties sąnaudos, tiriamos algoritmų pritaikymo lygiagrečiam veikimui GPU galimybės, naudojant kelias skirtingų kartų Nvidia CUDA architektūros vaizdo plokštes. Taip pat apžvelgiamos egzistuojančios, bei siūlomos naujos klasikinių algoritmų modifikacijos.

Atlikus tyrimą nustatytas greičiausias trikampių sankirtos nustatymo algoritmas iš dviejų tarpusavyje lygintų variantų, nustatyta kurios kartos vaizdo plokštės geba lygiagrečiais skaičiavimais aplenkti nuoseklius CPU algoritmus, pateikiamos siūlomos tiesioginio kreipimosi modifikacijos nuosekliai veikiančių tolygaus tinklelio ir rekursinio aštuntainio medžio erdvės skaidymo metodų perkėlimui į GPU. Atliktas erdvės skaidymo metodų atskirų algoritmų fazių laikų palyginimas, pateikiamos įžvalgos apie metodų tinkamumą tam tikriems uždaviniams spręsti. Galiausiai palygintos greičiausiai veikiančio nuoseklaus CPU ir lygiagretaus GPU algoritmų greitaveikos.

RAKTINIAI ŽODŽIAI

Trikampių sankirta, CUDA, lygiagretusis programavimas

Intersection detection between 3D objects using CUDA

SUMARRY

Interection detection between objects is a relevant topic in computer graphics, physics based particle interaction systems and robotics. Precise intersection detection between objects is a very computationally expensive process, especialy when the objects are composed of a large set of constructional elements or there is a large number of interacting objects themselves. Thus, research of methods to speed up intersection detection is being carried out to this day. Observing a tendency, that the computational power of CPU is often not enough to perform complex intersection detection in real time, increasing attention is given to parallel computing in the GPU.

In this paper, research concerning triangle-triangle intersection algorithm efficiency is presented, multiple spatial subdivision methods are compared and evaluated based on their time and memory consumption. Also the parralelization of algorithms for GPU is examined, using Nvidia CUDA capable graphics cards of multiple generations. Lastly, existing modifications for classic intersection detection algorithms are analyzed and new ones suggested.

Research determines the fastest triangle detection algorithm out of two compared candidates, determines the generation of graphics cards whose parallel computing can outperform the CPU, presents direct access modifications to serial uniform grid and recursive octree algorithms for usage in the GPU. The research also compares the times of different phases of spatial subdivision algorithms, insight is presented about which methods are more suitable for which tasks. Finally, the research concludes with a time comparison of the fastest serial CPU algorithm and the fastest parallel GPU algorithm .

KEYWORDS

Triangle intersection, CUDA, parallel programming

TURINYS

LENTELIŲ SĄRAŠAS

PAVEIKSLŲ SĄRAŠAS

TERMINŲ IR SANTRUMPŲ ŽODYNAS

AABB (*angl. Axis-aligned Bounding Box*) tai minimali objektą apgaubianti stačiakampė dėžė kurios briaunos yra lygiagrečios koordinačių ašims.

Branduolys (*angl. Kernel*) tai kokius nors skaičiavimus atliekanti programuotojo parašyta funkcija kurią vykdo GPU. Viena branduolį lygiagrečiai gali vykdyti daugelis GPU gijų.

CPU (*angl. Central Processing Unit*) tai įtaisas vykdantis kompiuterinių programų instrukcijas, tokias kaip paprasta aritmetika, loginės bei įvesties/išvesties operacijos.

CUDA (*angl. Compute Unified Device Architecture*) tai lygiagrečių skaičiavimų platforma ir programavimo modelis skirtas firmos Nvidia grafiniams procesoriams.

GPU *(angl. Graphics processing unit)* grafinis procesorius, tai specializuotas procesorius pritaikytas greitam operacijų atmintyje atlikimui ir vaizdo generavimo spartinimui.

Gija *(angl. Thread)* instrukcijų seka, galinti būti vykdoma lygiagrečiai su kitomis instrukcijų sekomis.

1. ĮVADAS

Šiame tyrime nagrinėjamas objektų sankirtų nustatymas pasitelkiant CUDA technologiją. Įvade apžvelgiama temos probleminė sritis ir specifika, pagrindžiamas jos aktualumas, aprašomas tyrimo objektas, tyrimo tikslas, bei išvardinami uždaviniai, apžvelgiama viso dokumento struktūra.

1.1 Problemos aktualumas

Objektų sankirtos nustatymo uždavinys yra dažnai sutinkamas kompiuterinės grafikos, fizikinių dalelių sąveikos, robotikos uždaviniuose. Tikslus sankirtos tarp objektų nustatymas yra itin imlus skaičiavimams uždavinys, ypač kai objektai yra sudaryti iš pakankamai didelės konstrukcinių elementų aibės, arba kai pačių tarpusavyje sąveikaujančių objektų kiekis yra didelis. Norint nustatyti dviejų objektų sankirtą naudojant pilno perrinkimo (*angl. brute force*) metodą reikalingi $O(n^2)$ sudėtingumo skaičiavimai. Akivaizdu, kad realų pritaikymą turinčioje programoje objektų skaičius greičiausiai bus gerokai didesnis nei du ir gali siekti nuo kelių šimtų objektų iki šimtų tūkstančių objektų ar netgi daugiau, priklausomai nuo taikymų srities ir uždavinio. Trimatė scena taip pat gali būti dinamiška, objektai joje gali judėti, keisti savo išmatavimus ar kitus parametrus, gali keistis ir pačių objektų kiekis. Tokio dinamiško sąveikos skaičiavimo vykdymui realiu laiku, nepakanka sankirtos nustatymą atlikti tik vieną kartą, jis turi būti atliekamas tam tikrą kiekį kartų per laiko vienetą (sekundę). Kompiuterinių žaidimų srityje sankirtos nustatymas ir kiti fizikiniai skaičiavimai paprastai yra atliekami bent 20-30 kartų per sekundę. Be sankirtos nustatymo, dažniausiai dar yra vykdoma ir visa aibė kitų, ne ką mažiau sudėtingų skaičiavimų, su kuriais reikia dalintis turimais kompiuterio resursais ir skaičiavimo pajėgumu.

Siekiant efektyviau panaudoti kompiuterio resursus ir sumažinti atliekamų operacijų skaičių, buvo ir tebėra kuriami įvairūs metodai spartesniam sankirtos nustatymui. Dvi populiariausios metodų grupės yra gaubiantys tūriai ir erdvės skaidymas. Tiek vieni, tiek kiti, leidžia greičiau atmesti tikrai nesikertančius objektus naudojant tarpinius sankirtos nustatymus. Tačiau norint atlikti tikslų sankirtos tarp trimačių objektų nustatymą galiausiai tenka atlikti sankirtos nustatymą tarp objektus sudarančių trikampių. Egzistuoja įvairūs metodai ir algoritmai sankirtoms nustatyti, kurie yra sėkmingai taikomi ir šių dienų programose, tačiau išskirti vieno geriausio sprendimo visiems atvejams negalima. Dažnai metodai yra tarpusavyje kombinuojami ir pritaikomi konkrečiam atvejui. Problema išlieka aktuali todėl, kad su kiekviena nauja techninės įrangos karta, stengiamasi vis aukščiau pakelti ir trimatės grafikos kokybės kartelę, sukurti realistiškesnį virtualios aplinkos įspūdį. Tačiau tiesiog sukurti detalesnius trimačius modelius ir panaudoti naujesnę techninę įrangą nepakanka, būtina įgyvendinti vis augančioms duomenų apimtimis pritaikytus algoritmus. Galima įžvelgti tendenciją, kad net keleto branduolių CPU jau yra nepakankami daliai realaus laiko uždavinių spręsti ir todėl vis daugiau dėmesio skiriama grafiniams procesoriams, juose sprendžiami ne tik su kompiuterine grafika susiję uždaviniai bet ir bendro pobūdžio skaičiavimai.

Puikus pavyzdys kaip GPU leidžia išspręsti uždavinius, kuriems CPU galingumo nepakanka, yra pastaruoju metu aktyviai tyrinėjama sritis – realistiškos audeklo (*angl. cloth*) ar plaukų simuliacijos. Šiuo metu kompiuteriniuose žaidimuose dar galima sutikti plačiai paplitusius metodus, kuomet įvairūs audiniai ar veikėjų plaukai yra paprasčiausios statiškos ar iš anksto suanimuotos plokštumos. Tai rodo, kad CPU pajėgumas yra nepakankamas realistiškai simuliuoti audeklo ar plaukų judėjimą realiuoju laiku. Tačiau naudojant GPU, tikroviškas plaukų ir audeklo judėjimo apskaičiavimas, ankščiau sutinkamas nebent iš anksto sugeneruotose 3D animacijose, dabar gali būti atliekamos realiu laiku. Pasitelkiant sankirtos nustatymo algoritmus taip pat galima realizuoti dalelėmis grįstus skysčių, dūmų ir panašius efektus. Keičiantis technologijoms, keičiasi ir metodai, todėl tyrimai šioje srityje išlieka aktualūs.

Šiame tyrime nagrinėjama problema yra efektyvus sankirtos tarp trimačių objektų nustatymas, kuomet dėl didelio juos sudarančių trikampių kiekio nepakanka CPU pajėgumo. Problemos sprendimui pasitelkiamas vaizdo plokštės grafinis procesorius ir CUDA technologija.

1.2 Tyrimo sritis ir objektas

Potenciali tyrimo sritis šiuo atveju yra gana plati. Greitas sankirtos nustatymas yra itin aktuali sritis ne tik kompiuterinės grafikos ir realaus laiko virtualių sistemų srityje, bet ir fizikinių dalelių judėjimo uždaviniuose ar robotų trajektorijos planavime. Svarbus ne tik teorinis algoritmo modelis, bet ir jo pritaikymas konkrečiam uždaviniui ar aparatūrinei įrangai.

Tyrimo objektas – sankirtos nustatymo algoritmai, bei galimos įvairios jų modifikacijos, siekiant užtikrinti kuo didesnę greitaveiką, sprendžiant sankirtos nustatymo uždavinį tarp didelį konstrukcinių elementų kiekį turinčių trimačių objektų. Taip pat svarbu dėmesį skirti algoritmų išlygiagretinimui ir optimizavimui NVIDIA CUDA technologijai ir ją naudojančioms vaizdo plokštėms. CUDA suteikia ne tik dideles galimybes, bet ir iškelia naujus apribojimus, į kurių specifiką būtina įsigilinti, norint sudaryti efektyvų algoritmą.

1.3 Tikslas ir uždaviniai

Tyrimo tikslas yra palyginti CPU ir GPU sankirtos nustatymo algoritmų realizacijų efektyvumą. Tikslas yra išskaidytas į tokius uždavinius:

1. Palyginti keletą trikampių sankirtos nustatymo algoritmų ir išrinkti geriausią, kuris bus naudojamas tolesniuose eksperimentuose.
2. Įvertinti išlygiagretintų algoritmų pagreitėjimą lyginant su nuosekliais algoritmais.
3. Pasiūlyti algoritmų modifikacijas, įvertinti jas lyginant su klasikiniais algoritmais.
4. Palyginti kelių nuoseklių erdvės skaidymo metodų efektyvumą ir išrinkti geriausią.
5. Palyginti kelių lygiagrečių erdvės skaidymo metodų efektyvumą ir išrinkti geriausią.
6. Išskaidyti algoritmų laikų matavimą į atskiras fazes ir palyginti jų trukmę.
7. Tarpusavyje palyginti geriausią nuoseklų ir lygiagretų algoritmą.

1.4 Dokumento struktūra

Dokumente pateikiamas terminų ir santrumpų žodynas supažindina su dalykinės srities terminais. Įvado skyriuje apibūdinama tyrimo probleminė sritis, temos specifika, aptariami nagrinėjami metodai ir taikymo sritis, įvardijamas tyrimo tikslas bei jį sudarantys uždaviniai. Trikampių sankirtos nustatymo metodų analizės skyrius pateikia su tyrimu susijusių metodų analizę ir jų veikimo principus, literatūroje siūlomus sprendimus ir pateikiamus pastebėjimus. Siūlomų metodų specifikacijos skyrius pateikia susistemintą informaciją apie algoritmų įgyvendinimą. Pateikiamos UML diagramos iliustruojančius algoritmų veikimą. Rezultatų skyriuje yra pateikiami aparatūrinės ir programinės įrangos konfigūracijos parametrai, vartotojui siekiančiam pakartoti eksperimentus numatytomis aplinkybėmis. Taip pat pateikiami tyrimo metu atliktų eksperimentų gauti rezultatai. Išvadų skyriuje yra pateikiamos tyrimo uždavinių pagrindu suformuluotos ir rezultatų duomenimis paremtos išvados. Literatūros šaltinių sąrašas pateikia mokslinius straipsnius ir kitus šaltinius, kuriais remiantis buvo atliekamas šis tyrimas. Prieduose yra pateikiamas programos kodas vartotojams siekiantiems pakartoti tyrimo metu atliktus eksperimentus ar siekiantiems nuodugniau susipažinti su programos ir joje naudojamų algoritmų veikimo principais.

2. SANKIRTOS NUSTATYMO METODŲ ANALIZĖ

Šiame skyriuje apžvelgiamas sankirtos nustatymo uždavinys, nagrinėjami sankirtos nustatymo spartinimui naudojami gaubiančių tūrių ir erdvės skaidymo metodai bei jų variacijos. Taip pat, analizuojamas dviejų trikampių sankirtos nustatymo metodai. Apžvelgiamos lygiagrečių skaičiavimų pritaikymo uždavinio sprendimui galimybės, įvertinama galima jų įtaka greitaveikai ir atminties sąnaudoms. Galiausiai apžvelgiamos darbo priemonės, kurios bus naudojamos lygiagretiesiems skaičiavimams atlikti, t.y. nagrinėjama CUDA architektūra, bei apžvelgiamos optimizacijos kodo vykdymui grafiniame procesoriuje.

2.1 Sankirtos nustatymo uždavinys

Šiame darbe dėmesys bus skiriamas diskretaus tipo (*angl. discrete*) sankirtų nustatymui. Jo metu nagrinėjamas tik objektų kirtimasis tarpusavyje, t.y. sankirta yra nustatoma jau po jos įvykimo, kuomet objektai yra įsiskverbę vienas į kitą. Taigi, yra nustatomas tik pats objektų sankirtos faktas, o objektų atstatymas į taisyklingas pozicijas, kuriose jie nėra sankirtoje nebus nagrinėjamas.

Priešingas šiam tipui būtų tolydus arba nenutrūkstamas sankirtų nustatymas (*angl. continuous*), kuris apskaičiuoja objektų susidūrimus ir jų pasekmes dar prieš įvykstant susidūrimui. Tačiau šiame darbe jis nebus nagrinėjamas, nes dažniau yra sutinkamas fizikinių dalelių simuliacijose ir kolizijų nustatymuose. Tuo tarpu, šis darbas yra orientuotas ne į judančių objektų kolizijų nustatymą ir sąveiką tarpusavyje, bet į paties sankirtos fakto nustatymą tarp objektų turinčių sudėtingus trimačius modelius atvaizduojamus trikampių tinkleliais (*angl. triangle mesh*). Nepaisant to, abu minėtieji uždaviniai turi ir tam tikrų panašumų, ir viename ir kitame galima pritaikyti tuos pačius erdvės skaidymo (*angl. spatial subdivision*) principus bei paieškos optimizacijas.Daugelyje atvejų sankirtos nustatymo uždaviniui spręsti yra taikomas ne vienas konkretus algoritmas, o kelių algoritmų kombinacija [3]. Taip sankirtos nustatymas yra išskaidomas į tam tikrus lygmenis ar etapus, kiekviename kurių yra siaurinama duomenų apimtis ir rezultatai perduodami sekančiam etapui. Būtų galima išskirti šiuos etapus:

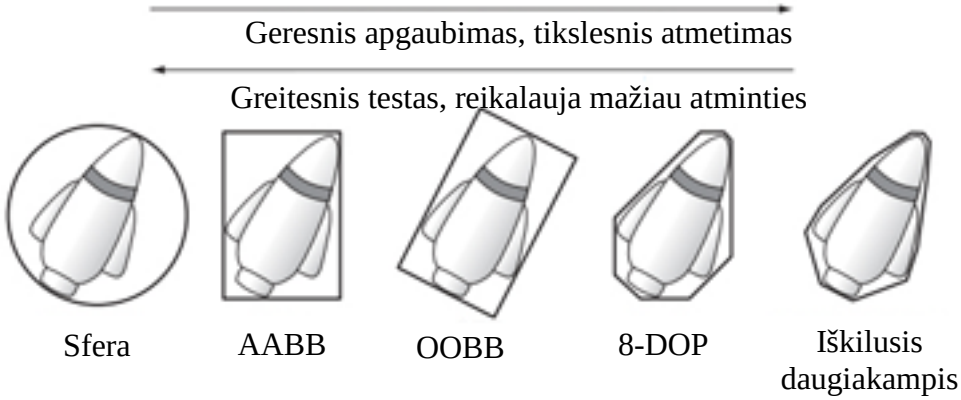
- Platusis etapas (*angl. broad phase*)
- Vidurinis etapas (*angl. mid-phase*)
- Siaurasis etapas (*angl. narrow phase*)

Itin paprastuose uždaviniuose gali būti pakankamas ir vienas lygmuo. Tačiau bendru atveju, stengiantis optimizuoti sankirtos nustatymo uždavinio greitaveiką, dažniausiai būna bent du lygmenys, vadinami plačiuoju ir siauruoju etapu. Plačiojo etapo metu dažniausiai yra pritaikomas erdvės padalinimas [1], kuris grąžina objektų patenkančių į tą pačią erdvės ląstelę poras. Po šio etapo gali sekti ir daugiau lygmenų vadinamų tarpiniais etapais, kurių kiekvienas tam tikrais metodais potencialiai gali dar labiau sumažinti galimai besikertančių objektų aibę, pašalinant atmestinus rezultatus (*angl. false positives*). Taip vieno lygmens gauti rezultatai yra perduodami sekančiam, kol yra pasiekiamas užsibrėžtas tikslumas. Be abejo, duomenų perdavimas tarp lygių taip pat užtrunka ir didelis kiekis tarpinių lygių gali būti pernelyg imlus resursams ir neatsipirkti. Galiausiai siaurojo etapo metu yra naudojami tiksliausi ir resursams imliausi skaičiavimai, kurių kiekį potencialiai ir buvo stengiamasi sumažinti praeituose etapuose. Vertėtų paminėti, kad siaurojo etapo metu nebūtinai turi būti atliekamas pilnai tikslus sankirtos nustatymas. Priklausomai nuo užsibrėžto tikslumo, šiame etape taip pat galima taikyti apytikslius sankirtos nustatymo algoritmus, naudojant gaubiančiuosius tūrius. Šiame darbe siaurojo etapo gražintus sankirtos nustatymo rezultatus laikysime galutiniais.

2.2 Gaubiantys tūriai

Vienas iš būdų paspartinti sankirtos aptikimo skaičiavimus – aproksimuoti objektus paprastesnėmis geometrinėmis figūromis. Kadangi gaubiantis tūris pilnai apgaubia visus objekto taškus, akivaizdu, jog nesikertant dviejų objektų gaubiantiems tūriams, tarpusavyje nesikirs ir patys objektai. Gaubiantys tūriai leidžia atlikti apytikslius testus, kurie padeda greitai atmesti tarpusavyje tikrai nesikertančius objektus, taip sumažinant objektų, kuriems reikia taikyti tikslias sankirtos nustatymo patikras, skaičių. Gaubiantys tūriai dažniausiai yra gana paprastos geometrinės figūros, todėl jų tarpusavio sankirtos nustatymas reikalauja mažiau aritmetinių operacijų, nei visų trikampių sudarančių objektus tarpusavio sankirtų tikrinimas. Keletas praktikoje paplitusių gaubiančių tūrių pavaizduoti žemiau esančiame paveikslėlyje. (žr. Pav. 2.1).

Kuomet nėra reikalingas itin didelis tikslumas, gaubiantys tūriai gali būti pakankami sankirtų nustatymui ir papildomai nereikia naudoti tikslesnių patikrinimų. Tai gan dažnai sutinkama kompiuteriniuose žaidimuose ieškant kompromiso tarp greitaveikos ir realistiškumo. Tačiau šio darbo metu neapsiribojama vien gaubiančių tūrių testų rezultatais, tiriama būtent ankščiau minėtoji perspektyva, jų pagalba sumažinti reikalingų tikslių patikrinimų skaičių.



Pav. 2.1 Sferos, AABB, OOBB, 8-DOP ir iškilaus daugiakampio palyginimas [5]

Nepaisant to, kad gaubiantys tūriai dažniausiai yra sąlyginai paprastos geometrinės figūros, ne visos geometrinės figūros yra efektyvūs gaubiantieji tūriai. Literatūroje yra išskiriama keletas savybių kuriomis turėtų pasižymėti gaubiantys tūriai [5]:

- Paprastas sankirtos patikrinimas
- Kuo tikslesnis sutapimas su apgaubto objekto kontūru
- Paprastas gaubiančio tūrio apskaičiavimas objektui
- Paprastos pasukimo ir postūmio operacijos
- Mažos atminties sąnaudos

Renkantis gaubiantį tūrį svarbu atkreipti dėmesį į tai, kad sankirtos nustatymo testas tarp dviejų tokių tūrių reikalautų kuo mažiau operacijų. Tačiau ne visada galima rinktis gaubiantį tūrį, kurio sankirtos patikrinimas yra greičiausias, taip pat reikia atsižvelgti ir į tai, kaip glaudžiai tūris gali apgaubti reikiamą objektą. Jei kartu apgaubiama ir daug tuščios erdvės, tai gali duoti per didelį kiekį klaidingų rezultatų tarpiniuose sankirtos nustatymo etapuose, todėl nukenčia greitaveika. Žinoma, skirtingiems objektams galima taikyti skirtingus gaubiančiuosius tūrius, tačiau tuomet reikės realizuoti kiekvieno iš tų tūrių sankirtos testus su visais kitais naudojamų tūrių tipais.

Nustatant sankirtas tarp judančių objektų, svarbu įvertinti gaubiančių tūrių poslinkio ir pasukimo operacijų greitaveiką. Sudėtingesnių gaubiančių tūrių, tokių kaip iškilieji apvalkai (*angl. Convex Hulls*) skaičiavimas objektams jau savaime gali būti išskiriamas kaip atskiras uždavinys, kurio sprendimą stengiamasi paspartinti panaudojant GPU [8]. Galiausiai renkantis gaubiančius tūrius reikia atsižvelgti ir į tai, kiek atminties reikės išskirti duomenų saugojimui. Keletas praktikoje paplitusių gaubiančių tūrių aprašyti žemiau esančioje lentelėje (žr. Lentelė 2.1).

Lentelė 2.1 Gaubiančių tūrių tipai

Gaubiantis tūris	Aprašymas
Sfera	Objektas apgaubiamas sfera. Idealiai tinka sferos tipo objektams, dalelėms.
Cilindras	Dažniausiai cilindro ašis yra fiksuojama pagal vertikalią scenos ašį.
Kapsulė	Cilindras su pussferėmis galuose. Dažnai taikomas apgaubti visą judantį veikėją kompiuteriniuose žaidimuose.
AABB	Stačiakampis gretasienis kurio kraštinės yra lygiagrečios koordinačių ašims.
OOBB	Orientuotas (pasuktas) pagal objektą stačiakampis gretasienis.
K-Dop	Apibendrintas AABB atvejis. Tai diskretus orientuotas politopas sudarytas iš K skaičiau plokštumų. Konstruojamas iš begalinių plokštumų kurios yra pasuktos patogiais skaičiavimams kampais ir artinamos iki objekto kol su juo susiliečia.
Iškilasis apvalkas	Mažiausias iškilusis apvalkas, į kurį telpa apgaubiamas objektas. Naudojamas objektams kuriuos būtų sunku aproksimuoti kitu primityviu tūriu.

2.3 Erdvės suskaidymas

Dar vienas efektyvus metodas paspartinti sankirtos aptikimą yra atmesti dalį tikrai nesikertančių objektų, panaudojant erdvės skaidymą (*angl. spatial partitioning*). Taikant erdvės skaidymą, remiantis tam tikromis taisyklėmis, visa erdvė yra padalinama į atskiras sritis, dažniausiai siekiant to, kad kiekvienoje srityje būtų ne daugiau kaip iš anksto apsibrėžtas skaičius objektų.

Kadangi visa erdvė yra padalinama į atskiras sritis, potencialiai tarpusavyje besikertantys objektai gali būti tik tose pačiose arba kaimyninėse srityse. Erdvės suskaidymui dažniausiai naudojami tinkleliai ar įvairios medžio tipo struktūros. Keleto populiariausių erdvės padalijimo metodų trumpi aprašymai yra pateikiami žemiau esančioje lentelėje (žr. Lentelė 2.2). Tolesniuose skyriuose plačiau panagrinėsime tinklelius ir aštuntainius medžius.

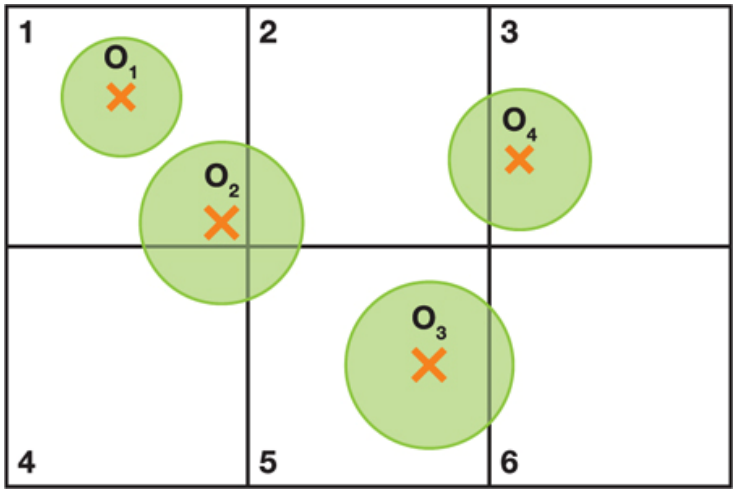
Lentelė 2.2 Erdvės suskaidymo metodai

Metodas	Aprašymas
Tinklelis	2D erdvė sudalijama į tolygaus dydžio langelius. Kaip ir kd medyje palaikomos efektyvios srities užklausos.
Dvejetainis medis	Dvejetainiai erdvės padalijimo medžiai rekursiškai padalina erdvę į atskiras sritis naudojant hiperplokštumas. Be sankirtos nustatymo taip pat plačiai naudojami ir objektų matomumo patikrai. Kd medžio, ketvirtainio medžio generalizacija.
Ketvirtainis/Aštuntainis medis	Ketvirtainis/Aštuntainis medis, tai erdvės padalijimo metodas, kai sričiai pasiekus tam tikrą vidinių elementų skaičių ji yra rekursiškai skaidoma į keturias/aštuonias dalis.
kd-medis	K dimensijų medis yra dvejetainis medis, kurio viršūnės yra k dimensijų taškai, kurių pagalba yra padalijama erdvė. Efektyvi struktūra kuomet reikia apdoroti ne vieno taško, bet tam tikros srities užklausas.
Gaubiančių tūrių hierarchijos	Gaubiančių tūrių hierarchija yra medžio tipo struktūra kurios lapuose patalpinama pradinė gaubiančių tūrių aibė, o medžio viršūnėse sukuriami gaubiantys tūriai dalinantys tą aibę į poaibius.

2.4 Tolygus tinklelis

Tolygus tinklelis (*angl. Uniform Grid*) skaido tam tikrą baigtinę erdvę į vienodo dydžio ląsteles. Kiekvienas scenoje esantis objektas yra priskiriamas vienai ar daugiau ląstelių, į kurias jis patenka. Norint atlikti sankirtos nustatymą tam tikram objektui, pakanka jį patikrinti su kitais tose pačiose ląstelėse esančiais objektais, nes akivaizdu, kad tik jie potencialiai gali su juo kirstis. Kadangi tinklelio ląstelių dydžiai yra vienodi, nustatyti kokioje ląstelėje randasi tam tikras taškas galima paprasčiausiai padalinant pasaulio koordinates iš ląstelės dydžio [5]. Kaimyninių ląstelių nustatymas taipogi yra trivialus.

Šio metodo problema yra tinkamo ląstelių dydžio parinkimas. Tinklelio dydis turi būti tinkamai parenkamas atsižvelgiant į objektų dydžius. Jei tinklelis bus per daug tankus, dideli objektai vienu metu papuls į daug ląstelių ir atliekant sankirtos nustatymą tam objektui, reikės jas visas patikrinti. Dar didesnė problema yra tuomet, kai objektas yra judantis. Tuomet pastoviai keičiasi ląstelės, į kurias jis patenka ir objekto-ląstelės ryšių atnaujinimams eikvojami skaičiavimų resursai. Jei tinklelis bus per stambus, į tą pačią ląstelę vienu metu gali papulti daug mažų objektų. Tikrinant vienam iš jų sankirtą, reikės atlikti patikrinimus su visais toje pačioje ląstelėje esančiais objektais, kai tuo tarpu metodo esmė ir yra tokių patikrinimų skaičių sumažinti. Blogiausiu atveju visi objektai gali papulti į vieną ląstelę. Parinkti tinklelio dydį yra lengva, kuomet visi objektai yra tokio pat ar gana panašaus dydžio, todėl šis metodas puikiai tinka fizikinių dalelių simuliacijoms [3]. Vertėtų paminėti, kad tam tikrais atvejais, kai vienu metu scenoje yra tiek didelių, tiek mažų objektų, parinkti teisingo tinklelio gali būti neįmanoma. Siekiant išspręsti tokias situacijas, kuomet yra sunku ar neįmanoma parinkti teisingą tinklelio tankumą, galima naudoti didelių objektų skaidymo į mažesnius metodus [1].

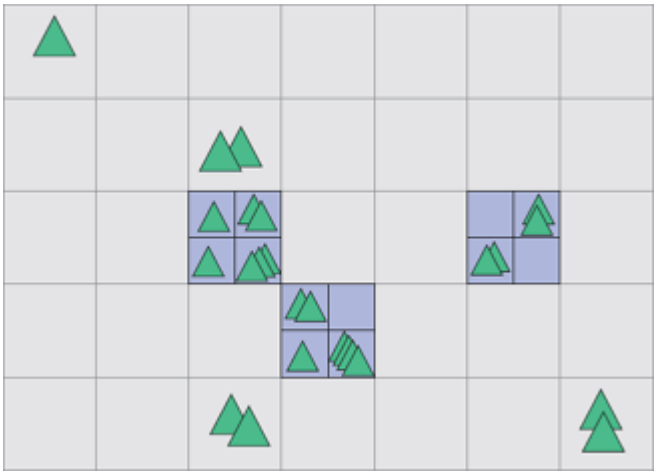


Pav. 2.2 Tolygus tinklelis 2D atveju [10]

Aukščiau esančiame paveikslėlyje galima išvysti tolygaus tinklelio variantą 2D erdvėje. (žr. Pav. 2.2). Šiuo atveju gaubiantys tūriai yra sferos. Vertėtų atkreipti dėmesį, kad sferos turi ryšį ne tik su tomis ląstelėmis kuriose yra jų centrai, tačiau ir su visomis kitomis ląstelėmis kurias jos kerta. Šiuo atveju gaubianti sfera O_2 vienu metu priklauso keturioms ląstelėms su kuriomis kertasi, taigi atliekant jai sankirtos nustatymą reikėtų ją tikrinti su visomis likusiomis sferomis, kadangi visos jos patenka bent į vieną iš minėtų keturių ląstelių. Taigi, šiuo atveju skaičiavimų nėra sutaupoma. Tačiau jeigu paimsime sferą O_1 galėsime matyti, kad nustatinėjant sankirtas jai, tolygus tinklelis atsiperka, tereikia atlikti vieną sankirtos testą su toje pačioje celėje esančia sfera O_2.

2.5 Hierarchinis tinklelis

Siekiant išspręsti ankstesniame skyrelyje aptarto tolygaus tinklelio celės dydžio parinkimo problematiką, kylančią dėl skirtingo objektų dydžio ir galimo jų judėjimo, galima taikyti kitokio tipo erdvės padalijimą vadinamą hierarchiniu tinkleliu (*angl. Hierarchical Grid*). Hierarchinio tinklelio atveju erdvę gali dalinti keli persidengiantys tarpusavyje skirtingo ląstelių dydžio tinkleliai surikiuoti ląstelių didėjimo tvarka. Iš esmės galima laikyti, kad hierarchinis tinklelis yra sudarytas iš tam tikro skaičiaus surikiuotų tolygių tinklelių. Hierarchinio tinklelio privalumas yra tas, kad jis yra tinkamas judantiems objektams saugoti, jame įgyvendinamos greitos įterpimo ir pašalinimo operacijos. Tinklelio struktūros atnaujinimui gali būti pasitelkti vadinamieji atidėliojantys (*angl. deferred*) metodai [2], kuomet remiantis euristika ir nuspėjant kiek toli per tam tikrą laiko tarpą realiai gali pajudėti objektai yra atnaujinamos tik tam tikros tinklelio sritys. Kuomet yra pasiekiamas tam tikras šių euristinių atnaujinimų slenkstis (*angl. threshold*) tik tuomet yra įvykdomas pilnas hierarchinio tinklelio atnaujinimas. Žinoma, norint sėkmingai taikyti euristiką, reikia turėti tam tikros papildomos informacijos apie objektus ir atitinkamai vykdyti paties hierarchinio tinklelio formavimą.



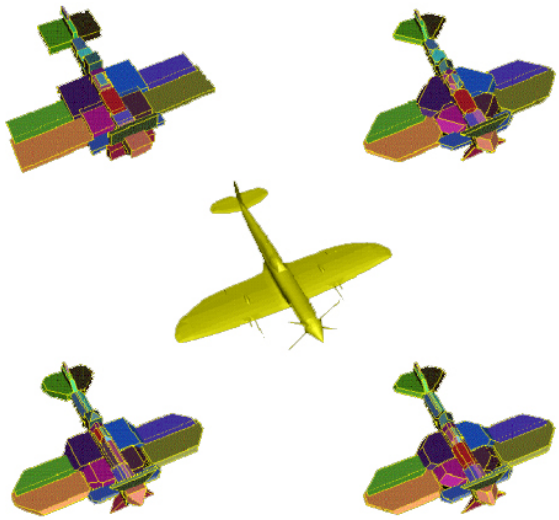
Pav. 2.3 Dviejų lygių hierarchinis tinklelis [2]

Kaip pastebima literatūroje [5], hierarchinis tinklelis iš esmės yra panaši į medį struktūra, tačiau veikia greičiau, kadangi jame yra atsisakoma viršutinių medžio lygmenų, į kuriuos talpinami objektai dažniausiai nepakliūna. Taip pat geresnę greitaveiką dinamiškų objektų atveju padeda užtikrinti ir ankščiau minėtos atidėliojančių ir euristinių metodų pritaikymo galimybės.

Skirtingai nei tolygaus tinklelio atveju, hierarchiniame tinklelyje elementai vienu metu yra dedami ir papuola tik į vieną ląstelę. Tai pasiekiama pasinaudojant tuo, kad iš visų hierarchinio tinklelio lygių yra parenkamas toks lygis kuriame ląstelės dydis yra pakankamas objektui patalpinti taip jog šis pakliūtų tik į tą vieną ląstelę, ir nesikirstų su jos kaimynėmis celėmis. Iš to galima daryti išvadą, kad hierarchinio tinklelio lygių ląstelių dydžius reikia parinkti pagal į jį dedamus objektus, t.y. žemutinis hierarchinio tinklelio lygis turėtų efektyviai talpinti mažiausią galimą objektą, o viršutinis tinklelio lygis turėtų būti pakankamai didelis, kad savyje patalpinti didžiausią galimą objektą. Viduriniai lygmenys savo ruožtu galėtų talpinti tarpinio dydžio objektus, tačiau tai nėra būtina, nes esant didelei objektų dydžių įvairovei, kiekvienam iš jų skirti atskirą lygį gali būti neefektyvu.

2.6 Gaubiančių tūrių hierarchijos

Kai yra reikalingas didesnis tikslumas, objektą galima apgaubti ne vienu konkrečiu tūriu, bet skirtingų gaubiančių tūrių tipų kombinacija, dar kitaip vadinama gaubiančių tūrių hierarchija (*angl. Bounding volume hierarchy*). Taikant tokį metodą, objektas faktiškai yra aproksimuojamas naudojant primityvias geometrines formas. Akivaizdu, kad norint tiksliau aproksimuoti sudėtingus objektus, bus reikalingas vis didesnis gaubiančių tūrių, taigi ir jiems saugoti reikalingos atminties kiekis ir nukentės greitaveika, tad šioje vietoje svarbu rasti tinkamą balansą tarp siekiamo aproksimacijos tikslumo ir pageidaujamos greitaveikos bei resursų sąnaudų. Žemiau esančiame paveikslėlyje galima išvysti iš k-Dop [11] gaubiančių tūrių suformuotas gaubiančių tūrių hierarchijas vienam konkrečiam objektui. (žr. Pav. 2.4).



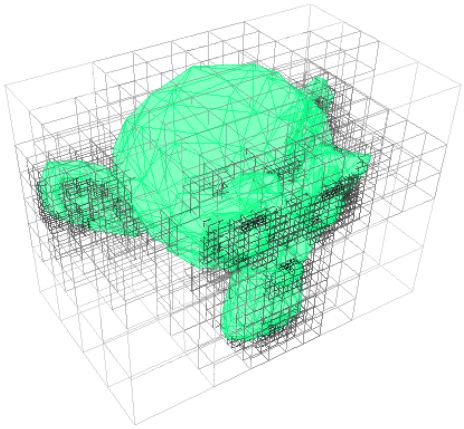
Pav. 2.4 Gaubiančių tūrių hierarchijos pavyzdžiai [11]

Gaubiančių tūrių hierarchijas galima panaudoti nebūtinai vieną objektą aproksimuojantiems gaubiantiems tūriams. Kadangi patys gaubiantys tūriai savaime nesumažina scenoje esančių objektų, kuriems reikia atlikti sankirtos nustatymą kiekio, gaubiančių tūrių hierarchiją galima panaudoti ir kaip erdvės padalijimo metodą [4], į ją sudedant tarpusavyje nesusijusius gaubiančius tūrius. Tokiu atveju gaubiančių tūrių hierarchijos panaudojimas padeda išgauti logaritminį reikalingų atlikti sankirtos nustatymų skaičių. Be abejo, tokiu atveju kai į gaubiančių tūrių hierarchiją sudedami judantys objektai, arba hierarchija yra pritaikoma objektui kuris gali deformuotis, gaubiančių tūrių hierarchiją kiekviename simuliacijos žingsnyje gali tekti dalinai atnaujinti ar sugeneruoti iš naujo.

Gaubiančių tūrių hierarchijos formavimui galima pasitelkti keletą metodų: iš apačios į viršų (*angl. bottom-up*), iš viršaus į apačią (*angl. top-bottom*) ir įterpimo (*angl. insertion*) metodai [5]. Formuojant gaubiančių tūrių hierarchiją yra formuojama medžio struktūra. Pradinė gaubiančių tūrių aibė tampa šio medžio lapais. Tuo tarpu medžio viršūnės grupuoja pradinės gaubiančių tūrių aibės elementus į poaibius ir apgaubia juos didesniais gaubiančiais tūriais. Analogiškai ir aukštesnieji medžio lygmenys gaubia esančius žemiau, kol galiausiai medžio šaknis yra vienas didelis gaubiantis tūris, į kurį papuola visi pradinės aibės gaubiantieji tūriai. Skirtingai nuo erdvės padalijimo tolygiu tinkleliu metodo, į gaubiančių tūrių hierarchiją elementai yra talpinami tik vieną kartą, dedant juos į medžio lapus. Kai, tuo tarpu, tolygaus tinklelio atveju, vienas elementas gali turėti ryšius su keliomis ar, blogiausiu atveju, visomis tinklelio ląstelėmis.

2.7 Aštuntainis medis

Aštuntainis medis (*angl. octree*), tai hierarchinis erdvės skaidymo metodas, kuomet erdvė yra sudalinama ašims orientuotais gaubiančiais stačiakampiais, dar vadinamais oktantais, tarpusavyje susietais tėvų–vaikų ryšiais. Aštuntainio medžio šaknis yra minimalus ašims orientuotas gaubiantis stačiakampis, į kurį telpa visi norimi patalpinti į medį duomenys. Šiuo atveju duomenys yra trikampiai, sudarantys trimatį modelį. Jei trikampių skaičius oktante viršija tam tikrą užsibrėžtą ribą, tas oktantas yra skaidomas į 8 naujus oktantus, tol, kol į oktantą papuolančių trikampių skaičius neviršys užsibrėžtos ribos arba bus pasiektas maksimalus leistinas medžio gylis. Dar viena galima sąlyga oktantų skaidymo sustabdymui yra minimalus pageidaujamas oktanto dydis. Žemiau esančiame Pav. 2.5 galime matyti trimatį objektą ir jo aštuntainį medį.



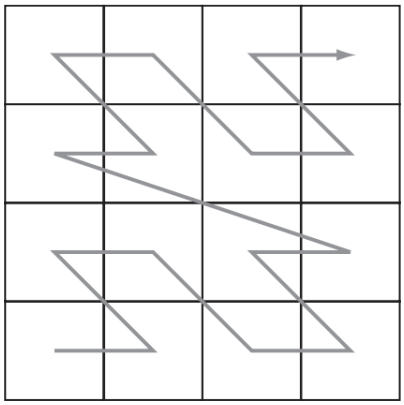
Pav. 2.5 Trimatis objektas ir jį talpinantis aštuntainis medis

Pirmasis medžio oktantas arba šaknis apgaubia visus žemesniuose lygiuose esančius oktantus, o visi likusieji medžio lygiai pilnai apgaubia žemiau esančiuosius. Tačiau ar persidengia tėvinių ir vaikinių oktantų skaidoma erdvė, priklauso nuo realizacijos ir pagal poreikius užsibrėžtų įvairių objektų sudėjimo į aštuntainį medį taisyklių. Aštuntainį medį galima realizuoti taip, kad visi į jį patalpinti trikampiai būtų tik vaikiniuose oktantuose, šiuo atveju galimos tokios situacijos, kad vienas trikampis atsidurs ant kelių oktantų ribos ir vienu metu papuls į keletą iš jų. Tai gali padidinti potencialių sankirtos tikrinimų skaičių. Kitas variantas, kiekvieną trikampį patalpinti į tokį oktantą, į kurį jis pilnai tilptų. Tačiau ir šiuo atveju galimi pertekliniai tikrinimai, nes sąlyginai didelis trikampis arba trikampis esantis trijų skaidymo ašių sankirtoje gali atsidurti medžio šaknyje ir turėtų būti atliekamas sankirtų nustatymas tarp jo ir visų likusių trikampių. Taip pat galimos įvairios kitos įterpimo į medį taisyklės, pvz.: įterpti trikampį į tokį medžio gylį, kad oktantų, į kuriuos jis papuola skaičius neviršytų tam tikros ribos. Yra įvairių kitų aštuntainio medžio variacijų, tokių, kaip pilnas aštuntainis medis, kurio visi oktantai yra skaidomi iki maksimalaus gylio nepriklausomai nuo duomenų kiekio, nereguliarus aštuntainis medis kurio tėviniai oktantai gali turėti nevienodo dydžio vaikinius oktantus ir t.t [5].

Taip pat egzistuoja ir skirtingi medžio formavimo metodai. Aukščiau esančiuose pastraipose laikoma, kad medis yra formuojamas įterpimo metodu (*angl. insertion*), t.y. į medį įterpiant po vieną trikampį, parenkant vieną ar kelis medžio oktantus, į kuriuos jis papuls. Taip pat medį galima formuoti iš viršaus į apačią (*angl. top-down*), iš pradžių visus trikampius įterpiant į medžio šaknį ir toliau skaidant medį tol, kol bus patenkintos užsibrėžtos anksčiau minėtos sąlygos. Dar vienas galimas variantas yra formuoti medį iš apačios į viršų (*angl. bottom-up*), sudedant visus trikampius į žemiausią galimą medžio lygį ir vėliau juos apjungiant į aukštesnius lygmenis, jei yra patenkinamos užsibrėžtos sąlygos.

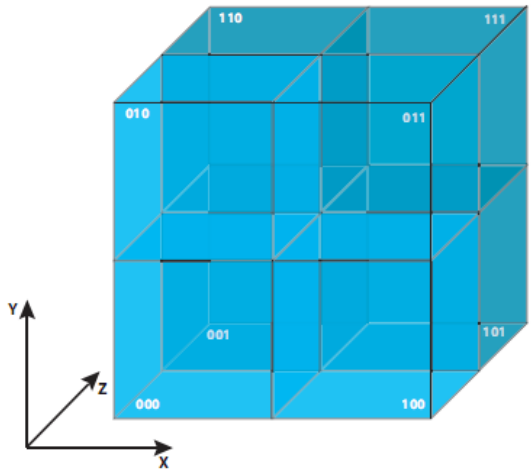
2.8 Tiesinis aštuntainis medis

Viena iš aštuntainio medžio variacijų yra tiesinis aštuntainis medis (*angl. Linear octree*). Pati medžio idėja yra tokia pati, kaip ir įprasto aštuntainio medžio, tačiau skiriasi medį atvaizduojančių duomenų pavidalas, visi tiesinio aštuntainio medžio oktantai yra nuosekliai tiesiškai surašyti į masyvą, nėra naudojamos nuorodų į kitus oktantus rodyklės ir rekursija. Tvarka, kuria medžio oktantai yra surašomi į masyvą, gali būti įvairi. Tiek literatūroje, tiek praktikoje dažniausiai sutinkama tvarka vadinama Mortono arba Z-tvarka, pavaizduota žemiau esančiame Pav. 2.6. Mortono tvarka suprojektuoja keleto dimensijų duomenis į vienmatę tvarką [19].



Pav. 2.6 Mortono tvarka [5]

Mortono tvarka dar yra vadinama Mortono kodais, kurie ir atskleidžia kodėl būtent šis elementų surašymo būdas yra palankus. Įprastame aštuntainiame medyje, tėvinis oktantas saugo rodykles į vaikinius oktantus, tačiau tiesiniame aštuntainiame medyje, kiekvienam oktantui pakanka žinoti savo indekso masyve numerį, kurį ir nurodo Mortono kodas, ir pagal jį, reikalui esant, pasiskaičiuoti savo vaikų indeksus. Vieno lygio aštuntainio medžio vaikams identifikuoti Mortono kodu, reikės 3 bitų (po vieną visoms trims ašims, trimačiu atveju). Mortono kodas apskaičiuojamas paprastai: oktantų, kurių centro x koordinatė mažesnė už jų tėvo centro x koordinatę, kodo pirmasis bitas bus lygus nuliui, kurių didesnė – lygus vienetui. Analogiškai antrojo ir trečiojo bito reikšmes lems atitinkamai y ir z koordinatės. Žemiau esančiame Pav. 2.7 pavaizduoti visų 8 oktantų Mortono kodai dvejetainiu pavidalu. Šiuos dvejetainius pavidalus pavertę į dešimtainius skaičius gautume masyvo indeksus nuo 0 iki 7. Aštuntainio medžio oktanto Mortono kodui reikalingų bitų skaičius yra lygus aštuntainio medžio lygių skaičiui padaugintam iš trijų.



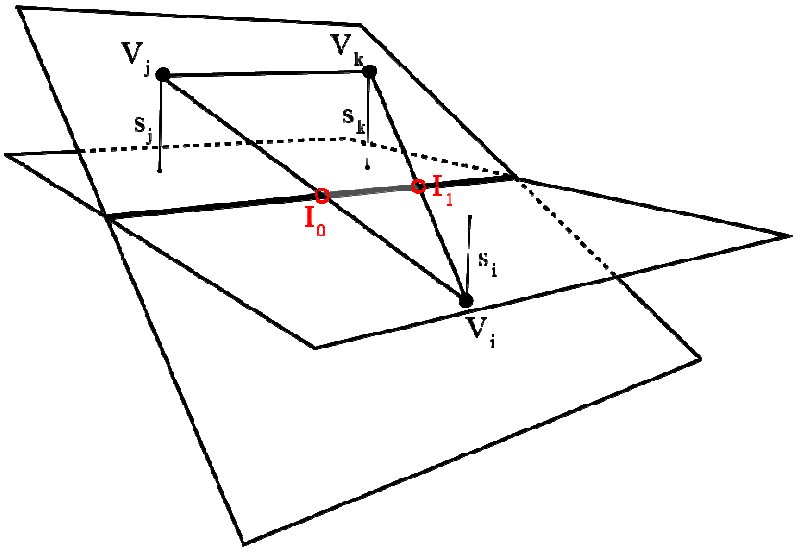
Pav. 2.7 Oktantų Mortono kodai [20]

2.8.1 Sankirtos tarp trikampių nustatymas

Sankirtos tarp trikampių nustatymas paremtas Möller trikampių sankirtos nustatymo algoritmu [17], su keliomis modifikacijomis ir optimizacijomis iš [16]. Metodo pradiniai duomenys yra abiejų trikampių viršūnių koordinatės, atitinkamai V_1, V_2, V_3 ir U_1, U_2, U_3. Naudojantis šiais pradiniais duomenimis abiem trikampiams yra suskaičiuojama: po 3 briaunų vektorius, atitinkamai v_1, v_2, v_3 ir u_1, u_2, u_3;po 1 normalės vektorių,n_1 ir n_2 ir atstumas iki koordinačių pradžios taško, d_1 ir d_2. Taip pat yra suskaičiuojamas trikampių plokštumų sankirtos tiesės vektorius m ir randama didžiausia absoliutinė jo reikšmė x [17].

2.8.2 Trikampio ir plokštumos sankirtos

Siekiant užtikrinti greitesnį algoritmo veikimą, tais atvejais kai trikampiai nesikerta, pirmiausia yra tikrinama ne pačių trikampių, o jų plokštumų sankirta. Tam yra atliekami du sankirtos testai: pirmojo trikampio su antrojo trikampio plokštuma ir antrojo trikampio su pirmojo trikampio plokštuma. Abu testai veikia tokiu pačiu principu, apskaičiuojant vieno iš trikampių viršūnių atstumus (atitinkamai s_1, s_2, s_3 ir q_1, q_2, q_3) iki kito trikampio plokštumos. Jei šių atstumų ženklai sutampa, vadinasi visos trikampio viršūnės, taigi ir jis pats, yra vienoje iš plokštumos pusių ir jos nekerta. Jei visi atstumai yra lygus nuliui, vadinasi trikampiai yra vienoje plokštumoje. Jei vienas ženklas skiriasi, vadinasi trikampis kerta plokštumą, tą iliustruoja Pav. 2.8. Viršūnės, kurios atstumo ženklas nesutampa su kitų atstumų ženklais, indeksas pažymimas i, o likusių dviejų j, k. Analogiškai antrojo trikampio viršūnių indeksai pažymimi a ir b, c.



Pav. 2.8 Trikampių plokštumų sankirta, paveikslėlis adaptuotas iš [17]

2.8.3 Intervalų tikrinimas

Įsitikinus, kad trikampių plokštumos kertasi, reikia surasti jų sankirtos tiesės intervalus priklausančius abiem trikampiams, t.y. taškus kuriuose trikampių kraštinės kerta sankirtos tiesę. Pavadinkime šiuos taškus I_0, I_1ir J_0, J_1, atitinkamai pirmąjam ir antrąjam trikampiui.Norėdami rasti tašką I_0 trimatėje erdvėje, galime pasinaudoti taško atkarpoje aprašymo formule:

$$I_0 = V_i + t \cdot (V_j - V_i); \tag{2.1}$$

čia I_0yra ieškomas taškas, V_i ir V_j žymi atkarpos galų taškus, o parametras$t \in [0,1]$.Ir atstumo iki antrojo trikampio plokštumos radimo formule:

$$I_0 = s_i + t \cdot (s_j - s_i); \tag{2.2}$$

Kadangi atstumas iki antrojo trikampio plokštumos yra žinomas (lygus nuliui, nes taškas ir yra ant plokštumos), formulėje 2.2 vietoje I_0 įstatę nulį, galime išsireikšti parametrą t:

$$t = \frac{-s_i}{(s_j - s_i)}; \tag{2.3}$$

Įstatę šią parametro t išraišką į formulę 2.1, gauname trikampio taško priklausančio sankirtos tiesei, apskaičiavimo formulę:

$$I_0 = V_i - \frac{s_i \cdot (V_j - V_i)}{(s_j - s_i)}; \tag{2.4}$$

Atlikę analogiškus veiksmus, gauname formulės ir likusiems taškams:

$$I_1 = V_i - \frac{s_i \cdot (V_k - V_i)}{(s_k - s_i)}; \tag{2.5}$$

$$J_0 = U_a - \frac{q_a \cdot (U_b - U_a)}{(q_b - q_a)}; \tag{2.6}$$

$$J_1 = U_a - \frac{q_a \cdot (U_c - U_a)}{(q_c - q_a)}; \tag{2.7}$$

Aukščiau išvardintose taškų apskaičiavimo formulėse esanti dalybos operacija, lyginant su daugyba yra gerokai lėtesnė. Chang ir Kim, pabrėžia, kad sankirtos fakto nustatymui nėra reikalingi tikslūs intervalų taškai, o tik pats intervalų persidengimo faktas, todėl dalybos operacijos yra pakeičiamos daugybos operacijomis, visas formules dauginant iš bendrų vardiklių [16]. Šį principą iliustruoja žemiau esančių formulių pertvarkymas jas subendravardiklinant:

$$I_0 = A + \frac{B}{x_0} \tag{2.8}$$

$$I_1 = A + \frac{C}{x_1} \tag{2.9}$$

$$J_0 = D + \frac{E}{x_0} \tag{2.10}$$

$$J_1 = D + \frac{F}{y_1} \tag{2.11}$$

čiaI_0, I_1, J_0, J_1yra ieškomi taškai kuriuose trikampių kraštinės kerta trikampių plokštumų sankirtos tiesę,A, B, C, D, E, F yra matematinės išraiškos, x_0, x_1, y_0, y_1 yra trupmenų vardikliai.

Atlikus subendravardiklinimą, gauname žemiau esančias taškų formules:

$$I_0 = A \cdot x_0 \cdot x_1 \cdot y_0 \cdot y_1 + B \cdot x_1 \cdot y_0 \cdot y_1 \tag{2.12}$$

$$I_1 = A \cdot x_0 \cdot x_1 \cdot y_0 \cdot y_1 + C \cdot x_0 \cdot y_0 \cdot y_1 \tag{2.13}$$

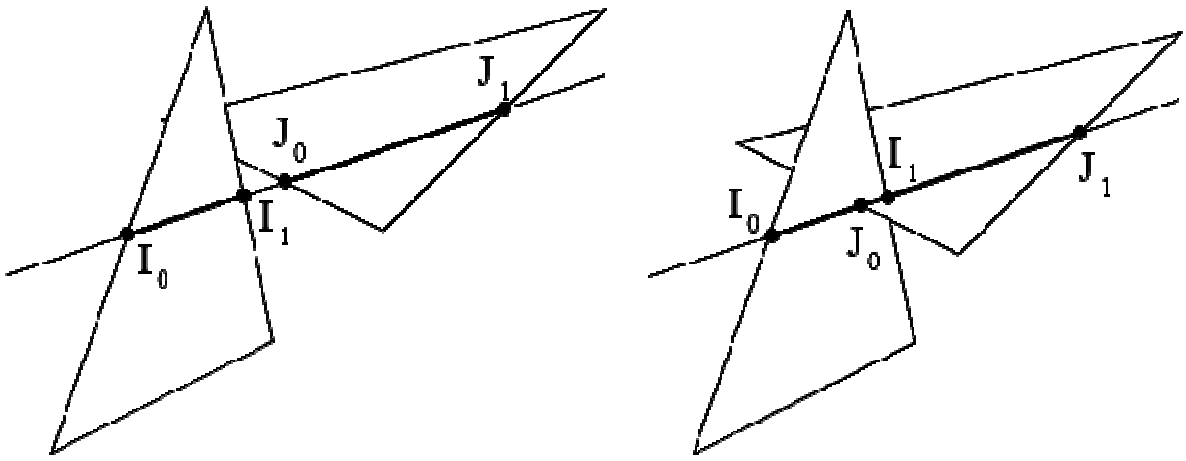
$$J_0 = D \cdot x_0 \cdot x_1 \cdot y_0 \cdot y_1 + E \cdot x_0 \cdot x_1 \cdot y_1 \tag{2.14}$$

$$J_1 = D \cdot x_0 \cdot x_1 \cdot y_0 \cdot y_1 + F \cdot x_0 \cdot x_1 \cdot y_0 \tag{2.15}$$

čia I_0, I_1, J_0, J_1 yra ieškomi taškai, kuriuose trikampių kraštinės kerta trikampių plokštumų sankirtos tiesę, A, B, C, D, E, F yra matematinės išraiškos.x_0, x_1, y_0, y_1 yra trupmenų vardikliai tapę daugikliais, kuriais yra subendravardiklinamos taškų skaičiavimo formulės.

Taip pat, siekiant sumažinti operacijų skaičių, siūloma, neskaičiuoti šių taškų trimatėje erdvėje, o projektuoti juos į vienmatę erdvę. Taigi ,skaičiavimuose naudojamos ne visos trys taškų koordinatės, o tik ta ašies koordinatė kuri buvo parinkta duomenų paruošimo žingsnyje iš vektoriaus m.

Nustačius intervalų taškus, atliekamas intervalų persidengimo testas, kuris grafiškai pavaizduotas žemiau esančiame Pav. 2.9.



Pav. 2.9 Intervalų persidengimo patikrinimas [17].

2.9 Darbo priemonių ir technologijų analizė

Susipažinus su sankirtos nustatymo metodais, toliau apžvelgiamos priemonės ir technologijas skaičiavimų greitaveikos paspartinimui, kurios naudojamos šio tyrimo metu. Pirmiausia trumpai apžvelgsime lygiagrečiuosius skaičiavimus ir jų panaudojimo galimybes iš teorinės pusės, po to pereisime prie šiuos skaičiavimus realizuojančios CUDA technologijos apžvalgos.

2.9.1 Lygiagretieji skaičiavimai

Norėdami geriau suprasti ir įvertinti, kokią potencialią naudą sankirtos nustatymo algoritmo greitaveikai gali duoti uždavinio išlygiagretinimas naudojant tiek CUDA, tiek kitas lygiagretaus programavimo technologijas, turime panagrinėti kaip tarpusavyje yra susiję uždavinio sprendimo laikas ir lygiagrečiai uždavinį sprendžiančių procesorių skaičius. Sprendžiant šį klausimą literatūroje yra dažnai sutinkami Amdalio ir Gustafsono dėsniai [12].

Amdalio dėsnis *(angl. Amdahl‘slaw)* pateiktas (2.16), nurodo, kaip turint fiksuoto dydžio uždavinį, kinta jo sprendimo laikas, kuomet jam spręsti yra pridedama daugiau procesorių.

$$S = \frac{1}{(1-P)+\frac{P}{N}}; \tag{2.16}$$

Čia S yra maksimalus galimas programos vykdymo pagreitėjimas, P yra trupmena nurodanti kuri kodo dalis gali būti išlygiagretinta, N yra procesorių skaičius kuriuose bus vykdoma lygiagreti programos kodo dalis. Iš formulės matyti, kad skaičiui P artėjant link vieneto, papildomi procesoriai atneša vis didesnę naudą ir atvirkščiai, kuo šis skaičius arčiau nulio, tuo procesorių skaičiaus daroma įtaka uždavinio sprendimo greičiui bus mažesnė. Taigi, norint pagreitinti uždavinio sprendimą, išeitis ne visuomet yra geresnė techninė įranga. Kartais gali būti parankiau skirti išteklius ir pastangas siekiant padidinti išlygiagretinamų programos kodo dalių skaičių.

Kaip ir buvo minėta, Amdalio dėsnis taikomas tuomet, kai turime fiksuoto dydžio uždavinį ir mus domina jo sprendimo trukmės sutrumpinimas. Tačiau gali būti ir toks atvejis, kuomet yra aktualu išlaikyti tokį patį sprendimo laiką, sprendžiant didesnį uždavinį. Tokiu atveju yra naudotinas Gustafsono dėsnis *(angl. Gustafson‘slaw)* nurodytas (2.17).

$$S = N + (1 - P)(1 - N); \tag{2.17}$$

Čia S yra maksimalus galimas programos vykdymo pagreitėjimas, P yra trupmena nurodanti kuri kodo dalis gali būti išlygiagretinta, N yra procesorių skaičius kuriuose bus vykdoma lygiagreti programos kodo dalis. Šis dėsnis nusako, kaip kinta uždaviniui spręsti reikalingas laikas pridedant daugiau procesorių, kuomet vienam procesoriui tenkančio uždavinio dydis nekinta, t.y. laikoma, kad uždavinio dydis auga kartu su procesorių skaičiumi. Praktikoje tai gali būti naudinga tuomet, kai pridėdami daugiau procesorių galime didinti sprendinio tikslumą, neprarasdami greitaveikos.

2.9.2 Nvidia CUDA technologija

CUDA (*angl. Compute Unified Device Architecture*), savo ruožtu nėra programavimo kalba ar įskiepis, o yra įvardijama kaip lygiagretaus skaičiavimo platforma ar programavimo modelis, leidžiantis panaudoti vaizdo plokštės teikiamus resursus, t.y. grafinį procesorių bei vidinę atmintį, įvairių, ne tik su kompiuterine grafika susijusių, bet ir kitokio pobūdžio uždavinių sprendimui tokiose srityse, kaip chemija, medicina, mechanika, elektronika, duomenų gavyba ir t.t.

CUDA programavimo modelyje dalyvauja tiek CPU tiek GPU. CPU ir jo atmintis literatūroje dažnai vadinama šeimininku*(angl. host)*, o GPU ir jo atmintis vadinama įrenginiu *(angl. device).* Tiek CPU, tiek GPU gali būti daugiau nei vienas. Funkcijos kurias vykdo GPU yra vadinamos branduoliais *(angl. kernel).* Viena branduolį gali lygiagrečiai vykdyti daugelis GPU gijų. Tipinė CUDA programos, paremtos šiuo modeliu, veiksmų seka būtų:

1. Šeimininko ir įrenginių atminties paskelbimas ir išskyrimas.
2. Šeimininko duomenų inicializavimas.
3. Duomenų persiuntimas iš šeimininko į įrenginį.
4. Vieno ar daugiau branduolių vykdymas.
5. Rezultatų persiuntimas iš įrenginio šeimininkui.

Taigi, galime pastebėti, kad CUDA technologijos integravimas ir naudojimas nėra trivialus procesas ir, norint pasiekti geriausius rezultatus, programos turi būti rašomos atsižvelgiant į šios technologijos architektūrą ir savybes, programų rašymo jai specifiką ir optimizavimo būdus. Šiam tikslui pasiekti galime pasirinkti vieną iš alternatyvų programos realizacijai [7]:

1. Paimti jau esamus algoritmų išeities tekstus ir nustatyti jų dalis, kurių skaičiavimo išlygiagretinimas ir perkėlimas į GPU atneštų didžiausią naudą greitaveikai.
2. Visiškai perrašyti algoritmo realizaciją iš naujo, nuo pat pradžių orientuojantis į tai, kad programa bus skirta veikti GPU ir naudos jam parankias duomenų struktūras.

Pirmoji alternatyva yra labiau tinkama tuomet, kai jau turime esamą programą ir norime ją paspartinti panaudodami lygiagrečiuosius skaičiavimus. Šiuo atveju galime turėti daug tarpusavyje susijusio liktinio *(angl. legacy)* programos kodo, tad jį visą perrašyti būtų sudėtinga. Todėl įvertinę siaurąsias programos vietas *(angl. bottlenecks)* ir esant galimybei jas išlygiagretinti, galime perrašyti tik jas.

Antroji alternatyva yra labiau priimtina tuomet, kai sprendžiame problemą nuo nulio ir dar neturime parašyto programos kodo arba norimo paspartinti programos kodo kiekis nėra labai didelis. Norėdami pasirinkti vieną iš alternatyvų, kuomet turime egzistuojantį programos kodą, galime įvertinti jo išlygiagretinimo galimybes, pasinaudojant anksčiau minėtais Amdalio ir Gustafsono dėsniais. Šiame tyrime, taikysime antrąjį variantą.

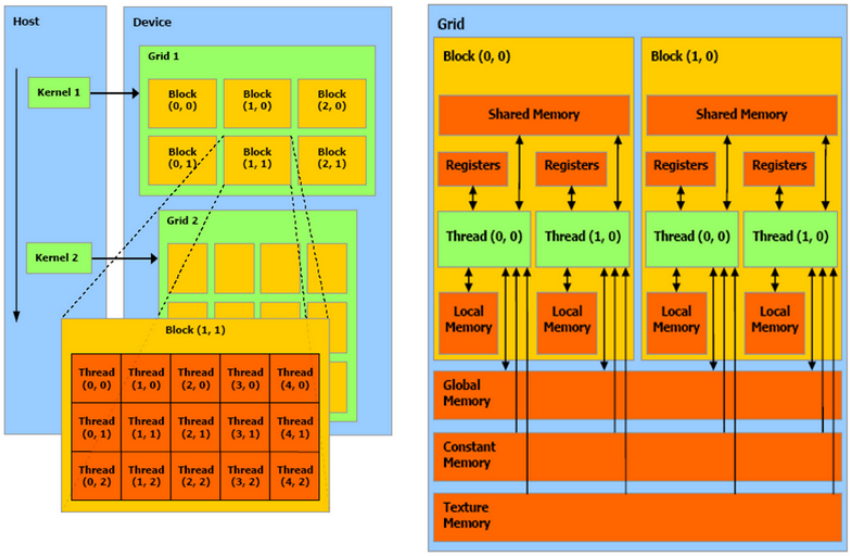
2.9.3 CUDA architektūra

Kaip jau buvo minėta, grafiniai procesoriai CUDA architektūroje yra vadinami įrenginiais. Viename įrenginyje gali būti keletas lygiagrečiai dirbančių multiprocesorių. Kuomet iš šeimininko kodo yra kviečiama tam tikra funkcija (branduolys), sukuriamas ją vykdančių gijų blokas. Gijų blokai yra apjungiami į gijų tinklelius, kuriuos vykdo įrenginio multiprocesoriai [13]. Pagrindiniai CUDA architektūros struktūriniai vienetai ir jų tarpusavio ryšiai yra aprašyti Lentelė 2.3

Lentelė 2.3 CUDA struktūriniai vienetai

Pavadinimas	Aprašymas
Įrenginys (GPU) *(angl. Device)*	Vienu metu gali vykdyti kelis branduolius(funkcijas). Sudarytas iš multiprocesorių (žr. žemiau).
Lygiagretusis multiprocesorius *(angl. Streaming Multiprocessor)*	Gali vykdyti keletą blokų vienu metu. Gijos ir jų blokai negali migruoti tarp skirtingų multiprocesorių.
Blokų tinklelis *(angl. Grid of blocks)*	Blokų tinkleliai sugrupuoja gijų blokus ir yra vykdomi visame GPU.
Gijų blokas *(angl. Block of threads)*	Sugrupuoja gijas ir yra vykdomi lygiagrečiuose multiprocesoriuose. Visos gijos bloke yra vykdomos lygiagrečiai.
Lynas *(angl. Warp)*	Viename bloke esančios gijos yra grupuojamos į lynus. Tam pačiam lynui priklausančios gijos yra įvykdomos vienu metu.
Gija *(angl. thread)*	Vienu metu GPU gali vykdyti tūkstančius lygiagrečių gijų. Visos gijos vykdo tą patį branduolio (žr. žemiau) kodą, tačiau vykdymo metu gali pasukti skirtingais loginiais išsišakojimais Tame pačiame bloke esančios gijos gali turėti bendrų duomenų.
Branduolys *(angl. Kernel)*	Funkcija kuri vykdoma grafiniame procesoriuje, kaip lygiagrečių gijų masyvas.

Svarbi CUDA architektūros dalis taip pat yra įvairūs jos atminties lygmenys, pavaizduoti Pav. 2.10. Iš jų svarbu išskirti ir paminėti globalią ir bendrąją*(angl. shared)* atmintis bei gijų registrus, kadangi jie bus svarbūs tolesniame optimizavimo skyrelyje. Globali atmintis yra prieinama visiems gijų blokams, tačiau veikia lėčiau nei bendroji atmintis, kuri yra bendra tik tarp tame pačiame bloke veikiančių gijų. Registrai yra atminties vienetai priskiriami individualioms gijoms.



Pav. 2.10 CUDA architektūros schema [14]

2.9.4 CUDA optimizacijos

Siekiant didesnės greitaveikos ir pilno CUDA galimybių panaudojimo, būtina programas optimizuoti atsižvelgiant į CUDA architektūros specifiką, o neretai ir į konkrečią vaizdo plokštę, kurią ketinama naudoti programos vykdymui. Ryoo, S., *et al.* tyrimuose buvo nustatyta keletas veiksmingiausių optimizavimo būdų greitaveikai padidinti [6]. Šiuos principus galima pritaikyti visoms plokštėms palaikančioms CUDA, ne tik tyrimo metu naudotiems modeliams.

Bene svarbiausias dalykas, į kurį reikia atkreipti dėmesį, rašant lygiagrečius skaičiavimus atliekančias programas GPU, yra užklausos į globalią atmintį. Šios užklausos, lyginant su skaičiavimais, trunka labai ilgai, tad, kol jos yra vykdomos procesoriai stovi be darbo. Siekiant tai išspręsti, siūloma didinti gijų kiekį ir jas daryti smulkesnes. Tuomet kol vienos gijos laukia resursų iš globalios atminties, galima pradėti vykdyti kitas. Taip pat siūloma naudoti į blokus apjungtose gijose esančią bendrąją atmintį *(angl. shared memory)* ir leisti gijoms tarpusavyje dalintis bendrai naudojamais duomenimis, kai tai yra įmanoma [15].

Kitas lygiagretaus kodo optimizavimo būdas yra taip vadinamosios klasikinės kompiliatorių optimizacijos, pašalinančios kodą kuris nėra tiesiogiai susijęs su skaičiavimais, kaip antai adresų apskaičiavimas ar išsišakojimai programos logikoje. Tai gali būti tokie metodai kaip tarpinių išraiškų *(angl. subexpression)* pašalinimas ir ciklų išskleidimas *(angl. loop unrolling).*

Vis tik, yra pastebima, kad šio pobūdžio optimizacijos ne visada atneša naudą ir kartais gali būti nuostolingos, priklausomai nuo programos specifikos [6]. Net ir subtilūs programos kodo pakeitimai gali įtakoti ją vykdančiai gijai priskiriamų registrų skaičių, taigi, ir patį gijų kiekį, kuris gali būti vykdomas vienu metu. Tad norint pritaikyti šio pobūdžio optimizacijas yra itin svarbu jas ištestuoti ir įsitikinti, jog jos tikrai įtakoja programos greitaveiką teigiamai.

2.9.5 *Thrust* biblioteka

Thrust yra C++ šablonų (*angl. template*) biblioteka skirta naudojimui su CUDA ir yra paremta, bei geba kartu dirbti su C++ programuotojams gerai pažįstama standartine šablonų biblioteka (*angl. Standard Template Library (STL)*) [18]. Panaši ir šių dviejų bibliotekų sintaksė. *Thrust* biblioteka pateikia lygiagrečius įvairių uždavinių sprendimui skirtus algoritmus tokius kaip masyvo elementų sumavimas, rikiavimas, dviejų masyvų sudauginimas panariui ir t.t. Biblioteka be šių paprastus uždavinius sprendžiančių algoritmų pateikia ir daugiau sudėtingesnių funkcijų, taip pat ir kelių funkcijų kombinavimo tarpusavyje galimybes.

Bibliotekos tikslas yra sutaupyti programuotojui laiko ir pakeisti komplikuotą bei sunkiai skaitomą gryną CUDA kodą į tvarkingesnius ir trumpesnius kreipinius, leisti programuotojui greitai išbandyti naujas idėjas ir sparčiau realizuoti uždavinį sprendžiančius prototipus. Bibliotekoje naudojami kintamųjų tipai savaime pasirūpina savo atminties išskyrimu ir išvalymu, bei duomenų kopijavimu iš kompiuterio atminties į vaizdo plokštės atmintį ir atgal. Thrust biblioteka taip pat suteikia didelį abstraktumo lygį, kadangi programuotojui panaudojus, pavyzdžiui rikiavimo funkciją, Thrust biblioteka automatiškai parenka optimaliausia rikiavimo implementaciją konkrečiai tam atvejui. Be *Thrust*, pats programuotojas turėtų rašyti šią užduotį atliekantį programos branduolį, parinkti įvairius jo vykdymo parametrus, kaip gijų skaičius bei tinklelio dydis, nuo kurių priklausytų ir programos greitaveika. Parinkti optimalius branduolio vykdymo parametrus nėra taip paprasta, tam reikalingos lygiagrečiojo programavimo bei CUDA architektūros žinios ir patirtis, informacija apie vaizdo plokštės parametrus. Taigi Thrust biblioteka padeda išvengti galimų programuotojo klaidų, parenkant šiuos parametrus, automatiškai juos nustatydama.

2.10 Reikalavimai sankirtos nustatymo uždavinio sprendimui

Atliekant literatūros analizę, buvo išnagrinėti sankirtos nustatymo srityje vykdomi moksliniai tyrimai, nustatyti šiuolaikiniai sankirtos nustatymo algoritmai ir metodai, paremti kelių lygmenų sankirtos nustatymu, kombinuojant kelis skirtingo tikslumo sankirtos nustatymo metodus. Tuo remiantis galima būtų iškelti tokius pirminius reikalavimus sprendimui:

- Sprendime turėtų būti naudojamas kelių lygmenų sankirtos nustatymas, susidedantis iš paeiliui vykdomų plačiojo, vidutinio ir siaurojo etapų.
- Kiekvienam etapui turėtų būti parenkamas tinkamas algoritmas, organizuojant etapus taip, kad pirmiausia didžiausiai aibei būtų taikomi greičiausiai veikiantys apytiksliai algoritmai, o lėtesni ir didesnį tikslumą turintys algoritmai būtų pritaikomi jau sumažintai potencialiai besikertančių objektų aibei, kitaip tariant greiti algoritmai turėtų dirbti su didelėmis duomenų apimtimis, lėti – su mažesnėmis.
- Etapų skaičius bei juose naudojami algoritmai turėtų būti parenkami taip, kad kiekvieno etapo metu atliekami skaičiavimai atsipirktų sumažindami galimai besikertančių objektų aibę ir nebūtų pertekliniai.
- Paskutinio, siaurojo etapo metu, nustatytos sankirtos turėtų tenkinti užsibrėžtus tikslumo reikalavimus ir gražinti teisingus rezultatus.
- Visų etapų veikla turėtų būti kaip įmanoma daugiau optimizuota ir pasižymėti užsibrėžtus reikalavimus tenkinančia greitaveika.
- Kadangi ne visi algoritmai yra tinkami lygiagretinimui, turėtų būti pasiūlytos galimos jų modifikacijos veikimui GPU.

3. SIŪLOMŲ SANKIRTOS METODŲ SPECIFIKACIJA

Šiame skyriuje pateikiamas projekto planas ir siekiami kokybės kriterijai, keliami funkciniai ir nefunkciniai reikalavimai sprendimui, bei įvardinamos sprendimo kūrimo priemonės. Taip pat pateikiama susisteminta informacija apie naudojamus metodus, pateikiamos algoritmų veikimo principus specifikuojančios diagramos.

3.1 Projekto planas

Šio tyrimo metu tarpusavyje lyginami nuosekliai ir lygiagrečiai veikiantys sankirtos nustatymo algoritmų variantai. Sprendžiamas uždavinys yra sankirtos tarp trimačių objektų nustatymas. Algoritmų duomenys yra trimačiai modeliai, sudaryti iš trikampių tinklelių, o rezultatas yra besikertančių trikampių porų sąrašas.

Be pilno perrinkimo algoritmų, uždavinio sprendimui, iš analizės skyriuje aprašytų erdvės skaidymo algoritmų parinkti ir realizuoti šie du metodai:

- Tolygus erdvės skaidymo tinklelis
- Aštuntainis medis

Priklausomai nuo algoritmo, sankirtos nustatymas atliekamas naudojant vieną ar daugiau etapų. Pilno perrinkimo algoritmų atvejų yra tik vienas etapas, kurio metu tikrinamos visos įmanomos abiejų objektų trikampių kombinacijų poros. Tolygaus tinklelio ir aštuntainio medžio atveju, sankirtos nustatymas skaidomas į tris etapus:

- Platusis etapas: tikrinama abiejų objektų ašims orientuotų gaubiančių tūrių sankirta.
- Vidurinis etapas: tikrinamos trikampių gaubiančių stačiakampių sankirtos su erdvės skaidymo ląstelėmis, t.y. formuojama erdvės skaidymo duomenų struktūra.
- Siaurasis etapas: tikrinamos sankirtos tarp atrinktų trikampių porų.

Plačiojo etapo metu, atliekama viena sankirtos nustatymo operacija tarp abiejų objektų ašims orientuotų gaubiančiųjų stačiakampių. Jei ši operacija gražina neigiamą rezultatą, tolesni etapai nėra vykdomi, besikertančių trikampių porų nėra. Jei operacijos rezultatas teigiamas, yra apskaičiuojama bendra abiejų objektų gaubiančių tūrių sritis, kuri bus suskaidyta vienu iš erdvės skaidymo algoritmų.

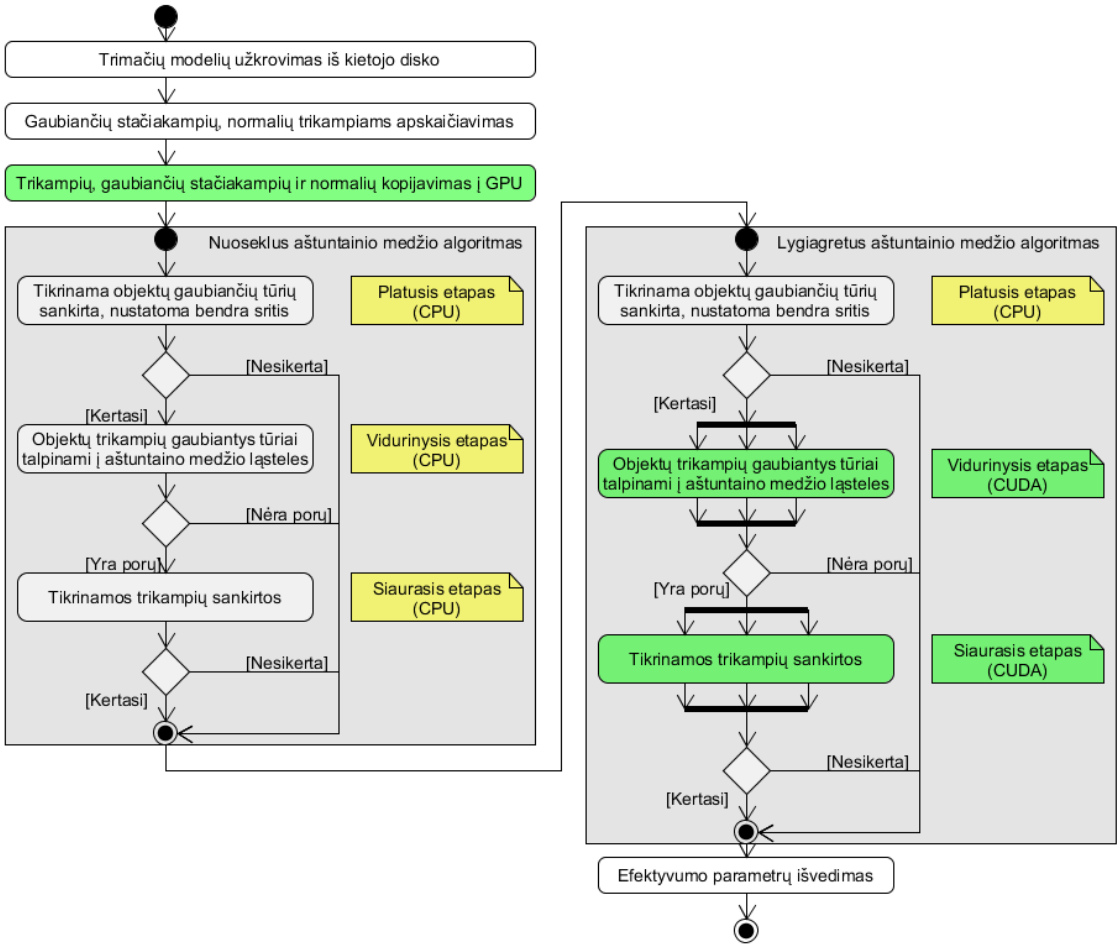
Vidurinio etapo metu, į suskaidytos erdvės ląsteles yra priskiriami abiejų objektų trikampiai. Tai atliekama pasitelkiant gaubiančiuosius tūrius, t.y. tikrinamos sankirtos tarp trikampius gaubiančių ašims orientuotų stačiakampių ir erdvės ląstelių, kurios taip pat nusakomos ašims orientuotais gaubiančiais stačiakampiais. Sudėjus trikampius į erdvės ląsteles, yra atliekama ląstelių, kuriose yra abiejų objektų trikampių paieška ir formuojamos visos šių trikampių porų kombinacijos.

Siaurojo etapo metu yra vykdomas tikslus sankirtos nustatymas tarp visų unikalių praeitame etape atrinktų trikampių porų. Šio etapo rezultatai yra besikertančių trikampių poros, ir šie rezultatai yra laikomi galutiniais.

Tyrimui atlikti taip pat realizuotos ir lygiagrečiai veikiančios šių algoritmų versijos. Tais atvejais kai algoritmų ar tam tikrų jų dalių išlygiagretinti negalima ar yra pernelyg sudėtinga, naudojamos šių algoritmų modifikacijos. Tai analizės skyriuje aprašytas tiesinis aštuntainis medis, leidžiantis panaikinti išlygiagretinimui netinkančią rekursiją, bei kintamo dydžio maišos

lentelę (*angl. hashtable*) pakeičiantys fiksuoto dydžio tiesioginio kreipimosi kodų metodai, aprašyti tolesniuose poskyriuose.

Išanalizavus esamus sprendimus ir susipažinus su sankirtų nustatymo metodika, bei lygiagrečių skaičiavimų naudojant CUDA technologiją specifika, buvo sudaryta veiklos diagrama pavaizduota žemiau esančiame Pav. 3.1. Algoritmų realizacija paremta tolesniuose paragrafuose aprašytais techninės ir programinės įrangos reikalavimais, kurie keliami būtent CUDA technologijos panaudojimui. Algoritmai taip pat turi tenkinti apibrėžtus funkcinius ir nefunkcinius reikalavimus sprendimui. Algoritmų tarpusavio palyginimas ir analizė vykdoma pagal išsikeltus siekiamus kokybės kriterijus, nusakančius pageidautiną algoritmų greitaveiką, tikslumą bei kitas savybes.

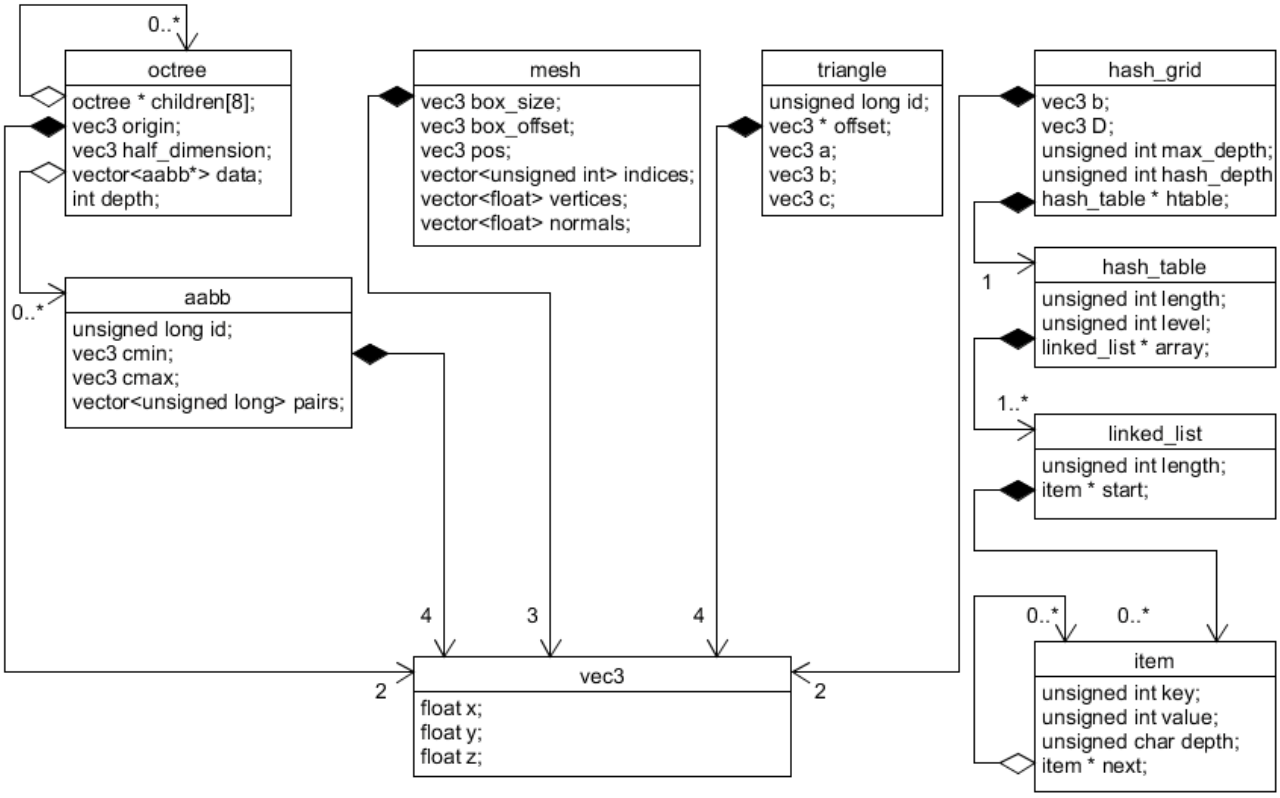


Pav. 3.1 Objektų sankirtos skaičiavimo veiklos diagrama

Kaip matyti iš Pav. 3.1, programos darbo pradžioje dalis skaičiavimams reikalingų duomenų yra užkraunama iš kietojo disko, dalis – apskaičiuojama. Visi algoritmams reikalingi duomenys taip pat yra nukopijuojami ir į GPU atmintį. Tai yra atliekama tik vieną kartą programos darbo pradžioje, siekiant išvengti ilgų duomenų kopijavimo operacijų kiekvieną kartą vykdant algoritmą. Toliau seka nuoseklus aštuntainio medžio algoritmas, sudarytas iš 3 etapų. Jam pasibaigus vykdomas lygiagretusis aštuntainio medžio algoritmas. Plačiajame etape yra nustatoma abiejų objektų gaubiančių stačiakampių bendra sankirtos sritis. Jei bendros srities nėra, vadinasi objektai nesikerta ir algoritmas baigia darbą. Jei bendra sritis egzistuoja, ji yra suskaidoma aštuntainiu medžiu. Viduriniame etape, kuris trunka ilgiausiai, yra atliekamas trikampius gaubiančių stačiakampių sudėjimas į aštuntainio medžio ląsteles. Yra ieškoma ląstelių į kurias buvo įterpti tiek vieno, tiek kito objekto trikampiai, nes iš jų galima sudaryti trikampių poras sekančiam etapui.

Jei tokių porų nėra, algoritmas baigia darbą. Jei porų yra, algoritmas tęsia darbą ir perduoda porų sąrašus paskutiniam, siaurajam etapui. Jo metu atliekami galutiniai sankirtų tarp trikampių porų tikrinimai.

Svarbu paminėti, kad lygiagretaus algoritmo atveju, platusis etapas yra vykdomas nuosekliai, kadangi tėra tikrinama viena sankirta tarp objektų gaubiančių tūrių, skaičiavimų neapsimoka perkelti į GPU. Tuo tarpu likę du etapai vykdomi lygiagrečiai, nes juose tikrinimų skaičius didesnis ir lygiagretieji skaičiavimai atsiperka. Pav. 3.2 pateikiama klasių diagrama apžvelgia, kaip galėtų atrodyti realizuoto sprendimo duomenų struktūros.



Pav. 3.2 Duomenų struktūrų klasių diagrama

Aukščiau esančioje diagramoje galima matyti esmines, sprendimui reikalingų duomenų struktūrų klases. Tai 3D vektoriaus klasė, kuri naudojama saugoti trimačio taško koordinates, objektų matmenis, normalių vektorius ir pan. Trikampių tinklelio klasėje, bus saugomi trimačio modelio taškų (*angl. vertex*) duomenys, bei iš jų sudaryti trikampiai, aprašomi trimis taškų indeksais. Taip pat reikalinga ašims orientuoto gaubiančio stačiakampio klasė kurios objektai naudojami aštuntainio medžio hierarchijoje. Galiausiai pavaizduota tinklelį realizuojanti klasė ir jai reikalinga maišos lentelės klasė, bei nuoseklaus sąrašo klasė su savo elemento klase.

Verta paminėti, kad diagramoje pavaizduotos klasės yra aktualios ir naudojamos tik CPU kode, tuo tarpu į GPU siunčiami duomenys neturi jokios klasių struktūros ir yra sudėti į paprastus masyvus. Taip yra siekiama sumažinti duomenims saugoti reikalingos atminties kiekį ir jų kopijavimo laiką, taip pat išvengti sudėtingų rodykles naudojančių hierarchinių struktūrų.

3.2 Funkciniai reikalavimai

Sprendimui yra keliami šie funkciniai reikalavimai:

- Algoritmai turi būti pritaikyti Nvidia CUDA architektūros vaizdo plokštėms.
- Algoritmų atliekamas sankirtų nustatymas turi būti tikslus, pateikiamos konkrečios besikertančių trikampių poros.
- Algoritmų pateikiamos besikertančių trikampių poros turi būti unikalios, t.y. nepateikiami dublikatai.

3.3 Nefunkciniai reikalavimai

Sprendimui yra keliami šie nefunkciniai reikalavimai:

- Algoritmų atminties sąnaudos turi būti kiek įmanoma mažesnės.
- Algoritmai turi veikti stabiliai ir sklandžiai.
- Turi būti skaičiuojamos atskirų algoritmų etapų trukmės.
- Turi būti skaičiuojama konstantinė ir dinaminė algoritmų naudojama atmintis.

3.4 Siekiami kokybės kriterijai

Atliekant tyrimą bus lyginami pilno perrinkimo, tolygaus tinklelio ir aštuntainio medžio sankirtų nustatymo trimačiams objektams algoritmai, bei keletas jų modifikacijų. Kiekvieną jų realizuojant bus siekiama patenkinti šiuos kriterijus:

- Algoritmai turi dirbti ne tik su trivialiais trimačiais objektais, bet ir su objektais turinčiais kelis šimtus tūkstančių trikampių.
- Algoritmai turi efektyviai išnaudoti techninės įrangos resursus, neeikvoti per daug atminties.

Taigi kaip svarbiausius kokybės kriterijus būtų galima išskirti algoritmų greitaveiką, atminties sąnaudas ir gebėjimą dirbti su dideliais duomenų kiekiais, t.y. sudėtingais trimačiais modeliais.

3.5 Sprendimo kūrimo metodai ir priemonės

Sprendimui realizuoti bus taikomi tiek bendro pobūdžio lygiagretaus programavimo metodai ir praktikos, tiek specifiniai su CUDA technologija susiję metodai.

Darbo priemonės yra CUDA įrankių rinkinys ir CUDA C/C++ kalba kompiliuojama nvcc kompiliatoriumi. CUDA C/C++ kalba yra paremta C/C++ kalbomis su papildomomis funkcijomis leidžiančiomis komunikuoti su GPU ir vykdyti lygiagrečius skaičiavimus. Nors CPU veikiančiame kode galima naudotis visomis įprastomis C/C++ funkcijomis, tačiau GPU kode iš jų mums bus prieinamas tik tam tikras poaibis.

3.6 Trikampių sudėjimas į tinklelį

Tiek tinklelio, tiek aštuntainio medžio atveju yra apskaičiuojama bendra objektų, tarp kurių yra tikrinama sankirta, sritis. Vienu iš testuojamų atveju ši sritis yra abiejų objektų gaubiančių tūrių sankirta (*angl. intersection*), kitu atveju – sąjunga (*angl. union*).Ši erdvė yra skaidoma tinklelio ar aštuntainio medžio erdvės skaidymo metodais. Kadangi aštuntainis medis yra formuojamas iš apačios į viršų, pirmiausia trikampius sudedant į žemiausią galimą medžio lygmenį, tiek tinklelio, tiek aštuntainio medžio atveju šis žingsnis yra identiškas. Jei tinklelio ląstelių skaičius išilgai kiekvienos ašies yra vienodas ir yra dvejeto laipsnis, iš tinklelio galima padaryti medį, todėl išsikėlę šį papildomą apribojimą, toliau nagrinėsime bendresnį trikampių sudėjimo į tinklelį atvejį.

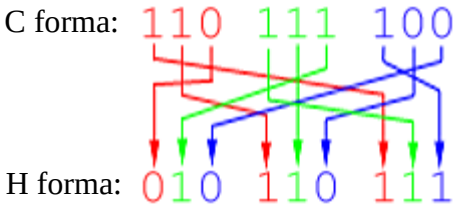
Įterpiant trikampius į tinklelį, šie gali papulti į kelias tinklelio ląsteles vienu metu. Žemiau pateikiama formulė (3.1) leidžia apskaičiuoti ląstelių intervalą, su kuriuo kertasi trikampio ašims orientuotas gaubiantis stačiakampis. Trikampių priskyrimas tinklelio ląstelėms vykdomas remiantis šia funkcija:

$$\mu = \left\lfloor \frac{2^L}{D} \left([0; D_B - \varepsilon] + (b - b_B) \right) \right\rfloor \tag{3.1}$$

čia$L$yra maksimalus medžio gylis,D–tinklelio trijų ašių matmenys, D_B – trikampio gaubiančio stačiakampio trijų ašių matmenys, birb_B – atitinkamai medžio ir trikampio gaubiančių stačiakampių pradžios taškų trimatės koordinatės, ε – labai mažas skaičius. Gauti intervalai, esantys slankaus kablelio skaičių pavidalu, yra sumažinami iki artimiausio sveiko skaičiaus (*angl. floor operation*). Pažymėtina tai, kad intervalai $[0; D_B - \varepsilon]$ ir μ yra aprašomi 2x3 matricomis, nurodančiomis intervalo pradžią ir pabaigą diskrečioje trimatėje erdvėje.

3.7 C, H ir Hp forma

Turėdami intervalą μ, galime apskaičiuoti visų į intervalą patenkančių ląstelių koordinačių (*angl. coordinate*), arba trumpiau, C formos kodus, ir sukurti reikiamas duomenų struktūras, atvaizduojančias suskaidytos erdvės ląstelių ir į jas papuolančių trikampių ryšius. C formos kodai leidžia lengvai sudėti trikampius į ląsteles, tačiau jie pasižymi keletu trūkumų. Pirmiausia, tarp skirtingo lygio ląstelių C formos kodų dešimtainių reikšmių yra dideli šuoliai, todėl jomis indeksuotas masyvas netolygiai pasiskirsto atmintyje ir apskritai nėra pilnai išnaudojamas. Kita problema yra ta, kad aštuntainio medžio atveju keli skirtinguose lygiuose esantys oktantai gali turėti vienodus kodus. C formos kodai nėra labai patogūs atliekant oktantų apjungimą.



Pav. 3.3 C formos kodo transformavimas į H formą

C formos kodus galima konvertuoti į patogesnę skaičiavimams ir optimaliau atmintyje pasiskirstančią hierarchinę (*angl. hierarchy*), arba trumpiau, H formą. Norint tai padaryti tereikia perrikiuoti C formos kodo bitus taip, kaip pavaizduota aukščiau Pav. 3.3. C formos kode, raudonai pažymėti bitai sudaro ląstelės x koordinatę dešimtainėje formoje, mėlyni – y koordinatę, žali – z. Būtent dėl šios priežasties, C formos kodas yra patogus trikampių priskyrimui į ląsteles. Atlikus

transformaciją į H formą, kiekviena bitų trijulė įgyja naują prasmę ir dabar saugo informaciją apie tai, kuriam iš vaikinių oktantų trikampis priklauso kiekviename medžio lygyje, todėl turint šį oktanto kodą, labai paprastą rasti jo tėvinį oktantą, pakanka numesti tris bitus iš kairės pusės, t.y. vieną vaikinį lygį. Šis konvertavimas neišsprendžia besidubliuojančių oktantų kodų problemos, todėl norint kodus paversti unikaliais, prie jų reikia pridėti konstantą. Anksčiau minėtas Mortono kodas yra analogiškas H formai, tačiau prie visų kodų prideda vienetą, taip paversdamas juos unikaliais, tačiau vis tiek išlieka šuolių tarp kodų reikšmių problema, kodai nėra nuoseklūs. Siekiant to išvengti, prie H formos yra pridedama tam tikra nuo oktanto lygio priklausanti konstanta γ, (žr. Lentelė 3.1), paverčianti kodus unikaliais ir nuosekliais. Šią formą pavadinsime Hp forma.

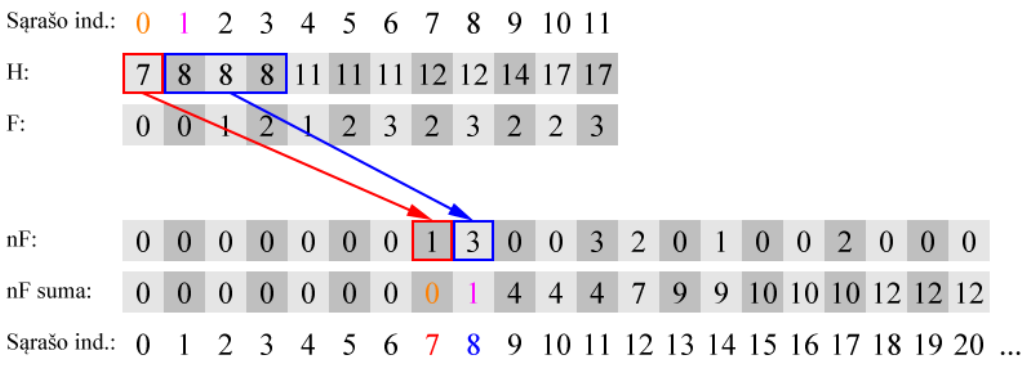
Lentelė 3.1 Hp forma

Lygis L	γ^L dešimtainis pavidalas	γ^Ldvejetainis pavidalas
0	0	0
1	1 + 0 = 1	1
2	1+8=9	1001
3	9+64=73	1001001
4	73+512=585	1001001001
5	585+ 4096=4681	1001001001001
6	4681+32768=37449	1001001001001001
7	37449+262144=299593	1001001001001001001
8	299593+2097152=2396745	10010010010010010010
9	2396745+ 16777216 = 19173961	10010010010010010010001001
10	19173961 + 134217728=153391689	10010010010010010010010010001

3.8 Oktantų apjungimas

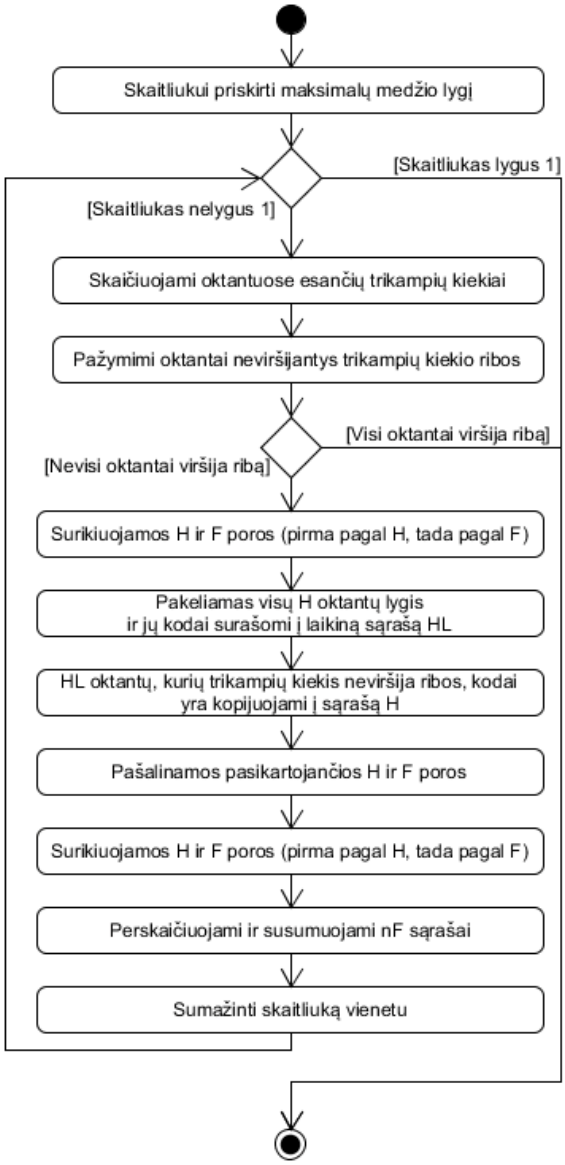
Aštuntainio medžio atveju, galima kai kuriuos žemesnio lygio oktantus apjungti į aukštesnius, tėvinius oktantus pagal užsibrėžtą maksimalaus trikampių kiekio oktante apribojimą. Oktantų apjungimo algoritmo veiklos diagrama yra pateikta žemiau esančiame Pav. 3.5. Diagramoje matyti, kad pagrindiniai veiksmai yra atliekami su H ir F sąrašais. H sąraše yra saugomi oktantų kodai Hp formos pavidalu. F sąraše yra saugomi unikalūs trikampių indeksai. Nors šie du sąrašai atskiri, jų tarpusavio elementai yra susieti oktanto-trikampio priklausomybės ryšiais, kuriuos privalu išsaugoti, atliekant sąrašų rikiavimo, porų šalinimo ir kitas operacijas. Tokių operacijų su suporuotais sąrašais vykdymui praverčia technologijų analizės skyriuje aprašyta *Thrust* biblioteka.

Kitas svarbus sąrašas yra nF, pavaizduotas žemiau esančiame Pav. 3.4. Šiame sąraše į indeksus, kuriuos atitinka sąrašo H reikšmės yra surašomi trikampių oktantuose kiekiai, t.y. unikalios H-F poros, kur H yra lygus masyvo nF indeksui. Sumuojant šio sąrašo elementus, gaunama informacija nuo kelinto indekso H-F sąrašuose prasideda konkretaus oktanto trikampiai, o atėmus gretimas reikšmes galima atgauti pirminę informaciją apie oktante esančių trikampių kiekį.



Pav. 3.4 H,F, nF ir nF sumos sąrašai

Svarbu paminėti tai, kad abu objektai, tarp kurių yra tikrinama sankirta, turi savo atskirus H1, F1 ir nF1 ir H2, F2 ir nF2 sąrašus, tačiau paprastumo dėlei algoritmo diagramoje šie sąrašai minimi bendrais H, F ir nF pavadinimais. Pakanka suprasti tai, kad diagramoje vaizduojami veiksmai yra atliekami su abiejų objektų sąrašais atskirai, išskyrus vienintelį atvejį, kuomet yra tikrinama ar trikampių kiekis neviršija užsibrėžtos ribos. Nors objektai ir turi savo individualius oktantų-trikampių sąrašus, formaliai abu objektai yra patalpinti į tą pačią medžio struktūrą ir medžio skaidymas, bei oktantų apjungimas abiem objektams turi būti vienodas.



Pav. 3.5 Oktantų apjungimo algoritmo veiklos diagrama

Kaip matyti iš aukščiau esančios diagramos (žr. Pav. 3.5), lygio kėlimo cikle yra sukuriamas laikinas oktantų kodų masyvas HL. Jame oktantų lygis keliamas remiantis paprasta formule:

$$H_T = \frac{(H-1)}{8} \tag{3.2}$$

čia H yra oktanto kodas, H_T – lygiu aukštesnio oktanto, t.y. oktanto tėvo kodas. Vienetas yra atimamas tam, kad kodas būtų transformuotas iš Hp formos atgal į H formą. Dalinama iš 8, nes oktantas turi 8 vaikus.

3.9 Trikampių porų sudarymas

Išsiaiškinus, į kurias bendros erdvės ląsteles papuola tiek vieno, tiek kito objekto trikampiai, reikia sudaryti šių porų sąrašusf^1 ir f^2, kurie bus pateikiami sekančiam, paskutiniam algoritmo žingsniui, atliekančiam tikslų sankirtos nustatymą tarp dviejų trikampių. Tam tikslui vykdomi du lygiagretūs branduoliai, atitinkamai surašantys poras iš vieno ir kito objekto trikampių. Žemiau yra pateikiamas jų pseudokodas (žr. gpu_create_pairs_1 ir gpu_create_pairs_2).

```
funkcija gpu_create_pairs_1
/*
 i - gijos indeksas
 n - F1 masyvo ilgis
 a - kelintas trikampis iš oktanto H1[i]
 b - nuo kelinto įrašo prasideda rašymas į sąrašą f1
 m - kiek kartų įrašyti į rezultatų sąrašą f1
 n1 - pirmojo objekto oktantų pradžios indeksų masyvas
 n2 - antrojo objekto oktantų pradžios indeksų masyvas
 nm - panariui sudaugintų masyvų n1 ir n2 elementų masyvas
 H1 - trikampių oktantų Hp forma masyvas
 F1 - trikampių indeksų masyvas
 f1 - rezultatų masyvas
*/
Su kiekviena i-tąja gija nuo 0 iki n
    a = i - n1[ H1[i]];
    m = n2[ H1[i]+1]- n2[ H1[i]];
    b = nm[ H1[i]]+ a * m;

    Visiems j = b; j < b + m; j=j+1:
        f1[j]= F1[i]
```

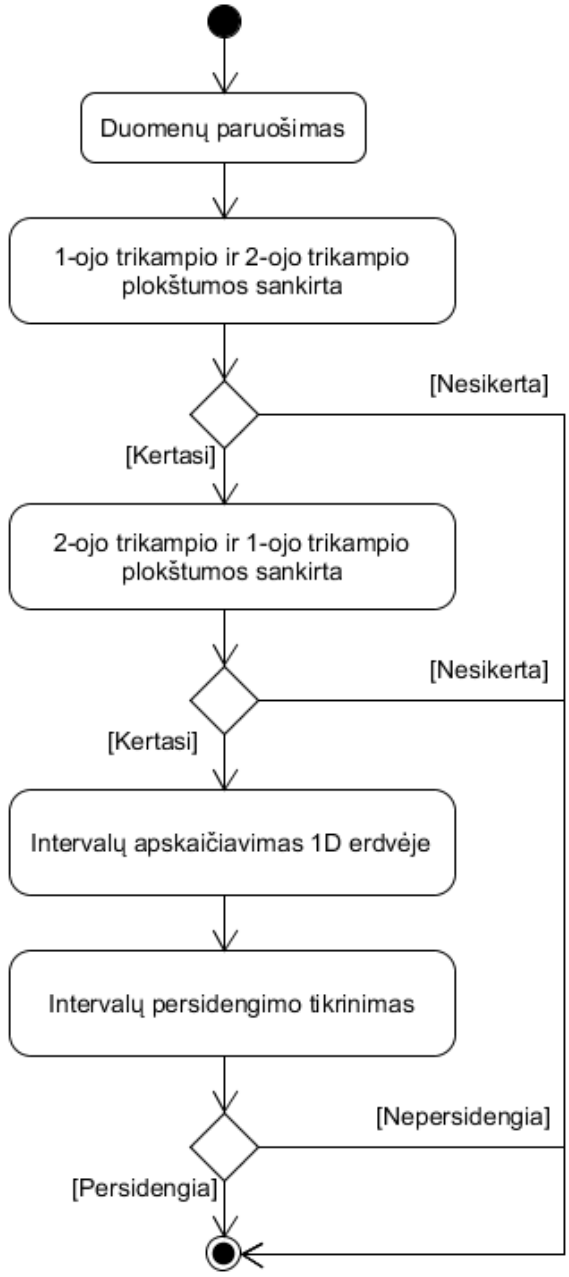
```
funkcija gpu_create_pairs_2
/*
 i - gijos indeksas
 n - F2 masyvo ilgis
 a - kelintas trikampis iš oktanto H2[i]
 b - nuo kelinto įrašo prasideda rašymas į sąrašą f2
 m - kokiais tarpais rašyti į rezultatų sąrašą f2
 n1 - pirmojo objekto oktantų pradžios indeksų masyvas
 n2 - antrojo objekto oktantų pradžios indeksų masyvas
 nm - panariui sudaugintų masyvų n1 ir n2 elementų masyvas
 H2 - trikampių oktantų Hp forma masyvas
 F2 - trikampių indeksų masyvas
 f2 - rezultatų masyvas
*/
Su kiekviena i-tąja gija nuo 0 iki n
    a = i - n1[ H1[i]];
    m = n1[ H2[i]+1]- n1[ H2[i]];

    Visiems j = nm[ H2[i]]+ a; j < n[ H2[i]+1]; j=j+m:
        f2[j]= F2[i]
```

Šių funkcijų suformuoti sąrašai gali turėti daug pasikartojančių porų, todėl, prieš perduodant porų sąrašus sekančiam algoritmo žingsniui, reikia panaikinti pasikartojančias poras. Sąrašų f^1 ir f^2 ilgiai vienodi ir yra lygus n, atlikus pasikartojančių porų pašalinimą, ši sąlyga nepasikeičia. Po šių operacijų galima paleisti $i = 0,1 \dots n-1$ gijų, kurių kiekviena skaičiuos f_i^1 ir f_i^2 trikampių sankirtą.

3.10 Trikampių sankirtos nustatymo metodas

Turint abiejų objektų trikampių porų sąrašus, galima atlikti paskutinį algoritmo žingsnį, tikslų sankirtos tarp trikampių nustatymą. Išsamus metodo aprašymas yra pateikiamas sankirtos nustatymo metodų analizės skyriaus dalyje. Pav. 3.6 yra pateikiamas analizės dalyje aprašytu metodu ir jo modifikacijomis paremtas trikampių sankirtos nustatymo algoritmo tarp vienos trikampių poros modelis.

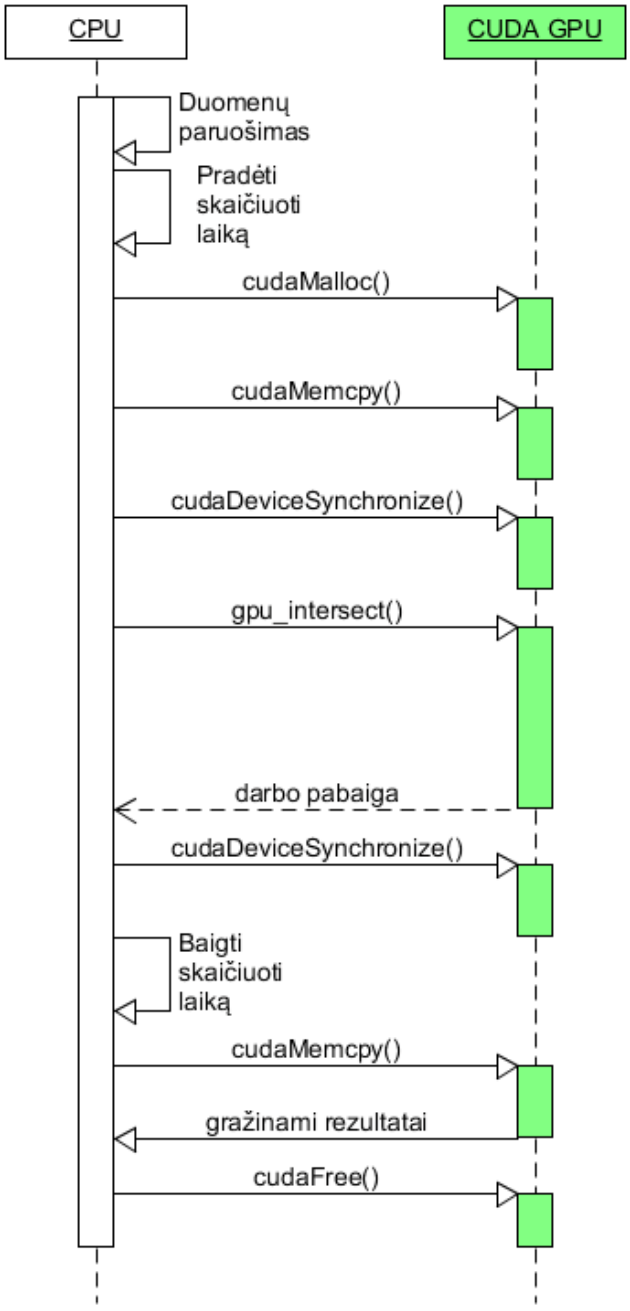


Pav. 3.6 Trikampių sankirtos nustatymo algoritmo veiklos diagrama

Kaip galima matyti iš Pav. 3.6, algoritme nuosekliai yra tikrinamos trys sąlygos. Jei bent viena iš sąlygų yra netenkinama, vadinasi tarp trikampių poros sankirtos nėra ir algoritmas yra nutraukiamas, grąžinamas rezultatas – trikampiai nesikerta. Priešingu atveju, trikampiai kertasi.

3.11 Lygiagretus sankirtų tikrinimas GPU

Anksčiau minėtas trikampių sankirtos nustatymo algoritmas programoje yra vykdomas nuosekliai, naudojant CPU, ir lygiagrečiai, naudojant GPU. Abiejų realizacijų greitaveikos yra palyginamos. Nuoseklus algoritmas veikia paprasčiausiai tikrindamas visas trikampių poras cikle, naudodamas anksčiau aprašytą sankirtos nustatymo algoritmą. Tuo tarpu lygiagretus algoritmo vykdymas yra kiek sudėtingesnis, ir reikalauja papildomų funkcijų kopijuojančių duomenys į GPU, bei sinchronizuojančių gijų darbą iškvietimo. Žemiau esantis Pav. 3.7 iliustruoja CPU ir GPU kodo bendradarbiavimą.



Pav. 3.7 Lygiagretaus algoritmo sekos diagrama

Kiti lygiagretūs algoritmai naudojami programoje vadovaujasi tokia pat schema, išskyrus tuos atvejus, kai yra naudojami *Thrust* bibliotekos metodai. Dirbant su jais nereikia atskirai vykdyti atminties išskyrimo ir kopijavimo operacijų, nes jomis pasirūpina patys bibliotekos duomenų tipai.

4. REZULTATŲ ANALIZĖ

Tyrimui atlikti buvo suprojektuoti pilno perrinkimo, erdvės skaidymo tinkleliu ir aštuntainiu medžiu metodai. Realizuoti nuosekliai CPU veikiantys ir lygiagrečiai GPU veikiantys, CUDA naudojantys algoritmai. Lygiagretiems algoritmams pritaikytos modifikacijos pakeičiančios nuoseklių algoritmų dalis netinkamas ar neefektyvias lygiagrečiam vykdymui, t.y. atsisakyta rekursijos ir nepastovų dydį turinčių maišos lentelių, jas pakeičiant tiesioginio kreipimosi metodais. Toliau pateikiama eksperimentų vykdymo sąlygos, rezultatai bei išvados.

4.1 Aparatūrinė įranga

Eksperimentai atlikti naudojant tris kompiuterius pasižyminčius skirtingais techninės ir programinės įrangos parametrais. Žemiau esančioje Lentelė 4.1 yra pateikiami šių kompiuterių parametrai.

Lentelė 4.1 Eksperimentams atlikti naudotų kompiuterių parametrai

	PC 1	**PC 2**	**PC 3**
Kompiuterio modelis	Acer Aspire 7738G	-	-
Procesorius	Intel (R) Core (TM) 2 Duo CPU	Intel (R) Core (TM) i5-2500 CPU	Intel (R) Core (TM) i7-4790 CPU
Procesoriaus dažnis	2100 MHz	3300 MHz	3600 MHz
OS	Windows 7 32bit	Windows 7 64bit	Windows 8.1 64bit
Atmintis	3066 MB	8156 MB	16314 MB
Vaizdo plokštė	Nvidia GeForce GT 130M	Nvidia Quadro 5000	Nvidia GeForce GTX 970
Dažnis	1500 MHz	1026 MHz	1367 MHz
Compute capability	1.1	2.0	5.2
Iš viso atminties	1024 MB	2560 MB	4095
Pastovios atminties	64 KB	64 KB	64 KB
Maksimali bendra atmintis bloke	16 KB	48 KB	48 KB
Maksimalus tinklelio dydis	65535 x 65535 x 1	65535 x 65535 x 65535	2147483647 x 65535 x 65535
Maksimalus bloko dydis	512 x 512 x 64	1024 x 1024 x 64	1024 x 1024 x 64
Multiprocesoriai x branduoliai	4 x 32	11 x 88	13 x 104
Maksimalus gijų skaičius bloke	512	1024	1024
Maksimalus registrų skaičius bloke	8192	32768	65536
Warp dydis	32 gijos	32 gijos	32 gijos

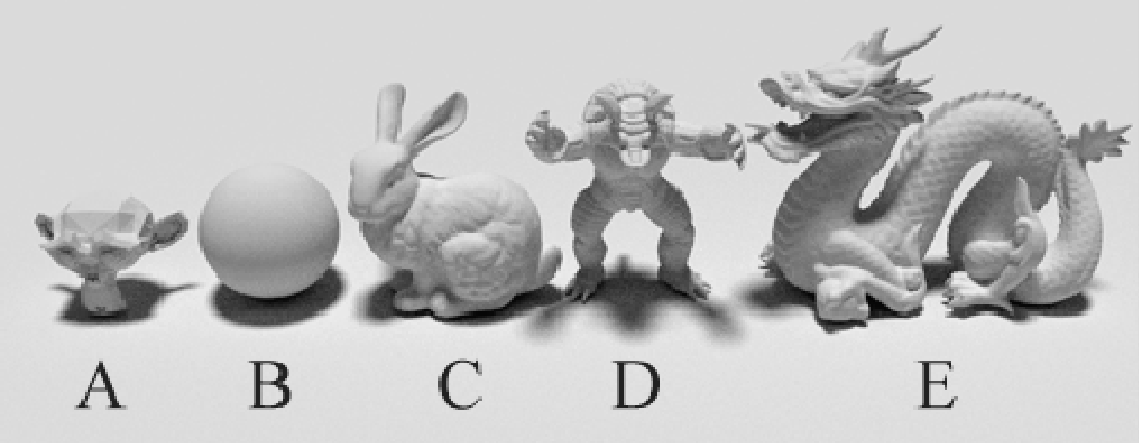
4.2 Programinė įranga

Tyrime naudota programinė įranga:

- **Integruota programų kūrimo aplinka**
  - Microsoft Visual Studio 2012
- **NVIDIA GPU Computing Toolkit 6.5 (PC 1, PC 2), 7.0 (PC 3)**
  - Bibliotekos:
    - Thrust
    - CUDA MathLibrarymath.h
    - helper_math.h
- **Trimačių modelių konvertavimui ir paruošimui**
  - Blender 2.72b

4.3 Eksperimentuose naudojami duomenys

Eksperimentų metu yra naudojami visiems laisvai prieinami trimačiai modeliai. Sankirtos yra nustatinėjamos tarp įvairių šių modelių porų. Tyrime naudoti modeliai yra pavaizduoti žemiau esančiame Pav. 4.1, o jų trikampių skaičius bei šaltiniai yra pateikiami žemiau esančioje Lentelė 4.2.



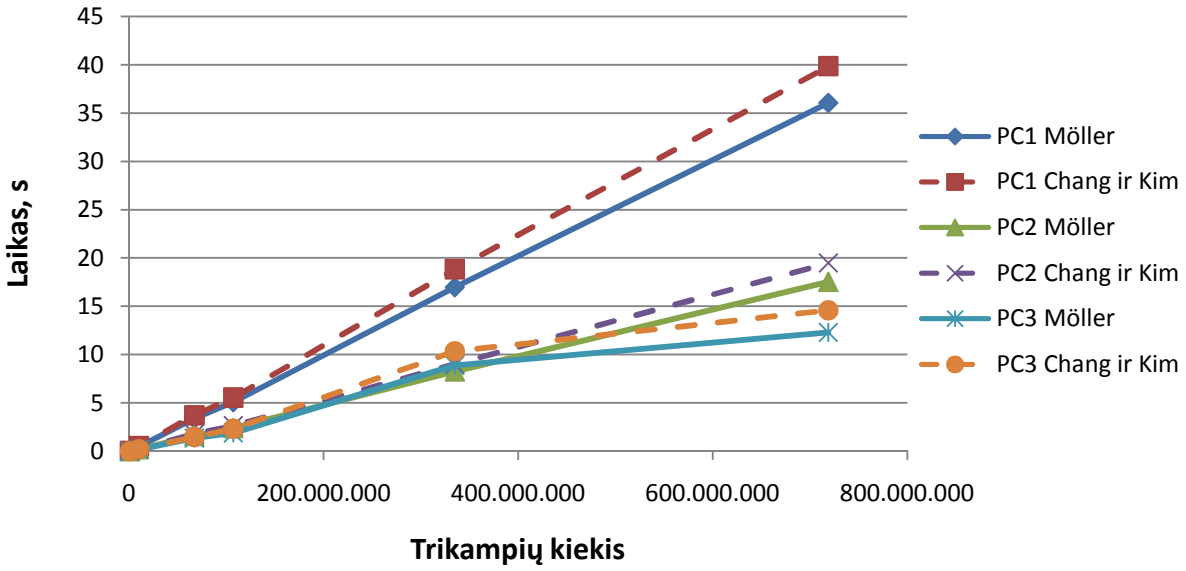
Pav. 4.1 Eksperimentų trimačiai modeliai

Lentelė 4.2 Trimačių modelių duomenys

Modelis	Pavadinimas	Trikampių skaičius	Šaltinis
A	Suzzane	968	Blender programos ruošinys
B	Sphere	10 350	Sukurta su Blender programa: Create UV Sphere
C	Stanford Bunny	69 451	Stanford skanuotų modelių repozitorija[21]
D	Stanford Armadillo	345 944	Stanford skanuotų modelių repozitorija[21]
E	Stanford Dragon	871 414	Stanford skanuotų modelių repozitorija[21]

4.4 Trikampių sankirtos algoritmų greitaveikos palyginimas

Siekiant nustatyti greičiausią trikampių sankirtos skaičiavimo metodą, kuris bus naudojamas tolesniuose tyrimuose, atliktas Möller, bei Chang ir Kim trikampių sankirtos nustatymo algoritmų greitaveikos palyginimas. Palyginimas atliktas vykdant du pilno perrinkimo algoritmus, tikrinančius visų įmanomų trikampių porų sankirtas tarp dviejų trimačių modelių. Žemiau esančiame grafike (žr. Pav. 4.2) yra pateikiama šių dviejų algoritmų vykdymo laikų priklausomybė nuo trikampių skaičius, leidžianti daryti išvadą, kuris algoritmas yra greitesnis.



Pav. 4.2 Algoritmų vykdymo laikų priklausomybė nuo trikampių skaičiaus

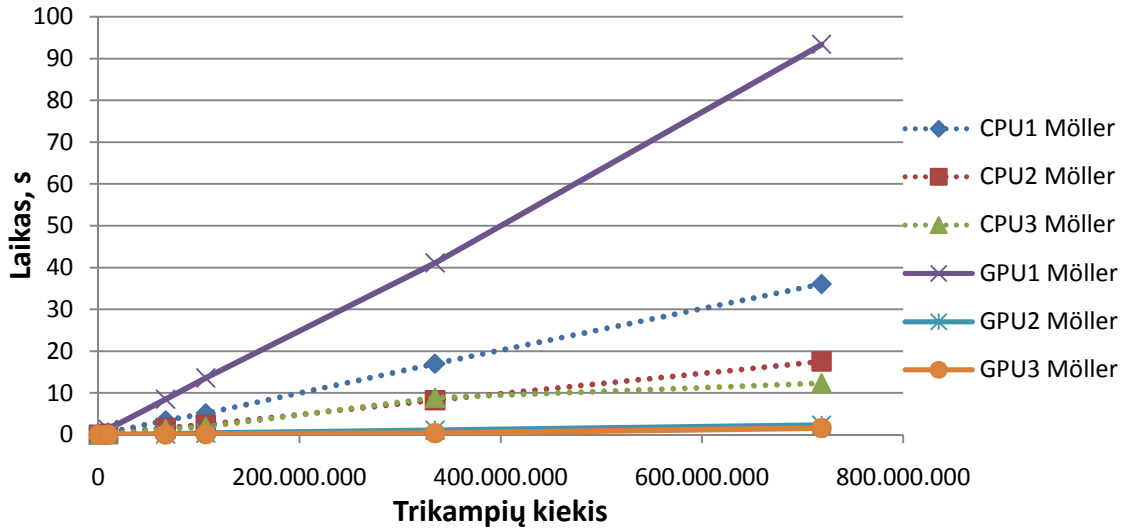
Iš grafiko matyti, kad visais atvejais Möller algoritmas veikė greičiau nei Chang ir Kim, kadangi Chang ir Kim turi papildomą tarpinių duomenų pasiruošimo žingsnį. Taip pat galima įžvelgti, kad algoritmų greitaveiką įtakojo naudojama techninė įranga, greitesni procesoriai skaičiavimams sugaišo žymiai mažiau laiko. Žemiau esančioje Lentelė 4.3 yra pateikiami algoritmų vykdymo laikai, kurių pagrindu buvo sudarytas šis grafikas. Visi skaičiavimai buvo atliekami 5 kartus, todėl lentelėje pateikiami skaičiai yra 5 matavimų vidurkiai.

Lentelė 4.3 Möller bei Chang ir Kim nuoseklių algoritmų laikai

CPU						
Objektų pora	A x A	A x B	A x C	B x B	A x D	B x C
Trikampių skaičius	937024	10018800	67228568	107122500	334873792	718817850
PC1 Möller	0,048	0,481	3,391	5,097	16,979	36,057
PC1 Chang ir Kim	0,051	0,528	3,731	5,566	18,846	39,858
PC2 Möller	0,024	0,228	1,608	2,413	8,267	17,549
PC2 Chang ir Kim	0,025	0,249	1,809	2,649	9,035	19,479
PC3 Möller	0,019	0,175	1,378	1,894	8,888	12,309
PC3 Chang ir Kim	0,022	0,225	1,494	2,341	10,369	14,601

4.5 CPU ir GPU trikampių sankirtos nustatymo algoritmų palyginimas

Nustačius, kad Möller algoritmas veikia greičiau, buvo realizuota lygiagrečiai veikianti šio algoritmo versija ir atliktas jos palyginimas su nuosekliai veikiančiu Möller algoritmu. Žemiau esančiame Pav. 4.3 yra pateikiami testų rezultatai.



Pav. 4.3 Nuoseklaus ir lygiagretaus algoritmų greitaveikos priklausomybė nuo trikampių sk.

Kaip matyti grafike, pirmojo kompiuterio atveju lygiagretus algoritmo variantas nepasiteisino ir veikė gerokai lėčiau nei nuoseklusis variantas, tačiau antrojo ir trečiojo kompiuterio atveju, lygiagretaus algoritmo pranašumas yra akivaizdus. Šis grafikas buvo sudarytas remiantis prieš tai buvusioje Lentelė 4.3 esančiais Möller algoritmo laikais, bei žemiau esančioje Lentelė 4.4 pateikiamais lygiagretaus Möller algoritmo vykdymo laikais. Visi skaičiavimai buvo atliekami 5 kartus, todėl lentelėje pateikiami skaičiai yra 5 matavimų vidurkiai.

Lentelė 4.4 Lygiagretaus Möller algoritmo vykdymo laikai

GPU						
Objektų pora	A x A	A x B	A x C	B x B	A x D	B x C
Trikampių skaičius	937024	10018800	67228568	107122500	334873792	718817850
GPU1 Möller	0,178	1,291	8,487	13,591	41,108	93,394
GPU2 Möller	0,004	0,036	0,128	0,356	1,092	2,278
GPU3 Möller	0,003	0,019	0,069	0,138	0,356	1,600

GPU1 Möller algoritmo lėtumą galima paaiškinti tuo, kad pirmajame kompiuteryje yra naudojama senos kartos vaizdo plokštė, turinti mažai vidinės gijų atminties (registrų), todėl sugaištanti pernelyg daug laiko duomenų kopijavimui, nes atminties nepakanka visus duomenis užsikrauti į registrus iš karto. Tuo tarpu tyrime naudotos naujesnės vaizdo plokštės pasižymi didesniais atminties resursais ir todėl aplenkia nuosekliai veikiantį algoritmą. Šis faktas mums suteikia vertingos informacijos apie tai, kaip svarbu yra įvertinti naudojamos techninės įrangos parametrus ir galimybes, nes tas pats algoritmas vienoje vaizdo plokštėje gali veikti labai greitai, o kitoje būti visiškai neefektyvus.

Šią problemą ateityje būtų galima spręsti, lygiagretųjį algoritmą specialiai pritaikant senesnei vaizdo plokštei jį išskaidant į dvi atskiras dalis ir taip neviršijant maksimalaus registrų skaičiaus. Kitas variantas būtų mėginti optimizuoti algoritmą ir pašalinti įvairius tarpinius ir papildomus kintamuosius naudojamus skaičiavimuose, taip sumažinant reikalingų registrų skaičių.

4.6 Erdvės skaidymo algoritmų palyginimas

Palyginus pilno perrinkimo algoritmų greitaveiką, toliau lyginami erdvės skaidymo metodus naudojantys algoritmai, siekiant išrinkti efektyviausią CPU ir efektyviausią GPU algoritmą. Tarpusavyje lyginami nuosekliai veikiantys erdvės skaidymo algoritmai yra maišos lentele paremtas tinklelis ir rekursinis aštuntainis medis. Kaip jau buvo minėta, rekursija yra netinkama lygiagrečiam vykdymui, o maišos lentelės grandinėlių ilgis yra nepastovus, todėl buvo realizuotos šių algoritmų modifikacijos lygiarečiam veikimui, tai tiesioginio kreipimosi tinklelis ir tiesioginio kreipimosi aštuntainis medis. Taip pat, šios modifikacijos buvo perkeltos ir į CPU, siekiant palyginti ar jos veiks greičiau, nei klasikiniai metodai be modifikacijų. Žemiau esančioje Lentelė 4.5 yra pateikiami nuoseklių algoritmų greitaveikos palyginimo rezultatai.

Lentelė 4.5 Nuoseklių erdvės skaidymo algoritmų greitaveikos palyginimas

CPU					
Objektų pora	B + B	A + D	B + C	C + C	B + E
Trikampių skaičius	20700	346912	79801	138902	881768
PC1					
Maišos lentelės tinklelis	**0,068**	1,371	**0,277**	**0,480**	3,194
Rekursinis aštuntainis medis	**0,068**	**1,365**	0,282	0,494	**3,085**
Tiesioginio k. tinklelis	0,074	1,367	0,296	0,534	3,183
Tiesioginio k. aštuntainis medis	0,082	1,379	0,323	0,614	3,376
PC2					
Maišos lentelės tinklelis	**0,035**	0,734	**0,173**	**0,234**	1,679
Rekursinis aštuntainis medis	**0,035**	0,732	0,176	0,242	**1,607**
Tiesioginio k. tinklelis	0,038	**0,731**	0,181	0,258	1,641
Tiesioginio k. aštuntainis medis	0,043	0,737	0,196	0,295	1,738
PC3					
Maišos lentelės tinklelis	**0,031**	**0,475**	**0,094**	**0,197**	1,381
Rekursinis aštuntainis medis	0,034	**0,475**	0,100	0,203	**1,306**
Tiesioginio k. tinklelis	**0,031**	**0,475**	0,103	0,222	1,328
Tiesioginio k. aštuntainis medis	0,034	0,481	0,103	0,256	1,391

Iš lentelės duomenų matyti, kad beveik visais atvejais greičiau veikė klasikiniai metodai, be tiesioginio kreipimosi modifikacijų. Nors abiejų klasikinių metodų laikai labai panašūs, vis tik geriau pasirodė Maišos lentelės pagrindu veikiantis tinklelis, todėl ir buvo atrinktas tolesniems testams.

Toliau sekančioje Lentelė 4.6 yra pateikiami lygiagrečiai veikiančių tiesioginio kreipimosi tinklelio ir tiesioginio kreipimosi aštuntainio medžio algoritmų greitaveikos palyginimai. Kaip ir anksčiau, visų testų rezultatai yra 5 atskirų bandymų vidurkiai.

Lentelė 4.6 Lygiagrečių erdvės skaidymo algoritmų greitaveikos palyginimas

GPU					
Objektų pora	B + B	A + D	B + C	C + C	B + E
Trikampių skaičius	20700	346912	79801	138902	881768
PC1					
Tiesioginio k. tinklelis	**0,218**	**1,919**	**0,444**	**0,712**	**3,661**
Tiesioginio k. aštuntainis medis	0,275	1,928	0,456	0,737	3,706
PC1					
Tiesioginio k. tinklelis	**0,083**	**0,717**	**0,194**	**0,308**	**1,724**
Tiesioginio k. aštuntainis medis	0,086	0,723	0,201	0,318	1,738
PC1					
Tiesioginio k. tinklelis	**0,209**	**0,644**	**0,222**	**0,256**	**1,497**
Tiesioginio k. aštuntainis medis	**0,209**	0,656	0,228	0,263	1,506

Lentelė 4.6 matyti, kad iš lygiagrečiųjų algoritmų, praktiškai vienareikšmiškai greičiau veikė tiesioginio kreipimosi tinklelis. Tokį rezultatą galima paaiškinti tuo, kad abu metodai yra labai panašūs, tačiau tiesioginio kreipimosi aštuntainis medis turi papildomą žingsnį apjungiantį erdvės ląsteles. Šis žingsnis padeda sutaupyti algoritmui reikalingos atminties kiekį, tačiau naudoja papildomas operacijas, dėl to šis metodas ir buvo lėtesnis. Vis tik, galima įžvelgti, kad laikų skirtumas yra gan nežymus, tad ateityje būtų galima ląstelių apjungimo žingsnį nuodugniau optimizuoti ir taip gauti metodą kuris jei ir neaplenks tinklelio greičiu, tai bent jau pasižymės mažesnėmis atminties sąnaudomis.

4.7 Erdvės skaidymo algoritmų atskirų fazių laikų palyginimas

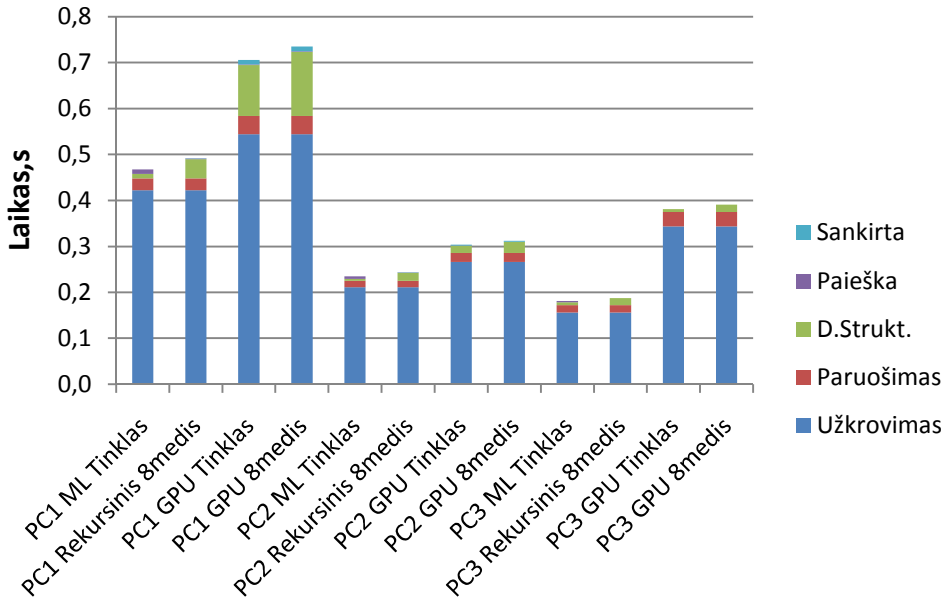
Siekiant išsiaiškinti kurios algoritmų dalys užima ilgiausiai laiko ir kurias dalis labiausiai apsimoka optimizuoti, buvo atlikti atskirų algoritmų fazių laikų palyginimai. Viso buvo išskirtos šios 5 algortimo fazės:

1. Duomenų užkrovimas iš kietojo disko
2. Tarpinių duomenų paruošimas (gaubiančių tūrių generavimas, normalių skaičiavimas)
3. Duomenų struktūrų paruošimas (tinklelio, aštuntainio medžio, maišos lentelės formavimas ir kt.)
4. Trikampių porų paieška duomenų struktūroje
5. Visų trikampių porų sankirtų skaičiavimas

Verta paminėti, kad algoritmą pritaikant realioje programoje, pirmas dvi fazes užtenka atlikti vieną kartą programos darbo pradžioje ir šiuos duomenis išsaugojus toliau galima vykdyti tik sekančias tris fazes. Priklausomai nuo algoritmų pritaikymo srities, taip pat yra galimybė nekartoti trečiosios fazės, jos rezultatus išsaugojant ir tik dalinai atnaujinant, tačiau šio tyrimo metu, toks funkcionalumas nebuvo realizuotas ir lieka kaip potencialus galimas algoritmų patobulinimams tolesniuose tyrimuose.

4.7.1 Nedidelės sankirtos srities testas

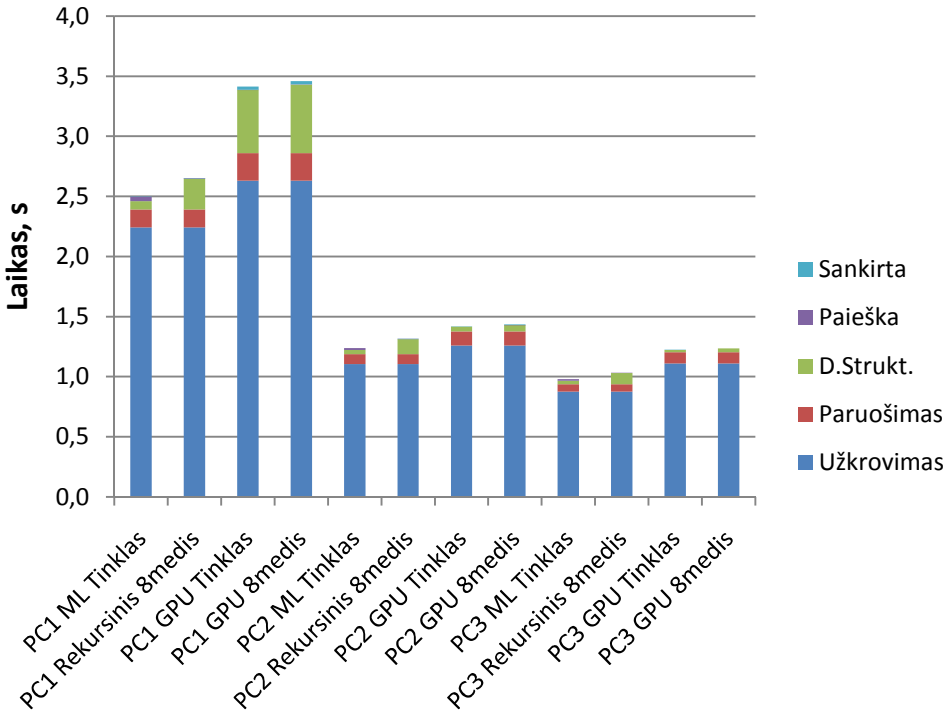
Taigi, pirmojo testinio atveju metu tarp dviejų C objektų besikertančių nedidele sritimi buvo vykdomi sankirtos nustatymai nuosekliais maišos lentelės tinklelio ir rekursinio aštuntainio medžio, bei lygiagrečiais tiesioginio kreipimosi tinklelio ir tiesioginio kreipimoi aštuntainio medžio erdvės skaidymo algoritmais. Žemiau esančiame Pav. 4.4. pateikiami testų rezultatai.



Pav. 4.4 C+C objektų poros nedidelės sankirtos srities testas

Kaip matyti iš Pav. 4.4, kuomet sankirtos tarp objektų sritis yra sąlyginai nedidelė, algoritme didžiausią laiko dalį užima duomenų užkrovimas. Pirmojo kompiuterio atveju sekanti ilgiausiai trunkanti fazė yra duomenų struktūrų formavimas, o antrojo ir trečiojo kompiuterio atveju apylygės duomenų struktūrų ir tarpinių duomenų paruošimo fazės. Trečiojo kompiuterio atveju, netgi galima įžvelgti, kad duomenų struktūrų generavimo fazė yra trumpesnė už tarpinių duomenų paruošimą. Tuo tarpu paskutinė sankirtos nustatymo fazė užima itin mažą laiko dalį.

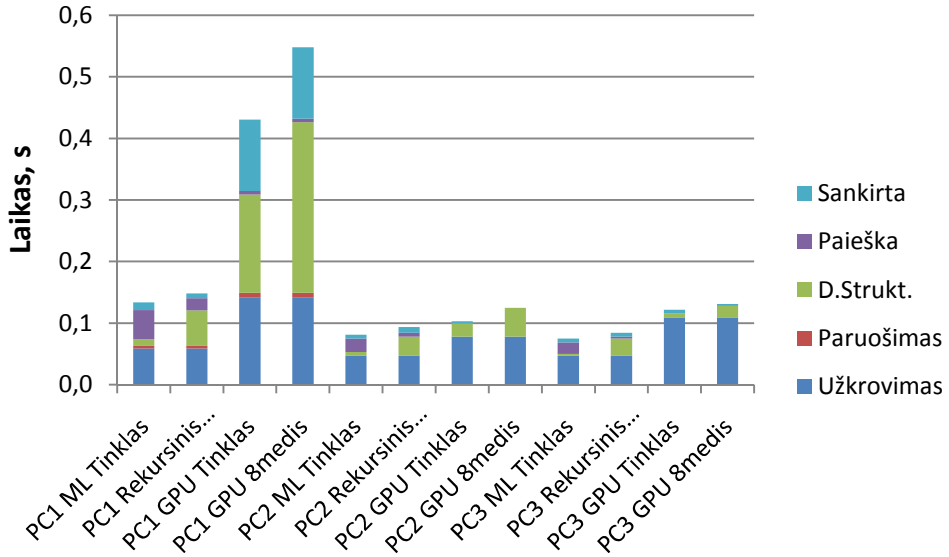
Analogiškame teste naudojant daugiau trikampių turinčią D objektų porą, galima įžvelgti tokias pačias tendencijas (žr. Pav. 4.5).



Pav. 4.5 D+D objektų poros nedidelės sankirtos srities testas

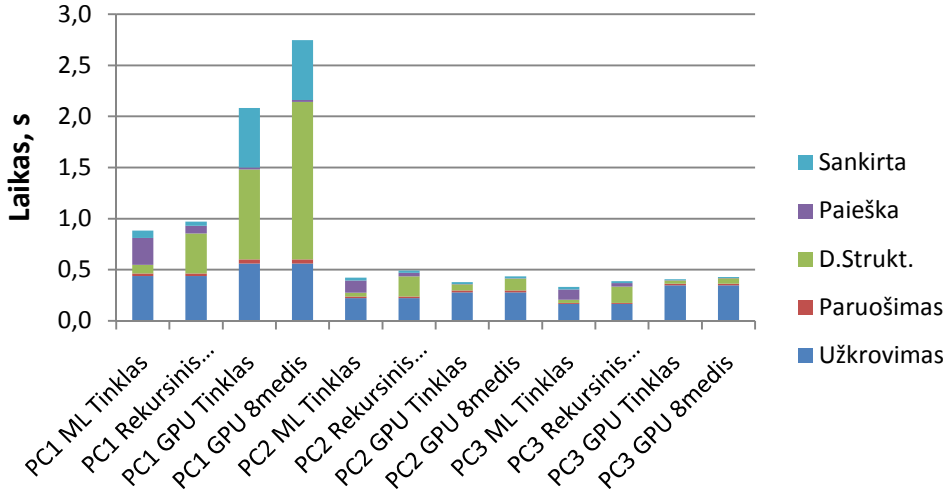
4.7.2 Didelės sankirtos srities testas

Šio eksperimento metu, buvo tikrinamos objektų poros, kurios praktiškai pilnai persidengia, t.y. du identiški modeliai sudėti vienas ant kito ir vienas iš jų per 0.1 reikšmę patrauktas į šoną. Žemiau esančiame Pav. 4.6 yra pavaizduoti B objektų poros didelės sankirtos testo rezultatai.



Pav. 4.6 B+B objektų poros didelės sankirtos srities testas

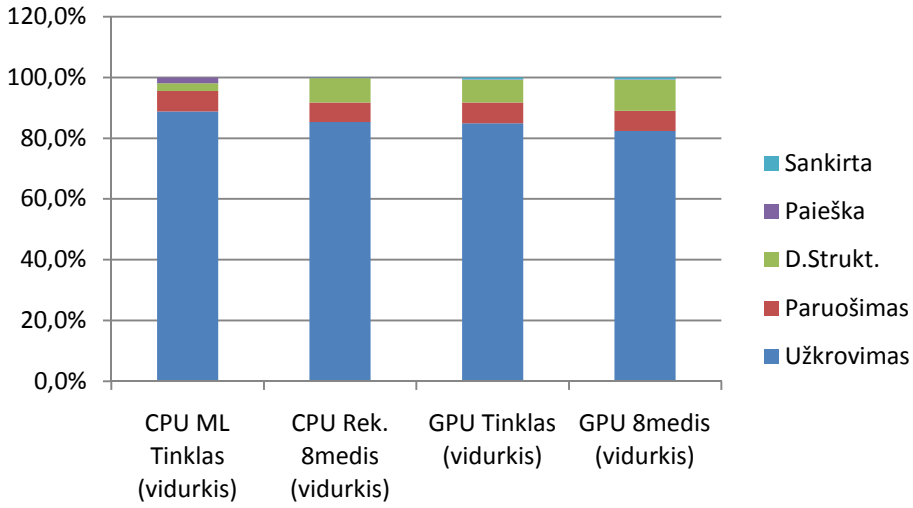
Kaip matyti iš Pav. 4.6, pirmojo kompiuterio atveju situacija pasikeitė, ir duomenų užkrovimas nebėra ilgiausia fazė, ją beveik visais atvejais pakeitė duomenų struktūrų paruošimas ir sankirtų nustatymas. Tuo tarpu antrajame ir trečiajame kompiuteriuose, dėsningumas išliko panašus į praeitų testų ir duomenų užkrovimas beveik visais atvejais išliko ilgiausiai trunkanti fazė. Galima išskirti, kad maišos lentelės tinklelio atveju, gerokai padidėjo paieškos duomenų struktūroje laikas. Taip yra dėl to, kad maišos lentelės dydis yra fiksuotas, ir didėjant į ją talpinamų duomenų kiekiui, atsiranda daugiau pasikartojančių maišos lentelės raktų susidūrimų (*angl. collision*) ir padidėja maišos lentelės grandinėlių ilgis, kartu ir paieškos trukmė. Tokias pat tendencijas galima įžvelgti ir Pav. 4.7, vaizduojančiame analogišką eksperimentą su daugiau trikampių turinčia D objektų pora.



Pav. 4.7 C+C objektų poros didelės sankirtos srities testas

4.7.3 Normalizuotų fazių laikų įvertinimas

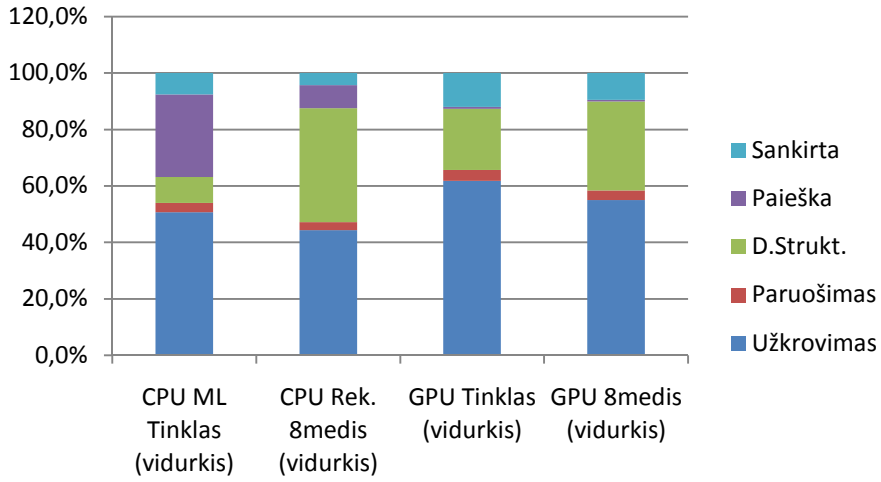
Paėmus visų trijų kompiuterių rezultatų vidurkius ir normalizavus trukmes, galima išreikšti algoritmų fazių trukmes procentine išraiška. Žemiau esančiame Pav. 4.8 pateikiami normalizuoti rezultatų vidurkiai, mažos sankirtos srities testui.



Pav. 4.8 Normalizuoti mažos sankirtos srities rezultatų vidurkiai

Iš Pav. 4.8 matyti jau minėta tendencija, kad kai sankirtos sritis yra maža, didžiąją dalį laiko užima duomenų užkrovimas, gerokai mažesnę dalį – tarpinių duomenų paruošimas ir duomenų struktūrų formavimas, mažiausiai – pačių sankirtų tikrinimas.

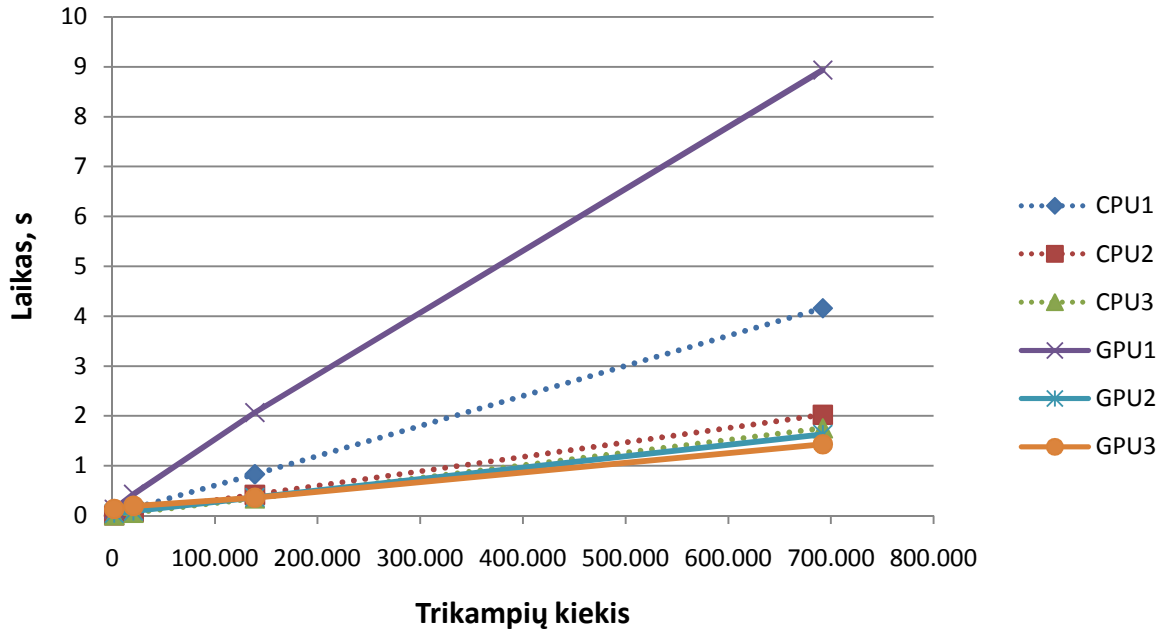
Žemiau esančiame Pav. 4.9 vaizduojančiame normalizuotus didelės sankirtos srities testo rezultatų vidurkius, galime įžvelgti, kad ilgiausiai ruošiama duomenų struktūra yra nuoseklus rekursinis aštuntainis medis, trumpiausiai ruošiama, tačiau ilgiausiai trunkančią paiešką turinti – maišos lentelės tinklelis. Abiejuose lygiagrečiuose algoritmuose paieškos fazė yra vykdoma labai greitai. Iš Pav. 4.9 grafiko atrodytų, kad lygiagretieji algoritmai sankirtų tikrinimui skiria nemažą laiko dalį, tačiau taip yra dėl itin lėtų pirmojo kompiuterio rezultatų kurie smarkiai prailgina skaičiavimų trukmę.



Pav. 4.9 Normalizuoti didelės sankirtų srities rezultatų vidurkiai

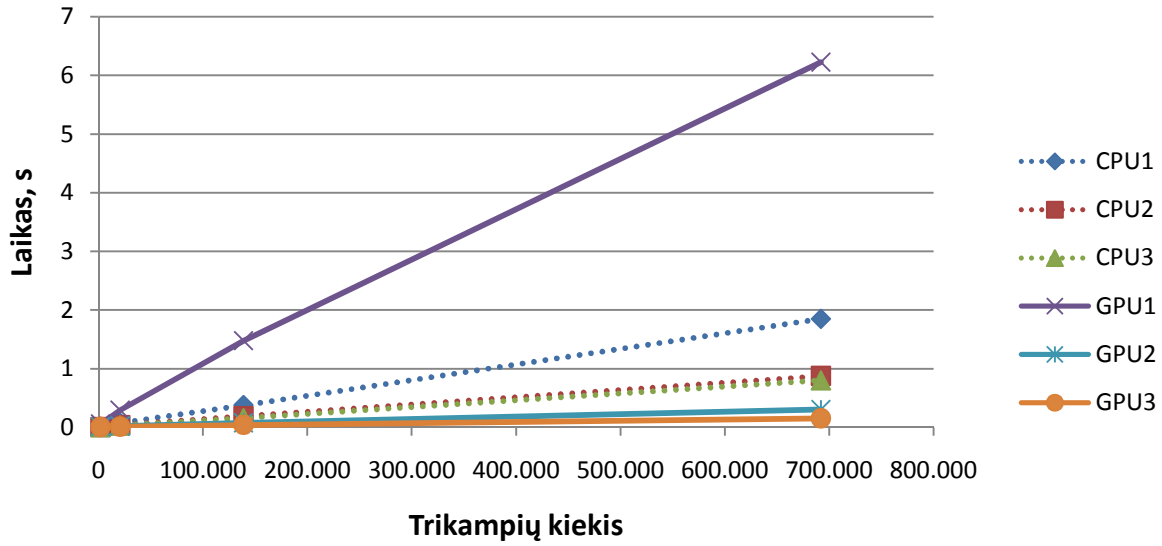
4.8 Geriausio CPU ir geriausio GPU algoritmų greitaveikos palyginimas

Nustačius geriausią nuoseklų algoritmą maišos lentelės tinklelį ir geriausią lygiagretų algoritmą tiesioginio kreipimosi tinklelį, atliktas jų tarpusavio greitaveikos palyginimas, kurio rezultatus galima matyti žemiau esančiame Pav. 4.10.



Pav. 4.10 CPU tinklelio ir GPU tinklelio algoritmų laikų palyginimas

Iš Pav. 4.10 matyti, kad naudojant naujesnes vaizdo plokštes, lygiagretus algoritmas yra nežymiai pranašesnis už nuoseklų, tačiau jei naudojama senos kartos vaizdo plokštė, lygiagretus algoritmas žymiai atsilieka. Iš algoritmų laikų atėmus duomenų užkrovimą ir tarpinių duomenų paruošimą, kurie, kaip anksčiau buvo minėta, gali būti atliekami tik kartą programos darbo pradžioje ir išsaugomi, galime matyti tikslesnį vaizdą, pavaizduotą žemiau esančiame Pav. 4.11.



Pav. 4.11 Tinklelių laikų be duomenų užkrovimo/paruošimo palyginimas

Iš Pav. 4.11 matyti tikslesnis algoritmų greitaveikos palyginimas, galima įžvelgti kad antrojo ir trečiojo kompiuterio rezultatuose tarp lygiagiagrečių ir nuoseklių algoritmų yra didesnis atotrūkis.

5. IŠVADOS

1. Möller trikampių sankirtos nustatymo algoritmas veikė greičiau, nei Chang ir Kim, dėl mažesnio tarpinių duomenų paruošimo operacijų skaičiaus ir mažesnių atminties sąnaudų.

2. Pilno perrinkimo algoritmų testai rodo, kad išlygiagretintų algoritmų pagreitėjimas priklauso nuo vaizdo plokštės kartos. 1.0 *Compute Capability* pasižyminčios vaizdo plokštės su mažesniu registrų skaičiumi vykdomas lygiagretus algoritmas veikė 2,7 karto lėčiau nei tame pačiame kompiuteryje vykdytas nuoseklus CPU algoritmas. Kitų dviejų vaizdo plokščių atveju, vidutinis lygiagretaus algoritmo pagreitėjimas lyginant su nuosekliu buvo 7,2 karto.

3. Pasiūlytos tiesioginio kreipimosi modifikacijos rekursiniam aštuntainiam medžiui ir maišos lentelės tinkleliui pasiteisino lygiagrečių algoritmų veikime, tačiau šioms modifikacijoms reikalingų duomenų struktūrų ir algoritmų pritaikytų lygiagrečiam veikimui, perkėlimas į CPU nepasiteisino. Lyginant su klasikiniais algoritmais padidėjo atminties sąnaudos ir vykdymo laikas.

4. Tiek rekursinis aštuntainis medis, tiek maišos lentele paremtas tinklelio algoritmas yra apylygiai savo greitaveika, tačiau pirmasis yra šiek tiek greitesnis kuomet objektų sankirtos sritis yra nedidelė (galimas pritaikymas kolizijų tikrinimui), o pastarasis yra greitesnis objektams su dideliais arti vienas kito esančiais trikampių kiekiais (galimas pritaikymas objektų modeliavime kietų kūnų geometrija (*angl. Constructive Solid Geometry*)). Maišos lentelės tinklelis buvo iki 12% lėtesnis esant nedidelėms objektų sankirtos sritims, tačiau iki 44% greitesnis atliekant sankirtą tarp dviejų identiškų objektų, perslinktų per nedidelį atstumą.

5. Lygiagretus tiesioginio kreipimosi tinklelis eksperimentuose pasirodė nežymiai greitesnis už lygiagretų tiesioginio kreipimosi aštuntainį medį, dėl greitesnio duomenų struktūros konstravimo laiko (nebuvo vykdomas ląstelių apjungimas). Tinklelis vidutiniškai veikė 1,03 karto greičiau.

6. Lyginant algoritmų fazių laikus, mažos sankirtos srities tyrime, visi algoritmai ilgiausiai užtruko duomenų užkrovimo fazėje (82-89%), tarpinių duomenų paruošimas užėmė 7%, o duomenų struktūrų sudarymas 3-10% bendro vykdymo laiko, kai tuo tarpu paieška duomenų struktūrose ir sankirtos tikrinimas užėmė mažiau nei 1%. Didelės sankirtos srities tyrime, duomenų užkrovimas nuosekliems algoritmams truko 44,4-50,7% bendro vykdymo laiko, ir 55,0-61,8% lygiagretiems algoritmams. CPU maišos lentelės tinklelis užtruko mažiausiai laiko struktūros formavimui (9,1%), tačiau daugiausiai laiko paieškai joje atlikti (29,3%). Rekursinis aštuntainis medis užtruko ilgiausiai generuodamas savo duomenų struktūrą (40,3%), ir 8,2% laiko užtruko paieškai joje vykdyti. Lygiagretūs algoritmai duomenų struktūras generavo 21,7-31,6% laiko, o trikampių porų paieškai jose atlikti sunaudojo mažiau nei 1% vykdymo laiko.

7. Geriausio CPU ir geriausio GPU algoritmo palyginimai rodo, kad 2.0 ir 5.2 *Compute Capability* pasižymintys GPU mažos sankirtos srities testus atlieka greičiau, kai trikampių kiekis yra 140000 ar daugiau. Taigi norint, kad ilgai trunkančios duomenų kopijavimo operacijos į GPU atsipirktų, reikia užtikrinti, kad kopijuojamų duomenų kiekis yra pakankamas didelis ir gali pilnai apkrauti grafinį procesorių skaičiavimais.

LITERATŪRA

[1] Wong, T., H.; Leach, G.; Zambetta, F. *Virtual subdivision for GPU based collision detection of deformable objects using a uniform grid*. 2012. [interaktyvus] [žiūrėta 2013-12-14] Prieiga per internetą: http://link.springer.com/article/10.1007%2Fs00371-012-0706-z/fulltext.html

[2] Fan, W. *et al. A Hierarchical Grid Based Framework for Fast Collision Detection*. 2011 [interaktyvus] [žiūrėta 2013-12-14] Prieiga per internetą: http://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2011.02019.x/full

[3] Pazouki,A.; Mazhar, H.; Negrut, D.*Parallel collision detection of ellipsoids with applications in large scale multibody dynamics*.2012.

[4] Bäckman, N. *Collision Detection of Triangle Meshes using GPU*, 2011. [interaktyvus] [žiūrėta 2014-01-14] Prieiga per internetą: http://umu.diva-portal.org/smash/record.jsf?searchId=1&pid=diva2:403566

[5] Ericson, C. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)*, 2005. ISBN:1-55860-732-3

[6] Ryoo, S. *et al.Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA*. 2008

[7] Cohen, J.*Fluid Simulation with CUDA*, 2009. [interaktyvus] [žiūrėta 2014-01-24] Prieiga per internetą: http://www.nvidia.com/content/GTC/documents/SC09_Fluid_Sim_Cohen.pdf

[8] Stein A.; Geva, E.; El-Sana, J.*CudaHull: Fast parallel 3D convex hull on the GPU*, 2012.

[9] Tang, M. *et al.GPU accelerated convex hull computation*, 2012.

[10] Le Grand, S. *Broad-Phase Collision Detection with CUDA* [interaktyvus] [žiūrėta 2013-12-14] Prieiga per internetą: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch32.html

[11] Klosowski, J., T. *et al.Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs*, 1998. [interaktyvus] [žiūrėta 2013-12-26] Prieiga per internetą: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=675649

[12] NVIDIA Corporation. *Cuda toolkit documentation v 6.0. Best practices guide*, 2014. [interaktyvus] [žiūrėta 2014-06-07] Prieiga per internetą: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[13] R. Inam. *An Introduction to GPGPU Programming - CUDA Architecture*, 2010 [interaktyvus] [žiūrėta 2014-06-11] Prieiga per internetą: http://www.diva-portal.org/smash/get/diva2:447977/FULLTEXT01.pdf

[14] J. Balfour.*Introduction to CUDA*, 2011. [interaktyvus] [žiūrėta 2014-06-11] Prieiga per internetą: http://mc.stanford.edu/cgi-bin/images/f/f7/Darve_cme343_cuda_1.pdf

[15] D. Tarjan.*CUDA memories*, 2011. [interaktyvus] [žiūrėta 2014-06-11] Prieiga per internetą: http://mc.stanford.edu/cgi-bin/images/5/5f/Darve_cme343_cuda_2.pdf

[16] Chang, J.;Kim, M. *Efficient triangle–triangle intersection test for OBB-based collision detection*, 2009. [interaktyvus] [žiūrėta 2014-12-07] Prieiga per internetą: http://www.sciencedirect.com/science/article/pii/S0097849309000430

[17] Möller, T. *A fast triangle-triangle intersection test*, 1997. [interaktyvus] [žiūrėta 2014-12-08] Prieiga per internetą: http://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/pubs/tritri.pdf

[18] NVIDIA Corporation.*Thrust :: CUDA Toolkit Documentation* [interaktyvus] [žiūrėta 2015-03-29] Prieiga per internetą: http://docs.nvidia.com/cuda/thrust

[19] Warren, M.,S.; Salmon, J., K.*A Parallel Hashed Oct-Tree N-Body Algorithm.*1993.

[20] Castro, R. *et al.Statistical optimization of octree searches.*2006.

[21] The Stanford 3D Scanning Repository, [interaktyvus] [žiūrėta 2015-05-21] Prieiga per internetą: https://graphics.stanford.edu/data/3Dscanrep/