KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
INFORMACINIŲ SISTEMŲ INŽINERIJOS STUDIJŲ PROGRAMA

MARTYNAS ŠIUKŠČIUS

SUSIDŪRIMŲ PAIEŠKOS, NAUDOJANT LYGIAGREČIUS SKAIČIAVIMUS, METODŲ TYRIMAS

Magistro darbas

Darbo vadovas
lekt. dr. Kęstutis Jankauskas

KAUNAS, 2013

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
INFORMACINIŲ SISTEMŲ INŽINERIJOS STUDIJŲ PROGRAMA

MARTYNAS ŠIUKŠČIUS

SUSIDŪRIMŲ PAIEŠKOS, NAUDOJANT LYGIAGREČIUS SKAIČIAVIMUS, METODŲ TYRIMAS

Magistro darbas

Darbo vadovas:
lekt. dr. Kęstutis Jankauskas
2013-05-24

Recenzentas:
lekt.dr. Šarūnas Packevičius
2013-05-24

Atliko:
IFM-1/1 gr. studentas
Martynas Šiukščius
2013-05-24

Kaunas, 2013

AUTORIŲ GARANTINIS RAŠTAS
DĖL PATEIKIAMO KŪRINIO

20.. - ..... - ..... d.

Kaunas

Autoriai, Martynas Šiukščius

patvirtina, kad Kauno technologijos universitetui pateiktas baigiamasis magistro (toliau vadinama - Kūrinys) Susidūrimų paieškos, naudojant lygiagrečius skaičiavimus, metodų tyrimas.

pagal Lietuvos Respublikos autorių ir gretutinių teisių įstatymą yra originalus ir užtikrina, kad

1) jį sukūrė ir parašė Kūrinyje įvardyti autoriai;
2) Kūrinys nėra ir nebus įteiktas kitoms institucijoms (universitetams) (tiek lietuvių, tiek užsienio kalba);
3) Kūrinyje nėra teiginių, neatitinkančių tikrovės, ar medžiagos, kuri galėtų pažeisti kito fizinio ar juridinio asmens intelektinės nuosavybės teises, leidėjų bei finansuotojų reikalavimus ir sąlygas;
4) visi Kūrinyje naudojami šaltiniai yra cituojami (su nuoroda į pirminį šaltinį ir autorių);
5) neprieštarauja dėl Kūrinio platinimo visomis oficialiomis sklaidos priemonėmis.
6) atlygins Kauno technologijos universitetui ir tretiesiems asmenims žalą ir nuostolius, atsiradusius dėl pažeidimų, susijusių su aukščiau išvardintų Autorių garantijų nesilaikymu;
7) Autoriai už šiame rašte pateiktos informacijos teisingumą atsako Lietuvos Respublikos įstatymų nustatyta tvarka.

Autorius

Martynas Šiukščius _____

(parašas)

TURINYS

LENTELIŲ SĄRAŠAS

PAVEIKSLIUKŲ SĄRAŠAS

Susidūrimų paieškos, naudojant lygiagrečius skaičiavimus, metodų tyrimas

Santrauka

Susidūrimų paieška - tai dviejų ar daugiau objektų susikirtimo radimas. Praktikoje susidūrimų paieška taikoma šiose srityse: kompiuteriniuose žaidimuose, netiesinėje baigtinių elementų analizėje, dalelių hidrodinamikoje, daugiafunkcinės dinamikos analizėje, įvairiose fizikos simuliacijose ir kt.

Egzistuoja daugybė susidūrimų paieškos algoritmų, iš kurių populiariausi yra erdvinio skaidymo, hierarchinio struktūrizavimo ir atrinkimo bei rūšiavimo metodai. Šiame darbe yra tiriamas šių algoritmų veikimas ant *CPU* (*Central processing unit*) ir ant *GPU* (*Graphics processing unit*), analizuojami susidūrimų paieškos nustatymo būdai bei nagrinėjamos pasirinktų algoritmų veikimo spartinimo galimybės panaudojant *CUDA* (*Compute Unified Device Architecture*) technologiją. Ši technologija yra Nvidia sukurta nauja duomenų apdorojimo architektūra išnaudojanti grafinio procesoriaus resursus bendro pobūdžio skaičiavimams. Darbe iškeltų tikslų pasiekimui yra realizuotos kelios bazinės algoritmų versijos, jų pritaikymo lygiagretiems skaičiavimams galimybės ir taip pat atliekami bazinių algoritmų laiko, reikalingo skaičiavimams atlikti, grafinio procesoriaus atminties sąnaudos bei įvairių veikimo laiką įtakojančių faktorių tyrimai. Darbo pabaigoje aptariami lygiagretaus programavimo privalumai pritaikant nagrinėjamai temai. Šiame darbe atlikti tyrimai parodė, jog perduodant skaičiavimus į *GPU* pasiekiamas 200 kartų didesnis nagrinėjamų algoritmų našumas negu atliekant skaičiavimus naudojant *CPU*.

Raktiniai žodžiai

Susidūrimų paieška, grafinis procesorius, skaičiavimų spartinimas, lygiagretus skaičiavimai.

Collision detection methods using parallel computing

Summary

Collision detection is a well-studied and active research field where the main problem is to determine if one or more objects collide with each other in 3D virtual space. Collision detection is an issue affecting many different fields of study, including computer animation, physical-based simulation, robotics, video games and haptic applications.

There is a big variety of collision detection algorithms of witch spatial subdivision, octree and sort and sweep are three of them. In this document we provide a short summary of collision detection algorithms, but the main focus will be on analyzing and increasing their performance working on *CPU* (*orig. Central processing unit*) and *GPU* (*orig. Graphics processing unit*) separately by making use of *CUDA* (*orig.Compute Unified Device Architecture*) technology. This technology is a part of Nvidia, witch helps the use of graphics processor for general-purpose computation. Main goal of this research is achieved by performing analysis of implemented spatial subdivision, octree and sort and sweep algorithms. This analysis consists of both general performance, parallelization performance and various performance affecting factors analyses. At the end of the document, the advantages of parallel programming adapted to the present subject are discussed. In this paper, research has shown that transfering calculations to the *GPU* can increase algorithm performance up to 200 times thant using *CPU*.

Keywords

Collision detection, GPU-based parallel computing, spatial subdivision.

1. TERMINŲ BEI SANTRUMPŲ ŽODYNAS

CPU	(*orig. Central processing unit*) loginis įtaisas, apdirbantis duomenų srautą
GPU	(*orig. Graphics processing unit*) loginis įtaisas, apdirbantis ir atvaizduojantis duomenų srautą.
CUDA	(*orig.Compute Unified Device Architecture*) NVIDIA sukurta (2006 metais) nauja duomenų apdorojimo architektūra išnaudojanti grafinio procesoriaus resursus (GPU).
DRAM	(*orig. Dynamic random-access memory*) operatyvioji kompiuterio atmintis.
host	CUDA literatūroje naudojamas terminas, kitaip vadinamas CPU.
GPGPU	(*orig. General-Purspose computation on Graphics Processing Unit*) – būdas, leidžiantis atlikti dažniausiai su centriniu procesoriumi atliekamus bendro pobūdžio skaičiavimus grafinio procesoriaus pagalba.
Vaizdo generavimas	(angl. *rendering*) - procesas, kurio eigoje yra sudaromas modelio atvaizdavimo paveikslas pagal pateiktus modelio duomenis.
ALU	(*orig. Arithmetic Logical Number*) - tai kompiuterio dalis, kuri atlieka visus aritmetinius skaičiavimus (pvz. sudėtį ar daugybą) ir taip pat visas palyginimo operacijas (pvz. *OR, AND, XOR*).
API	(*orig. application programming interface*) - tai protokolas skirtas naudoti kaip programinės įrangos komponentų sąsaja, kuri įgalina šių komponentų bendravimą tarpusavyje.
Rasterizacija	(*angl. rasterization*) - tai paveiksliuko, kuris yra vektorinis, pavertimas į paveiksliuką, kuris yra sudarytas iš pikselių arba taškų. Šis metodas naudojamas išvesties įrenginiams tokiems kaip monitoriams, spausdintuvams arba saugoti paveiksliuką *bitmap* formatu.
SDK	(*orig. Software development kit*) - programinės įrangos kūrimo įrankių rinkinys.
CGMA	(*orig. Compute to Global Memory Access*) - koeficientas, apskaičiuojamas lyginant atliekamų skaičiavimų kiekį su globalios atminties užklausų skaičiumi.

2. ĮVADAS

2.1 Problemos aktualumas

Praeityje Moore'o dėsnis teigė, jog skaičiavimų techninės įrangos galingumas kas du metus išauga beveik dvigubai ir kad bet kokia programa gali veiki greičiau ant naujesnio procesoriaus. Tačiau dabar pagal Moore'o dėsnį auga ne procesorių galingumas, bet procesoriaus branduolių skaičius. Rezultate, programų veikimas paspartės tik tuomet, kai jos efektyviai panaudos vis daugėjantį procesoriaus branduolių skaičių.

Kiekvienais metais vis labiau augantis grafinių (angl. *Graphics Processing Unit*; toliau – *GPU*) bei centrinių (angl. *Central Processing Unit*; toliau – *CPU*) procesorių pajėgumas bei besivystančios programinės technologijos, reikalaujančios vis daugiau skaičiavimų, verčia programinės įrangos kūrėjus ieškoti naujų būdų bei metodų jų efektyviam panaudojimui. Grafinio bei centrinio procesorių našumo augimo nevienodi tempai ir vis labiau didėjantis grafinio procesoriaus našumo atotrūkis paskatino bendro pobūdžio skaičiavimų vykdymo grafinio procesoriaus pagalba (angl. *General-Purpose computing on Graphics Processing Unit*; toliau – *GPGPU*) technologijų atsiradimą. Šios technologijos, tokios kaip *CUDA, OpenCL,* Accelerated Parallel Processing (*APP*), ir kt., įgalina bei supaprastina lygiagretinamų algoritmų įgyvendinimą naudojant *GPU*. *GPGPU* technologijos gali būti panaudotos sprendžiant bendrinio pobūdžio uždavinius. *GPGPU* vystymu buvo siekta užtikrinti būtent pastarųjų uždavinių spartinimą, tačiau *GPGPU* technologijų išpopuliarėjimą labiausiai paskatino įvairaus pobūdžio 2D bei 3D grafikos vaizdavimu pagrįsta programinė įranga. Vienas iš plačiausiai ir geriausiai žinomų tokios programinės įrangos pavyzdžių yra kompiuteriniai žaidimai. Būtent juose pradėtas naudoti *GPU* daug skaičiavimų reikalaujantiems uždaviniams spręsti. *GPU* pajėgumo augimas bei su tuo susijusių programinės įrangos technologijų kaita verčia permąstyti įvairių algoritmų realizacijos koncepcijas bei ieškoti naujų būdų esamų technologijų panaudojimui.

Ne išimtis yra ir susidūrimų paieškos algoritmų problema, kuri yra plačiai nagrinėjama skirtingų paskirčių procesorių panaudojimo kontekste. Dėmesys yra teikiamas *GPGPU* sprendimams ir kartu iškylančiai bendrai atliekamų skaičiavimų perkėlimo iš *CPU* į *GPU* problemai spręsti. Tai yra itin aktuali problema ir vis geresnių sprendimų radimo svarba šiame kontekste didės.

2.2 Tyrimo sritis ir objektas

Susidūrimų paieškos algoritmai tarp geometrinių objektų yra žinomi kaip pagrindiniai uždaviniai daugelyje sričių, tokiuose kaip kompiuteriniai žaidimai, fizikos simuliacijos, robotų judesio planavimas ir kt. Priklausomai nuo modelio struktūros, užklausų tipo ir simuliavimo aplinkos, yra naudojami skirtingi susidūrimų paieškos algoritmai, kurie yra skirtingai optimizuojami konkrečiai situacijai. Tradiciniai susidūrimų paieškos algoritmai yra atliekami ant *CPU*, naudojant įvairius metodus (pvz. aprėpties dėžes). Daugumai aukščiau minėtų sričių, susidūrimų paieškos algoritmai tampa veikimo našumo trikdžiais, nes šie algoritmai yra skaičiavimams reiklūs procesai. Tai paskatino plataus masto mokslinius tyrimus tobulinant susidūrimų paieškos algoritmus, panaudojant lygiagrečius skaičiavimus.

2.3 Tyrimo tikslas ir uždaviniai

Šios tiriamojo pobūdžio analizes tikslas yra išanalizuoti realizuotų tradicinių susidūrimų paieškos algoritmų našumo rezultatus. Realizacijai bus panaudotas lygiagretaus programavimo įrankis *CUDA*, kuris visus algoritmo skaičiavimus perduoda grafinei plokštei (angl. *GPU*) atlikti. Tikslas išskaidytas į tokius uždavinius:

1. Tradicinių susidūrimų paieškos algoritmų rezultatų tyrimas perduodant skaičiavimus skirtingos paskirties procesoriams.
2. Ištirti laiko bei kadrų dažnio priklausomybę nuo objektų skaičiaus keičiant algoritmų parametrus.

3. Ištirti grafinio procesoriaus atminties sąnaudas reikalingas realizuotiems algoritmams.

Darbe atlikta išsami esamų tradicinių susidūrimų paieškos algoritmų analizė, pasirinktų susidūrimų paieškos algoritmų pritaikymas pasitelkiant skirtingos paskirties procesorius (*CPU* ir *GPU*) bei jų našumo palyginimas.

Šio darbo analizei atlikti, prireiks programavimo žinių rašant algoritmus C++ kalba. Bus pasitelkiamas Microsoft Visual Studio įrankio paketas bei OpenGL biblioteka, kuri atvaizduoja gaunamus rezultatus 3D erdvėje. Vykdant nuosekliai visus šios analizės darbus, pirmiausiai realizuojami pasirinkti algoritmai ir analizuojamas jų veikimas naudojant *CPU* skaičiavimams atlikti, o vėliau - *GPU*

2.4 Dokumento struktūra

Šio darbo dokumentacijos struktūra yra paremta užsibrėžtais tikslais, įvykdant juos nuosekliai ir papunkčiui. Etapai, į kuriuos bus atsižvelgta, yra šie:

1. Išanalizuojami su šia tema susiję algoritmai bei naudojamos technologijos.
2. Pasirinkti tinkamiausi algoritmai ir metodai užduočiai atlikti.
3. Panaudojant skirtingos paskirties procesorius (*CPU* ir *GPU*) išanalizuoti algoritmų našumo skirtumus.

3. ANALITINĖ DALIS

3.1 Susidūrimų paieškos algoritmų analizė

Susidūrimų paieška tarp 3D objektų yra svarbus fizikos simuliacijų, kompiuterinio dizaino, molekulinio modeliavimo ir kitų aplikacijų komponentas. Vienas iš efektyvesnių būdų yra trijų etapų metodas. Pirmasis etapas vadinamas plačiuoju (angl. *broad phase*). Šiame etape sankirtų radimo algoritmai nenaudoja sudėtingų skaičiavimų, o atsako i klausimą: "Kurie objektai turi didelę galimybę susidurti?". Šiam klausimui atsakyti, dažniausiai naudojamos aprėpties dėžės (angl. *bounding boxes*). Šis etapas yra greitas, nenaudoja daug skaičiavimo resursų, todėl yra patogus ir naudingas pradiniame sankirtų radimo algoritmų etape, tačiau objektų sankirtos nėra tikslios.

Po pirmojo etapo atliktų veiksmų, toliau vykdomas tarpinis etapas (angl. *mid-phase*). Priklausomai nuo objektų struktūros sandaros šis etapas gali būti nereikalingas ir praleidžiamas. Šis etapas aprašytas *3.4.2* dalyje.

Atrinkus tuos objektus, kurie turi didžiausią galimybę susidurti su kitais objektai, pradedamas trečiasis etapas, vadinamas siauruoju (angl. *narrow phase*). Šiame etape objektų sankirtos yra tikslinamos, todėl šis etapas yra brangus skaičiavimų atžvilgiu. Siaurasis etapas yra naudingas, kai yra daugybė objektų atkritusių pirmajame etape.

Susidūrimų paieška yra svarbi dalis beveik kiekvienoje fizikos simuliacijoje, įskaitant specialius efektus žaidimuose ir filmuose ir taip pat moksliniuose tyrimuose. Iš esmės, susidūrimų paieška gali būti skirstoma į du modulius:

- Susidūrimų aptikimas (angl. *collision detection*).
- Susidūrimų reagavimas (angl. *collision response*).

Pirmajame žingsnyje telkiamas dėmesys į susidūrimų aptikimą bei specifinės informacijos perdavimą antrajam žingsniui, kad šis galėtų apskaičiuoti pakeitimus. Šiame darbe bus domimasi pirmuoju moduliu.

Susidūrimų paieškos algoritmai gali būti kvalifikuojami į kategorijas:

- nepertraukiami (angl. *continuous*)
- diskretūs (angl. *discrete*)

Nepertraukiami susidūrimų paieškos algoritmai naudoja laiko parametrizuojamas lygtis apskaičiuoti dviejų objektų tarpusavio pradinį susidūrimo kontaktą ir taip pat jo būseną. Tuo tarpu diskretūs algoritmai nagrinėja objektų trajektorijas bei susidūrusių objektų skvarbą vienas į kitą. Šiame darbe bus domimasi pirmąja kategorija.

Susidūrimų paieškos sistema gali būti suskaidyta į tris pagrindinius aspektus [13]:

- platusis etapas (angl. *broad phase*).
- tarpinis etapas (angl. *mid-phase*).
- siaurasis etapas (angl. *narrow phase*).

3.1.1 Platusis etapas

"Brutalios jėgos" (angl. *brute force*) algoritmo įgyvendinimas pirmajame susidūrimų paieškos etape, turint n objektų sudaro atliekant $n\,(n-1)/2$ susidūrimų paieškos testų, todėl turi $O(n^2)$ sudėtingumą. Alternatyvūs algoritmai [14], pavyzdžiui tokie kaip rūšiavimas (angl. *sort*) ir nurašymas (angl. *sweep*) arba erdvinis skaidymas (angl. *spatial subdivision*) pasiekia vidutini $O(n \log n)$ susidūrimų paieškos testų kiekį ir geriausiu atveju gali turėti $O(1)$ sudėtingumą, tačiau blogiausiu atveju sudėtingumas lieka $O(n^2)$.

Plačiajame etape susidūrimų paieškos algoritmuose siekiama pašalinti tas objektų poras, kurios nesusiduria virtualioje erdvėje, išmetant tiek objektų kiek įmanoma daugiau. Visi objektai esantys virtualioje aplinkoje yra sudedami į aprėpties dėžes, kurios supaprastina susidūrimų paiešką. Tuomet

kiekvieno objekto aprėpties dėžė yra tikrinama su kiekviena kita dėže ar jos nesusiduria. Tuo atveju, jeigu dviejų objektų aprėpties dėžės nesikerta, tuomet šią objektų porą galima atmesti ir toliau jos į susidūrimų paieškos algoritmą įtraukti nereikia. Tik tuo atveju, jei dviejų objektų aprėpties dėžės kertasi viena su kita, šiems objektams reikalingas tolimesnis susidūrimų paieškos algoritmų procedūros etapas. Plataus etapo galutinis rezultatas yra potencialių besikertančių objektų rinkinys.

3.1.2 Tarpinis etapas

Šiame etape, rinkinys, kuris buvo gautas iš susidūrimų paieškos plačiojo etapo, toliau yra tikrinamas. Priklausomai nuo objekto sudėtingumo struktūros, gali būti naudojama aprėpties dėžių hierarchija, kurią reikalinga patikrinti atliekant susidūrimų paieškos algoritmą. Jeigu objektai virtualioje aplinkoje yra paprasti (pvz. išgaubti), tuomet aprėpties dėžių hierarchija nebūtina ir šį etapą galima praleisti. Tuo tarpu, jei objektai yra sudėtingos struktūros (pvz. neišgaubti), aprėpties dėžių hierarchija tuomet yra reikalinga. Pastaruoju atveju, kiekviena potenciali besidurianti objektų pora yra patikrinama ar ji nesikerta su atitinkamomis hierarchijos aprėpties dėžėmis. Rezultate, gaunamos susidūrusios objektų poros, kurių aprėpties dėžės taip pat kertasi.

3.1.3 Siaurasis etapas

Kai objektų poros, kuriose objektai potencialiai kertasi tarpusavyje, pasiekia susidūrimų paieškos siaurąjį etapą, konkretus susidūrimų paieškos algoritmas yra atliekamas patikrinti esamų objektų geometrijos. Rezultate, susidūrimų paieškos sistema praneša, kad susidūrimas įvyko arba neįvyko tarp dviejų objektų. Pirmuoju atvejų, jeigu susidūrimų paieškos sistema yra fizikos variklio sistemos dalis, tuomet siaurasis etapas turi pranešti besikertančias objektų poras ir taip pat jų susidūrimo tašką. Gavus šiuos rezultatus iš siaurojo etapo, šie duomenys pateikiami į susidūrimų reagavimo modulį, kuriame apskaičiuojami veiksmai po susidūrimo. Šiame darbe susidūrimų reagavimas nebus analizuojamas.

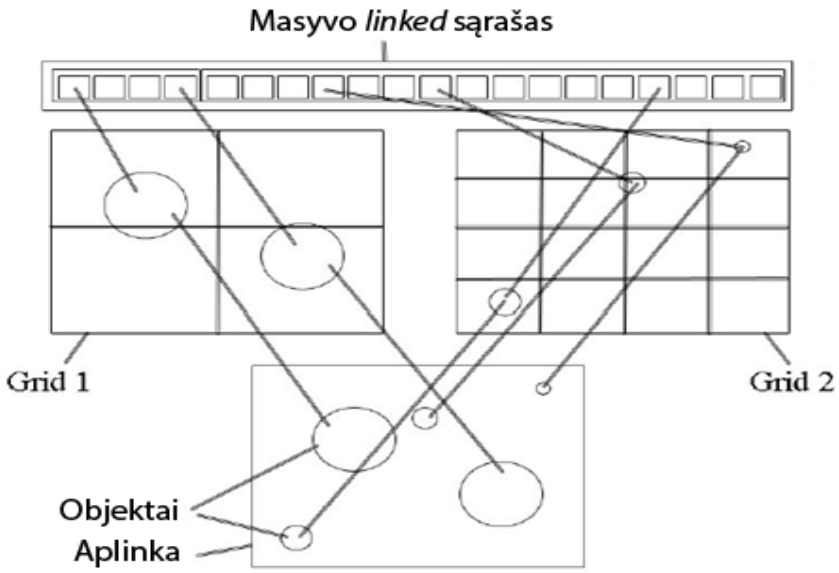
3.1.4 Erdvės padalijimo algoritmas

Pradžioje, naudojant *grid* algoritmą, aplinka yra suskaidoma į 8 vienodas dalis. Sekančio lygio *grid* struktūra turės 64 dalis ir t.t.. Skirtingai nuo *octree*, *grid* dengia visą nagrinėjamą aplinką, o ne tik mažas jo dalis, kaip žemesnio lygio octree medis.

Vienu laiko momentu, yra tik vienas *grid* algoritmo mazgas, kuris saugo rodykles į kiekvieno *grid* esančių dalių *linked* sąrašus. Tai yra pasiekiama, naudojant objektų rodyklių sąrašą, kur kiekviena rodyklė rodo į pirmąjį *linked* sąrašo objektą. Kiekviena masyvo pozicija atstovauja tam tikrą *grid* dalį. Šiuo atveju, pirmos 8 pozicijos priklauso pirmajam grid, sekančios 64 pozicijos priklauso sekančiam grid ir t.t..

Norint įrašyti objektą į *grid* algoritmo struktūrą, reikia patikrinti objekto aprėpties sferos diametrą ir taip pat grid dalių dydį. Naudojant *grid* algoritmą, yra apibrėžta, jog aprėpties sferos diametras negali viršyti ketvirtadalio *grid* dalies. Jei aprėpties sfera yra didesnė, tuomet ji yra perduodama kitam grid, kurio dalys yra didesnės. Yra galimybė nustatyti skirtingas paklaidas tikrinant aprėpties sferas, tačiau jų spindulys turi būti ne didesnis kaip *grid* dalies pusė. Tai yra svarbi šio algoritmo sąlyga vykdant susidūrimų paieškos testus. Tam, kad nuspręsti, kurioje *grid* dalyje ir masyvo vietoje saugoti objektą ir jo rodyklę, naudojama erdvinė *hash* funkcija [8], kuri naudoja aprėpties sferos koordinates bei *grid* lygį, į kurį bus įrašomas objektas.

3.1 pav. iliustruoja kaip tai yra pasiekiama dvimatėje erdvėje. Pirmosios keturios pozicijos, esančios masyve, yra atsakingos saugant objektų rodykles susijusias su pirmuoju masyvu. Tuo tarpu, sekančios 16 pozicijos yra susijusios su aukštesnio lygio *grid*.



3.1 pav. Aplinkos suskaidymas į 2 grid

Naudojant tam tikras *hash* funkcijas, reikalauja didelių skaičiavimo resursų, tačiau jos grąžina tikslius rezultatus. Šios funkcijos įgalina kiekvieną *grid* dalį sujungti su konkrečiu *hash* skaičiumi. To rezultate, gaunamas mažesnis galimai besidūriančių objektų skaičius.

Objektų pašalinimas iš *grid* yra lengvas ir greitas procesas, dėl *grid* dalių numeracijos, kuri buvo atlikta įrašymo metu. Tokiu būdu, yra žinoma, kurioje masyvo dalyje objektas randasi. Taip pat yra galimybė ieškoti objekto naudojant objektų vidinės *grid* dalies numerį. Tokiu atveju yra naudojamos erdvinės *hash* funkcijos.

Šiame *grid* algoritme susidūrimų paieška atliekama tikrinant objektų aprėpties sferas su kitomis aprėpties sferomis, esančiomis toje pačioje *grid* dalyje. Skirtingai nuo *octree* algoritmo, objektai grid struktūroje yra įrašomi į tas dalis, kuriose yra aprėpties sferų centro taškai. Taip pat yra galimybė, jog aprėpties sfera gali būti didesnė už grid dalies pusę dydžio. To rezultate, objektai gali susidurti su kitais objektais, kurie randasi ne gretimose *grid* dalyse, o šiek tiek toliau. Norint nuspręsti kurias *grid* dalis ir kuriuos objektus tikrinti susidūrimų paieškai, galima panaudoti *hash* funkcijas, paduodant joms aprėpties sferų spindulį ir *grid* dalių dydį. Šis procesas atliekamas visiems esantiems grid. Šio algoritmo pseudo kodas aprašytas žemiau (žr. *Grid_CheckForCollisions*).

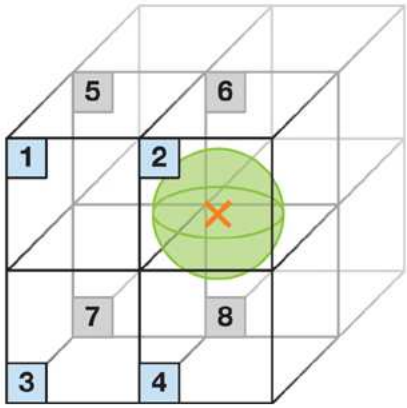
```
Procedure Grid_CheckForCollisions
// objektas X = objektas, kuris bus naudojamas tikrinant grid struktūroje

{01}begin
{02}  For all grid levels do
{03}  {
{04}     Q = hasfunction(X, gridlevel, grid section size)
{05}     X does naive cillision detection agains grid section Q's linked list
{06}     probedistance = radius of bounding sphere + grid section size * object to grid ratio
{07}     for all directions
{08}     {
{09}        Y = hashfunction(X + probedistance, gridlevel, grid section size)
{10}        X does naive cillision detection agains grid section Y's linked list
{11}     }
{12}  }
{03}end.
```

3.1.5 Lygiagretus erdvės padalijimo algoritmas

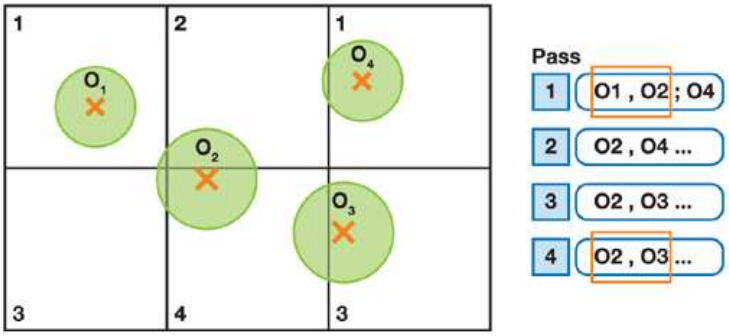
Erdvės padalijimas yra vienas iš tradicinių susidūrimų paieškos algoritmų. Šis algoritmas padalija erdvę į tolygų tinklelį, taip, kad tinklo gardelė yra tokio dydžio kaip didžiausias esamas objektas. Kiekviena tinklo gardelė turi objektų sąrašą, kurie priklauso tai gardelei. Sankirtų testai tarp dviejų objektų atliekami tik tuomet kai abu objektai priklauso tai pačiai tinklo gardelei arba yra greta

esančiose. Arba vienas ir tas pats objektas gali priklausyti kelioms gardelėms. Tai priklauso nuo tinklelio gardelių bei objektų dydžio (žr. 3.2 pav. [7]).



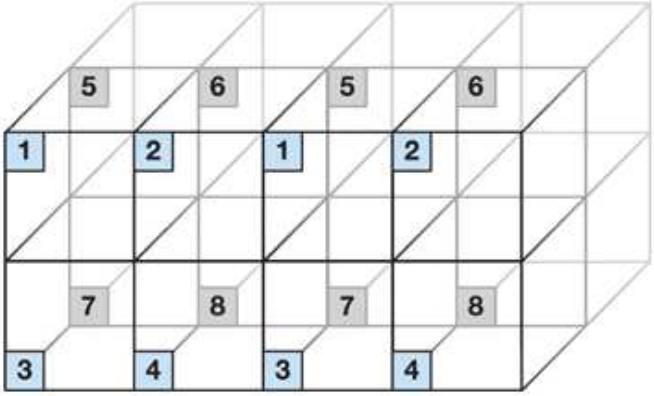
3.2 pav. Sfera 3d erdvėje, kuri priklauso $\mathbf{2^3 = 8}$ tinklo gardelėms

Lygiagretus erdvės padalijimo algoritmas (angl. *grid*) yra sudėtingesnis, kadangi vienas ir tas pats objektas gali priklausyti daugiau nei vienai tinklo gardelei viename laiko momente. Tai pasunkina išlygiagretinimo procesą, todėl turi egzistuoti tikrinimai, kad to paties objekto sankirtos testai nebūtų tikrinami kelis kartus tuo pačiu metu. Norint išspręsti šią problemą, reikalinga panaudoti gardelių erdvinį atskirimą (angl. *spacial separation*) ir taip pat kiekviena tinklo gardelė turi būti didesnė už didžiausią esamą objektą. Laikantis šių taisyklių vykdomi atskiri kiekvienos tinklo gardelės objektų sankirtų testai. 2d erdvėje tai reiškia, jog reikalingi keturi skaičiavimo patikrinimai, kurie apima visas galimas tinklo gardeles (žr. 3.3 pav. [7]). Šioje pavyzdinėje iliustracijoje pavaizduota kaip yra atskirtos dvi sankirtos testų poros, kurioms priklauso tas pats objektas O_2.



3.3 pav. Objektų sankirtos testų atskirimas

Esant 3d erdvei, reikalingi aštuoni skaičiavimo patikrinimai bei tinklo gardelių sunumeravimas, kaip pavaizduota iliustracijoje (žr. 3.4 pav. [7]).



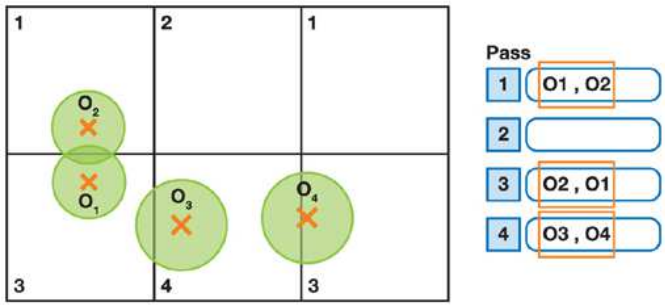
3.4 pav. Erdvinio suskaidymo gardelės, kurias galima panaudoti lygiagretiems skaičiavimams

Yra du specialūs atvejai, kuomet reikia pernumeruoti tinklo gardeles atvirkštiniu būdu. Pirmuoju atveju, kai objekto būsena nesikeičia, bet tik atliekami sankirtos testai. Šiuo atveju, gardelės, kuriose yra tokie objektai, galima apjungti ir atlikti sankirtų testus kartu.

Kita komplikacija atsiranda kuomet reikia išlygiagretinti skaičiavimus yra ta, jog reikia saugotis atlikti tų pačių objektų sankirtos testus kelis kartus vienu laiko momentu. Tai įvyksta kuomet du objektai užima kelias tinklo gardeles ir jų centro taškai nėra toje pačioje gardelėje. Ši problema įveikiama laikantis taisyklų:

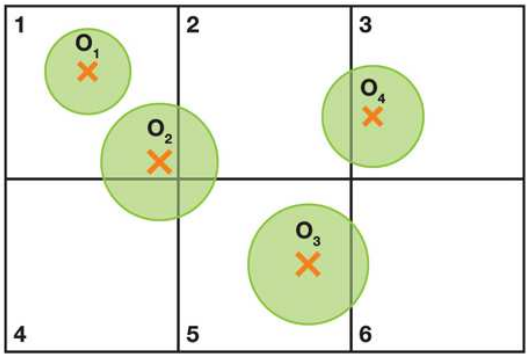
- Kiekvienam objektui išskirti $d + 2^d$ kontrolinius bitus, kur *d* saugo informacija kurioje tinklo gardelėje randasi objekto centro taškas, o 2^d - tinklo gardelių informacija, kurias liečia objektas savo aprėpties dėže.
- Kiekvieną objektą sumažinti $sqrt(2)$ karto ir užtikrinti, kad tinklelio gardelės yra bent pusantro karto didesnės už sumažintą didžiausią objektą.

Prieš vykdant dviejų objektų sankirtos testą laiko momentu *T*, reikia sužinoti tinklo gardelės, kurioje randasi objektai, tipą bei identifikavimo numerį *T'*. (angl. *object's home cell*). Šiame pavyzdyje (žr. 3.5 pav. [7]) objektų O_1ir O_2sankirtos testas yra praleidžiamas laiko momentu $T = 3$, nes abu objektai dalinasi gardeles pažymėtas numeriais 1 ir 3, bei objekto O_2centro taškas priklauso gardelei su identifikavimo numeriu $T' = 1$. Šis dviejų objektų sankirtos testo praleidimas yra saugus, nes jis buvo atliktas laiko momentu $T = 1$. Tuo tarpu objektų O_3 ir O_4 sankirtos testas praleidžiamas laiko momente $T = 4$, nes abu objektai dalinasi bendras tinklo gardeles su numeriais 3 ir 4. Šių objektų sankirtos testo praleidimas yra saugus. Jų tarpusavio sankirta negalima, nes jie perdengia skirtingas tinklo gardeles su identifikavimo numeriu 3, todėl jų centro taškų nuotolis yra didesnis už esamą didžiausio objekto skersmenį.



3.5 pav. Iliustruotas pavyzdys, objektų grupavimas

Sankirtos testai tarp dviejų objektų vykdomi tik tuomet kai jie abu atsiranda vienoje tinklo gardelėje ir bent viena iš jų centro taškas priklauso tai pačiai gardelei. Pavyzdžiui (žr. 3.6 pav. [7]) sankirtų testas atliekamas tarp objektų O_1ir O_2, nes abu objektai turi centro taška toje pačioje tinklelio gardelėje nr. 1 ir taip pat sankirtos testas bus atliekamas tarp O_2 ir O_3, nes abu objektai priklauso gardelei nr. 5 ir objekto O_3 centro taškas priklauso minetai gardelei. Tačiau tarp objektų O_2 ir O_4 sankirtos testas nebus vykdomas, nes abiejų centro taškai yra skirtingose tinklo gardelėse, nors jie abu ir priklauso vienai gardelei nr. 2.



3.6 pav. Sfera 3d ervėje, kuri priklauso $\mathbf{2^3 = 8}$ tinklo gardelėms

Paprasčiausio erdvinio skaidymo įgyvendinimo metu kiekvienam objektui sukuriamas identifikavimo numeris, kuris nurodo, kurioje tinklo gardelėje jis randasi ir pagal juos vykdomi sankirtos testai.

Vienas iš būtų pagerinti susidūrimų paieškos algoritmų efektingumą, yra pertvarkyti virtualią aplinkos erdvę į tinklelio gardeles (angl. *grid cells*). Tuomet kiekvienoje tinklo gardelėje galima atlikti brutalios jėgos metodą, reikalingą susidūrimų paieškai. Pagrindinis privalumas naudojant tinklus, yra tas, jog duomenų struktūros gali būti naudojamos statiškai. Tai reiškia, kad tinklelio detalumas neturi būti atnaujinamas kiekviena kartą, ir kad objektai esantys tinkle gali būti atnaujinami labai greitai. Iš tiesų, tinklo detalumas gali būti nekintamas programos veikimo metu, bet apskaičiuojamas išankstinio apdorojimo metu. Tačiau tinkleliai taip pat turi ir rimtų apribojimų:

- Sunku įvertinti tinklo gardelių detalumą skirtingiems scenarijams.
- Yra tikimybė, jog objektai gali priklausyti kelioms tinklo gardelėms vienu laiko momentu. Tai reikalauja papildomų tikrinimų bei atminties.
- Tinklas neprisitaiko prie objektų išsidėstymo scenoje.

Tam, kad sukurti 3d aplinkos tinklą, pirmiausia sukuriamas *AABB* (angl. axis aligned bounding box), kuris apima visą erdvę, reikalingą susidūrimų paieškai atlikti. Tuomet sukurtas *AABB* yra užpildomas mažesniais, nepersidengiančiais *AABB*, priklausomai nuo tinklo detalumo lygio, nurodyto prieš programos paleidimą. Kiekviena iš šių *AABB* atitinka tinklo konkrečiai gardelei. Todėl, norint sužinoti objekto poziciją tinklelyje, tiesiog reikia patikrinti su kuriomis tinklo gardelėmis persidengia tikrinamo objekto aprėpties dėžė. Ta gardelė, kurioje yra objekto aprėpties dėžės centro taškas, gali būti laikoma kaip pradinė gardelė susidūrimų paieškai atlikti ir tik po to išsiaiškinti likusias kitas. Tuomet, kai visi objektai yra sutalpinti į atitinkamas tinklo gardeles, reikia pereiti per tas gardeles, kuriose įvyko objektų atnaujinimas, ir perduoti jas susidūrimų paieškos algoritmams.

3.1.6 Brute Force algoritmas

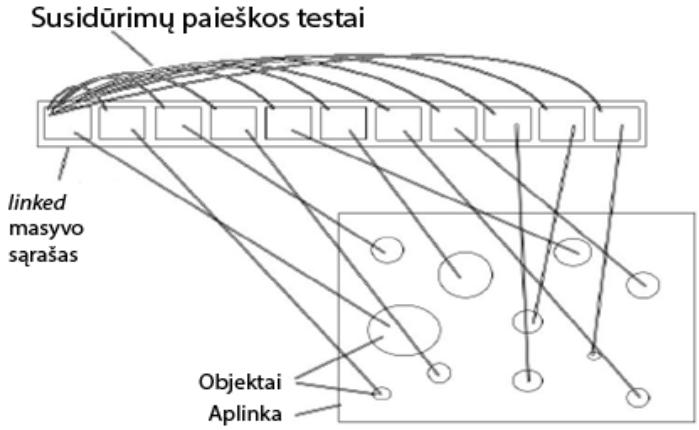
Šis paprastas algoritmas saugo objektų rodykles į esamus objektus *linked* sąraše. Kiekvieno objekto rodyklė rodo į sekančio objekto rodyklę. Svarbu nepamiršti to, kad *linked* sąrašas saugo ne pačius objektus, bet tik jų rodykles. Objektai saugoji atskirai.

Objekto pozicija erdvėje ir dydis neturi reikšmės kurioje *linked* sąrašo vietoje objektas bus įrašytas, tai atliekama atsitiktiniu būdu ir neturi reikšmės vykdant šį algoritmą. 3.7 iliustracijoje pavaizduota algoritmo struktūra ir kaip pirmasis *linked* sąrašo objektas yra tikrinamas su kitais likusiais objektais vykdant susidūrimų paiešką.

Objektų rodyklės yra įrašomos į sąrašo pradžią. Tai galima atlikti bet kuriuo laiko momentu dėka paprastos *linked* sąrašo struktūros.

Norint ištrinti konkrečia objekto rodyklę, reikia eiti per sąrašą tol, kol pasiekiama norimo objekto rodyklė. Kai tai įvyksta, tuomet ši rodyklę galima ištrinti.

Susidūrimų paieškos testai atliekami pereinant per turimą objektų rodyklių sąrašą ir vykdant susidūrimų paieškos testus su kiekvienu esančiu objektu. Šio algoritmo pseudo kodas yra aprašytas žemiau (žr. *BruteForce_CheckForCollisions*).



3.7 *Brute force* algoritmo struktūra

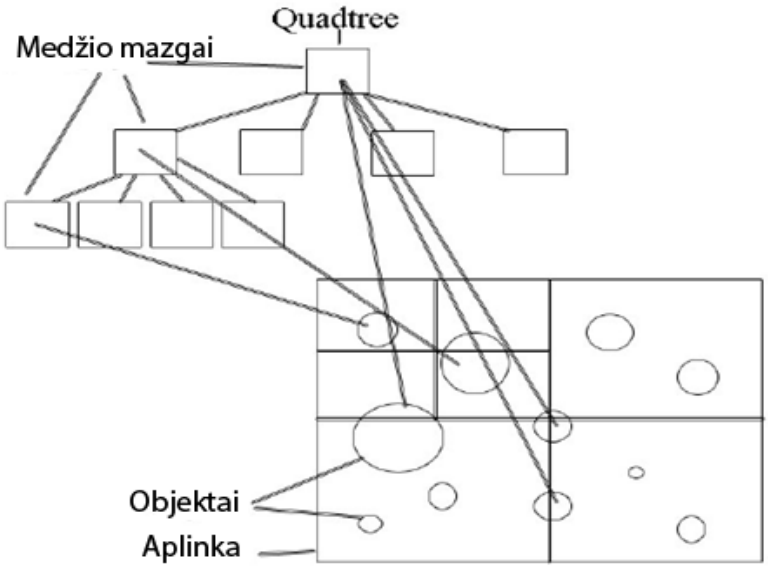
```
Procedure BruteForce_CheckForCollisions
// objektas X = pirmas objektas linked sąraše
{01}begin
{02}  Repeat while X is not Null
{03}  {
{04}     Object Y = First objekct in linked list
{05}     Repeat while Y is not Null
{06}     {
{07}        if X is not Y
{08}        {
{09}           X tests for intersection with Y
{10}        }
{11}        Y = Next object in linked list
{12}     }
{13}     X = Next object in linked list
{14}  }
{25}end.
```

3.1.7 Aštuntainio medžio algoritmas

Octree algoritmo struktūra yra tokia, kur kiekvienas jos mazgas padalina esamą aplinką į vienodas dalis, sukurdamas 8 naujus ir mažesnius octree mazgus. Pirmo lygio octree struktūra padalina aplinką į 8 mažesnes dalis, tuo tarpu antrajame lygyje, kiekvienas mazgas iš 8 esamų, sukuria dar po 8 mazgus ir t.t..

Tokia *octree* algoritmo struktūra [9] gali saugoti objektų rodykles ne tik medžio lapuose bet ir mazguose. To rezultate, objektų rodykles galima įrašyti bet kurioje medžio vietoje. Sąrašas, kuris vadinamas *linked*, yra naudojamas saugant duomenis medžio mazguose. Kiekvienas mazgas saugo rodyklę į objektą, kuris saugo rodyklę į kitą objektą ir t.t.. Naudojant tokį sąrašą, atminties naudojimas yra mažesnis nei saugojant visas rodykles sudėtingoje struktrūroje. Objektų rodykles taip pat gali būti greitai įterpiamos sąrašo pradžioje esant pastoviam laikui.

Vykdant objekto rodyklės įterpimą, pradžioje ji yra patikrinama pradiniame octree lygyje. Jei objekto aprėpties dėžę visiškai telpa aukštesniame *octree* lygyje, objektas bus perkeliamas į tą lygį. Šis procesas vykdomas tol, kol objekto aprėpties dežė nebetelpa aukštesniame *octree* lygyje. 3.8 pav. iliustruoja aplinkos skaidymą, kuris buvo aprašytas ankščiau, pasitelkiant *octree* algoritmą.



3.8 pav. Virtualios aplinkos skaidymas *octree* algoritmo pagalba

Objektų ištrynimas iš *octree* struktūros yra panašus kaip ir įrašymas, pereinant per medžio mazgus ir naudojant objektų aprėpties dėžių koordinates bei dydi. Norint ištrinti objektą, reikia pereiti per objektų rodyklių sąraša ir surasti reikiamą objektą.

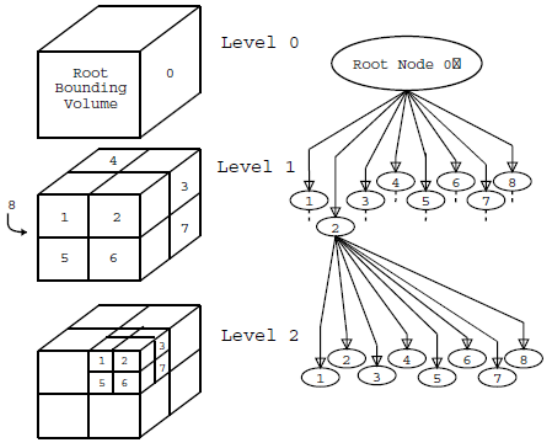
Naudojant *octree* algoritmą susidūrimų paieškai atlikti, pradžioje tam tikro objekto aprėpties dėžė bus patikrinama su visais objektais, kurie randasi aukštesniame *octree* medžio lygyje. Jei objektas nėra surastas konkrečiame mazge, taip pat kaip įterpiant arba ištrinant objektą, patikrina esamo objekto aprėpties dėžę, ar ji gali tilpti žemesniame lygyje. Tuomet perkeltas objektas yra tikrinamas su visais objektais esančiais tame pačiame mazge. *Octree* algoritmo pseudo kodas aprašytas žemiau (žr. *Octree_CheckForCollisions*).

```
Procedure Octree_CheckForCollisions
// objektas X = objektas, kuris nebus patikrintas octree struktūros
//objektas Y = pirmas objektas linked sąraše

{01}begin
{02}  while Y is not NULL
{03}  {
{04}     if X is not Y
{05}     {
{06}        X tests for intersection with Y
{07}     }
{08}     else
{09}     {
{10}        X was found on this level
{11}     }
{12}     Y = Next object in linked list
{13}  }
{14}  if X was fount on this level
{15}  {
{16}     for all children nodes
{17}     {
{18}        X does collision test against child node
{19}     }
{20}  }
{21}  else
{22}  {
{23}     M = what children node X bounding sphere fits in
{24}     X does collision test against M
{25}end.
```

Tinklelio algoritmas gali būti patobulinamas pritaikant aštuntainio medžio (*angl. octreee*) algoritmą (žr. 3.9 pav. [16]). Skirtingai nuo tinklo, *octree* algoritmas turi savybę prisitaikyti prie objektų išsidėstymo 3d erdvėje. Tačiau, dėl šios savybės octree algoritmas yra reiklus

kompiuteriniams resursams, nes šio algoritmo struktūra turi būti nuolatos atnaujinama kiekviename kadre. Kita šio algoritmo problema yra ta, jog jis reikalauja didelio kiekio atminties resursų dėl tinklo gardelių skaičiaus ir dėl to, kad objektai gali priklausyti daugiau nei vienai gardelei.



3.9 pav. *Octree* algoritmo duomenų struktūra

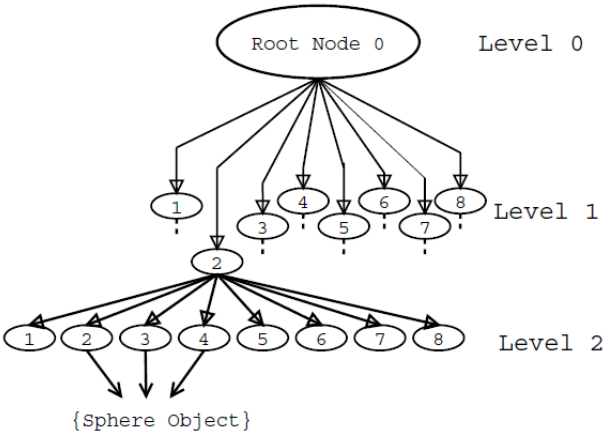
Octree duomenų struktūra priklauso tokiai struktūros klasei, kurią sudaro atskirų tūrio vienetų hierarchija. Šis algoritmas naudoja skaldyti ir valdyti metodą (angl. *devide and conquer*) keliaujant ir vykdant paiešką duomenų struktūroje ir taip pat turi įvairias šios struktūros panaudojimo sritis, tokias kaip susidūrimų paieška bei interaktyvi 3D grafika.

Tam, kad sukurti octree struktūrą, pradžiai, reikia panaudoti *AABB*, kuris aprėpia visą susidūrimų paieškos erdvę ir kuris tampa medžio pradiniu tašku. Po to, visi objektai esantys šiame taške, yra įtraukiami į mazgus, kurie yra padalijamas į 2x2x2, sąrašą. Kiekviena tinklo gardelė tampa naujas octree medžio mazgas ir šis procesas yra kartojamas tol, kol pasiekiamas atstumas nuo pradinio *octree* medžio taško. Sukūrus *octree* struktūrą, toliau galima apdoroti duomenis, einant nuo *octree* medžio pradinio taško žemyn per esančius lapus. Kai koks nors medžio mazgas yra pasiekiamas, iškviečiamas susidūrimų paieškos vienas iš algoritmų, kuris apskaičiuoja tame mazge esančius objektus ar jie nesusiduria. Pabaigus susidūrimų paieška konkrečiame mazge, pereinama prie to mazgo esamų vaikų.

Kalbant apie *octree* algoritmą, yra keletas terminų, kurie apibūdina šio algoritmo komponentus. Terminas *octant* naudojamas norint apibūdinti vieną iš aštuonių mažų kubų. *Octree* duomenų struktūroje, kaip ir kitose panašaus tipo struktūrose, mazgai, kurie nusako *octree* medžio šakas, vadinami lapais. *Octants*, kurie priklauso *octree* medžio lapams, vadinami *voxels*. Pastarasis apibūdina mažiausią erdvės vienetą, kuris gali būti nagrinėjamas šiame tūryje.

Octree duomenų struktūra paprastai įgyvendinama kaip rodyklinė medžio duomenų struktūra, nes rodyklėmis pagrįstas medis leidžia aukšto lygio lankstumą ir paprastumą statant bei pasiekiant reikiamus duomenis. Viena iš rodyklinio medžio funkcijų yra ta, kad būtinybei esant, *octants* gali būti pridedamos arba ištrinamos iš *octree* duomenų struktūros. Vidinėje *octants* duomenų struktūroje, tėviniai *octants* saugo rodykles į vaikinius *octants*, kurie yra naudojami vaikštant *octree* duomenų struktūroje. Papildomai, be rodyklių, kurios reikalingos naviguojant, *octants* saugo duomenų įrašą arba rodyklę i duomenis, kurie yra susiję su *actant*.

Octree duomenų struktūra puikiai tinka trijų matmenų duomenims saugoti, dėl būdo, kuriuo *octree* padalija ir suindeksuoja erdvę. Norint atvaizduoti duomenis *octree* struktūroje, duomenys yra susiejami su *octant*. *Octant* pozicija ir dydis nurodo duomenų vietą bei tūrį. Kai duomenys yra saugomi tokiame octant, kuris nėra *octree* duomenų struktūros lapas, tai reiškia, kad duomenys užima visus žemiau esančius *octant.* Toks sąryšis tarp octants ir duomenų iliustruojamas žemiau (žr. 3.10 pav. [16]).

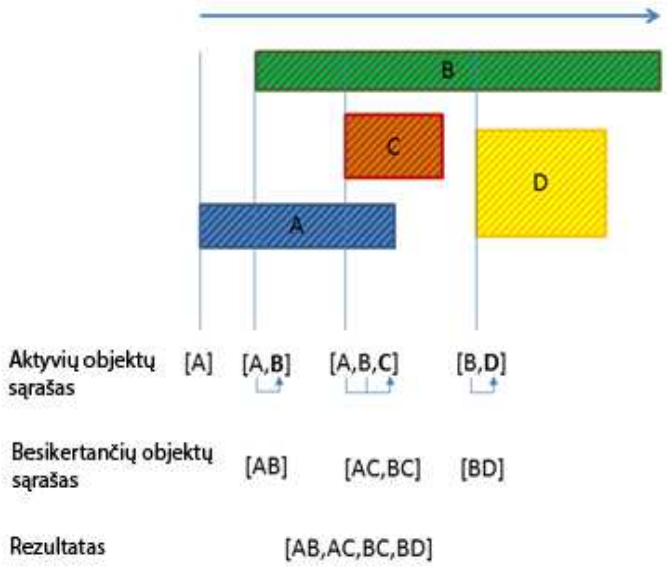


3.10 pav. *Octree* duomenų struktūra, kuri saugo objekto vietą mazguose

Norint įrašyti duomenis į octree, reikia pakartotinai pereiti per visą octree duomenų struktūrą ir ieškoti tų octants, į kuriuos reikia įrašyti naujus duomenis. Pradedant nuo pradinio mazgo (angl. *root*), duomenys yra tikrinami ar jie nesusiduria su to mazgo aštuoniais vaikais. Ėjimas per mazgus ir jų vaikus tęsiamas tol, kol nauji duomenys kertasi su konkrečiais *actants*. Priklausomai nuo vartotojo poreikių, *octree* duomenų struktūros apėjimas gali būti nutraukiamas tinkamu laiko momentu. Paprastai, dauguma *octree* algoritmų užtrunka $O(\log n)$ laiko, kur n yra *octree* medžio mazgų gylis.

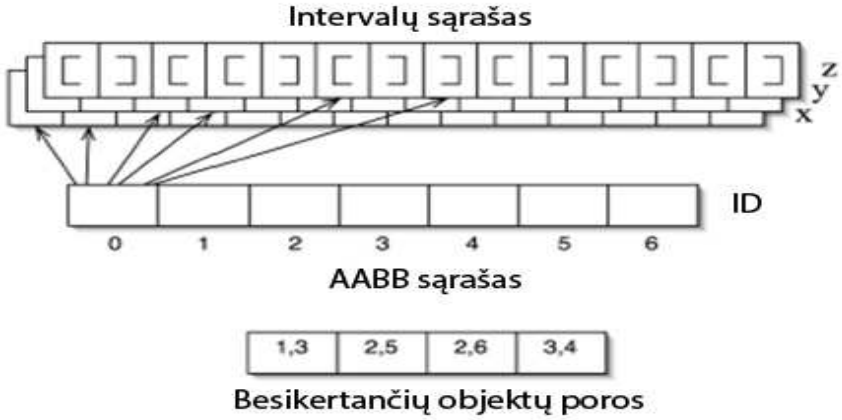
3.1.8 Sort and Sweep algoritmas

Šis algoritmas yra dažniausiai naudojamas plačiajame susidūrimų paieškos etape. Algoritmo tikslas yra surasti tuos objektus kurie persidengia, projektuojant kiekvieno objekto pradžios bei pabaigos taškus ant kiekvienos Dekarto ašies [10]. Algoritmo struktūra pavaizduota žemiau (žr. 3.11 pav.).



3.11 pav. *sort and sweep* algoritmo struktūra

Yra trys pagrindinės duomenų struktūros įgyvendinant *sort and sweep* algoritmą. Surūšiuotas kiekvienos ašies intervalų sąrašas, kuriame saugomi objektų pradžios bei pabaigos taškai. Papildomai, kiekviename objekte, turinčiame pradžios ir pabaigos taškus, egzistuoja mazgas, kuris saugo nuorodas į *AABB* duomenų sąrašą. Kai konkretus objektas turi atnaujinti savo *AABB* dėl savo judėjimo, šis objektas atnaujina kintamuosius naudojant šias mazgu nuorodas. To rezultate, gaunamas objektų rinkinys tų objektų, kurie turi didelę tikimybę susidurti visose trijuose ašyse. *Sort and sweep* algoritmo duomenų struktūra pavaizduota žemiau (žr. 3.12 pav. [10]).



3.12 pav. *sort and sweep* algoritmo duomenų struktūra

Šio algoritmo kintamųjų reikšmių lentelė pavaizduota žemiau (žr. 3.1 lent.).

3.1 lentelė Terminų aprašymai

Symbolis	Reikšmė
n	AABB skaičius
k	Naudojamų ašių skaičius
m	Įrašymų bei ištrynimų skaičius
u	Atnaujinimai (judantys objektai)
s	Apkeitimo veiksmai
o	Persidengiančių AABB skaičius
e	Persidengiančių AABB skaičiaus pasikeitimai

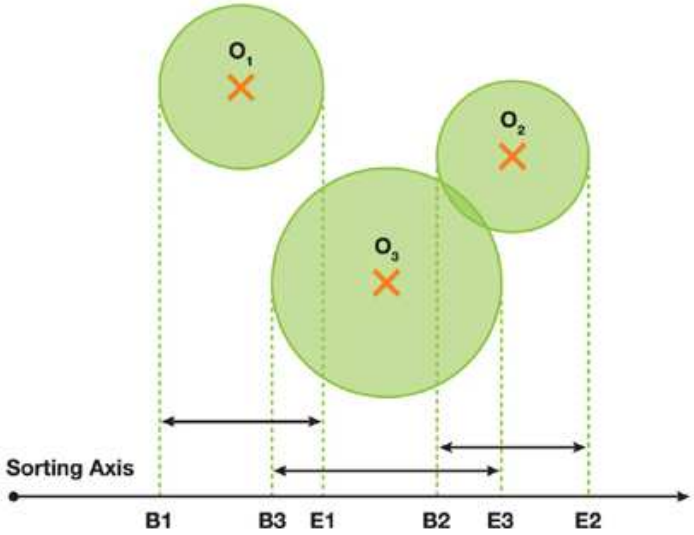
Galutinis *sort and sweep* algoritmo sudėtingumas yra pavaizduotas (1) formulėje, o pseudo kodas aprašytas žemiau (*žr. Sort&Sweep_CheckForCollisions*).

$$O(uk + sk + e + mnk + o), \qquad (1)$$

```
Procedure Sort&Sweep_CheckForCollisions
{01}begin
{02}  algorithm coordinate-sort-axe(L:List(endpoints))
{03}  scan = head(L)
{04}  while L unsorted do
{05}  {
{06}     mark = right(scan), left = left(scan)
{07}     while scan < left do
{08}     {
{09}        if scan = b_i and left = e_j then
{10}        {
{11}           increment counter(i,j)
{12}           if counter(i,j) = 3 then
{13}           {
{14}              report overlap(i,j)
{15}           }
{16}        else if scan = e_j and keft = b_i then
{17}        {
{18}           decrement counter(i,j)
{19}           if counter(i, j) = 2 then
{20}           {
{21}              remove overlap(i,j)
{22}           }
{23}        end if
{24}        scan = left, left = left(scan)
{25}     end while
{26}     scan = mark
{27}  end while
{06}end.
```

Šiame algoritme, kiekvieno objekto *O* aprėpties dėžė projektuojama ant *x*, *y* arba *z* ašių pasirinktinai, apibrėžiant objekto sankirtos intervalą $[S_i, E_i]$, kur B_i žymi sankirtos intervalo pradžią,

o E_i - sankirtos pabaigą. Dviejų objektų, kurių sankirtų intervalai nepersidengia, negali susikirsti vienas su kitu. Kiekvieno objekto du žymekliai yra patalpinami į sąrašą su 2n įrašų. Toliau šis sąrašas yra surūšiuojamas didėjimo tvarka ir pereinamas nuo pradžios iki pabaigos. Kiekvienas surastas B_i markeris šiame sąraše, įdeda objektą *O* į aktyvų objektų sąrašą. Kiekvienas surastas E_i markeris - ištrina objektą *O* iš aktyvių objektų sąrašo. Tokiu būdu, sankirtų testų skaičius sumažėja, kadangi kiekvieną aktyvų objektą reikia tikrinti tik su kitu objektu esančiu tame pačiame sąraše. Trijų objektų pavyzdys yra pavaizduotas iliustracijoje (žr. 3.13 pav. [7]).



3.13 pav. Trijų objektų rūšiavimo ir nurašymo algoritmas

Šio algoritmo įgyvendinimas nėra sudėtingas, todėl yra geras pirmojo sankirtos etapo pradinis atskaitos taškas. Be to, dėl erdvinio koherentiškumo šis algoritmas yra greitesnis ir turi $O(n)$ sudėtingumą.

AABB gali būti išreiškiamas panaudojant jo tris intervalus (po vieną kiekvienai ašiai). Dviejų *AABB* sankirta yra užfiksuojama tada ir tik tada, kai jų visi trys intervalai kertasi vienas su kitu. Šis, *Sort and Sweep* algoritmas laiko visus intervalus atskiruose surūšiuotuose sąrašuose (kiekvienas sąrašas kiekvienai koordinatei) bei išnaudoja kadrų nuoseklumą. Tai reiškia, kad iš ankstesnio kadro sąrašai yra beveik surūšiuoti esamame kadre, nes objektai nejuda per daug toli tarp kadrų. Įterpimo ir rūšiavimo algoritmai naudojami tam, kad palaikyti sąrašus surūšiuotus. Analizuojant sąrašo gretimus intervalus galima gauti visas objektų poras, kurios kertasi tarpusavyje, plačiajame susidūrimų paieškos etape.

Sort and Sweep algoritmas gali būti išlygiagretinamas ir susinchronizuotas jo vykdymas, panaudojant tris šiuos žingsnius:

1. paleidžiama viena procesoriaus giją (angl. *thread*) kiekvienam objektui, kuri apskaičiuoja to objekto aprėpties dėžę, perkelia objektus į pasirinktą koordinačių ašį ir surašo visų objektų pradžios bei pabaigos taškus į išvesties masyvą.
2. masyvas yra surūšiuojamas didėjimo tvarka.
3. paleidžiama viena procesoriaus gija kiekvienam masyvo elementui. Jei masyvo elementas yra pabaigos taškas, tuomet procesoriaus gija yra sustabdoma. Priešingu atveju, jei tai pradžios taškas, tuomet procesoriaus gija eina masyvo elementais į priekį, vykdydama sankirtos testus tol, kol pasiekia tam tikrą pabaigos tašką.

Akivaizdžiausias šio algoritmo trūkumas yra tai, jog blogiausiu atveju gali tekti atlikti iki $O(n^2)$ susidūrimų paieškos testų trečiajame žingsnyje, nepriklausomai nuo to, kiek aprėpties dėžių kertasi 3D erdvėje. Taip pat tai nepriklauso ir nuo pasirinktos koordinačių ašies, nes esant blogiausiam atvejui, reikia patikrinti tam tikrą objektą su kitu esamu objektu, nors pastarasis gali būti labai nutolęs nuo pirmojo. Nors ši savybė pasireiškia abiejuose šio, *sort and sweep* algoritmo

įgyvendinimo etapuose, yra kitas veiksnys, į kurį reikia atsižvelgti, išlygiagretinant ši algoritmą - nuokrypis (angl. *divergence*).

3.1.9 Nukrypimas

Nukrypimas - tai dviejų gretimų proceso gijų darbo įvertinimas, ar jos atlieka ta patį ar skirtingus veiksmus. Yra du skirtingi nukrypimas tipai:

- *vykdymo nukrypimas* - reiškia, jog proceso gijos vykdo skirtingą kodą arba atlieka skirtingus kontrolės srauto sprendimus.
- *duomenų nukrypimas* - reiškia, jog proceso gijos skaito arba rašo skirtingas vietas atmintyje.

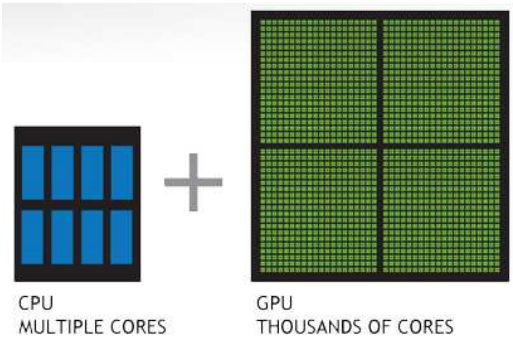
Našumo požiūriu, abu šie tipai yra blogi, atliekant išlygiagretinimą ir gali įtakoti programos veikimą.

Tradiciniuose *CPU* procesoriuose, didelės duomenų saugyklos talpinamos šalia vykdymo etapo. Daugeliu atvejų, tai padaro atmintį pasiekiamą. Duomenys, kuriuos norima pasiekti yra visada laikinojoje atmintyje (angl. *cache*). Tačiau ši struktūra tampa netinkama, kai yra norima išlygiagretinti procesus. Todėl, tokiu atveju, reikalinga pateikti didesnį kiekį procesoriaus gijų, panašaus dydžio talpykloms. Tai tampa ne kliūtis, jei šios gijos atlieka daugiau ar mažiau panašią užduotį tuo pačiu metu, kadangi jų bendro darbo rinkinys vis dar linkęs likti gana mažas. Bet jei jie daro kažką visiškai skirtinga, tuomet esamas rinkinys subyra.

Trečiasis *sort and sweep* algoritmo žingsnis dažniausiai kenčia nuo vykdymo nukrypimo. Šio algoritmo įgyvendinime, kuris buvo aprašytas ankščiau, proceso gijos, kurios pasiekia objekto pabaigos tašką, baigia darbą, o tos gijos kurios liko, tęs darbą. Tos proceso gijos, kurios bus atsakingos už didelius objektus, atliks daugiau darbo, negu tos, kurios yra atsakingos už mažus objektus. Jeigu virtualioje aplinkoje yra didelis kiekis skirtingo dydžio objektų, gretimų proceso gijų vykdymo laiko skirtumas bus didelis. Kitaip tariant, vykdymo nukrypimas bus aukštas, todėl kentės programos veikimo našumas.

3.2 Lygiagretaus programavimo metodų bei įrankio analizė

GPU skaičiavimų dėka, programos veikimas paspartėja, nes jų dalys, kurios reikalauja didelių skaičiavimo resursų, perduodamos minėtai *GPU*, tuo tarpu likusios dalys dirba ant *CPU*. Dėl to, iš vartotojo puses, programos veikimas ženkliai pagerėja. *CPU* ir *GPU* yra galingas derinys, nes *CPU* procesoriai susideda iš kelių optimizuotų branduolių, tuo tarpu *GPU* procesorius susideda iš tūkstančių mažesnių bei efektyvesnių branduolių, kurie yra sukurti lygiagrečiam darbui (žr. 3.14 pav.).



3.14 pav. CPU ir GPU procesorių branduolių struktūra

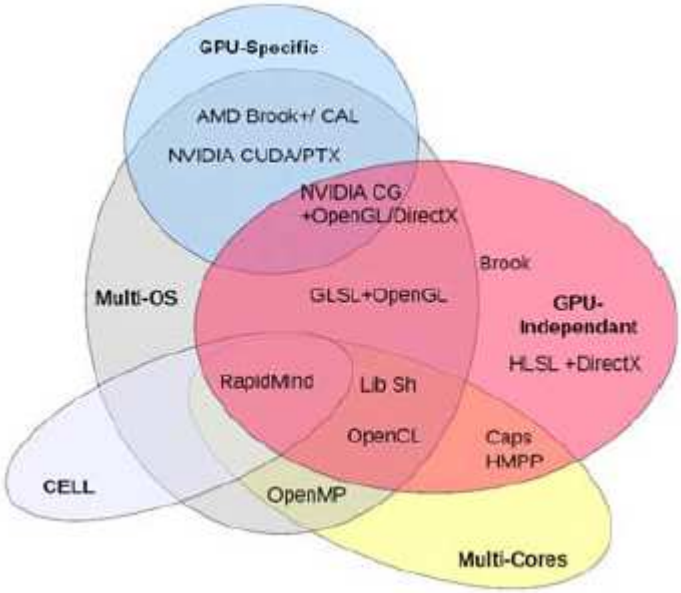
3.2.1 GPU pagrindu veikiantys algoritmai

Vaizdo pagrindu veikiantys algoritmai buvo modifikuojami tam, kad išnaudoti augančią grafinės įrangos skaičiavimo galią [15]. *GPU* yra efektyvus rasterizuojant daugiakampius, taip pat *GPU* pagrindu veikiantys susidūrimų paieškos algoritmai lengvai rasterizuoja esamus objektus ir atlieka 2D arba 2.5D persidengimo testus. Paduodant keletą primityvių objektų, šis algoritmas

grąžina objektų poras, kuriose objektai gali kirstis vienas su kitu. Šias grąžintas poras toliau gali skaičiuoti *CPU*. Geras privalumas naudojant *GPU* yra tas, kad nereikia perskaičiuoti tūrinių duomenų struktūrų ir yra efektyvus naudojant su standžiais (angl. *rigid*) bei deformuojamais objektais. Taip pat GPU tampa naudingas įrankis skaičiuojant lauko atstumus, naudojant vienodą erdvinį tinklelį. Be to, galima atlikti matomumo skaičiavimus, naudojant uždarumo užklausas ir skaičiuoti tiek objekto viduje, tiek kelių objektų tarpusavio susidūrimus.

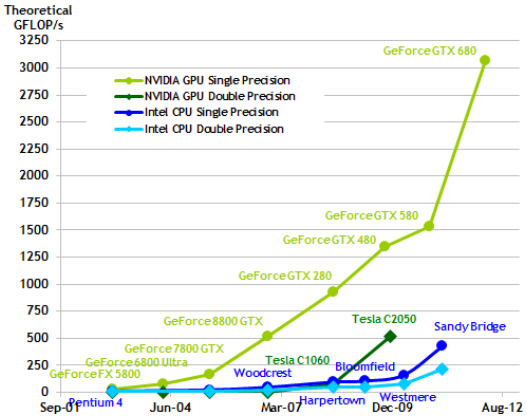
3.2.2 NVidia CUDA

GPU panaudojimas matematiniams skaičiavimams nėra senas būdas [12]. Devintojo dešimtmečio laikotarpyje, keletas tyrinėtojai naudojo grafinės plokštės rasterizatorius (angl. *rasterizer*) ir *Z* buferius pagerinti Voronoi kelio radimo algoritmus. Tačiau revoliuciją sukėlęs įvykis tapo 2003 metais sukurtos grafinės paprogramės (angl. *shaders*), kurios įgalino matricų skaičiavimu grafinėse plokštėse. Taip pat, šiais metais OpenGL ir Direct3D atliko daug tyrimų šioje srityje. *Brook* buvo pirmoji *C* programavimo kalbos plėtinys, kuris leido panaudoti grafikos plokštę kaip procesorių lygiagretiems skaičiavimams. 2007 metais Nvidia sukūrė programavimo kalbą ir taip pat programinę įrangą vadinamą *CUDA*, kuri išnaudoja *GPU* teikiamas galimybes pagerinti kompiuterinius skaičiavimus, naudojant procesoriaus gijas bei lygiagretaus programavimo principus. Visos aukšto lygio GPGPU programavimo kalbos [11] pavaizduotos iliustracijoje (žr. 3.15 pav.).

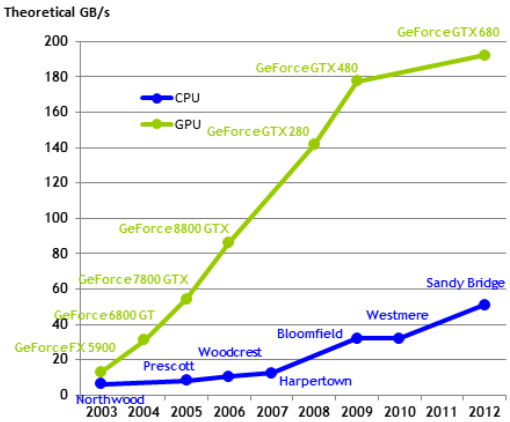


3.15 pav. Aukšto lygio GPGPU programavimo kalbos

CUDA (angl. *Compute Unified Device Architecture*) - tai *NVIDIA* sukurta nauja duomenų apdorojimo architektūra išnaudojanti grafinio procesoriaus resursus. Grafikos procesorius tapo kaip duomenų apdorojimo įrenginys. Per pastaruosius dešimt metų grafikos procesoriai (*GPU*) smarkiai patobulėjo ir tapo galingais skaičiavimo įrenginiais. Su dideliu branduolių kiekiu ir labai dideliu atminties pralaidumu šiandien grafikos procesoriai turi neįtikėtiną galią tiek grafiniuose tiek negrafiniuose skaičiavimuose. Skatinami nepasotinamos rinkos kurti programinę įrangą, kuri atlieka skaičiavimus realiu laiku, turi didelės raiškos 3D grafiką ir programuojamus grafikos procesorius (*GPU*), išsivystė į procesorius, kurie turi daugybę branduolių su lygiagretinimo funkcija, milžinišką skaičiavimo galią ir labai didelę atminties pralaidumą. *CPU* ir *GPU* augimas ir jų parametrinės galimybės pavaizduotos iliustracijose žemiau (žr. 3.16pav. [1] ir 3.17 pav. [1]).

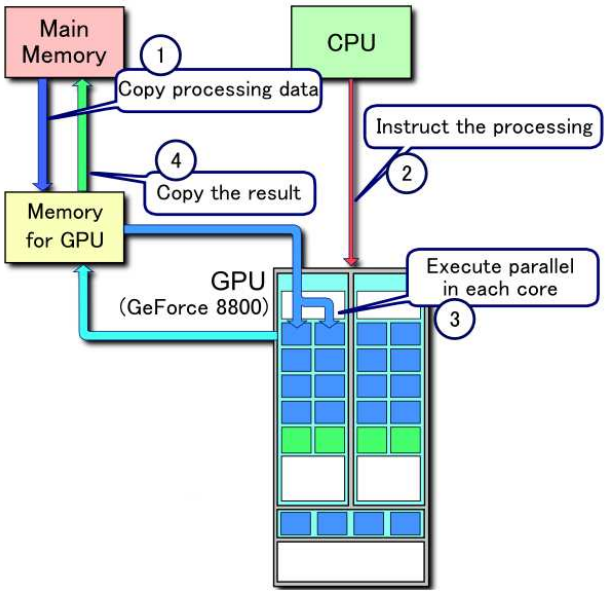


3.16pav. Slankiojo kablelio operacijų per sekundę CPU ir GPU



3.17 pav. CPU ir GPU atminties augimo diagrama

CUDA suteikia programų kūrėjams prieigą prie virtualių programavimo komandų rinkinių ir lygiagretaus skaičiavimo elementų atminties, esančios *GPU*. Tokiu būdu, įvairūs skaičiavimai, kuriems reikalingas didelis kiekis nuoseklių veiksmu ir procesoriaus darbo, visi šie skaičiavimai yra padalinami lygiagrečiai (priklausant nuo turimos grafinės plokštės savybių) ir atliekami vienu laiko momentu. Išlygiagretintų procesų veikimo principas pavaizduotas žemiau esančioje schemoje (žr. 3.18 pav.). Pasitelkiant *CUDA*, įvairūs sudėtingi skaičiavimai gali būti atliekami realiu laiku (angl. *real time*).

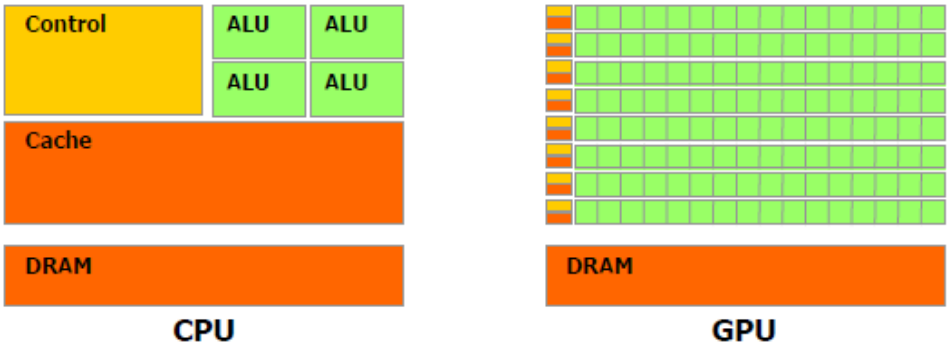


3.18 pav. Duomenų apdorojimas naudojant *CUDA*

Duomenų apdorojimo proceso seka naudojant *CUDA*:

1. Duomenys nukopijuojami iš pagrindinės atminties į *GPU* atmintį.
2. *CPU* nurodo kuriuos procesus reikia atlikti *GPU*.
3. *GPU* įvykdo paralelinius lygiagrečius skaičiavimus kiekviename branduolyje atskirai.
4. Nukopijuojami rezultatai iš *GPU* atminties į pagrindinę atmintį.

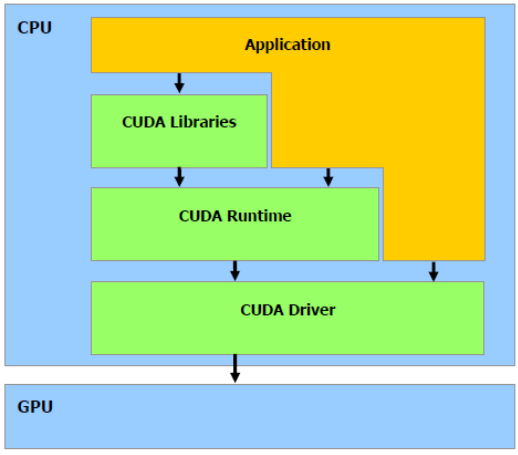
Pagrindinė *GPU* evoliucijos priežastis yra ta, jog ji yra specializuota grafikos apdorojimui, kas reikalauja labai intensyvių, lygiagrečių skaičiavimų, todėl yra suprojektuota taip, kad daugiau tranzistorių yra skirta duomenų apdorojimui negu spartinančiai atminčiai ir srautų valdymui (žr. 3.19 pav. [1]).



3.19 pav. *CPU* ir *GPU* architektūros struktūra

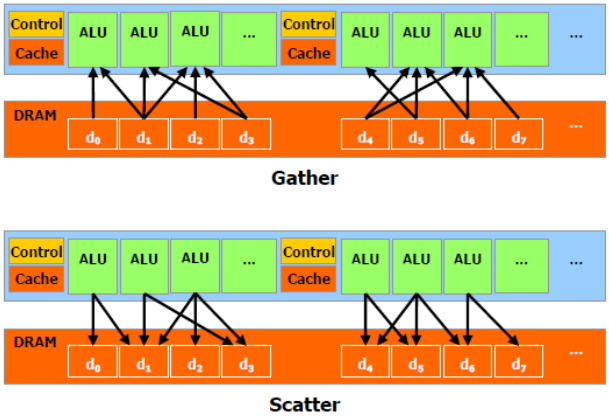
Tiksliau kalbant, *GPU* ypač tinka lygiagretiems skaičiavimams – ta pati programa yra suskaidoma į daug duomenų elementų ir vykdoma lygiagrečiai, tokiu būdu pasiekiamas didelis skaičiavimų našumas, paprastesnis srautų valdymas. Daugelyje programų, apdoroti didelius duomenų rinkinius, pavyzdžiui, masyvus, galima naudoti lygiagretų duomenų apdorojimo programavimo modelį, siekiant paspartinti skaičiavimus. Trimatės grafikos vaizdavime dideli duomenų rinkiniai yra susiejami su lygiagrečiomis gijomis. Panašiai vaizdo kodavimo ir dekodavimo, vaizdo mastelio keitimo, stereo vaizdo ir kitose programose galima naudoti lygiagretų duomenų apdorojimą. Iš tiesų, daugelis algoritmų, nebūtinai susijusių su grafikos apdorojimu, gali būti paspartinti naudojant lygiagretų duomenų apdorojimą. Iki šiol priėjimas prie grafikos procesoriaus skaičiavimo resursų, ne grafikos apdorojimo tikslams, buvo keblus. *GPU* buvo galima programuoti tik per grafikos *API* (Aplikacijų programavimo sąsaja), kas lėmė sudėtingą programavimo įsisavinimą pradedantiesiems ir netinkamus *API* įrankius ne grafikos programų kūrimui. *GPU* programos galėjo duomenis paimti iš bet kurios *DRAM* atminties vietos, bet negalėjo įrašyti į bet kurią vietą. Tai labai sumažino programavimo lankstumą. Kai kurios programos buvo stabdomos *DRAM* atminties mažo pralaidumo, dėl to nepakankamai išnaudodavo *GPU* skaičiavimo galią.

CUDA – bendrosios paskirties lygiagrečiųjų skaičiavimų architektūra, kuri panaudoja Nvidia *GPU* skaičiavimams su slankaus kablelio skaičiais. Palaikoma įranga GeForce 8 serija, Quadro FX 5600/4600, ir Tesla sprendimai. Operacinės sistemos daugiafunkcinio apdorojimo (angl. *multitasking*) sistema yra atsakinga už prieigą prie *GPU* kelių *CUDA* ir grafikos programų vienu metu. *CUDA* programinė įranga yra sudaryta iš kelių sluoksnių, kaip pavaizduota struktūrinėje diagramoje (žr. 3.20 pav. [1]) : aparatinės įrangos tvarkyklės, aplikacijų programavimo sąsajos (*API*) ir dviejų, bendros paskirties, aukšto lygio matematinių bibliotekų „*CUFFT*“ ir „*CUBLAS*“.



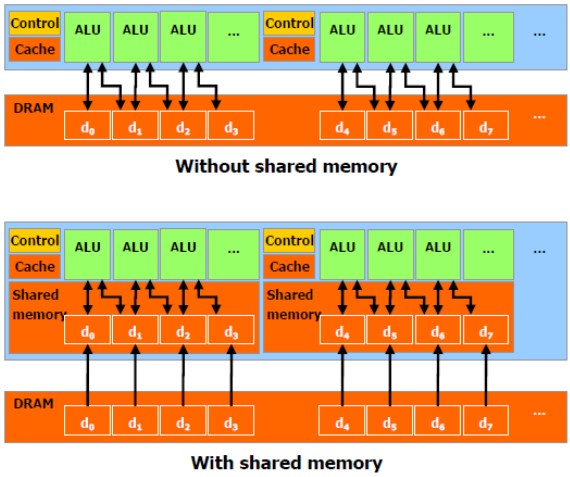
3.20 pav. CPU struktūros sluoksniai

Cuda pateikia bendrą *DRAM* atminties adresavimą, kaip pavaizduota iliustracijoje (žr. 3.21 pav. [1]), didesniam programavimo lankstumui pasiekti. Žiūrint iš programavimo pusės tai reiškia galimybę skaityti ir rašyti į bet kurią atminties vietą taip kaip dirbant su *CPU*.



3.21 pav. *DRAM* atminties adresavimo schema

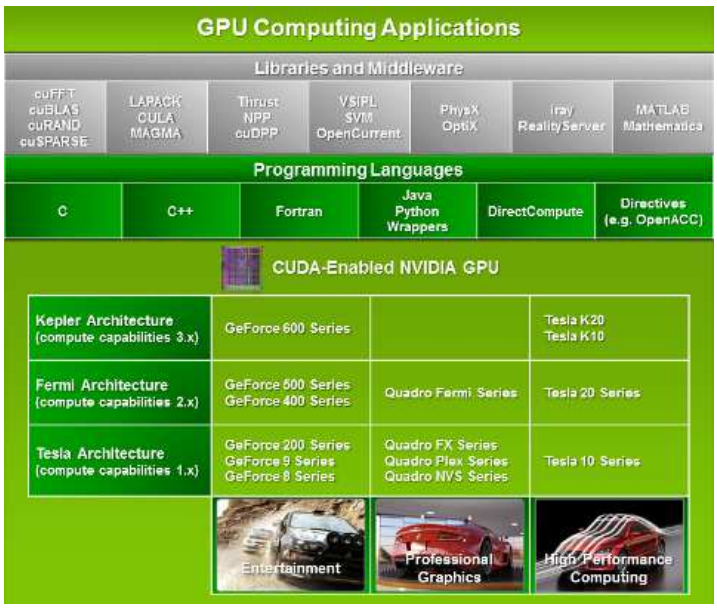
CUDA vienas iš pranašumų tai lygiagrečių gijų spartinančioji atmintis arba „On-Chip„ bendra (*angl. shared*) atmintis su labai didelių skaitymo bei rašymo greičiu, kurią gijos naudoja duomenims tarpusavyje keistis. Kaip pavaizduota 3.22 pav. [1] vykdomos programos lygiagrečiai, įgauna pranašumą, sumažindamos duomenų apsikeitimo su *DRAM* laiką, taip sumažindamos priklausomumą nuo *DRAM* pralaidumo.



3.22 pav. DRAM pralaidumo schema

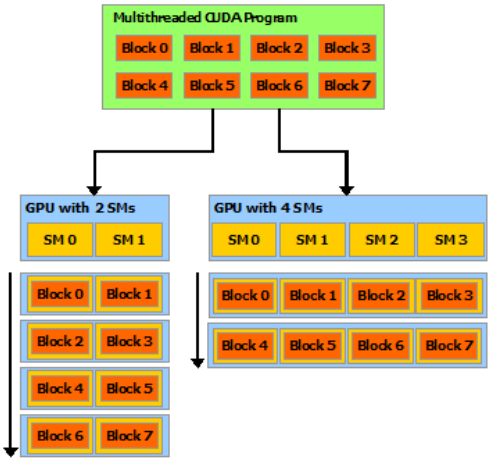
Programų kūrėjai prie *CUDA* platformos gali prieiti per jos palaikomas bibliotekas, per kompiliatorių direktyvas (angl. *compiler directives*) ir per standartinių programavimo kalbų (C, C++ and Fortran) plėtinius. Programuotojai kurie naudoja C/C++ gali naudoti *'CUDA* C/C++', kurios yra sukompiliuotos su NVCC arba naudojant NVidia LLVM (angl. *Low Level Virtual Machine*). Fortran programuotojai gali naudoti *'CUDA* Fortran', kuris yra sukompiliuotas su PGI CUDA Fortran rengėju (angl. *compiler*).

Be to, *CUDA* platforma palaiko ir kitas skaičiavimo sąsajas, tokias kaip: *OpenCL*, *Direct Compute* ir C++AMP (angl. *Accelerated Massive Parallelism*) (žr. 3.23 pav. [1]).



3.23 pav. *CUDA* įrankių diagrama

Šiai laikais *CPU* ir *GPU* vienas didžiausių privalumų yra tai, kad pagrindinių procesorių lustai dabar yra lygiagrečios sistemos. Be to, jų lygiagretumo galimybės ir apimtys vis dar auga pagal Moore'o dėsnį. Šis privalumas leidžia programų kūrėjams apjungti abu procesorius *CPU* ir *GPU* ir išnaudoti juos išgaunant didesnį našumą. Tokiu būdu, didelės apimties uždaviniai yra dalinami į mažesnes dalis ir suskaidomi į blokus, kuriuose šios dalys yra realizuojamos atskirai nuo kitų dalių arba skaidomos į dar mažesnes dalis. Dalinimo iliustracija pavaizduota žemiau esančioje diagramoje (žr. 3.24 pav. [1]). Iš iliustracijos galima spręsti, jog *GPU*, kuris turi daugiau procesorių (angl. multiprocessor), atliks darbą greičiau negu *GPU*, su mažesnių kiekiu procesorių.



3.24 pav. Automatinis skaidymas

Mokslinėje litaratūroje *CPU* ir *GPU* dar kitaip yra vadinami:

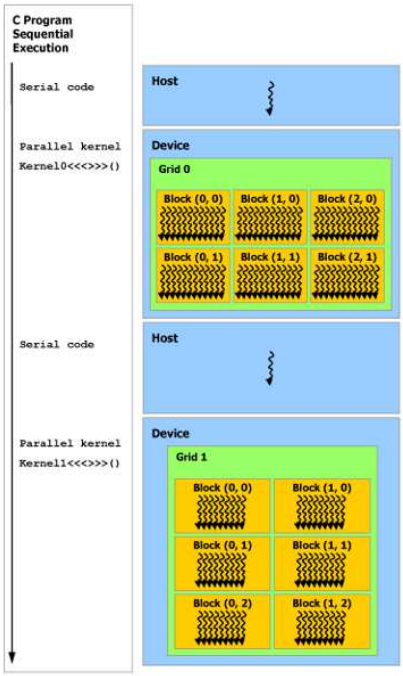
- *CPU - host.*
- *GPU - device*.

Naudojant *CUDA* C++, ši programavimo kalba yra praplečiama, suteikiant programuotojams apsibrėžtas C++ funkcijas, kitaip vadinamus branduolius (angl. *kernels*). Kai šias funkcijas iškviečia, jos yra vykdomos n kartų lygiagrečiai pasitelkiant n *CUDA* gijų (angl. *threads*).

Branduolys yra apibrėžiamas naudojant __global__ deklaracijos funkciją ir *CUDA* gijas, kurios vykdo nurodyto branduolio panaudojimą skaičiavimams. Reikiamo branduolio numeris nurodomas naudojant naują <<<...>>> vykdymo konfigūracijos sintaksę. Kiekvienai gijai yra suteikiamas unikalus gijos ID, kuris yra prieinamas per branduolio kintamąjį (angl. *threadID*).

3.2.3 Įvairiapusis programavimas su *CUDA*

CUDA programavimo modelis *CUDA* gijas vykdo atskirai vienas nuo kito įrenginiuose, kurie veikia kaip *host* koprocesorius paleistoje programoje. Tokiu atveju, dalis programos yra vykdoma *GPU*, o likusi dalis lieka - *CPU* (žr. 3.25 pav. [1]). *CUDA* programavimo modelis sumodeliuotas taip, jog abu, tiek *CPU*, tiek *GPU* naudoja atskirą DRAM atmintį.



3.25 pav. Įvairiarūšio programavimo struktūra

3.2.4 Pagrindinė Nvidia *GPU* ir *CUDA* programavimo koncepsija

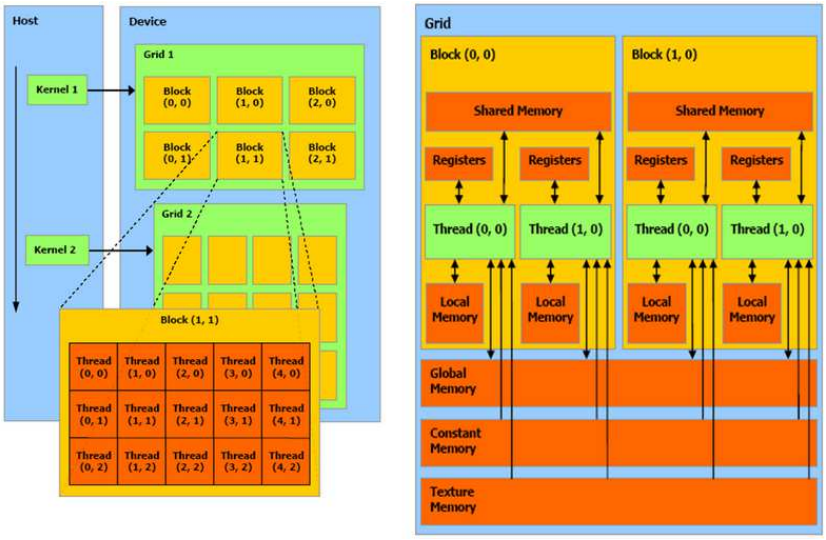
Norint išlygiagretinti veiksmus naudojant *CUDA*, reikia laikytis tam tikrų lygiagretinimui reikalingų žingsnių [2], kurie yra:

1. Inicijuojamas *GPU*
2. *GPU* atminties paskirstymas
3. Kopijuojami duomenys iš *host* į *GPU*
4. Paleidžiami branduoliai ant *GPU*
5. Kopijuojami duomenys iš *GPU* į host
6. Atlaisvinama *GPU* atmintis
7. Išjungiamas *GPU*
8. Gauti duomenys apdorojami *CPU*

Grafinis procesorius gali būti naudojamas kaip skaičiavimo įrenginys vykdant aplikacijos dalis bei jų funkcijas, kurios:

1. turi būti atliekamos daug kartų.
2. gali būti izoliuotos.
3. dirba savarankiškai pagal skirtingus duomenis.

Tokios funkcijos gali būti rengiamos paleisti ant *device*. Tokia programa vadinama branduoliu (angl. *kernel*). Gijų grupė, kuri vykdo branduolį, yra organizuojamas kaip gijų blokų tinklelis. Grafinio procesoriaus branduolių struktūra pavaizduota žemiau (žr. 3.26 pav. [2]).



3.26 pav. Grafinio procesoriaus branduolių struktūra

Kiekvienas *CUDA device* turi kelis atminties tipus, kuriuos panaudojant, galima pasiekti aukštą *CGMA* koeficiento reikšmę. To rezultate gaunamas greitas branduolių vykdymo laikas. Kintamieji, kurie priklauso registrams ir bendrai atminčiai, gali būti pasiekiami labai greitai. Registrai yra priskirti individualios gijoms, t.y. kiekviena gija gali pasiekti tik savo registrus. Branduolio funkcija dažniausiai naudoja registrus saugojant kintamuosius, kurie yra dažno naudojimo ir yra privatūs kiekvienoje gijoje. Bendra atmintis yra priskirta gijų blokams, t.y. visos gijos esančios tame pačiame bloke gali pasiekti kintamuosius, esančius bendroje atmintyje, kuri yra priskirta konkrečiam gijų blokui. Bendra atmintis yra efektyvi priemone gijoms dalinantis savo atliktais darbais.

Likę atminties tipai yra: globali ir pastovi atmintis. Šios atmintys gali būti naudojamos iškviečiant *host API* funkcijas, tokias kaip įrašymas ir skaitymas. Globali atmintis gali būti pasiekiama absoliučiai visų esamų gijų bet kuriuo programos vykdymo laiko momentu. Pastovi atmintis leidžia tik skaitymui prieiga prie duomenų ir yra greitesnė už globalią atmintį.

Aparatūros išteklių taisyklės:

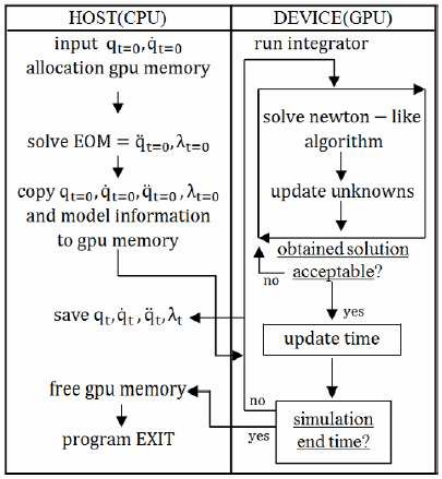
- Viename bloke gali būti ne daugiau kaip 512 gijų.
- Gijos gali dalintis bendra atmintimi su kitomis gijomis esančiomis tame pačiame bloke.
- Gijos gali sinchronizuoti su kitomis gijomis esančiomis tame pačiame bloke.
- Globali, bendra ir tekstūrinė atmintys yra prieinamos visoms gijoms esančioms visuose blokuose.
- Kiekviena gija turi registrus ir privačią atminti.
- Kiekvienas blokas gali naudoti daugiausiai 8.192 registrus, vienodai padalintus visoms gijoms.
- Viename procesoriuje vienu laiko metu galima vykdyti iki 8 blokų ir 768 gijų
- Blokas yra vykdomas tik ant vieno procesoriaus (pvz. negalima perjungti ant kito procesoriaus).
- Blokas gali būti vykdomas ant bet kurio procesoriaus aštuonių procesorių.

3.3 Lygiagretus programavimas su *CUDA*

Tam, kad išlygiagretinti susidūrimų paieškos algoritmus, šiame darbe yra naudojama *CUDA* C programavimo kalba, sukurta *Nvidia*. Kadangi *GPU* turi daugiau atminties nei *CPU*, svarbu yra suprasti ir efektyviai panaudoti *GPU* atminties funkcijas [4]. Norint naudoti *GPU* atmintį, ji turi būti išskirta iš *CPU*. O po to ši panaudota atmintis turi būti išvaloma. Jeigu *GPU* atminčiai reikalingi *CPU* duomenys, juos galima pasiekti naudojant *CUDA API* funkcijas. *CUDA API* siūlo naudingas lygiagrečiam programavimui funkcijas. Lygiagrečiame programavime, duomenų perdavimai tarp *CPU* ir *GPU* turi būti suminimizuoti, nes tokie perdavimai turi mažesnį pralaidumą, lyginant su

duomenų perdavimu tarp globalios atminties ir *GPU*. Pradžioje, *CPU* apskaičiuoja pradinius sistemos duomens (t_0), perkelia juos į *GPU* atmintį ir atlaisvina atmintį kai perkelimas yra baigtas. *GPU* paleidžia integratorių ir perkelia duomenis kiekviename laiko žingsnyje.

Struktūrinė schema pavaizduota iliustracijoje žemiau (žr. 3.27 pav.[17]). *CPU* dalyje, kiekvieno objekto pozicija ir greitis yra inicijuojami, o pagreitį galima gauti sprendžiant judėjimo lygtis (angl. *motion equations*; toliau *EOM*). Šie vektoriai yra perkeliami į *GPU* atmintį. *GPU* dalyje, visos objektų pozicijos ir greičiai yra gaunami kiekviename laiko žingsnyje, naudojant *HHT* integratorių [6]. Jei gautas sprendimas yra priimtinas, tuomet jie yra išsaugojami *CPU* atmintyje ir sprendimo laikas yra atnaujinamas. Jei atnaujinimo laikas buvo paskirtas pabaigos laikui, tuomet *CPU* išvalo atmintį esančią *GPU* ir sustabdo simuliacijos darbą.



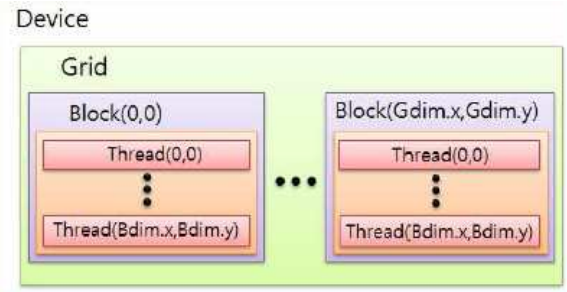
3.27 pav. Lygiagretaus programavimo struktūrinė schema

GPU skaičiavimai atliekami vykdant branduolio funkciją "__global__". Kai yra iššaukiama ši funkcija, tam tikras procesoriaus gijų ir blokų skaičius yra nurodomas <<<Dg, Db, Ns>>> sintaksės, kuri gali būti *int* arba *dim3* tipo, pagalba. Branduolio pagrindinėje funkcijoje, blokų tinklelis ir procesoriaus gijos yra sugeneruojamos (žr. 3.29 pav.). Kiekvienas šio tinklo blokas gali būti atpažintas pagal vienmačiu, dvimačiu arba trimačiu indeksu, pasiekiamu per branduolyje aprašytu kintamuoju *blockIdx*. Kiekvienai procesoriaus gijai, kuri vykdo branduolio pagrindinę funkciją, yra skiriamas unikalus gijos ID, kuris yra prieinamas per branduolio aprašytą *threadIdx* kintamąjį. Gijos bloko dimensijos yra pasiekiamos per branduolio aprašytą *blockDim* kintamąjį.

Naudojant lygiagrečius skaičiavimus, kiekvienoje procesoriaus gijoje, įmanoma vienu metu vykdyti anksčiau minėtą branduolio pagrindinę funkciją. Jeigu branduolio funkcija yra atlikta, tuomet kiekviena procesoriaus gija gauna duomenis skaičiavimams atlikti. Veiksmų paskirstymo diagrama tarp *CPU* ir *GPU* pavaizduota žemiau (žr.3.28 pav.), tuo tarpu duomenų paskirstymas grafiniame procesoriuje yra pavaizduotas diagramoje (žr. 3.29 pav. [17]).

	Dalelių sistema
CPU skaičiavimai	Rezervuoti CPU ir GPU reikalingą atmintį Įvesti pradinius objektų pozicijų ir greičio duomenis Apskaičiuoti EOM, reikalingą dalelių akceleracijai Perkopijuoti CPU atmintį į GPU atmintį
Lygiagretūs skaičiavimai (GPU)	Patikrinti ar visos sistemoje esančios dalelės egzistuoja simuliuojamoje erdvėje
	Susidūrimų paieška naudojant Grid arba Sort and Sweep algoritmą
CPU skaičiavimai	Išvaloma atmintis

3.28 pav. Lygiagretaus programavimo eigos diagrama, atliekant susidūrimų paiešką

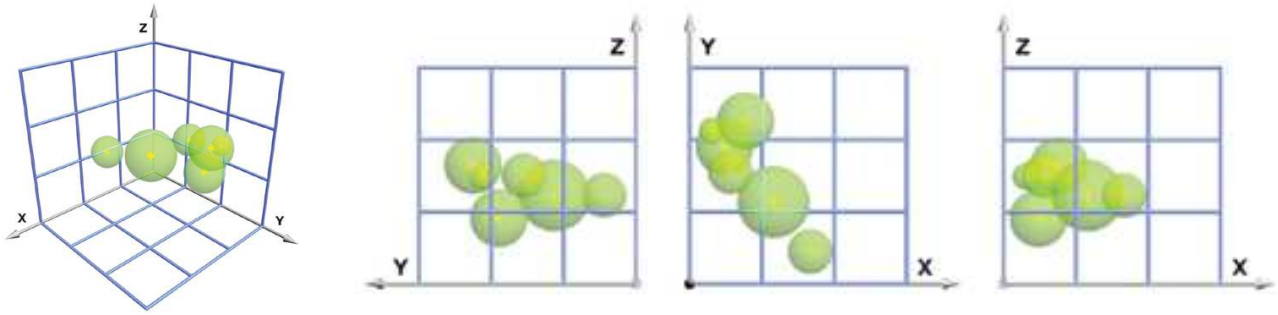


3.29 pav. Lygiagretaus programavimo struktūrinė schema

3.4 Erdvės suskaidymo įgyvendinimas panaudojant *CUDA* technologiją

Šiame skyriuje bus aprašomas erdvinio skaidymo įgyvendinimas pasitelkiant *CUDA* technologiją [7] (žr. 3.30 pav.) bei atlikti šie darbai:

- Objektų ir tinklo gardelių identifikavimo numerių sąrašo išsaugojimas prietaiso (angl. *device*) atmintyje.
- Tinklo gardelių identifikavimo numerių sąrašo sukūrimas, panaudojant objektų aprėpties dėžes. (angl. *bounding boxes*).
- Algoritmo aprašymas reikalingas rūšiuoti sąrašo duomenis.
- Indeksavimo lentelės sukūrimas.



3.30 pav. Objektų rinkinys, esantis 3D erdvėje, kuri yra suskaidyta į tinklelį

3.5 Įgyvendinimas

Kiekvieno objekto atributai vadinami identifikavimo numeriai (angl. *ID*), kurie yra naudojami tam, kad pasiekti papildomus objektų parametrus, tokius kaip:

- Objekto aprėpties apimtis (angl. bounding volume) žemo lygio susidūrimų testų metu
- Tinklo gardelių identifikavimo numerių sąrašo konstrukciją.
- Kontrolės bitus (angl. *control bits*), kurie yra naudojami kaupiant bitų būsenas, tokias kaip namų gardelės tipą ir užimamų gardelių tipus.

Tinkamas būdas saugoti šią informaciją, yra panaudoti masyvo struktūrą. Siekiant sumažinti pralaidumo reikalavimus rūšiuojant šiame masyve, pageidautina laikyti šiuos atributus į du atskirus masyvus, kuriuose yra po $n \; x \; 2d$ elementų:

- Pirmasis masyvas vadinamas tinklo gardelių identifikavimo numerių sąrašo masyvas, kuriame talpinami kiekvieno objekto gardelės identifikavimo numeris.
- Antrasis masyvas vadinamas objektų identifikavimo numerių masyvas, kuriame talpinami kiekvieno objekto identifikavimo numeris ir kontrolės bitai.

Daroma prielaida, jog kiekvieną kartą, kai yra perskirstomi pirmojo masyvo elementai, taip pat perskirstomi ir antrojo masyvo elementai.

3.6 Tinklo gardelių identifikavimo numerių masyvo konstravimas

Po to, kai pakankamas talpinimas buvo paskirtas, sekantis žingsnis, vykdant susidūrimo aptikimo testus, yra užpildyti pirmąjį masyvą. Vertinant tam tikrą objektą, gali būti tokių tinklo gardelių, kurios persidengia su to objekto aprėpties tūrio dėže, tačiau jos nėra objekto namu gardelė (angl. *home cell*) arba kitaip dar vadinama *H* gardele. Tokios tinklo gardelės vadinamos fantomų gardelės (angl. *phantom cells*) arba *P* gardelės. Kaip buvo minėta ankščiau, esant vieno objekto P gardelei ir kito objekto P gardelei, tuomet šie objektai niekada negali susidurti toje pačioje gardelėje, nes abiejų objektų centro taškai atskirti tarp šių gardelių esančios kitos gardelės ir taip pat gardelių diametras yra dvigubai didesnis už didžiausią esamą objektą.

Kiekvieno objekto *H* gardelės identifikacijos numeris yra to objekto centro taško maišos (angl. *hash*) koordinatės. Tipinė 3D maiša apsirašoma taip:

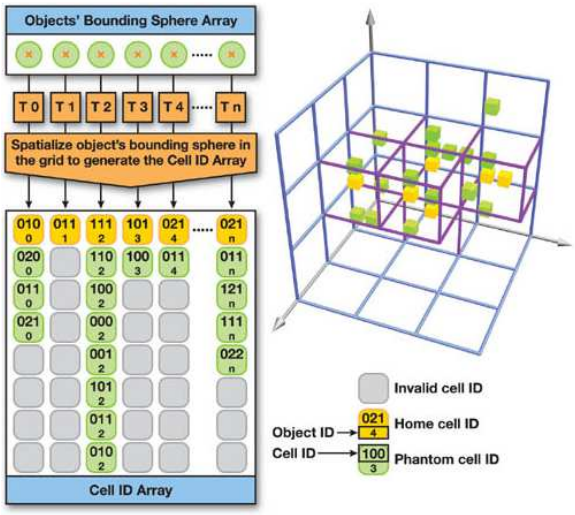
```
GLuint hash = ((GLuint)(pos.x / CELLSIZE) << XSHIFT) |
              ((GLuint)(pos.y / CELLSIZE) << YSHIFT) |
              ((GLuint)(pos.z / CELLSIZE) << ZSHIFT);
```

Šios lygties paaiškinimai:

- *pos* - objekto pozicija erdvėje.
- *CELLSIZE* - tinklo gardelės dydis.
- *XSHIFT*, *YSHIFT* ir *ZSHIFT* - iš anksto nustatytos konstantos, nustatančios kiek bitų priskiriama kiekvienai maišos koordinatei. Šis maišos kodas yra talpinamas gardelių identifikavimo numerių sąraše. Tuo pačiu metu, objektų identifikavimo numerių masyvo atitinantis elementas yra atnaujinamas taip pat. Tuomet galima nustatyti kontrolės bitus, kurie rodo, jog tai yra objekto *H* gardelė.

Kiekvieno objekto *P* gardelės identifikavimo numeriai talpinami į masyvą tam, kad patikrinti ar jos persidengia su *H* gardelėmis. Laikantis taisyklės, jog tinklo gardelės yra dvigubai didesnės už didžiausią esamą objektą, vienu laiko momentu visi objektai gali kirstis tik su 2^d tinklo gardelėmis. Tai reiškia, jog tam tikras objektas gali turėti vieną *H* gardelę ir $2^d - 1$ *P* gardelių. Jeigu objektas turi mažesnį P gardelių skaičių, tuomet likusios gardelės yra pažymimos `"0xffffffff"` ir reiškia, jog jos yra neprieinamos.

Šių *H* ir *P* tinklo gardelių identifikavimo numerių sukūrimas yra atliekamas kiekvieno objekto vienos gijos (angl. *thread*). Kuomet objektų yra daugiau nei gijų, tuomet kiekviena gija apdorojo keletą objektų. Tiksliau kalbant, gija *j* esanti gijų bloke *i* apdoroja objektus $iB + j, iB + j + nT, iB + j + 2nT$ ir t.t., kur *B* yra viename bloke esančių gijų skaičius, o *T* yra bendras gijų skaičius. Tinklo gardelių identifikavimo numerių masyvo konstravimas pavaizduotas iliustracijoje (žr. 3.31 pav.[7]).



3.31 pav. Objektų pavaizduotų iliustracijoje konstrukcinis vaizdas

4. PROJEKTINĖ DALIS

4.1 Projekto planas

Šiame darbe bus atliekama susidūrimų paieškos algoritmų analizė panaudojant lygiagrečius skaičiavimus. Šiai analizei atlikti bus pasitelkta sąlyginai nauja *NVidia* lygiagrečių skaičiavimų technologija, vadinama *CUDA*. Tai yra lygiagrečių skaičiavimų architektūra ir programavimo modelis, kuris leidžia ženkliai padidinti kompiuterių našumą pasitelkiant grafikos procesorių (*GPU*). Ši technologija pradėta naudoti nuo 2006 metų ir iki šiol plačiai naudojama įvairiose aplikacijose, fizikos simuliacijose, kompiuteriniuose žaidimuose ir moksliniuose tyrimuose. Tam, kad būtų galima naudoti šią *NVidia* technologiją, programinės įrangos kūrėjai, mokslininkai ir tyrėjai gali įtraukti *GPU* palaikymą į savo programas, naudojant vieną iš trijų metodų:

1. Senas bibliotekas, kurios naudoja tik *CPU*, pakeisti arba papildyti naujomis su *GPU* palaikymu.
2. Panaudojant *OpenACC* greitiklių direktyvas, kurių pagalba yra automatiškai išlygiagretinamas kodas, parašytas *Fortran* arba *C* programavimo kalbomis.
3. Naudoti tradicines programavimo kalbas tokias kaip *C*, *C++*, *C#*, *Fortran*, *Java*, *Python* ir t.t., kurios palaiko lygiagretų programavimą.

Šio darbo projektinei daliai atlikti yra pasitelkiamas trečiasis iš aukščiau paminėtų metodų. Realizavus pasirinktus susidūrimų paieškos algoritmus, jie bus naudojami tolimesnei analizei atlikti. Pasitelkiant Visual Studio programavimo įrankį, šie algoritmai yra pritaikyti taip, jog jų skaičiavimams atlikti yra pasitelkiamas ne *CPU*, o *GPU*. Tai įvykdžius, prognozuojamas ženklus realizuotų algoritmų veikimo apimties padidėjimas.

4.2 Techninės ir programinės įrangos reikalavimai

Norint realizuoti bet kokį algoritmą, kuris naudoja *GPU* skaičiavimams atlikti, pasitelkiant *CUDA*, reikalingi tiek programinės įrangos, tiek techninės įrangos reikalavimai.

Techninės įrangos reikalavimai:

- Grafinis procesorius, kuris palaiko lygiagretaus programavimo technologiją - *CUDA*. Grafinių procesorių sąrašas, kuriame yra surašyti visi grafiniai procesoriai palaikantys *CUDA* technologiją yra čia [5]. Tai yra *Tesla, Quadro, NVS* ir *GeForce* gaminiai.

Programinės įrangos reikalavimai norint dirbti su *CUDA*:

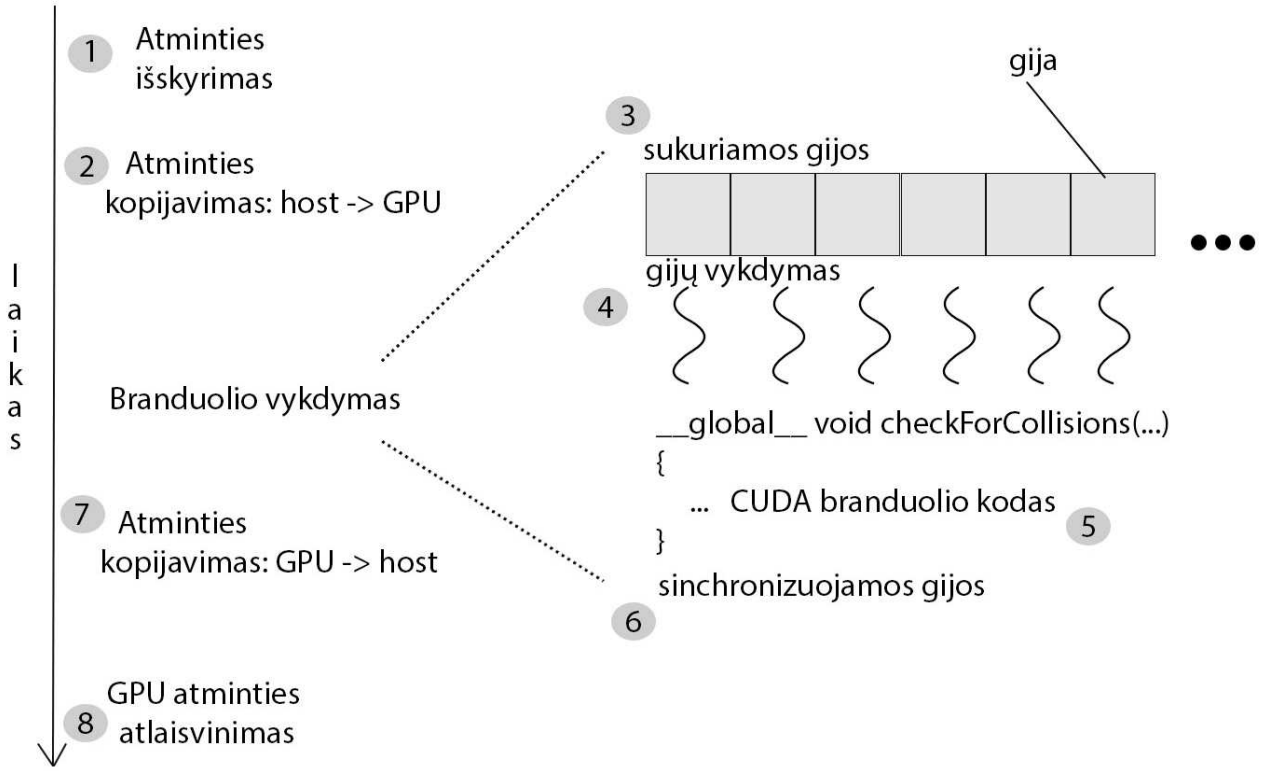
- *CUDA* priemonų rinkinys (angl. *toolkit*).
- *CUDA SDK*.
- *CUDA* programinės įrangos tvarkyklės (angl. *drivers*).
- *Microsoft Visual Studio 2008, 2010* arba *Microsoft Visual C++ Express.*
- *Microsoft Windows XP, Vista, 7, 8* arba *Windows Server 2003 arba 2008.*

4.3 Algoritmų realizavimas

Projektuojant pasirinktus algoritmus ir jų skaičiavimus perduodant į *GPU*, buvo vadovaujamasi trimis pagrindiniais žingsniais:

1. **Integracija.** Sukuriami visi reikalingi komponentai atvaizduoti simuliaciją. Taip pat šiame žingsnyje integruojami objektų parametrai (greitis ir pozicija), reikalingi judėjimui simuliuojamoje erdvėje.
2. **Pasirinkto algoritmo struktūros kūrimas.** Šiame etape realizuojamas pasirinktas algoritmas (*grid, octree* ar *sort and sweep*).
3. **Susidūrimų paieškos vykdymas.** Šis, paskutinysis žingsnis yra pats skaičiavimams reikliausias, todėl jame esantys reikalingi metodai yra aprašomi ir vykdomi grafiniame procesoriuje.

Taip pat yra laikomasi lygiagretaus programavimo, naudojant *CUDA*, darbo eigos principų, kurie pavaizduoti iliustracijoje žemiau (žr. 4.1 pav.).



4.1 pav. Darbo eiga išlygiagretinant algoritmą

4.1 pav. iliustracijos sunumeruoti paaiškinimai:

1. Išskiriama atmintis *GPU*. *GPU* ir *CPU* atmintys yra fiziškai atskirtos, todėl programuotojai turi valdyti išskyrimų kopijas.
2. Kopijuojama atmintis iš *CPU* į *GPU*.
3. Atliekama gijų konfigūracija: pasirenkamas optimaliausias blokų bei tinklelio dimensijos sprendžiamai problemai.
4. Paleidžiamos sukonfigūruotos gijos.
5. Sinchronizuojamos *CUDA* gijos, tam kad įsitikinti, jog *device* baigė visas savo užduotis prieš darant sekančias operacijas *GPU* atmintyje.
6. Kai gijos baigia darbą, atmintis yra kopijuojama iš *GPU* į *CPU*
7. Atlaisvinama *GPU* atmintis.

Realizuojant pasirinktus algoritmus, buvo sukurti metodai reikalingi šiems algoritmams. Jų aprašymo lentelė žemiau (žr. 4.1 lent.).

4.1 lentelė. Pagrindinių algoritmų metodų aprašai

Metodas	Aprašymas
initOpenGL	*OpenGL* bibliotekos pagalba, sukuriamas programos langas, kuris yra naudojamas objektų simuliacijai atvaizduoti
initSphereSystem	Sukuriama nauja SphereSystem klase
initParams	Sukuriami algoritmo parametrai ir priskiriamos jų reikšmės
initMenu	Sukuriamas programos menu, kurio pagalba galima keisti algoritmų parametrus ir atlikti veiksmus
display	Šis metodas atvaizduoja reikiamus komponentus *OpenGl* sukurtame lange
initialize	Šiame metode išskiriama *CPU* ir *GPU* atmintis
addSphere	Į simuliacijos erdvę pridedamos sferos
setSphereArray	Pridėtos sferos įdedamos į masyvą
getSphereArray	Šio metodo pagalba, gaunamas sferų masyvas
update	Šis metodas vykdomas kiekviename kadre
setConstantParameters	Šiame metode kopijuojami parametrai į pastovią atmintį
updateSpheres	Atnaujinami sferų parametrai bei jų pozicijos
integrateSphereSystem	Integruojami sferų parametrų kintamieji
calculateGridHash	Surašomi tinklelio parametrai naudojant hash funkcijas
sortSphereArrayList	Sferų masyvas yra surūšiuojamas
checkForCollisions	Šiame metode atliekami susidūrimų paieškos testai
initGrid	Sukuriamas simuliacijos erdvės tinklelis
initOctree	Sukuriamas simuliacijos erdvės octree medis
updateNodes	Atnaujinami octree medžio mazgai ir juose esantys objektai
createSphereListFromChosenAxis	Pagal pasirinktas koordinačių ašis projektuojami objektų pradžios ir pabaigos taškai ir surašomi į masyvą
findCUDADevice	*CUDA* funkcija kuri suranda tinkamą naudojimui grafinį procesorių
cudaDeviceReset	Atlaisvinama grafinio procesoriaus atmintis

4.3.1 Eksperimentiniai išvesties duomenys

Be šių pagrindinių metodų, reikalingų pasirinktiems algoritmams, buvo suprojektuoti išvesties duomenys, kurių pagalba, tyrimų eksperimentinėje dalyje, bus analizuojami šių algoritmų veikimo našumas. Išvesties duomenys yra šie:

- laikas - apskaičiuojamas per kiek laiko yra įvykdomi susidūrimų paieškos testai vykdant konkretų algoritmą. Laikas skaičiuojamas milisekundėmis (ms).
- kadrai per sekundę - apskaičiuojamas kadrų skaičius įvykstantis per vieną sekundę. Šio kintamojo pagalba galima stebėti algoritmų veikimo našumą ir analizuoti kadrų dažnio kitimą esant skirtingiems algoritmų parametrų reikšmėms.
- grafinio procesoriaus atminties sąnaudos - panaudojus *CUDA* vieną iš funkcijų (4.1), kurios parametrai $free$ ir $total$ reiškia atitinkamai laisvą ir visą grafinio procesoriaus atmintį. Šis metodas grąžina atminties kiekį baitais ir yra naudingas analizuojant skirtingų algoritmų priklausomumą nuo grafinio procesoriaus atminties kiekio, reikalingo konkrečiam algoritmui.

$$cudaMemGetInfo(size\_t * free, size\_t * total); \quad (4.1)$$

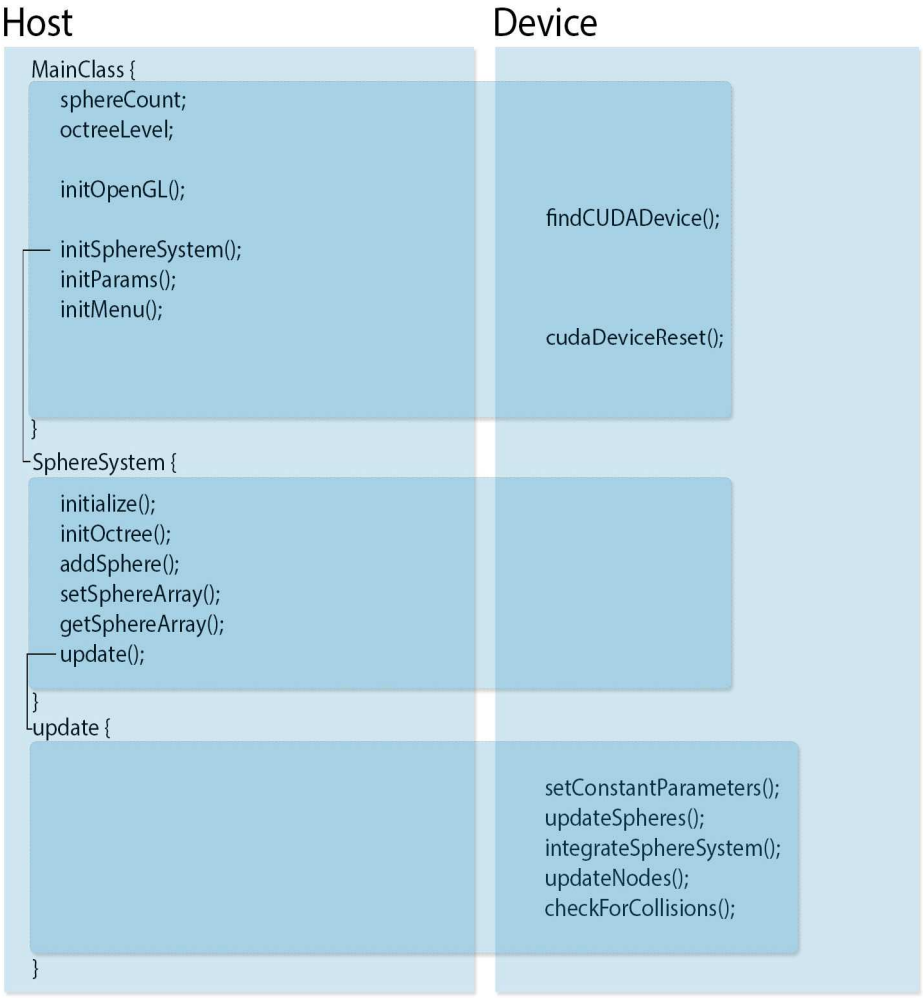
4.3.2 Duomenų struktūra GPU

Norint simuliuoti fizinį išgaubtų objektų judėjimą, tų objektų atributai turi būti saugojami. Skaičiavimams atlikti, naudojant *CPU*, dažniausiai saugojama *1D* masyvuose, tačiau atliekant

skaičiavimus pasitelkus grafinį procesorių geriau yra naudoti *2D* masyvus norint saugoti kintamuosius, dėl *device* apribojimų ir optimizavimo priežasčių.

4.3.3 Octree algoritmo išlygiagretinimas

Realizuotas *octree* algoritmas. Šio algoritmo pagrindiniai metodai paskirstyti tarp *CPU* ir *GPU*. Daug skaičiavimų reikalaujantys metodai buvo perduodami į *GPU* ir ten išlygiagretinami. Algoritmo schema pavaizduota žemiau (žr. 4.2 pav.).

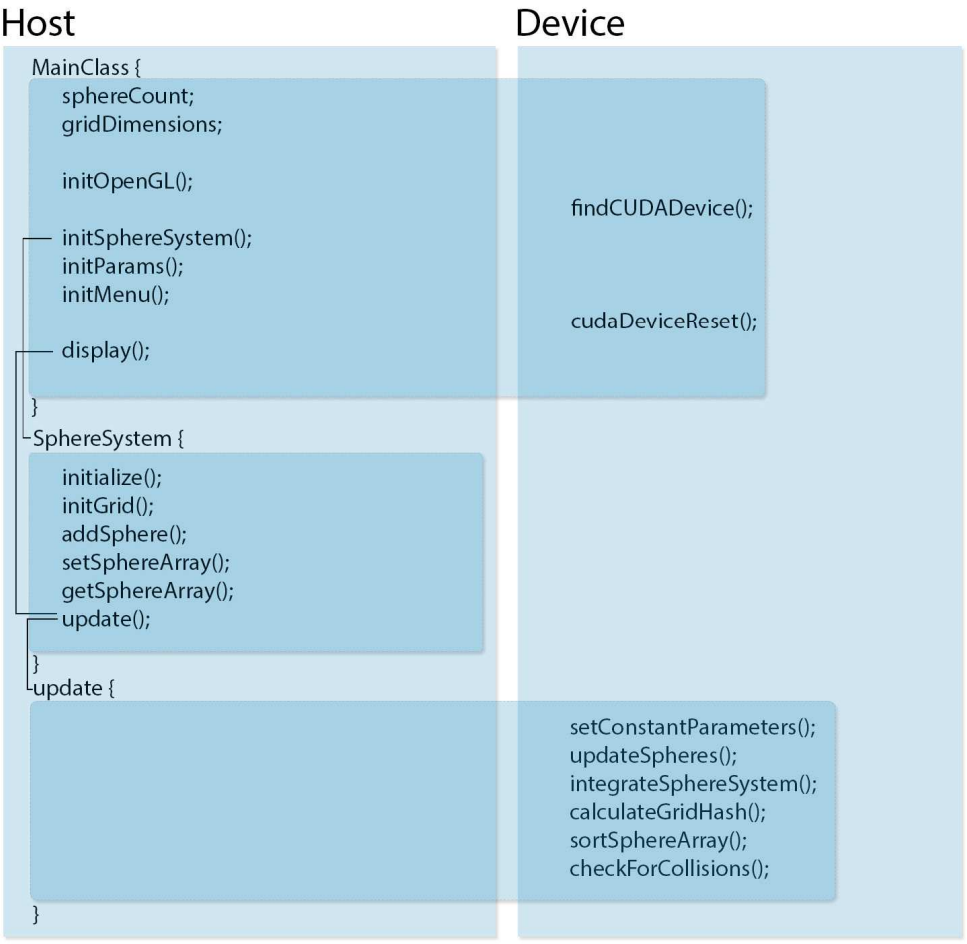


4.2 pav. *Octree* algoritmo pagrindinių metodų paskirstymas tarp *CPU* ir *GPU*

Programos vykdymo metu pasirenkamas norimas octree medžio lygis, pagal kurį, $initOctree()$ metodo pagalba, yra konstruojama šio algoritmo struktūra. Taip pat nurodant norimą sferų skaičių, $addSphere()$ metodo pagalba, jų rodyklės yra sudedamos į *octree* medžio struktūrą. Įvykdžius pirmuosius du žingsnius (integraciją ir pasirinkto algoritmo struktūros kūrimą), toliau vykdomas $update()$ metodas, kuris yra kviečiamas kiekvieną kadrą. Šiame metode yra nustatomos parametrų reikšmės, atnaujinama sferų pozicija, integruojami sukurta sferų sistema į *device*. Kiekviename kadre atnaujinami medžio struktūros mazgai ir juose esančios objektų rodyklės. Paskutinysis metodas $checkForCollisions()$ vykdo susidūrimų paieškos testus atskirai kiekviename mazge ir taip sumažinant šių testų skaičių.

4.3.4 Grid algoritmo išlygiagretinimas

Realizuotas *grid* algoritmas. Šio algoritmo pagrindiniai metodai paskirstyti tarp *CPU* ir *GPU*. Daug skaičiavimų reikalaujantys metodai buvo perduodami į *GPU* ir ten išlygiagretinami. Algoritmo schema pavaizduota žemiau (žr. 4.3 pav.).

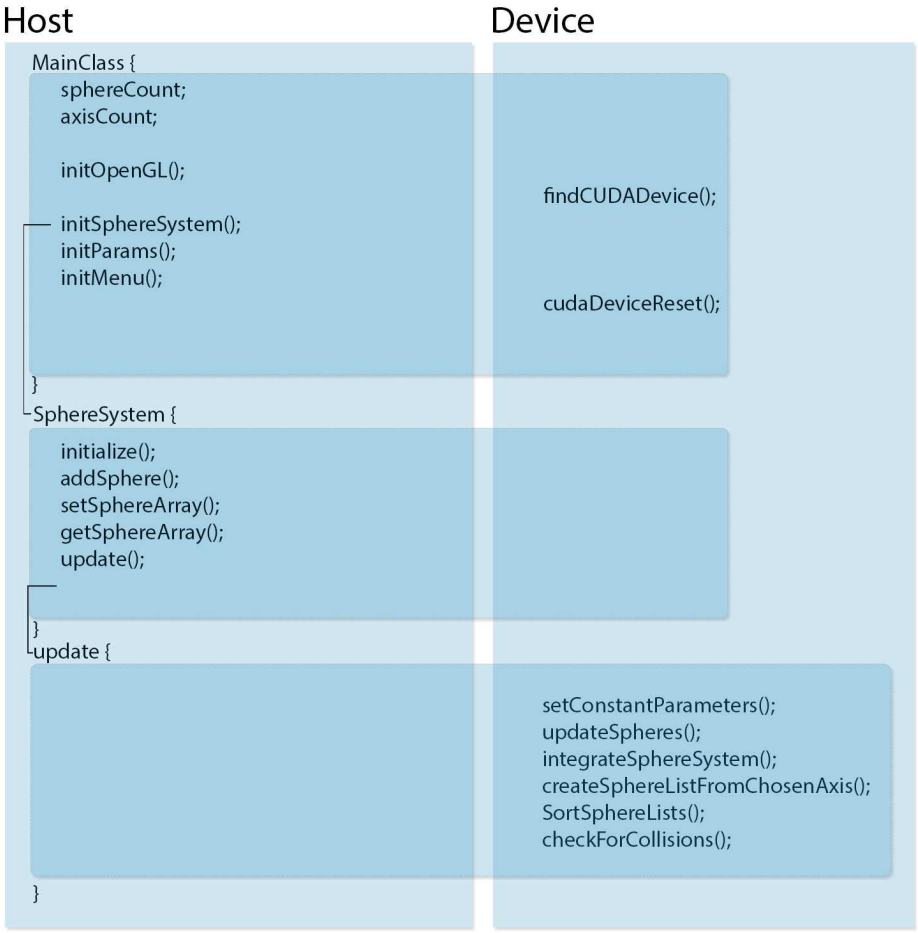


4.3 pav. *Grid* algoritmo pagrindinių metodų paskirstymas tarp *CPU* ir *GPU*

Šio algoritmo metodai, kuriems reikia didelių skaičiavimo resursų yra $calculateGridHash$ ir $checkForCollisions$. Pirmajame metode *hash* funkcijų pagalba suskaičiuojami ir surašomi kiekvieno objekto parametrai. Tuo tarpu $checkForCollisions$ metode atliekami susidūrimų paieškos testai kiekvienoje gardelėje atskirai ir taip pat šalia esančiose.

4.3.5 Sort and Sweep algoritmo išlygiagretinimas

Realizuotas *grid* algoritmas. Šio algoritmo pagrindiniai metodai paskirstyti tarp *CPU* ir *GPU*. Daug skaičiavimų reikalaujantys metodai buvo perduodami į *GPU* ir ten išlygiagretinami. Algoritmo schema pavaizduota žemiau (žr. 4.4 pav.).



4.4 pav. *Sort and sweep* algoritmo pagrindinių metodų paskirtymas tarp *CPU* ir *GPU*

Šio algoritmo metodai, kuriems reikia didelių skaičiavimo resursų yra $createSphereListFromChosenAxis()$ ir $checkForCollisions()$. Pirmajame metode projektuojami objektų pradžios ir pabaigos taškai ant pasirinktų koordinačių ašių atskirai ir surašomi į išvesties masyvus. Tuomet surūšiavus visus šiuos masyvus didėjimo tvarka, atliekami susidūrimų paieškos testai kiekviename masyve.

5. TYRIMO EKSPERIMENTINĖ DALIS

5.1 Atliekamo tyrimo metodologija

Šiame darbe yra atliekami susidūrimų paieškos algoritmų našumo tyrimai. Algoritmų našumas yra matuojamas nustatant jų pilną vykdymo laiką. Visi šiame darbe pateikti laiko matavimai yra išskaičiuojami kaip trijų to paties matavimo paleidimų vidurkis. Visų matavimų metu yra išlaikoma ta pati matavimui naudojamos aparatūrinės bei programinės įrangos konfigūracija. Matavimai yra tiesiogiai priklausomi nuo matavimo įrangos, todėl jų atlikimas kitoje aplinkoje gali pateikti kitokius rezultatus.

5.2 Pasirinktų tyrimų paskirtis

Šiame darbe vykdomų tyrimų sąrašas:

1. Algoritmų laiko priklausomybė nuo objektų kiekio tyrimas.
2. Laiko priklausomybė nuo objektų kiekio, keičiant parametrus, tyrimas.
3. Kiekybinis algoritmų laiko ir našumo palyginimo tyrimas.
4. Grafinio procesoriaus atminties sąnaudų tyrimas.

Šiame darbe atliekamų tyrimų aprašymai:

1. Nustatyti susidūrimų paieškos algoritmo *grid, octree ir sort and sweep* vykdymo laiko bei kadrų dažnio priklausomybę nuo objektų kiekio. Bendru būdu ištirti šių algoritmų veikimą skirtingų paskirčių procesoriuose.
2. Nustatyti visų tiriamų algoritmų veikimo laiko bei kadrų dažnio priklausomybę keičiant jų parametrus. Bendru būdu ištirti šių algoritmų veikimą skirtingų paskirčių procesoriuose.
3. Bendru būdu ištirti susidūrimų paieškos algoritmų veikimą skirtingų paskirčių procesoriuose. Pateikti svarių argumentų, jog grafinis procesorius gali būti sėkmingai naudojamas susidūrimų paieškos algoritmo *grid, octree ir sort and sweep* veikimo tobulinimui. Kiekybiškai palyginti skirtingų paskirčių procesorių vykdymo laikus ir našumo skirtumus.
4. Atlikti grafinio procesoriaus atminties sąnaudų tyrimą realizavus pasirinktus algoritmus.

5.3 Tyrimui naudojamos įrangos bei parametrų aprašas

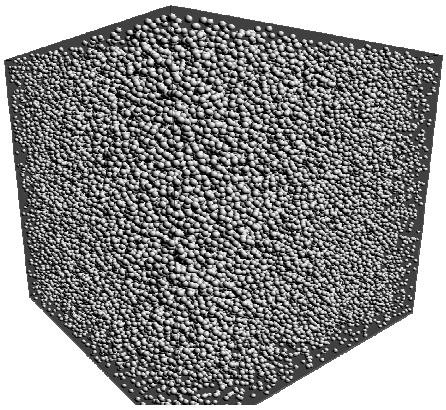
Tyrimui atlikti buvo pasirinkta aparatūrinė bei programinė įranga aprašyta lentelėje žemiau (žr. 5.1 lent.).

5.1 lentelė. Naudojamo kompiuterio techninių duomenų lentelė

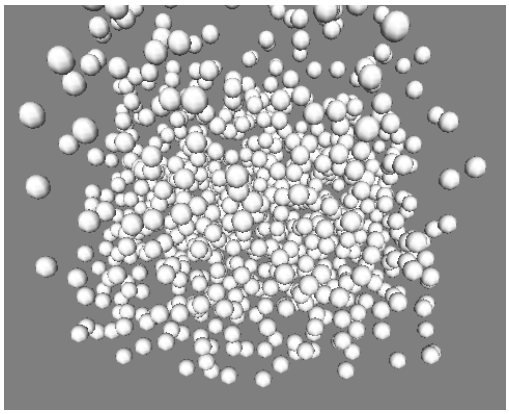
Procesorius	AMD Athlon(tm) II X4 640 3,00 GHz
Atmintis (RAM)	6,00 GB
Systemos tipas	64-bitų operacinė sistema
Grafinis procesorius	GeForce GTS 450
NVidia tvarkyklės versija	306,97
DirectX palaikymas	11,1
CUDA branduolių	192
Video atmintis	1024 MB GDDR5
Programa/Biblioteka/Įrankis	Microsoft Visual Studio, OpenGL, CUDA C

Visų atliktų tyrimų rezultatai yra tiesiogiai priklausomi nuo aukščiau aprašytos įrangos. Tai reiškia, jog kitos konfigūracijos atveju yra galimybė gauti skirtingus rezultatus.

Norint atlikti tyrimus, buvo realizuoti trys susidūrimų paieškai pritaikyti algoritmai: *grid, octree ir sort and sweep* ant *CPU* ir *GPU*. 5.1 bei 5.2 paveiksluose pavaizduotos šių algoritmų simuliacijos veikimas, skaičiavimus atliekant *CPU* (žr. 5.1 pav.) ir *GPU* (žr. 5.2 pav.).



5.1 pav. Sferų simuliacijos vaizdas ant *GPU* (50 000 sferų)

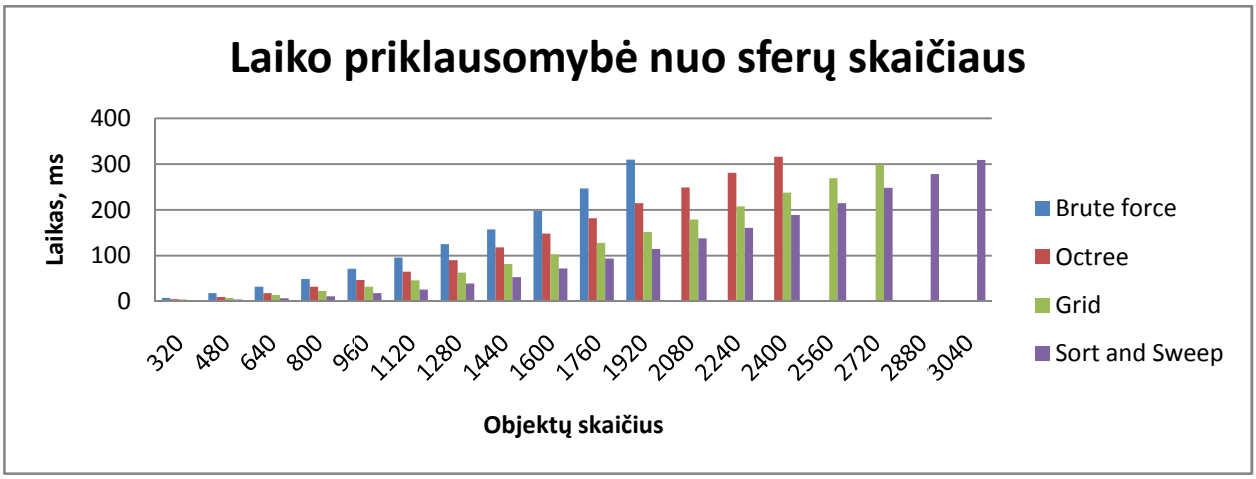


5.2 pav. Sferų simuliacijos vaizdas ant *CPU* (640 sferų)

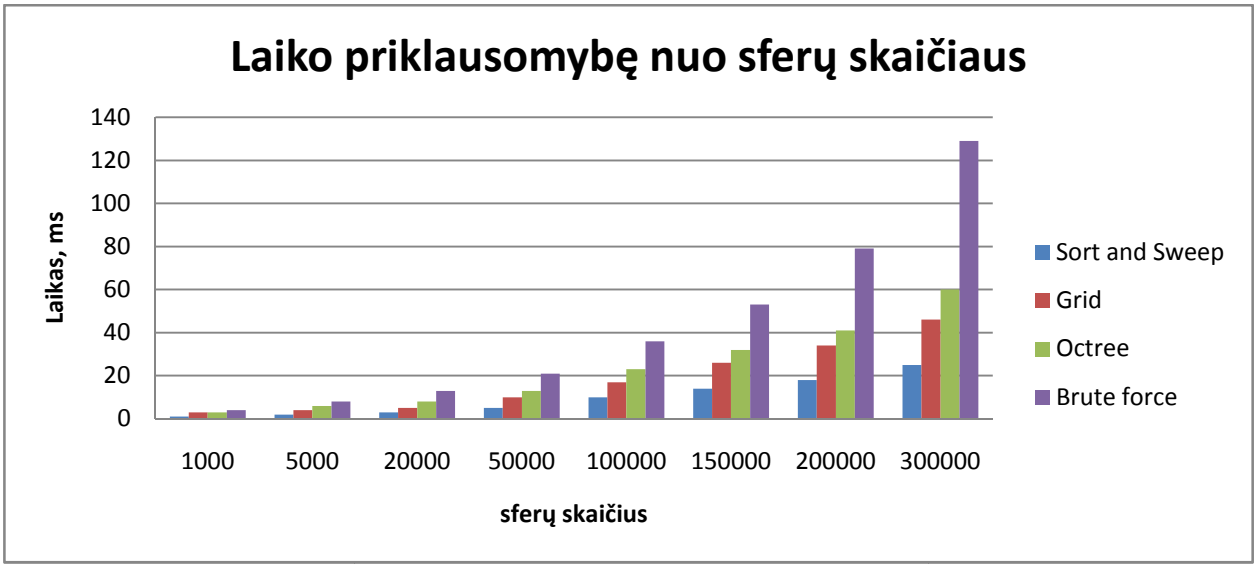
5.4 Tyrimo rezultatai

5.4.1 Algoritmų laiko priklausomybė nuo objektų kiekio tyrimas

Šiame tyrime, realizavus pasirinktus algoritmus buvo nustatyti šių algoritmų vykdymo laiko ir kadrų dažnio priklausomybė nuo objektų kiekio simuliuojamoje erdvėje (žr. 5.3 pav. ir 5.4 pav.). Algoritmai realizuoti ant skirtingų paskirčių procesorių *CPU* ir *GPU*.



5.3 pav. Laiko priklausomybė nuo objektų skaičiaus (*CPU*)



5.4 pav. Laiko priklausomybė nuo objektų skaičiaus (*GPU*)

Gavus šiuos duomenis galima teigti, jog mažiausiai laiko skaičiuojant susidūrimų paiešką užtrunka *sort and sweep* algoritmas. *Grid* algoritmas yra taip pat geras, siekiant optimizuoti susidūrimų paiešką, tačiau jo veikimo našumas šiek tiek nusileidžia *sort and sweep* algoritmui dėl sudėtingesnės struktūros ir duomenų struktūros atnaujinimo. Nors *Octree* ir *Grid* algoritmai mažai kuo skiriasi, tačiau iš gautų rezultatų galima teigti, kad *Octree* algoritmo našumas yra prastesnis už *Grid*, dėl to, kad *Octree* duomenų struktūra reikia nuolatos atnaujinti.

Tyrimo nr. 1 metu, didinant objektų skaičių, buvo matuojamas kiekvieno algoritmo laikas, reikalingas susidūrimų paieškai atlikti. Tas pats eksperimentas buvo vykdomas tris kartus ir gautas laiko vidurkis buvo fiksuojamas. Vykdant eksperimentus ant *CPU*, objektų skaičiaus intervalas buvo pasirinktas nuo 80 iki 3040. Buvo nuspręsta, jog jei kurio nors algoritmo skaičiavimo laikas užtrunka daugiau nei 300 milisekundžių (0.3sec), tas algoritmas toliau yra nebetiriamas. Gautus rezultatus galima pamatyti lentelėje žemiau (žr. 5.2 lent.)*Sort nat Sweep* algoritmo pagalba, ties 0.3 sec, kurių prireikia algoritmui atlikti susidūrimų paieškos testus, gali palaikyti apie 3000 objektų. Tuo tarpu prasčiausias tirtas algoritmas - apie 1900 objektų.

5.2 lentelė. Objektų skaičius esant tam pačiam susidūrimų paieškos algoritmų skaičiavimo laikui (300ms)**.**

	Brute force	Octree	Grid	Sort and Sweep
Objektų skaičius	1920~	2420~	2780~	3020~

Procentalus objektų skaičiaus padidėjimas, naudojant realizuotus algoritmus (žr. 5.3 lent). Palyginimui paimtas prasčiausias *Brute force* algoritmas esant apie 1920 objektų simuliuojamoje aplinkoje. Gauti rezultatai rodo, kad *octree*, *grid* ir *sort and sweep* algoritmai gali palaikyti atitinkamai 19%, 31% ir 36% didesnį kiekį objektų erdvėje.

5.3 lentelė. Procentalus objektų padidėjimas naudojant susidūrimų paieškos algoritmus lyginant su *brute force*

	Octree	Grid	Sort and Sweep
Procentalus objektų skaičiaus padidėjimas	19%	31%	36%

Realizavus tuos pačius algoritmus ant *GPU* gauti rezultatai pavaizduoti diagramoje (žr. 5.4 pav.). Panaudojus *GPU*, galima sugeneruoti kur kas didesnį objektų skaičių simuliuojamoje aplinkoje, esant konkrečiam susidūrimų paieškai reikalingam laikui.

Tyrimo nr. 1 metu, didinant objektų skaičių, buvo matuojamas kiekvieno algoritmo laikas, reikalingas susidūrimų paieškai atlikti. Tas pats eksperimentas buvo vykdomas tris kartus ir gautas laiko vidurkis buvo fiksuojamas. Vykdant eksperimentus ant *GPU*, objektų skaičiaus intervalas buvo pasirinktas nuo 1000 iki 300 000. Algoritmams pasiekus maksimalų užsibrėžtą objektų skaičių, fiksuojamas laikas, kuris yra reikalingas susidūrimų paieškai atlikti. Gautus rezultatus galima pamatyti lentelėje žemiau (žr.5.4 lent.).

5.4 lentelė. Laikas, reikalingas atlikti susidūrimų paieškos testus

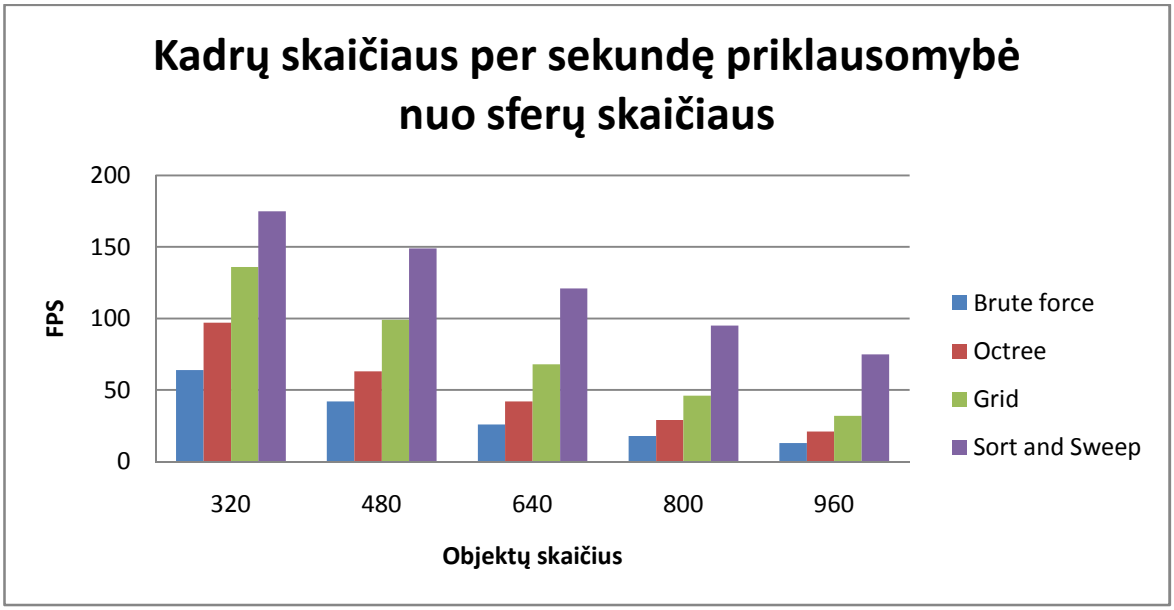
	Brute force	Octree	Grid	Sort and Sweep
Laikas, ms	129	60	46	25

Iš gautų rezultatų galima spręsti, kad, optimizuojant susidūrimų paieškos algoritmus, geriausias algoritmas yra *Sort and Sweep*. Esant objektų skaičiui *n = 300 000* šiam algoritmui užtenka 25 milisekundžių apskaičiuojant objektų susidūrimų paieškos testus, tuo tarpu *Grid* ir *Octree* algoritmai užtrunka atitinkamai 46 bei 60 milisekundžių. Šie trys algoritmai, *sort and sweep, grid ir octree*, yra kur kas spartesni už *Brute force* algoritmą ir yra atitinkamai 5.16, 2.8 ir 2.16 karto greitesni už jį laiko požiūrių.

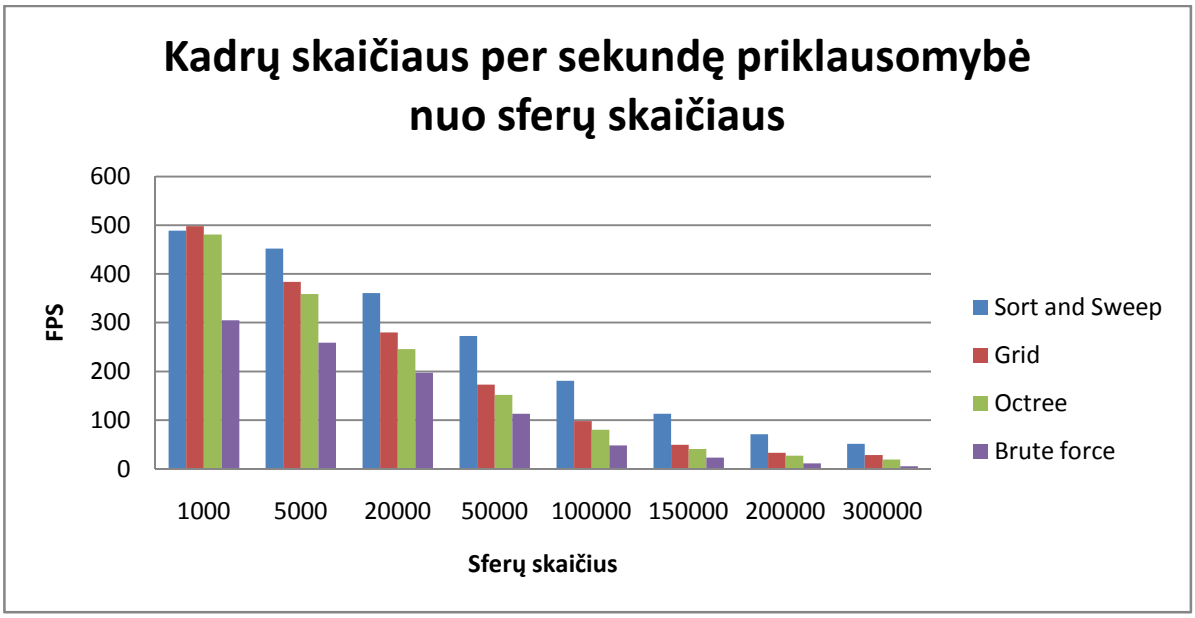
Lyginant *CPU* ir *GPU* gautus rezultatus, gautas dramatiškas objektų skaičiaus padidėjimas realizavus šiuos algoritmus ant *GPU*. Pvz. net ir geriausiai veikiantis ant *CPU* algoritmas (*sort and sweep*), negali prilygti prasčiausiam *GPU* algoritmui (*brute force*).

Atlikus realizuotų algoritmų kadrų dažnio priklausomybę nuo objektų skaičiaus simuliuojamoje aplinkoje gauti atvirkščiai analogiški rezultatai ankščiau tirtiems rezultatams, kur buvo tiriama algoritmų laiko, reikalingo atlikti susidūrimų paieškos testus, priklausomybė nuo objektų skaičiaus.

Buvo gauta, jog didinus objektų skaičių simuliuojamoje aplinkoje, kadrų skaičius krenta (žr. 5.5 pav. ir 5.6 pav.). Iš šių rezultatų matyti, jog našiausiai ir didžiausią kadrų dažnį palaiko *Sort and Sweep* algoritmas.



5.5 pav. Kadrų skaičiaus priklausomybė nuo sferų skaičiaus (*CPU*)



5.6 pav. Kadrų skaičiaus priklausomybė nuo sferų skaičiaus (*GPU*)

5.4.2 Laiko priklausomybė nuo objektų kiekio, keičiant parametrus, tyrimas

Nustatyta visų tiriamų algoritmų veikimo laiko bei kadrų dažnio priklausomybę keičiant jų parametrus. Bendru būdu ištirtas šių algoritmų veikimas skirtingų paskirčių procesoriuose (žr. 5.5 lent.).

5.5 lentelė. Laiko ir kadrų skaičiaus per sekundę priklausomybės lentelė nuo keičiamų algoritmo parametrų ir objektų skaičiaus, kur *ms* žymimas laikas milisekundėmis, o *fps* - kadrų skaičius per sekundę.

Objektų skaičius	Octree						Grid							
	Level 1		Level 6		Level 8		grid 32		grid 64		grid 128		grid 256	
	ms	fps	ms	fps	ms	fps	ms	fps	ms	fps	ms	fps	ms	fps
1000	2	261	1	481	2	443	3	498	3	442	2	350	4	261
5000	3	209	2	359	3	329	4	384	4	341	3	256	5	209
20000	7	157	4	246	6	216	5	280		260	4	198	7	157
50000	12	80	8	152	10	132	10	173	7	155	6	126	9	80
100000	20	45	13	80	16	64	17	98	11	75	12	70	15	45
1500000	31	29	19	41	24	35	26	49	16	48	16	40	20	30
2000000	45	18	25	27	34	21	34	33	22	35	24	32	26	21
300000	75	9	39	19	58	12	58	28	33	27	53	21	36	13

Kaip ir buvo minėta ankščiau, *grid* bei *octree* algoritmų veikimo našumas gali skirtis dėl naudojamos simuliuojamos aplinkos, objektų skaičiaus bei pačiame algoritme esančių parametrų. Atlikus šį tyrimą buvo gauta, jog keičiant *grid* algoritmo tinklo gardelių dydį bei *octree* algoritmo medžio gyli gaunami rezultatai, kurie daugiau ar mažiau skiriasi vienas nuo kito. Tai yra todėl, kad susidūrimų paieškai reikalingas laikas yra teisiogiai priklausomas nuo keičiamų parametrų. Įvykdžius keletą eksperimentų su skirtingomis parametrų reikšmėmis buvo nustatyta, jog grid algoritmas su tinklo gardelių dydžiu 64 pasiekia didžiausią veikimo našumą. Esant 300 000 objektų simuliuojamoje aplinkoje, algoritmui užtenka 33 milisekundžių apskaičiuoti susidūrimų paieškos testus ir programa veikia ties 27 kadrais per sekundę. Tačiau, esant tinklo dydžiui lygiam 32, gaunamas minimalus kadrų skaičiaus padidėjimas iki 28, bet laikas reikalingas susidūrimų paieškai atlikti yra beveik dvigubai didesnis. Tai yra dėl to, kad, esant tinklo gardelių dydžiui lygiam 32, algoritmui tenka atlikti kur kas didesnį susidūrimų paieškos testų skaičių kiekvienoje gardelėje, kadangi jos yra didesnės. Tuo tarpu *octree* algoritmas su medžio gylio lygiu 6 yra pats optimaliausias ir našiausiai veikiantis variantas iš visų ištirtų, kur esant tokiam pačiam objektų kiekiui, susidūrimų paieška užtrunka 39 milisekundes ir programa viekia apytiksliai ties 19 kadrų dažniu.

5.4.3 Kiekybinis algoritmų laiko ir našumo palyginimo tyrimas

Rezultatai, gauti tyrimo nr. 3 metu, atvaizduoti lentelėje žemiau (žr. 5.6 lent.). Buvo fiksuojamas objektų skaičius programai veikiant 50 kadrų dažniu ir ties 25 milisekundėms atskirai, kurių prireikdavo algoritmams atlikti susidūrimų paieškos testus.

5.6 lentelė. Algoritmų našumo palyginimo lentelė tarp *CPU* ir *GPU*

Algoritmas	Objektų skaičius programai veikiant ties 50 fps		Objektų skaičius algoritmams užtrunkant 25 ms		Objektų padidėjimas	
	CPU	GPU	CPU	GPU	50 fps	25 ms
Brute force	400	90000	560	5500	225	98
Octree	600	140000	720	100000	233	139
Grid	700	150000	850	150000	214	176
Sort and Sweep	1020	300000	1120	300000	294	268

Įvykdžius visus užsibrėžtus tikslus, realizavus visus algoritmus ant *CPU* bei *GPU* ir atlikus visus reikiamus eksperimentinius tyrimus, galima teigti, jog grafinis procesorius yra tinkamas šiems algoritmas ir jų atliekamiems skaičiavimams. Pasitelkus grafinį procesorių, algoritmai gali veikti kur kas efektingiau ir našiau, palaikant kur kas didesnį simuliuojamų objektų skaičių. Šiame darbe buvo išsiaiškintas našiausias algoritmas iš visų tirtų, tai buvo *Sort and Sweep* algoritmas. Išlygiagretinus šį algoritmą, panaudojant *CUDA* technologiją, buvo gautas 300 kartų našumo padidėjimas objektų skaičiaus atžvilgiu. Iš to galima spręsti, jog grafinis procesorius gerai susidoroja su procesais, kurie reikalauja didelių skaičiavimų.

5.4.4 Grafinio procesoriaus atminties sąnaudų tyrimas

Šiame tyrime buvo analizuojamas blokų bei gijų skaičius reikalingas išlygiagretinti algoritmus panaudojant *CUDA* įrankį. Tyrimo rezultatai pavaizduoti lentelėje žemiau (žr. 5.7 lent.).

5.7 lentelė. Blokų bei gijų skaičius reikalingas realizuotiems algoritmams. Pasirinktas bendras skaičius $\boldsymbol{n = 256}$, kuris reiškia gijų skaičių viename bloke.

Objektų skaičius	Grid		Octree		Sort and Sweep	
	Blokų skaičius	Gijų skaičius	Blokų skaičius	Gijų skaičius	Blokų skaičius	Gijų skaičius
1000	12	3072	18	4608	15	3840
5000	60	15360	65	16640	63	16128
10000	120	30720	125	32000	123	31488
20000	237	60672	249	63744	240	61440
50000	588	150528	594	152064	591	151296
100000	1173	300288	1181	302336	1176	301056
150000	1758	450048	1765	451840	1761	450816
200000	2346	600576	2356	603136	2349	601344
300000	3516	900096	3529	903424	3519	900864

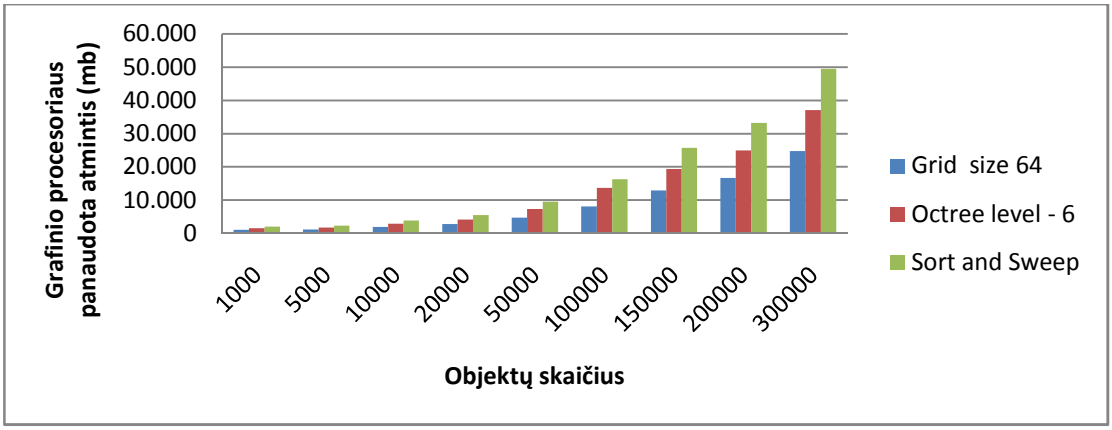
Dėl *CUDA* aparatūrinių išteklių pirmosios taisyklės, kuri teigia jog gijų skaičius negali viršyti 512 gijų viename bloke, buvo pasirinktas optimalus 256 gijų skaičius viename bloke. Atlikus šį eksperimentą, buvo gauta, jog gijų skaičius yra tiesiogiai priklausomas nuo objektų skaičiaus, kadangi kiekvienam naujam objektui buvo sukuriama nauja gija.

Taip pat, šiame tyrime buvo analizuojamos grafinio procesoriaus atminties sąnaudos, reikalingos kiekvienam realizuotam algoritmui. Buvo nustatyta, jog vien tik sukurti pasirinktų algoritmų struktūrą, tai pareikalauja atitinkamų grafinio procesoriaus atminties sąnaudų. *Grid* algoritmo struktūra, esant gardelių dydžiui lygiam 64, pareikalauja 16 MB grafinio procesoriaus atminties. Tuo tarpu *octree* (level 6) - 18 MB, o *sort and sweep* - 25 MB. Rezultatai pavaizduoti lentelėje žemiau (žr. 5.8 lent.).

5.8 lentelė. Grafinio procesoriaus atminties sąnaudos reikalingos algoritmų struktūrai sukurti.

Atminties kiekis	Grid				Octree				Sort and Sweep
	grid 32	grid 64	grid 128	grid 256	level 1	level 3	level 6	level 8	visos 3 ašys
MB	1	2	16	128	2	3	18	156	25

Taip pat, šiame tyrime buvo analizuojama grafinio procesoriaus atminties sąnaudų priklausomybė nuo generuojamų objektų skaičiaus simuliuojamoje erdvėje. Grafinio procesoriaus atminties sąnaudų rezultatai bei jų augimas, didinant objektų skaičių, pavaizduoti diagramoje (žr.5.7 pav.) ir lentelėje (žr. 5.9 lent.).



5.7 pav. Grafinio procesoriaus atminties sąnaudų diagrama

Atliktus šį tyrimą, buvo pastebėta, jog didinant objektų skaičių simuliuojamoje aplinkoje, grafinio procesoriaus sąnaudos didėja. Tai yra todėl, kad taip pat didėja ir gijų bei blokų skaičius, o kiekvienam naujam blokui sukurti yra išskiriama grafinio procesoriaus atmintis.

5.9 lentelė. Grafinio procesoriaus atminties sąnaudos reikalingos realizuotiems algoritmams. Lentelėje pateiktos reikmės yra megabaitais (MB).

Objektų skaičius	Grid				Octree			Sort and Sweep
	Grid 32	Grid 64	Grid 128	Grid 256	Level 1	Level 6	Level 8	Visos 3 ašys
1000	0,816	0,827	0,715	0,824	1,023	1,535	2,148	2,046
5000	1,191	0,949	0,941	0,941	1,133	1,700	2,379	2,266
10000	1,618	1,106	1,098	1,098	1,908	2,862	4,007	3,816
20000	2,930	1,543	1,535	1,535	2,746	4,119	5,767	5,492
50000	3,535	4,535	4,570	4,543	4,850	7,275	10,185	9,500
100000	7,754	7,922	7,957	7,910	9,125	13,688	19,163	16,250
150000	13,934	12,660	12,660	12,676	12,875	19,313	27,038	25,750
200000	16,410	16,410	16,410	16,410	16,629	24,944	34,921	33,258
300000	24,410	23,910	24,418	24,410	24,746	37,119	51,967	49,492

Sudėjus gautus rezultatus esančius 5.9 lent. ir 5.8 lent. gauname, jog realizuoti algoritmai *grid*, *octree* ir *sort and sweep*, su optimaliausiais jų parametrais, išanalizuotais ankstesniuose tyrimuose, simuliuojamoje aplinkoje esant 300000 objektų, išnaudoja atitinkamai 26 MB, 40 MB ir 75 MB grafinio procesoriaus atminties.

6. IŠVADOS

1. Realizavus tradicinius susidūrimų paieškos *grid*, *octree ir sort and sweep* algoritmus, buvo nustatyta, jog savo paprastumu įgyvendinime bei veikimo našumu išsiskyrė paskutinysis *sort and sweep* algoritmas. Esant tam pačiam objektų skaičiui, jis atlieka susidūrimų paiešką 2,4 kartų greičiau už *octree* ir 1,8 kartų už *grid*. Šio algoritmo veikimui įtaką padarė maža grafinio procesoriaus divergencija, dėl simuliuojamoje aplinkoje panaudotų vienodo dydžio objektų. Tačiau šis algoritmas reikalauja daugiausiai grafinio procesoriaus atminties resursų, dėl globalios atminties naudojimo bei vykdant visų ašių susidūrimų paiešką. Tuo tarpu *grid* bei *octree* algoritmų veikimas yra gana panašus lyginant tarpusavyje, dėl jų įgyvendinimo principų panašumų, tačiau šiek tiek nusileidžia *sort and sweep* algoritmui dėl jų duomenų struktūros nuolatinio atnaujinimo būtinumo. Įvykdžius visus šio darbo eksperimentinius tyrimus buvo nustatyta, jog tradiciniai susidūrimų paieškos algoritmai tokie kaip *grid*, *octree* ir *sort and sweep* algoritmai yra tinkami išlygiagretinimui bei naudojimui grafiniuose procesoriuose. To rezultate, gaunamas ženklus algoritmų našumo padidėjimas. Pavyzdžiui *sort and sweep* algoritmo veikimas naudojant *CPU* kadrų dažniui esant 60, gali būti palaikoma vos 1000 objektų simuliuojamoje aplinkoje, tuo tarpu, išlygiagretinus šį algoritmą ir perdavus susidūrimų paieškos testų skaičiavimus grafiniam procesoriui, esant tam pačiam kadrų skaičiui, lygiam 60, gali palaikyti net apie 200 000 objektų virtualioje aplinkoje, kurie sąveikauja tarpusavyje. To rezultate gautas 200 kartų didesnis *GPU* našumas už *CPU*. Tačiau, kaip ir buvo minėta ankščiau, algoritmų rezultatai priklauso ne tik nuo jų įgyvendinimo išlygiagretinant, tačiau taip pat ir nuo naudojamos aparatūrinės įrangos, t.y. esant spartesniam grafiniam procesoriui su didesniu kiekių branduolių, galima gauti dar geresnius rezultatus.
2. Šiame darbe buvo nustatytos ištirtų algoritmų parametrų reikšmės, su kuriomis algoritmai pasiekia aukščiausią našumą. Tai yra *grid* algoritmas su tinklelio gardelių dydžių 64, *octree* algoritmas su medžio lygiu 6 ir *sort and sweep* tikrinimas visomis ašimis. *Grid* ir *octree* algoritmų atžvilgiu, atliekant eksperimentinius tyrimus buvo pastebėtas našumo kylimas iki minėtų parametrų reikšmių ir našumo kritimas viršijus šias reikšmes. Tuo tarpu *sort and sweep* algoritmo našumas yra priklausomas nuo pasirinktų ašių skaičiaus, t.y. pasirinkus, pavyzdžiui, tik dvi ašis, susidūrimų paieškos testų atliekama daugiau.
3. Lygiagretinant algoritmus naudojant įrankį *CUDA*, buvo nustatyta, jog išlygiagretinimui grafiniame procesoriuje yra labai svarbu parinkti optimalų gijų bei blokų skaičių, nes tai įtakoja grafinio procesoriaus atminties sąnaudų kitimą apkrovimą bei efektyvumą. Taip pat parinkus skirtingą atminties tipą, galima gauti mažesnes arba didesnes grafinio procesoriaus atminties sąnaudas. Šiame darbe buvo naudojami globalios ir bendros atminties tipai. To rezultate, buvo gauta, jog *grid* algoritmo, esant maksimaliam šiame darbe naudojamų objektų skaičiui, lygiam 300000, grafinio procesoriaus atminties sąnaudos yra lygios 23,91 MB. Tuo tarpu likę du algoritmai, *octree* ir *sort and sweep*, naudoja atitinkamai 35% bei 52% daugiau grafinio procesoriaus atminties.

7. LITERATŪROS SARAŠAS

[1] CUDA C Programming Guide. [Tinkle] Prieiga per internetą: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model [žiūrėta 2013-01-22].

[2] Basic concepts of NVIDIA GPU and CUDA programming. [Tinkle] Prieiga per internetą: http://geco.mines.edu/tesla/cuda_tutorial_mio/index.html [žiūrėta 2013-01-23].

[3] Cuda Tutorial #1. [Tinkle] Prieiga per internetą: http://web.eecs.utk.edu/~lindsey/cuda/project2/cuda.html [žiūrėta 2013-01-23].

[4] EM Photonics, Inc. CUDA Reference Manual, skyrius 5,7, 2011.

[5] CUDA GPUs. A list of graphics processors witch supports CUDA [Tinkle] Prieiga per internetą: https://developer.nvidia.com/cuda-gpus [žiūrėta 2013-05-13].

[6] Dan, Negrut; Gisli Ottarsson, On an Implementation of the Hilber-Hughes-Taylor Method in the context of Index 3 Differential-Algebraic Equations of Multibody Dynamics, 2006.

[7] Grand, S. Broad-Phase Collision Detection with CUDA [Tinkle] Prieiga per internetą: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch32.html [žiūrėta 2013-05-16].

[8] Lefebvre, S; Hoppe, H. Perfect Spatial Hashing, Microsoft Research, 2007.

[9] Liljeby, J. An evaluation of grid based broad phase collision detection for real time interactive environments, Bachelor Thesis, ComputerScience, University of Gavle, 2011.

[10] Daniel, J; Tracy, Samuel R. Buss, Brian; M.Woods. Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal, University of California San Diego, 2008.

[11] Avril, Q; Gouranton, V; Arnaldi, B. New trends in collision detection performance, Université Européenne de Bretagne, France, pages 5-7, 2009.

[12] GPU Collision Detection, Graeme Haddow, BSc (Hons) Computer Games Technology. University of Abertay Dundee, skyrius 2, 2011

[13] Rocha, R; Rodrigues, M; Taddeo, L. Performance Evaluation of a Hybrid Algorithm of Collision Detection in Crowded Interactive Envirinments. Universidade de Fortaleza (UNIFOR), Fortaleza, Brazil, pages 2-4, 2006.

[14] Eberly,D. Real-Time Collision Detection, Magic Software, Inc, skyrius 7, 2005.

[15] Backman, N. Collision Detection of Triangle Meshes using GPU. Umea University, department of computing science, Sweeden, pages 17-20, 2010.

[16] Lucchesi, B. A Parallel Linear Octree Collision Detection Algorithm, University of Nevada, Reno, pages 15-16, 2002.

[17] Hye-Young, Jung; Chul-Woong, Jun; and Jeong-Hyun Sohn, GPU-based collision analysis betwenn a multi-body system and numerous particles, Pukyong National University, page 4, 2012.