KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Jonas Kaulakis

Kompiuterinių žaidimų varikliuko architektūra

Magistro darbas

Darbo vadovas

dr. Tomas Blažauskas

Kaunas, 2006

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Jonas Kaulakis

Kompiuterinių žaidimų varikliuko architektūra

Magistro darbas

Kalbos konsultantė	Lietuvių k. katedros lekt.	Atliko	IFM-01 gr. stud. Jonas Kaulakis
2006-05			
Recenzentas	dr. Eduardas Bareiša	Vadovas	dr. Tomas Blažauskas
2006-05			

Kaunas, 2006

Kompiuterinių žaidimų varikliuko architektūra

Santrauka

Žaidimų varikliukas – tai įvairių pagalbinių specializuotų įrankių ir funkcijų visuma, skirta žaidimams kurti. Tai komponentas, skirtas pakartotinam naudojimui įvairiuose žaidimuose. Kaip ir bet kokiam kitam sėkmingai pakartotinai naudojamam komponentui, žaidimų varikliukui yra svarbu būti tinkamai suprojektuotam.

Taigi pagrindinis šio darbo tikslas – išanalizavus šių dienų žaidimų varikliuko kontekstą ir esančius programinės įrangos architektūrinius modelius, pateikti lankstų ir lengvai praplečiamą architektūrinį sprendimą, tinkantį žaidimų varikliukui.

Išanalizavę įvairius architektūros tipus ir jų realizavimo metodus, bei žaidimo varikliukų funkcinius reikalavimus, realizavome varikliuko prototipą, naudodami duomenimis valdomą architektūrą, kaip labiausiai tinkančią ir tenkinančią reikalavimus varikliukams.

Pasirinktos architektūros pagrindu kiekvienam varikliuko moduliui pasiūlėme detalų architektūrinį sprendimą ir jo realizavimo būdus.

The Architecture for Computer Game‘s Engine

Annotation

Game engine is a set of supporting tools and services for game development. It is a component designed for reuse in different games. Therefore it is very important for game engine to be designed properly as for any successfully used reusable component.

The main objective in this research is to present flexible and easily extensible architectural solution suitable for the game engine, based on the analysis of today’s game engine context and existing software architecture design.

During the analysis we reviewed different types of software architecture and its implementation methods, and defined functional requirements for game engine. As the result we have chosen data-driven architecture design as the most suitable for the engine development and created our game engine prototype.

For each game engine module we have provided design solution and its implementation approach referring to chosen architecture.

Raktiniai žodžiai

Žaidimų varikliukas, žaidimas, programinės įrangos architektūra, detalioji architektūra, žaidimų varikliuko architektūra, architektūros tipai, duomenimis valdoma architektūra.

Key Word

Game engine, game, software architecture, software design, game engine design, software architecture types, Data-Driven design.

Turinys

Paveikslėlių sąrašas

Lentelių sąrašas

1. Įvadas

1.1. Dokumento paskirtis

Šis dokumentas supažindina su žaidimų varikliuko projektavimo ir architektūros principais. Taip pat aprašomi įprastinės programinės įrangos projektavimo ir architektūros skirtumai ir panašumai. Dokumente plačiau apžvelgiami žaidimų varikliukų architektūros sprendimo metodai, bei įvairūs tų metodų privalumai ir trūkumai.

1.2. Pagrindimas

Kasmet vis didesnė pramogų verslo dalis tenka kompiuterinių žaidimų pramonei. Kaip ir bet kokioje rinkoje norint didint paklausą ir patenkinti vartotojus reikia didint kokybę, žaidimų atveju tai būtų žaidimo grafika, dirbtinis intelektas (DI), fizika, muzika ir t.t. Gamintojai šitoje situacijoje ieško būdų didinti kokybę stipriai nedidinant kaštų. Ir jų sprendimas būna vienareikšmis – naudoti žaidimų varikliuką: jis tiesiog realizuoja visas technologines naujoves, o gamintojams lieka jas panaudoti, taip gamintojai gali sutelkti savo dėmesį į žaidimo idėją, o ne į techninį jo realizavimą. Taip pat didelis privalumas yra tas, kad žaidimų varikliuką galima naudoti keletui žaidimų, tai leidžia dar labiau sumažinti kaštus. Toks gamintojų reikalavimas vis labiau apsunkina varikliukų gamybą, iš jų norima kuo didesnio lankstumo tuo pačiu ir kuo didesnio abstrakcijos lygio, bei kuo daugiau naujovių. Visą tai realizuoti yra tikrai nelengvas darbas.

1.3. Tikslai

Pagrindinis šio darbo tikslas – išanalizuoti šių dienų žaidimų varikliuko kontekstą bei pasiūlyti žaidimų varikliuko bendrinę architektūrą, tenkinančią šio konteksto reikalavimus. Taip pat pateikti tokios architektūros realizavimo metodus, iliustruojant juos UML diagramomis, aptarti jų privalumus ir trūkumus.

Atlikus žaidimų varikliukų architektūros analizę, sukurti žaidimų varikliuko prototipą, grįstą pasiūlyta architektūra.

2. Informacijos šaltinių apžvalga

2.1. Užsienio ir Lietuvos literatūros apžvalga

Besiplečiant kompiuterinių žaidimų rinkai vis daugiau atsirandą knygų apie kompiuterinių žaidimų kūrimą ir programavimą. Nors literatūros pasirinkimas ir didelis, tačiau daugelis knygų skirtos specifiniai žaidimų kūrimo sričiai ir tik nedaugelis knygų aprašo žaidimų varikliukų ir žaidimų architektūrinius sprendimus.

2.1.1. Užsienio literatūros apžvalga

„3D Game Engine Design“ knyga labai detaliai aprašo žaidimo varikliuko matematinį pagrindimą, tačiau joje nėra kalbama apie žaidimo varikliuko projektavimą iš programinės įrangos pusės.

„3D Game Engine Programming“ knyga praktiniais pavydžiais iliustruoja žaidimų varikliukų kūrimo ir realizavimo metodus. Šioje knygoje apžvelgiami beveik visi žaidimų varikliuko kūrimo aspektai, plačiai ir suprantamai juo aprašant. Tačiau kai kurie pasiūlyti sprendimai nėra tinkami sudėtingesnėms problemoms spręsti. Šią knygą galima pasiūlyti pradedantiems žaidimų programuotojams.

„Game Programing Gems“ serija. Ši knygų serija kiekvienais metais pasipildo nauja knyga. Šiose knygose yra didelis naudingų straipsnių rinkinys, kurie siūlo labai originalius specifinių problemų sprendimus, todėl jose nerasite vientisos žaidimų varikliuko projektavimo ir kūrimo metodikos.

2.1.2. Lietuvos literatūros apžvalga

Lietuvių kalba žaidimų varikliukų kūrimą aprašančių knygų neegzistuoja taip pat nėra ir jokių straipsniai.

Lietuvoje galima rasti svetainę http://www.gamedev.lt palaikoma kelių entuziastų, kurioje galima rasti keletą trumpų pamokėlių ir patarimų žaidimų programuotojams.

2.2. Varikliukų apžvalga

Dar vienas informacijos šaltinis yra žaidimų varikliukų išeities tekstų analizė. Egzistuoja dviejų tipų žaidimų varikliukai: atviro ir uždaro (komerciniai) kodo.

Atviro kodo varikliukai yra kuriami tam tikros žmonių bendruomenės, tai dažniausiai būna žmonės entuziastai, kurie savo laisvą laiką skiria varikliukų programavimui arba tai būna žmonės, kurie naudoja šiuos varikliukus savo projektams realizuoti ir jie padedami bendruomenės pastoviai stengiasi išplėsti varikliuko funkcionalumą.

Uždaro kodo varikliukus dažniausiai kuria komercinės įmonės, tam darbui samdydamos įvairių sričių profesionalus. Savaime suprantama jie nenori paviešinti savo žaidimų išeities kodų, taip išvaistydami savo investicijas, prarasdami pelną ir panaikindami savo konkurentabilumą. Tačiau egzistuoja du atvejai kada netgi tokie žaidimai žaidimų kūrėjams leidžia pasisemti įvairių žinių apie žaidimų ir žaidimų varikliukų kūrimą:

- Pirmasis atvejis tai kada žaidimų varikliukų kūrėjai, praėjus ne mažam laiko tarpu, ir varikliukui praradus savo konkurentabilumą, paviešina savo varikliuko išeities tekstus.

- Antrasis atvejis tai kada žaidimų kūrėjai norėdami prailginti žaidimų varikliuko gyvavimo laiką ir kartu pareklamuoti savo žaidimų varikliuką, atskleidžia žaidimo išeities tekstus ir žaidimo varikliuko SDK (angl. *Software Development Kit*) paketą.

SDK paketas leidžia kurti žaidimus ant pateikto varikliuko. Savo ruoštu žaidimų išeities tekstai tarnauja kaip pavyzdys. Šiuo atveju niekada nėra atskleidžiami žaidimų varikliukų išeities tekstai, tačiau visiškai paslėpt išeities tekstą yra neįmanoma, kadangi reikalinga sąsają su žaidimo varikliuku. Šia sąsaja dažniausiai būna bendri žaidimo ir žaidimo varikliuko antraščių failai, kurie įtraukiami į SDK paketą. Detaliau apie antraščių failus skaitykite priede *Priedas 1 Dinaminės ir statinės bibliotekos.*

2.2.1. Atvirojo kodo varikliukai

Atviro kodo varikliukai dažniausiai nėra labai aukštos kokybės, dėl savo mėgėjiškos prigimties, tačiau egzistuoja keletas išimčių.

Ogre – tai labai lankstus, universalus trimatės grafikos varikliukas, parašytas C++ kalba. Jame visos realizavimo detalės paslėptos po abstrakčiomis sąsajomis, visa varikliuko architektūra yra gerai apgalvota. Tačiau, kaip minėjau, tai yra grafikos varikliukas, o ne žaidimų varikliukas. Tai reiškia, kad šis varikliukas neturi žaidimams specializuotų fikcijų, ir kartais yra persudėtingas. Taip pat jame nėra viso kito žaidimams reikalingo funkcionalumo, tokio kaip garso, fizikos, tinklo ir kitokių bibliotekų [22].

Irrlicht – tai trimatės grafikos varikliukas parašytas C++, tačiau dar pritaikytas ir .NET platformai. Šis varikliukas, lyginant su Ogre varikliuku, yra paprastesnis ir lengviau suprantamas, tačiau nėra toks universalus [21].

Crystal Space – tai varikliukas taip pat prašytas C++ kalba. Tačiau šis varikliukas, lyginant su kitais, palaiko tik vieną vaizdavimo biblioteką – OpenGL [20].

Visi šie varikliukai turi labai platų funkcionalumą. Visi išvardinti varikliukai gali veikti keliose skirtingose operacinėse sistemose. Dažniausiai jie visi naudoja įvairias pagalbines bibliotekas. Jei pagalbinės bibliotekos nėra integruotos į patį varikliuką, jų galima rasti įvairiose varikliuko prieduose arba modifikacijose.

2.2.2. Egzistuojantys žaidimų SDK

Tai profesionalių žaidimų kūrėjų sukurtų žaidimų SDK. Į SDK dažniausiai įeina, ne tik žaidimo išeities tekstas, antraščių failai bei bibliotekos, bet ir visi reikalingi įrankiai ir dokumentacija.

CryEngine – CryTek kompanijos varikliukas pirmą kartą buvo panaudotas kuriant FarCry žaidimą. Labai aukštai vertinamas žaidimų varikliukas, turi labai platų funkcionalumą, patogius papildomus įrankius, taip pat yra labai gerai dokumentuotas. Varikliukas ir žaidimas realizuoti C++ kalba [18].

Half-Life Engine – Valve kompanijos varikliukas pirmą kartą panaudotas to paties pavadinimo žaidimui Half-Life kūrti. Tai gana senas, tačiau vis dar populiarus varikliukas, taip pat realizuotas C++ kalba [17].

Half-Life 2 Engine – taip pat Valve kompanijos varikliukas pirmą kartą panaudotas to paties pavadinimo žaidimui Half-Life 2 kūrti. Tai naujausias įmonės Valve kūrinys, realizuotas C++ ir galima sakyti tik pradėjo savo gyvenimą [17].

Quake III Engine – Id Software kompanijos produktas pirmą kartą panaudotas to paties pavadinimo žaidimui Quake III kurti. Tai taip pat gana senas varikliukas, tačiau Id Software vienintelė garsi įmonė, kuri po tam tikro laiko atskleidė savo varikliukų išeities tekstus. Quake III Engine buvo paskutinis toks varikliukas, bet tikėkimės, kad nepaskutinis. Šis varikliukas realizuotas C kalba [19].

3. Programinės įrangos architektūra

3.1. Programinės įrangos architektūros apibrėžimas

Programinės įrangos architektūra aprašo programinius modulius ir ryšius tarp jų [14]. Sudėtingėjant ir didėjant programinei įrangai reikalingi paprastesni ir lengviau suprantami vaizdavimo ir projektavimo principai [14]. Todėl pagrindinis programinės architektūros realizavimo tikslas – mažinti programinės įrangos sudėtingumą, padaryti ją lengviau valdomą ir suprantamą. Realizuoti sistemą, nesinaudojant jokia architektūra, o žvelgiant į sistemą kaip vieną visumą, yra labai nepatogu nes:

- nėra bendros sistemos suvokimo idėjos;
- kiekvienam kūrėjų komandos nariui reikia susikoncentruoti ties visa sistema, o ne tik ties atskiru jos komponentu;
- neįmanomas autonominis komandos narių darbas prie to paties projekto;
- sudėtingesnis komandos, užtikrinančios kokybę, darbas, kai iškart testuojama visa sistema, o ne atskiri moduliai.

3.2. Architektūros tipai

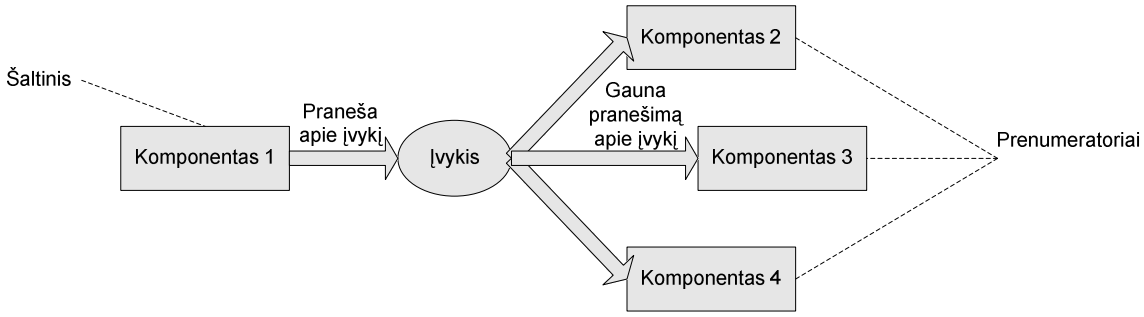
Egzistuoja įvairūs architektūros tipai. Kiekvienas jų turi savo privalumų ir trūkumų. Architektūros tipą įprastai apibūdina ryšių tarp komponentų realizavimas arba/ir komponentų grupavimo principai [14].

Pasirenkamas architektūros tipas dažniausiai priklauso nuo sprendžiamo uždavinio specifikos bei tam tikrų sąlygų tokių, kaip patikimumo, greičio, pakartotinio naudojimo, saugumo ir panašiai. Dažniausiai programinė įranga realizuojama pasirenkant ne vieną konkrečią architektūrą, o tam tikrą jų derinį. Architektūros pasirinkimą dažnai lemia ir programavimo kalbos pasirinkimas, arba, galima sakyti, ir atvirkščiai. Pavyzdžiui, būtų labai nepatogu naudoti į objektus orientuotą architektūrą C kalboje. Todėl labai svarbu tinkamai išrinkti architektūros ir programavimo kalbos derinį [14].

Toliau šiame skyriuje trumpai apžvelgiame pagrindinius plačiai naudojamus architektūros tipus.

3.2.1. Įvykių architektūra

Pastaruoju metu stipriai išpopuliarėjo architektūros tipas pagrįstas įvykiais. Komponentai ne tiesiogiai kviečia vienas kito metodus, o transliuoja įvykius visiems komponentams, kurie buvo užsiregistravę prenumeratorių sąraše. Ši veiksmų seka pavaizduota 1 pav. paveikslėlyje.

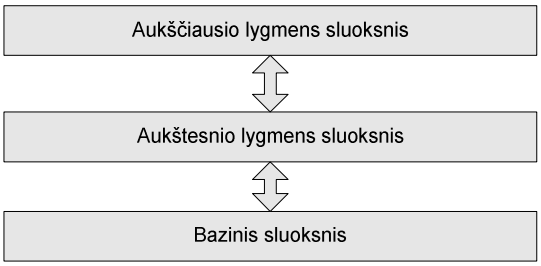


1 pav. Įvykių architektūra

Šios architektūros pagrindinis privalumas – komponentams bendraujant tarpusavyje nei vienas (įvykių generatorius), nei kitas (prenumeratorius) komponentas iš ankso gali nežinoti apie kito egzistavimą.

3.2.2. Daugiasluoksnė architektūra

Daugiasluoksnė architektūra – tai architektūra, kurios pagrindinis bruožas yra komponentų suskirstymas į tam tikrus lygmenis, kurių kiekvienas naudojasi žemesniais lygmenimis ir suteikia paslaugas aukštesniam lygmeniui. Kaip pavaizduota 2 pav. paveikslėlyje kiekviename lygmenyje bendraujama tik su dviem kitais lygmenimis, išskyrus aukščiausią ir žemiausią, kurie bendrauja tik su vienu lygmeniu. Tai leidžia aiškiai ir gana lengvai apibrėžti kiekvieno lygmens paskirtį. Šios architektūros pagrindinis privalumas yra kartu ir jos pagrindinis trūkumas, kai kurias sistemas sunku suskirstyti į atskirus ir griežtai apibrėžtus lygmenis.



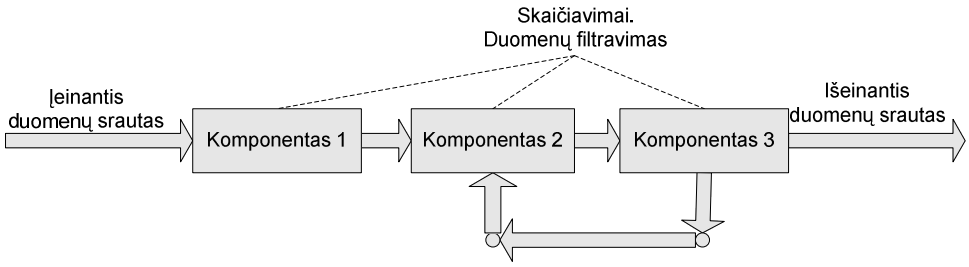
2 pav. Daugiasluoksnė architektūra

Ši architektūra dažnai naudojama įvairaus tipo verslo programose. Jose naudojama trijų lygmenų architektūra: pirmasis lygmuo – duomenų priėjimo, antrasis – verslo logikos, trečiasis – duomenų vaizdavimo. Aiškiai apibrėžtus lygmenis galima keisti kitu lygmeniu, kuris teikia tas pačias paslaugas (pvz. vaizdavimo lygmenį galima pakeisti kitu vaizdavimo lygmeniu, taip programa nesunkiai gali būti perkelta iš MS Windows aplinkos į internetinę aplinką).

Ši architektūra taip pat labai populiari įvairių bendravimo protokolų, kurie sukurti OSI pagrindu, realizavimui (pvz., FTP protokolas naudojasi žemesnio lygmens protokolo TCP/IP paslaugomis, kad atliktų savo funkcijas).

3.2.3. Duomenų srauto architektūra

Šioje architektūroje duomenys, kaip įėjimas, paduodami vienam komponentui ir jo išėjimas paduodamas kitam komponentui, kaip įėjimas. Taip sudaromi ryšiai tarp komponentų. Šie ryšiai pavaizduoti 3 pav. paveikslėlyje.



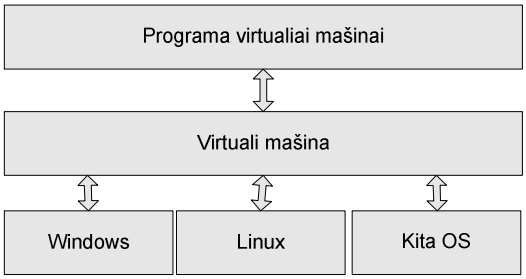
3 pav. Duomenų srauto architektūra

Kiekvienas duomenų perdavimo tipas yra griežtai apibrėžtas, todėl komponentai yra visiškai tarpusavyje nepriklausomi ir nežino, nei kas jam duomenis perduoda, nei kas tuos duomenis priima. Kiekvienas komponentas apibrėžiamas pagal tai, kokius duomenis jis priima ir atiduoda. Pagrindiniai šios architektūros privalumai:

- sistemos paprastumas, visa sistema vaizduojama kaip įėjimai/išėjimai;
- vienas komponentas gali būti lengvai pakeistas kitu;
- lengvas sistemos palaikymas, nauji komponentai gali buti lengvai pridedami ir seni pakeičiami naujais;
- lengvas lygiagretumo palaikymas, kiekvienas komponentas gali būti realizuotas atskiru komponentu.

3.2.4. Virtualios mašinos architektūra

Virtuali mašina – tai programinė įranga, izoliuojanti programas, kurias naudoja vartotojas, nuo techninės įrangos [8]. Virtualios mašinos kuriamos įvairioms operacinėms sistemoms, todėl programa, parašyta virtualiai mašinai, gali būti vykdoma skirtingose operacinėse sistemose [8]. Nenaudojant virtualios mašinos, kiekvienai operacinei sistemai reiktų parašyti atskirą programos kodą. Labiausiai žinoma virtuali mašina – Sun Microsystem Java Virtual Machine. Paveikslėlyje 4 pav. vaizduojama virtualios mašinos architektūra.



4 pav. Virtualios mašinos architektūra

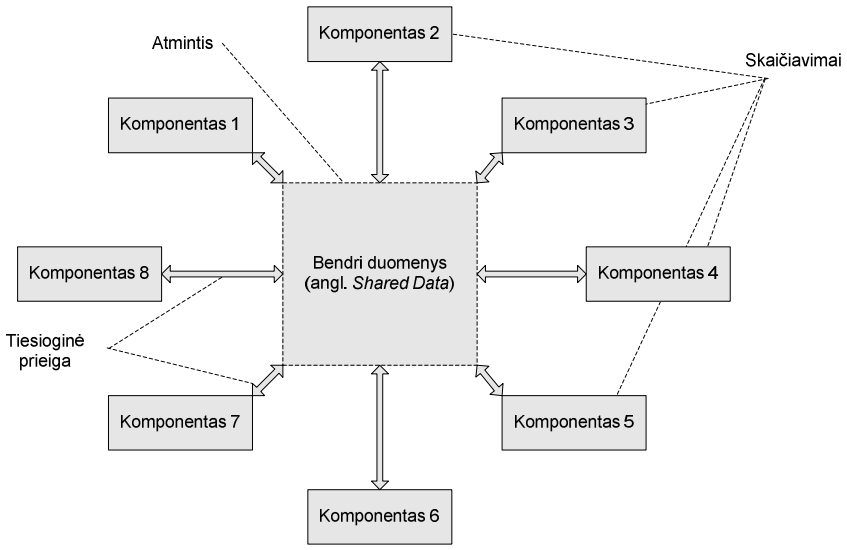
Virtuali mašina veikia konkrečioje operacinėje sistemoje ir tiesiogiai naudoja sistemos resursus. Programa, parašyta virtualiai mašinai, sąveikauja tik su virtualia mašina.

Pagrindinis šios architektūros trūkumas – lėtas veikimas, dėl atsiradusio papildomo tarpinio lygmens tarp programos ir sistemos resursų.

Didžiausias šios architektūros privalumas yra tas, kad ta pati programa, parašyta virtualiai mašinai, gali būti vykdoma įvairiuose operacinėse sistemose, nereikia kurti skirtingų programos versijų skirtingoms platformoms. Dar vienas šios architektūros privalumas yra tas, kad virtuali mašina gali turėti papildomų naudingų funkcijų, pavyzdžiui, šiukšlių surinkimą (ang. *Garbage Collection*).

3.2.5. Saugyklos architektūra

Kaip pavaizduota 5 pav. paveikslėlyje saugyklos architektūrą sudaro: centrinis duomenų saugojimo komponentas ir kiti nepriklausomi komponentai, kurie sąveikauja su centralizuota duomenų saugykla.

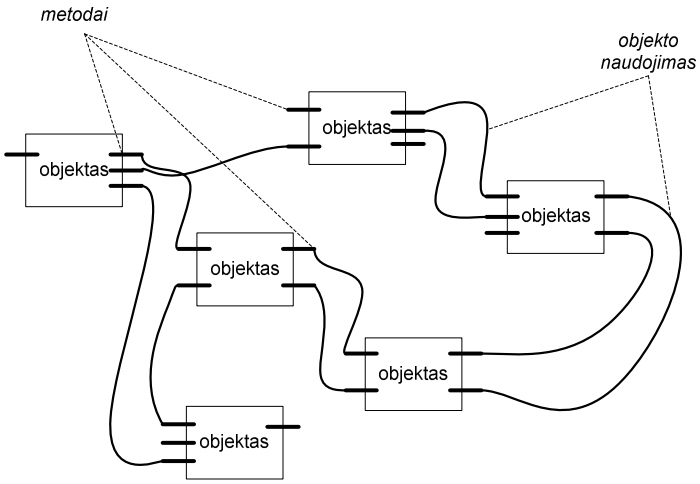


5 pav. Saugyklos architektūra

Komponentai atlieka tam tikrus duomenų apdorojimus ir savyje duomenų nesaugo. Pagrindinis saugyklos architektūros privalumai yra tai jog visi duomenys yra valdomi per vieną komponentą, tai labai palengvina duomenų vientisumo ir teisingumo palaikymą.

3.2.6. Į objektus orientuota architektūra

Į objektus orientuotoje architektūroje duomenys ir operacijos, atliekamos su tais duomenimis, yra saugomos abstrakčiame duomenų tipe arba objekte [10]. Šios architektūros komponentai yra objektai, abstraktaus tipo egzemplioriai. Kaip pavaizduota 6 pav. paveikslėlyje objektai tarpusavyje sąveikauja, kviesdami vienas kito procedūras ir funkcijas.



6 pav. Į objektus orientuota architektūra

Kiekvienas objektas yra atsakingas už savo vidinės būsenos vientisumo saugojimą, šios vidinės būsenos kiti objektai nemato ir tiesiogiai keisti negali, todėl vieno objekto vidinį realizavimą galima keisti kitu, nepakeitus pastarojo klientų [10]. Galimybė sujungti duomenis ir metodus, kurie jais manipuliuoja, leidžia projektuotojams spręsti problemas, jas išreiškiant per objektus ir jų tarpusavio sąveika. Pagrindinis šios architektūros trūkumas yra tas, kad kiekvienas objektas iš anksto turi žinoti apie kitus objektus, kuriais jis norės naudotis [10].

3.3. Detali architektūra (angl. design)

Detali programos architektūra – tai žemesnio ir kartu žemiausio lygio architektūros aprašymas. Šiame žingsnyje yra nusileidžiama iki lygio, kuriame UML klasių diagramomis aprašomi architektūriniai sprendimai.

3.3.1. Į objektus orientuota detali architektūra (angl. *object oriented design*)

Į objektus orientuota (OO) **detalios** architektūros specifikacija yra tokia pati, kaip ir į objektus orientuotos architektūros (detaliau skaitykite šio dokumento skyriuje *3.2.6 Į objektus orientuota archtektūra*), tik žemesnio lygio. Šios detalios architektūros pagrindinis duomenų tipas yra klasė. Pagrindinės metodo sąvokos yra šios: polimorfizmas, paveldėjimas ir apvilkimas (angl. *incapsulation*).

Kartais projektuotojai susiduria su panašiomis problemomis, dirbdami skirtingose dalykinėse srityse. Šioms problemoms dažnai egzistuoja bendri architektūriniai sprendimai, šie sprendimai vadinami projektavimo šablonais (angl. *design patterns*) [15]. Dar kitaip projektavimo šablonai – bendrinis sprendimas dažnai pasitaikantiems uždaviniams spręsti. Projektavimo šablonas aprašomas UML projektavimo kalba. Pagrindinis projektavimo šablonų privalumas – iš anksto žinomos jų silpnosios ir stipriosios vietos. Dažnai projektuotojui sprendžiant įvairius uždavinius yra suku įžvelgti, atpažinti problemos sprendimą ir panaudoti projektavimo šablonus [15].

3.3.2. Duomenimis valdoma detali architektūra (angl. *data driven design*)

Duomenimis valdomos detalios architektūros pagrindinis principas – aiškiai atskirti programos duomenis nuo programos kodo, kuris manipuliuoja šiais duomenimis [9]. Pagrindiniai į objektus orientuotos ir duomenimis valdomos detalių architektūrų skirtumai yra šie [5]:

- Duomenimis valdomojoje detalioje architektūroje duomenys nenaudojami vien tik kokio nors objekto būsenai saugoti, o nusako programos vykdymo eigą [9].
- OO detalioje architektūroje pagrindinis tikslas yra duomenų apvilkimas (paslėpimas). Duomenų valdomoje architektūroje pagrindinis tikslas – kurti kuo universalesnį ir nepriklausomą kodą [10].

Ir viena ir kita detalios architektūros gali būti naudojamos tame pačiame projekte, nes jos sprendžia skirtingus uždavinius.

4. Žaidimų sistemos architektūra

4.1. Dabartinė padėtis

Žaidimai ir įprastinė programinė įranga vystėsi skirtingai. Įprastinė programinė įranga iš didžiulių centralizuotų valdiklių pamažu persikėlė ir į mūsų kasdieninius asmeninius kompiuterius. Žaidimai, savo ruoštu, visą laiką buvo kuriami mažoms sistemoms ir tik vėliau buvo pradėti kurti asmeniniams kompiuteriams. Kadangi žaidimai visą laiką stengdavosi iš sistemos, stipriai apribotos resursais, gauti maksimalų veikimo greitį, bet kokie architektūriniai sprendimai, lengvinantys žaidimo realizavimą, plėtimą, palaikymą ar testavimą, būdavo atmetami dėl žaidimo greičio stabdymo. Iš tiesų, seniau žaidimai nebūdavo perdaug sudėtingi ir neturėdavo galimybių būti papildomi įvairiomis funkcijomis. Todėl šių savybių nebuvimas nebuvo didelė problema [13].

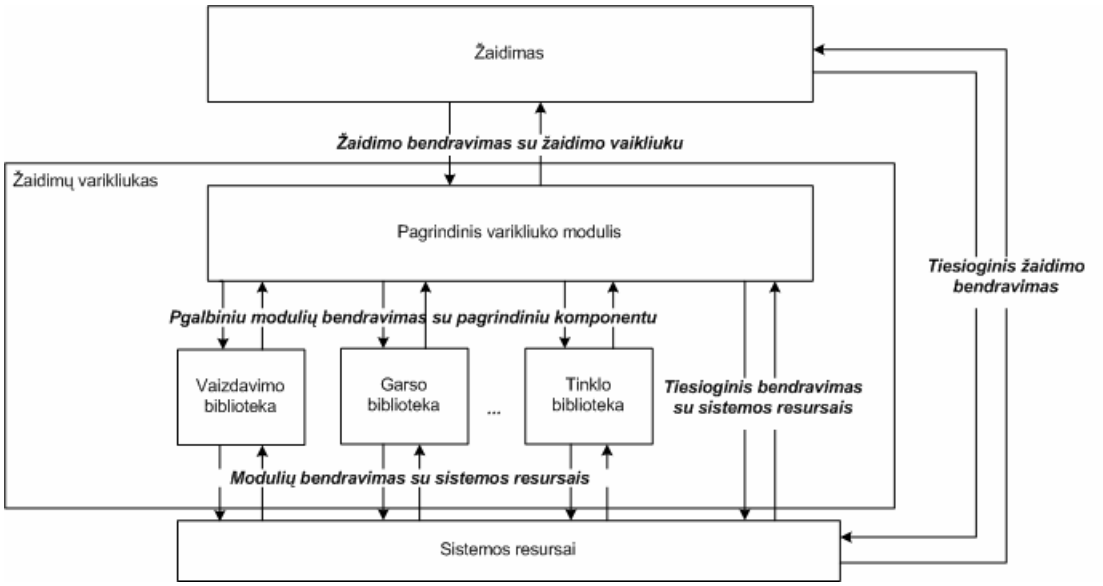
Apibendrinus, galima teigti, kad žaidimų reikalaujamas greitis – tai viena iš priežasčių, kodėl šiuo metu labai skiriasi žaidimų ir įprastinės programinės įrangos projektavimo ir architektūros lygiai [13].

4.2. Bendrinė architektūra

Egzistuoja daug įvairių žaidimų varikliukų, vieni labiau ištobulinti ir kainuoja brangiai, kiti paprastesni ir kartais būna visiškai nemokami. Tačiau ir vieni ir kiti turi panašią bazinę architektūrą, kuri pavaizduota 7 pav.

Architektūros schemoje pavaizduoti žaidimų sistemos moduliai ir ryšiai tarp jų. Šioje schemoje galime išskirti tris pagrindinius modulius:

- žaidimas,
- žaidimų varikliukas,
- sistemos resursai.



7 pav. Žaidimo sistemos bendrinė architektūra

Šie moduliai sąveikauja taip:

- Žaidimo sąveika su žaidimų varikliuku – šis ryšys atskleidžia visą varikliuko funkcionalumą kuriamiems žaidimams, todėl jis turi būti gerai ir protingai suprojektuotas.

- Pagalbinių modulių sąveika su pagrindiniu varikliuko komponentu – šis ryšys yra projektuojamas vidinėje žaidimo architektūroje, todėl žaidimų kūrėjams jis visiškai nesvarbus, tačiau realizuojant žaidimų varikliuką, jis turi būti teisingai įvertintas.

- Tarpusavio pagalbinių modulių sąveika – šis ryšys taip pat nematomas žaidėjų kūrėjams, tačiau jis taip pat labai svarbus paties varikliuko organizavimui ir dažnai būna gana sudėtingas.

- Pagrindinio modulio sąveika su sistemos resursais – šis ryšys dažniausiai nepageidaujamas, jis pažeidžia resursų apvilkimo (angl. *incapsulation*) principą. Tai sukelia resurso naudojimo nevientisumą visame žaidimo vaikiuke, kas savo ruoštu sukelia žaidimo perkėlimo problemas žaidimus pritaikant kitoms sistemoms.

- Tiesioginė žaidimo sąveika su sistemos resursais – šis ryšys yra nepageidaujamas ir visais būdais jo reiktų vengti. Jis atsiranda tada, kai

žaidimų varikliukas neužtikrina žaidimui reikalingo funkcionalumo rinkinio, todėl žaidimui tenka pačiam realizuoti ir vykdyti kreipinius į sistemos resursus. Šis sprendimas sukelia vieną didelę problemą – varikliuko funkcionalumas išsibarstomas žaidime ir varikliuke. Žaidimo perkėlimas nuo vienos sistemos ant kitos sukelia papildomas problemas. Paprasčiausias problemos sprendimas būtų trūkstamo funkcionalumo praplėtimas žaidimų varikliuke.

Analizuojant žaidimų varikliuką kaip atskira sistemą, joje išsiskiria du ryšių tipai, per kuruos žaidimų varikliukas sąveikauja su išore:

- pagrindinių modulių sąveika su sistemos resursais,
- žaidimo sąveika su žaidimų varikliuku.

4.2.1. Žaidimas

Šis modulis yra atsakingas už visą žaidimo logiką, jis sprendžia kokios logikos taisyklės bus realizuotos žaidime, ar tai bus sporto stimuliatorius, ar tai bus realaus laiko strateginis žaidimas ir panašiai. Žaidimas fiziškai būna realizuojamas kaip vykdomoji rinkmena (angl. *file*) arba kaip viena ar kelios dinaminė bibliotekos.

4.2.2. Sistemos resursai

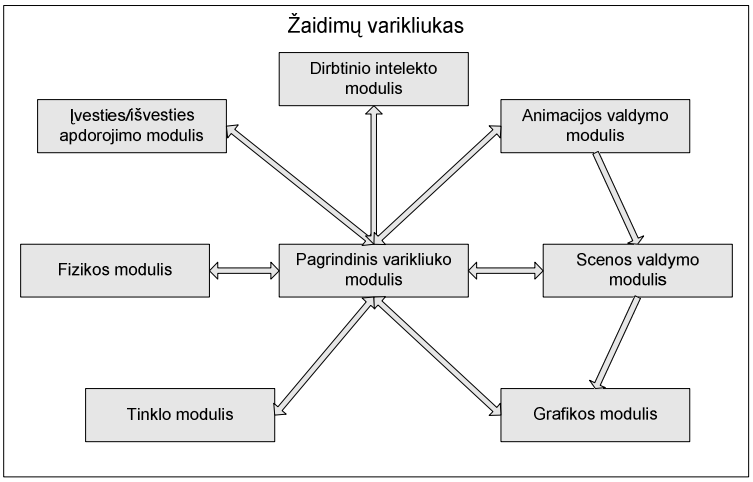
Sistemos resursai – tai techninė įranga (angl. *hardware*), kurios pagalba žaidimas gali vykdyti savo kodą ir sąveikauti su žaidėju per įvesties/išvesties įtaisus. Įprastai žaidimų variklių programuotojams nereikia tiesiogiai programuoti techninės įrangos, tai dažniausiai būna atliekama pasinaudojus įvairiomis pagalbinėmis statinėmis ir/arba dinaminėmis bibliotekomis. Jos leidžia supaprastinti programuotojų užduotis, žvelgiant į techninės įrangos suteikiamas paslaugas (angl. *services*) kaip į programines funkcijas. Šios bibliotekos dažniausiai būna realizuotos operacinės sistemos gamintojų kaip programų kūrimo sąsaja (angl. *API - Application Programming Interface*), arba aparatūros gamintojų kaip pagalbinės bibliotekos, arba laisvai platinamos trečiųjų šalių. Direct3D ir OpenGL yra tarpusavyje konkuruojančios, pačios žymiausios 3D grafikos programų kūrimo sąsajos.

4.2.3. Žaidimų varikliukas

Žaidimų varikliuko modulio pagrindinis tikslas – suteikti žaidimų kūrėjams priemones kurti žaidimus, nesigilinant į technines jų realizavimo detales. Žaidimų varikliukas, naudodamas abstrakčias sąsajas, leidžia atsiriboti nuo konkrečios techninės realizacijos, žemo lygio bibliotekų. Toks atsiribojimas įgalina žaidimo perkėlimą nuo vienos techninės platformos ant kitos, nekeičiant žaidimo kodo. Šį privalumą labai sėkmingai išnaudoja žaidimų kūrėjai, kurdami ir išleisdami žaidimus iškart kelioms platformoms. Kitas labai didelis žaidimų variklių privalumas yra tas, jog varikliukas žaidimui suteikia funkcijas, kurios yra perdaug specifines, kad būtų realizuojamos žemo lygio bibliotekose arba apskritai realizuotos kokioje nors bibliotekoje.

Projektuojant ir realizuojant žaidimą pagrindinis funkcinis reikalavimas yra veikimo greitis. Varikliukas privalo veikti kiek įmanoma greitai. Šio reikalavimo netenkinimas žaidimų varikliuką padarytų bereikalingu ir visiškai nepritaikomu. Kadangi šis reikalavimas taikomas žaidimų varikliukui kaip visam moduliui, tai, savaime suprantama, kad jis taikomas ir kiekvienam žaidimų moduliui atskirai. Kartais architektūriškai teisingus sprendimus tenka keisti ne tokiais tinkamais vien dėl greitaveikos reikalavimų.

Pats žaidimų varikliukas yra persudėtinga struktūra, kad į jį galima būtų žiūrėti kaip į vientisą modulį, todėl jis dar būna skaldomas ir aprašomas smulkesniais moduliais ir jų sąryšiais. Toks pagrindinių modulių rinkinys vaizduojamas paveikslėlyje 8 pav.



8 pav. Žaidimo varikliuko pagrindinai moduliai ir ryšiai tarp jų

Žaidimų varikliuko modulių skaičius priklauso nuo varikliuko funkcionalumo, taip pat egzistuoja tam tikras modulių rinkinys be kurių neįmanomas bet koks žaidimas.

5. Žaidimų varikliuko architektūra

5.1. Aukščiausio lygmens architektūra

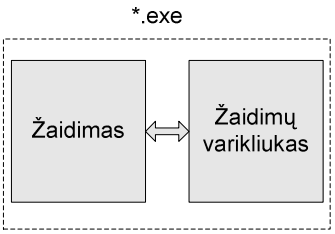
Pirmasis architektūrinis sprendimas, kurį reikia atlikti projektuojant žaidimą – pasirinkti tinkamą žaidimo ir žaidimo varikliuko modulių fizinį realizavimą ir tarpusavio bendravimo metodą, tai yra, pasirinkti aukščiausio lygio architektūrą. Egzistuoja keli fizinio modulio realizavimo variantai:

- fiziškai vientisa architektūra tai kada žaidimas realizuojamas kaip vykdomasis failas, o varikliukas kaip statinė biblioteka;
- aktyvaus žaidimo architektūra tai kada žaidimas realizuojamas kaip vykdomasis failas, o varikliukas kaip dinaminė biblioteka;
- aktyvaus varikliuko architektūra tai kada žaidimas realizuojamas kaip dinaminė biblioteka, o varikliukas kaip vykdomasis failas.

Detaliau apie statines ir dinamines bibliotekas skaitykite priede *Priedas 1. Dinaminės ir statinės bibliotekos*.

5.1.1. Fiziškai vientisa architektūra

Fiziškai vientisos architektūros (žaidimas realizuojamas kaip vykdomasis failas, o varikliukas kaip statinė biblioteka) pasirinkimas yra pats paprasčiausias, tačiau kartu jis yra ir neuniversaliausias. Kaip pavaizduota paveikslėlyje 9 pav., žaidimas susideda tik iš vienos vykdomosios rinkmenos, tokio žaidimo platinimas yra labai paprastas.



9 pav. Fiziškai vientisa architektūra

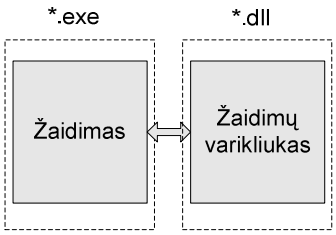
Kūrimo metu visas išeities tekstas apjungiamas į galutinę programą, tai panaikina klaidas, susijusias su dinaminių bibliotekų užkrovimu, vykdymo metu.

Šio metodo pagrindinis trūkumas yra tas, kad žaidimo moduliai nėra suskaldyti fiziškai. Šis trūkumas apsunkina bendros sistemos suvokimą, modulių ir ryšių tarp jų identifikavimą, padaro beveik neįmanomą modulinį testavimą. Programos atnaujinimo metu turi būti atnaujinta ir vykdomoji rinkmena, nors gal didelė dalis kodo nėra pasikeitus.

Kita šio metodo atmaina – nenaudoti jokių bibliotekų, o visą išeities tekstą kompiliuoti vieną kartą iškart į galutinę vykdomąją rinkmeną. Šio metodo pagrindinis trūkumas – pailgėjęs kompiliavimo laikas. Dėl išvardintų trūkumų šis metodas labiausiai tinka realizuojant mažus žaidimus.

5.1.2. Aktyvaus žaidimo architektūra

Aktyvaus žaidimo architektūra (žaidimas realizuojamas kaip vykdomoji rinkmena, o varikliukas kaip dinaminė biblioteka) metodas dažniausiai naudojamas tada, kai žaidimų varikliukas kuriamas kaip visiškai nepriklausomas modulis. Tokiu atveju žaidimas dažniausiai pats užtikrina programos vykdymo eigą. Kitaip tariant, žaidimas kontroliuoja programos tėkmę (yra aktyvus) ir valdo varikliuką. Aktyvaus žaidimo architektūros schema vaizduojama paveikslėlyje 7 pav.



10 pav. Aktyvaus žaidimo architektūra

Aktyvaus žaidimo architektūros realizavimas pavaizduotas lentelėje 1 lentelė. Joje matoma, kad žaidimas kaskart kviečia *Run* metodą, kuris atlieka žaidimo varikliuko būsenos atnaujinimą pagal pasikeitusią žaidimo logiką.

1 lentelė. Aktyvaus žaidimo realizavimo išeities tekstas

Žaidimo išeities tekstas	Žaidimų varikliuko išeities tekstas

`int main()`
`{`
`  // inicializuojamas core varikliukas`
`  <...>`
`  while(core->Run())`
`  {`
`  // atliekama įvairi zaidimo logika`
`  }`
`  return 0;`
`}`
	`class Core`
	`{`
	`  <...>`
	`public:`
	`  int Run()`
	`  {`
	`  //varikliuko atnaujinimo logika`
	`  };`
	`};`

Nors šioje architektūroje žaidimas ir yra aktyvus, tačiau jis gali visą valdymą perduoti varikliukui. Tai reiškia, kad pagrindinis programos ciklas bus vykdomas varikliuke, o ne žaidime.

Kaip pavaizduota lentelėje 2 lentelė., pasitelkiant stebėtojo (angl. *Observer*) projektavimo šabloną, žaidimas yra realizuotas paveldint *Lisiner* klasę ir realizuojant *Update* metodą, kuriame vykdomą žaidimo logika. Vėliau sukuriamas pastarosios klasės objektas ir kaip klausytojas prikabinamas prie klasės *Core* (žaidimo varikliuko) objekto *core* kaip klausytojas. Vėliau varikliukas jau atsakingas už žaidimo atnaujinimą.

2 lentelė. Aktyvaus žaidimo valdymo perdavimas varikliukui

Žaidimo išeities tekstas	Žaidimų varikliuko išeities tekstas

`class Games : public Lisiner`
`{`
`public:`
`  int Update()`
`  {`
`  //atliekama įvairi žaidimo logika`
`  }`
`};`
 	
`int main()`
`{`
`// inicializuojamas core varikliukas`
`  <...>`
`  Game *game = new Game();`
`  core->Attach(game);`
`  core->Run();`
 	
`}`
	`class Lisiner`
	`{`
	`public:`
	`  virtual int Update() = 0;`
	`}`
	
	`class Core`
	`{`
	`  <...>`
	
	`public :`
	`  while( ! Notify() )`
	`  {`
	`// varikliuko atnaujinimo logika`
	`  <...>`
	`  }`
	`  void Attach(Lisiner *lisiner){/*prideda klausytoja*/}`
	`  void Detach(Lisiner *lisiner){/*išima klausytoją*/}`
	`private:`
	`  int Notify(){/*praneša visiems kalusytojams iškviečian Update()*/}`
	`}`

Ryšys žaidimų varikliuko ir žaidimo arba, kitaip tariant, varikliuko sukūrimas iškviečiant jį iš dinaminės bibliotekos realizuojamas dviem būdais:

- Eksportuojant visą varikliuko klasę panaudojant importo statines bibliotekas. Šis metodas patogesnis jį naudojant tereikia nurodyti importavimo biblioteką.

```
Core *core = new Core();
```

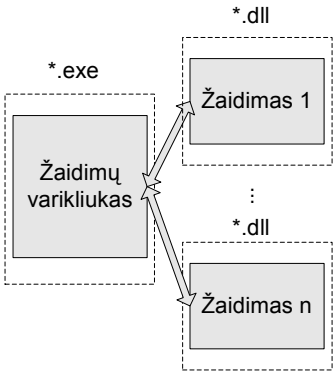
- Neeksportuojat varikliuko klasės, o eksportuojant funkciją, kuri naudojama žaidimo varikliuko sukūrimui. Šis metodas leidžia valdyti klasės sukūrimą, tačiau žaidimo kviečiami metodai privalo būti virtualūs.

```
Core *core = CreateCore();
```

Detaliau apie statines ir dinamines bibliotekas skaitykite priede *Priedas 1. Dinaminės ir statinės bibliotekos.*

5.1.3. Aktyvaus varikliuko architektūra

Aktyvaus varikliuko metodas (žaidimas realizuojamas kaip dinaminė biblioteka, o varikliukas kaip vykdomoji rinkmena) yra atvirkštinis antrajam. Šiame metode visą žaidimo valdymą atlieka žaidimo varikliukas. Aktyvaus varikliuko architektūra pavaizduota 11 pav.



11 pav. Aktyvaus varikliuko architektūra

Pasinaudojus tuo, kad žaidimų Varikliukas atlieka visą valdymą, žaidėjų kūrėjai realizavo žaidimų modifikavimo sistemą (angl. *mod*). Ši sistema leidžia žaidimų vartotojams ne tik žaisti, bet ir kurti žaidimus, modifikuojant esamą žaidimą. Žaidimų varikliukas atlieka modifikuotų žaidimų valdymą ir užkrovimą. Žaidimų modifikacijos stipriai prailgina žaidimų gyvavimą ir populiarumą, visos žaidimų modifikacijos privalomos platinti nemokamai. Modifikacijas palaikančių žaidimų kūrimo ir platinimo strategija būna tokia:

- išleidžiamas žaidimas, naudojantis modifikuotus žaidimus palaikantį varikliuką;
- prigesus žaidimo populiarumui, išleidžiamas žaidimo varikliuko SDK (angl. *Software Development Kit*) kartu su žaidimo išeities tekstais, taip įkvėpiant žaidimui naujam gyvenimui.

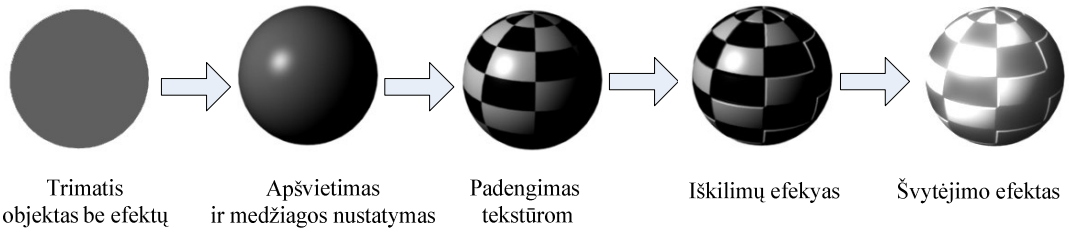
Šioje situacijoje žaidėjas tik išlošia, nes jis gauną, ne tik originalų žaidimą, už kurį sumokėjo, bet ir nemokamai visas šio žaidimo modifikacijas. Pats populiariausias, plačiai žinomas yra „Counter-Strike“ – žaidimo „Half-Life“ modifikacija.

5.2. Žaidimų varikliuko kontekstas

Šiame skyriuje apžvelgiami pagrindiniai žaidimų varikliuko moduliai ir jų funkcionalumas.

5.2.1. Vaizdavimo modulis

Kiekvienas žaidimas reikalauja vienokio ar kitokio vaizdavimo. Vaizdo apdorojimo modulis yra vienas iš pagrindinių žaidimų varikliukų modulių. Jis dažniausiai aukštame lygyje apvelka (angl. *incapsulate*) žemesnio lygio grafines bibliotekas.



12 pav. Trimačio objekto vaizdavimas

Visas šio modulio darbas susiveda į kelis trimatės grafikos vaizdavimo veiksmus, kurie pavaizduoti 12 pav. paveikslėlyje:

- Trimačius objektus (angl. *mesh*) aprašančios struktūros supratimas ir manipuliavimas. Varikliukas privalo efektyviai ir greitai vaizduoti ekrane pateiktas viršūnes, leidžiant pasirinki kameros poziciją, orientaciją, vaizdo piešimo vietą ekrane ir daugelį kitų dalykų.
- Objekto apšvietimas – tai objekto spalvos intensyvumo keitimas, atsižvelgiantį šviesos šaltinio tipą, atstumą iki jo, šviesos kritimo kampą ir objekto paviršiaus fizikines savybes. Kartais šviesa naudojama ir kitiems tikslams, nei tik spalvos intensyvumui nustatyti, pavyzdžiui, jos pagalba galima simuliuoti objekto iškilimus. Kitas labai svarbus apšvietimo rezultatas yra šešėliai. Šešėliai leidžia lengviau suvokti pasaulio daiktų išdėstymą.
- Objekto padengimas tekstūromis. Objektai dažniausiai nebūna nuspalvinti vieną vientisa spalva, o būna padengti tam tikrais raštais. Šiuos raštus trimačiame pasaulyje atitinka įvairios tekstūros. Tekstūros labai dažnai

naudojamos ir kitiems tikslams, pavyzdžiui, simuliuoti pasaulio apšvietimą, iškilimus.

- Galutinis vaizdavimas – tai paskutinis etapas, kurio metu galutiniai vaizdo taškai yra išvedami į ekraną. Šiuo etapo metu įmanomi įvairūs globalaus vaizdo pakeitimai, tokie kaip švytėjimo efektas.

5.2.2. Scenos valdymas

Scenos valdymo modulis glaudžiais susijęs su vaizdavimo moduliu, vaizdavimo modulis užtikrina tik trikampių piešimą ekrane, tuo tarpu scenos valdymo modulio užduotis efektyviai išnaudoti vaizdavimo modulį. Neefektyvus grafinio modulio naudojimas gali privesti prie didelio spartos praradimo

Pagrindinis vaizdo apdorojimo procesorius - GPU (angl. *Graphics Processing Unit*). Šiuo metu šie procesoriai nuolat tobulėja ir gali apdoroti daugybę trikampių (pagrindinė trimačio pasaulio figūra), tačiau šį kiekį netiesiogiai galima padidinti, naudojant scenos valdymo mechanizmus.

Pirma šio mechanizmo idėja – išmesti iš vaizdavimo proceso visas figūras, kurios nėra matomos konkrečiu momentu, taip fiktyviai didinamas trikampių, kuriuos reikia apdoroti, kiekis. Šios figūros – tai dažniausiai figūros, kurios yra už žaidėjo žvilgsnio ribų arba uždengtos kitom nepermatomomis figūromis.

Antras būdas kaip padidinti vaizduojamų trikampių kiekį yra naudoti vaizdavimo modulio metodus tokia tvarka, kuri atsižvelgia į vidines vaizdo spartintuvų konstrukcijų savybe. Tokiu būdų galima efektyviai išnaudoti vidines vaizdo spartintuvo atmintis ir GPU. GPU yra tarsi konvejeris, kuris pagal nustatytus išorinius parametrus, vieną po kito apdoroja jam pateiktas viršūnes. Šis konvejeris greičiausiai dirba tada, kai yra rečiausiai keičiami išoriniai parametrai. Todėl vaizduojamus trimačius objektus yra labai svarbu juos vaizduoti tokia tvarka, kad šie išoriniai parametrai kuo rečiau keistųsi [7]. Šie išoriniai parametrai gali būti: tekstūros, šviesos nustatymai, paviršiau savybių nustatymai ir t.t.

5.2.3. Animacija

Pasaulio objektai trimačiame žaidime sudaryti iš daugybės trikampių [16]. Šie trikampiai dažnai turi būti transformuojami savo lokalioje kartais ir globalioje koordinačių sistemoje,

pavyzdžiui, žmogaus modelis pakelia ranką arba, kitaip tariant, jis pakeičia savo rankos padėtį centro (pilvo) arba lokalios koordinačių sistemos atžvilgiu. Objekto centras ir lokalios koordinačių sistemos centras nėra visiškai tas pats, tačiau dažniausiai jie sutampa. Tokias transformacijas turi užtikrinti animacijos modelis [16].

Anksčiau dėl procesoriaus lėtumo buvo plačiai naudojama kadrų (angl. *keyframe*) animacija. Jos principas – kiekvienu laiko momentu fiksuoti kiekvienos objekto viršūnės padėtį. Vėliau, atkuriant animaciją, kiekvieno naujo kadro metu yra užkraunama atitinkama nauja animacija, tai yra užkraunamos visos viršūnės iš naujo.

Šiuo metu vis labiau populiarėja kitas animacijos tipas – skeleto (angl. *skeleton*) animacija. Šios animacijos metu kiekviena objekto viršūnė yra susieta su tam tikru „kaulu“ (angl. *bone*), kurie tarpusavyje sujungti „sąnariais“ (angl. *joint*). Kaulai ir sąnariai tai vidinė objekto struktūra panaši į žmogaus. Taigi animuojant kaulus, tai yra juos sukinėjant per sąnarius, kartu animuojamos ir susietos objekto viršūnės [16].

Šio metodo pagrindinis privalumas yra optimalus atminties išnaudojimas ir universalumas. Atmintis yra sutaupoma kiekvienam naujam kadrui nesaugant visų transformuotų viršūnių, o saugant tik vieną viršūnių rinkinį, kurį transformuojant atitinkamo kadro kaulų transformacijos matricomis, gaunamos reikalingo kadro transformuotos viršūnės [16]. Kaulų transformacijos informacija kartu su vienu viršūnių rinkiniu užima žymiai mažiau duomenų, nei saugant kiekvieno kadro viršūnių rinkinį.

Kaulų transformaciją galima apjungti ir naudoti kelias animacijas vienu metu, pavyzdžiui, turint veikėjo šaudymo, bėgimo ir ėjimo animacijas, galima realizuoti veikėjo bėgimo ir šaudymo, bei ėjimo ir šaudymo animacijas. Kaulų transformacijos leidžia naudoti sklandaus perėjimo algoritmus iš vienos animacijos į kitą, kai jų primas ir paskutinis kadras nesutampa.

Kaulų animacija atlieka modeliuotojas veikėjų animavimo metu. Vėliau ši kaulų animacija išsaugoma transformacijų pavidalu. Žaidimo metu šios transformacijos naudojamos transformuojant objektų viršūnes.

Kaulų animaciją apibūdina dvi charakteristikos [16]:

- kaip kalų animacija veikia objekto viršūnes;
- kaip atliekami pati kaulų animacija, norint pasiekti tam tikrą tikslą.

Egzistuoja dvi pirmosios charakteristikos atmainos:

- Pirmojo metodo atveju viena viršūnė yra priskiriama tik prie vieno kaulo (angl. *stiching*).
- Antrojo metodo atveju viena viršūnė gali būti priskiriama prie kelių kaulų, tam naudojami koeficientai, kurie nusako kiek viršūnė bus paveikta tam tikro kaulo transformacijos (angl. *skinning*). Norint, kad animaciją atrodytų natūraliai ir jokios viršūnės neliktų kaboti erdvėje, svarbu, kad vienos viršūnės bendra koeficientų suma būtų lygi vienetui.

Pagrindinis antrojo metodo privalumas yra tas, jog jo pagalba galima realizuoti tolygesnę animaciją, tačiau šis metodas reikalauja didesnių laiko sąnaudų [3].

Antroji charakteristika taip pat turi dvi atmainas:

- Tiesioginė (angl.*forward*) animacija. Šiuo atveju, pavyzdžiui, norint, kad veikėjas pasiektų puodelį stovintį ant stalo, animatoriui reikia tam tikru būdu pasukti rankos, o gal ir nugaros kaulus.
- Atvirkštinė (angl. *inverse*) animacija. Šiuo atveju animatoriui tereikia nurodyti, kad veikėjas ranka nori pasiekti puodelį, o visų kitų kaulų transformacijos atliekamos automatiškai.

Pagrindinis antrojo metodo privalumas yra tas, jog šios animacijos pagalba galima atlikti įvairiausius judesius, prieš tai juos nesuanimavus, tačiau šį metodą gana sunku suvaldyti ir priversit atlikti pageidaujamus kaulų judesius. Pirmojo metodo pagrindinis privalumas yra tas, jog jo pagalba galima išgauti labai subtilią animaciją, tačiau jis reikalauja didesnio animatoriaus darbo.

5.2.4. Fizikos imitavimas

Šiuo metu vis greičiau besivystant kompiuteriniai grafikai, žaidėjai pradeda reikalauti ne tik gražaus pasaulio vaizdo, bet ir natūraliai sąveikauti su juo [11]. Šiam tikslui pasiekti reikia trimačio pasaulio objektams suteikti fizikines savybes, tokias kaip masė, pagretis ir panašiai, bei priversti juos raguotį į tarpusavio sąveiką. Šio modulio sprendžiamas problemas galima suskaidyti į du smulkesnius uždavinius: susidūrimo aptikimo ir atsako generavimo [11].

Susidūrimo aptikimo sistemos pagrindinė užduotis – laiku, tiksliai ir greitai aptikti objektų tarpusavio susidūrimus [12].

Atsako sistema pagal kūno judėjimo parametrus, įvertinus kūno tarpusavio susidūrimus, privalo suskaičiuoti naują kūno padėtį [12].

Dažnai fizikos sistemos turi įvairius papildomus fizikinius objektus tokius kaip spyruokles, slopintuvus, apribojimų objektus. Šie papildomi fizikiniai objektai leidžia geriau simuliuoti realų pasaulį.

5.2.5. Resursų valdymo modulis

Resursų valdymo modulis – tai pagalbinis žaidimų varikliuko modulis, kuris stipriai susijęs su kitais moduliai. Šio modulio pagrindinis tikslas – sekti ir valdyti žaidime naudojamus resursus. Resursai – tai vienokio ar kitokio pobūdžio duomenys, kurie įprastai saugomi kietajame diske. Tačiau žaidimo vykdymo metu šie resursai turi būti saugomi greitai prieinamoje atmintyje, ši atmintis būna dviejų tipų: operatyvioji atmintis, skirta bet kokio tipo duomenims saugoti, ir vaizdo bei AGP (angl. *Accelerated Grahics Port*) spartinančioji atmintis, skirtos vaizdo informacijai saugoti.

AGP spartinančioji atmintis – tai operatyvinės atminties dalis, kuri būna rezervuota vaizdo operacijoms atlikti. Vaizdo atmintis - tai atmintis esanti vaizdo plokštėje. Tačiau šios atmintys būna ribotos, todėl sprendimas iš anksto užkrauti visus reikalingus resursus yra netinkamas. Šiai problemai spręsti ir yra naudojamas resursų valdymo modulis. Jo pagrindinė užduotis – neviršyti nurodyto atminties biudžeto, reikalingiausius resursus laikant greitai prieinamoje atmintyje, nereikalingus (pasenusius) resursus šalinant iš atminties. Visi resursai anksčiau ar vėliau turi būti apdorojami (panaudojami). Resursų vartotojas – tai dažniausiai kokio nors tipo procesorius, todėl efektyviausia yra laikyti resursus „arčiausiai“ prie juos apdorojančių procesorių.

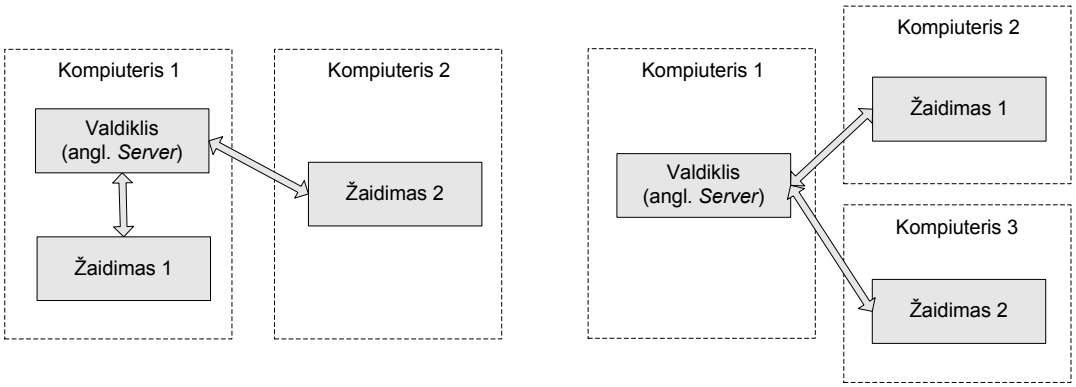
Žaidimų sistemoje mažų mažiausia egzistuoja du procesoriai: CPU (angl. *central processing unit*) ir GPU (angl. *graphic processing unit*). GPU greičiausiai prieinama atmintis – yra vaizdo atmintis. CPU greičiausiai prieinama atmintis yra operatyvioji atmintis. AGP spartinančioji atmintis naudojama kai CPU ir GPU reikalingas priėjimas prie tų pačių resursų, tai atmintis, prie kurios abu procesoriai gali prieiti pakankamai greitai, tačiau ne taip greitai kaip prie savo atminties.

5.2.6. Tinklo sistema

Greitėjant ir didėjant interneto tinklui vis daugiau žmonių turi galimybę juo naudotis. Savaime suprantama tie žmonės taip pat nori išnaudoti interneto galimybes žaidžiant su kitais žaidėjais internete. Kaip taisyklė populiariausi žaidimai yra tie, kurie sėkmingiausiai išnaudoja žaidimo keliese galimybes, nes kiekvienas žaidimo partija yra naujas iššūkis prieš naują žaidėją, o ne prieš tą patį kompiuterį.

Tinklo sistemos darbas yra užtikrinti sėkmingą bendravimą tarp kelių nutolusių kompiuterių pasinaudojus tinklo resursais. Šia problemą galima išskaidyti į keletą smulkesnių: patikimas duomenų perdavimas nepatikimais ryšio kanalais, sėkmingas laiko veiksmų sinchronizavimas tarp kelių kompiuterių.

Egzistuoja keli tinklo architektūrų modeliai: centralizuoto valdiklio (angl. *Server*) ir žaidėjo-valdiklio. Šių architektūrų schemos vaizduojamos paveikslėlyje 13 pav.: A. Žaidėjo-valdiklio modulis, B – centralizuoto valdiklio modelis.



A. Žaidėjo-valdiklio modelis

B. Centralizuoto valdiklio modelis

13 pav. Tinklo architektūra

Žaidėjo-valdiklio architektūroje žaidėjas ir valdiklis veikia viename kompiuteryje, kiti žaidėjai jungiasi prie šiame kompiuteryje veikiančio valdiklio. Ši architektūra labai patogi.

Centralizuoto valdiklio architektūroje žaidimų valdiklis veikia atskirame tam skirtame kompiuteryje ir visi žaidėjai jungiasi prie šio kompiuterio. Centralizuotas valdiklis dažnai šioje architektūroje vykdo žaidėjų ir žaidimo pasaulio valdymo veiksmus.

Žaidimai, kuriose veiksmas vyksta pastoviai, nenutrūkstamai (pvz., pirmo asmens šaudyklės arba žaidimas vaidmenimis) ir vieno ar kelių žaidėjų pasitraukimas iš žaidimo nepadarys didelės įtakos žaidimui, palaiko ir vieną ir kitą architektūros modelį. Tuo tarpu

žaidimai, kurie turi fiksuotą žaidimo tikslą ir pabaigą, fiksuotą žaidėjų skaičių ir griežtesnes žaidimo taisykles (pvz., strateginiai žaidimai), dažniausiai realizuoja tik žaidėjo-valdiklio architektūrą.

5.2.7. Įvesties sistema

Įvesties modulio pagrindinis tikslas yra pranešti žaidimui apie įvedimo signalo pasikeitimus. Įvesties įrenginiai dažniausiai būna klaviatūra ir pelė, tačiau kartais yra naudojami ir kiti įrenginiai: vairalazdė, vairas arba kiti įvairūs prietaisai.

5.2.8. Dirbtinis intelektas

Dirbtinis intelektas (DI) žaidimuose – tai sumanaus, gudraus elgesio imitavimas kompiuteriu [1].

Realizuoti personažus, sugebančius optimaliai suskaičiuoti kelią iki priešo ar taikliai jį nušauti, nėra sunku, tačiau žaidėjui varžytis su tokiu virtualiu priešu nėra įdomu [2]. Tokiu atveju siekiant sudaryti realistiškumo pojūtį, reikia sukurti mažiau erzinantį – klystantį dirbtinį intelektą.

Žaidimuose galima išskirti tokius pagrindinius dirbtinio intelekto uždavinius:

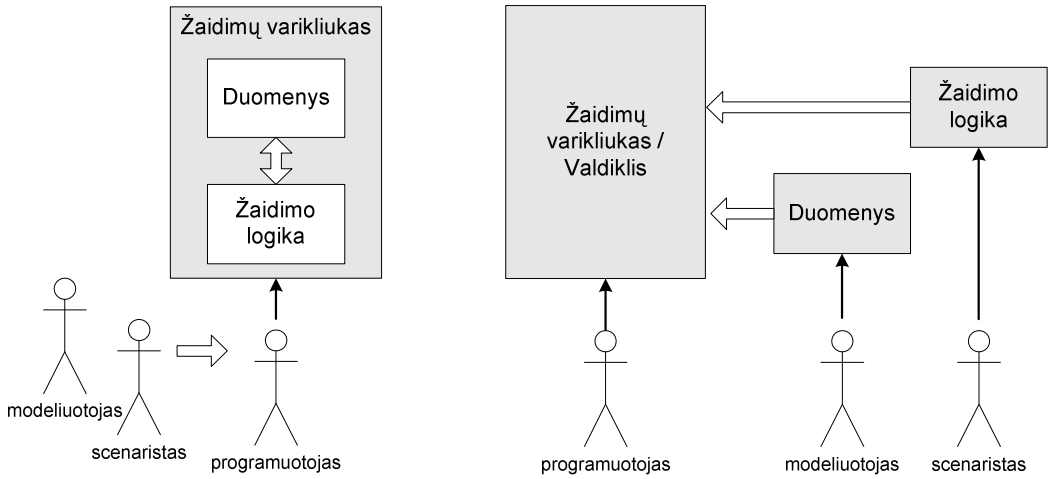
- kelio paieška – svarbus, ypatingo kūrėjo dėmesio reikalaujantis uždavinys, kuris pasitaiko daugelyje žaidimų. Kelio paieška – tai kelio tarp dviejų taškų radimas, įvertinus kelio sudėtingumą ir kliūtis.
- sprendimų priėmimas – visuose žaidimuose, kuriuose yra kompiuterių valdomų žaidėjų, nepriklausomai nuo žaidimo tipo, tenka atlikti sprendimų priėmimą. Sprendimų priėmimas – tai žaidėjo elgesys tam tikru laiko momentu ir tam tikroje situacijoje bei reakcija į pasikeitimus aplinkoje. Tariama, kad žaidėjas „priima sprendimą“ kaip pasielgti, kokius veiksmus atlikti tam tikroje situacijoje.
- žaidėjų išskirtinių savybių generavimas – tai žaidėjų įvairių charakteristikų reikšmių generavimas pagal tam tikrus norimus parametrus.
- žaidimo logikos valdymas – bendras žaidimo eigos valdymas pagal numatytą žaidimo scenarijų.

6. Siūlomas žaidimų varikliuko architektūros modelis ir jo realizacija

Egzistuoja įvairios žaidimų varikliukų architektūros, kiekvienas varikliukas turi savąją architektūrą. Nėra vienos geriausios architektūros, kiekviena architektūra turi savo pliusų ir minusų, vienos yra labiau pritaikytos konkretiems žaidimams, kitos labiau abstrakčios ir universalesnes.

Mes siūlome savo į objektus orientuotą architektūrinį sprendimą žaidimų varikliukui, remdamiesi duomenimis valdoma detaliąja architektūra. Šią architektūrą pasirinkome dėl realizavimo patogumo C++ kalboje ir dėl patogaus žaidimo varikliuko funkcionalumo objektinio reprezentavimo. Paveikslėlyje 14 pav. vaizduojamas skirtumas tarp įprastinės žaidimų varikliuko (A. paveikslėlio dalis) ir duomenimis valdomo varikliuko (B. paveikslėlio dalis) schemų.

Mūsų pasirinktame modelyje (14 pav. B.) žaidimų programuotojai gali koncentruotis ties žaidimo programavimu, o scenaristai ir modeliuotai gali patys laisvai manipuliuoti duomenimis ir žaidimo logika.



A. Įprastinė žaidimų varikliuko sistema

B. Duomenimis valdomo žaidimų varikliuko sistema

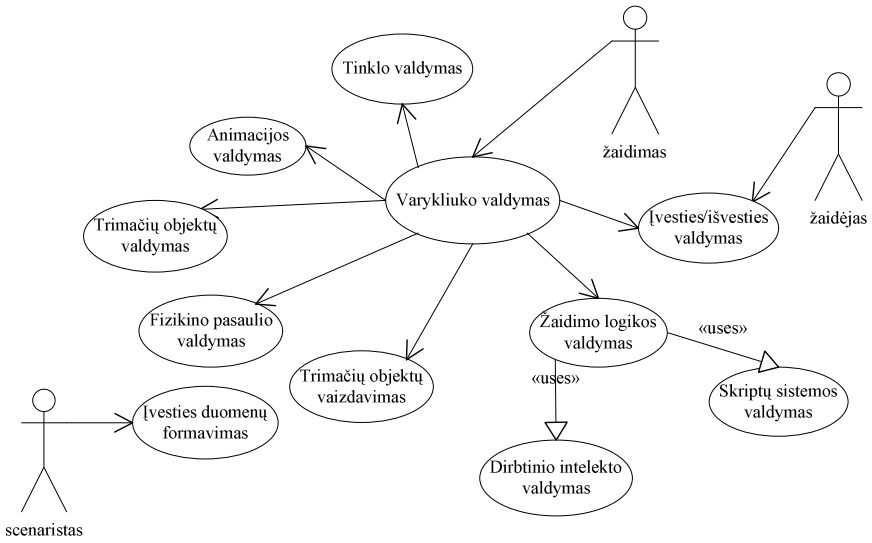
14 pav. Žaidimo valdymo ir žaidimo logikos atskyrimas

Šiame skyriuje aprašome reikalavimus, keliamus mūsų žaidimų varikliuko prototipui, ir architektūrą, tenkinančią šios reikalavimus.

6.1. Žaidimų varikliuko panaudojimo atvejai

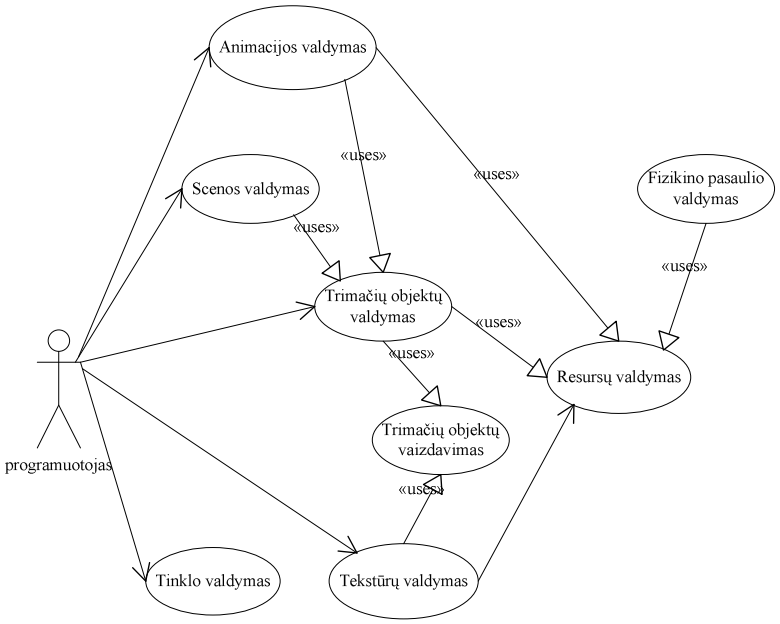
Savo kuriamam funkcinius reikalavimus varikliukui pateiksime dviem pjūviais:

1. Panaudojimo atvejai iš žaidimų kūrėjų perspektyvos vaizduojame paveikslėlyje 15 pav. Čia matome pagrindinius panaudojimo atvejus ir žaidimo bei scenaristo sąveiką su jais.



15 pav. Žaidimų varikliuko panaudojimo atvejai iš žaidimų kūrėjų perspektyvos

2. Panaudojimo atvejai iš programuotojų perspektyvos, vaizduojami paveiksle 16 pav. Čia matome panaudojimo atvejus, kurie yra naudojami programuojant žaidimus, bei jų tarpusavio sąveiką.



16 pav. Žaidimų varikliuko funkciniai reikalavimai iš žaidimų programuotojų perspektyvos

6.2. Aukščiausio lygmens architektūra

Mes pasirinkome aktyvaus žaidimo architektūros tipą (daugiau apie šį architektūros tipą skaitykite šio dokumento skyriuje *5.1.2 Aktyvaus žaidimo architektūra*), nes šio architektūros tipo tarpkomponentinė sąveika yra lengviau realizuojama ir suprantama. Be to, mes neturėjome tikslo, kad varikliukas palaikytų žaidimų modifikacijas.

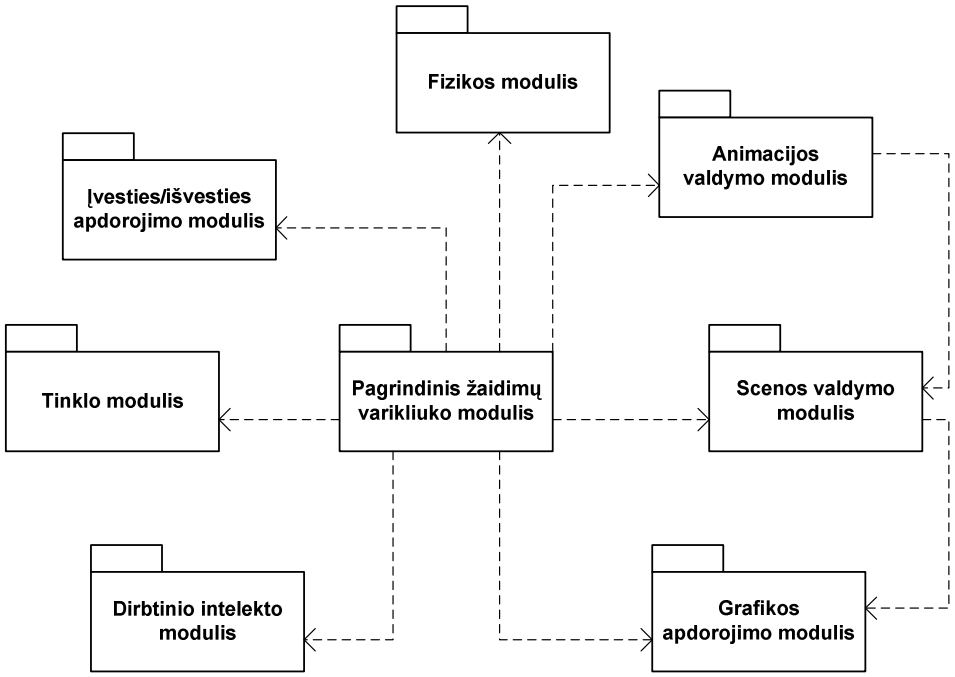
Kiekvieną varikliuko modulį (apie modulius skaitykite šio dokumento skyriuje *6.3 Žaidimų varikliuko architektūra*) realizavome kaip dinaminę biblioteką (apie dinamines bibliotekas detaliau skaitykite šio dokumento priede *Priedas 1. Dinaminės ir statinės bibliotekos*), o žaidimas, kuriamas mūsų žaidimų varikliukui, turės būti realizuotas kaip vykdomoji rinkmena (angl. *Executable File*).

6.3. Žaidimų varikliuko moduliai

Tam, kad mūsų varikliukas atitiktų iškeltus funkcinius reikalavimus (apie funkcinius reikalavimus skaitykite šio dokumento skyriuje *6.1 Žaidimų varikliuko panaudojimo atvejai*), nusprendėme realizuoti tokius žaidimų varikliuko modulius:

- *Fizikos modulis* – modulis, imituojantis žaidimo objektų tarpusavio fizikinę sąveiką.
- *Grafikos apdorojimo modulis* – modulis, atliekantis žaidimo grafikos apdorojimą.
- *Animacijos valdymo modulis* – modulis, atliekantis žaidimo trimačių objektų formų ir pozicijos pasikeitimą.
- *Įvesties/išvesties apdorojimo modulis* – modulis, vykdantis įvesties/išvesties įrenginių signalų apdorojimą.
- *Scenos valdymo modulis* – modulis, atliekantis scenos objektų valdymą.
- *Tinklo modulis* – modulis, leidžiantis vykdyti žaidimą tarp kelių žaidėjų.
- *Dirbtinio intelekto modulis* – modulis, atliekantis žaidimo logikos valdymą.
- *Pagrindinis žaidimų varikliuko modulis* – modulis, kuris atlieka visų aukščiau išvardintų modulių valdymą ir sąveiką su žaidimu.

Paveikslėlyje 17 pav. vaizduojami visi mūsų žaidimų varikliuko prototipo moduliai ir jų sąveiką.



17 pav. Žaidimų varikliuko modulių diagrama

6.4. Detali modulių architektūra

Kiekvienas žaidimų varikliuko modulio funkcionalumą pateikėme per vieną klasę, kurią realizavome pasinaudoję fasado projektavimo šablonu.

6.4.1. Vaizdavimo modulis

Grafikos modulis dažniausiai skaidomas į dvi dalis: priklausomą ir nepriklausomą nuo žemesnio lygio bibliotekų (tvarkyklių). Šuo metu egzistuoja dvi plačiai žinomos ir naudojamos grafikos bibliotekos: DirectX ir OpenGL.

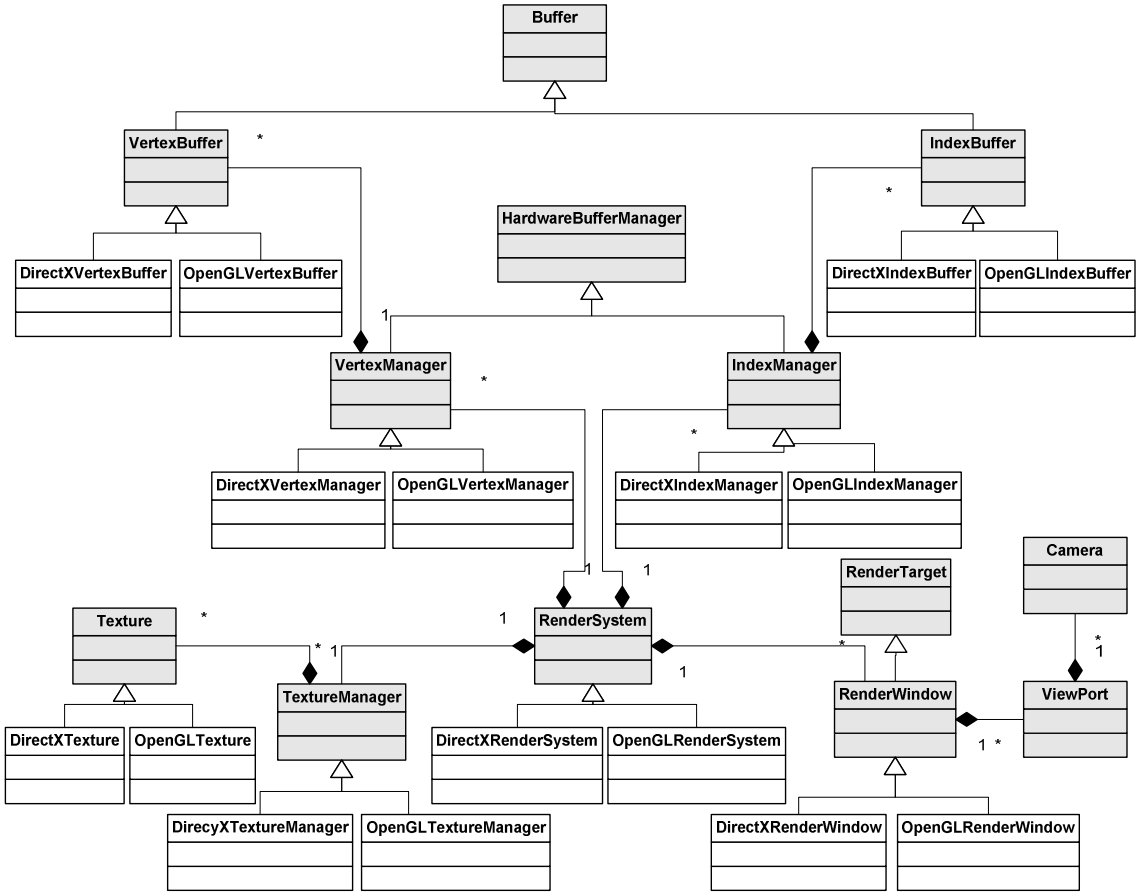
DirectX biblioteka veikia tik Microsoft Windows operacinėje sistemoje, tuo tarpu OpenGL veikai įvairiuose sistemose. Tačiau DirectX yra geriau išplėtotą ir norint kurti varikliuką su šia biblioteka reikalingas mažesnis įdirbis. OpenGL veikimas ant kelių sistemų yra labai didelis privalumas, todėl ta pati programa, sukompiliuota ant kelių skirtingų sistemų, turėtų veikti identiškai. Nesvarbu kokia biblioteka bus pasirinkta, tačiau ją visada pravartu paslėpti po abstrakčiu sluoksniu, kad vėliau, reikalui esant, po šiuo sluoksniu galėtų atsirasti bet kokia konkreti biblioteka.

Lentelėje 3 lentelė.pateikėme savo grafikos valdymo modulio klases ir jų aprašymus.

3 lentelė. Grafikos vaizdavimo modulio klasės

Nr.	Klasės pavadinimas	Klasės paskirtis
1.	*RenderSystem*	Abstrakti klasė, paslepianti vaizdavimo sistemos realizaciją.
1.1.	*DirectXRenderSystem*	Konkreti klasė, realizuojanti vaizdavimo sistemą, pritaikyta DirectX grafinei bibliotekai.
1.2.	*OpenGLRenderSystem*	Konkreti klasė, realizuojanti vaizdavimo sistemą, pritaikyta OpenGL grafinei bibliotekai.
2.	*TextureManager*	Abstrakti klasė, paslepianti tekstūrų valdymo realizaciją.
2.1.	*DirectXTextureManager*	Konkreti klasė, realizuojanti tekstūrų valdymą, pritaikyta DirectX grafinei bibliotekai.
2.2.	*OpenGLTextureManager*	Konkreti klasė, realizuojanti tekstūrų valdymą, pritaikyta OpenGL grafinei bibliotekai.
3.	*RenderTarget*	Abstrakti klasė, apibrėžianti scenos atvaizdavimo sritį.
3.1.	*RenderWindow*	Abstrakti klasė, paslepianti lango realizaciją.
3.1.1.	*DirectXRenderWindow*	Konkreti klasė, realizuojanti lango realizaciją , pritaikyta DirectX grafinei bibliotekai.
3.1.2.	*OpenGLRenderWindow*	Konkreti klasė, realizuojanti lango realizaciją , pritaikyta OpenGL grafinei bibliotekai.
4.	*HardwareBufferManager*	Abstrakti klasė, paslepianti vaizdavimo buferio realizaciją.
4.1.	*IndexManager*	Abstrakti klasė, paslepianti viršūnių indeksų valdymo realizaciją.
4.1.1.	*DirectXIndexManager*	Konkreti klasė, realizuojanti viršūnių indeksų valdymo realizaciją , pritaikyta DirectX grafinei bibliotekai.
4.1.2.	*OpenGLIndexManager*	Konkreti klasė, realizuojanti viršūnių indeksų valdymo realizaciją , pritaikyta OpenGL grafinei bibliotekai.
4.2.	*VertexManager*	Abstrakti klasė, paslepianti viršūnių valdymą.
4.2.1.	*DirectXVertexManager*	Konkreti klasė, realizuojanti viršūnių valdymą , pritaikyta DirectX grafinei bibliotekai.
4.2.2.	*OpenGLVertexManager*	Konkreti klasė, realizuojanti viršūnių valdymą , pritaikyta OpenGL grafinei bibliotekai.
5.	*Buffer*	Abstrakti klasė, paslepianti buferio realizaciją.
5.1.	*VertexBuffer*	Abstrakti klasė, paslepianti viršūnių buferio realizaciją.
5.1.1.	*DirectXVertexBuffer*	Konkreti klasė, realizuojanti viršūnių buferį , pritaikyta DirectX grafinei bibliotekai.
5.1.2.	*OpenGLVertexBuffer*	Konkreti klasė, realizuojanti viršūnių buferį , pritaikyta OpenGL grafinei bibliotekai.
5.2.	*IndexBuffer*	Abstrakti klasė, paslepianti viršūnių indeksų realizaciją
5.2.1.	*DirectXIndexBuffer*	Konkreti klasė, realizuojanti viršūnių indeksus, pritaikyta DirectX grafinei bibliotekai.
5.2.2.	*OpenGLIndexBuffer*	Konkreti klasė, realizuojanti viršūnių indeksus, pritaikyta OpenGL grafinei bibliotekai.
6.	*Texture*	Abstrakti klasė, paslepianti tekstūros funkcionalumo realizaciją.
6.1.	*DirecXTexture*	Konkreti klasė, realizuojanti tekstūros funkcionalumą , pritaikyta DirectX grafinei bibliotekai.
6.2.	*OpenGLTexture*	Konkreti klasė, realizuojanti tekstūros funkcionalumą , pritaikyta OpenGL grafinei bibliotekai.
7.	*Camera*	Klasė, apibrėžianti kameros funkcionalumą.
8.	*ViewPort*	Klasė, apibrėžianti scenos atvaizdavimo sritį lange.

Paveikslėlyje 18 pav. pavaizduota mūsų grafikos valdymo modulio klasių diagrama, kuri realizuoja atsiribojimą nuo konkrečių grafikos bibliotekų.



18 pav. Buferių ir langų valdymo klasių diagrama

Fasadinė klasė *RenderSystem* yra abstrakti ir ji nusako visas funkcijas, kurias privalo realizuoti vaizdavimo klasė. Egzistuoja keli šios klasės realizavimai: *DirectXRenderSystem* ir *OpenGLRenderSystem*. *RenderSystem* klasė taip pat realizuoja fabriko metodo projektavimo šabloną, kuris užtikrina vaizdavimo sistemos resursų valdymo objektų sukūrimą, tokių kaip, *TextureManager*, *VertexManager*, *IndexManager* ir *RenderWindow*. *TextureManager*, *IndexManager* ir *VertexManager* taip pat realizuoja fabriko metodo projektavimo šabloną norint užtikrinti teisingą resursų sukūrimą.

Vaizdavimo biblioteka dažnai turi daug pagalbinių klasių, kurios yra nepriklausomos nuo žemo lygio grafinių bibliotekų. Norint duomenis perduoti žemesnio lygmens bibliotekoms, dažnai juos tenka konvertuoti į tam tikrus formatus. Šie duomenys būna reprezentuojami klasėmis, kurios nėra susijusios su dideliais grafikos resursais, todėl duomenų konvertavimas

į grafikos bibliotekos suprantą formatą yra priimtinas sprendimas. Keletas tokių duomenų klasių pavyzdžių: *Vector, Matrix, Quaternio, Material* ir panašiai.

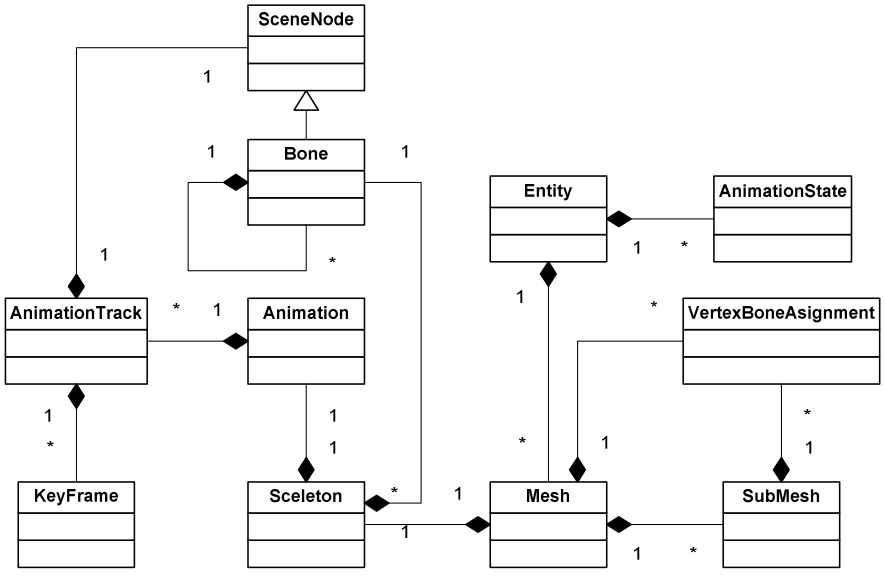
Kartais sudėtingesnį žemo lygo bibliotekų funkcionalumą, kuris nėra tiesiogiai susijęs su aparatine įranga, tenka pakartotinai realizuoti žaidimo varikliuke, taip padidinant nepriklausomumą nuo žemo lygio bibliotekų. Keletas tokių klasių pavyzdžių: M*esh, Submesh, Animation.*

Lentelėje 4 lentelė. pateikėme savo grafikos modulio trimačių objektų valdymo klasių aprašus.

4 lentelė. Trimačių objektų valdymo klasės

Nr.	Klasės pavadinimas	Klasės paskirtis
1.	*Mesh*	Klasė, saugojanti savyje trimačio objekto struktūrą.
2.	*SubMesh*	Klasė, sauganti trimačio objekto dalis, kurios yra vienodai vaizduojamos.
3.	*SceneNode*	Klasė, sujungianti pasaulio objektus į hierarchinę struktūrą.
4.	*Movable*	Klasė, nusakanti judinamo objekto savybes.
4.1.	*Entity*	Klasė, skirta saugoti *SubEntity* klasės objektams ir juos bendrai valdyti.
5.	*VertexData*	Klasė, skirta aprašyti *VertexBuffer* panaudojimą.
6.	*IndexData*	Klasė, skirta aprašyti IndexBuffer panaudojimą.
7.	*SubEntity*	Klasė, skirta *SubMesh* vaizdavimui.
8.	*Renderable*	Abstrakti klasė, skirta paslėpti vaizdavimo realizaciją.
9.	*Material*	Klasė, sauganti savyje trimačio objekto vaizdavimo savybes.

Paveikslėlyje 19 pav. pavaizduota mūsų grafikos modulio trimačius objektus aprašanti klasių diagrama.



19 pav. Trimačių objektų valdymo klasių diagrama

Pagrindinė klasė, kuri aprašo trimatį objektą – yra *Mesh* klasė. Dažniausiai trimatis objektas susideda iš keliu smulkesnių dalių, kurios turi skirtingas vaizdavimo savybes (kaip peilio rankena ir peilio ašmenys, viena matinė kita blizganti), todėl *Mesh* klasė turi savyje keletą *SubMesh* klasės objektų. *Mesh* klasė taip pat turi ir *VertexData* klasės objektą, kuris saugo informaciją apie trimačio objekto naudojamas viršūnes. *VertexData* klasė yra bendra visiems *SubMesh* klasėms, tačiau kiekviena *SubMesh* klasė gali turėti ir savo nuosavą *VertexData* objektą.

Kiekvienas *Mesh* klasės objektas turi bent vieną *SubMesh* klasės objektą. *SubMesh* klasės objektas savyje taip pat turi *IndexData* klasės objektą, kuris nusako viršūnių indeksus.

Entity ir *SubEntity* klasės yra naudojamos trimačių objektų vaizdavimui aprašyti. Vienas *Mesh* ir jam priklausantys *SubMesh* dažnai gali turėti skirtingą atvaizdavimą (pvz., baliono geometrija ta pati, tačiau jis gali būti įvairių spalvų), tai aprašo *Entity* ir *SubEntity* klasės.

Visada vaizduojamos tik *SubEntity* klasės, tai realizavome paveldint *Renderable* klasę. *Entity* klasė yra klasė, kuri saugo to paties trimačio objekto *SubEntity* klasės objektus, ir, paveldėdamos *Movable* klasę, laidžia trimačiam objektui judėti pasaulyje.

Node klasę naudojame trimačio pasaulio objektų tarpusavio sąryšiui nusakyti. Pastarosios klasės pagalba sudarome medžio struktūrą, kurios pagalba, judinant tėvinius mazgus, juda ir vaiko mazgai. Prie šių mazgų suteikėme galimybę prikabinti *Movable* klasės objektus, kurie kartu judės su šiais mazgais.

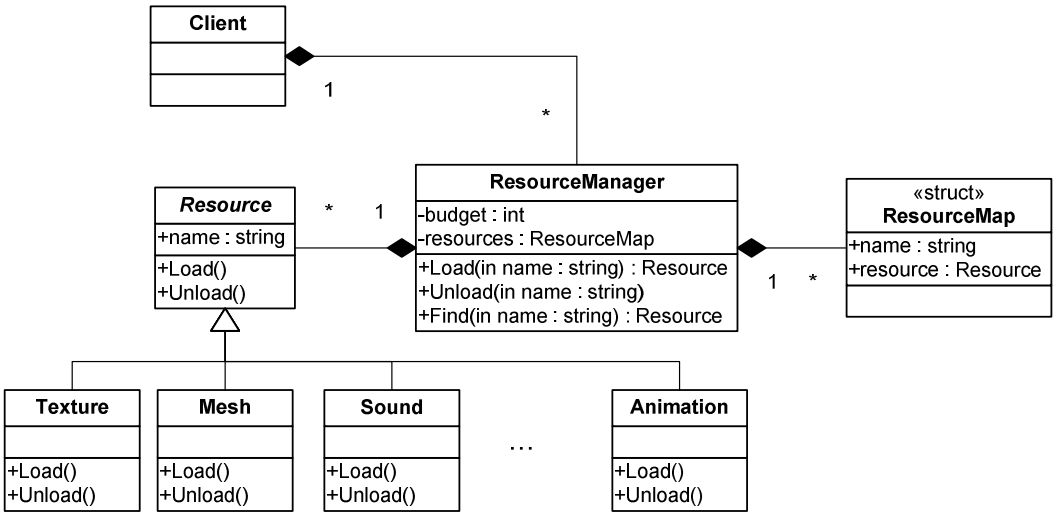
6.4.2. Resursų valdymo modulis

Resursų valdymo modulį realizavome kaip klasę, manipuliuojančią abstrakčia resursų klase. Šio modulio klasių aprašus pateikėme lentelėje 5 lentelė.

5 lentelė. Resursų valdymo klasės

Nr.	Klasės pavadinimas	Klasės paskirtis
1.	*Client*	Klasė, besinaudojanti resursais (tekstūra, garsu ar kitais).
2.	*ResourceManager*	Klasė, valdanti resursų naudojimą
3.	*ResourceMap*	Struktūra, sauganti resurso pavadinimo ir paties resurso sąryšį.
4.	*Resource*	Abstrakti klasė, paslepianti resursų realizavimą.
4.1.	*Texture*	Abstrakti klasė, paslepianti tekstūros funkcionalumo realizaciją.
4.2.	*Sound*	Garso resurso klasė.
4.3.	*Animation*	Animacijos resurso klasė.
4.4.	*Mesh*	Klasė, saugojanti savyje trimačio objekto struktūrą.

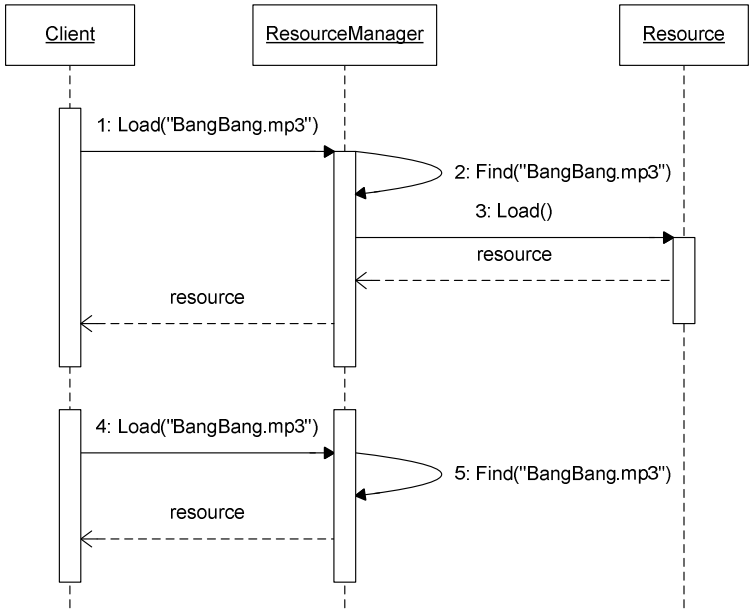
Resursų modulio klasių diagrama vaizduojama paveikslėlyje 20 pav.



20 pav. Resursų valdymo modulio klasių diagrama

Kiekvienas konkretus resursas privalo realizuoti *Load* ir *Unload*, kuriame aprašoma kiekvieno konkretaus resurso užkrovimo ir atlaisvinimo logika.

Paveikslėlyje 21 pav. vaizduojame resursų modulio dinaminį vaizdą.



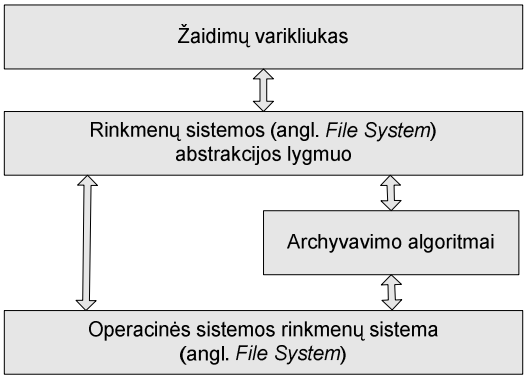
21 pav. Resursų valdymo modulio dinaminis vaizdas

Resursų valdiklio paprašius užkrauti konkretų resursą, perduodant jo pavadinimą per *name* parametrą, jis pirma ieško ar resurso nėra tarp jau užkrautų resursų. Jei resursas

randamas jis perduodamas klientui, jei resurso nepavyko rasti jis sukuriamas, įdedamas į jau sukurtų resursų sąrašą ir grąžinamas klientui. Taip pat egzistuoja maksimalus atminties biudžetas, kurį resursų valdiklis gali išnaudoti. Viršijus šį biudžetą, resursų valdiklis gali grąžinti resursą kurį panaudojus vartotojas iškart suprastų apie resursų viršijimą (pavyzdžiui, tekstūrų atveju, grąžinama raudona tekstūra su užrašu „ERROR“), arba išmetamas klaidos pranešimas.

6.4.2.1. Resursų suspaudimas

Labai dažnai žaidimo varikliukui reikia pasiekti suspaustus resursus, esančius kokiame nors archyve. Žaidimų programuotojams nepatogu kreiptis į archyvus kokiu nors išimtiniu būdu, todėl mes siūlome šį kreipimosi procesą paslėpti po abstrakčiu rinkmenų sistemos (angl. *file system*) lygmeniu. Tokio resursų paslėpimo schema vaizduojama paveikslėlyje 22 pav.

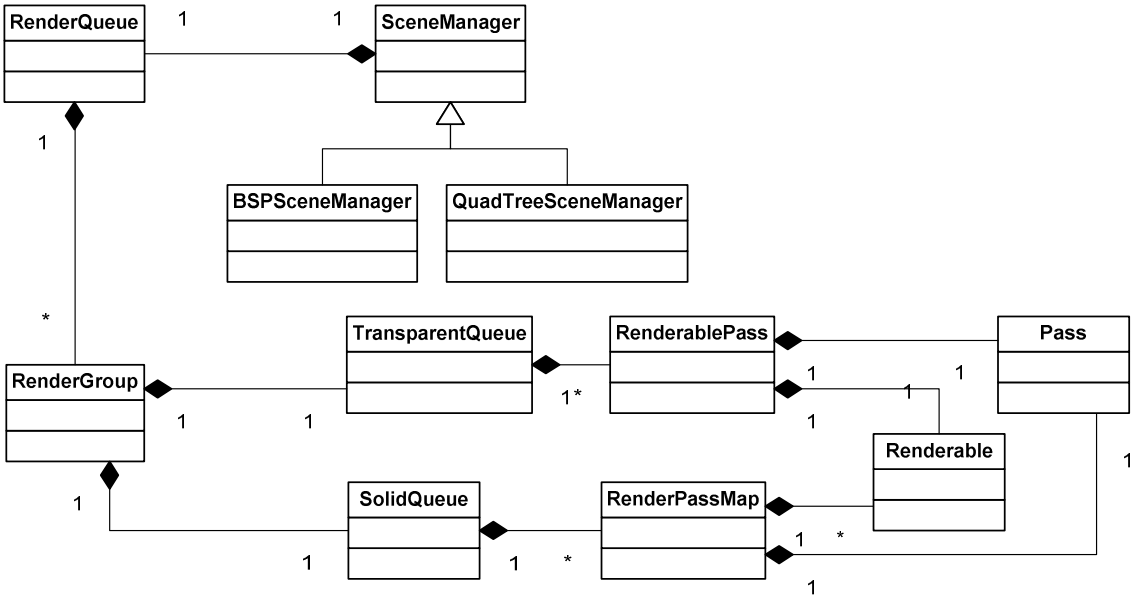


22 pav. Abstrakčios rinkmenų sistemos schema

Šiame resursų paslėpimo lygmenyje paprastos rinkmenos atidaromos, naudojant standartines operacinės sistemos funkcijas, o archyvai yra išskleidžiami, naudojant tam tikrą išskleidimo algoritmą. Tokiu atveju resursų lokalizacija yra unifikuota, programuotojui nurodant resursą, archyvas traktuojamas kaip sistemos katalogas (pvz. /resursai/archyvas/vidinis_katalogas/resursa.txt, kur „archyvas“ iš tiesų yra „archyvas.zip“ rinkmena).

6.4.3. Scenos modulis

Scenos valdymo modulio pagrindinis tikslas yra greitai rasti matomus trimačius objektus, bei efektyviai juos vaizduoti. Šiam tikslui pasiekti mes siūlome architektūra pavaizduotą 23 pav. paveikslėlyje.



23 pav. Scenos valdymo modulio klasių diagrama

Šio modulio klasių aprašymai pateikti lentelėje 6 lentelė.

6 lentelė. Animacijos modulio klasės

Nr.	Klasės pavadinimas	Klasės paskirtis
1.	*RenderQueue*	Klasė, realizuojanti vaizdavimo eilių saugojimą, valdymą ir vaizdavimą.
2.	*SceneManager*	Abstrakti klasė, paslepianti scenos valdymo, t.y. matomų objektų radimo, realizacija.
2.1.	*BSPSceneManager*	Konkreti klasė, realizuojanti scenos valdymą pagal BSP (angl. *Binary Space Partitioning*) algoritmą.
2.2.	*QuadTreeSceneMAnager*	Konkreti klasė, realizuojanti scenos valdymą formuojant medį ir rekursiškai dalinant erdvę į keturias sritis.
3.	*RenderGroup*	Saugo tam tikru metu vaizduojamų objektų sąrašą.
4.	*TransparentQueue*	Permatomų objektų sąrašas.
5.	*Pass*	Saugo atominį viršūnių transformavimo ir vaizdavimo veiksmą, kurį atlieka vaizdo plokštė.
6.	*RenderPassMap*	Saugo *Renderable* objektus, kurie bus vaizduojami tam tikru *Pass* objekto nustatymui.
7.	*SolidQueue*	Nepermatomų objektų sąrašas
8.	*RenderablePass*	Saugo sąryšį tarp *Renderable* ir *Pass* objektų.

Pagrindinė fasadinė klasė yra *SceneManager*, naudojama vaizduojamų objektų radimui ir efektyviam vaizdavimui. *RenderQueue* klasė saugo visas objektų vaizdavimo eiles *RenderGroup*.

RenderGroup klasės objektų dažniausiai būna sukuriama keletas, kiekvienas objektas turi tam tikrą prioritetą, šis prioritetas nusako kokia tvarka jie bus vaizduojami. Prioritetai dažniausiai būna asocijuojami su tam tikru metu piešiamais loginiai objektais: pasaulio dangus, įprastiniai objektai, viršutinis žaidimo meniu (angl. *Heads Up Dsiplay*).

Kiekviena *RenderGroup* savyje saugo dvi vaizdavimo eiles: permatomiems ir nepermatomiems objektams.

Nepermatomi objektai yra sugrupuojami pagal *Pass* klasę. *Pass* klasė skirta saugoti konkretų vaizdo plokštės nustatymų rinkinį Nustačius tam tikrą vaizdo plokštės būseną yra vaizduojami visi to nustatymo trimačiai objektai, taip sumažinant vaizdo plokštės parametrų pasikeitimų kartus.

Norint gauti teisingą permatomų objektų vaizdavimą, objektai yra vaizduojami surūšiavus juos pagal atstumą iki kameros. Šuo atvejų kiekvienam objektui dažniausiai tenka keisti vaizdo plokštės nustatymus.

Egzistuoja ir keltas kitų objekto grupavimo metodų, kurių čia neaptarsime.

6.4.4. Įvesties modulis

Mūsų įvesties modulio architektūra yra gana paprasta. Dažniausiai įvesties įrenginiai bendrauja su sistema dviem metodais:

- kiekvieną kartą įvesties įrenginiai pasitikrina ar nėra paspaustas koks nors klavišas;
- panaudojus stebėtojo projektavimo šabloną, įvesties įrenginys praneša žaidimui apie klavišų paspaudimo pasikeitimą.

Pirmasis variantas yra efektyvesnis už antrąjį, nes pirmame variante žaidimas pasiima įvesties simbolius visus iškarto. Antru atveju varikliukas praneša apie kiekvieną įvesties simbolį atskirai. Egzistuoja kombinuoti pastarųjų metodų variantai, pranešant apie įvesties įvykius perduodant juos ne po vieną, o keletą iškarto.

Įvesties įrenginiai yra labai skirtingi, todėl nėra labai patogu visus įvesties įrenginius paslėpti po abstrakčia sąsaja.

Mes realizuodami įvesties modulį, pasirinkome pirmąjį variantą, t.y. kiekvienas įvesties įrenginys pasitikrina, ar nėra paspaustas koks nors klavišas ar mygtukas.

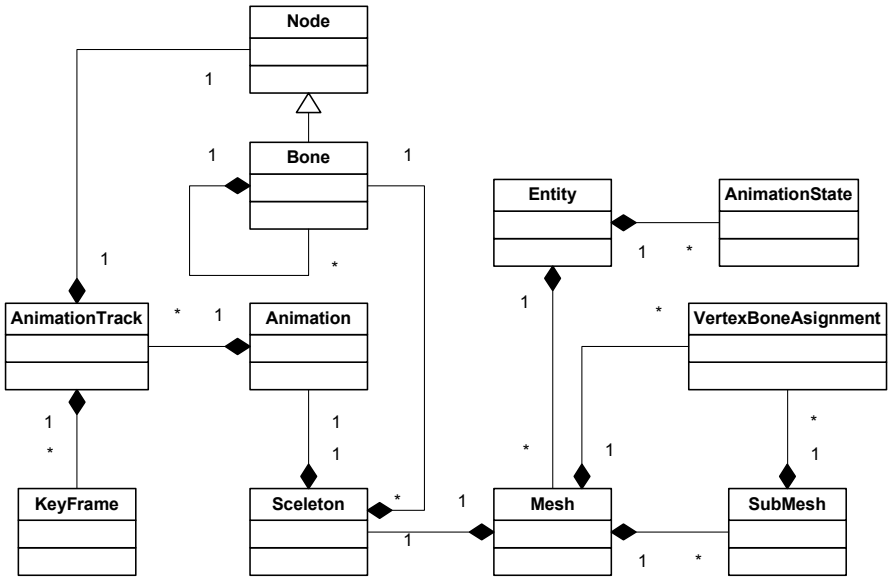
6.4.5. Animacijos modulis

Mūsų realizuoto animacijos valdymo modulio klasės ir jų aprašymai pateikti lentelėje 7 lentelė.

7 lentelė. Animacijos valdymo klasės

Nr.	Klasės pavadinimas	Klasės paskirtis
1.	*Bone*	Klasė skirta skeletui formuoti.
2.	*AnimationTrack*	Klasė, skirta saugoti vienos animacijos kadrus.
3.	*Animation*	Klasė, realizuojanti *AnimationTracks* valdymą.
4.	*AnimationState*	Klasė, sauganti animacijos būsena tam tikru laiko momentu
5.	*VertexBoneAsignment*	Viršūnių, kaulų ir svorių susiejimas.
6.	*Sceleton*	Atsakinga už visą skeleto animacijos veikimą.
7.	*KeyFrame*	Animacijos kadras, saugantis objekto būseną tam tikro laiko momentu.

Animacijos modulio detali architektūra vaizduojama paveikslėlyje 24 pav.



24 pav. Animacijos modulio detali architektūra

Pagrindinė klasė yra *Animation*. Ši klasė saugo kelias atskiras animacijas *AnimartionTrack* (animacijos takelis). Kiekviena atskira animacija yra sudaryta iš kelių *KeyFrame* klasių. Šios klasės saugo animacijos kadrą arba kitaip tariant vietos, mastelio ir posūkio reikšmes tam tikru laiko momentu. Kadangi dažniausiai kadrų būna žymiai mažiau,

nei reiktų rodyti, tai vieta, mastelis ir posūkis yra suliejami tarp artimiausių dviejų kadrų. Tam tikras animacijos takelis yra susiejamas su tam tikra *Node* klase, kurią paveldi *Bone* klasę.

Bone (kaulo) klasė skirta skeleto animacijai formuoti. *Bone* klasė turi rekursinį ryšį, kuris leidžia formuoti medžio struktūros hierarchiją, kur medžio kamienas būtų pagrindinis kaulas. Pagrindinio kaulo transformacija daro įtaką visų kitų kaulų galutiniai transformacijai. Nuorodą į šį kaulą turi *Skeleton* klasė, kuri atsakinga už visą skeleto animacijos veikimą. *Mesh* ir *SubMesh* savyje saugo kaulų ir viršūnių tarpusavio susiejimą ir susiejimo svorius.

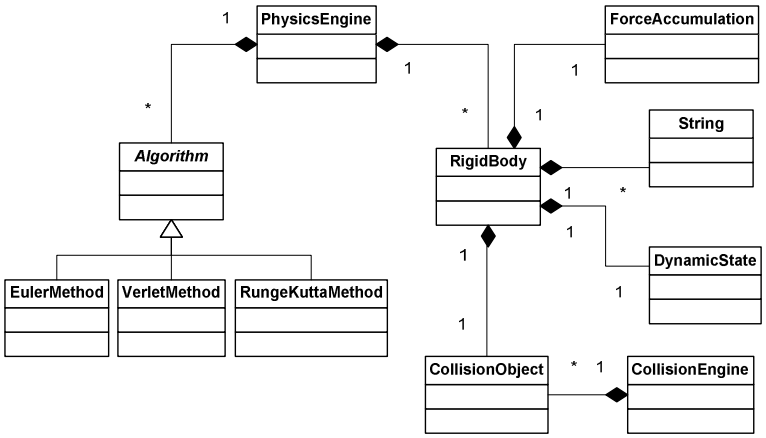
6.4.6. Fizikos modulis

Mūsų fizikos modulio klasės ir jų aprašymai pateikti lentelėje 8 lentelė.

8 lentelė. Fizikos valdymo klasės

Nr.	Klasės pavadinimas	Klasės paskirtis
1.	*PhisicsEngine*	Fizikos klasė, imituojanti fizikinius reiškinius
2.	*Algorithm*	Abstrakti klasė, paslepianti algoritmo realizaciją.
2.1.	*EulerMethod*	Eulerio metodo realizacija.
2.2.	*VerletMethod*	Verlet metodo realizacija.
2.3.	*RungeKuttaMethod*	Runges-Kutp metodo realizacija.
3.	*ForceAccumulation*	Klasė, sauganti objektą veikiančių jėgų atstojamąją.
4.	*String*	Klasė, nusakanti tamprų ryšį tarp kelių kietųjų kūnų.
5.	*RigidBody*	Klasė, sauganti informaciją apie kietųjų kūnų savybes.
6.	*DynamicState*	Klasė, sauganti kietojo kūno dinaminę būseną.
7.	*CollisionObject*	Klasė, aprašanti kūno kontūrus.
8.	*CollisionEngine*	Klasė, saugojanti ir aptinkanti tarpusavio objektų susidūrimus.

Fizikos modulio detali klasių schema pavaizduota 25 pav. paveikslėlyje.

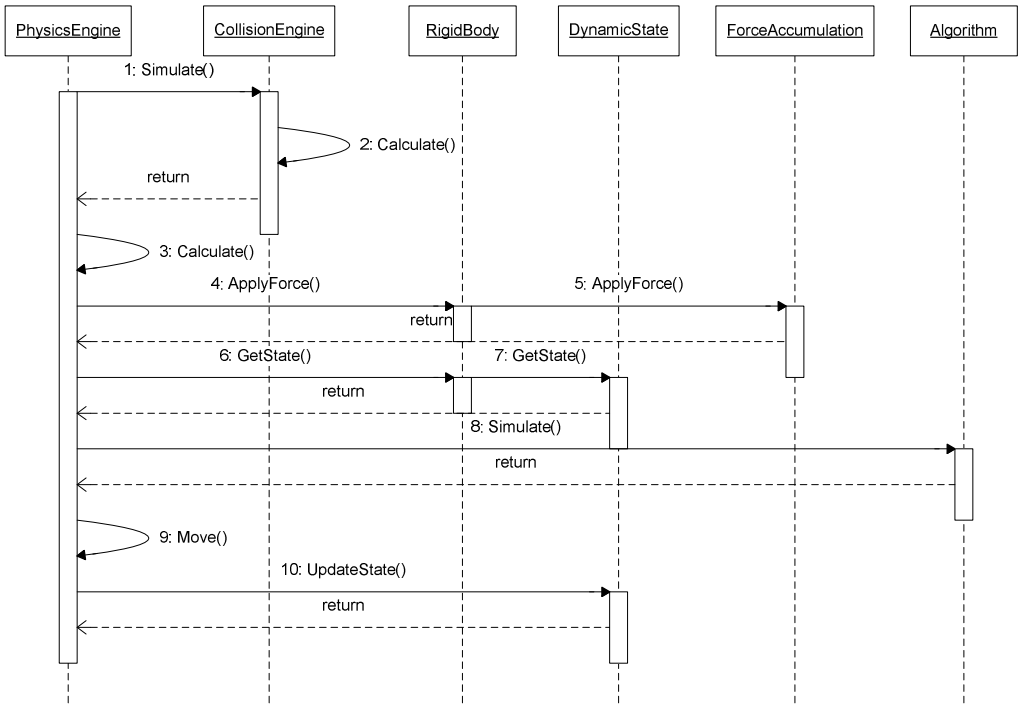


25 pav. Fizikos modulio klasių diagrama

PhysicsEngine – fasadinė klasė, simuliuojanti žaidimo fizikinį pasaulį. Pasinaudodami strategijos projektavimo šablonu, suteikėme galimybę *PhysicsEngine* klasei naudoti vieną iš kelių algoritmų: Eulerio, Verpeto ar Rungės-Kuto. *PhysicsEngine* savyje saugo visus žaidimo pasaulio kietuosius kūnus – *RigidBody*. Kietojo kūno klasė savyje saugo nekintančius duomenis aprašančius kietąjį kūną. Kiekvienas kūnas savyje saugo *DynamicState* objektą, kuris nusako kintančias laike kūno savybes: greitį, kampinį greitį, vietą erdvėje, orientaciją erdvėje ir t.t. Kiekvienas kietasis kūnas yra susijęs su klase, aprašančia kūno kontūrus, - *CollisionObject*. Visi *CollisionObject* objektai saugomi *CollisionEngine* klasėje. Pastaroji klasė atsakinga už objekto tarpusavio susidūrimų aptikimą.

Paveikslėlyje 26 pav. pavaizduota fizikos modulio sekų diagrama. Fizikos dinaminį modulį galima aprašyti tokiais žingsniais:

1. Visų pirma yra randami visi objektai, kurie tam tikru laiko momentu tarpusavyje kertasi.
2. Vėliau surandami susikirtimo taškai ir jų normalės.
3. Suskaičiuojamos jėgos, kurios veikia kiekvieną kūną. Šios jėgus susumuojamos į bendrą jėgos vektorių ir saugomos klasėje *ForceAccumulation*.
4. Pritaikius tam tikra integravimo algoritmą, yra suskaičiuojamos naujos kūnų būsenos.
5. Pagal suskaičiuotas naujas būsenas, kūnai yra pajudinami žaidimų pasaulyje.



26 pav. Fizikos modulio sekų diagrama

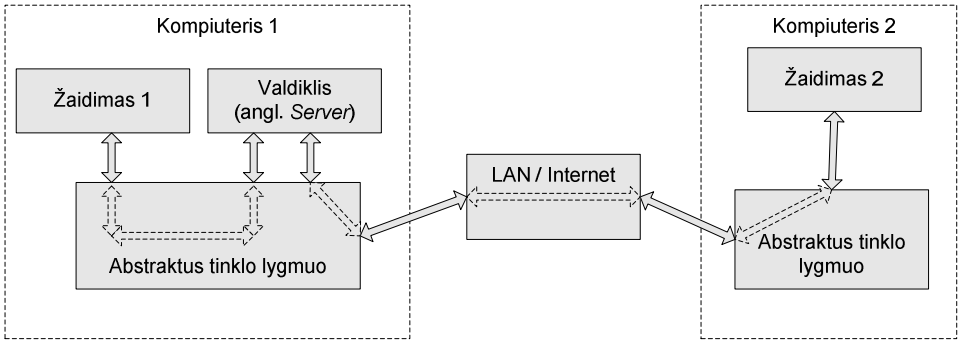
6.4.7. Tinklo modulis

Tinklo modulio architektūra labai priklauso nuo žaidimo tipo, kuriam bus kuriamas žaidimo varikliukas. Šio modulį reikia suprojektuoti ir realizuoti kuo ankstesnėje stadijoje, nes tinklo modulis labai dažnai yra stipriai susietas su paties žaidimo realizavimu.

Mūsų pasiūlyta žaidimo tinklo modulio architektūra geriausiai tinka pirmo asmens šaudyklės žaidimui realizuoti. Ji remiasi kliento/valdiklio architektūros modeliu, kur žaidėjas kaip klientas jungiasi prie serverio, nesvarbu ar tai vieno asmens žaidimas ar kelių.

- Vieno žaidėjo atveju serveris veikia žaidėjo kompiuteryje ir žaidėjas jungiasi per fiktyvią tinklo sąsają prie serverio.
- Kelių žaidėjų atveju žaidėjas jungiasi prie kitame kompiuteryje veikiančio serverio, ir fiktyvi tinklo sąsaja pakeičiama realia tinklo sąsaja.

Šį procesą galima stebėti 27 pav. paveikslėlyje.



27 pav. Abstraktaus tinklo lygmens naudojimas

6.4.8. Dirbtinio intelekto modulis

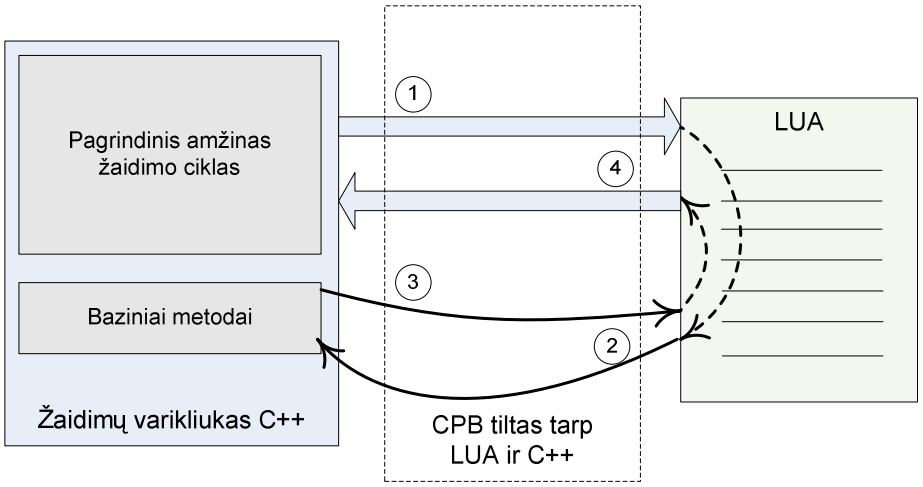
Realizuojant dirbtinį intelektą, naudojome duomenimis valdomą detalią architektūrą.

Duomenų atskirimą atlikome naudodami LUA scenarijus. Visus su DI susijusius duomenis ir logiką nukėlėme į LUA rinkmenas, kurias be programuotojų pagalbos gali patogiai keisti žaidimų kūrėjai. Šie scenarijai yra valdomi žaidimų varikliuke: žaidimo metu jie užkraunami į operatyviąją atmintį ir vykdomi.

Scenarijų valdyme galima išskirti tokius esminius objektus:

- LUA scenarijai – tai žaidimo logikos elementai, iškelti į išorinę atmintį, į atskiras rinkmenas.
- C++ funkcijos ir metodai, valdantys žaidimo personažus ar kitus žaidimo objektus. Į šias funkcijas ir metodus yra kreipiamasi iš LUA scenarijų.
- Scenarijų valdiklis – žaidimo varikliuko objektas, atliekantis scenarijų užkrovimą, vykdymą ir sunaikinimą žaidimo metu.

LUA ir C++ kalbų sujungimo patogumui galima naudoti pagalbines bibliotekas, pavyzdžiui, CPB biblioteką, tai tiltas tarp LUA ir C++ [6].



28 pav. LUA kalba parašytų scenarijų vykdymo schema

Paveikslėlyje *28 pav.* vaizduojama LUA kalba parašytų scenarijų valdymo schema. Scenarijų valdymas – tai veikla, susidaranti iš tokių žingsnių (žingsniai išvardinti vykdymo eilės tvarka):

1. Žaidime užkraunamas LUA scenarijus.
2. Vykdomas LUA scenarijus, LUA scenarijaus vykdymo metu kreipiamasi į C++ funkcijas ir metodus.
3. Iš C++ funkcijų ir metodų grąžinami rezultatai į LUA scenarijų.
4. LUA scenarijų rezultatai grąžinami į žaidimą.

Duomenų ir žaidimo logikos atskirimo nuo varikliuko naudojimas sistemoje paverčia modulius (kurie naudoja tokį atskirimą) duomenimis valdomais (angl. *Data-Driven*). Toks scenarijų naudojimas praplečia žaidimų varikliuko funkcionalumą ir suteikia žaidimų kūrėjams galimybę laisvai manipuliuoti duomenimis.

Scenarijų naudojimas yra patogus ir lankstus, tačiau reikia įvertinti tai, kad scenarijų užkrovimas į operatyviąja atmintį ir LUA funkcijų vykdymas trunka tam tikrą laiką. Todėl derėtų scenarijų valdymą organizuoti taip, kad LUA scenarijai būtų užkraunami kuo rečiau, o jų vykdomos funkcijos truktų kuo ilgiau.

7. Detalios architektūros tipų tyrimas

Egzistuoja įvairūs detalios architektūros tipai, savo varikliuko realizavimui pasirinkome duomenimis valdoma detalią architektūrą.

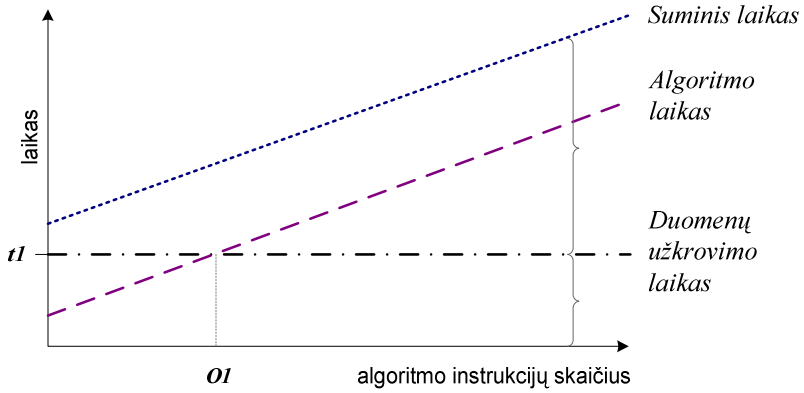
Tolimesniuose skyriuose ištyrėme kuo mūsų pasirinkta architektūra pranašesnė ir labiau patraukli, nei kito tipo (arba įprastinės) detaliosios architektūros.

7.1. Greičio palyginimas

Lyginant duomenimis valdomos detalios architektūros ir kitos detalios architektūros veikimo spartą, galime teigti, kad spartesnės yra įprastinė detalioji architektūra. Tai atsitinka dėl kelių priežasčių:

- Duomenims valdomos architektūros atveju reikalingas apdorojimo žingsnis, kurio metu reikiami duomenys būtų paruošti apdorojimui arba vykdymui. Kitaip tariant, atsiranda tarpinis lygmuo tarp programos ir sistemos resursų.
- Įprastinės detalios architektūros atveju duomenims nereikalingas išankstinis apdorojimas. Be to šių duomenų panaudojimą papildomai optimizuoja kompiliatoriai.

Tačiau pastoviai didėjant procesorių greičiui, ir naudojant sudėtingesnius algoritmus, vis mažesnė procesoriaus apkrovimo dalis tenka duomenų užkrovimo veiksmui. Tai galime pastebėti paveikslėlyje 29 pav.



29 pav. Duomenų užkrovimo ir algoritmo vykdymo laiko santykis

Pagrindinis mūsų pasirinktos architektūros trūkumas – blogas duomenų užkrovimo laiko ir algoritmų vykdymo laiko santykis, vykdant nesudėtingus algoritmus. Tačiau naudojant sudėtingus algoritmus, algoritmo vykdymo laiko ir duomenų užkrovimo laiko santykis gerėja. Apibendrinant, galima teigti, kad tokios architektūros naudojimas pasiteisina, vykdant sudėtingus skaičiavimus.

Be to, mūsų pasirinktas architektūros tipas yra naudingas, vykdant skirtingus skaičiavimus su tais pačiai duomenimis, nes tokiu atvejų duomenis bus užkraunami vieną kartą visiems skaičiavimams.

Greičio eksperimentinis tyrimas

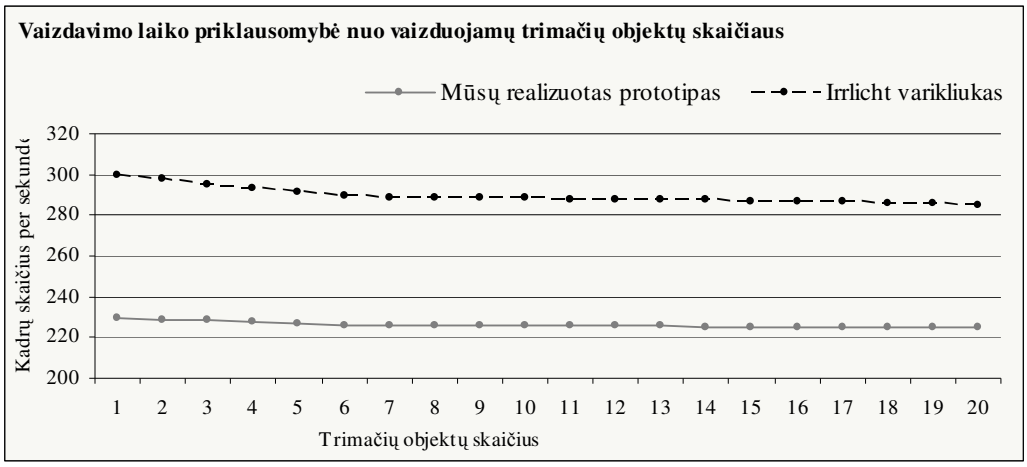
Atlikome savo realizuoto prototipo ir Irrlicht [21] varikliuko (įprastinės architektūros) trimačių objektų vaizdavimo greitį.

Sistemos, kurioje atlikome tyrimą, techninės charakteristikos yra tokios:

- AMD 2Ghz procesorius
- GeForce 6600 GT tipo grafinis procesorius
- 1 GB operatyviosios atminties.

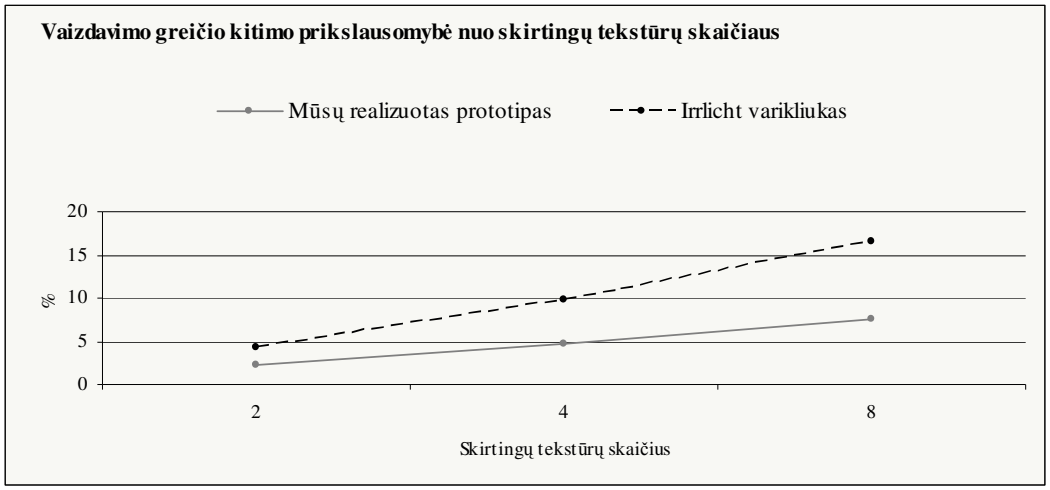
Ištyrėme trimačių objektų vaizdavimo spartą naudojant skirtingų tekstūrų kiekius. Atlikome trys testus, kuriuose vaizdavome nuo 1 iki 20 trimačių objektų vienu metu. Pirmojo testo metu visiem objektams taikėme vieną tekstūrą, antrojo – keturias, trečiojo – aštuonias skirtingas tekstūras.

Paveikslėlyje 30 pav. vaizduojam objektų vaizdavimo greičio priklausomybė nuo vaizduojamų objektų skaičiaus.



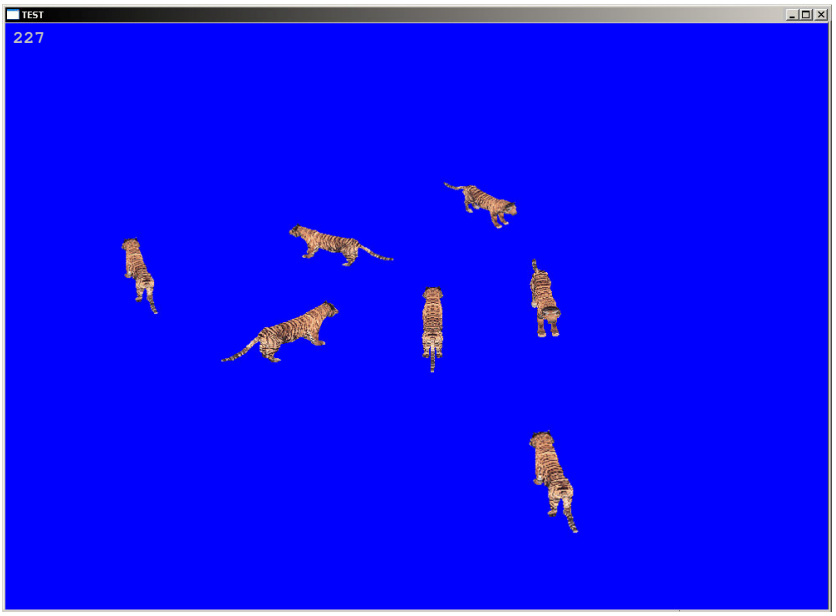
30 pav. Vaizdavimo laiko priklausomybė nuo vaizduojamų objektų skaičiaus

Kaip matome, iš grafiko, vaizduojamo 30 pav. paveikslėlyje, Irrlicht žaidimų varikliukas stipriai lenkia mūsų sukurtą prototipą (jo veikimo greitis – 300 kadrų per sekundę apdorojant vieną objektą, kai mūsų prototipo greitis – 230), nes mūsų prototipas nėra pilnai ištestuotas ir optimizuotas.



31 pav. Vaizdavimo greičio kitimo priklausomybė nuo skirtingų tekstūrų skaičiaus

Tačiau kaip galima pastebėti iš paveikslėlyje 31 pav. vaizduojamo grafiko, kad Irrlicht didėjant lektūrų kiekiui krinta greičiau, nei mūsų varikliuke. Tai galima paaiškinti tuo, kad mūsų duomenimis valdomos architektūros varikliukui yra didesnės galimybės duomenų srautų optimizavimui, nei įprastinės architektūros varikliukui.



32 pav. Mūsų prototipas vaizduoja septinis vienos tekstūros objektus

Paveikslėlyje 32 pav. pateiktas mūsų prototipo lango vaizdas, kai vaizduojami septyni objektai su viena tekstūra.

Priede *Priedas 3. Greičio tyrimo rezultatai* pateikti išsamūs tyrimo rezultatai, su visai gautais duomenimis.

7.2. Patogaus naudojimo ir lankstumo palyginimas

Duomenimis valdoma architektūra yra patogesnė naudoti, nes duomenys esantys išorėje yra žymiai lengviau keičiami nei duomenys įkompiliuoti į programą.

Norint pakeisti duomenis įkompiliuotus į programą, reikia:

1. sustabdyti programos veikimą;
2. tarp įvairių programavimo kalbos specifinių duomenų susirasti reikiamus duomenis ir juos pakeisti;
3. sukompiliuoti programą;
4. paleisti ją iš naujo;
5. atlikti veiksmus su programa, kad pasiekti testinį scenarijų.

Norint pakeisti programos veikimą ir ištestuoti pakeitimus, duomenimis valdomoje architektūroje, tereikia:

1. juos pakeisti išorinėje laikmenoje,
2. priversti programą atnaujinti užkrautus duomenis, net neperkraunant programos.

Tai labai patogu derinant programos veikimą.

Iš pateiktų scenarijų aiškiai matosi labai didelis duomenimis valdomos architektūros pranašumas. Ši architektūra atskirdama duomenis nuo juos apdorojančios programos, leidžia patogesnį, gilesnį ir vaizdesnį duomenų keitimą ir interpretavimą.

Pastaruoju metu vis populiarėjančios scenarijų kalbos leidžia pasyviems duomenims tapti aktyviais, tai yra ne programa interpretuoja duomenis, o duomenys vykdo programos kodą. Ši savybė papildomai padidina duomenimis valdomos detalios architektūros lankstumą.

7.3. Pakartotinio panaudojimo palyginimas

Duomenimis valdoma detali architektūra stipriai padidina ir palengvina pakartotinio panaudojimo galimybes. Kadangi duomenys saugomi išorėje, o programa tik apdoroja šiuos duomenis, tai nauji įvesties duomenys jau savaime yra pakartotinis programos panaudojimas.

7.4. Darbuotojų rolių atskyrimas

Panaudojant duomenimis valdomą architektūrą įvesties duomenų formavimą gali atlikti bet koks žmogus, supažindintas su duomenų struktūra. Tai leidžia programuotojams atsiriboti nuo duomenų derinimo darbų, šiam darbui tikslingai pasamdžius reikalingus darbuotojus, ir sutelkti dėmesį į programos kodo rašymą.

Taip išskaidžius darbuotojų roles, padidėja jų produktyvumas ir siaurėja specializacija. Žinoma duomenims valdomos detalios architektūros programai realizuoti reikalingas ilgesnis laiko tarpas, tačiau žvelgiant į ateitį taip sutaupomas programuotojo darbo laikas, ypač jei įvedimo duomenys labai dažnai keičiami. Dažnai duomenų įvedimui naudojami įvairūs papildomi įrankiai, kurie pateikia duomenis geriau suprantama forma.

8. Išvados

Išanalizavę įvairius architektūros tipus ir jų realizavimo metodus, bei žaidimo varikliukų funkcinius reikalavimus, varikliuko prototipui realizuoti pasirinkome duomenimis valdomą architektūrą, kaip labiausiai tinkančią ir tenkinančią reikalavimus varikliukams.

Pasirinktos architektūros pagrindu kiekvienam varikliuko moduliui pasiūlėme detalų architektūrinį sprendimą ir jo realizavimo būdus.

Sukūrėme varikliuko prototipą, pagrįstą pasirinkta ir išanalizuota architektūra.

Tyrimo metu palyginome įprastines architektūras su pasirinktąja, duomenimis valdoma architektūra. Atlikdami palyginimus, parodėme, kad pasirinkome tinkamą architektūros tipą: jis yra lankstesnis, pakartotinai panaudojamas, lengvai praplečiamas, skatina darbų specializavimą ir yra perspektyvus. Vienintelis jo trukumas yra lėtesnis veikimas, tačiau jis, didėjant procesoriaus galingumui, tampa neesminis.

Kadangi sukūrėme funkcionuojantį žaidimų varikliuko prototipą, galime teikti, kad yra įmanoma realizuoti lankstų žaidimų varikliuką, naudojant duomenimis valdomąją detalią architektūrą.

9. Literatūra

[1] Sanchez-Crespo Dalmau, D. *Core Techniques and Algorithms in Game Programming*, New Riders, 2003, p. 888

[2] Bourg, D.; Seiman, G. *AI for Game Developers*, O'Reilly, 2004, p. 400

[3] Woodland, R. *Filling the Gaps – Advanced Animation Using Stitching and Skinning / Game Programming Gems*, Charles River Media, 2000, p. 614

[4] Wilson, K. *Data-Driven Design / Game Archtect.net,* http://gamearchitect.net/Articles/DataDrivenDesign.html, 2002 05 29

[5] Raymond, E. *The Art of Unix Programming,* http://www.faqs.org/docs/artu/ch09s01.html , 2003

[6] Amy, T. *CPB – Bridging LUA and C++*, http://thomasandamy.com/projects/CPB, 2006 05

[7] Riguer G. *Performance Optimization Techniques for ATI Graphics Hardware with DirectX® 9.0.* ATI Technologies Inc., *2002 12*

[8] *Virtual Machine / Wikipedia, The Free Encyclopedia*, http://en.wikipedia.org/wiki/Virtual_machine, 2006 05 20.

[9] Rabin, S. *The Magic of Data-Driven Architecture / Game Programming Gems*, Charles River Media, 2000, p. 614.

[10] Boer, J. *Object-Oriented Programming and Design Techniques / Game Programming Gems*, Charles River Media, 2000, p. 614.

[11] Eberly, D. *Game Physics*, Morgan Kaufmann Publisher, 2004, p. 776.

[12] Bergen, G. *Collision Detection in Interactive 3D Environments*, Morgan Kaufmann Publisher, 2004, p. 278.

[13] Eberly, D. *3D Game Engine Design. A Practical Approach to Real-Time Computer Graphics*, Morgan Kaufmann Publisher, 2000, p. 561.

[14] Albin, S. *The Art of Software Architecture: Design Methods and Techniques*, John Wiley & Sons, 2003, p. 312.

[15] Alexandrescu, A. *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison Wesley, 2001, p. 352.

[16] Adams, J. *Advanced Animation with DirectX*, Premier Press Game Game Development, 2003, p. 354.

[17] *Half-Life Engine*, http://www.valvesoftware.com/, 2006 05 12.

[18] *CryEngine*, http://www.crytek.de/technology/index.php?sx=cryengine, 2006 05 20.

[19] *Quake III Engine*, http://www.idsoftware.com/, 2006 05 17.

[20] *Crystal Space*, http://crystal.sourceforge.net/, 2006 05 20.

[21] *Irrlicht*, http://irrlicht.sourceforge.net/, 2006 05 17.

[22] *Ogre*, http://www.ogre3d.org/, 2004 11 20.

10. Terminų ir santrumpų žodynas

Šiame skyriuje pateikiamas darbe naudojamų dalykinės srities terminų ir santrumpų žodynas.

Virtual machine – tai programinė įranga, izoliuojanti programas, kurias naudoja vartotojas, nuo techninės įrangos.

DirectX, OpenGL – bibliotekos, skirtos apdoroti grafiką.

LUA scenarijai (angl., *LUA scripts*) – trumpos programos, kurias galima vykdyti be kompiliavimo, parašytos LUA kalba.

GPU (*Grahics Processing Unit*) – grafinis procesorius, kuriame realizuotos pradinės grafinės funkcijos, spartinančias grafinių programų veikimą, nuimmančios nereikalingą skaičiavimą nuo CPU.

CPU (*Central Processing Unit*) – pagrindinis kompiuterio procesorius.

3D (*Three Dimensional*) – trimatė erdvė.

API (*Application Programming Interface*) – programų kūrimo sąsaja.

C/C++ – programavimo kalbos.

HUD (*Heads Up Display*) – meniu, kuris rodomas virš visos kitos informacijos.

DI – dirbtinis intelektas.

UML (*Unified Modeling Language*) – standartizuota modeliavimo kalba.

SDK (*Software Development Kit*)- programinės įrangos kūrimo įrankiai, naudojami realizuoti įvairių egzistuojančių programų praplėtimus arba papildymus.

OSI (*Open System Interconnection*) – standartas nustatantis tinklo protokolų lygių funkcionalumo pasiskirstymą.

TCP/IP (*Transmission Control Protocol/ Internet Protocol*) – patikimas tinklo protokolas naudojamas informacijai perduoti.

FTP (*File Transfer Protocol*) – tinklo protokolas naudojamas rinkmenų perdavimui.

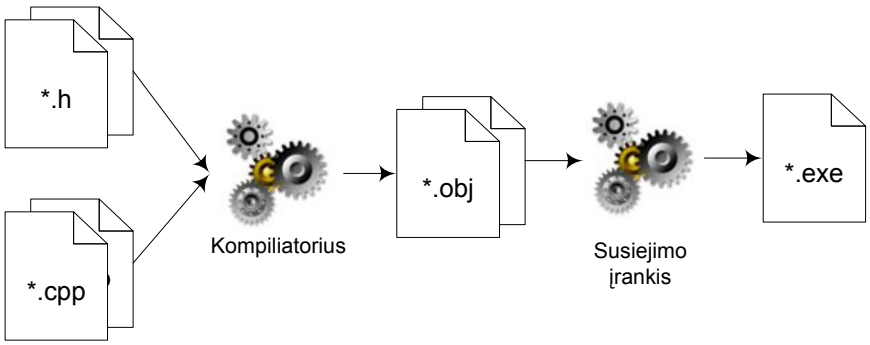
Priedas 1

Dinaminės ir statinės bibliotekos

Labai retai programas sudaro tik vienas vykdomasis failas, įprastai vykdomasis failas naudoja įvairias papildomas bibliotekas. Pagrindinis šio aprašymo tikslas supažindint su bibliotekų vaikimo princais, bei kodo pasikirstymu tarp jų.

Programos konstravimas (angl. *building*)

Norint gerai suvokti statinių ir dinaminių bibliotekų naudojimą, reikia gerai suprasti vykdomojo failo konstravimo procesą.



1 pav. Vykdomojo failo kūrimas

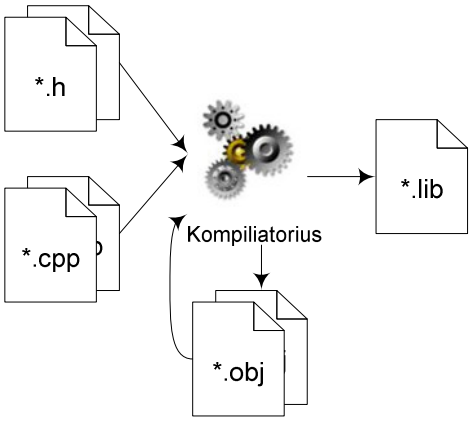
Paveikslėlyje 1 pav. pateikta vykdomojo failo konstavimas panaudojus C/C++ schema. Kiekviena C/C++ programa susideda iš antraščių failo (angl. *header*) ir kodo failų (angl. *source*). Bet koks kodo failas gali į save įtraukti neribotą kiekį antaščių failų. Programos konstavimas susideda iš dviejų etapų: kompiliavimo ir susiejimo.

Pirmojo etapo metu kodo failai kartu su visais įtrauktais antraščių failais kompiluojami į objektų (angl. *object*) failus. Objekto faile saugomas sukompiluotas išeities tekstas, tačiau jo vieta galutiniame vykdomajame faile kol kas nežinoma. Kiekvienam atskiram kodo failui yra sukuriamas atskiras objekto failas, jame yra sukompiluotas kodas iš atitinkamo kodo failo ir visų antaščių, kurios yra įtrauktos į kodo failą. Taigi išeities tekstas, kuris yra užrašytas kodo failuose, yra įkompiluojamas tik į vieną objekto failą. Savo ruoštu išeities tekstas, esantis antraščių failuose, gali būti įkompiluojamas į kelis objektų failus, atsižvelgiant į tai, kiek kartų jis buvo įtrauktas į kodo failus.

Antrojo etapo metu susiejimo įrankio pagalba, visi objekto failai susiejami į galutinį vykdomąjį failą.

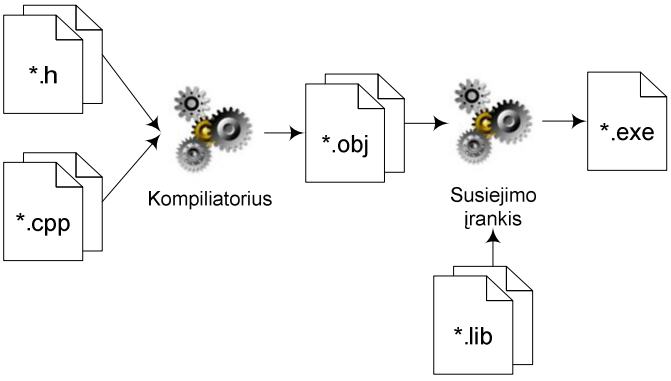
Statinės bibliotekos

Statinės bibliotekos kuriamos bibliotekos objekto failus apjungiant į vieną failą, kuris vėliau būna sujungiamas su objekto failais, kuriant galutinį vykdomąjį failą. Apjungimo procese gali dalyvauti ir kelios statinės bibliotekos.



2 pav. Statinės bibliotekos sukūrimas

Paveikslėlyje 2 pav. vaizduojamas statinės bibliotekos kūrimo procesas. Šaime procese nedalyvauja susiejimo įrankis, nes nėra kuriamas joks vykdomasis failas. Proceso rezultatas yra bibliotekos failas, kuris gali būti panaudotas kuriant galutinį vykdomąjį failą 3 pav. Kodas sikompiluotas statinėje bibliotekoje bus susietas su visais vykdomaisiai, kurie tik jį naudos.



3 pav. Statinės bibliotekos panaudojimas

Dinaminės bibliotekos

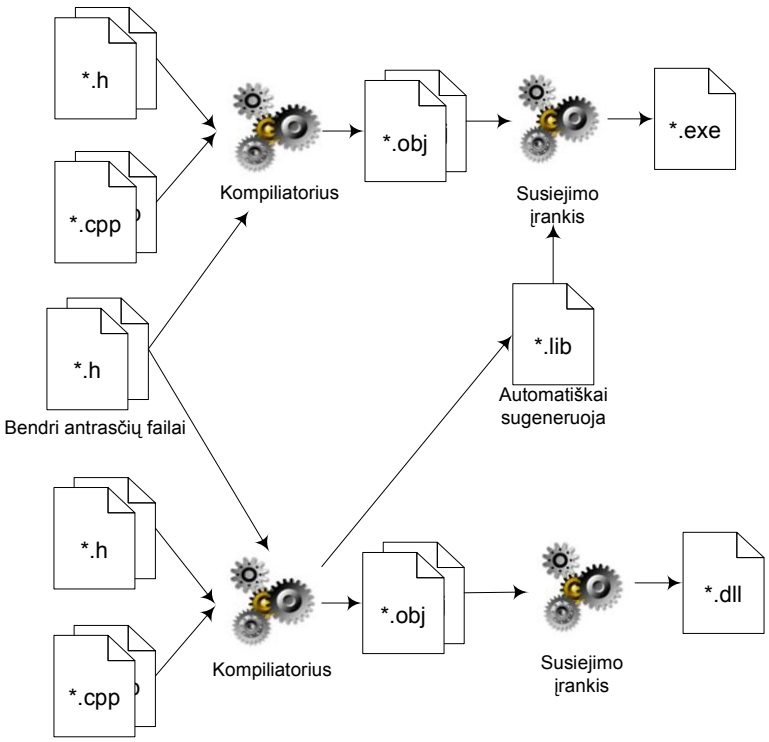
Dinaminės bibliotekos kūrimo procesas labai panašus į vykdomojo failo kūrimo procesą, esminis šio proceso skirtumas yra tas jog gaunama dinaminė biblioteka ir importavimo statinė biblioteka. Dinaminės bibliotkos naudojimas yra sudėtingesnis nei statinės. Statinės bibliotekos atveju tereikia nurodyti susiejimo įrankiui kur galima rasti atitinkama statinę bibioteką. Dinaminės bibliotekos atveju egzistuoja du panaudojimo būdai nematomas (angl. *implicit*) ir matomas (angl. *explicit*).

ms-help://MS.VSCC.2003/MS.MSDNQTR.2003APR.1033/vccore/html/_core_determine_which_linking_method_to_use.htm

Primasis variantas yra paprastesnis. Jame yra naudojama statinė biblioteka, kuri buvo sugeneruota kuriant dinaminę biblioteką. Vykdomo failo kūrimo metu ši statinė biblioteka yra susjungiama su vykdomuoju failu. Importavimo bibliotekos užduotis – vykdymo metu užkrauti dinaminę biblioteką ir atitinkamai iškviesti reikalingas funkcijas iš dinaminės bibliotekos.

Antruoju atveju sugeneruota importo biblioteka yra nenaudojama, o visas dinaminės bibliotekos užkrovimas ir metodų kvietimas yra vykdomas rankiniu būdu. Šis būdas yra sudėtingesnis, tačiau pirmasis variantas gali būti nepritaikomas dėl šių priežasčių:

- Programa išanksto nežino kokia dinaminė biblioteka bus kviečiama. Šis apribojimas neleidžia realizuoti įskiepų (angl. *plug-in*) sistemos naudojant pirmąjį metodą.
- Programa, naudojanti primąjį metodą, bus uždaroma jei ji neras atitinkamų dinaminių bibliotekų
- Programa, naudojanti pirmąjį metodą ir bendraujanti su daug dinaminių bibliotekų, gali lėtai pasileisti nes jei riekia užkrauti visas dinamines bibliotekas, kurias ji gali naudoti. Antru atveju bibliotekos gali būti užkraunamos tik jų prireikus.
- Programai, naudojančiai antrąjį metodą, nereikia susiejimo metu susisieti su importavimo statine biblioteka.



4 pav. Dinaminės bibliotekos sukūrimas ir naudojimas.

Paveikslėlyje 4 pav. vaizduojamas dinaminės bibliotekos sukūrimas ir naudojimas. Viršutinės schemos dalyje pavaizduotas vykdomojo failo kūrimas, kuris naudosis dinamine biblioteka. Naudojant nematomą dinaminės bibliotekos iškvietimą yra naudojama automatiškai sugeneruota statinė biblioteka, naudojant matomą iškvietimą statinė biblioteką yra susiejama su vykdomuoju failu.

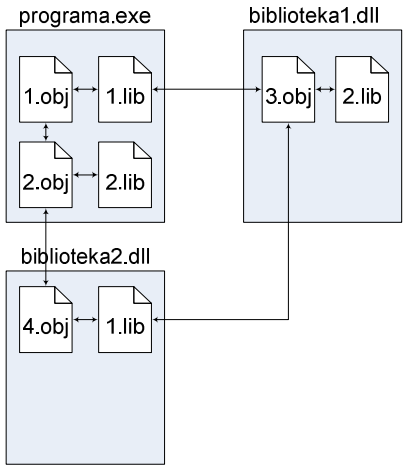
Kuriant dinaminę biblioteką, reikalingas būdas kaip nurodyti funkcijas, klases ir kintamuosius, kuruos bus galima kviesti iš išorės. Toks aprašas saugomas bendruose antraščių failuose. Šie failai yra naudojami tiek kompiluojant dinaminę biblioteką tiek ir vykdomąjį failą. Dinaminės bibliotekos kodas bus saugomas tik vienoje vietoje ir vykdomieji failai naudos tą patį bibliotekos kodą.

Kodo pasiskirstymas

Programuojant statines ir dinamines bibliotekas labai svarbu suvokti koks išeities tekstas kur atsiduria jį sukompiliavus ir susiejus. Statinės bibliotekos kodas visada yra susiejamas su vykdomuoju failu arba dinamine biblioteka. Dinaminės bibliotekos kodas saugomas atskirame dinaminės bibliotekos faile. Situacija geriau padės suprasti paveikslėlyje 5 pav. 5

pav.pateikatas pavyzdys. Jame vykdomoji programa naudoja dvi dinamines ir keleta statinių bibliotekų. „1.lib“ statinė importavimo biblioteka yra automatiškai sugeneruota „biblioteka1.dll“ dinaminės bibliotekos, pasinaudojus ja yra nematomai išvkiečiamas „biblioteka1.dll“ kodas . „2.lib“ yra atskira statinė biblioteka, kadangi ją naudoja ir „programa.exe“ ir „biblioteka1.dll“, ji yra susieta su šiais dviem failais. „biblioteka2.dll“ yra matomu būdų iškviečiama iš „programa.exe“.

Kaip galima pastebėti, dinaminė „biblioteka1.dll“ yra kviečiama iš kelių vietų, tačiau kintamieji ir kodas yra bendri visai sistemai. Tuo tarpu statinė biblioteka „2.lib“ yra atskira „programa.exe“ ir „biblioteka1.dll“ failams, taigi nors kodas ir yra tas pats, bet jis saugomas skirtingose vietose ir tarpusavyje niekaip nesusieja, tai yra pakeitimai, atlikti vykdymo metu faile „programa.exe“ saugomoje bibliotekoje „2.lib“, nebus matomi faile „biblioteka1.dll“ saugomoje bibliotekoje „2.lib“.



5 pav. Dinaminės bibliotekos sukūrimas ir naudojimas.

Priedas 2

Kietųjų kūnų ir audinių fizikinės sistemos

Besiplečiant žaidimų rinkai vis labiau stiprėja konkurencija tarp žaidimų gamintojų ir jų platintojų. Dažniausiai šią konkurencinę kovą laimi įmonės pateikiančios rinkai žaidimus, kuriuose yra pritaikytos įvairios novatoriškos idėjos. Idėjos gali būti skirtingos pradedant nuo naujo žaidimo valdymo baigiant naujo žanro sukūrimu. Vėliau tokie žaidimai tampa etaloniniais, pagal kuriuos lygiuojasi visi kiti. Visai neseniai viena iš tokių naujovių buvo fizikos simuliacija žaidimuose, kuo toliau tuo labiau ir plačiau yra plėtojama šita idėja.

Architektūra

Fizikos varikliukas įprastai susideda iš šių modulių :

1. *Susidūrimų aptikimo sistema*. Ji atsakinga už objektų tarpusavio susidūrimų aptikimą, jis paremtas skaičiuojamosios geometrijos metodais, bei tam tikrais fizikiniais dėsniais.

2. *Jėgų surinkimo sistema*. Šis modulis privalo rasti ir susumuoti visus objektą veikiančių jėgų vektorius į vieną bendrą vektorių. Kita jo užduotis rasti ir susumuoti objekto kampinius momentus, jie atsiranda kai objektą veikia jėga nukreipta ne į jo masės centrą

3. *Objektų naujų pozicijų apskaičiavimas*. Šio modulio užduotis žinant objekto kampinį momentą ir jėgą veikiančią masės centrą, rasti naują objekto padėtį ir orientaciją

Mes plačiau apžvelgsime antrą ir trečią modulius.

Jėgų surinkimo sistema

Šiame etape mums reikia surasti visas jėgas, kurios veikia visus kūnus. Jėgos aprašomos kaip vektoriai (turi kryptį ir dydį). Jėgos, veikiančios kūną, gali atsirasti dėl įvairiausių priežasčių, kaip pavyzdys tai gali buti ištempta spyruoklė prikabinta prie kūno arba jėgos atsiradusios susiduriant kūnams. Kūną veikiančias jėgas galima suskirstyti į du tipus:

- Jėgos veikiančios kūno masės centrą.
- Jėgos veikiančios ne kūno masės centrą.

Vektoriškai sudėjus pirmo ir antro tipo jėgas gauname jėgą, kuri nusako kūno padėties pokytį.

Antrojo tipo jėgos nusako kūno orientacijos pokytį. Iš šių jėgų yra skaičiuojamas kūno sukimo momentas. Kūno sukimo momentą nusako:

$$\vec{\tau} = \vec{r} \times \vec{F} \,(1),$$

kur $\vec{r}$ - vektorius einantis iš masės centro į tašką, kurį veikia jėga $\vec{F}$.

Galutinis kūno sukimo monetas skaičiuojamas vektoriškai sudedant visus kūno sukimo momentus. Pats surinkimo procesas yra gana paprastas, sudėtingiausia yra rasti ir teisingai suskaičiuoti kūną veikiančias jėgas.

Objekto naujos padėties apskaičiavimas

Šiame etape mes turime jėgą, kuri veikia kūno masės centrą, ir kūno sukimo momentą $\vec{\tau}$ laiko momentu t.

1. Kūno padėties radimas

Kaip visi žinome iš antrojo Niutono dėsnio:

$$\vec{F} = m\vec{a}\,,$$

iš čia mes galime rasti kūno pagreitį:

$$\vec{a} = \frac{\vec{F}}{m}.$$

Tačiau mums reikia rasti naują kūno padėtį laiko momentu t. Iš pirmo žvilgsnio gali pasirodyti, kad šioje situacijoje galima būtų panaudoti tolygiai greitėjančio judėjimo formulę (pagreitis išlieka pastovus $\vec{a} = const$):

$$s = s_0 + \vec{v}_0 t + \frac{\vec{a}t^2}{2}\,(2),$$

bet ši formulė mūsų atveju netinka, nes tai nėra tolygiai greitėjantis judėjimas. Tai gali įrodyti paprasčiausias pavyzdys:

Tarkime turime prie spyruoklės prikabintą kūną. Jį veikia jėga, kurią nusako Huko dėsnis:

$$\vec{F} = -k\Delta\vec{x},$$

iš čia:

$$\vec{a} = \frac{-k\Delta\vec{x}}{m}.$$

Kaip matome kūno pagreitis nėra pastovus, jis kinta priklausomai nuo spyruoklės pailgėjimo, t.y. nuo kūno padėties, todėl formulės (2) naudoti negalime.

Šioje situacijoje mums gali pasitarnauti išvestinės. Kaip žinome funkcijos $x(t)$ išvestinės reikšmė taške x_0 pagal t apibūdina funkcijos kitimo spartą. Išvestinė žymima $\frac{df}{dt}$ arba f'. Savo ruoštu funkcijos išvestinė gali turėti savo išvestinė, ši išvestinė vadinasi antros eilės išvestine ir žymima f''. Suprantama, kad egzistuoja trečios ir aukštesnių eilių išvestinės. Objekto pozicijos pokyčio spartą laike apibūdina greitis, tuo pačiu pagreitis apibūdina greičio kitimo spartą laike. Sekant šiais samprotavimais galima parašyti:

$$v = \frac{dx}{dt}, \text{ arba } v = x'$$

ir

$$a = \frac{dv}{dt} \text{ arba } a = v' = x''.$$

Tai yra fundamentalios formulės, kuriomis remiantis yra sukonstruoti visi fizikos varikliukiai. Taigi iš šių formulių matome, koks yra sąryšis tarp kūną veikiančio pagreičio a ir kūno padėties x. Pagreitis a yra tai ką mes turime ir x yra tai ko mes ieškome.

Šios formulės sudaro lygčių sistemą ir jos vadinamos diferencialinėmis lygtimis. Šios lygtys skiriasi nuo įprastinių lygčių tuo, kad sprendinys yra ne viena reikšmė, o funkcijų šeima $x(t)$. Mūsų atveju ši funkcija nusako kūno padėtį laiko momentu t. Taip pat galime pastebėti, kad pagreitis yra antros eilės kelio išvestinė, tokios lygtys vadinamos antros eilės diferencialinėmis lygtimis.

2. Kūno orientacijos radimas

Skaičiuojant naują kūno orientaciją nauju laiko momentu naudojamos šios išraiškos:

$$\vec{L} = I\vec{\omega} \quad (3),$$

kur $\vec{L}$ - kampinis kūno impulsas, $\vec{\omega}$ - kampinis kūno greitis, kurio mes ir ieškome, I- tai matrica, kuri nusako kūno masės pasiskirstymą. Šis kintamasis dažniausiai vadinamas tensoriumi. I matricos įstrižainės elementai nurodo kokia jėga reikia paveikti kūną, kad jis pasisuktų atitinkamai apie x, y, z ašis. Kiti matricos elementai nurodo, kiek kūnas pasisuks x, y, z ašimi, paveikus jėga pagal kitą kūno ašį x, y, z. Tai yra, jei mes kūną suksime pagal x ašį kiti elementai nurodys kiek kūnas pasisuks pagal z ir y ašis, atitinkamai tai galioja ir pagal kitas ašis. Tensorius priklauso nuo kūno formos (pvz. cilindras, tuščiaviduris cilindras, kubas, sfera ir t.t.), ir naudojant tam tikrus parametrus yra suskaičiuojamas išanksto.

$$\frac{d\vec{L}}{dt} = \vec{\tau} \quad (4),$$

kur $\vec{\tau}$ - sukimo momentas, kurį mes suskaičiavome jėgų surinkimo etape. Belieka išspręsti šią diferencialinę lygtį, ir galų gale

$$\vec{\omega} = \frac{\vec{L}}{I} \text{ išreiškus iš (2). (5)}$$

Diferencialinių lygčių sprendimo metodai

Egzistuoja du diferencialinių lygčių sprendimo būdai: analitinis ir skaitinis. Pirmas labai tikslus, tačiau labai nepatogus ir reikalaujantis didelio pradinio laiko sąnaudų, antras ne toks tikslus ir nereikalauja didelių laiko sąnaudų.

1. Analitinis metodas

Šis sprendimo metodas yra pats tiksliausias, tačiau tūri didelių trūkumų dėl kurių praktiškai yra nepanaudojamas.

Visų pirma rasti dėsnį (funkciją), pagal kurį kinta kūno pagreitis, yra gana sudėtinga. Tačiau net jį radus, rasti tikslų diferencialinės lygties sprendimą, įvertinus tai, kad vienų metu gali sąveikauti daugybė objektų, yra neįmanoma arba labai sudėtinga. Taip pat šiuo atveju, būtų labai nepaprasta į sistemą įvesti naujus sąveikaujančius objektus ar naujas veikiančias jėgas, tai reikalautų lygčių perskaičiavimo. Tačiau diferencialinę lygtį reiktų išspręsti tik vieną kartą, toliau, radus $x(t)$, simuliacijai reikalingi tikslūs duomenys būtu randami gana paprastai ir greitai. Bet vienas pliusas, nors ir didelis, negali atpirkti begalės minusų.

2. Skaitinis metodas

Šis sprendimo metodas nėra toks tikslus, tačiau jo pagalba galime nesunkiai rasti net ir sudėtingiausios sistemos kūno padėtį laiko momentu *t.* Pagrindinis šio sprendimo būdo skirtumas nuo prieš tai aptarto yra tas, kad mes neiškosime $x(t)$. Iš tikrųjų mums šios funkcijos ir nereikia, mums reikia šios funkcijos reikšmės tam tikru momentu *t*, kitaip tariant mus nedomina kaip kinta kūno padėtis bėgant laikui, mus domina kūno padėtis tam tikru laiko momentu. Tai mes pasiekiame pradedant nuo $t_0 = 0$ tam tikru būdu, pažingsniui artėjant prie momento *t*.

3. Metodų skirtumai

Taigi pirmu būdu mes galime rasti funkcijos reikšmę momentu *t* iškarto, antru būdu mums reikalingas tam tikras iteracijų skaičius tiesiogiai proporcingas momento *t* dydžiui. Įprastai iteracinis skaičiavimas nėra pageidaujamas, tačiau mūsų situacijoje vienu ar kitu atveju reikalingi tarpiniai skaičiavimai, kadangi mums kūno padėtį reikia atnaujinti pastoviai. Taigi iki tam tikro momento sukauptus skaičiavimus mes galime panaudoti kitoje iteracijoje, taip šis metodo trukumas mūsų situacijoje tampa nesvarbus.

Pavyzdys geriau iliustruoja šį svarbų skirtumą tarp dviejų metodų.

Tarkime turime objektą, kuris juda tam tikru dėsniu $x(t)$, žinome, jog kūnas pradiniu laiko momentu $t_0 = 0$ yra tam tikroje pradinėje padėtyje.

- Pirmu atveju mes randame $x(t)$ išsprendę diferencialinę lygtį,
- Antru atveju mes turime metodą (funkciją $F(t, a(t))$, kuri priklauso nuo laiko ir mums žinomos pagreičio funkcijos)

Norint rasti kūno padėtį laiko momentu *t*:

- Pirmu atveju įsistatome momentą *t* į funkciją $x(t)$ ir randame kūno padėtį žinodami pradinę kūno padėtį.
- Antru atvejų
  - Randamas minimalus iteracijų skaičius reikalingas pasiekti norimą momentą *t*. Ši reikšmė dažniausiai pasirenkame kaip konstanta prieš pradedant simuliaciją.
  - Kiekvienoje iteracijoje skaičiuojama funkcijos $F(t_0, a(t_0))$ reikšmę atsižvelgiant į prieš tai buvusias reikšmes, kol pasiekiame $t_n = t$ kur $t_{n+1} = t_n + \Delta t$, galutinė funkcijos reikšmė ir bus mūsų ieškoma kūno padėtis.

Taigi pagrindiniai šio metodo tikslumas ir stabilumas priklauso nuo $F(t, a(t))$ dar kitaip ši funkcija vadinama *integravimo metodu.*

4. Integravimo metodai (išreikštiniai)

Kaip minėjome anksčiau diferencialinės lygties skaitinis sprendimo stabilumas ir tikslumas pilnai priklauso nuo pasirinkto integravimo metodo, nuo integruojamos funkcijos tipo ir taip pat nuo integravimo žingsnio. Egzistuoja tam tikra priklausomybė, kurioje didinant stabilumą ir tikslumą mažėja simuliacijos greitis ir atvirkščiai, didinant sistemos greitį mažėja tikslumas ir stabilumas:

- Tikslumas nusako, kiek stipriai gauta integravimo metodo reikšmė skiriasi nuo tikrosios reikšmės, kuri būtų gaunama suskaičiavus ją iš funkcijos $x(t)$. Paklaidos atsiranda dėl keleto priežasčių:
  - Kompiuteris negali saugoti begalinio tikslumo slankaus kablelio skaičius. Ši situacija dažnai sprendžiama slankaus kablelio skaičius saugant 64 bituose (double) vietoje įprastinių 32 bitų (float), bet tai, savaime suprantama, prailgina operacijų trukmę ir saugojimui skirtą vietą, bei labiau apkrauna sistemos magistralę.
  - Dėl fiksuoto Δt dydžio ir įvairių kitų priežasčių (keletą iš jų mes aptarsim vėliau) $F(t, a(t))$ duoda rezultatą su tam tikros eilės paklaida.
- Stabilumas tai labai svarbus faktorius, kuris nusako ar paklaidos atsirandančios dėl integravimo metodo netikslumo kaupiasi ir ar tas kaupimaisi gali išvesti sistemą iš stabilios būsenos (gautos integravimo reikšmės labai stipriai skiriasi nuo tikrosios

reikšmės). Sistemai išėjus iš stabilios būsenos gali būti stebimas „sprogimas“, tai yra kai objektai pradeda judėti dideliais greičiais ir chaotiškomis trajektorijomis.

Nedidelės tikslumo problemos dažniausiai nėra kritiškos, o kartais net nepastebimos, tačiau stabilumo problemos nepastebėti neįmanoma.

Egzistuoja keletas integravimo metodų:

- Eulerio,
- Verleto,
- Rungės-Kuto.

Kiekvieno iš jų tikslas yra rasti nežinomos funkcijos $x(t)$ reikšmę taške *t*, kai žinoma šios funkcijos pirmos, atros arba aukštesnės eilės išvestinė. Dabar šiek tiek plačiau apžvelgsime kiekvieną iš šių metodų.

4.1. Eulerio metodas

Tai pats paprasčiausias, lengviausiai suprantamas ir greičiausias integravimo metodas, tačiau tuo pačiu jis yra pats nestabiliausias ir netiskliausias.

Kaip minėta anksčiau, funkcijos išvestinė apibūdina funkcijos kitimo spartą taške, taigi šio metodo esmė yra suskaičiuoti šį pokytį iš išvestinės ir tariant, kad šis pokytis yra pastovus laiko intervale Δt, gauti naują *x* reikšmę:

$$x_{n+1} = x_n + \Delta t \cdot x'(t) \quad (6).$$

Šio metodo netikslumas yra tame, kad mes tariame, kad išvestinės reikšmė nekinta tam tikrame intervale, nors iš tikrųjų funkcijos išvestinė bet kuriame taške gali būti kitokia. Nagrinėjant Verleto integravimo metodą galėsime pamatyti kokio tiksliai dydžio paklaida yra padaroma naudojant Eulerio metodą. Kaip matosi iš mūsų samprotavimų ir formulės (6), metodo tikslumas priklauso nuo Δt: kuo mažesnis Δt tuo dažniau mes skaičiuojame naują reikšmę ir tuo mažesnę darome paklaidą.

Mūsų atveju skaičiuojant kūno padėtį laiko momentu *t* Eulerio metodą reikia taikyti du kartus:

- pirmą kartą norint rasti kūno greitį,
- antrą kart norint rasti kūno padėtį.

Taigi formulės būtų:

- greičio radimui:

$$v_{n+1} = v_n + \Delta t \cdot a(t),$$

- kūno padėties radimui:

$$x_{n+1} = x_n + \Delta t \cdot v(t) \text{ arba tiksliau } x_{n+1} = x_n + \Delta t \cdot v_n .$$

Kaip matome pirmo etapo metu gautas greitis naudojamas skaičiuojant galutinę kūno padėtį antram etape, tai dar labiau pablogina metodo tikslumą, nes galutinė kūno padėtis skaičiuojama naudojant *netikslų* metodą, kuriame naudojam *netiksliai* paskaičiuotas reikšmes, gautas tuo pačiu *netiksliu* metodu. Taip pat galime pastebėti, kad kūno greitis ir kūno padėtis skaičiuojami ir kaupiami atskirai. Tai yra trūkumas, dėl kurio kūno greitis ir pagreitis gali stipriai išsiderinti tarpusavyje, savo ruoštu tai padidina metodo nestabilumą.

4.2. Rungės-Kuto metodas

Šis metodas yra tikslesnis nei Eulerio metodas. Jo formulė yra tokia:

jeigu $x' = f(t, x)$

tai $x_{n+1} = x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4),$

kur $k_1 = f(t_n, x_n),$

$$k_2 = f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1),$$

$$k_3 = f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_2),$$

$$k_4 = f(t_n + h, x_n + hk_3).$$

Šis metodas dar vadinamas ketvirtos eilės Rungės-Kuto metodu. Egzistuoja paprastesnis, tačiau mažiau tikslūs trečios ir antros eilės metodai. Antros eilės Rungės-Kuto metodo formulė būtų:

$$x_{n+1} = x_n + k_2,$$

kur

$$k_1 = f(t_n, x_n)$$

$$k_2 = f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1)$$

Išties Eulerio metodas yra pirmos eilės Rungės-Kuto metodas. Kaip minėjau Rungės-Kuto metodas yra tikslesnis nei Eulerio, tačiau šis metodas reikalauja daug daugiau skaičiavimų. Taip pat suskaičiuoti funkcijos reikšmę $f(t_n, x_n)$ kelis kartus su skirtingais argumentais gali būti labai ilgas procesas. Todėl jis labiau tinka specifiniams atvejams, kur formulės $f(t_n, x_n)$ skaičiavimas yra greitas procesas. Šis metodas taip pat kaip ir Eulerio metodo atveju turi trūkumą, jog norint gauti naują kūno padėtį reikia pirma rasti jo greitį ir tik tada kūno padėtį, su visomis iš to išplaukiančiomis pasekmėmis.

4.3. Verleto metodas

Verleto integravimo metodas pirmiausiai buvo pradėtas naudoti molekulinėje dinamikoje apskaičiuoti molekulės padėtį. tik vėliau buvo pradėtas naudoti žaidimų fizikos varikliukuose. Verleto integravimo metodas yra išvedamas iš Teiloro eilutės. Teiloro eilutė apie tašką $x = a$ yra lygi:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + ... + \frac{f^{(n)}(a)}{n!}(x-a)^n + ...$$

, kitaip tariant funkciją $f(x)$, kuri tenkina tam tikras sąlygas, galima išskleisti jos išvestinių eilute.

Taigi tarkime funkcija $x(t)$ nusako kūno padėtį laiko momentu *t*. Tada galime išskleisti šią funkciją Teiloro eilute apie tašką $x = t_0$ ir užrašyti dvi išraiškas, vieną žingsniu per Δt pirmyn ir kitą – atgal:

$$x(t_0 + \Delta t) = x(t_0) + \Delta t x'(t_0) + \frac{\Delta t^2 x''(t_0)}{2!} + \frac{\Delta t^3 x''(t_0)}{3!} + O(\Delta t^4) \quad (7)$$

ir

$$x(t_0 - \Delta t) = x(t_0) - \Delta t x'(t_0) + \frac{\Delta t^2 x''(t_0)}{2!} - \frac{\Delta t^3 x''(t_0)}{3!} + O(\Delta t^4),$$

kur $O(\Delta t^4)$ ketvirtos ir aukštesnių eilių Teiloro eilutės narei.

Sudėję šias išraiškas ir išreiškę $x(t_0 + \Delta t)$ gauname:

$$x(t_0 + \Delta t) = 2x(t_0) - x(t_0 - \Delta t) + \Delta t^2 x''(t_0) + O(\Delta t^4),$$

tai ir yra Verleto algoritmo išraiška.

Kaip galima pastebėti iš formulės joje nefigūruoja pirmoji funkcijos išvestinė, tai leidžia tiesiogiai gauti kūno padėtį iš antro laipsnio išvestinės, arba, kitaip tariant, pagreičio.

Taigi naudojant šį metodą nereikia atskirai skaičiuoti ir kaupti kūno greičio, tai yra privalumas prieš anksčiau aptartus du metodus.

Tačiau šis metodas nėra savaime prasidedantis (angl. *self starting*). Jam reikalingi tam tikri skaičiavimai prieš pradedant simuliaciją, tai yra reikia rasti $x(t_0 - \Delta t)$ kai $t_0 = 0$, kad metodas galėtų tvarkingai dirbti. Bet šiuos skaičiavimus reikia atlikti tik vieną kartą ir šis minusas nublanksta prieš gaunama pliusą: nereikia kaupti greičio.

Įdėmiau pažiūrėjus, galima pastebėti, jog Eulerio metodas yra dalinė Teiloro eilutės išraiška. Runges-Kuto metodą taip pat galima išreikšti iš Teiloro eilutės, bet tai jau nėra taip parasta. Eulerio metodas iš (7):

$$x(t_0 + \Delta t) = x(t_0) + \Delta t x'(t_0).$$

Dabar galime pamatyti kokio dydžio paklaidą daro Eulerio metodas (atmetamos antros ir aukštesniu eilių Teiloro eilučių nariai), palyginus su Verleto metodo paklaida $O(\Delta t^4)$.

Verleto metodas naudojamas kai Δt yra pastovus, tačiau jį galima pritaikyti ir kintančiam Δt.

Galima paminėti, kad kiekvieną iš šių metodų galima padaryti adaptyviu, tai yra, kad Δt kistų priklausomai nuo funkcijos savybių: padidėtų ten, kur funkcija mažai kinta, ir sumažėtų ten, kur stipriai kinta.

Audinio* vizualizacija

Turime sistemą, kuri aprašyta ankstesniuose skyriuose, galima sukurti įvairių įdomių efektų. Vienas iš tokių yra audinio simuliacija. Mus supa įvairiausi audiniai, tai tiesiog mūsų įprastiniai drabužiai, kuriuos mes dėvime kiekvieną dieną, todėl natūralu, kad mes juos

* *angl.* cloth

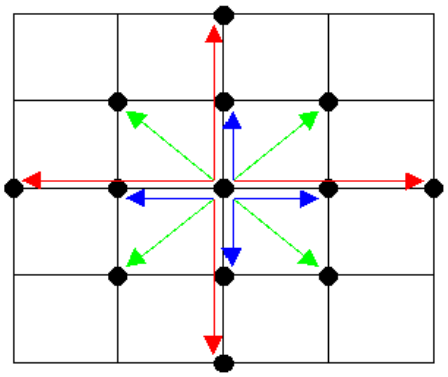
norime matyti ir žaidimuose. Jau ilgą laiką kompiuterinės grafikos paketuose menininkai (3D modeliuotojai) simuliuodavo audinius panašiu principu, kurio naudojamasi ir dabar. Vėliau atsirado specializuotos šių paketų funkcijos skirtos audiniu simuliavimui. Tačiau ir vienas ir kitas metodas turėjo vieną didelį trūkumą: buvo labai lėti ir tekdavo laukti nemažą laiko tarpą kol kompiuteris atlikdavo visus skaičiavimus. Bet laikui bėgant, kompiuteriai spartėjo ir, supaprastinus algoritmą, atsirado galimybė viską simuliuoti realiu laiku.

1. Audinio struktūra

Audinio struktūrą sudaro dalelių sistema tarpusavį sujungta spyruoklėmis. Spyruoklės aprašomos Huko dėsniu:

$$\vec{F} = -k\Delta\vec{x} - bv \quad (8)$$

Visos dalelės tolygiai pasiskirsto audinio plote. Dalelės dažniausiai nėra piešiamos, jos naudojamos tik informacijai saugoti, kuri vėliau bus naudojama audinio atvaizdavimui. Audinį ramybės būsenoje įsivaizduojame kaip popieriaus lapą. Dalelės tarpusavyje susietos trimis ryšių tipais:



6 pav. Ryšiai siejantys daleles: tempimo (mėlynas), šlyties (žalias), lenkimo (raudonas)

- *Pirmas tipas – tempimo*. Jis neleidžia audiniui perdaug išsitempti. Kaip žinome audinys dažniausiai perdaug nesitempia, todėl šio ryšio standumas dažniausiai būna pakankamai didelis. Šiuo ryšių sujungiamos keturios kaimyninės dalelės pagal vertikalę ir horizontalę.
- *Antras tipas – šlyties*. Šio ryšio dalelės dažniausiai turi mažesnį standumą. Šiuo ryšių sujungiamos keturios kaimyninės dalelės pagal įstrižaines.
- *Trečias tipas – lenkimo*. Šio tipo ryšis neleidžia audiniui susilenkti stačiu kampu. Šio ryšio standumas paprastai būna pats mažiausias. Šiuo ryšių sujungiamos

aštuonios kaimyninės dalelės pagal horizontalę, vertikalę bei įstrižaines, į visas puses praleidžiant po vieną artimiausią dalelę.

Kiekvieno tipo standumas priklauso nuo audinio tipo (pavyzdžiui ar tai bus švelnus šilkas ar standi oda). Spyruoklių ilgis pradinio laiko momentu (kai audinys yra popieriaus lapo pavidalo) bus jos ilgis ramybės būsenoje. Svarbiausia yra pirmo tipo spyruoklės, mažiau svarbi yra antro ir mažiausiai trečio tipo spyruoklės. Žaidimuose priklausomai nuo norimo pasiekti realistiškumo lygio ir turimų resursų, galima nenaudoti trečio tipo, arba netgi trečio ir antro tipo spyruoklių.

Kiekvienai spyruoklei be standumo *k* koeficiento yra nurodomas ir slopinimo koeficientas *b*, kurie naudojami skaičiuojant spyruoklės sukeliamą jėgą (8). Kiekvieną dalelę aprašo jos pozicija erdvėje. Taip pat, jei audinys yra vientisas (taip dažniausiai ir būną), visoms dalelėm yra priskiriama pastovi, vienoda masė. Bendra dalelių masė sudaro audinio masę.

2. Audinio simuliacija

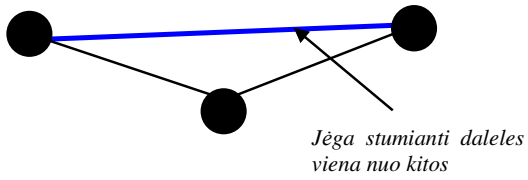
Algoritmo tikslas atnaujinti dalelės padėtį įvertinus visas dalelę veikiančias jėgas. Šis procesas yra toks pat kaip kietųjų kūnų dinamikos simuliacija, kurią mes aprašėme ankstesniuose skyriuose, tai yra:

- susidūrimų aptikimo sistema,
- jėgų surinkimo sistema
- naujos kūno padėties apskaičiavimo sistema.

Visuose šiuose etapuose įmanomos įvairios optimizacijos atsižvelgiant į tai, kad objektas sudarytas ne iš atsitiktinai išdėstytų dalelių, o dalelių ramybės būsenoje formuojančių taisyklingą struktūrą.

Audinio susidūrimo sistema su kitais objektais yra gana problematika. Visų pirma dėl to, jog dalelių kiekis gali būti gana didelis ir antra – objektų formos kartais būna gana sudėtingos. Šioje situacijos sudėtingi susidūrimo objektai dažniausiai pakeičiami paprastesniais objektais, kurie grubiai aproksimuoja sudėtingesnį savo antrininką. Tiesą pasakius, taip dažnai daroma sudėtingų objektų susidūrimams aptikti. Dar sudėtingesnis susidūrimų aptikimas yra tarp to paties audinio dalelių. Taip pat įmanomi tokie efektai kaip vėjas, kurio poveikis turėtų būti skaičiuojamas ne į atskiras daleles, o į trikampius, kurie sudaromi iš gretimų dalelių, tačiau šių ir keletą kitų efektų čia mes neaptarsime.

Jėgų surinkimo etape vidinės jėgos skaičiuojamos gana primityviai. Gali būti sunku suvokti kaip veikia lenkimo spyruoklės, tai galima būtų paaiškinti taip: tarkime mes turime tris daleles, kurios nuosekliai sukabintos spyruoklėmis ir guli vienoje tiesėje, o pirma ir paskutinė dalelė sukabinta lenkimo spyruokle, tai, jeigu mes bandysime vidurinę dalelę kelti į viršų, lenkimo spyruoklė priešinsis pirmos ir paskutinės dalelės suartėjimui, taip tarsi priešinsis kampo atsiradimui, kurį sudarys spyruoklės tarp pirmos ir antros dalelės ir spyruokle tarp antros ir trečios dalelės (2 pav.).



2 pav. Lenkimo jėga

Naujo dalelių apskaičiavimo etape, galima pasirinkti vieną iš anksčiau aptartų integravimo metodų arba bet kokį kitą metodą. Reiktų pastebėti, kad audinio simuliacijoje naudojamos spyruoklės pasižymi dideliu standumu, tai sukelia aukštesnių eilių išvestinių staigesnius pokyčius, todėl patartina naudoti kiek įmanomą tikslesnius integravimo metodus. Šiuo atveju Eulerio integravimo metodą naudoti nepatartina dėl savo pernelyg didelių paklaidų.

Egzistuoja ir trimatis audinio atvejis, tai yra dalelės išdėstomos ne plokštumoje, o trimatėje erdvėje (supaprastintas atvejis būtų kubas), taip mes galėtume gauti objektus, kurie galėtų keisti savo formas, tokie objektai dažnai atrodo kaip želė drebučiai (paprasčiausias variantas – želė kubas).

Šiais laikais vis stipriau besiplėtojant GPU (angl. *graphics processing unit*) procesorių rinkai ir panaudojimo galimybėmis, NVIDIA (viena iš kelių didžiausių GPU gamintojų) pristatė projektą kartu su išeities tekstais, kaip panaudojus GPU programavimo galimybes galima realizuoti audinio simuliaciją. Tai dar kartą įrodo kiek svarbi yra ši fizikos simuliacijos vystymosi šaka. GPU pasižymi labai dideliu paralelizmu, kas leido greičiau apdoroti didelį kiekį dalelių.

Literatūra

[1] STOKES, Sam. *Collision detection in the simulation of rigid body motion*: Robinson College, 2005, p. 60

[2] EBERLY, David. *Dynamic Collision Detection using Oriented Bounding Boxes*: Geometric Tools, Inc., 2002, p. 43

[3] MORRIS, Dan. *What the Hell is the Inertia Tensor?*

[4] ZELLER, Cyril. *Cloth*: NVIDIA Corporation, 2005.

[5] DRASCO, Steve. *The Inertia Tensor and After Dinner Tricks.* 1998

[6] VOLINO, Pascal; MAGNENAT THALMANN, Nadia. *Interactive Cloth Simulation: Problems and Solutions*: MIRALab, University of Geneva.

[7] BARAFF, David; WITKIN, Andrew. *Large Steps in Cloth Simulation*: Robotics Institute Carnegie Mellon University

[8] BOXERMAN, Eddy; ASCHER, Uri. *Decomposing Cloth*: University of British Columbia, 2004

[9] EBERLY, David H.. *Game physics*: Elsevier, 2004, p. 775

[10] LANDER, Jeff. *Lone Game Developer Battles Physics Simulator*: www.gamasutra.com, 2000.02

[11] LANDER, Jeff. *Graphic Content: Devil in the Blue-Faceted Dress: Real-time Cloth Animation*: Game Developer Magazine, 1999.05

[12] HECKER, Chris. [straipsniai]: ttp://www.d6.com/users/checker/dynamics.htm#articles

Priedas 3

Greičio tyrimo rezultatai

Atlikome savo realizuoto prototipo ir Irrlicht varikliuko (įprastinės architektūros) trimačių objektų vaizdavimo greitį.

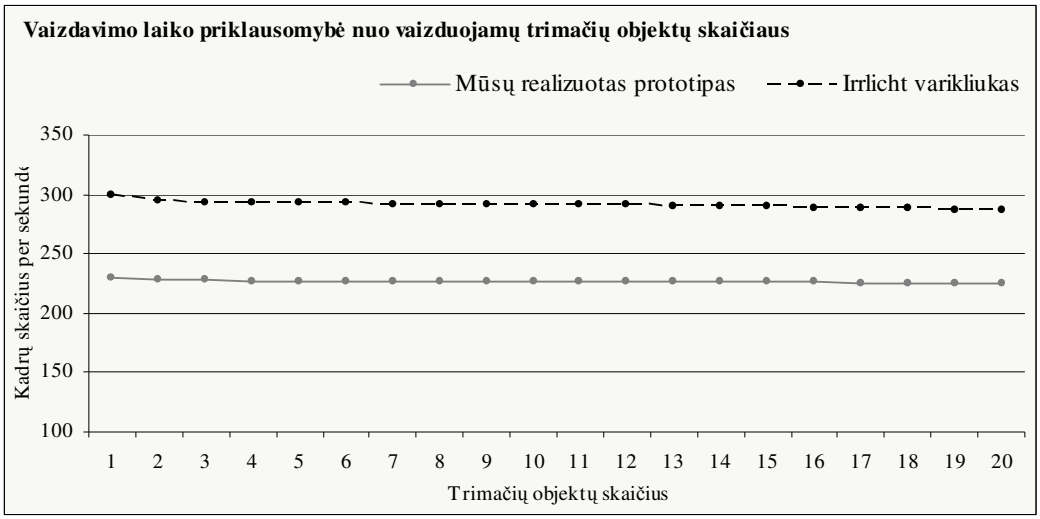
Sistemos, kurioje atlikome tyrimą, techninės charakteristikos yra tokios:

- AMD 2Ghz procesorius
- GeForce 6600 GT tipo grafinis procesorius
- 1 GB operatyviosios atminties.

Ištyrėme trimačių objektų vaizdavimo spartą naudojant skirtingų tekstūrų kiekius. Atlikome trys testus, kuriuose vaizdavome nuo 1 iki 20 trimačių objektų vienu metu. Pirmojo testo metu visiem objektams taikėme vieną tekstūrą, antrojo – keturias, trečiojo – aštuonias skirtingas tekstūras.

Pirmasis testas: 1 tekstūra visiems objektams

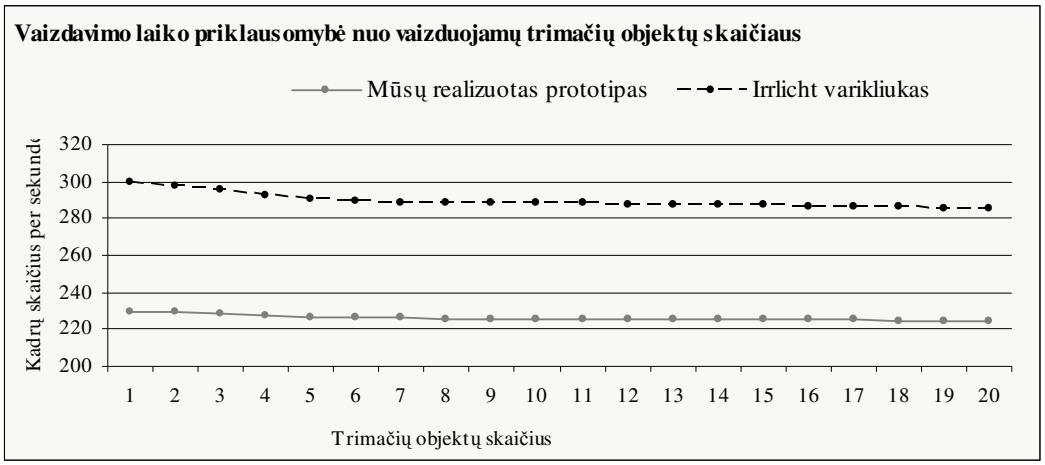
Objektų skaičius	Prototipas (kadrai per sekundę)	Irllichts (kadrai per sekundę)
1	230	300
2	228	294
3	227	293
4	227	293
5	227	293
6	227	292
7	227	292
8	227	292
9	227	292
10	227	291
11	227	291
12	226	291
13	226	290
14	226	290
15	226	289
16	226	289
17	226	289
18	225	288
19	225	287
20	225	287
	5	13
Greičio sumažėjimas	2%	4%



1 pav. Vaizdavimo laiko priklausomybė nuo vaizduojamų objektų skaičiaus, esant 4 tekstūroms

Antrasis testas: 4 tekstūros visiems objektams

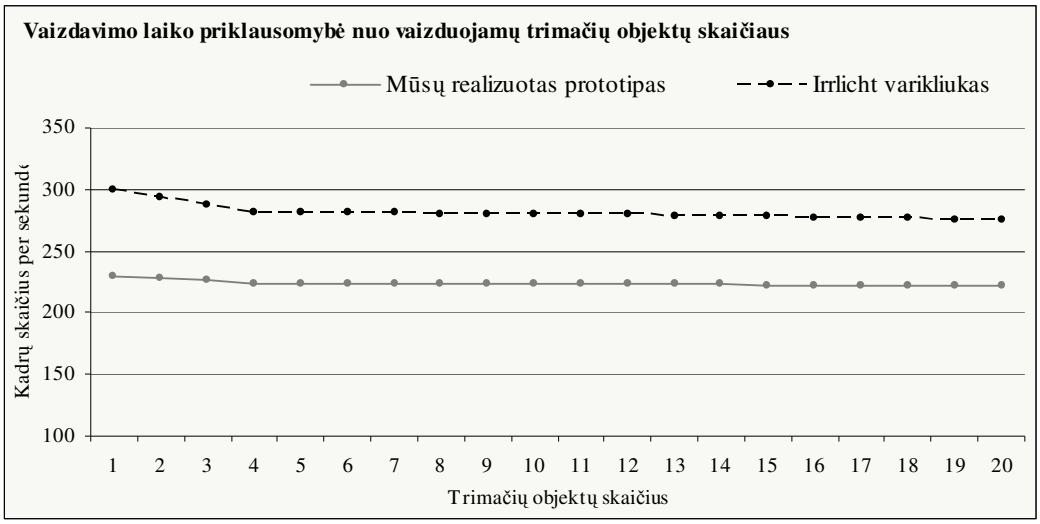
Objektų skaičius	Prototipas (kadrai per sekundę)	Irllichts (kadrai per sekundę)
1	230	300
2	229	297
3	228	295
4	227	293
5	227	291
6	226	290
7	226	289
8	226	289
9	226	289
10	226	288
11	226	288
12	226	288
13	225	288
14	225	287
15	225	287
16	225	287
17	225	286
18	225	286
19	225	286
20	225	285
	5	15
Greičio sumažėjimas	2%	5%



2 pav. Vaizdavimo laiko priklausomybė nuo vaizduojamų objektų skaičiaus, esant 4 tekstūroms

Trečiasis testas: 8 tekstūros visiems objektams

Objektų skaičius	Prototipas (kadrai per sekundę)	Irllichts (kadrai per sekundę)
1	230	300
2	228	294
3	226	287
4	224	281
5	224	281
6	224	281
7	224	281
8	223	281
9	223	280
10	223	280
11	223	280
12	223	279
13	223	279
14	223	278
15	223	278
16	222	277
17	222	277
18	222	276
19	222	276
20	222	275
	8	25
Greičio sumažėjimas	4%	8%

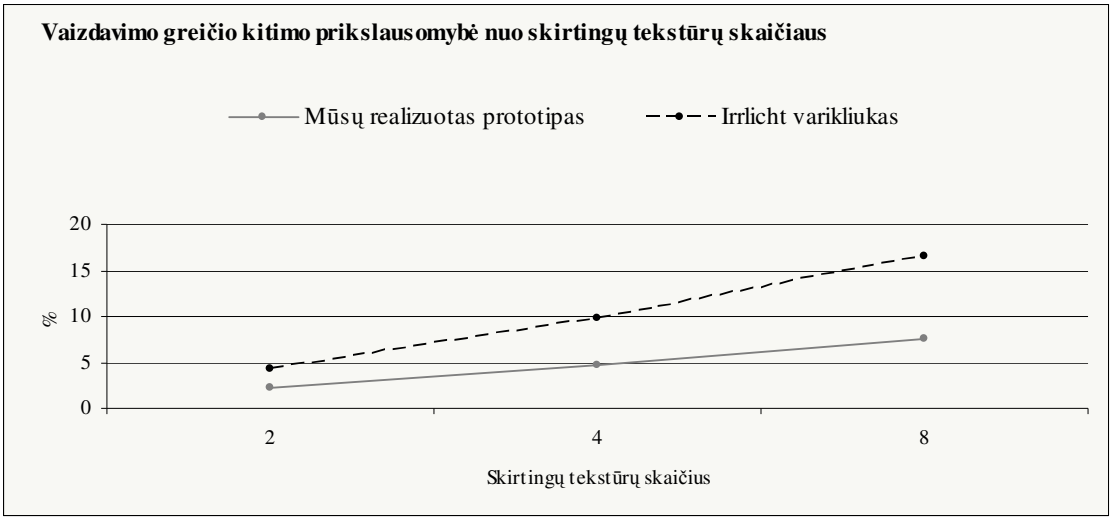


3 pav. Vaizdavimo laiko priklausomybė nuo vaizduojamų objektų skaičiaus, esant 8 tekstūroms

Apibendrinimas

Kaip matome, iš grafikų, pateiktų aukščiau, Irrlicht žaidimų varikliukas stipriai lenkia mūsų sukurtą prototipą (jo veikimo greitis – 300 kadrų per sekundę apdorojant vieną objektą, kai mūsų prototipo greitis – 230), nes mūsų prototipas nėra pilnai ištestuotas ir optimizuotas.

Tačiau kaip galima pastebėti iš paveikslėlyje žemiau, kad Irrlicht didėjant lektūrų kiekiui krinta greičiau, nei mūsų varikliuke. Tai galima paaiškinti tuo, kad mūsų duomenimis valdomos architektūros varikliukui yra didesnės galimybės duomenų srautų optimizavimui, nei įprastinės architektūros varikliukui.



4 pav. Vaizdavimo greičio kitimo priklausomybė nuo skirtingų tekstūrų skaičiaus