KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Jelena Fiodorova

Kompiuterinių žaidimų dirbtinio intelekto varikliuko uždaviniai ir jų sprendimas

Magistro darbas

Darbo vadovas

dr. Tomas Blažauskas

Kaunas, 2006

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Jelena Fiodorova

Kompiuterinių žaidimų dirbtinio intelekto varikliuko uždaviniai ir jų sprendimas

Magistro darbas

Kalbos konsultantė
Lietuvių k. katedros doc. J. Mikelionienė

2006 05 18

Vadovas
dr. Tomas Blažauskas

Recenzentas
dr. Armantas Ostreika

2006 05 18

Atliko
IFM-02 gr. stud.
Jelena Fiodorova

Kaunas, 2006

Kompiuterinių žaidimų dirbtinio intelekto varikliuko uždaviniai ir jų sprendimas

Santrauka

Žaidime dirbtinis intelektas (DI) – tai programos kodas, kuris valdo žaidimo personažus (žaidėjus, agentus), tam tikroje žaidimo pasaulio aplinkoje priverčia juos tarsi protingai reaguoti į susiklosčiusią situaciją.

Dirbtinį intelektą žaidimuose sudaro tokie uždaviniai: kelio paieška, sprendimų priėmimas, žaidėjų išskirtinių savybių generavimas, žaidimo logikos valdymas ir kiti. Tačiau šių uždavinių sprendimai labai priklauso nuo žaidimo tipo, jo scenarijaus, bendros žaidimo logikos.

Pagrindinis šio darbo tikslas – apibendrinti DI uždavinių sprendimus taip, kad juos būtų galima naudoti daugelyje įvairių žaidimų (kuriuose sprendžiami kelio paieškos ir sprendimų priėmimo uždaviniai), neatsižvelgiant į žaidimo tipą, logiką ar scenarijų.

Naudojant duomenimis valdomą detalią architektūrą realizavome sprendimų priėmimą ir kelio paiešką, taip atskirtume duomenų ir logikos lygmenis. Taip suteikėme galimybę žaidimų kūrėjams galimybę laisvai manipuliuoti duomenimis ir logika.

Atlikę savo realizuotos sprendimų priėmimo funkcijos tyrimą, nustatėme, kad mūsų sukurtas funkcijas yra paprasta naudoti, jos yra lanksčios.

The Problems and Solutions of Artificial Intelligence Engine for Games

Annotation

Game Artificial Intelligence (AI) is the code in game that makes the computer-controlled opponents (agents) to make smart decisions in game world.

There are some AI problems that are essential in games: pathfinding, decision making, generating player's characteristics, game's logic management. However these problems are gameplay dependant.

The main goal of this study – to generalize AI problems' solutions for games of many kinds, that is, to make AI solutions gameplay independent.

We have achieved this goal by using data-driven design in our solutions. We separated the game logic and the game code levels by using this approach. Such separation gave us an opportunity to manipulate game logic and data freely.

We have examined our decision making system and determined that it is flexible and is easy of use.

Raktiniai žodžiai

Dirbtinis intelektas, žaidimas, žaidimų varikliukas, kelio paieška, A* algoritmas, sprendimų priėmimas, LUA scenarijai, baigtiniai automatai, duomenimis valdoma detalioji architektūra

Key Word

Artificial Intelligence, Game, Game Engine, Path Finding, A* Algorithm, Decision Making, LUA scripts, Finite State Machines, Data-Driven Design.

Turinys

Paveikslėlių sąrašas

Lentelių sąrašas

1. Įvadas

Dirbtinis intelektas (DI) – programinė įranga, imituojanti racionalų žmogaus mastymą ir elgesį [1]. Tai akademinis požiūris į dirbtinį intelektą.

Žaidime dirbtinis intelektas – tai programos kodas, kuris priverčia kompiuteriu valdomus personažus (žaidėjus, agentus) priimti protingus sprendimus, esant tam tikroms aplinkybėms žaidimo pasaulyje [3]. Kitaip tariant, iš galimų sprendimų aibės pasirinkamas sprendimas, labiausiai tinkantis esančiai situacijai.

Skirtingai nuo akademinio DI, žaidimuose dirbtinis intelektas yra į rezultatus orientuotas [3]. Žaidėjui nesvarbu, kaip *mąstys* kompiuterių valdomi personažai, jam įdomu, kaip jie *pasielgs* – svarbu rezultatas.

Dažnai terminas dirbtinis intelektas naudojamas apibūdinti bet kokiai žaidimo reakcijai į žaidėjo veiksmus, pavyzdžiui, tam tikros animacijos vaizdavimas [3]. Kartais netgi tokie algoritmai, kaip personažo pozicijos pakeitimas ar susidūrimų aptikimas, yra vadinami DI algoritmais [3]. Dėl to, kalbant apie DI, svarbu tiksliai apibrėžti apie kokius dirbtinio intelekto algoritmus bus kalbama.

Šiame darbe apžvelgėme siaurą žaidimo dirbtinio intelekto sritį – agento tipo elgesį aprašančius algoritmus.

Agentas – virtualus personažas žaidimo pasaulyje. DI agentai – tai aktyvūs (besipriešinantys žaidėjui) kompiuteriu valdomi personažai (pavyzdžiui, monstrai, pavieniai kareiviai), arba nesipriešinantys (angl. *Nonplaying Character*) veikėjai (pavyzdžiui, ginklų pardavėjas, kurio užduotis žaidime – pasakyti nurodyto ginklo kainą ir paduoti tam tikrą ginklą žaidėjui), arba pagalbiniai veikėjai (pavyzdžiui, animuoti gyvūnai) [1].

1.1. Pagrindimas

Įdomus žaidimo dirbtinis intelektas – tai vienas iš pagrindinių sėkmingo žaidimo komponentų [1]. Realizuoti intelektą, sugebantį optimaliai suskaičiuoti kelią iki priešo ar taikliai jį nušauti, nėra sunku, tačiau žaidėjui varžytis su tokiu virtualiu priešu neįdomu [2]. Tokiu atveju derėtų sudaryti realistiškumo pojūtį ir intelektą sukurti mažiau erzinantį – klystantį. Iš kitos pusės, mes nenorime kurti kvailo intelekto tik dėl realistiškumo.

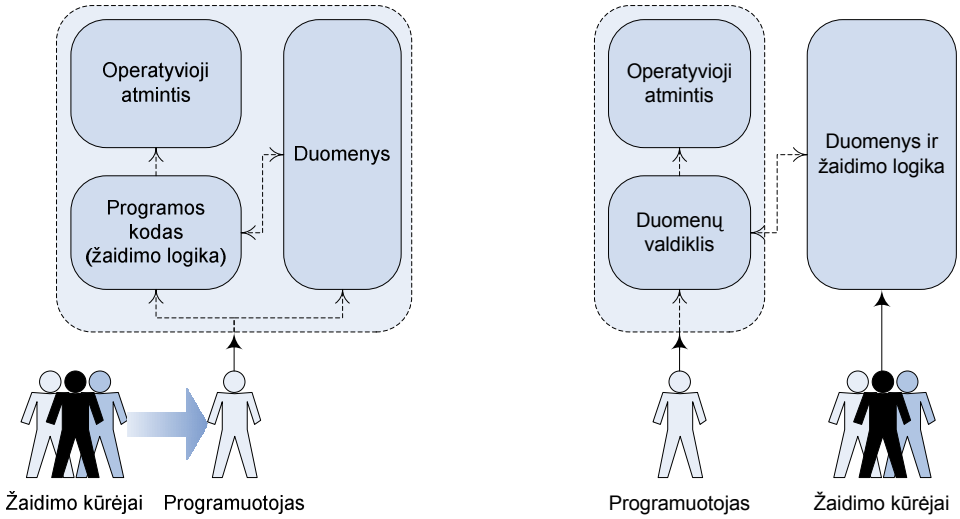
Žaidimas turi iššaukti, sudominti ir pritraukti žaidėją. Taigi, DI turi išlaikyti pusiausvyrą tarp sofistiško, išvystyto ir daugiau ar mažiau žmogiško (neoptimaliai suvokiančio pasaulį) elgesio generavimo [1].

1.2. Tikslai

Dirbtinį intelektą žaidimuose sudaro tokie uždaviniai: kelio paieška, sprendimų priėmimas, žaidėjų išskirtinių savybių generavimas, žaidimo logikos valdymas ir kiti. Tačiau šių uždavinių sprendimai labai priklauso nuo žaidimo tipo, jo scenarijaus, bendros žaidimo logikos.

Taigi, pagrindinis šio darbo tikslas – apibendrinti DI uždavinių sprendimus taip, kad juos būtų galima naudoti daugelyje įvairių žaidimų (kuriuose sprendžiami kelio paieškos ir sprendimų priėmimo uždaviniai), neatsižvelgiant į žaidimo tipą, logiką ar scenarijų. Kitaip tariant, uždavinių sprendimai turi būti tinkantys veiksmo (angl. *Action Games*), nuotykių (angl. *Adventure Games*), realaus laiko strateginiams (angl. *Real Time Strategy*) žaidimams, taip pat žaidimams vaidmenimis (angl. *Role Playing Games*).

Šiame darbe nustatytą tikslą pasiekėme, naudojant duomenimis valdomą detalią architektūrą. Sprendimų priėmimo ir kelio paieškos uždaviniams pasirinkome ir modifikavome sprendimo metodus taip, kad atskirtume duomenų ir logikos lygmenis.



A. Duomenys ir logika realizuoti sistemoje B. Duomenys ir logika atskirti nuo valdymo

1 pav. Duomenų vieta sistemos atžvilgiu

Paveikslėlyje *1 pav.* vaizduojamos dvi sistemos: A – duomenys ir žaidimo logika aprašyti kartu su programos kodu, B – duomenys ir logika atskirti nuo valdymo. Savo darbe naudojome duomenų atskyrimą, kuris vaizduojamas paveikslėlyje *1 pav. B.*

1.3. Dokumento struktūra

Šis dokumentas susidaro iš keturių pagrindinių skyrių:

- Skyriuje *2 Apžvalga* pateikėme knygų, kuriuose nagrinėjamos žaidimų DI problemos, aprašymus. Taip pat apžvelgėme esančius žaidimų varikliukus ir juose realizuotus DI uždavinius.

- Skyriuje *3 DI varikliuko pagrindinių uždavinių sprendimo metodai* aprašėme dažniausiai naudojamus sprendimų priėmimo ir kelio paieškos metodus, ir šių metodų palyginimus. Taip pat pateikėme LUA ir PYHTON scenarijų kalbų trumpus aprašymus ir šių dviejų kalbų palyginimą.

- Skyriuje *4 DI vaikiuko pagrindinių uždavinių realizavimas* detaliai aprašėme kelio paieškos ir sprendimo priėmimo žaidimuose metodų realizaciją, pagrįsta duomenimis valdoma detaliąja architektūra.

- Skyriuje *5 Eksperimentinis tyrimas* palyginome įprastinių būdu (projektavimo šablonais) realizuoto sprendimo priėmimo programą ir mūsų sprendimų priėmimo programą (kurioje naudojome duomenimis valdomą detaliąją architektūrą).

2. Apžvalga

Nedaug yra parašyta knygų apie žaidimų programavimą, bet dar mažiau apie DI programavimą [3]. Pirmame šio skyriaus poskyryje apžvelgiama užsienio bei Lietuvos literatūra. Pateikiamas pagrindinių šaltinių sąrašas su turinio santrumpomis.

Antrame poskyryje apžvelgiami žaidimų rinkoje esantys žaidimų varikliukai. Supažindinama, kokios DI funkcijos dažniausiai būna realizuotos žaidimų varikliukuose.

Trečiame poskyryje pateikiamos pagrindinių DI varikliuko uždavinių formuluotės.

2.1. Žaidimų DI užsienio ir Lietuvos literatūroje

2.1.1. Užsienio literatūros apžvalga

Kaip jau buvo minėta, apie žaidimų dirbtinio intelekto kūrimą yra parašytos tik kelios knygos. Šiame poskyryje yra apžvelgiamos knygos bei kiti informacijos šaltiniai, kuriuose skirtas dėmesys žaidimų dirbtiniam intelektui.

2.1.1.1. Knygos apie dirbtinį intelektą žaidimuose

2002 metais išleistoje knygoje „DI žaidimų programavimo išmintis“ (*AI Game Development Wisdom*) surinkta daugiau nei 70 straipsnių apie dirbtinį intelektą žaidimuose. Straipsniai apima tokias DI kryptis, kaip kelio paieška, taktinis grupės žaidimas ir judėjimas, sprendimų priėmimas. Straipsniuose pateiktos konkrečios problemos ir jų sprendimai su pavyzdžiais iš realių žaidimų.

2004 metais buvo išleistas šios knygos tęsinys – „DI žaidimų programavimo išmintis 2“ (*AI Game Development Wisdom 2*). Šioje knygoje dar daugiau straipsnių apie DI problemas ir jų sprendimus. Vienas knygos skyrelis skirtas apsimokymo algoritmams, apžvelgiamos šių algoritmų naudojimo realiuose žaidimuose perspektyvos.

Taip pat nemažai dėmesio DI uždaviniams yra skirta knygų serijoje „Žaidimų programavimo brangakmeniai“ (*Game Programming Gems*), kurios pirmoji knyga buvo išleista 2000 metais. Kiekvienoje šios serijos knygų yra po vieną skyrių apie DI programavimą žaidimams. Šiose knygose be DI uždavinių sprendimų, yra pateikiami pasiūlymai DI optimizavimo srityje.

„DI žaidimų varikliuko programavimas“ (*AI Game Engine Programming*) – dar viena svarbi knyga apie žaidimų DI. Šioje knygoje pateikiamas DI uždaviniai ir aprašomi jų sprendimai, atsižvelgiant į skirtingus žaidimų tipus, analizuojama bendrinė DI modulio architektūra ir pagrindiniai DI varikliuko uždaviniai.

Visos minėtos knygos – pagrindiniai šio darbo informacijos šaltiniai.

2.1.1.2. Internetinės svetainės apie DI žaidimuose

Nemažai straipsnių, mokomosios ir metodinės medžiagos bei kitų publikacijų apie žaidimų DI galima rasti internete. Didelis medžiagos apie DI archyvas yra sukauptas internetinėse svetainėse „AI depot“ (http://ai-depot.com) ir „Game AI“ (http://www.gameai.com). Čia galima rasti naujausią informaciją apie žaidimų dirbtinį intelektą, DI problemų analizę ir sprendimų paaiškinimus. Šiose svetainėse be gausybės straipsnių apie įvairius DI kūrimo aspektus ir mokomosios medžiagos, taip pat yra pateikiamos esančių žaidimų DI recenzijos ir apžvalgos.

Didžiausiose interneto svetainėse apie žaidimų kūrimą taip pat galima rasti daug informacijos apie DI. Tokiose žaidimų kūrimo svetainėse, kaip „Game Development“ (http://www.gamedev.net) ir „Gamasutra“ (http://www.gamasutra.com) yra DI skiltys, kuriose publikuojami straipsniai apie žaidimų DI kūrimą.

2.1.1.3. Kiti informacijos apie DI žaidimuose šaltiniai

Yra sukurta daug atviro kodo žaidimų varikliukų, kurie galėtų būti pavyzdžiu žaidimų varikliukų kūrėjams, tačiau dažniausiai jie būna silpnai dokumentuoti.

2.1.2. Lietuvos literatūros apžvalga

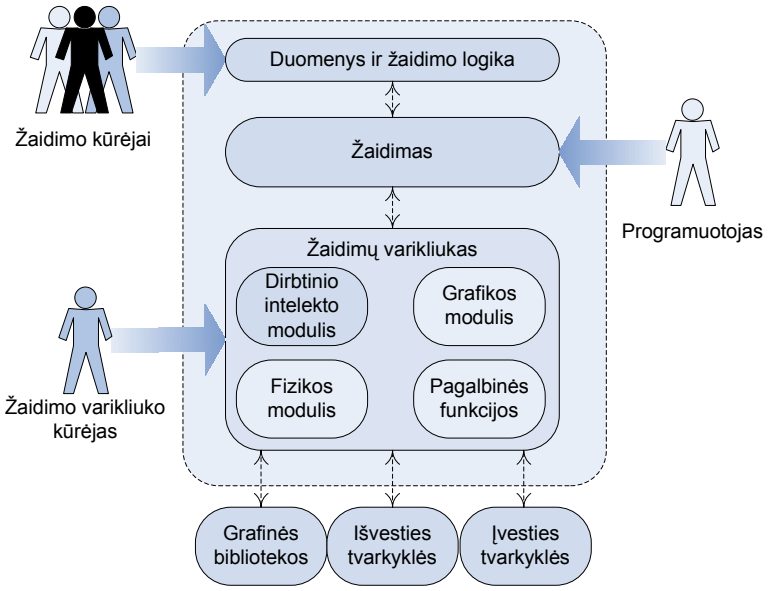
Lietuvoje yra tik kelios įmonės, kuriančios žaidimus, tačiau sukurtų žaidimų varikliukų viešai jos neplatina, kaip ir neskelbia informacijos apie žaidimų kūrimą.

Lietuvių kalba nėra išleista knygų, netgi nerašomi straipsniai žaidimų kūrimo tema.

Tačiau yra kelios prieigos internete, sukurtos mėgėjų. Didžiausia iš jų – „GameDev“ http://www.gamedev.lt, deja, ir šioje svetainėje nėra užsiminama apie žaidimų dirbtinį intelektą.

2.2. Žaidimų varikliuko apibrėžimas

Žaidimo varikliukas yra tarpininkas tarp žaidimo programinės įrangos, žaidimo kūrėjų ir grafinių bibliotekų bei įvesties/išvesties įrenginių. Žaidimo varikliuko veiklos kontekstas vaizduojamas paveikslėlyje *2 pav.*



2 pav. Žaidimo, naudojančio žaidimų varikliuką, komponentų išdėstymas

Žaidimų varikliukas – tai įvairių pagalbinių specializuotų įrankių visuma, skirta žaidimų kūrimui palengvinti. Sistema susideda iš:

- pagrindinių modulių: grafikos, fizikos ir dirbtinio intelekto;
- pagalbinių funkcijų rinkinio: saugojimo/paleidimo sistema, atminties valdymo modulis ir kitos funkcijos;
- pagalbinių įrankių: modelių eksportavimo ir importavimo, lygių kūrimo, šrifto kūrimo.

2.3. DI funkcionalumas esančiuose varikliukuose

Atlikome esančių rinkoje žaidimų varikliukų apžvalgą. Lentelėje 1 pateikti apžvalgos rezultatai, akcentuojamos dirbtinio intelekto funkcijos žaidimuose. (Apie šią apžvalgą detaliau skaitykite priede 6).

1 lentelė. Žaidimų varikliukų dirbinio intelekto funkcijos

Vieta	Pavadinimas	Sprendimų priėmimas	Kelio paieška
komerciniai			
1	Torque Game Engine	BA	
2	TV3D SDK 6	-	-
3	3DGameStudio	BA	+
4	Reality Engine	BA	+
5	Cipher	-	-
6	Deep Creator	-	-
7	3Impact	-	-
8	C4 Engine	-	-
9	3D Rad	-	+
10	AgentFX	-	-
atviro kodo			
1	Ogre	-	-
2	Crystal Space	-	-
3	Irrlicht	-	-
4	jME	-	-
5	Panda 3D	BA	-
6	Reality Factory	+	+
7	RealmForge GDK	NT, BA	+
8	The Nebula Device 3	-	-
9	OpenSceneGraph	-	-
10	Axiom	-	-

* BA – baigtiniai automatai; NT – neuroniniai tinklai

Apžvalga parengta pagal „*3D Engines Database“* interneto svetainės duomenis [9].

2.4. DI varikliuko pagrindinių uždavinių formulavimas, apžvalga

Iš apžvalgos, pateiktos skyrelyje *2.3 „DI funkcionalumo esančiuose varikliukuose apžvalga“*, galima pastebėti, kad dažniausiai varikliukuose realizuojamos DI funkcijos yra sprendimų priėmimas bei kelio paieška.

Šios funkcijos gali būti pritaikytos žaidime neatsižvelgiant į jo tipą.

3. DI varikliuko pagrindinių uždavinių sprendimų metodai

Šiame skyriuje apžvelgiami pagrindinių DI varikliuko uždavinių (kelio paieškos ir sprendimų priėmimo) sprendimai.

3.1. Kelio paieškos metodai

Veiksmingas ir našus kelio radimas yra dažniausiai pasitaikantis veiksmas, kurį vykdo kiekvienas žaidėjas, neatsižvelgiant į personažo paskirtį ar žaidimo tipą [4].

Judėjimas žaidimo pasaulyje gali būtis suskirstytas į dvi dalis: judėjimas atvirame lauke ir judėjimas patalpoje. Karo imitaciniuose žaidimuose kariai turi pasiekti priešą, apeinant įvairias kliūtis atviroje vietovėje. Personažai pirmojo asmens žaidimuose turi sugebėti pabėgti arba priartėti prie priešo pastatuose ar požemiuose. Skirtumas yra tik atstumuose, tačiau dažniausiai šios dvi dalys būna labai atskirtos, nes taip yra patogiau [5].

3.1.1. Įvairūs algoritmai

Trumpiausią kelią žaidime galima rasti keliais būdais. Šiame skyriuje apžvelgiami žaidimuose dažniausiai naudojami kelio paieškos metodai.

Deikstros algoritmas

Pagal Deikstros (angl. *Dijkstra*) algoritmą žemėlapio celės pradedamos tirti nuo kelio pradžios. Aplankomos visos artimiausios dar neaplankytos celės ir įtraukiamos į aplankytųjų celių sąrašą. Taip daroma tol, kol pasiekiama tikslo celė.

Šis algoritmas užtikrina, kad bus rastas trumpiausias kelias tarp dviejų celių.

BSF algoritmas

BFS (angl. *Best First Search*) – veikia panašiai kaip ir Deikstros algoritmas, tačiau rinkdamasis naują viršūnę, renkasi viršūnę, artimesnę tikslo viršūnei (skirtingai nuo Deikstros, kur yra parenkama viršūnė, artimesnė pradinei).

Šis algoritmas yra daug greitesnis už Deikstros algoritmą, nes naudoja euristinę funkciją, viršūnės įvertinimui (negali būti tiksliai žinoma kuri viršūnė yra artimesnė tikslui) [18]. Tačiau BSF algoritmas neužtikrina trumpiausio kelio radimo [18].

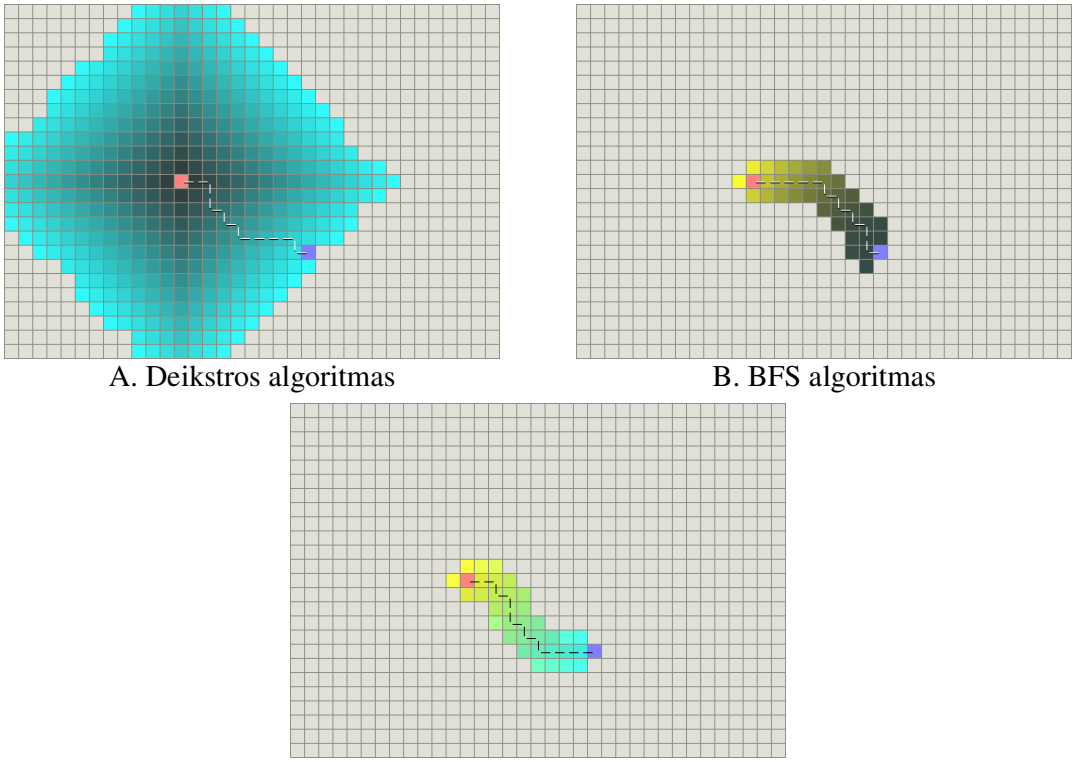
A* algoritmas

A* algoritmas tai BSF ir Deikstros algoritmų sąjunga: kaip Deikstros algoritmas užtikrina trumpiausio kelio radimą ir kaip BSF algoritmas naudoja euristines funkcijas žemėlapio celėms įvertinti [18].

Šis algoritmas yra dažnai naudojamas žaidimuose, nes yra labai lankstus (galima sukurti įvairias celių įvertinimo funkcijas) ir gali būti panaudotas skirtingiems uždaviniams spręsti [18]. Apie A* algoritmą detaliau skaitykite šio dokumento skyriuje *3.1.2 A* algoritmas*.

Deikstros, BSF ir A* algoritmų palyginimas

Tyrinėjant BSF ir A* algoritmų veikimą žemėlapyje be kliūčių, iš paveikslėlyje *3 pav. B* ir *C* pateiktų žemėlapių vaizdų galima pastebėti, kad BSF ir A* veikia labai panašiai [18]. Deikstros algoritmas (jo veikimas vaizduojamas *3 pav. A* paveikslėlyje) smarkiai atsilieka, jo veikimo metu yra apeinama žymiai daugiau celių [18].



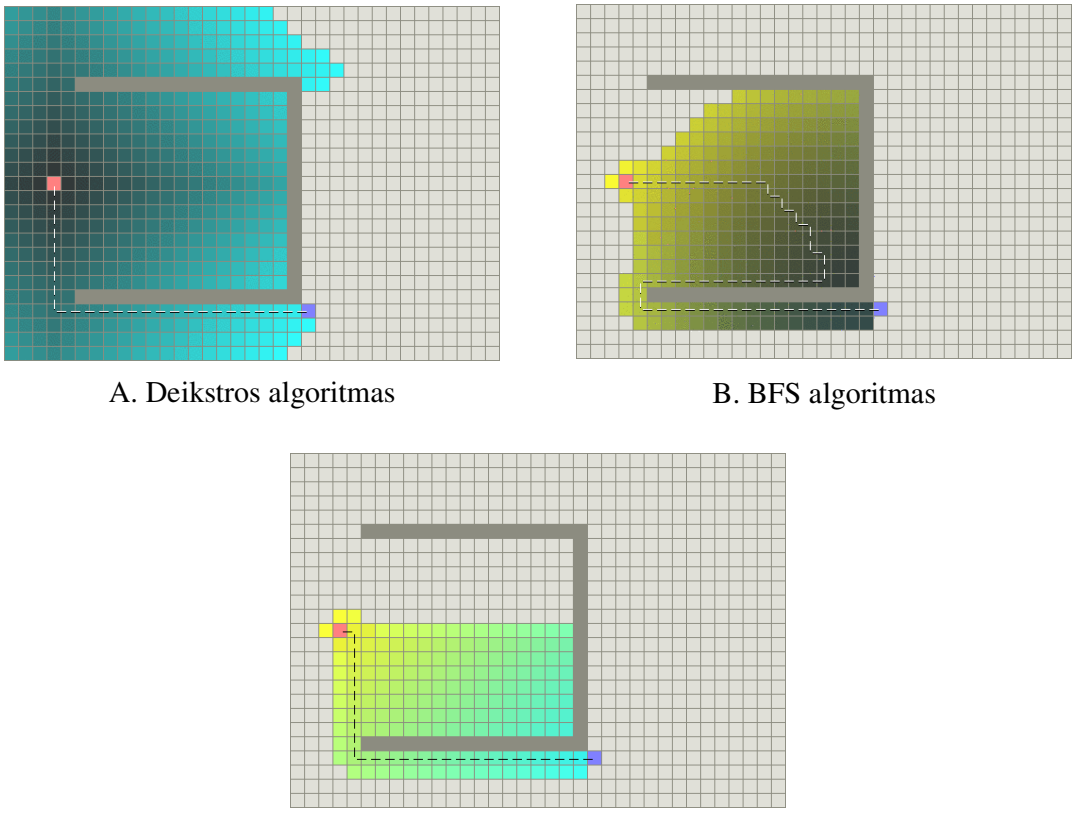
A. Deikstros algoritmas

B. BFS algoritmas

C. A* algoritmas

3 pav. Trumpiausio kelio radimas be kliūčių Deikstra, BFS ir A algoritmais [18]*

Tyrinėjant Deikstros, BSF ir A* algoritmų veikimą žemėlapyje su kliūtimis, iš paveikslėlyje *4 pav. A, B* ir *C* pateiktų žemėlapių vaizdų galima pastebėti, kad Deikstros algoritmu yra patikrinama labai daug celių [18].



A. Deikstros algoritmas

B. BFS algoritmas

C. A* algoritmas

4 pav. Trumpiausio kelio radimas su kliūtimi Deikstra, BFS ir A algoritmais [18]*

BSF algoritmu kelias surandamas greičiau nei Deikstros algoritmu, tačiau BFS rastas kelias nėra geras (*4 pav.*). Iš šių trijų algoritmų žaidimams labiausias tinkamas yra A* algoritmas, nes juo veikimo metu yra surandamas trumpiausias kelias ir jo paieškos metu patikrinama mažai celių [18].

3.1.2. A* algoritmas

A* algoritmas nuo 1968 metų yra naudojamas įvairioms problemoms spręsti [6]. Šiuo metu yra labai dažnai taikomas žaidimų kūrime: tai vienas iš pagrindinių kelio paieškos algoritmų žaidimuose. A* algoritmas yra naudojamas trumpiausiam keliui tarp dviejų taškų (jei toks kelias egzistuoja) rasti. Šiame darbe bus apžvelgti A* algoritmo pagrindiniai principai.

Naudojami terminai:

- **Žemėlapis** – sritis, kurioje yra surandamas kelias tarp dviejų pozicijų.
- **Mazgas** – struktūra, kuri atstovauja vieną poziciją žemėlapyje. Mazge saugoma kelio paieškos eigos informacija. Tą pačią poziciją žemėlapyje gali atstovauti du ir daugiau mazgų.
- **Svoris** – mazgo įvertinimas. Ieškant kelio, reikia įvertinti daugybę faktorių (kaina, laikas, naudojama energija ir pan.). Svoris – tai suminis visų kelio faktorių dydis. Kelio paieškos uždavinys – surasti kelią, turintį mažiausią svorį.

3.1.2.1. Algoritmas

Kelio paieškos metu, naudojant A* algoritmą, žemėlapis yra pateikiamas mazgais, kurie atstovauja poziciją žemėlapyje. Tačiau, net tik pozicija yra saugoma mazge, dar yra trys papildomi atributai: *f* – tinkamumo (angl. *fitness*), *g* – tikslo (angl. *goal*) ir *h* – euristinė (angl. *heuristic*) reikšmės [7]:

- *g* – tai mazgų nuo pradinio iki šio svorių suma.
- *h* – apytikris, numatomas mazgų svoris iki tikslo. Kadangi kelio svoris iki tikslo nėra žinomas, yra tik spėjimas, todėl ši dedamoji vadinama euristine.
- *f* – tai *g* ir *h* dedamųjų suma. Tai geriausias nuspėjamas kelio, kuris eina per šį mazgą, svoris.

Šių atributų tikslas – įvertinti, kiek tikėtinas kelias yra iki dabartinio mazgo. Komponentė *g* – tai, ką mes galime tiksliai suskaičiuoti, kadangi tai kelio iki šio mazgo (visų mazgų nuo pradinio iki šio svorių suma) svoris. Komponentė *h* – yra spėjama reikšmė, mes nežinome koks bus kelio nuo šio mazgo iki tikslo svoris. Kuo tikslesnis bus mūsų spėjimas, tuo greičiau bus rastas trumpiausiais kelias [7].

Papildomai algoritmui reikia dviejų sąrašų:

- *Naudotinų mazgų sąrašas* – susidaro iš mazgų, kurie dar nebuvo apžvelgti,
- *Nenaudotinų mazgų sąrašas* – susidaro iš jau apžvelgtų mazgų.

Mazgą vadiname apžvelgtu tada, kai patikriname visus mazgus, sujungtus su šiuo mazgu, suskaičiuojame jų atributus (*f*, *g* ir *h*) ir įtraukiame juos į *Naudotinų mazgų sąrašą* [7].

Kelio paieškos pradžioje *Nenaudotinų mazgų* sąrašas yra tuščias, o *Naudotinų mazgų sąraše* yra tik vienas pradžios mazgas. Kiekvienos iteracijos metu mazgai išimami iš

Naudotinų mazgų sąrašo ir yra apžvelgiami. Apžvelgti mazgai įtraukiami į *Nenaudotinų mazgų sąrašą* [6].

A* algoritmą galima užrašyti taip [7]:

1. Tegul *P* = kelio pradžia.
2. Priskirti *f*, *g* ir *h* reikšmes mazgui *P*.
3. Įtraukti *P* į *Naudotinų mazgų sąrašą*. Šiuo metų *P* yra vienintelis mazgas *Naudotinų mazgų sąraše.*
4. Tegul *B* = „geriausias“ mazgas iš *Naudotinų mazgų sąrašo* (geriausias mazgas – tai mazgas su mažiausia dedamosios *f* reikšme).
    a. Jei *B* == paskirties mazgas, sustoti. Kelias rastas.
    b. Jei *Naudotinų mazgų sąrašas* == tuščias, išeiti. Kelias negali būti rastas.
5. Tegul *C* = mazgas, sujungtas su mazgu *B,* kuriame nėra kliūties.
    a. Priskirti *f*, *g* ir *h* reikšmes mazgui *C*.
    b. Patikrinti, ar mazgas yra *Naudotinų mazgų sąraše* arba *Nenaudotinų mazgų sąraše*.
        i. Jei yra viename iš šių sąrašų, patikrinti, ar naujas kelias yra efektyvesnis (su mažesne dedamosios *f* reikšme).
            1. Jei taip, atnaujinti kelią.
        ii. Jei mazgo sąrašuose nebuvo, įtraukti mazgą *C* į *Naudotinų mazgų sąrašą*.
    c. Kartoti žingsnį 5 visiems mazgams, sujungtiems su *B.*
6. Perkelti mazgą *B* iš *Naudotinų mazgų sąrašo* į *Nenaudotinų mazgų sąrašą*. Kartoti nuo 4 žingsnio.

3.1.2.2. A algoritmo sudėtingumas*

A* algoritmo sudėtingumas priklauso nuo euristinio metodo, pagal kurį spėjama *h* dedamoji. Algoritmo sudėtingumas yra polinominis tada, kai spėjamas euristinis svoris *h* tenkina sąlygą (1):

$$\left|h(x) - h^*(x)\right| \le \mathrm{O}(\log h^*(x)); \qquad (1)$$

čia h^* - optimalus (tikslus) kelio svoris nuo x mazgo iki tikslo.

Blogiausioje situacijoje (kai nuo pradžios iki tikslo nėra kelio) algoritmo sudėtingumas yra eksponentinis (e^n) [8].

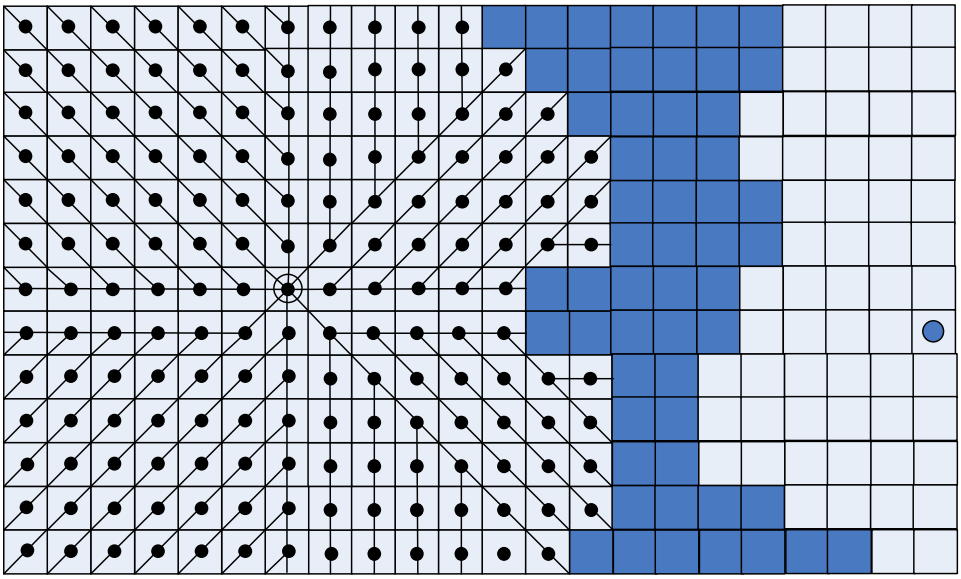
3.1.2.3. Silpnosios algoritmo vietos

Nors A* algoritmas yra geriausias kelio paieškos algoritmas, tačiau jis turi būti naudojamas atsargiai, kitaip resursai gali būti buti naudojami neoptimaliai.

Dideliems žemėlapiams šimtai ir net tūkstančiai mazgų gali būti saugomi *Naudotinų mazgų* ir *Nenaudotinų mazgų sąrašuose*. Tokie sąrašai naudos daugiau atminties nei yra leistina. Be to, A* algoritmas gali užimti visą procesoriaus laiką [6].

Sistemos resursai yra labiausiai naudojami tada, kai kelio tarp dviejų pozicijų nėra. Tokioje situacijoje yra peržiūrimi visi galimi keliai nuo pradžios mazgo.

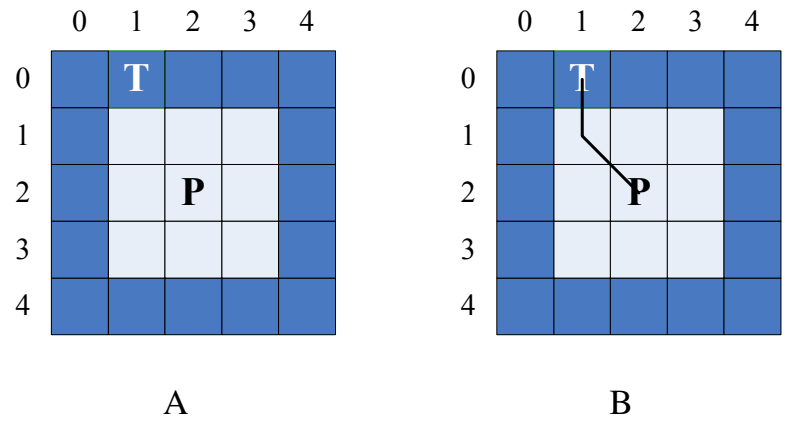
Paveikslėlyje *5 pav.* vaizduojama situacija, kada nuo pradžios taško nėra kelio iki pabaigos taško [6].



5 pav. Kelio paieška, kai nuo pradžios taško nėra kelio iki tikslo

Pavyzdys

Pateiks kelio radimas A* algoritmu. Paveikslėlyje *6 pav.* (A) vaizduojamas žemėlapis, kuriame atliekama kelio paieška. Žymė *P* – tai pradžios taškas, *T* – tikslo taškas [7].



6 pav. Paprastas žemėlapis kelio paieškai

Pradžios mazgui *P* nustatome atributus *g*, *f* ir *h*. Pradžioje *g* = 0. Reikšmė *h* yra skaičiuojama įvairiai, atsižvelgiant į užduotį, situaciją ir kitus veiksnius. Paprasčiausias metodas, dar vadinamas „Manheteno atstumu“ (*Manhattan Distance*), yra horizontalių ir vertikalių svorių skirtumų suma (2).

$$h = |tx - px| + |ty - py|, \tag{2}$$

kai kelio pradžios taškas yra (*px, py*), o tikslo taškas – (*tx, ty*).

Naudojantis „Manheteno atstumo“ principu ir įvertinus tai, kad pradžios taško koordinatės yra (2, 2), o tikslo – (1, 0), *P* mazgo *h* dedamoji bus skaičiuojama taip:

$$h = |1 - 2| + |0 - 2|$$

$$h = 1 + 2 = 3$$

Kadangi *g* = 0, *h* = 3, tai *f* = *g* + *f* = 0 + 3 = 3. Dabar reikia peržiūrėti visus mazgus, sujungtus su *P*. Visų vaikų g dedamųjų reikšmės bus lygios 1 (esančio mazgo *g* lygi tėvo *g* dedamosios ir žingsnių iki esančio mazgo sumos, mūsų atveju tėvo *g* = 0 ir pereinama per vieną žingsnį). Visos *h* dedamųjų reikšmės bus skirtingos, tačiau nesunku pastebėti, kad mazgo, kurio koordinatės (1, 1) svoris bus mažiausias. Taigi kitas apžvelgiamas mazgas bus (1, 1).

Šis (1, 1) mazgas yra sujungtas su (1, 0), (1, 2), (2, 1) ir (2, 2) mazgais. Dabar reikia nustatyti, kokie iš jų yra *Naudotinų* arba *Nenaudotinų mazgų sąrašuose*:

- mazgas (2, 2) yra *Nenaudotinų mazgų sąraše,* nes buvo peržiūrėti visi jo vaikai;

- mazgai (1, 2) ir (2, 1) yra *Naudotinų mazgų sąraše*, nes jų vaikai dar nebuvo peržiūrėti (o jie patys yra pradžios mazgo (2, 2) vaikai);
- mazgas (1, 0) yra naujas mazgas, jo nėra nei viename iš sąrašų.

Suskaičiavus *g*, *f* ir *h* atributus, akivaizdu, kad mazgas (1, 0) turės mažiausią svorį. Kitos iteracijos metu pastebime, kad mazgas (1, 0) – tikslo mazgas.

Taigi trumpiausias kelias nuo *P* iki *T* mazgo yra toks: (2, 2) (1, 1) (1, 0). Paveikslėlyje *6 pav.* (B) yra parodytas surastas kelias.

3.2. Sprendimų priėmimo metodai

Visuose žaidimuose, kuriuose yra kompiuterių valdomų žaidėjų, neatsižvelgiant į žaidimo tipą, 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.

3.2.1. Įvairūs algoritmai

Šiame skyrelyje apžvelgiami skirtingi algoritmai, kurie dažniausiai naudojami žaidimuose sprendimų priėmimui realizuoti.

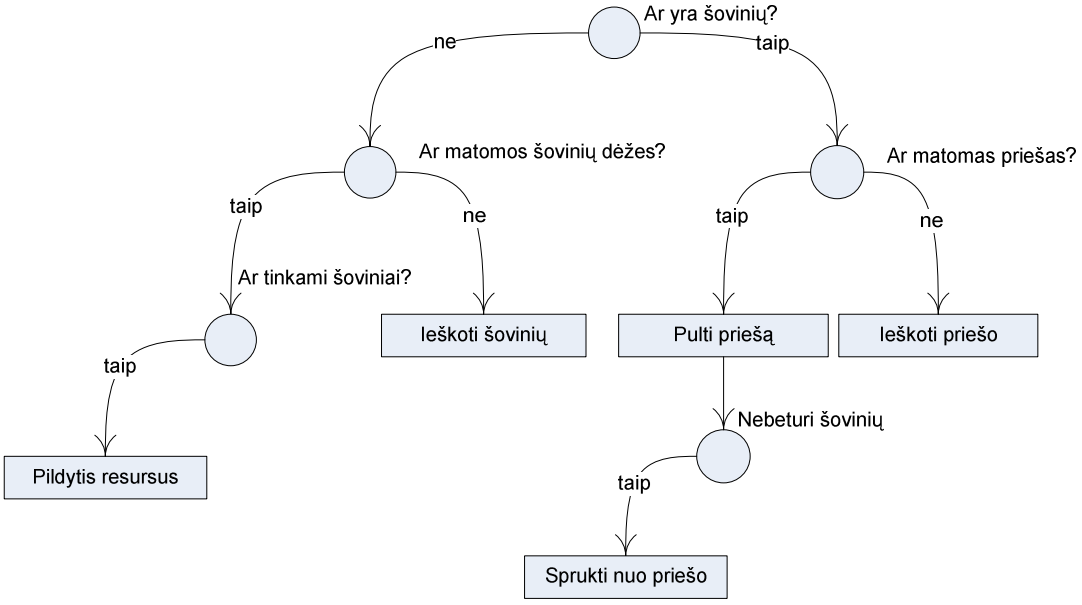
3.2.1.1. Taisyklėmis grįstos (angl. Rule-Based) sistemos

Paprasčiausias būdas žaidimuose realizuoti sprendimų priėmimą – naudoti taisyklių sistemą [2]. Taisyklių sistemos pagrindas – taisyklių, kurios aprašo personažo veiklą, rinkiniai. Kiekviena taisyklė turi tokią formą (3):

$$sąlyga \rightarrow veiksmas \tag{3}$$

Aprašant personažo veiklą, formuojamos tokio tipo taisyklės (pavyzdžiui, *jei nėra šovinių* $\rightarrow$ *ieškoti šovinių*), vėliau žaidimo metu jos yra skaitomos, analizuojamos ir, jei tenkinama taisyklės *sąlyga*, tai vykdomas *veiksmas*. Sudėtingiems ir painiems ryšiams aprašyti naudojami sprendimų medžiai (angl. *Decision Trees*) [1].

Paveikslėlyje *7 pav.* vaizduojamas žaidimo personažo veiklą aprašančių taisyklių rinkinys, išreikštas sprendimų medžiu. Šioje schemoje apskritimais žymimos taisyklės *sąlygos*, o stačiakampiais – *veiksmai*.

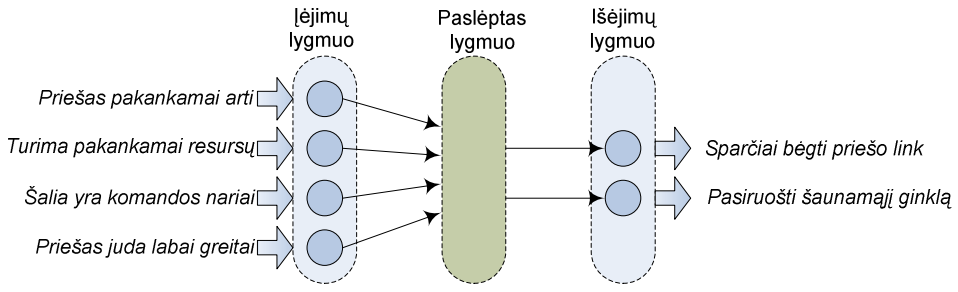


7 pav. Sprendimų medžio schema

Taisyklėmis grįstas sprendimų priėmimas – tai paprasta ir lengvai programuojama metodika. Šį metodą naudojant varikliuke, reiktų taisyklių aprašus kurti naudojant scenarijų kalbą ir numatyti virtualią mašiną šių scenarijų vykdymui.

3.2.1.2. Neuroniniai tinklai

Žaidimuose naudojami neuroniniai tinklai susidaro iš trijų lygmenų: įėjimų, paslėpto ir išėjimų. Neuroninio tinklo pagalba gali būti nuspręsta kokų veiksmą turi atlikti personažas, susiklosčius tam tikrom aplinkybėm [1]. Paveikslėlyje *8 pav.* vaizduojamas labai paprastas ir mažas neuroninis tinklas, turintis keturis įėjimus ir du išėjimus.



8 pav. Sprendimų priėmimas, naudojant neuroninį tinklą

Duomenys iš įėjimo lygmens patenka į paslėptą lygmenį, kur pagal įvairias formules ir charakteristikas sugeneruojami išėjimai. Šie suformuoti išėjimai ir yra veiksmai, kuriuo personažas turi atlikti.

Šis metodas pateikia nenuspėjamus ir žaidėjams įdomius rezultatus. Tačiau, kaip matome iš apžvalgos, pateiktos šio dokumento skyrelyje *2.3 „DI funkcionalumo esančiuose varikliukuose apžvalga“*, tik viename žaidimų varikliuke sprendimų priėmimas yra realizuotas naudojant neuroninius tinklus. Taip yra todėl, kad šis metodas yra gana sudėtingas ir naudoja daug kompiuterio resursų [1].

3.2.1.3. Baigtiniai būsenų automatai

Populiariausias metodas skirtas sprendimams priimti ir žaidėjų veiksmų eigai aprašyti – baigtiniai automatai (BA). BA nusako kaip turi elgtis žaidėjas tam tikru laiko momentu, kaip turi reaguoti į vienus ar kitus įvykius ar žaidėjo elgesį [10]. Detaliau apie BA skaitykite šio dokumento skyriuje *3.2.2 Baigtiniai automatai*.

3.2.2. Baigtiniai automatai

Baigtiniai automatai – tai dažniausiai naudojama šiuolaikinių žaidimų DI kūrimo metodika. Taip yra todėl, kad jie yra lankstūs, paprasti, lengvai išplečiami ir gali būti pritaikyti įvairiose situacijose [11].

Iš baigtinių automatų teorijos žinoma, kad BA – tai sistema, susidedanti iš:

- baigtinės būsenų aibės *S*;
- baigtinio įėjimų raidyno *I*;
- baigtinės perėjimų (iš vienų būsenų į kitas) sąlygų aibės *T(s, i)*.

Baigtiniame automate yra viena pradžios būsena ir nulis arba daugiau *priimančių* būsenų [11].

BA yra skirstomi į dvi klases:

- Mili (angl. *Mealy*) automatai – veiksmai vykdomi perėjimų tarp būsenų metu;
- Muro (angl. *Moore*) automatai – veiksmai atliekami būsenų vykdymo metu.

DI teorijoje BA galima įsivaizduoti kaip paprastą objekto elgesio kaitos (galbūt atsižvelgiant į aplinkos pokyčius) laike aprašą. Žaidimuose realizuojant sprendimų priėmimą baigtiniais automatais dažniausiai yra naudojami Muro automatai, t. y. objekto veiksmai yra atliekami būsenos vykdymo metu. Toks modelis yra intuityviai suprantamas, nes yra natūralu,

kad objektas būdamas tam tikroje būsenoje atlieka tam tikrus veiksmus [11]. Tačiau kartais gali būti naudinga ir atlikti veiksmus vykdant perėjimą tarp būsenų [11].

BA naudojimas DI srityje praplečia jo teorinį apibrėžimą ir standartinį naudojimą tokiais keturiais aspektais:

- Kiekviena būsena būtinai turi kodo fragmentą, kuris nusako objekto elgesį. Taip keičiantis būsenoms, keičiasi objekto elgesys.
- Perėjimų tarp būsenų sąlygos *T* neatskirtos nuo būsenų, kiekvienoje būsenoje yra tikrinamos perėjimo į kitas būsenas sąlygos.
- Priimančių būsenų sąvoka netenka prasmės. Galima teigti, kad žaidimų BA neturi priimančių būsenų, yra tik vieno elgesio pabaiga ir kito pradžia.
- Įėjimai *I* yra pateikiami nuolat iki kol yra sunaikinamas BA arba baigiamas žaidimas.

Apibendrinant galima teigti, kad BA žaidimuose paskirtis – valdyti objektus [11].

3.2.2.1. BA stambios būsenos

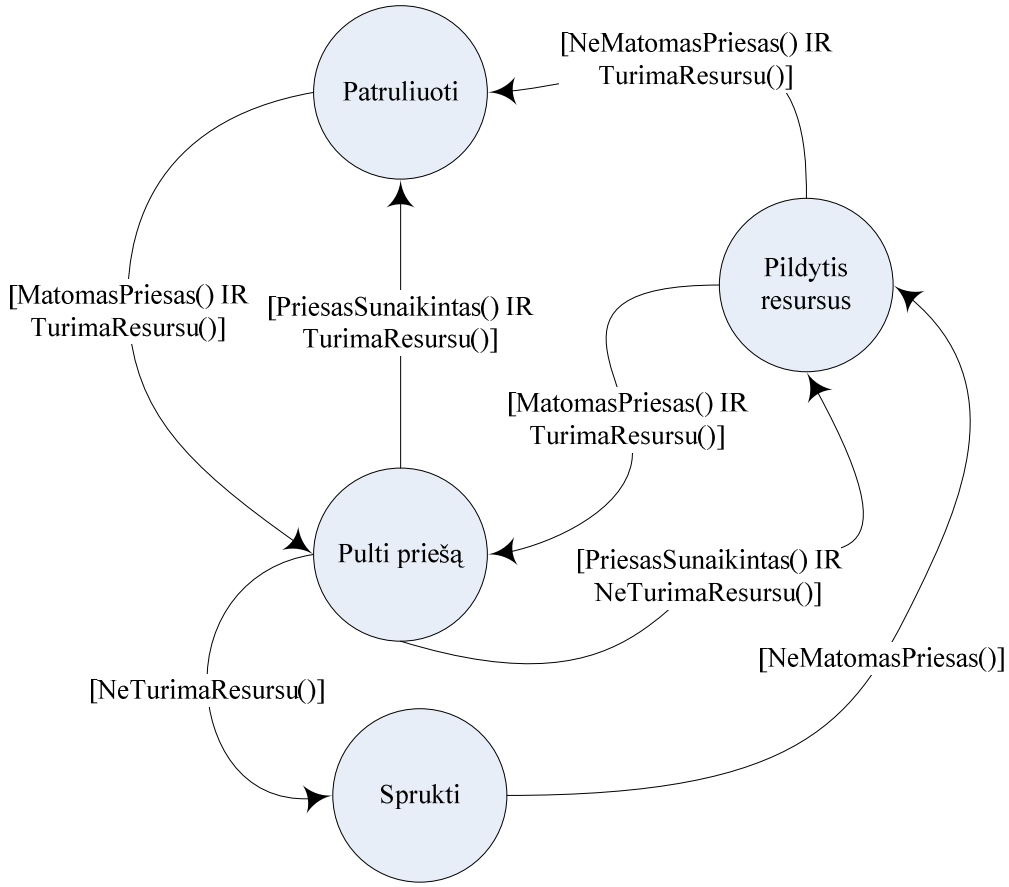
Dažniausiai žaidėjo elgesys yra aprašomas veiksmų seka, todėl BA naudojimas žaidėjo elgesiui valdyti yra labai tinkamas.

Pateiksime trumpą agento tipo žaidėjo veiklos aprašymą, kuris šiame darbe bus naudojamas kaip pavyzdys, analizuojant BA. Lentelėje 2 aprašytos žaidėjo veiklos taisyklės: kaip žaidėjo veikla keičiasi, atsižvelgiant į aplinkos pokyčius.

2 lentelė. Žaidėjo veiklos taisyklės

Nr.	Esama būsena	Perėjimo sąlyga	Būsena, į kurią pereinama
1.	Patruliuoti	*MatomasPriešas* IR *TurimaResursų*	Pulti priešą
2.	Pulti priešą	*PriešasSunaikintas* IR *TurimaResursų*	Patruliuoti
3.	Pulti priešą	*NeTurimaResursų*	Sprukti
4.	Pulti priešą	*PriešasSunaikintas* IR *NeTurimaResursų*	Pildytis resursus
5.	Sprukti	*NeMatomasPriešas*	Pildytis resursus
6.	Pildytis resursus	*MatomasPriešas* IR *TurimaResursų*	Pulti priešą
7.	Pildytis resursus	*NeMatomasPriešas* IR *TurimaResursų*	Patruliuoti

BA galima vaizdžiai pateikti grafo pavidalu. Paveikslėlyje *9 pav.* vaizduojamas automatas, aprašantis žaidėjo elgesį, pateiktą lentelėje 2. Apskritimais vaizduojamos žaidėjo būsenos, kurios nusako žaidėjo elgesį, rodyklėmis – „sprendimai“ pakeisti būseną, virš rodyklių užrašyta būsenos keitimo sąlyga.

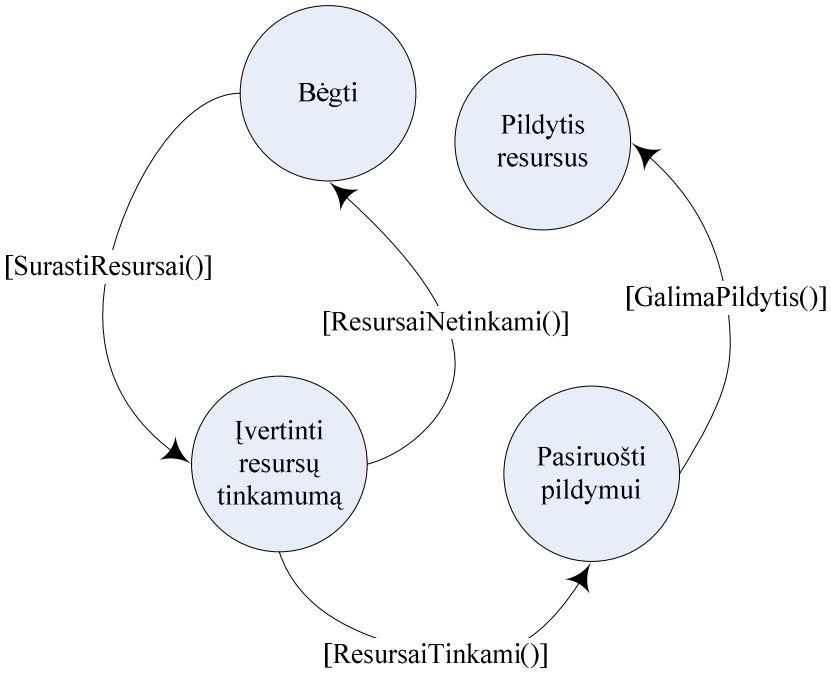


9 pav. Žaidėjo elgesį aprašantis baigtinis automatas

Taip naudojant BA, galima aiškiai pavaizduoti ne tik kelių veiksmų žaidėjų elgesį, bet ir gana painų bei sudėtingą elgesį. Analizės ir projektavimo metu labai greitai paruošiami BA eskizai, be to, jie yra lengvai ir efektyviai realizuojami programiškai (apie BA realizavimą detaliau skaityti šio dokumento punkte *3.2.2.3 BA realizavimas*).

3.2.2.2. BA smulkios funkcijos

Būsenas būtų galima pakartotinai panaudoti, jeigu veiksmai, atliekami būsenos vykdymo metu, būtų atominiai. Atominiai veiksmai – tai nedalomi žaidėjų veiksmai, susidarantys iš vieno pobūdžio veiklos, pvz. bėgimas, ėjimas, šaudymas ir kiti. Galima sukurti tokias būsenas, kurios atspindėtų kuo smulkesnius veiksmus. Taip, pavyzdžiui, būseną „Pildytis resursus“ galima pakeisti tokiomis būsenomis: „Bėgti“, „Įvertinti resursų tinkamumą“, „Pasiruošti pildymui“, „Pildytis resursus“ (*10 pav.*).



10 pav. Būsenos „Pildyti resursus“ suskaidymas į atominius veiksmus

Tokie nedideli veiksmai, kaip „Bėgti“ gali būti vykdomi ir atliekant kitus veiksmus, pvz., „Pulti priešą.

3.2.2.3. BA realizavimas

Būsenų automatą programiškai realizuoti galima įvairiai. Šiame skyriuje pateikiami dažniausių žaidimo kūrimo būdų aprašymai su pavyzdžiais (realizuojant paveikslėlyje *9 pav.* vaizduojamą būsenų automatą).

„Switch...case“ konstrukcijos

Nesudėtingai ir greitai BA galima realizuoti naudojant „switch...case“ konstrukcijas. Tai „if..then“ sąlygų sakinių ir kreipinių į veikos funkcijas visuma su jungiklių tarp būsenų. Toks būdas yra paprastas, nereikalaujantis ypatingų programavimo įgūdžių, tačiau tinkantis tik mažam kiekiui būsenų aprašyti.

Didžiausias šios realizacijos privalumas – greitas kodo kompiliavimas ir spartus sukompiliuoto kodo vykdymas. Tačiau tokios „switch..case“ konstrukcijos nėra tinkamos realizuojant DI varikliuką. Kuriant sistemą, kuri bus pakartotinai naudojama (pavyzdžiui kituose žaidimuose), reikia numatyti sistemos plečiamumo galimybes. O šis sprendimas nėra lankstus. Norint pridėti naują būseną ar naują perėjimo sąlygą, reikia keisti programos kodą ir jį perkompiliuoti po pakeitimų. Tokiems veiksmams atlikti būtina turėti programavimo žinių.

Tai yra vienas pagrindinių šio sprendimo trūkumų. Žemiau pateiktas programos kodo fragmentas, kuriame realizuotas BA, naudojant „switch..case“ konstrukcijas.

```
// vaikščioti nurodytomis trajektorijomis
#define Patruliuoti 0

// priartėti prie prie priešo, nusitaikius šauti
#define PultiPriesa 1

// stengtis nutolti nuo priešo
#define Sprukti 2

// surasti reikiamus resursus ir pasipildyti turimas atsargas
#define PildytisResursus 3

<...>

// patikrinti ar būsena nepasikeitė
switch (state)
{
  case 0:
  {
    // vykdyti patruliavimo veiksmus
    // tikrinti perėjimų į kitas būsenas sąlygas
    break;
  }
  case 1:
  {
    // vykdyti priešo puolimo veiksmus
    // tikrinti perėjimų į kitas būsenas sąlygas
    break;
  }
  case 2:
  {
    // vykdyti sprukimo veiksmus
    // tikrinti perėjimų į kitas būsenas sąlygas
    break;
  }
  case 3:
  {
    // vykdyti resursų papildymo veiksmus
    // tikrinti perėjimų į kitas būsenas sąlygas
    break;
  }
}
```

Programos fragmentas 1: BA realizuotas naudojant „switch...case“ konstrukcijas

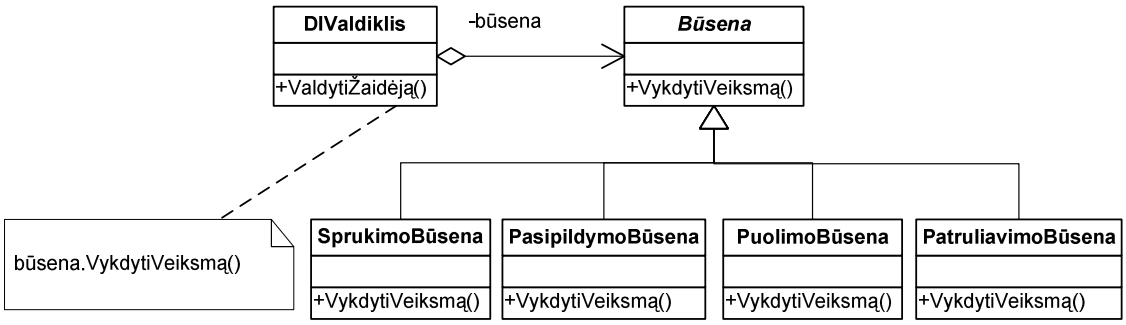
Kitas didelis trūkumas – didėjant veiklos taisyklių ir ryšių tarp būsenų skaičiui, didėja sąlygų aprašymo sudėtingumas, „if...then“ konstrukcijų gylis. Programos kodas tampa perkrautas ir nebesuprantamas.

Būsenų projektavimo šablono (angl. *State Design Pattern*) naudojimas

Kur kas pranašesnis yra į objektus orientuotas sprendimas. Realizuojant BA galima naudoti būsenų projektavimo šabloną (angl. *State Design Pattern*) [13]. Tokio sprendimo esminiai akcentai – sukuriama bazinė abstrakti klasė – sąsaja (angl. *Interface*), kurioje yra

aprašyta pagrindinių metodų aibė. Šią bazinę klasę paveldi konkrečios klasės, aprašančios būsenų logiką, kuriose realizuojami bazinės klasės metodai. Klasėje – valdiklyje sukuriamas dabartinės būsenos klasės objektas, kuris, keičiantis būsenai, pasikeičia į naujos būsenos klasės objektą.

Paveikslėlyje *11 pav.* pateikta šio sprendimo klasių diagrama. Čia matome abstrakčią bazinę klasę *Būsena*, konkrečias būsenos klases: *SprukimoBūsena, PasipildymoBūsena, PuolimoBūsena, PatruliavimoBūsena*, klasę valdiklį - *DIValdiklis.*



11 pav. BA, realizuoto naudojant būsenų projektavimo šabloną, klasių diagrama

Toks sprendimas yra intuityviai suprantamas ir lengvai programuojamas. Šį modelį nesunku papildyti naujomis būsenomis (tereikia naujoms būsenoms sukurti konkrečias klases, kurios pavedėtų bazinę klasę, ir juose realizuoti būsenų logiką). Taip pat greitai ir be didelių pastangų galima praplėsti arba pakeisti jau esančių būsenų logiką [13].

3.2.2.4. Išvados

Be to, kad baigtiniais automatais galima valdyti bet kokį žaidimo objektą, jie yra plačiai naudojami programuojant DI dar ir dėl tokių priežasčių [12]:

- BA naudojami siekiant sumažinti žaidėjų veiklos sudėtingumą. Taip žaidėjų veiklos yra skaidomos į smulkius, atominius veiksmus.

- BA yra naudingi sinchronizuojant DI veiksmus su išoriniais įvykiais, tokiais, kaip animacija, garsas, laikmačiai. BA suteikia kūrėjams galimybę integruoti animaciją su DI veikimu, neįvendat griežtų apribojimų pačios animacijos meninei vizijai.

- BA yra paprasta derinti ir ieškoti juose klaidų, palyginus su kitais DI metodais, tokiais, kaip neuroniniai tinklai ar genetiniai algoritmai. BA veikimas yra tiksliai apibrėžtas, t.y. tam tikroms sąlygoms ir įėjimams yra vienareikšmiškai apibrėžta kurioje iš esamų būsenų turi būti objektas.

3.3. Scenarijų kalbos

Didėjant žaidimo projektui, didėja išeities teksto kompiliavimo laikas. Tik kelių konstantų pakeitimas dideliame projekte gali užtrukti iki 10 min., nes atlikus pakeitimus būtina kodą perkompiliuoti. Siekiant to išvengti, konstantas ir kitus duomenis programuotojai iškelia į atskiras rinkmenas ir sukuria įrankius, kurie nuskaito ir išanalizuoja (angl. *parse*) šias duomenų rinkmenas [13]. Dabar tokie įrankiai yra kuriami vis rečiau, nes egzistuja labai patogios ir tinkančios šiems veiksmams atlikti scenarijų kalbos (angl. *scripting languages*).

Žaidimuose įvairiems DI elementams, logikai ar veiklai aprašyti labai dažnai yra naudojamos supaprastintos programavimo kalbos (angl. *scripting languages*), tokios kaip LUA, PYTHON, JAVA, Rubi, PHP ir kitos.

Šiame skyriuje trumpai supažindinama su dažniausiai žaidimuose naudojamomis scenarijų kalbomis LUA ir PYTHON, išryškinami šių kalbų naudojimo žaidimuose privalumai ir trūkumai.

3.3.1. LUA apžvalga

LUA – per paskutinius penkerius metus sparčiai išpopuliarėjusi, scenarijų programavimo kalba. LUA scenarijai buvo naudojami tokiose žaidimuose, kaip *Escape From Monkey Island, Homeworld 2, Far Cry, Baldur‘s Gate* [3]. Ši kalba yra greita, ją paprasta naudoti, nors ji ir turi didelį funkcionalumą. Tai yra nemokamas, atviro kodo produktas. LUA yra lengvai įterpiama į kitas programavimo aplinkas [13].

Daugiau apie LUA kalbą skaitykite priede *Priedas 1. LUA programavimo kalbos apžvalga.*

3.3.2. PYTHON apžvalga

PYTHON tai aukšto lygio programavimo kalba, sukurta kaip C kalbos praplėtimas [16]. Ši kalba yra orientuota į objektus, turi papildomus paketus grafikos vaizdavimui. Žaidimuose dažniausiai naudojama žaidimų scenarijams aprašyti. Ši kalba yra gerai dokumentuota, turi daug aiškiai aprašytų funkcijų [16].

3.3.3. LUA ir PYTHON kalbų palyginimas

Pastaruoju metu lėtai, bet užtikrintai LUA kalba žaidimų kūrime pirmauja prieš PYTHON kalbą. Lyginat šias dvi kalbas, galima pastebėti tokius LUA kalbos trūkumus:

- LUA kalbos funkcionalumas yra mažai dokumentuotas. Yra tik kelį įvadiniai programavimo LUA kalba vadovai. Nėra bendros žinių bazės, daug nedokumentuotų funkcijų.
- Slankaus kablelio skaičių naudojimas LUA kalboje yra daugelio klaidų šaltinis. Tam kad teisingai naudoti slankaus kablelio skaičius, reikia atlikti papildomus programavimo veiksmus [14].
- PYTHON kalbai yra sukurta daugiau pagalbinių įrankių ir įvairių bibliotekų, kurios praplečia kalbos funkcionalumą ir naudojimo galimybes [14].

Visų išvardintų LUA kalbos trūkumų priežastis yra ta, kad LUA kalba atsirado labai neseniai ir per trumpą laiką dar nebuvo paruošta visą dokumentacija ir ne ištaisytos visos klaidos. Laikui bėgant, trūkumų skaičius mažės.

LUA kalba turi ir tokius svarbius privalumus:

+ LUA žymiai greitesnė, lengviau išmokstama, neperkrauta bereikalingu funkcionalumu. Ja gali programuoti ir ypatingų programavimo žynių neturintys žaidimų lygių kūrėjai.
+ LUA kalbos sintaksė pritaikyta į procedūras orientuotam programavimui, be to kalboje yra dinaminis kintamųjų tipizavimas [3].
+ LUA kalba naudoja pakankamai mažai kompiuterio resursų, o tiksliau operatyviosios atminties [14].
+ LUA kalboje yra automatinis naudojamos atminties valdymas su „šiukšlių surinkėju“ (angl. *Garbage Collector*) [3].

Tačiau svarbiausias LUA kalbos privalumas yra tas, kad ši kalba yra labai lengvai integruojama su įvairiomis aplinkomis. Yra sukurtos lengvai naudojamos programų kūrimo sąsajos (angl. *Application Programming Interfase*), skirtos duomenų apsikeitimui tarp LUA ir kitų programavimo kalbų [3]. Pavyzdžiui, viena iš tokių sąsajų – CPB – „tiltas“ tarp LUA ir C++ kalbų (angl. *Bridging LUA and C++*) [15]. Dėl to, kad yra sukurti tokie įrankiai, LUA kalba idealiai tinka žaidimų kūrimui [3].

3.3.4. Išvados

Dėl LUA kalbos privalumų, pateiktų šio dokumento skyriuje *3.3.3 LUA ir PYTHON kalbų palyginimas*, savo varikliuko duomenims aprašyti ir scenarijams kurti pasirinkome LUA kalbą.

Duomenų atskirimas nuo algoritmo labai palengvina programavimą ir padidina programų pritaikomumą. Scenarijų naudojimas dar daugiau padidiną programos išplečiamumą, nes šiuo atveju atskiriami ne tik duomenis, bet ir dalis funkcijų.

4. DI varikliuko pagrindinių uždavinių realizavimas

4.1. Duomenimis valdoma sistema (angl. Data-Driven System)

Pagrindiniam darbo tikslui – duomenų ir žaidimo logikos atskirimui nuo žaidimo programos kodo – pasiekti naudojome duomenimis valdomą detalią architektūrą (angl. *Data-Driven Design*).

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 (tokio darbo schema pateikta paveikslėlyje *1 pav.*). Šie scenarijai yra valdomi žaidimų varikliuke: žaidimo metu jie užkraunami į operatyviąją atmintį ir vykdomi.

4.1.1. Sistemos aprašymas

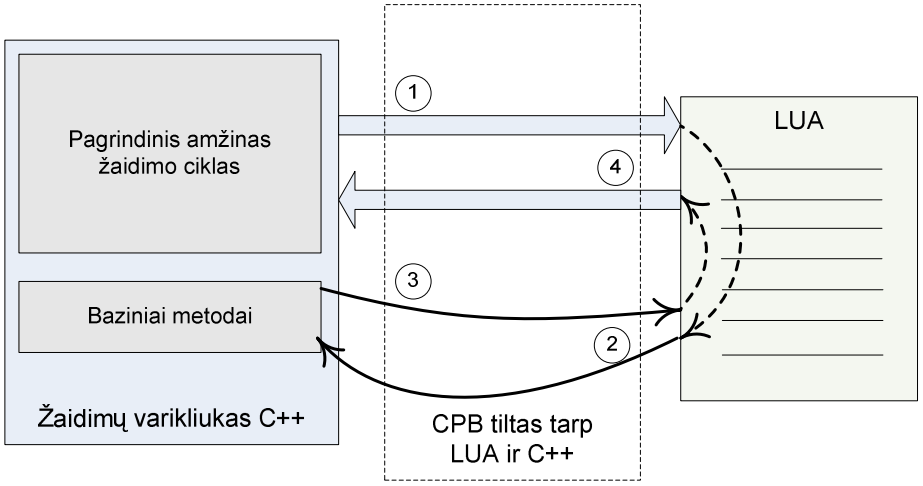
Scenarijų valdyme galima išskirti tokius esminius objektus:

- 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 – tiltą tarp LUA ir C++ kalbų [15].

4.1.2. Veikimo principai

Paveikslėlyje *12 pav.* vaizduojama LUA kalba parašytų scenarijų valdymo bendrinė schema. Joje vaizduojamas visų LUA (nepriklausomai nuo sprendžiamo uždavinio) valdymas.



12 pav. LUA kalba parašytų scenarijų vykdymo 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.

4.1.3. DI pagrindinių uždavinių sprendimas

Kelio paiešką ir sprendimų priėmimą realizavome, remdamiesi duomenimis valdomų sistemų principais.

Kelio paieškoje į LUA scenarijus nukėlėme personažų savybių aprašus, ieškomo kelio įvertinimo funkciją bei rasto kelio įveikimo funkciją. Žaidimo kūrėjai gali kiekvienam

žaidimo personažui sukurti unikalius savybių aprašus ir kelio įvertinimo ir įveikimo funkcijas. Apie kelio paieškos realizavimą skaitykite šio dokumento skyriuje *4.2 Kelio paieška*.

Realizuojant sprendimų priėmimą, visų BA būsenų aprašymai bei personažų surišimas su būsenomis yra nukelti į LUA rinkmenas. Žaidime kiekvienos iteracijos metu vykdant personažo valdymą, yra užkraunamos ir analizuojamos su tuo personažu susietos būsenos. Apie sprendimų priėmimo realizaciją skaitykite šio dokumento skyriuje *4.3 Sprendimų priėmimas.*

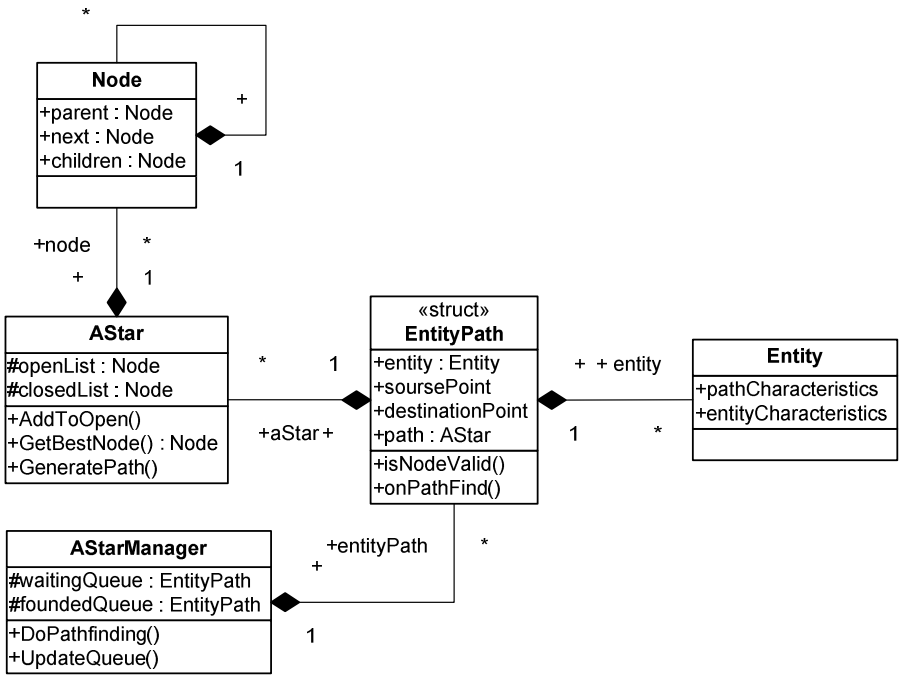
4.2. Kelio paieška

Kelio paieškai DI varikliuke realizuoti pasirinkome A* algoritmą. Tačiau jo klasikinį realizavimą šiek tiek praplėtėme. Į LUA scenarijus nukėlėme personažų savybių aprašus bei kelio įvertinimo funkcijas.

4.2.1. A* algoritmo projektavimas

Paveikslėlyje *13 pav.* vaizduojama A* algoritmo realizacijos klasių diagrama. Kelio paieškai atlikti naudojamos tokios klasės:

- *Entity* klasė aprašo personažus, personažo savybės, susijusios su kelio paieška ar kelio įveikimu yra užkraunamos iš LUA scenarijų.
- *EntityPath* struktūra skirta saugoti nuorodai į personažą ir informacijai apie personažui ieškomą ar rastą kelią: kelio pradžios ir pabaigos taškai, rastas kelias, nuorodos į kelio įvertinimo ir įveikimo funkcijas, kurios užkraunamos iš LUA scenarijų.
- *Node* – vieną kelio celę apibūdinanti klasė, joje yra nuorodos į į trumpiausią kelią patenkančias celes ir į artimiausias žemėlapio celes.
- *AStar* – A* algoritmo pagrindinė klasė su *Naudotinų mazgų* ir *Nenaudotinų mazgų sąrašais*. Apie A* algoritmą skaitykite šio dokumento skyriuje *3.1.2 A* algoritmas*.
- *AStarManager* – klasė skirta valdyti visus žaidimo personažus, kuriems reikia surasti kelią.



13 pav. A algoritmo realizavimo klasių diagrama*

4.2.2. Pasiektas rezultatas

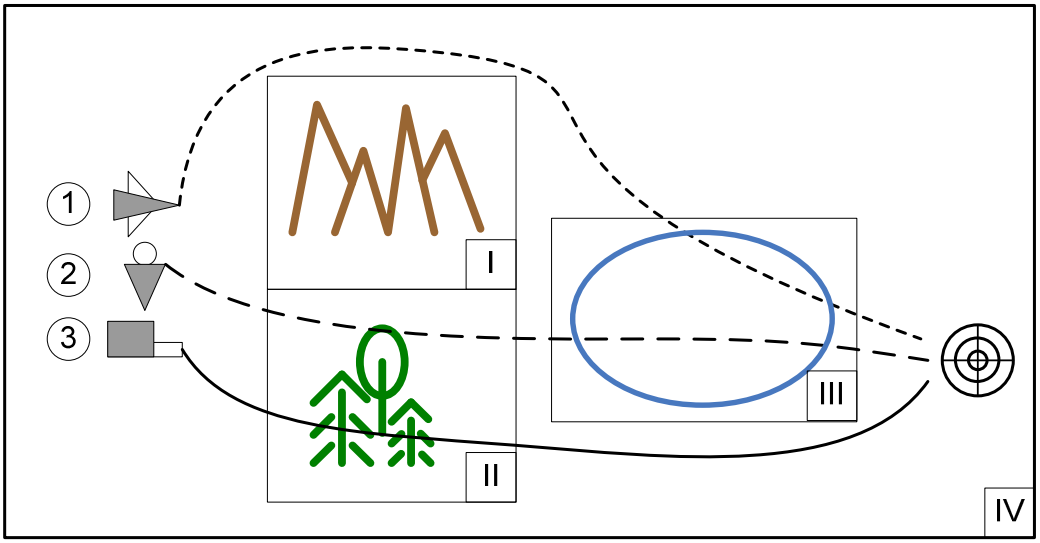
Kelio paiešką realizavome naudodamiesi A* algoritmu, tačiau algoritmą pritaikėm taip, kad kiekvienam personažui būtų galima nurodyti ypatingas savybes ir taisykles, pagal kurias vyksta kelio įvertinimas, kelio įveikimas ir kitos su kelio paieška susijusios veiklos. Taip pavyko duomenis atskirti nuo algoritmo, t.y. naudojant tą patį algoritmą skirtingiems žaidėjams, algoritmas suranda skirtingą kelią.

Lentelėje 3 pateikiami trijų tipų personažai ir jų charakteristikos: kiekvienos žemėlapio srities įvertinimas kiekvienam personažui. Pavyzdžiui, pirmojo tipo žaidėjas – lėktuvas – negali skristi virš kalnų ir miško, turi juos apskristi, tačiau be papildomų sąnaudų gali skristi virš vandens arba lygumos.

3 lentelė. Mazgų srityse svoris žaidimo veikėjams

Sritis / Objektas	I kalnai	II miškas	III vanduo	IV lyguma
1	∞	∞	1	1
2	∞	5	3	1
3	∞	7	∞	1

Skirtingas charakteristikas turintiems personažams naudojant tą patį algoritmą mūsų modifikuota kelio paieška suranda skirtingus kelius personažams, atsižvelgiant į jų charakteristikas. Šių kelių trajektorijos vaizduojamos paveiksle *14 pav.*



14 pav. Optimalieji personažų keliai, priklausantys nuo personažo tipo

Iš lentelės 3 matome, kad trečiasis objektas negali judėti pirmąja (kalnais) ir trečiąja (vandeniu) sritimis, tačiau gali važiuoti antrąja (mišku). Mūsų modifikuotas A* algoritmas surado jam tokį trumpą kelią, kuris eina mišku, bet aplenkia vandenį.

4.3. Sprendimų priėmimas

Šiame skyriuje pateikiamas baigtinių automatų modelis, pagrįstas duomenimis valdomomis sistemomis, skirtas žaidimo personažams valdyti.

4.3.1. Būsenų automato išplėtimas

Kaip jau buvo minėta šio dokumento skyriuje *4.3 Spendimų priėmimas,* BA – tai dažniausiai naudojama metodika sprendimų priėmimui realizuoti. Mes pateiksime šiek tiek modifikuotą šio sprendimo variantą.

4.3.1.1. Baigtinio automato būsenų specifikavimas

Kiekvienoje žaidėjo veiklos būsenoje galima išskirti tris tokius etapus:

- būsenos pradžios etapas – šio etapo veiksmai atliekami vieną kartą tik įėjus į būseną, pavyzdžiui animacijos užkrovimas, kelio pradinės atkarpos radimas ar kiti paruošiamieji darbai;

- būsenos pagrindinis etapas – šis etapas vykdomas begaliniame cikle kol nepereinama į kitą būseną. Čia reikia apsirašyti visas veiklas, kurias žaidėjas turi atlikti būdamas šioje būsenoje. Šis etapas vykdomas kiekvienos (arba kas antros, kas trečios, ar kitu dažnių) žaidimo iteracijos metu.

- būsenos pabaigos etapas – šio etapo veiksmai atliekami vieną kartą išeinant iš būsenos. Būsenos pabaigos etapo veiksmai gali būti skirti nebenaudojamų objektų trynimui, atminties ir kitų resursų atlaisvinimui.

Kiekvieną šių etapų galima įsivaizduoti kaip atskirą funkciją.

4.3.1.2. BA būsenų aprašymas LUA scenarijais

Būsenoms aprašyti yra patogu naudoti scenarijų kalbą (apie scenarijų kalbų privalumus skaitykite šio dokumento punkte *3.3 LUA ir Python scenarijų kalbos*). Žemiau pateikiamas programos, parašytos LUA scenarijų kalba, fragmentas, kuriame aprašoma trijų etapų būseną.

```
--Veiksmai, atliekami įėjus į būseną (vieną kartą)--
function OnEnter(entity_handle)
  PrepareWeapon()
  FindShortestPath()
  end --function OnEnter()

--Veiksmai, atliekami esant būsenoje (cikle, kol nepasikeičia būsena)--
function OnExecute(entity_handle)
  LockOnTarget()
  Shot()
  GetCloser()
  end --function OnExecute()

--Veiksmai, atliekami prieš pakeičiant būseną (vieną kartą)--
function OnExit(entity_handle)
  HideWeapon()
  end --function OnExit()
```

Programos fragmentas 2: BA būsenos etapų aprašymas, naudojant LUA kalbą

Prie tokio aprašymo dar galima pridėti ir būsenos pakeitimo sąlygų patikrinimą. Patikrinimas, taip pat kaip ir etapai, gali būti realizuotas funkcija, kurios grąžinamasis rezultatas – naują žaidėjo būsena.

```
--Būsenos keitimo sąlygų tikrinimas--
function ChangeState(entity_handle)
   if EnemyDead(entity_handle) == 1 and GetResources(entity_handle) > 1
     then return "PATROL_STATE"
   else
     if GetResources(entity_handle) < 1
       then return "RUNAWAY_STATE"
     end
   end
   end --function ChangeState()
```

Programos fragmentas 3: BA sąlygos tikrinimas, naudojant LUA kalbą

Programos fragmente 3 aprašyta funkcija, kurioje atliekamas perėjimo tarp būsenų sąlygos tikrinimas.

Tiek etapų funkcijose, tiek sąlygos partikrinimo funkcijoje yra kreipiamasi į tokius metodus: *GetCloser(), GetResources(), HideWeapon()*. Šie metodai aprašyti C++ kode ir gali būti naudojami skirtingiems žaidėjams valdyti.

4.3.1.3. BA būsenų aprašymų generavimas

Būsenų aprašams generuoti sukūrėme pagalbinį įrankį. Juo naudojantis grafinėje aplinkoje galima sukurti būsenų automatą, ir įrankis sugeneruos LUA scenarijus, aprašančius automato būsenas.

Baigtinių būsenų automatų aprašų generavimo įrankis skirtas kurti automatų aprašams LUA kalba. Kiekviena būsena yra aprašoma skirtingoje rinkmenoje. Žaidimo metu šios rinkmenos yra užkraunamos, nuskaitomi jų aprašai ir žaidimų objektai valdomi pagal aprašuose pateiktas funkcijas.

Funkcijos, kurias vartotojas gali pasirinkti, yra realizuotos žaidimo kode C++ kalba. Žaidimų varikliuke yra pateiktas esminių funkcijų rinkinys, tačiau, jei vartotojas pageidauja šį rinkinį praplėsti ir yra kompetentingas, gali sukurti savo funkcijas.

Apie būsenų aprašų generavimo įrankį skaitykite priede *Priedas 2. Būsenų automato aprašų generatorius. Naudojimo vadovas.*

4.3.2. BA, aprašyto LUA scenarijais, valdymas

Šio dokumento skyriuje *4.3.1 Būsenų automato praplėtimas* aprašytą baigtinį automatą reikia integruoti į žaidimą.

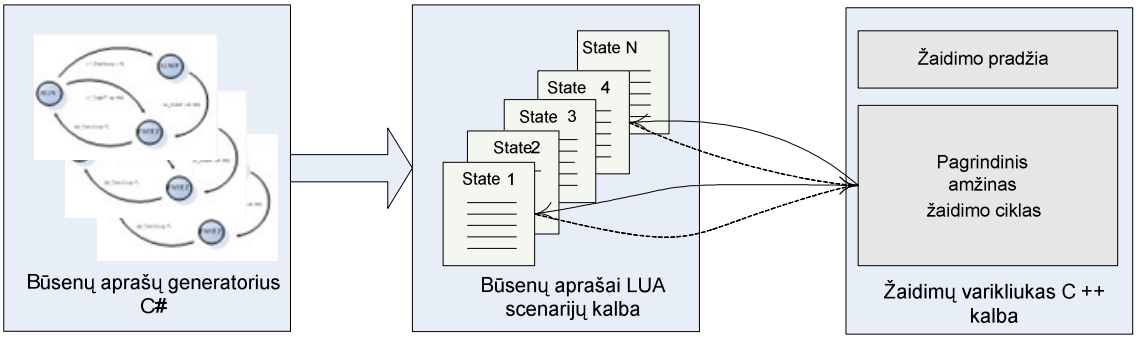
4.3.2.1. Būsenų automatų valdymas

Integruojant LUA scenarijais aprašytus automatus į žaidimą, būtina sukurti būsenų valdiklį. Visą BA sistemą galima išskaidyti į tokias sudedamąsias:

- **Būsenų aprašai**, LUA scenarijais aprašytos būsenos. Jos saugomos išorinėje atmintyje, kiekviena būsena atskiroje rinkmenoje, žaidimo vykdymo metu užkraunamos į operatyviąją atmintį.
- **Žaidėjus valdančios funkcijos**, aprašytos C++ kalba funkcijos, kurios atlieka žaidimo personažų valdymą. Šio funkcijos yra kviečiamos iš LUA scenarijų būsenų vykdymo metu.
- **BA valdiklis** – objektas, kuriame atliekamas visų žaidimo personažų būsenų valdymas.
- **Duomenų persikeitimo modulis –** tai pagalbinės funkcijos, skirtos sujungti LUA ir C++ kalbas. Mes naudojome CPB tiltą tarp LUA ir C++ [15].
- **LUA scenarijų talpykla (angl. *cache*)** – visi anksčiau užkrauti LUA scenarijai kurį laiką saugomi talpykloje tam, kad prireikus juos būtų galima greičiau pasiekti.

Paveikslėlyje 15 pav. pateikta būsenų valdymo schema. Žaidimo būsenų valdymą galima suskirstyti į tokius žingsnius:

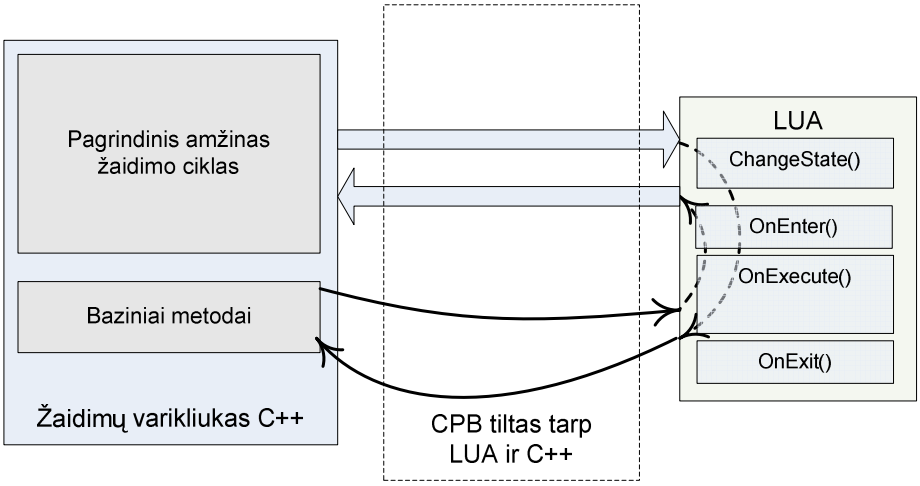
1. Žaidimo programuotojai naudojasi būsenų aprašų generatoriumi ir sukuria būsenų aprašų rinkmenas.
2. Žaidimo personažams priskiriami automatai, kurie aprašo jų elgesį.
3. Žaidimo vykdymo metu būsenų valdiklis užkrauna būsenas į operatyviąją atmintį ir pagal būsenų kodą vykdo žaidimų personažų valdymą.



15 pav. Būsenų, aprašytų LUA kalba, integracijos su žaidimu schema

Paveikslėlyje *16 pav.* vaizduojama vienos būsenos apdorojimo eigos schema. Ši veiksmų seka yra vykdoma kiekvienam žaidimo personažui, valdomam kompiuterio, žaidimo

pagrindinio ciklo iteracijų metu tam tikrų iš anksto nustatytu dažniu (pavyzdžiui, kas penkerias iteracijas).



16 pav. Būsenų, aprašytų LUA kalba, valdymo schema

BA valdiklio veiksmų eigą kiekvienam personažui galima aprašyti taip:

1. Gauti esamą būseną.
2. Jei esamos būsenos nėra LUA scenarijų talpykloje, užkrauti esamą būseną į operatyviąją atmintį.
3. Vykdyti būsenos *ChangeState()* metodą iš LUA scenarijaus. Vykdymo metu kviesti reikiamas C++ kalbos funkcijas. Tikrinti grąžintą rezultatą
4. Jei grąžinamas naujos būsenos pavadinimas:
   a. Vykdoma esamos būsenos *OnExit()* funkcija.
   b. Personažui priskiriama nauja esama būsena.
   c. Jeigu senos būsenos nebuvo LUA scenarijų talpykloje, tai įtraukti seną būseną į LUA scenarijų talpyklą.
   d. Nauja esama būsena užkraunama į operatyviąją atmintį.
   e. Vykdoma naujos esamos būsenos *OnEnter()* funkcija.
5. Jei negrąžinamas naujos būsenos pavadinimas (būsenos nereikia keisti), tai vykdoma esamos būsenos *OnExecute()* funkcija.

5. Eksperimentinis tyrimas

Tyrėme dviejų programų veikimo greitį ir išanalizavome naudojimo patogumą, toje pačioje aplinkoje su tais pačiais duomenimis.

Pirmoji programa – projektavimo šablonais realizuotas baigtinis automatas, atliekantis žaidimo personažų valdymą (įprastinė baigtinių automatų realizacija). Šios programos duomenys ir žaidimo logika yra įprogramuoti į patį žaidimo kodą.

Antroji programa – tai mūsų modifikuotas baigtinis automatas, realizuotas naudojant duomenimis valdomą detaliąją architektūrą. Šioje programoje duomenys ir žaidimo logika yra atskirti nuo žaidimo kodo, nukelti į LUA scenarijus (detaliau apie realizaciją skaitykite šio dokumento skyriuje *4.3 Sprendimų priėmimas*).

Palyginome ir aprašėme abiejų programų naudojimo patogumą ir veikimo greičius.

5.1. Naudojimo patogumas

Bandėme pridėti po vieną naują būseną būsenų automatuose, kuriuos apdoroja abi programos. Pateiksime veiksmų, kuriuos reikia atlikti norint praplėsti esantį automatą (apie šį baigtinį automatą skaitykite šio dokumento skyriuje *3.2.2.1 BA stambios funkcijos*), sekas.

- Pirmoji programa (įprastinė BA realizacija):
    1. Sugalvoti naujos būsenos logiką.
    2. Suprojektuoti būsenos integravimą į esantį baigtinį automatą.
    3. Atsidaryti žaidimo programos kodą.
    4. Surasti vietą programos kode, kur yra aprašomos visos būsenos.
    5. Sukurti naujos būsenos kodą.
    6. Sukurti perėjimus tarp senų būsenų ir naujos.
    7. Perkompiliuoti žaidimo kodą.
    8. Paleisti žaidimą. Patikrinti naujos būsenos veikimą.
- Antroji programa (mūsų modifikuoto BA realizacija, remiantis duomenimis valdoma architektūra):
    1. Sugalvoti naujos būsenos logiką.
    2. Suprojektuoti būsenos integravimą į esantį baigtinį automatą.

3. Sukurti naujos būsenos LUA scenarijų (tai labai patogu padaryti, naudojantis mūsų sukurtų BA būsenų aprašų generatoriumi, apie šį įrankį detaliau skaitykite priede *Priedas 2. Būsenų automato aprašų generatorius. Naudojimo vadovas.*).
4. Pakeisti esančius LUA scenarijus, kuriuose reikia naujų perėjimo sąlygų.
5. Paleisti žaidimą. Patikrinti naujos būsenos veikimą.

Iš išvardintų žingsnių, kuriuos reikia atlikti norint pridėti naują būseną į automatą, matosi, kad mūsų pasiūlytas BA realizavimas yra ne tik greičiau atliekamas (tik 5 žingsniai), bet ir yra žymiai paprastesnis. Jo metu nereikia perkompiliuot viso žaidimo kodo, o naują būseną galima sukurti naudojantis grafine sąsaja.

Priede *Priedas 3 Tyrimo rezultatai. Naudojimo patogumas* pateikiami būsenų aprašymai sukurti su pirmąja programa (būsenų programos kodas) ir antrąja (būsenų aprašų generavimo įrankio vaizdas).

Taigi, galime teigti, kad antroji programa yra žymiai pranašesnė už pirmąją.

5.2. Veikimo greitis

5.2.1. Tyrimo aplinka

Abiejų programų veikimo greitis buvo tiriamas viename kompiuteryje. Jo techninės ir programinės įrangos charakteristikos išvardintos žemiau.

- Techninės charakteristikos yra šios:
  - Procesorius – Intel Pentium M, 1.73 GHz
  - Operatyvioji atmintis – 504 MB
  - Kietasis diskas – Toshiba MK6026GAX, 55.8 GB
- Operacinė sistema – Microsoft Windows Server 2003 Enterprise Editon Service.

5.2.2. Tyrimo aprašymas

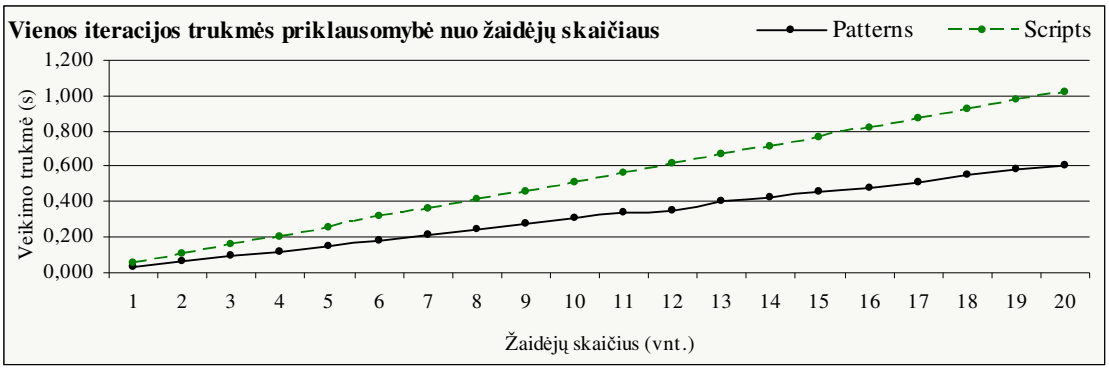
Atlikome tris testus. Kiekvieno testo metu abi programas vykdėme 100 iteracijų cikle. Tyrėme programų veikimą, apdorojant baigtinį automatą iš 4 būsenų ir 7 perėjimų tarp jų. Apie šį baigtinį automatą skaitykite šio dokumento skyriuje *3.2.2.1 BA stambios funkcijos.*

Kiekvieno testo metu nuosekliai didinome valdomų personažų skaičių. Testai vienas nuo kito skiriasi būsenų gyvavimo trukme. Būsenų gyvavimo trukmė – tai iteracijų skaičius, kuris nusako kiek ilgai galima šią būseną neatnaujinti. Pirmojo testo metu būsenos trukmė – 5 iteracijos, antrojo – 6, trečiojo – 7. Tyrimo apibendrintus rezultatus pateikėme šio dokumento

skyrelyje *5.2.3 Tyrimo rezultatai*. Visus tyrimo duomenis galima rasti priede *Priedas 4. Tyrimo rezultatai. Veikimo greitis*.

5.2.3. Tyrimo rezultatai

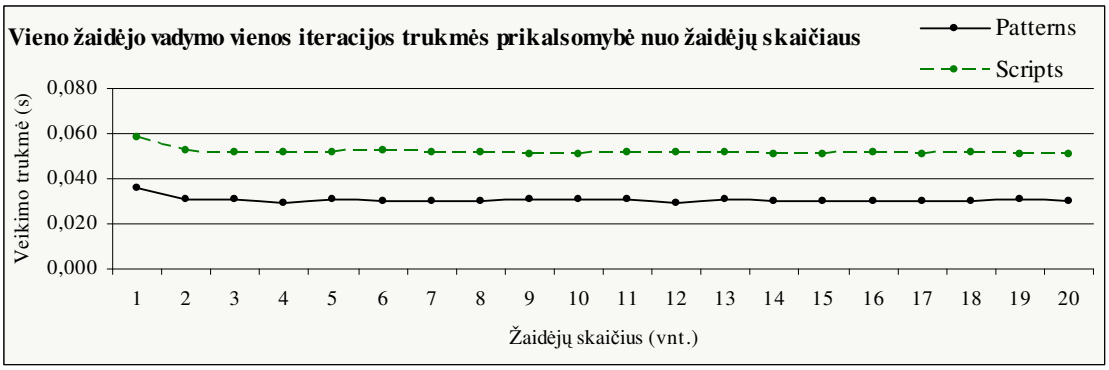
Paveikslėlyje *17 pav.* vaizduojama vienos iteracijos apdorojimo trukmės priklausomybė nuo apdorojamų žaidėjų skaičiaus. Matome, abiejų programų veikimo trukmė didėja tiesiškai, didinant personažų skaičių. Tačiau antroji programa (brūkšninė tiesė) veikia lėčiau, ir jos veikimo trukmė sparčiau didėja.



17 pav. Iteracijos trukmės priklausomybė nuo personažų skaičiaus

Paveikslėlyje *18 pav.* vaizduojama vieno personažo valdymo vienos iteracijos trukmė. Galima pastebėti, kad vieno personažo apdorojimo trukmė nepriklauso nuo vienu metu apdorojamų personažų skaičiaus: kai žaidime reikia valdyti du personažus, kiekvienam sugaištama po 0,052 ms, kai žaidime yra 20 žaidėjų – 0,051 ms. Antrosios programos vieno personažo apdorojimo laikas yra beveik 0,5 karto mažesnis.

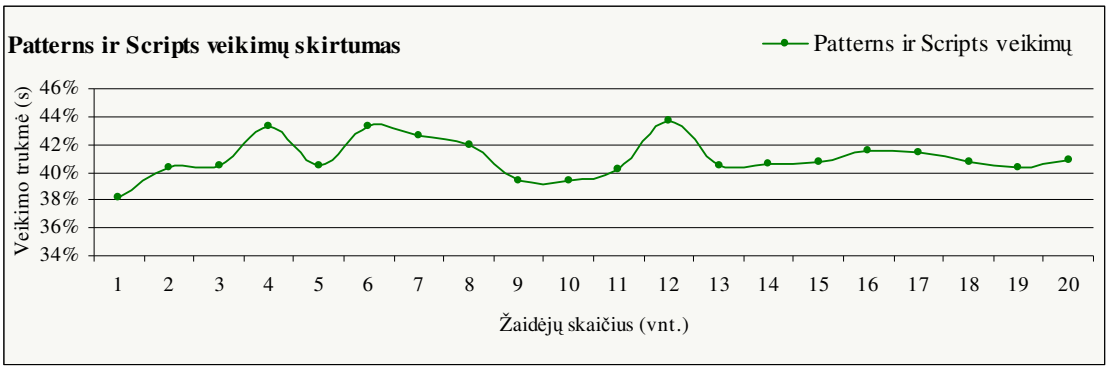
Taigi, galime daryti išvadą, kad vieno personažo valdymo (tiek pirmąja programa tiek antrąja) trukmė yra pastovi, nepriklauso nuo vienu metu apdorojamų personažų skaičiaus.



18 pav. Vieno personažo valdymo trukmės priklausomybė nuo personažų skaičiaus

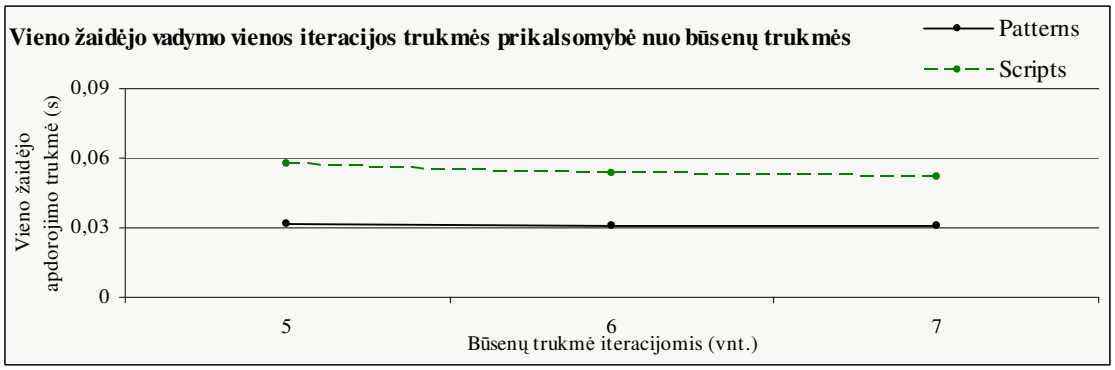
Paveikslėlyje *19 pav.* vaizduojamas abiejų programų veikimo trukmių palyginimas. Grafiką galima paaiškinti taip: pirmasis tiesės taškas (1, 38%) reiškia, kad valdant vieną personažą, antroji programa veikė 38% (nuo pirmosios veikimo trukmės) ilgiau nei pirmoji.

Taigi, matome, kad antroji programa veikia vidutiniškai 41% ilgiau nei pirmoji.



19 pav. Abiejų programų veikimo greičio palyginimas

Paveikslėlyje 20 pav. vaizduojamas bendras trijų testų rezultatas – personažo valdymo trukmės priklausomybė nuo būsenų trukmės. Matome, kad personažo valdymo laikas mažėja didėjant būsenų trukmei. Be to, antrosios programos valdymo trukmė mažėja sparčiau, lyginant su pirmąja.



20 pav. Personažo apdorojimo trukmės priklausomybė nuo būsenų trukmės

6. Išvados ir rekomendacijos

6.1. Išvados

Išanalizavę žaidimų DI funkcijas, apibrėžėme pagrindinius DI žaidimų varikliuko uždavinius, kurie gali būti naudojami skirtingų tipų žaidimuose: kelio paiešką ir sprendimų priėmimą.

Apžvelgę įvairius kelio paieškos ir sprendimų priėmimo metodus, išsirinkome labiausiai tinkamus: A* kelio paieškai ir BA sprendimų priėmimui.

Realizavome pagrindinius DI žaidimų varikliuko uždavinius. Šiems uždaviniams išspręsti naudojome duomenimis valdomą detaliąją architektūrą. Taip pasiekėme mums aktualų duomenų ir žaidimo logikos atskyrimą nuo bendro programos kodo. Kas suteikia galimybę laisvai manipuliuoti duomenimis ir logika neperkompiliuojant programos kodo.

Realizavimo metu sukūrėme pagalbinį įrankį būsenų aprašymams generuoti. Naudojant šį įrankį grafinės sąsajos pagalba galima greitai ir patogiai sukurti būsenų aprašus, neparašius nei vienos kodo eilutės. Būsenas su šiuo įrankių gali kurti asmenys, neturintys programavimo žinių.

Ištyrėme savo realizuotą sprendimų priėmimo programą. Palyginome jos veikimo greitį su įprastinių būdu realizuota sprendimų priėmimo programa. Iš tyrimo rezultatų, galime pastebėti, kad mūsų programa yra daug lankstesnė ir patogesnė naudoti. Tačiau ji veikia lėčiau už įprastinę programą, nes paprastų C++ funkcijų vykdymas veikia greičiau nei LUA scenarijų funkcijų vykdymas.

Šiuo metu mūsų realizuotų sprendimų priėmimo ir kelio paieškos dar negalima naudoti realiuose žaidimuose, nes jie netenkina žaidimams keliamų greičio reikalavimų. Jas derėtų optimizuoti.

6.2. Rekomendacijos

Mūsų realizuotą sprendimų priėmimą galima optimizuoti naudojant tikslų sąrašus (angl. *Goal Pipe*). Tokiu atveju užkrovus būseną personažui reikia ne vykdyti jos funkcijas, o

įtraukti funkcijas į funkcijų saugyklą (tikslų sąrašus), iš kurių vėliau funkcijos imamos personažams valdyti. Toks sprendimas yra realizuotas *CryTech* žaidimų varikliuke [19].

Mūsų realizuotą sprendimų priėmimą galima praplėsti ir papildyti nauju funkcionalumu:

- BA žinučių perdavimo/gavimo mechanizmas – realizavus tokį mechanizmą, žaidėjai ne tik patys galėtų atsinaujinti savo būsenas, bet ir atsirastų galimybė, kad kiti objektai iš išorės gali pranešti personažams, kad reikia atsinaujinti būsenas [20].
- BA praplėtimas animacijos vaizdavimui – galima sukurti sąryšį tarp personažų būsenų ir animacijos, kuri turi būti rodoma vykdant esamą būseną.

7. 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] Schwab, B. *AI Game Engine Programming*, Charles River Media, 2004, p. 593.

[4] Stout, W. *Smart moves: Intelligent Path-Finding* / *Gamasutra*, 1999 02 12, http://www.gamasutra.com/features/19990212/sm_01.htm.

[5] Simpson, J. *Game Engine Anatomy 101* / *ExtremeTech*, 2002 04 12, http://www.extremetech.com/article2/0,1558,594,00.asp.

[6] Stout, B. *The Basics of A* for Path Planning* / *Game Programming Gems*, Charles River Media, 2000, p. 614.

[7] Matthews, J. *Basic A* Pathfinding Made Simple* / *AI Game Programming Wisdom*, Charles River Media , 2002, p. 672.

[8] *A* Search Algorithm* / *Wikipedia, The Free Encyclopedia*, http://en.wikipedia.org/wiki/A%2A_search_algorithm, 2006 04 17.

[9] *3D Engines Database*, http://www.devmaster.net/engines/, 2004 11 12.

[10] Rosado, G. *Implementing a Data-Driven Finite-State Machine* / *AI Game Programming Wisdom 2*, Charles River Media, 2004, p. 732.

[11] Fu, D.; Houlette R. *The Ultimate Guide to FSMs in Games* / *AI Game Programming Wisdom 2*, Charles River Media, 2004, p. 732.

[12] Farris, Ch. *Function Pointer-Based, Embedded Finite-State Machines* / *Game Programming Gems 3*, Charles River Media, 2002, p. 663.

[13] Buckland, M. *Programming AI by Exmple*, Worldware Publishing, 2005, p. 495.

[14] *LUA Versus PYTHON* / *LUA-Users Wiki,* http://lua-users.org/wiki/LuaVersusPython, 2006 04 25.

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

[16] Gutschmidt, T. *Game Programming wiht Python, Lua and Ruby,* Premier Press, 2003, p. 472.

[17] Champandard, A. *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors,* New Riders Publishing, 2003, p. 744.

[18] Aimit, J. *A* Comparison / Amits Game Programming Site,* http://theory.stanford.edu/%7Eamitp/GameProgramming/, 2004 11 09.

[19] CryTech, *AI Manual ,* http://www.craytech.com, 2004 09 14.

[20] Rabin, S. *Designing a General Robust AI Engine* / *Game Programming Gems*, Charles River Media, 2000, p. 614.

8. Terminų ir santrumpų žodynas

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

8.1. Santrumpų žodynas

DI – dirbtinis intelektas (angl. *Artificial Intelligence*) – protingo virtualių žaidėjų elgesio imitavimas kompiuteriu.

BA – baigtiniai automatai (angl. *Finite-State Machines*) – deterministiniai baigtiniai būsenų automatai, DI varikliuke naudojami kompiuteriu valdomų žaidėjų elgesiui nusakyti.

8.2. Terminų žodynas

DI varikliukas (angl. *AI Engine*) – funkcijų ir įrankių visuma, naudojama kaip pagalbinė priemonė žaidimo dirbtiniam intelektui sukurti.

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

Sprendimų priėmimas – DI uždavinys, kuomet kompiuterių valdomam žaidėjui yra parenkamas tam tikras veiklos šablonas, pagal kurį formuojamas žaidėjo elgesys ir reakcija į aplinką, tariama, kad žaidėjas „priima sprendimą“ kokį veiksmą atlikti.

Atominiai veiksmai – tai nedalomi žaidėjų veiksmai, susidarantys iš vieno pobūdžio veiklos, pvz. bėgimas, ėjimas, šaudymas ir kiti.

Agentas – virtualus personažas žaidimo pasaulyje. Tai dažniausiai būna aktyvūs (besipriešinantys žaidėjui) kompiuteriu valdomi personažai, bet gali būti ir nežaidžiantys veikėjai.

Priedas 1

LUA programavimo kalbos apžvalga

Įvairiems DI elementams, logikai ar veiklai aprašyti labai dažnai yra naudojamos tokios supaprastintos programavimo kalbos (*Scripting Languages*), tokios kaip LUA, PYTHON, JAVA.

Šiame skyriuje trumpai supažindinama su LUA programavimo kalbos ypatumais ir aprašoma kaip galima įvykdyti informacijos pasikeitimą tarp LUA scenarijų ir C++ programos.

LUA apžvalga

LUA – per paskutinius penkerius metus sparčiai išpopuliarėjusi, supaprastinta programavimo kalba. LUA scenarijai buvo naudojami tokiose žaidimuose, kaip *Escape From Monkey Island, Homeworld 2, Far Cry, Baldur's Gate.* Ši kalba yra greita, turi didelį funkcionalumą, tačiau ją paprasta naudoti. Tai yra nemokamas, atviro kodo produktas. LUA yra lengvai įterpiama į kitas programavimo aplinkas.

LUA kalbos pagrindai

Aprašant kintamuosius LUA kalboje, nereikia nurodyti kintamojo tipo. LUA atpažįsta tik aštuonis tipus:

nil

nil skiriasi nuo visų kitų LUA tipų, jis yra naudojamas parodyti reikšmės nebuvimą. Jei norima ištrinti kažkokį kintamąjį, galima jam priskirti *nil* tipą.

number

Visi skaičiai LUA yra traktuojami kaip slankaus kablelio skaičiai.

string

Tai 8 bitų elementų masyvas. Su *string* tipo kintamaisiais galima atlikti sujungimo operaciją, ji žymima .. (du taškai).

```
--...
kaina=82
Print("Sios prekes kaina: "..kaina.." litai")
--...
```

Bus gautas toks atsakymas:

```
Sios prekes kaina: 82 litai
```

Programos fragmentas 1: LUA string *tipo kintamųjų pavyzdys*

table

Tai labai veiksmingas tipas. Jį galima įsivaizduoti kaip lentelę ar raktas-reikšmė tipo masyvą.

```
--vienmačio masyvo sukūrimas
lentele={}

--reikšmių suteikimas
lentele[1]=1
lentele[2]="Labas"

--vienmačio masyvo sukūrimas
lentele2={}

--reikšmių suteikimas
lentele2[„pirmas"]=1
lentele2[„antras"]="Labas"

--lentelės reikšmės keitimas
lentele2.pirmas=26

--dvimačio masyvo sukūrimas
dvimatis={}
dvimatis[1]={}
dvimatis[1][1]=0
dvimatis[1][2]=1
dvimatis[2]={1=1,2=0}
```

Programos fragmentas 2: LUA table *tipo galimybės*

function

LUA kalboje yra tipas *function*, kuris gali būti priskirtas kintamajam. Vėliau funkcija gali būti iškviečiama naudojant kintamojo vardą. Kadangi LUA kalboje kintamiesiems nereikia nurodyti tipų, tai ir funkcijų parametrams ar grąžinamai reikšmei tipų nurodyti nereikia. Funkcijos blokas baigiamas žodžiu *end*.

```
sudetis=function(d1,d2)
  return d1+d2
end

function sudetis(d1,d2)
  return d1+d2
end
```

Programos fragmentas 3: Du funkcijų aprašymo būdai LUA kalboje

user data

Šis tipas skirtas saugoti įvairius C/C++ klabos duomenis. Šio tipo kintamieji negali būti sukurti arba keičiami LUA kalboje, būtina naudoti C/C++ sąsajas.

thread

Šis tipas leidžia atskiras gijas palesti vykdymui atskirose srityse.

boolean

Loginis tipas, jo reikšmės *nil* ir 0 yra traktuojamos kaip „*false*“, visa kita – „*true*“.

LUA kalboje yar trys loginiai operatoriai: *and, or* ir *not.* Jų veikimas panašus į C++ loginių operatorių veikimą, antra sąlygos dalis yra tikrinama tik tada, kada tai yra būtina.

Taip pat LUA kalboje yra naudojamos sąlygų struktūros *if ... then ... else* bei ciklai *for, repeat* ir *while*. Jų sintaksė yra tokia pati kaip ir C++ analogiškų struktūrų sintaksė.

LUA integracija su C++

Norint C++ aplinkoje kviesti LUA funkcijas arba vykdyti LUA scenarijus, reikia į C++ programos kodą įtraukti tokias eilutes:

```
extern "C"
{
  #include <lua.h>
  #include <lualib.h>
  #include <luaxlib.h>
}
```

Programos fragmentas 4: LUA bibliotekų įtraukimas į C++ programą

Po bibliotekų įtraukimo, būtina pranešti kompiliatoriui, kad bus naudojamos LUA bibliotekos, tai galima padaryti dviem būdais: arba parašyti tokias kodo eilutes, kurios

pateiktos žemiau, arba nurodyti C++ projekto nustatymuose, kad bus naudojamos *lua.lib ir lualib.lib* bibliotekos.

```
#pragma comment(lib, "lua.lib")
#pragma comment(lib, "lualib.lib")
```

Programos fragmentas 5: LUA bibliotekų įtraukimas į C++ programą

Kiekvienas LUA scenarijus bus vykdomas struktūroje su dinamiškai išskirta atmintimi *lua_State*. Prieš kviečiant LUA scenarijus, reikia sukurti šią struktūrą.

```
lua_State *l=lua_open();
```

Programos fragmentas 6: LUA struktūros sukūrimas C++ programoje

Literatūra

[1] Mat Buckland, Programming Game AI by Example, Wordware Publishing, 2005, p. 495.

[2] „The Evolution of an Extension Language: A History of Lua“ / „SBLP 2001 invited paper“, http://www.lua.org/history.html, 2006 01 25.

Priedas 2

Būsenų automato aprašų generatorius. Naudojimo vadovas

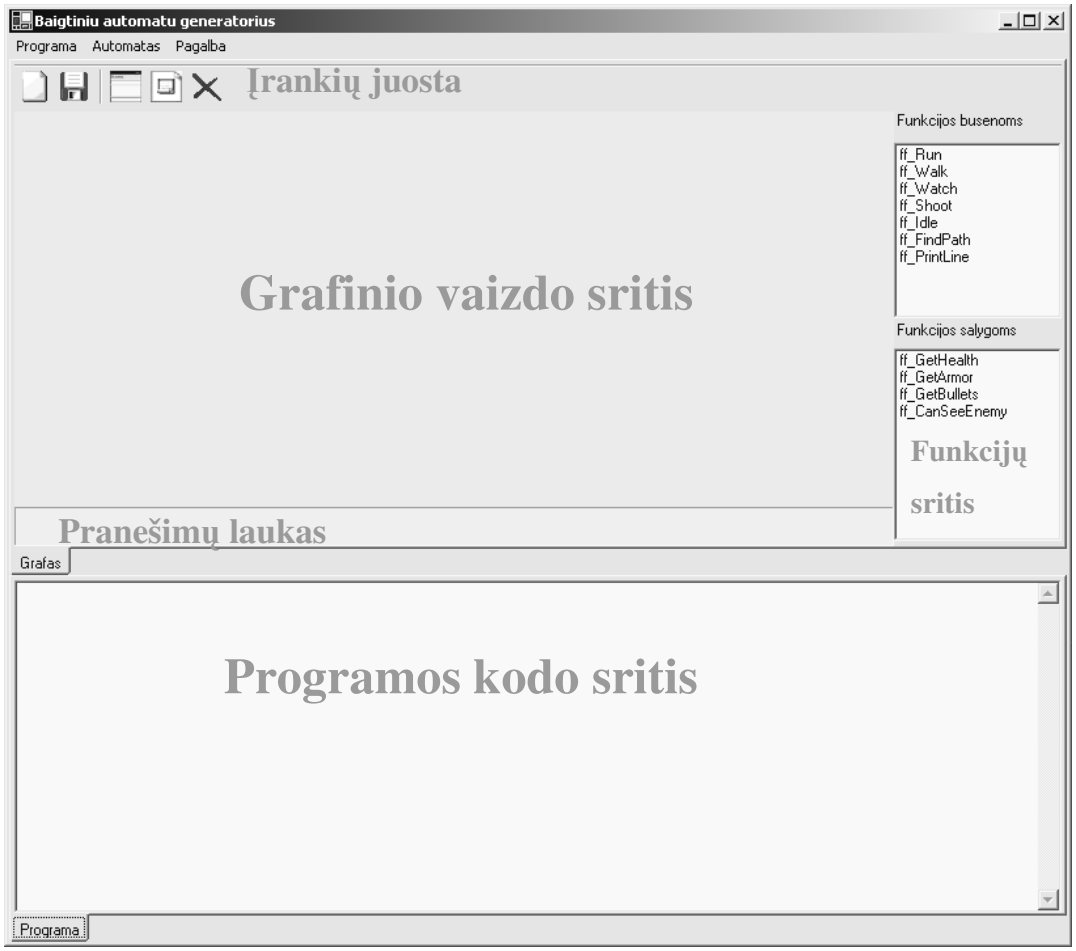
Įrankio aprašymas

Baigtinių būsenų automatų aprašų generavimo įrankis skirtas kurti automatų aprašams LUA kalba. Kiekviena būsena yra aprašoma skirtingoje rinkmenoje. Žaidimo metu šios rinkmenos yra užkraunamos, nuskaitomi jų aprašai ir žaidimų objektai valdomi pagal aprašuose pateiktas funkcijas.

Funkcijos, kurias vartotojas gali pasirinkti, yra realizuotos žaidimo kode C++ kalba. Žaidimų varikliuke yra pateiktas esminių funkcijų rinkinys, tačiau, jei vartotojas pageidauja šį rinkinį praplėsti ir yra kompetentingas, gali sukurti savo funkcijas (apie papildomų funkcijų kūrimą skaitykite šio skyriaus skyrelyje: 0 *Papildomų funkcijų būsenų automatui kūrimas*).

Įrankio pagrindiniai objektai, naudojami būsenoms kurti, yra aprašyti šio skyriaus skyrelyje: 0 *Įrankio objektų aprašymas*.

Įrankio pagrindinio lango vaizdas pateiktas žemiau, paveiksle 1. pav.: *Pagrindinis langas*.



1. pav.:*Pagrindinis langas*

Pagrindiniame įrankio lange matosi kelios esminės lango sritys, kurios aprašytos lentelėje Lentelė 1.: *Būsenų automato aprašo generavimo įrankio lango sritys.*

Lentelė 1.: *Būsenų automato aprašo generavimo įrankio pagrindinio lango sritys*

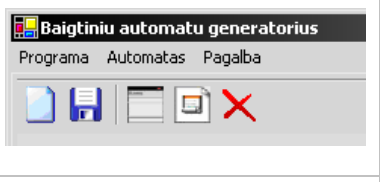
NR.	SRITIS	APRAŠYMAS
1	Įrankių juosta	Sparčiosios kreipties mygtukų juosta, kurioje iškelti svarbiausi ir dažniausiai naudojami įrankio veiksmai. Apie įrankių juostos mygtukų aprašymai pateikti lentelėje Lentelė 2.: *Būsenų automato aprašo generavimo įrankių juostos mygtukų aprašymai*
2	Grafinio vaizdo sritis	Tai sritis, kurioje vaizduojamas grafinis automato vaizdas. Šioje srityje galima tampyti būsenas, keisti perėjimų tarp būsenų sąlygas, gražiai ir patogiai

		išsidėlioti būsenas, kad matytųsi visas automatas.
3	Programos kodo sritis	Šioje srityje rodomas aktyviosios būsenos LUA kodas (aktyvavus būseną ir paspaudus kodo rodymo mygtuką). Šioje srityje galima kopijuoti kodo segmentus bei redaguoti kodą (jis bus išsaugotas)
4	Pranešimų laukas	Šiame lauke išvedama kontekstinė pagalba vartotojui, bei pranešimai apie klaidas (pvz., Apie negalimus perėjimus tarp būsenų)
5	Funkcijų sritis	Šioje srityje pateikiami funkcijų sąrašai. Yra du sąrašai: • Funkcijos būsenoms – tai funkcijos, kurias patogiausia įkelti į būsenas • Funkcijos sąlygoms – tai funkcijos, kurias rekomenduojama naudoti sąlygose

Visi svarbiausi įrankio veiksmai yra iškelti į įrankių juostą. Įrankių juostos mygtukų pagalba galima:

- sukurti / ištrinti būsenas
- išsaugoti automatą
- sukurti naują automatą
- peržiūrėti aktyvuotos būsenos kodą.

Įrankių juosta vaizduojame paveikslėlyje 2. pav.: *Įrankių juosta*.



2. pav.:*Įrankių juosta*

Visi įrankių juostos mygtukai ir veiksmai, kurie atliekami tų mygtukų paspaudimo metu, yra detaliai aprašyti žemiau, lentelėje Lentelė 2.: *Būsenų automato aprašo generavimo įrankių juostos mygtukų aprašymai.*

Lentelė 2.: *Būsenų automato aprašo generavimo įrankių juostos mygtukų aprašymai*

NR.	MYGTUKAS	PAVADINIMAS	APRAŠYMAS
1		Naujas automatas	Seno automato ištrynimas ir visų laukų išvalymas.
2		Išsaugoti automatą	Automato išsaugojimas. Kiekvienai automato būsenai sukuriama atskira rinkmena su būsenos aprašymu LUA kalba.
3		Nauja būsena	Naujos būsenos sukūrimas.
4		Parodyti būsenos kodą	Aktyvuotos būsenos kodo vaizdavimas
5		Ištrinti būseną	Būsenos trynimas.

Pagrindiniai veiksmai, kuriuos galima atlikti įrankio pagalba, yra šie:

- *Naujas automatas* – Seno automato ištrynimas ir visų lango sričių išvalymas. Prieš atliekant trynimą, paklausiama vartotojo ar išsaugoti automatą (jei automatas buvo kuriamas), jei vartotojas pasirenka atsakymą „taip“ – automatas išsaugomas, jei „ne“ – neišsaugomas. Vėliau, abiem atvejais, visos lango sritys yra išvalomos.

- *Išsaugoti automatą* – Sukuriamos rinkmenos, aprašančios kiekvieną būseną. Vartotojo paprašoma nurodyti esantį katalogą (arba sukurti naują), kuriame bus sukurtos rinkmenos su būsenų aprašymais (atskira rinkmena kiekvienai būsenai).

- *Nauja būsena* – Sukuriama nauja būsena, jai automatiškai suteikiamas pavadinimas, kurį vėliau rekomenduojama pakeisti.

- *Ištrinti būseną* – Prieš atliekant trynimą vartotojo klausiama ar jis tikras, kad nori ištrinti būseną. Jei atsakoma „Taip“ – būsena ir visi su ja susiję ryšiai yra ištrinami.

Norint uždaryti įrankį, neišsaugojus automato (jei jis buvo kurtas), vartotojui bus pasiūlyta išsaugoti sukurtą automatą.

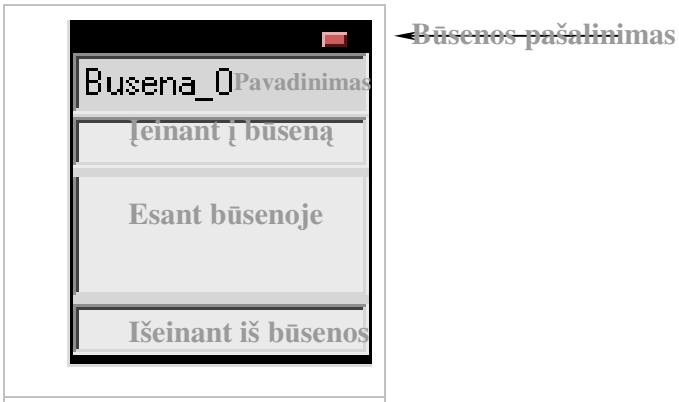
Įrankio objektų aprašymas

Automatą galima sudaryti naudojant du pagrindinius objektus:

- Būsena
- Perėjimas į kitą būseną

Automatų aprašų generavimo įrankio būsenos vaizdas pateiktas paveikslėlyje 3. pav.: *Būsena*. Kiekviena automato būsena turi turėti unikalų pavadinimą, šio vardu ir bus pavadinta būsenos aprašo rinkmena, pagali šį pavadinimą žaidimo metu bus kviečiamas aprašymas.

Būsena susidaro iš trijų sričių. Šios sritys yra detaliai aprašytos lentelėje Lentelė 3.: *Būsenų sritys*.



3. pav.:*Būsena*

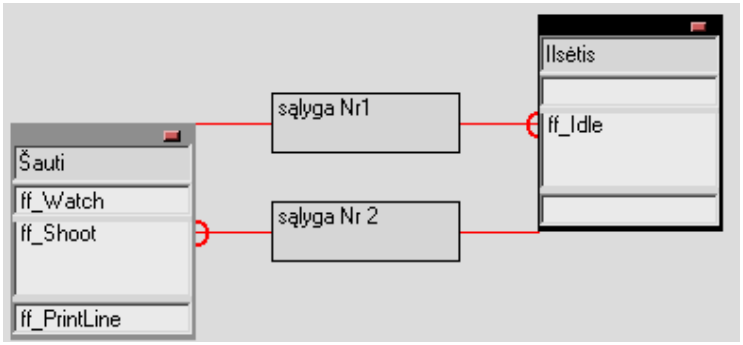
Lentelė 3.: *Būsenų sritys*

NR.	PAVADINIMAS	APRAŠYMAS	PASIKARTOJIMAI
1	Pavadinimas	Būsenos pavadinimas	-
2	Įeinant į būseną	Veiksmų sąrašas, kurie bus atliekami įėjus į būseną	Vieną kartą, įėjus į būseną
3	Esant būsenoje	Veiksmų sąrašas, kurie bus atliekami esant būsenoje	Daug kartų cikle, kol bus nepereita į kitą

			būseną
4	Išeinant iš būsenos	Veiksmų sąrašas, kurie bus atlikti išėjus iš būsenos	Vieną kartą, išėjus iš būsenos
5	Būsenos pašalinimas	Būsenos ir sąlygų, susietų su šią būsena pašalinimas iš automato	-

Į būsenos sritis gali būti nutemptos funkcijos iš funkcijų būsenoms sąrašo (apie funkcijų sritį galima perskaityti lentelėje Lentelė 1.: *Būsenų automato aprašo generavimo įrankio lango sritys*). Apie būsenų kūrimą galima perskaityti šio skyriaus skyrelyje 0 *Būsenų automato aprašų kūrimas*.

Tarp automato būsenų galima sukurti perėjimus. Perėjimai vaizduojami paveikslėlyje 4. pav.: *Perėjimai tarp būsenų.*



4. pav.:*Perėjimai tarp būsenų*

Šį paveikslėlį galima pakomentuoti taip: iš būsenos *Šauti* įvykus sąlygai *Sąlyga Nr. 1* pereinama į būseną *Ilsėtis*, iš būsenos *Ilsėtis* įvykus sąlygai *Sąlyga Nr. 2* pereinama į būseną *Šauti*. Apie sąlygos formavimą galima perskaityti šio skyriaus skyrelyje 0 *Būsenų automato aprašų kūrimas*.

Būsenų automato aprašų kūrimas

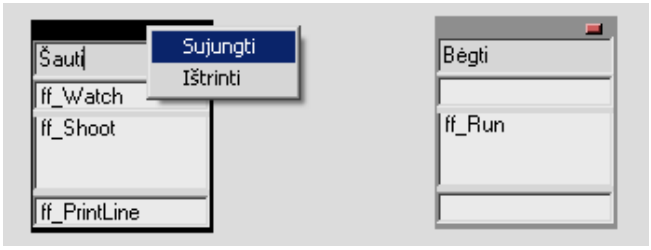
Norint sukurti būsenų automato aprašą, reikia atlikti žemiau pateiktus veiksmus.

BŪSENŲ KŪRIMAS:

1. Paspaudžiame įrankių juostos mygtuką *Nauja būsena*.
2. Atsiradusioje būsenoje pakeičiame pavadinimą.
3. Iš dešinėje esančio funkcijų būsenoms sąrašo įtempiame į norimas būsenos sritis norimas funkcijas.
4. Kartojame žingsnius 1-4 tiek, kiek norime sukurti būsenų.

PERĖJIMŲ TARP BŪSENŲ KŪRIMAS:

1. Paspaudžiame dešinį pelės klavišą ant būsenos, iš kurios norime sukurti perėjimą.
2. Iš atsiradusio kontekstinio meniu pasirenkame *Sujungti* (paveikslėlis 5. pav.: *Perėjimų tarp būsenų kūrimas*).



5. pav.:*Perėjimų tarp būsenų kūrimas*

3. Paspaudžiame kitą būseną, į kurią norime atlikti perėjimą.
4. Atsiradusiame stačiakampyje suformuojame norimą sąlygą.
5. Kartojame žingsnius nuo 1 iki 4 tiek, kiek norime sukurti perėjimų.

SĄLYGOS FORMAVIMAS:

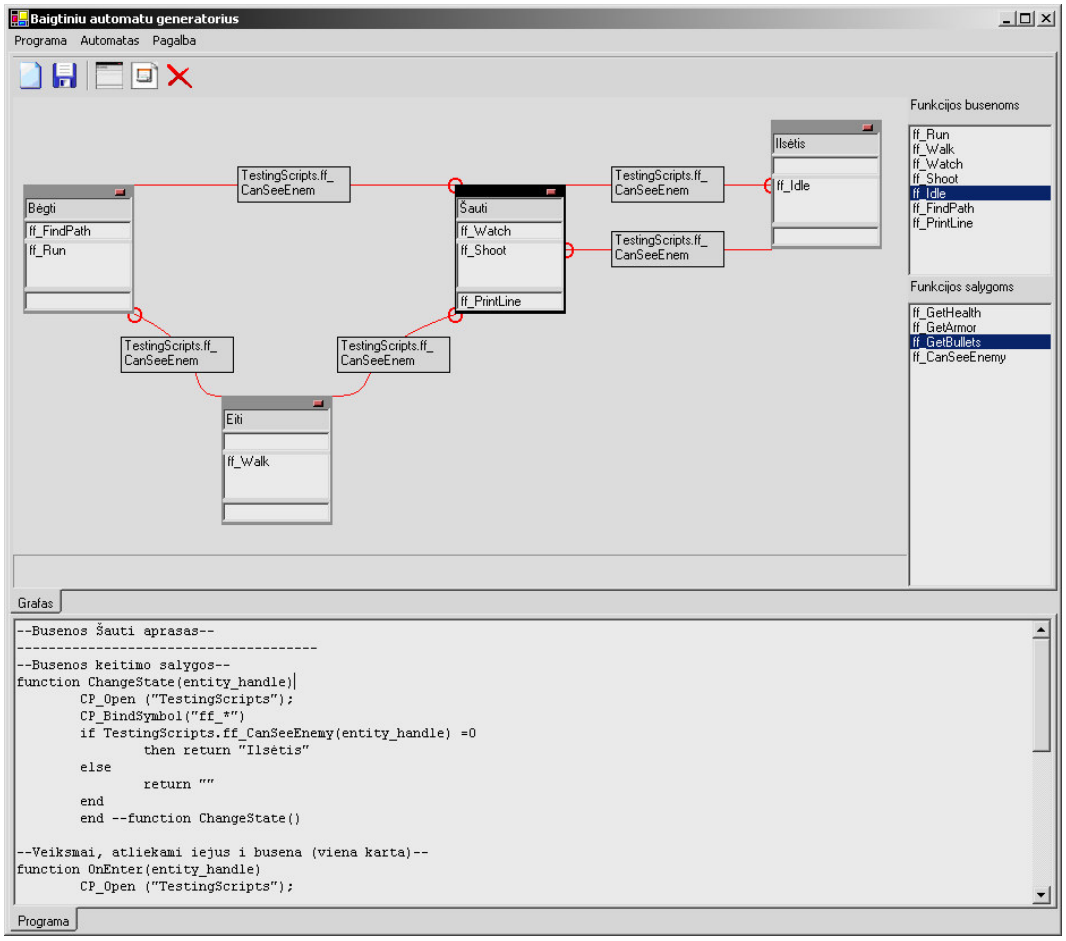
1. Paspaudžiame sąlygos, kurią norime suformuoti stačiakampį.
2. Į sąlygos stačiakampį nutempiame funkciją iš pagrindinio lango funkcijų sąlygoms sąrašo (apie funkcijų sritį galima perskaityti lentelėje Lentelė 1.: *Būsenų automato aprašo generavimo įrankio lango sritys*).
3. Įtempus funkciją, suformuojame sąlygą, naudojantis ženklais (<, >, =, <>), loginiais operatoriais (and, or) ir realiaisiais skaičiais (0, 1, 1.5, 10 ir t.t.). Pvz., įtempus funkcijas *ff_CanSeeEnemy* ir *ff_GetBullets*, galima suformuoti tokią sąlygą:

```
TestingScripts.ff_CanSeeEnemy(entity_handle)=1 AND
TestingScripts.ff_GetBullets(entity_handle)>10
```

Šią sąlygą galima paaiškinti taip: jei žaidėjas mato priešą ir turi daugiau nei 10 šovinių, pereiti į kitą būseną (tikriausiai puolimo).

4. Žingsnius nuo 1 iki 3 kartojame tiek, kiek turime tuščių sąlygų stačiakampių.

Atlikus visus auščiau išvardintus veiksmus, gaunamas baigtinis automatas (automato vaizdas pateiktas paveiksle 6. pav.: *Būsenų automatas*).



6. pav.:*Būsenų automatas*

Norint peržiūrėti tam tikros būsenos LUA kodą, reikia aktyvuoti būseną (paspausti ant būsenos su pele) ir paspausti įrankių juostos mygtuką *Rodyti būsenos kodą*. Kodo srityje (apie kodo sritį skaitykite šio dokumento lentelėje Lentelė 1.: *Būsenų automato aprašo generavimo įrankio langa sritys*) bus rodomas pasirinkto būsenos aprašas LUA kalba.

Norint išsaugoti būsenas, būtina paspausti mygtuką iš įrankių juostos *Išsaugoti* ir pasirinkti katalogą, į kurį bus sukurtos rinkmenos su būsenų aprašais.

Pavyzdinis būsenos aprašas:

```
--Busenos Šauti aprasas--
-------------------------------------
--Busenos keitimo salygos--
--Veiksmai, atliekami iejus i busena (viena karta)--
function OnEnter(entity_handle)
      CP_Open ("TestingScripts");
      CP_BindSymbol("ff_*")
      end --function OnEnter()

--Veiksmai, atliekami esant busenoje (cikle, kol nepakeicia busena)--
function OnExecute(entity_handle)
      CP_Open ("TestingScripts");
      CP_BindSymbol("ff_*")
      TestingScripts.ff_Shoot(entity_handle)
      end --function OnExecute()

--Veiksmai, atliekami pries pakeiciant busena (viena karta)--
function OnExit(entity_handle)
      CP_Open ("TestingScripts");
      CP_BindSymbol("ff_*")
      end --function OnExit()
```

Papildomų funkcijų būsenų automatui kūrimas

Varikliuke yra numatyta funkcijų, kurios gali būti vykdomos būsenose arba gali būti naudojamos sąlygose, bazė. Tačiau, jei vartotojas pageidauja ir yra pakankamai kompetentingas (moka programuoti C++ kalba) gali sukurti savo išskirtinių funkcijų. Kūriant tokias funkcijas būtina patenkinti šias sąlygas:

- Funkcijos pavadinimas turi prasidėti taip: „*ff_*“ (pvz., *ff_RunForCover*)
- Funkcijos turi turėti vieną parametrą *integer* tipo (į kurį bus duodamas kompiuterių valdomo objekto identifikacinis numeris).

Funkcijas rekomenduojama rašyti atskiroje rinkmenoje, kurios antraštė įtraukiama į pagrindinę žaidimo rinkmeną.

Apribojimai ir galimos klaidos

Šiuo įrankiu kuriami automatai turi šiokius tokius apribojimus. Apribojimai ir klaidų pranešimai, kurie rodomi pažeidus apribojimus ir pagalba kaip išvengti apribojimų pažeidimų pateikti lentelėje Lentelė 4.: *Sistemos klaidos.*

Lentelė 4.: *Sistemos klaidos*

NR.	KLAIDA	KAIP TAISYTI?
1	Būsenos pavadinimas turi būti unikalus, jau egzistuoja būsena su tokiu pavadinimu.	Pakeisti būsenos pavadinimą į tokį, kuriuo dar nepavadina nei viena automato būsena
2	Toks ryšys jau egzistuoja. Iš vienos būsenos turi būti tik vienas perėjimas.	Būtina suformuoti perėjimo sąlygą taip, kad ji apimtus visus norimus perėjimus.
3	Visos būsenos turi turėti bent po vieną sąryšį su kitomis būsenomis.	Reikia įvesti naują sąryšį su kuria nors iš būsenų.
4	Automatas turi susidaryti bent iš vienos būsenos.	Negalima išsaugoti automato, kuriame nėra nei vienos būsenos.
5	Būsenoje turi būti kviečiama bent viena funkcija.	Ištrinti būsenas be funkcijų, arba įkelti į būseną funkcijas iš sąrašo

Priedas 3

Tyrimo rezultatai. Naudojimo patogumas

Išanalizavome dviejų programų naudojimo patogumą, toje pačioje aplinkoje su tais pačiais duomenimis.

Pirmoji programa – projektavimo šablonais realizuotas baigtinis automatas, atliekantis žaidimo personažų valdymą (įprastinė baigtinių automatų realizacija). Šios programos duomenys ir žaidimo logika yra įprogramuoti į patį žaidimo kodą.

Antroji programa – tai mūsų modifikuotas baigtinis automatas, realizuotas naudojant duomenimis valdomą detaliąją architektūrą. Šioje programoje duomenys ir žaidimo logika yra atskirti nuo žaidimo kodo, nukelti į LUA.

Palyginome ir aprašėme abiejų programų naudojimo patogumą ir veikimo greičius.

Programų naudojimo patogumas

Pateikiame būsenų aprašymai sukurti su pirmąja programa (būsenų programos kodas) ir antrąja (būsenų aprašų generavimo įrankio vaizdas).

1. Projektavimo šablonais realizuotas BA

Programos kodo antraštinė rinkmena (angl. *Header File*)

```
class BaseState
{
  // Methods
   public:
      virtual void ExecuteState(AIManager *manager, Entity *entity) =0;
      virtual void ChangeState(AIManager *manager, Entity *entity) =0;
};


// Patrol
/* ======================== */
class Patrol : public BaseState
{
   public:
      void ChangeState(AIManager *manager, Entity *entity);
      void ExecuteState(AIManager *manager, Entity *entity);
};

// RunAway
/* ======================== */
class RunAway : public BaseState
{
   private:
```

```
      void ChangeState(AIManager *manager, Entity *entity);
   public:
      void ExecuteState(AIManager *manager, Entity *entity);
};

// Attack
/* ======================== */
class Attack : public BaseState
{
   private:
      void ChangeState(AIManager *manager, Entity *entity);
   public:
      void ExecuteState(AIManager *manager, Entity *entity);
};

// Refill
/* ======================== */
class Refill : public BaseState
{

   public:
      void ExecuteState(AIManager *manager, Entity *entity);
      void ChangeState(AIManager *manager, Entity *entity);
};
```

Programos kodo išeities tekstų rinkmena (angl. *Source File*)

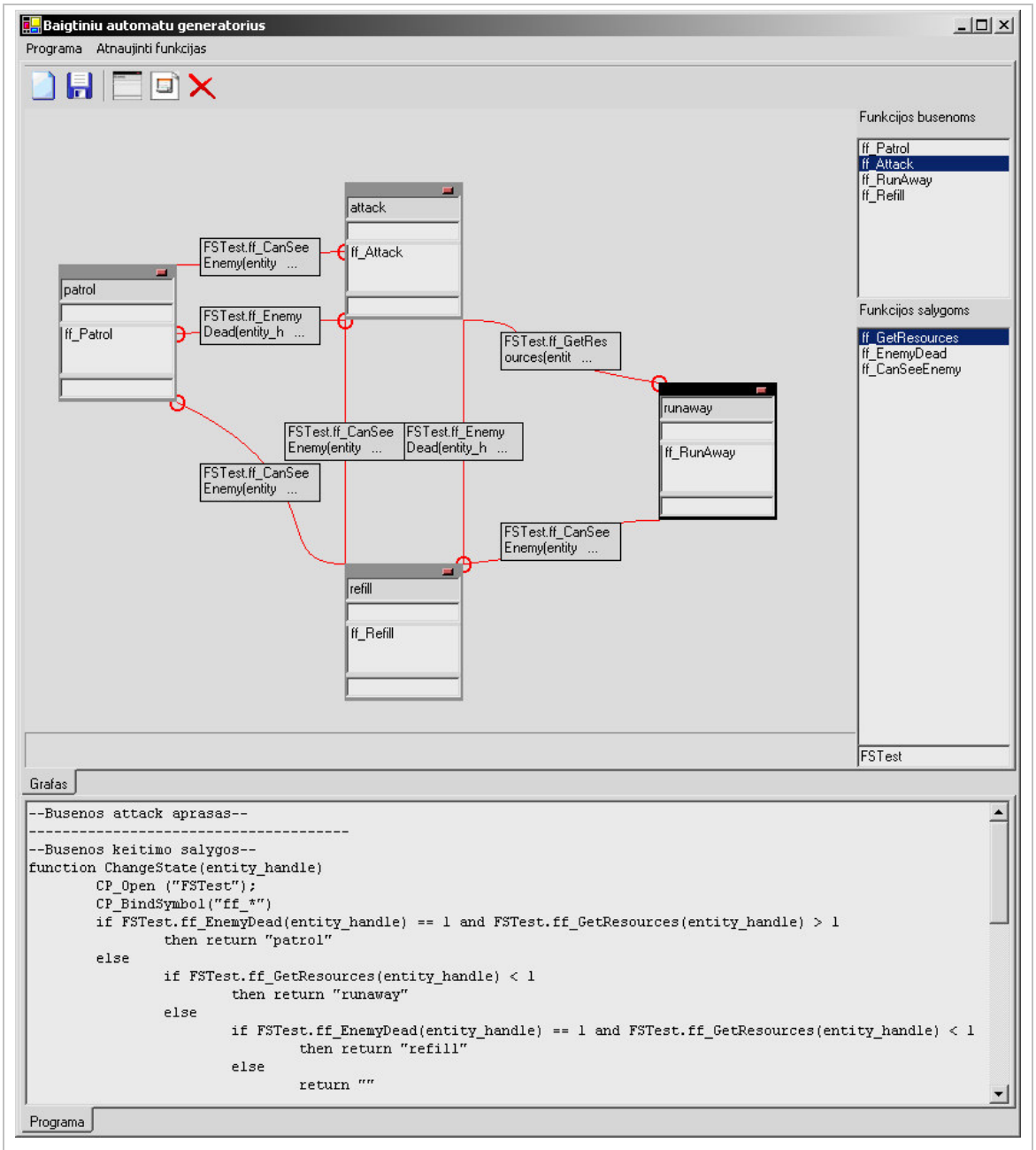
```
// Patrol
/* ======================== */
void Patrol::ChangeState(AIManager *manager, Entity *entity)
{
   if ((ff_CanSeeEnemy(entity->entity_handle)==1)&&(ff_GetResources(entity->entity_handle)>1))
   {
      entity->current_state = new Attack();
      entity->switched = true;
   }
}
void Patrol::ExecuteState(AIManager *manager, Entity *entity)
{
   if (entity->check_updates>update_needed)
   {
        this->ChangeState(manager, entity);
      if (!entity->switched)
      {
         ff_Patrol(entity->entity_handle);
         entity->check_updates++;
      }
   }
   else
   {
      ff_Patrol(entity->entity_handle);
      entity->check_updates++;
   }
}

// Refill
/* ======================== */
void Refill::ChangeState(AIManager *manager, Entity *entity)
{
   if ((ff_CanSeeEnemy(entity->entity_handle)==0)&&(ff_GetResources(entity->entity_handle)>1))
   {
      entity->current_state = new Patrol();
      entity->switched = true;
   }
   else if ((ff_CanSeeEnemy(entity->entity_handle)==1)&&(ff_GetResources(entity-
>entity_handle) >1))
   {
      entity->current_state = new Attack();
      entity->switched = true;
   }
}
void Refill::ExecuteState(AIManager *manager, Entity *entity)
{
   if (entity->check_updates>update_needed)
   {
      this->ChangeState(manager, entity);
      if (!entity->switched)
```

```
      {
         ff_Refill(entity->entity_handle);
         entity->check_updates++;
      }
   }
   else
   {
      ff_Refill(entity->entity_handle);
      entity->check_updates++;
   }
}
// Attack
/* ======================== */
void Attack::ChangeState(AIManager *manager, Entity *entity)
{
   if ((ff_EnemyDead(entity->entity_handle) == 1) && (ff_GetResources(entity->entity_handle) >
1))
   {
      entity->current_state = new Patrol();
      entity->switched = true;
   }
   else if (ff_GetResources(entity->entity_handle) < 1)
   {
      entity->current_state = new RunAway();
      entity->switched = true;
   }
   else if ((ff_EnemyDead(entity->entity_handle) == 1) && (ff_GetResources(entity-
>entity_handle) < 1))
   {
      entity->current_state = new Refill();
      entity->switched = true;
   }
}
void Attack::ExecuteState(AIManager *manager, Entity *entity)
{
   if (entity->check_updates>update_needed)
   {
      this->ChangeState(manager, entity);
      if (!entity->switched)
      {
         ff_Attack(entity->entity_handle);
         entity->check_updates++;
      }
   }
   else
   {
      ff_Attack(entity->entity_handle);
      entity->check_updates++;
   }
}
// RunAway
/* ======================== */
void RunAway::ChangeState(AIManager *manager, Entity *entity)
{
   if (ff_CanSeeEnemy(entity->entity_handle) == 0)
   {
      entity->current_state = new Refill();
      entity->switched = true;
   }
}
void RunAway::ExecuteState(AIManager *manager, Entity *entity)
{
   if (entity->check_updates>update_needed)
   {
      this->ChangeState(manager, entity);
      if (!entity->switched)
      {
         ff_RunAway(entity->entity_handle);
         entity->check_updates++;
      }
   }
   else
   {
      ff_RunAway(entity->entity_handle);
      entity->check_updates++;
   }
}
```

2. Būsenų aprašų generavimo įrankio pagalba sukurtas BA ir būsenų aprašai

Paveikslėlyje *1.pav.* vaizduojamas to paties automato, kuris aprašytas pirmame punkte, grafinis vaizdas, sukurtas naudojantis mūsų BA aprašų generavimo šrankiu.

Apatinėje formos srityje matome automatiškai sugeneruoto būsenos aprašymo LUA scenarijų kalba.



1.pav. Būsenų automato grafinis vaizdas

Priedas 4

Tyrimo rezultatai. Programų veikimo greičiai

Tyrėme dviejų programų veikimo greitį toje pačioje aplinkoje su tais pačiais duomenimis.

Pirmoji programa – projektavimo šablonais realizuotas baigtinis automatas, atliekantis žaidimo personažų valdymą (įprastinė baigtinių automatų realizacija). Šios programos duomenys ir žaidimo logika yra įprogramuoti į patį žaidimo kodą.

Antroji programa – tai mūsų modifikuotas baigtinis automatas, realizuotas naudojant duomenimis valdomą detaliąją architektūrą. Šioje programoje duomenys ir žaidimo logika yra atskirti nuo žaidimo kodo, nukelti į LUA.

Palyginome ir aprašėme abiejų programų veikimo greičius.

Tyrimo aplinka

Abiejų programų veikimo greitis buvo tiriamas viename kompiuteryje. Jo techninės ir programinės įrangos charakteristikos išvardintos žemiau.

- Techninės charakteristikos yra šios:
  - Procesorius – Intel Pentium M, 1.73 GHz
  - Operatyvioji atmintis – 504 MB
  - Kietasis diskas – Toshiba MK6026GAX, 55.8 GB
- Operacinė sistema – Microsoft Windows Server 2003 Enterprise Editon Service.

Tyrimo aprašymas

Atlikome tris testus. Kiekvieno testo metu abi programas vykdėme 100 iteracijų cikle. Tyrėme programų veikimą, apdorojant baigtinį automatą iš 4 būsenų ir 7 perėjimų tarp jų.

Kiekvieno testo metu nuosekliai didinome valdomų personažų skaičių. Testai vienas nuo kito skiriasi būsenų gyvavimo trukme. Būsenų gyvavimo trukmė – tai iteracijų skaičius, kuris nusako kiek ilgai galima šią būseną neatnaujinti. Pirmojo testo metu būsenos trukmė – 5 iteracijos, antrojo – 6, trečiojo – 7.

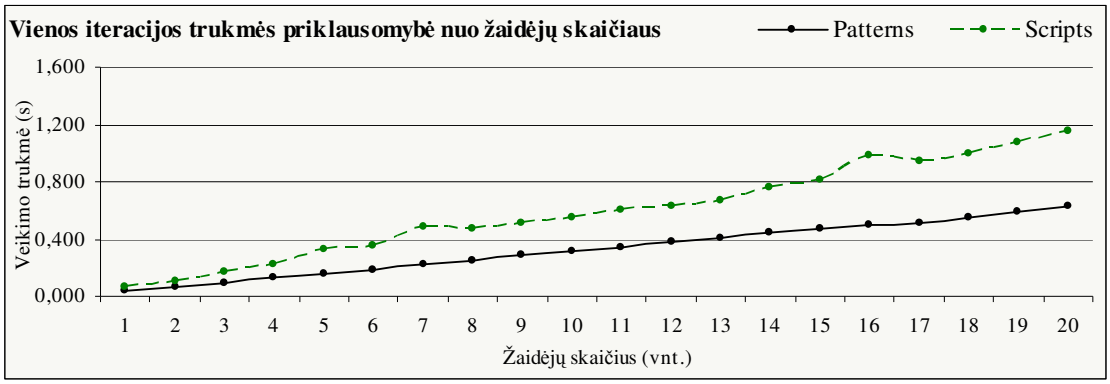
Testas Nr. 1

Pirmojo testo metu apdorojamas baigtinis automatas iš 4 būsenų ir 7 perėjimų tarp jų. Nuosekliai didinama valdomų personažų skaičius nuo 1 iki 20. Būsenos trukmė – 5 iteracijos.

Lentelėje žemiau pateikiami tyrimo rezultatai.

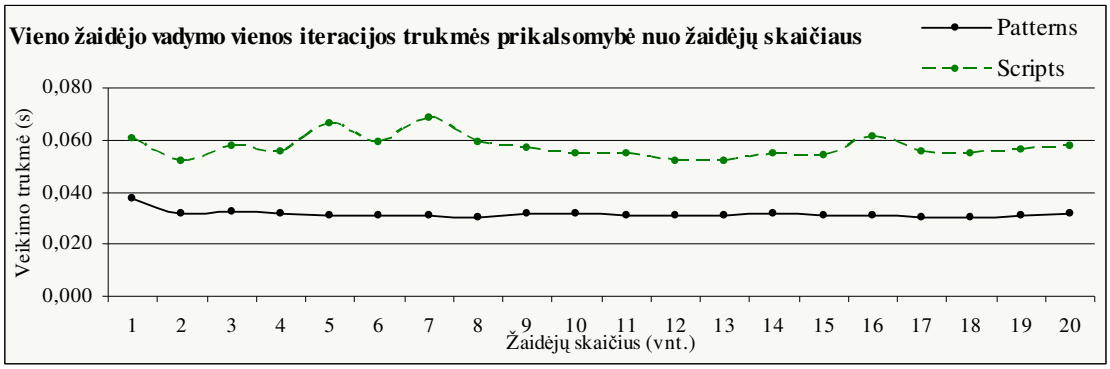
Žaidėjai	Iteracijos	Patterns 100 iteracijų	Scripts 100 iteracijų	Pattern 1 iteracija	Scripts 1 iteracija	Patterns 1 iteracija 1 žaidėjas	Scripts 1 iteracija 1 žaidėjas	Scriptai greitesni tiek procentų
1	100	3,734	6,050	0,037	0,060	0,037	0,060	38%
2	100	6,391	10,450	0,064	0,105	0,032	0,052	39%
3	100	9,703	17,300	0,097	0,173	0,032	0,058	44%
4	100	12,781	22,238	0,128	0,222	0,032	0,056	43%
5	100	15,500	33,250	0,155	0,332	0,031	0,066	53%
6	100	18,656	35,588	0,187	0,356	0,031	0,059	48%
7	100	21,797	48,088	0,218	0,481	0,031	0,069	55%
8	100	24,484	47,400	0,245	0,474	0,031	0,059	48%
9	100	28,656	51,138	0,287	0,511	0,032	0,057	44%
10	100	31,922	54,475	0,319	0,545	0,032	0,054	41%
11	100	33,844	60,174	0,338	0,602	0,031	0,055	44%
12	100	37,547	62,338	0,375	0,623	0,031	0,052	40%
13	100	40,625	67,513	0,406	0,675	0,031	0,052	40%
14	100	44,156	76,450	0,442	0,765	0,032	0,055	42%
15	100	46,594	81,375	0,466	0,814	0,031	0,054	43%
16	100	49,329	97,874	0,493	0,979	0,031	0,061	50%
17	100	51,719	94,262	0,517	0,943	0,030	0,055	45%
18	100	54,515	99,100	0,545	0,991	0,030	0,055	45%
19	100	59,375	107,338	0,594	1,073	0,031	0,056	45%
20	100	63,125	116,012	0,631	1,160	0,032	0,058	46%
	vidurkiai	32,72265	59,42064	0,32723	0,594206	0,031573	0,057234	45%

Paveikslėlyje *1 pav* vaizduojama iteracijos apdorojimo trukmės priklausomybė nuo žaidėjų skaičiaus. Matome, abiejų programų veikimo trukmė didėja, didinant personažų skaičių. Tačiau antroji programa veikia lėčiau, ir jos veikimo trukmė sparčiau didėja.



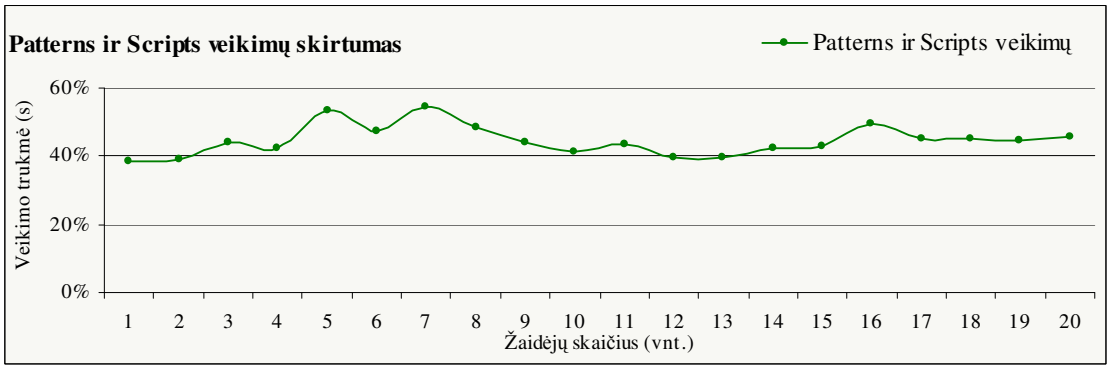
1 pav. Iteracijos trukmės priklausomybė nuo žaidėjų skaičiaus

Paveikslėlyje *2 pav.* vaizduojama vieno personažo valdymo vienos iteracijos trukmė. Galima pastebėti, kad vieno personažo apdorojimo trukmė nepriklauso nuo vienu metu apdorojamų personažų skaičiaus.



2 pav. Žaidėjo iteracijos trukmės priklausomybė nuo žaidėjų skaičiaus

Paveikslėlyje *3 pav.* vaizduojamas abiejų programų veikimo trukmių palyginimas. Grafiką galima paaiškinti taip: pirmasis tiesės taškas (1, 40%) reiškia, kad valdant vieną personažą, antroji programa veikė 40% (nuo pirmosios veikimo trukmės) ilgiau nei pirmoji.



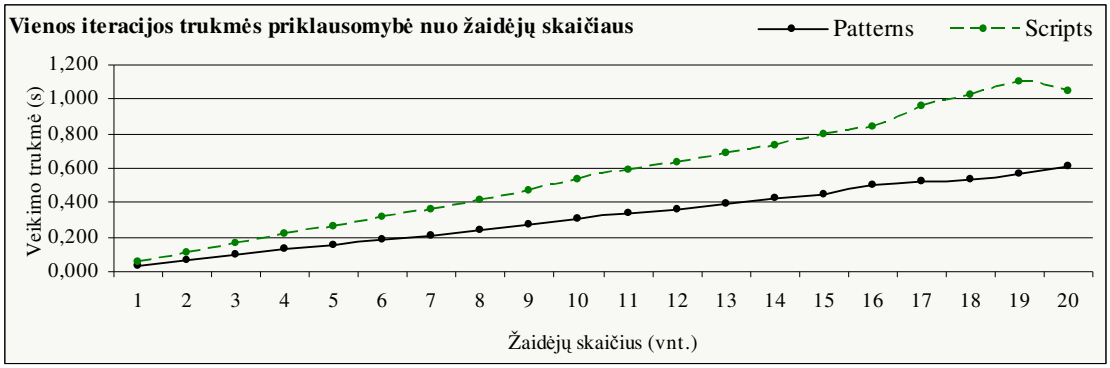
3 pav. Abiejų programų veikimo greičiai

Testas Nr. 2

Pirmojo testo metu apdorojamas baigtinis automatas iš 4 būsenų ir 7 perėjimų tarp jų. Nuosekliai didinama valdomų personažų skaičius nuo 1 iki 20. Būsenos trukmė – 6 iteracijos.

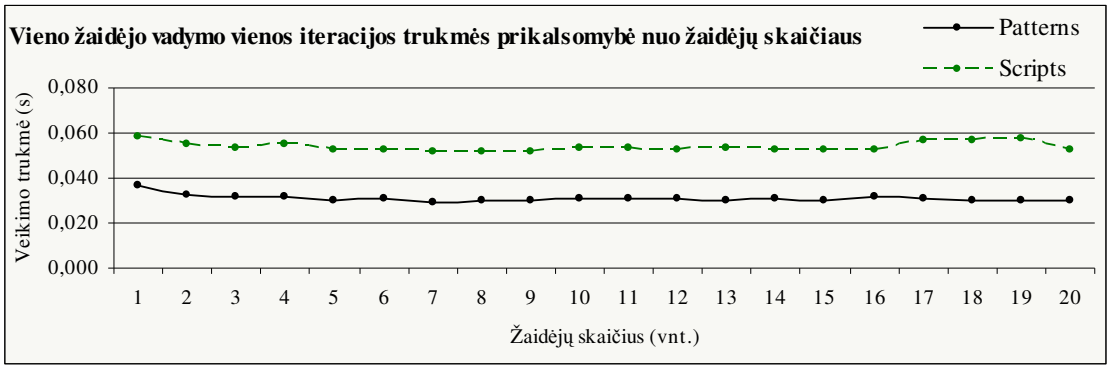
Žaidėjai	Iteracijos	Patterns 100 iteracijų	Scripts 100 iteracijų	Pattern 1 iteracija	Scripts 1 iteracija	Patterns 1 iteracija 1 žaidėjas	Scripts 1 iteracija 1 žaidėjas	Scriptai greitesni tiek procentų
1	100	3,656	5,813	0,037	0,058	0,037	0,058	37%
2	100	6,469	10,925	0,065	0,109	0,032	0,055	41%
3	100	9,422	16,013	0,094	0,160	0,031	0,053	41%
4	100	12,641	22,112	0,126	0,221	0,032	0,055	43%
5	100	15,078	26,313	0,151	0,263	0,030	0,053	43%
6	100	18,421	31,312	0,184	0,313	0,031	0,052	41%
7	100	20,188	36,438	0,202	0,364	0,029	0,052	45%
8	100	23,890	41,526	0,239	0,415	0,030	0,052	42%
9	100	27,203	46,475	0,272	0,465	0,030	0,052	41%
10	100	31,031	53,425	0,310	0,534	0,031	0,053	42%
11	100	33,907	58,487	0,339	0,585	0,031	0,053	42%
12	100	36,516	63,062	0,365	0,631	0,030	0,053	42%
13	100	39,297	69,238	0,393	0,692	0,030	0,053	43%
14	100	42,672	73,013	0,427	0,730	0,030	0,052	42%
15	100	45,000	79,238	0,450	0,792	0,030	0,053	43%
16	100	50,016	84,100	0,500	0,841	0,031	0,053	41%
17	100	51,875	96,187	0,519	0,962	0,031	0,057	46%
18	100	53,906	102,075	0,539	1,021	0,030	0,057	47%
19	100	56,734	109,962	0,567	1,100	0,030	0,058	48%
20	100	60,735	104,950	0,607	1,049	0,030	0,052	42%
vidurkiai		31,93285	56,53312	0,31933	0,565331	0,030832	0,0537703	43%

Paveikslėlyje *4 pav* vaizduojama iteracijos apdorojimo trukmės priklausomybė nuo apdorojamų žaidėjų skaičiaus. Matome, abiejų programų veikimo trukmė didėja tiesiškai, didinant personažų skaičių. Tačiau antroji programa (brūkšninė tiesė) veikia lėčiau, ir jos veikimo trukmė sparčiau
didėja.



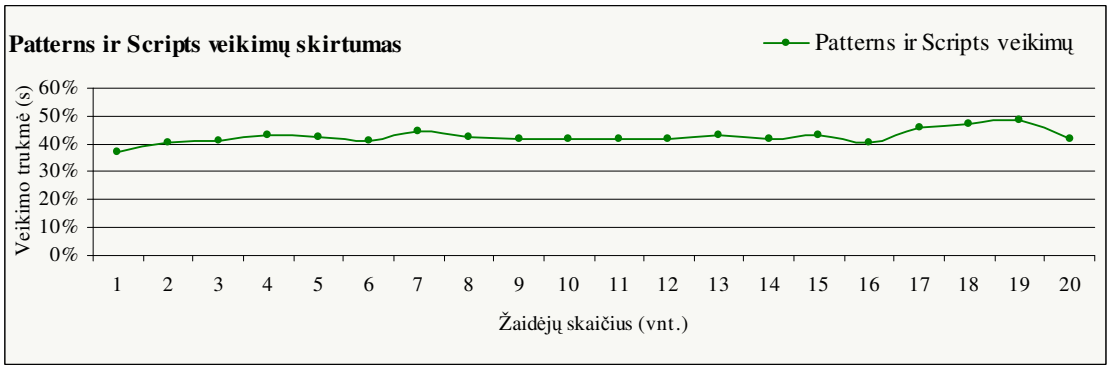
4 pav. Iteracijos trukmės priklausomybė nuo žaidėjų skaičiaus

Paveikslėlyje *5 pav.* vaizduojama vieno personažo valdymo vienos iteracijos trukmė. Galima pastebėti, kad vieno personažo apdorojimo trukmė nepriklauso nuo vienu metu apdorojamų personažų skaičiaus.



5 pav. Žaidėjo iteracijos trukmės priklausomybė nuo žaidėjų skaičiaus

Paveikslėlyje *6 pav.* vaizduojamas abiejų programų veikimo trukmių palyginimas. Grafiką galima paaiškinti taip: pirmasis tiesės taškas (1, 39%) reiškia, kad valdant vieną personažą, antroji programa veikė 39% (nuo pirmosios veikimo trukmės) ilgiau nei pirmoji.



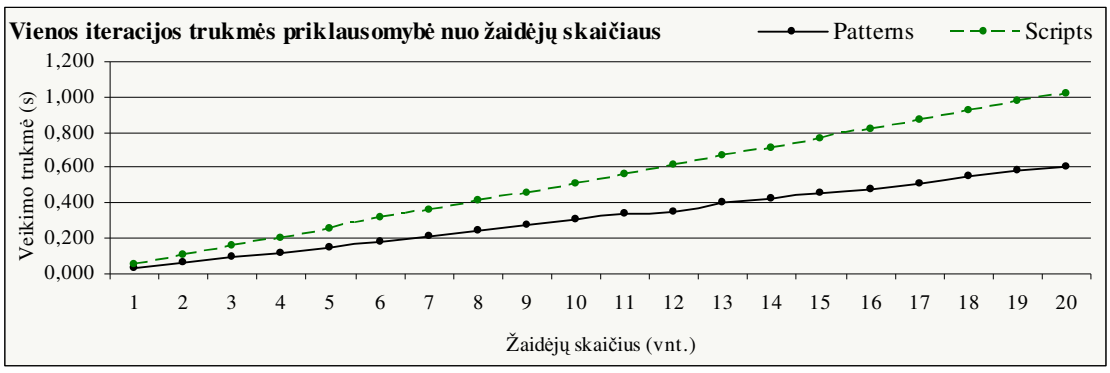
6 pav. Abiejų programų veikimo greičiai

Testas Nr. 3

Pirmojo testo metu apdorojamas baigtinis automatas iš 4 būsenų ir 7 perėjimų tarp jų. Nuosekliai didinama valdomų personažų skaičius nuo 1 iki 20. Būsenos trukmė – 7 iteracijos.

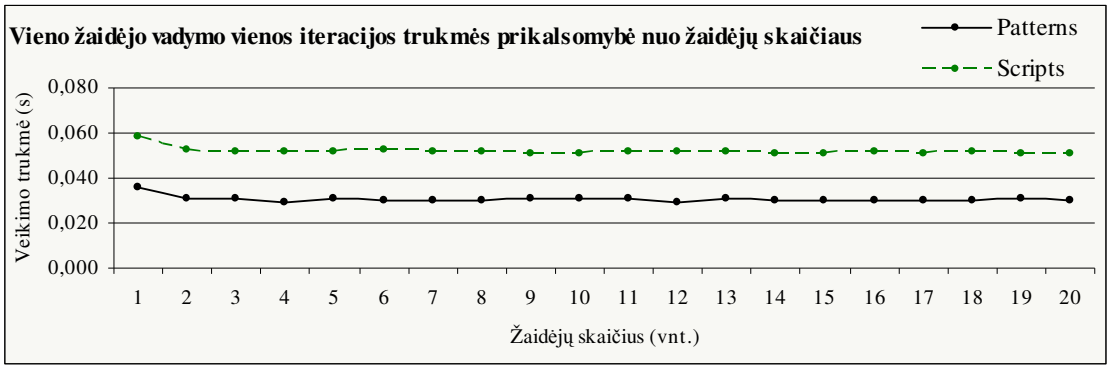
Žaidėjai	Iteracijos	Patterns 100 iteracijų	Scripts 100 iteracijų	Pattern 1 iteracija	Scripts 1 iteracija	Patterns 1 iteracija 1 žaidėjas	Scripts 1 iteracija 1 žaidėjas	Scriptai greitesni tiek procentų
1	100	3,609	5,832	0,036	0,058	0,036	0,058	38%
2	100	6,218	10,425	0,062	0,104	0,031	0,052	40%
3	100	9,234	15,513	0,092	0,155	0,031	0,052	40%
4	100	11,656	20,575	0,117	0,206	0,029	0,051	43%
5	100	15,328	25,750	0,153	0,257	0,031	0,051	40%
6	100	17,828	31,450	0,178	0,314	0,030	0,052	43%
7	100	20,907	36,450	0,209	0,365	0,030	0,052	43%
8	100	23,922	41,200	0,239	0,412	0,030	0,052	42%
9	100	27,891	45,975	0,279	0,460	0,031	0,051	39%
10	100	30,703	50,713	0,307	0,507	0,031	0,051	39%
11	100	33,937	56,813	0,339	0,568	0,031	0,052	40%
12	100	34,797	61,788	0,348	0,618	0,029	0,051	44%
13	100	40,078	67,338	0,401	0,673	0,031	0,052	40%
14	100	42,453	71,462	0,425	0,715	0,030	0,051	41%
15	100	45,500	76,762	0,455	0,768	0,030	0,051	41%
16	100	48,046	82,238	0,480	0,822	0,030	0,051	42%
17	100	50,969	87,013	0,510	0,870	0,030	0,051	41%
18	100	54,734	92,412	0,547	0,924	0,030	0,051	41%
19	100	58,015	97,175	0,580	0,972	0,031	0,051	40%
20	100	60,078	101,575	0,601	1,016	0,030	0,051	41%
	vidurkiai	31,795	53,923	0,318	0,539	0,031	0,052	41%

Paveikslėlyje *7 pav.* vaizduojama iteracijos apdorojimo trukmės priklausomybė nuo apdorojamų žaidėjų skaičiaus. Matome, abiejų programų veikimo trukmė didėja tiesiškai, didinant personažų skaičių. Tačiau antroji programa (brūkšninė tiesė) veikia lėčiau, ir jos veikimo trukmė sparčiau didėja.



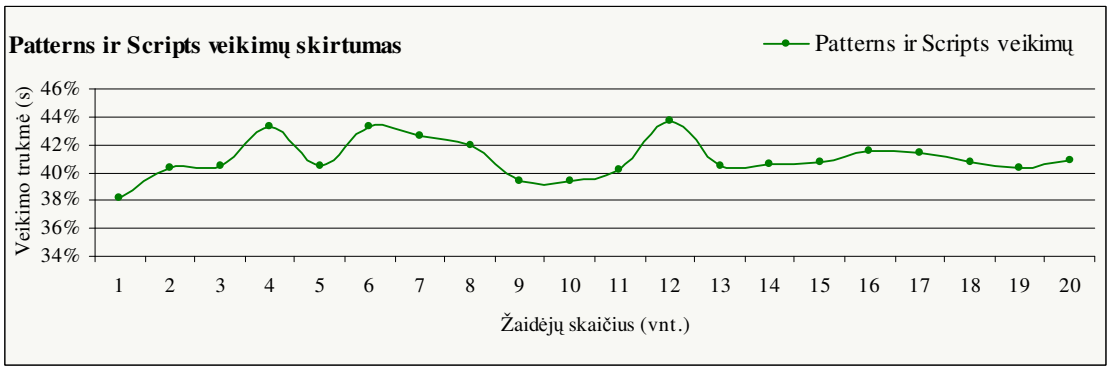
7 pav. Iteracijos trukmės priklausomybė nuo žaidėjų skaičiaus

Paveikslėlyje *8 pav.* vaizduojama vieno personažo valdymo vienos iteracijos trukmė. Galima pastebėti, kad vieno personažo apdorojimo trukmė nepriklauso nuo vienu metu apdorojamų personažų skaičiaus.



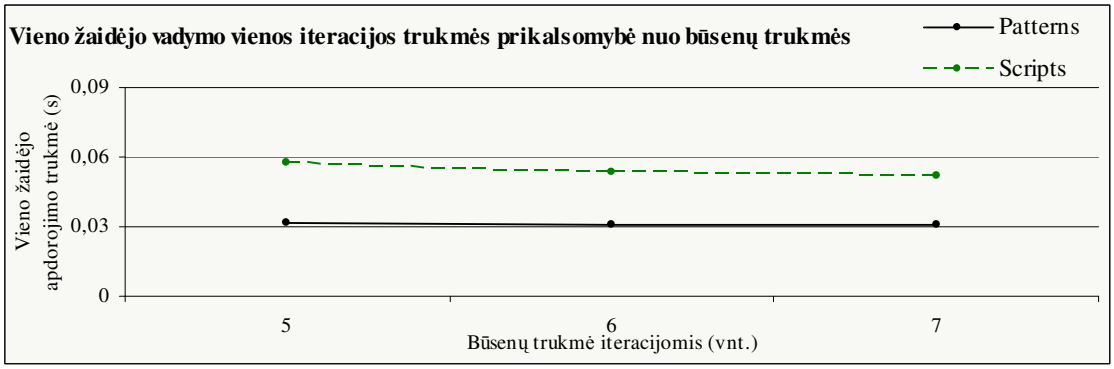
8 pav. Žaidėjo iteracijos trukmės priklausomybė nuo žaidėjų skaičiaus

Paveikslėlyje *9 pav.* vaizduojamas abiejų programų veikimo trukmių palyginimas. Grafiką galima paaiškinti taip: pirmasis tiesės taškas (1, 38%) reiškia, kad valdant vieną personažą, antroji programa veikė 38% (nuo pirmosios veikimo trukmės) ilgiau nei pirmoji.



9 pav. Abiejų programų veikimo greičiai

Paveikslėlyje *10 pav.* vaizduojamas bendras trijų testų rezultatas – personažo valdymo trukmės priklausomybė nuo būsenų trukmės. Matome, kad personažo valdymo laikas mažėja didėjant būsenų trukmei. Antrosios programos valdymo trukmė mažėja sparčiau.



10 pav.Personažo apdorojimo trukmės priklausomybė nuo būsenų trukmės