Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Rok Oblak

Simulacija množic heterogenih agentov v realnem času

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Matija Marolt

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V diplomski nalogi preučite področje simulacije obnašanja množic heterogenih agentov. Preučite in implementirajte pristope, ki pri simulaciji zagotavljajo zaznavanje okolja, izogibanje oviram in različna stanja obnašanja agentov ter hkrati omogočajo delovanje v realnem času. Izdelajte interaktivne simulacije, ki bodo tudi vizualno predstavile različne simulacijske scenarije.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Rok Oblak, z vpisno številko **63110310**, sem avtor diplomskega dela z naslovom:

Simulacija množic heterogenih agentov v realnem času

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Matije Marolta,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 26. avgusta 2014 Podpis avtorja:

Za podporo in vzpodbudo pri izdelavi diplomskega dela se zahvaljujem družini, prijateljem in mentorju Matiji Maroltu.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
VO	velocity obstacle	hitrostna ovira
RVO	reciprocal velocity obstacle	vzajemna hitrostna ovira
FSM	finite-state machine	končni avtomat
FPS	frames per second	sličic na sekundo
API	application programming interface	aplikacijski programski vmesnik

Povzetek

Simuliranje obnašanja agentov, ki predstavljajo osebke, je v zadnjih letih razširjeno na mnogo področjih. Del takih simulacij predstavljajo simulacije množic, kjer se naenkrat giblje in sodeluje večje število agentov. Pri tem je izziv najti ustrezno stopnjo kompleksnosti posameznega agenta, da bo mogoče hkrati simulirati večje število agentov in da bo njihovo obnašanje zadovoljivo. Dodaten problem predstavlja potreba po izvajanju simulacije v realnem času. V svojem delu sem napisal aplikacijo v pogonu Unity, ki se izvaja v realnem času in uporablja kombinacijo glavnih pristopov in tehnik za simulacijo množic heterogenih agentov, kot so modularna zgradba, zaznavanje okolja, izogibanje oviram, končni avtomat za modeliranje obnašanja, animator za prikaz animacij itd. V delu sem te pristope in tehnike podrobneje predstavil in komentiral rezultate aplikacije v različnih scenah, ki predstavljajo specifične situacije iz resničnega sveta.

Ključne besede: agent, množice, heterogenost, realno-časovno izvajanje, Unity, končni avtomat, modularnost.

Abstract

Behavioural simulation of agents representing humanoid characters has spread to many areas in recent years. A part of such simulations are crowd simulations, where large numbers of agents move and interact at the same time. Finding a suitable level of individual agent complexity so that large simulations are possible and suitable behaviour is reached, is challenging. In addition, executing such a simulation in real-time is problematic. In my work I developed a real-time application in Unity game engine which makes use of a number of main techniques and approaches for heterogeneous crowd simulations, such as modular architecture, environment sensing, obstacle avoidance, finite state machines for behaviour modeling, animator for animation visualisation etc. I thoroughly described and presented those approaches and techniques and commented on the results obtained in several different scenes which represent specific real-world situations.

Keywords: agent, crowds, heterogeneity, real-time execution, Unity, finite state machine, modularity.

Poglavje 1

Uvod

Danes obstaja veliko področij, kjer je potrebno simulirati raznorazne agente – virtualne osebke različnega izgleda in obnašanja, ki na take ali drugačne načine učinkujejo na okolje. Področja segajo od simuliranja primerov evakuacije velikega števila ljudi iz zgradb, vizualizacije premikanja velikega števila vojakov v bitkah (namesto uporabe velikega števila statistov za isti namen), simuliranje preostalih udeležencev v virtualnih okoljih v računalniških igrah, navigacija majhnih samovoznih robotov itd.

Tem robotom ali virtualnim osebkom v splošnem pravimo kar agenti. Agent je v simulaciji posamezna dinamična enota, šibko sklopljena z drugimi in zato zanjo velja, da v določeni meri ravna samodejno [5]. Po prostoru se giblje po lastni iniciativi in ne zgolj zaradi posledic fizikalnih sil nanj. Navadno je majhne velikosti, ima razne omejitve pri gibanju in mu je lastno posebno notranje stanje, ločeno od okolja, a odvisno od njega.

V domeni iz realnega sveta je agent npr. pešec, vozilo ali katerikoli drug objekt, kateremu pripada samostojni odločitveni proces [2]. Vsak agent pripada svoji fizični predstavitvi v okolju – svojemu telesu. To telo agentu omogoča, da zaznava okolico in nanjo deluje, tako s fizičnimi dejanji in komunikacijo kot tudi s premikanjem po njej, ter seveda omogoča njegovo vizualno predstavitev (v kolikor je ta prikazana na osnovi samega telesa).

Slika 1.1: Uporaba ne-realnočasovne simulacije množic v filmski industriji. Vir slike: New Line Cinema

1.1 Heterogenost

Heterogeni agenti so agenti, ki se v določeni meri razlikujejo med seboj. V [9] [14] denimo avtorji za heterogene agente smatrajo že samo sposobnost, da vsak agent poskuša priti do cilja, ki ni odvisen od ciljev drugih agentov in na poti do njega izvaja lastna dejanja. Drugod je definicija bolj stroga in definira razlikovanje v notranjih stanjih agentov in različno delovanje na več nivojih hierarhije; ta je v obliki motivacija – planiranje – določitev akcij – izvedba akcij. Taki agenti so sposobni opravljati več stvari in se obnašati ali prilagajati v več različnih situacijah. Modeli, ki se uporabljajo za izgradnjo takih agentov, so večinoma zasnovani tako, da je dodatne funkcionalnosti enostavno dodati in ustvariti več vrst agentov.

Heterogenost agentov najpogosteje vpliva na računsko zahtevnost simulacije določenega števila agentov predvsem zaradi večjega števila vejitev pri obravnavi več različnih situacij in dodatnega dela, ki ga agenti opravljajo, da delujejo bolj pametno in se ne premikajo zgolj po prostoru do določenega cilja. Dobri pristopi zaradi časovnih omejitev realno-časovnega izvajanja zato poskušajo število vejitev zmanjšati ter zmanjšati ali pohitriti izvajanje dodatnih računskih operacij. Učinkovit sistem heterogenih agentov tako omogoča širši nabor vseh mogočih stvari, ki so jih agenti sposobni izvajati. Hkrati taka zasnova omogoča, da agenti

v vsakem časovnem koraku izvajajo le majhen, konstantno velik del teh akcij, zato dodajanje novih sposobnosti ne bo bistveno vplivalo na celotno računsko zahtevnost. En takih modelov so končni avtomati, kjer je agentovo notranje stanje predstavljeno kot eno izmed stanj v takem avtomatu. Za druge namene se uporabljajo tudi celični avtomati, predvsem ko je cilj nenatančna predstavitev velikega števila agentov z dobro zastavljenim modelom obnašanja [7][6].

Simulacija množic poskuša modelirati gibanje velikega števila ljudi ali drugih samostojnih enot. Navadno je njen cilj raziskava vplivov, ki jih imata skupinsko gibanje in psihologija množic. Fokus takih raziskav je napovedovanje obnašanja množic in napovedovanje pojavov, ki se v njem dogajajo [10]. Ti pojavi so specifični za realne primere gibanja množic v odprtih in zaprtih prostorih in jih je mogoče ponoviti v simulacijah in tako ne le napovedovati, v katerih okoliščinah se pojavljajo, temveč tudi zasnovati in izboljšati te okoliščine (zasnova hodnikov v zgradbah, širina prehodov, število prehodov in morebitne vmesne ovire) za namene večje varnosti v primeru izrednih situacij [8][11][20]. Drugi pristopi dajo manj poudarka na gibanje množic in več poudarka na individualno obnašanje, uporabno v primerih, kjer ima človeški uporabnik (ali igralec) opravka z njimi [12].

1.2 Realno-časovne simulacije

Večina simulacij obnašanja množic agentov se ne izvaja v realnem času. Razlog je v tem, da izvajanje v realnem času postavi veliko omejitev največji računski zahtevnosti vsakega koraka simulacije, ki posledično vpliva na kakovost celotne simulacije. Če simulacija v navidezni resničnosti traja nekaj minut, imamo za izvedbo vseh časovnih korakov v primeru izvajanja v realnem času prav toliko časa (v primeru, da tudi čas v simulaciji teče z enako hitrostjo), sicer pa so nam na voljo ure ali dnevi. Izvajanje v realnem času v splošnem pomeni, da je simulacija vsako sekundo zmožna obdelati vsaj 15-20 časovnih korakov, kar človeku da občutek tekočega izvajanja. Bolj stroga definicija postavlja omejitve na največji čas izvajanja posameznega časovnega koraka, ki tako znaša le 50-66 ms, poleg tega pa mora biti čim bolj konstanten. Realno-časovno izvajanje pa ima pomembne prednosti, ki jih ni mogoče nadomestiti. Najprej tak način izvajanja omogoča neposredno vplivanje na simulacijo med samim izvajanjem; tako lahko med izvajanjem npr.

agentom spreminjamo cilje, jih dodajamo in odstranjujemo. To omogoča uporabo v aplikacijah kot so računalniške igre, poleg tega pa ponuja tudi hiter, sproten pregled nad simulacijo in ne šele po tem, ko je izračun končan. Tako izvajanje je zaradi krajše dolžine mogoče tudi večkrat ponoviti kot ne-realno-časovne simulacije.

Da implementacija simulacije množic teče v realnem času, mora biti za to prilagojena že od začetka, kar je seveda slabost, saj se simulacijam, katerih časovni koraki niso časovno omejeni, ni potrebno ubadati s pristopi, prilagojeni čim hitrejšemu izvajanju in se zato lahko osredotočijo na samo kvaliteto simulacije. Največji vpliv na računsko zahtevnost med posameznimi nalogami enostavne simulacija množic ima agentovo izmikanje dinamičnim oviram oz. povprečno število dinamičnih ovir, katerim se mora vsak agent izmikati, zato so bile razvite razne metode, ki ponujajo učinkovito in hitro rešitev problema gibanja brez trkov. V zadnjih letih pa je mogoče simulacije močno pohitriti z razdelitvijo računskega dela na različne niti izvajanja, s čimer je moč izkoristiti večprocesorske naprave in zadnja leta tudi grafične procesne enote [26]. Večnitnemu izvajanju je prilagojenih tudi večina metod za izmikanje oviram, ki omogočajo obravnavo vsakega agenta in njegovega dela pri izmikanju oviram neodvisno in sočasno z drugimi, kar pomeni, da lahko obdelavo agentov porazdelimo na različne niti izvajanja.

Kljub temu omejitve, ki jih postavi zahteva po realno-časovnem izvajanju, neposredno vplivajo na ključne lastnosti simulacije. Predvsem vplivajo na število agentov, ki lahko hkrati delujejo v navideznem svetu, kvaliteto njihove navigacije po svetu, kvaliteto njihovega notranjega modela oz. kvaliteto obnašanja ter sam izgled simulacije, v primeru da jo hočemo vizualno predstaviti, še posebno v če je predstavljena treh dimenzijah. Pogosto se sicer uporablja tudi dvodimenzionalna predstavitev, kjer so agenti predstavljeni kot krožci, kar praviloma predstavlja skoraj zanemarljivo malo računskega dela v primerjavi z izvajanjem drugih stvari.

1.3 Vizualizacija in zahtevnost

Če je vizualizacija trodimenzionalna in je vsak agent predstavljen s kompleksnim modelom (in ne, denimo, zgolj s kroglo ali kvadrom), je izrisovanje množic agentov še posebno velik problem pri simulacijah množic v realnem času. V primeru nerealno časovnega izvajanja se navadno izrisovanje razdeli, tako da se najprej izvede

simulacijo in ustvari izhodno datoteko, s katero potem izrisovalni pogon naknadno izriše slike. V realnem času pa dodatno izrisovanje pomeni zasedenost grafične procesne enote, ki bi se sicer lahko uporabljala tudi za namene simulacije množic, pa tudi centralne procesne enote, ki mora grafični enoti dostavljati podatke za izrisovanje – problematično je predvsem slednje, saj je dostavljanje ukazov za izrisovanje v primerih z večjim številom agentov zelo računsko zahtevno in lahko zato močno omejuje sposobnosti simulacije.

Ena izmed razdelitev pristopov simulacij množic je v tem, kako se obravnavajo posamezniki oz. agenti. *Makroskopski* model obravnava obnašanje celotne množice kot velik interaktiven sistem, brez podrobnega upravljanja s posameznimi agenti, temveč z uporabo statističnih in verjetnostnih tehnik. Po drugi strani *mikroskopski* model simulira množice na bolj intuitiven način – na ravni posameznih agentov, kar omogoča večji nadzor nad obnašanjem, večjo raznolikost in kompleksnost agentov in posledično bolj naravno gibanje v določenih situacijah, je pa zato tudi bolj računsko zahteven. [18].

1.4 Uporaba

Simulacija množic ima mnogo primerov uporabe: uporablja se za načrtovanje urbanih projektov kot so velike zgradbe, cone za pešce v mestih, prometna infrastruktura (agenti lahko predstavljajo tudi avtomobile in druga vozila), za simuliranje evakuacij, uporabo v zabavni industriji (filmi, promocijski videi ali reklame, videoigre), za potrebe vizualizacije vseh vrst itd. S heterogenim modelom agentov je mogoče oponašati velik spekter teh primerov uporabe in ustvariti agente, ki so sposobni hkrati delovati v več različnih situacijah.

Simulacija množic je še posebno pomembna za uporabo pri simuliranju evakuacije velikega števila ljudi iz raznih stavb in prevoznih sredstev (predvsem letal). Ustrezna simulacija v virtualnem okolju lahko v veliki meri nadomesti resnične evakuacijske vaje, saj pri tem ponuja vrsto prednosti, med njimi skrajšan čas izvajanja, cenejšo izvedbo ter odsotnost tveganja za poškodbe ljudi in objektov. Pomembno je, da simulacija pravilno oponaša različne tipe obnašanja posameznikov ter da ustrezno zajame individualizem posameznikov na dovolj veliki ločljivosti. Človeško obnašanje je namreč zelo zapleteno predvsem zaradi nedoločenosti pri

odločanju; pri tem se precej uporablja ti. mehka logika. Navadno se za uporabo simulacije množic za namene simuliranja evakuacij uporablja zelo preprosta 2-dimenzionalna predstavitev virtualnega prostora. Cilj agentov pri evakuaciji je najprej uiti neposredni grožnji, nato pa kar se da hitro doseči izhod ali drugo lokacijo, ki se smatra kot varna. Pri tem je glavni problem v iskanju optimalne poti do tega izhoda, saj je pot odvisna od množice drugih agentov ter tudi od tega, ali agent sploh ve, kje se izhod nahaja. Ko je ta problem rešen, pa je velik problem tudi reševanje zgoščin agentov, ki se pojavljajo v ozkih hodnikih in prehodih, saj se agenti med seboj prerivajo in pogosto neracionalno silijo v enake smeri in tako manjšajo največji pretok agentov iz prostora na varno.

1.5 Cilj

Cilj diplomske naloge je raziskati področje simulacije množic, opisati različne pristope in pogosto uporabljene tehnike.

V praktičnem delu je cilj simulirati čim večje število agentov v realnem času. Ti agenti naj bi bili heterogeni, torej naj bi se razlikovali tako v notranjem stanju kot po obnašanju, saj naj bi v okolju delovali samodejno in neodvisno od ostalih, ki pa bi jih zaznavali in z njimi komunicirali. Agenti bi pri predstavitvi uporabljali animacije.

Gibanje agentov naj bi bilo realistično tako po izgledu animacij kot po dejanskem premikanju. Agenti naj bi se tudi izmikali drug drugemu, s čimer bi bilo mogoče ponoviti določene pojave, ki se dogajajo pri gibanju množic. Arhitektura agenta naj bi zadostovala izvajanju različnih dejavnosti in bi omogočala enostavno dodajanje novih dejavnosti.

Na koncu naj bi bilo v nekaj različnih situacijah predstavljeno različno obnašanje agentov.

1.6 Struktura

Strukturno nalogo sestavlja pet poglavij. Prvo poglavje predstavlja uvod v področje in opiše nekaj usmeritev in zahtev, ključnih za to diplomsko nalogo.

Drugo poglavje predstavlja razvijalsko okolje in pogon Unity ter opisuje, kako je praktični del v tem okolju implementiran.

V tretjem poglavju je predstavljen problem navigacije agentov in opis implementiranih algoritmov, ki so ta problem reševali.

V četrtem poglavju je predstavljen model agenta, njegove komponente stanja njegovega obnašanja.

V petem poglavju je predstavljeno izvajanje aplikacije ter realizirane in možne pohitritve.

V šestem poglavju so predstavljene različne scene in komentirani rezultati praktične implementacije.

V sedmem in zadnjem poglavju se nahaja zaključek in sklep naloge, poleg tega so tu omenjene opažene težave ter je predlaganih nekaj možnih izboljšav in nadaljevanj dela.

Poglavje 2

Unity

Unity je igralni pogon in razvijalsko okolje, namenjeno izdelavi in poganjanju tridimenzionalnih aplikacij in iger na različnih platformah. Kot vsi sodobni igralni pogoni ima podprte vse komponente, ki so pri izdelavi in poganjanju takih aplikacij potrebni: izdelava in prevajanje skript in njihovo pripenjanje posameznim objektom, poganjanje APIjev za izrisovanje, podpora predvajanju prostorskega zvoka, pravilno izvajanje vse kode, ki predstavlja logiko aplikacije skozi posamezne časovne korake, sporočilne sisteme za komunikacijo med objekti, razne fizikalne knjižnice, vgrajene hitre pogosto uporabljane algoritme, napredno podporo animiranju objektov itd. Nudi tudi dobro podporo uvozu modelov, datotek različnih tipov ter uvozu dodatnih vsebin, ki so plačljive ali brezplačne in so na voljo v namenski spletni trgovini. Pogon oz. razvijalsko okolje je v osnovni različici na voljo brezplačno, njegove prednosti pa so poleg delovanja na več platformah predvsem v zmogljivosti ter intuitivnem uporabniškem vmesniku.

2.1 Igralni objekti

Unity temelji na ti. igralnih objektih, angl. *game objects*, ki so sestavljeni iz več komponent. Edina obvezna komponenta je komponenta *Transform*, ki vsebuje objektovo lokacijo, rotacijo in skaliranje. Vse ostale komponente so opcijske, zato so lahko taki objekti tudi »prazni« in ni nujno, da služijo kakšnemu namenu. Na objekte se pripenjajo skripte, pri čemer ima seveda lahko več objektov enako skripto, ki pa med izvajanjem navadno teče v več instancah. Skripte so

Slika 2.1: Razvojno okolje Unity

programi, napisani v enem izmed treh možnih programskih jezikih: C#, UnityScript in Boo. V primeru C# so ti programi kar razredi. Skriptiranje temelji na odprtokodni implementaciji .NET Framework, imenovani Mono. Skripte med izvajanjem kličejo uporabniško napisane funkcije in določene funkcije same definirajo (te med izvajanjem sproti kliče sam pogon). Med njimi so recimo *Start()*, *Update()* in *LateUpdate()*, ki se kličejo na začetku programa, v vsakem časovnem koraku in na koncu časovnih korakov, z njimi pa lahko definiramo logiko aplikacije tako, da določimo obnašanje objektov skozi posamezne časovne korake.

Med drugimi pomembnimi komponentami so recimo trkalna telesa ali trkalniki (angl. *collider*), ki se uporabljajo za detekcijo trkov med objekti, izvori zvoka, razni animatorji za namene animiranja objektov oz. njihovih delov, pa seveda tudi izrisovalniki (angl. *renderer*), ki skrbijo za grafični izris objektov, če jim pripada mreža trikotnikov.

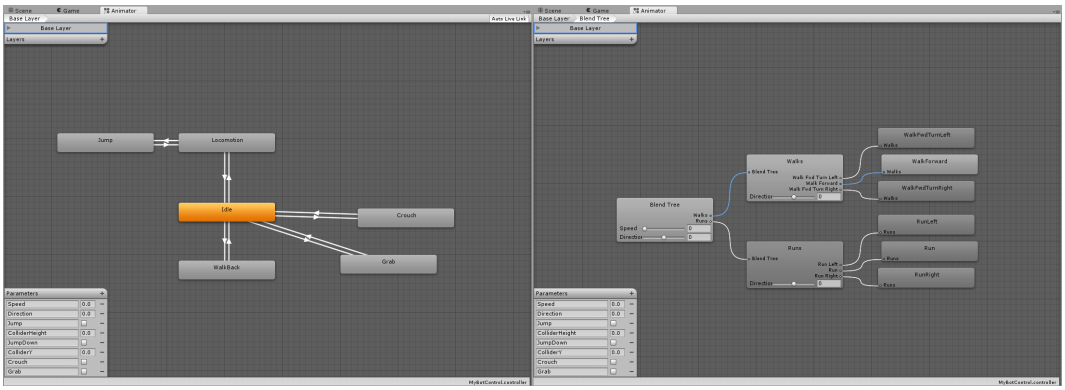
2.2 Pogon

Unity sem si za razvoj lastne aplikacije izbral zato, ker ima dobro izdelane vse podporne komponente za izdelavo simulacije v tridimenzionalnem prostoru. Dobro je tudi to, da so te komponente združene v enem samem pogonu, da npr. ni potrebno težavno povezovanje komponente za izvajanje same logike in komponente za grafični izris. Prav tako Unity ponuja razne knjižnice za uporabo v tridimenzionalnem prostoru, denimo knjižnico za fiziko, s katero je moč delati hitre poizvedbe po tem, ali se npr. kak objekt prekriva z drugimi, ali kaka daljica seka določene trkalnike itd.

Slabost samega pogona je v tem, da trenutno še ni podprto večnitno izvajanje posameznih skript. V verziji 4.5 namreč izvajanje skript še ni *thread-safe*, zato ob hkratnem izvajanju več skript ni zagotovljeno, da ne bo prihajalo do različnega zaporedja branja ali pisanja v določene spremenljivke, kar seveda lahko pripelje do napačnega izvajanja programa. Pomanjkljivost nameravajo razvijalci odpraviti z verzijo 5.0, ki pa za časa pisanja še ni na voljo. Kljub temu je pogon sposoben deloma izkoriščati več-jedrne procesorje tako, da na več niti razporedi druge naloge, recimo izvajanje ukazov grafičnega APIja, kar v končni fazi pomeni tudi to, da je jedro, na katerem se izvaja programska logika, manj obremenjeno.

2.3 Sporočanje

Različni objekti lahko do zunanjih spremenljivk in komponent – spremenljivk v skriptah in komponentah drugih objektov – dostopajo na različne načine. Eden izmed njih je preprosto branje (in pisanje) javnih spremenljivk, kar se lahko opravlja tudi preko javnih funkcij, poleg tega pa pogon Unity vključuje sporočilni sistem, s katerim lahko nek objekt pošilja sporočila enemu ali več (*broadcast*) drugim objektom, pri čemer pa smo omejeni na hierarhijo sestavljenih objektov, saj lahko tako komuniciranje poteka samo znotraj enega objekta, ki lahko vsebuje več drugih. Tako lahko starš prestreza sporočila otrok in jih posreduje naprej določenim drugim otrokom. V primeru neposrednega dostopanja do tujih objektov pa je na njih potrebno pridobiti referenco. Če je takih objektov veliko, je to lahko težavno, zato ima Unity vgrajene knjižnice za fiziko, ki izkoriščajo hitro iskanje objektov po iskalnem prostoru pod pogojem, da imajo objekti nameščene trkalnike. Tako lahko



Slika 2.2: Stanja animatorja in prehodi med njimi (levo) ter mešalno drevo stanja premikanja (desno)

denimo hitro preverimo, kateri objekti se nahajajo oz. sekajo določeno daljico, so znotraj določene krogle itd. V nasprotnem primeru moramo sami beležiti reference na objekte, ki jih potrebujemo, denimo z drevesi ali mrežami (še posebno, ko je igralna površina primarno v dveh dimenzijah, kot je tudi v mojem primeru).

2.4 Animiranje

Unity vsebuje lastno tehnologijo za podporo animaciji objektov, imenovano *Mecanim*. Animacija posameznega objekta je v tem načinu sestavljena iz končnega avtomata, v katerem vsako stanje predstavlja svojo animacijo, ki je lahko enostavna ali pa sestavljena iz več osnovnih animacij. Mecanim omogoča intuitivno dodajanje stanj v takem drevesu in definiranje prehodov med njimi. Animacije večinoma temeljijo na *okostjih*, ki so lahko za posamezno vrsto animiranih objektov standardizirani. Za humanoidne osebke, kot so ljudje in ljudem podobna fantazijska bitja, so tako definirana okostja, ki jih lahko apliciramo na poljuben ustrezen model in tako omogočimo njegovo animacijo. Animacija se izvaja nad enim ali več sklepi okostja v drevesni strukturi z inverzno kinematiko, mogoče pa je definirati omejitve pri gibanju posameznih sklepov.

Če je animacija stanja sestavljena iz več osnovnih animacij, je predstavljena s ti. mešalnim drevesom (angl. *blend tree*), iz katerega se razveji več osnovnih

animacij, v vozliščih drevesa pa lahko več osnovnih animacij kombiniramo, da ustvarimo kombinacijo več animacij, pri čemer *Mecanim* poskrbi za ustrezno zlitje iz ene osnovne animacije v drugo, tako da so vizualno prehodi čim gladkejši. Tako lahko npr. izvedemo animacijo teka naprej, ki je sestavljena iz osnovnih animacij teka naravnost, levo in desno. Mešalno drevo bo poskrbelo za gladek prehod med temi tremi animacijami glede na vrednost spremenljivke, ki določa vpliv posamezne animacije. Dodajati je moč nove animacije, če je okostje animiranega lika ustrezno, nato pa prehodom v in iz stanja nove animacije določimo zahteve in sprožilce ter še vizualno nastavimo hitrost animacije ter trajanje prehoda, da so mogoči gladki prehodi tudi med dvema sicer nezdružljivima animacijskima stanjema.

V skriptah potem nastavljamo ustrezne spremenljivke, ki najprej določijo stanje v avtomatu animacijskih stanj, potem pa z drugimi spremenljivkami določamo vpliv posameznih vej v mešalnem drevesu (če to obstaja), poleg vseh ostalih vrednosti, specifičnih za posamezno stanje, ter splošnih vrednosti animatorja, kot je npr. njegova hitrost.

Animator ima lahko neposreden vpliv na gibanje objekta, na katerem je nameščen. V tem primeru ustvarjalcu ni potrebno skrbeti za natančno nastavljanje spremenljivk, ki povezujejo prikazano animacijo s samim gibanjem, kar omogoča realističen prikaz premikanja. Slaba stran tega je, da je potrebno v tem primeru gibanje objekta izvajati preko animatorja, kar vpelje velik nabor dodatnih omejitev. Zato se uporablja tudi možnost, ko animator nima vpliva na gibanje objekta in skrbi zgolj za animacijo modela. Tu je potrebno natančno nastavljanje spremenljivk, ki povezujejo hitrost, usmeritev, nagib in druge spremenljivke objekta s stanjem animatorja, kar je težavno, če je cilj zelo realističen prikaz gibanja.

2.5 Implementacija v pogonu Unity

Vsak agent v pogonu Unity ima nastavljeno določeno oznako (angl. *tag*) in plast (angl. *layer*). Vsi navadni objekti privzeto pripadajo oznaki *Untagged*, z drugimi oznakami pa določamo poseben pomen objektov, ki so z njimi označeni. Pomen uporabe oznak je v pridobivanju reference na objekte določenih oznak in definiranju logike okoli njih. Podobno uporabo imajo plasti, kjer privzeto vsi navadni objekti pripadajo plasti *Default*, dodajanje drugih plasti pa se uporablja predvsem

za namene selektivnega izrisovanja in osvetljevanja objektov določene plasti na določeni kameri, za namene uporabe s fizikalnimi funkcijami, kjer lahko določamo maske plasti, s katerimi določene plasti upoštevamo, druge pa ne. Poleg tega so plasti uporabne za namene gradnje navigacijskih mrež, kjer lahko objekt vsake plasti definira njeno ceno, ki jo bo navigacijska mreža upoštevala pri izgradnji. Pri iskanju poti se bodo zato agenti izogibali plastem z veliko ceno.

Agenti v implementaciji so igralni objekti, izvedeni kot ti. montažni objekti. Z uporabo montažnih objektov je mogoče enostavno ustvarjati njihove instance, kar je uporabno zaradi večjega števila hkrati obstoječih enakih objektov. Vsi agenti so hierarhični nasledniki praznega igralnega objekta, ki predstavlja objekt z glavno skripto, zaradi katerega je mogoča njihova medsebojna komunikacija. Glavna skripta skrbi za sinhronizacijo komunikacije, vzpostavitev okolja, procesiranje uporabnikovega vnosa, grafični uporabniški vmesnik, beleženje podatkov o agentih, pomembnih za lokalno navigacijo, in drugo. Vsi objekti so postavljeni v različne scene, sestavljene iz raznih tridimenzionalnih objektov, med katerimi so tudi objekti, ki predstavljajo nepremične ovire in za katere je določeno, da se zapišejo v navigacijsko mrežo. Spremljanje situacije v sceni poteka preko glavne kamere, ki jo je moč upravljati s tipkovnico.

V pogled je dodan dvodimenzionalni grafični vmesnik, na katerem so gumbi, s katerimi je moč nastavljati velikost na novo dodanih skupin agentov, barv izbranih agentov in izbiro različnih scen. Na njem je tudi prikaz določenih informacij o sceni oz. dogajanju v njej.

Agente je mogoče na sceno dodajati v različno velikih skupinah na mesto, določeno z miškinim kurzorjem. Z miško jih je mogoče označevati s principom označevalnega pravokotnika, označene agente pa je mogoče urejati. Nastavljati jim je mogoče ciljno točko, pri čemer avtomatsko vstopijo v stanje premikanja, z gumbi na grafičnem vmesniku pa je označenim skupinam mogoče spreminjati izgled (barvo). Z označevanjem, nastavljanjem ciljnih točk in nastavljanjem izgleda je zato mogoče preprosto nastaviti situacijo, kjer se dve skupini agentov srečata med sabo in opazovati, kako si agenti utirajo pot mimo drugih agentov. Izbrane agente je moč tudi izbrisati.

Pri tem dodaten problem predstavlja uporaba animacij, saj je potrebno natančno zadeti povezavo med fizičnim premikanjem agentov, ki je posledica algo-

ritma lokalne navigacije, ter upravljanjem s stanji animatorja. Problematično je upravljanje s hitrostjo animacij, saj prehod stanj animatorja iz mirujočega preko hoje v tek ni linearen v smislu hitrosti premikanja. Prav tako je težavno že samo ugotavljanje smeri premikanja, saj se predvsem v situacijah zgoščin agentov njihove smeri usmeritve zelo hitro spreminjajo (zaradi delovanja algoritma lokalne navigacije). Ta problem je rešen z uporabo nizkoprehodnega filtra, ki z ustrezno jakostjo gladi usmeritev pri prikazu animacije. Potrebno je tudi nastavljati količino nagiba med gibanjem, ki je odvisna od hitrosti spremembe smeri.

Poglavje 3

Navigacija

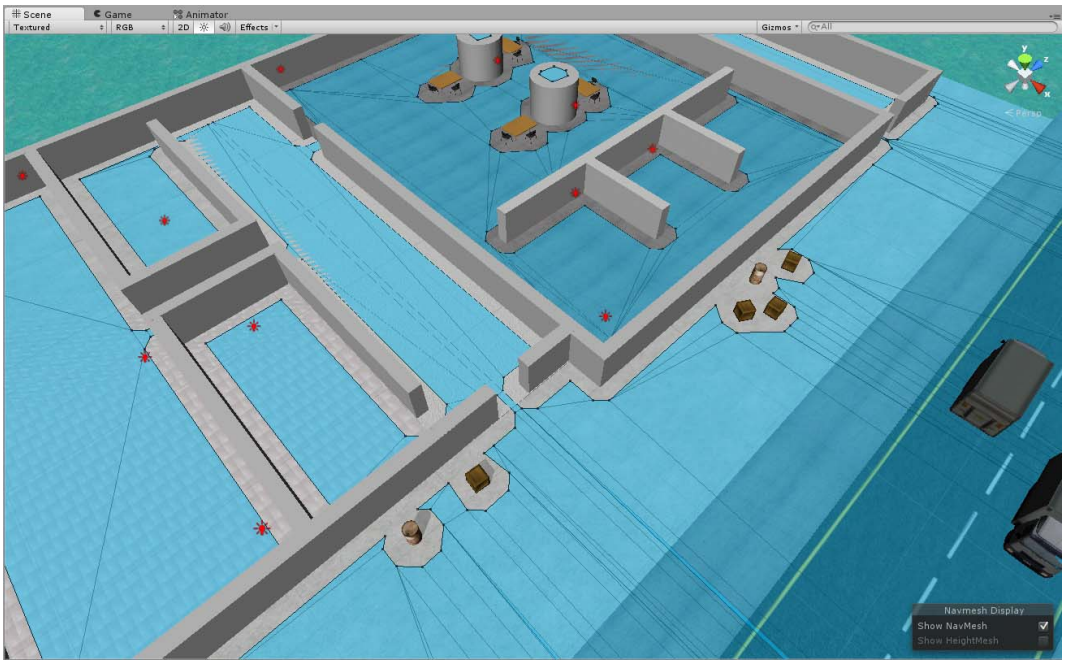
Navigacija v primeru virtualnega okolja in samostojnih agentov v njem pomeni sposobnost agentov, da najdejo ustrezno pot do želenega cilja. V primeru, ko je agent sam in v okolju ni fizičnih ovir, je navigacija trivialna, saj se agentu zgolj dodeli hitrost v smeri cilja. Vendar pa v splošnem temu ni tako – okolje je posejano s pregradami in preprekami, manjšimi nepremičnimi ovirami in drugimi dinamičnimi ovirami. Dinamične ovire so navadno drugi agenti ali pa bolj posplošene gibajoče ovire. Za prve velja, da se v splošnem poskušajo izmikati drugim oviram, za druge pa to večinoma ne velja.

V takih modelih se večinoma problem navigacije razdeli na dva ločena dela. Ta sta lokalna navigacija ali izmikanje oviram ter globalna navigacija ali iskanje poti. Med seboj se precej razlikujeta in omogočata hitro in učinkovito izvajanje celotne navigacije po kompleksnem prostoru.

3.1 Globalna navigacija

Sinonim za globalno navigacijo je iskanje poti po prostoru z nepremičnimi ovirami. Prav nepremičnost določenih ovir je razlog za ločeno obravnavo globalne navigacije. Ta poskrbi, da se ustvari pot v obliki povezanih točk v prostoru, ki bo agenta pripeljala do želenega cilja, brez da bi vmes prišlo do trka z nepremičnimi ovirami. Rezultat so lahko kompleksne poti, ki so rešitev problema tudi v primeru raznih labirintu podobnih koridorjev statičnih ovir.

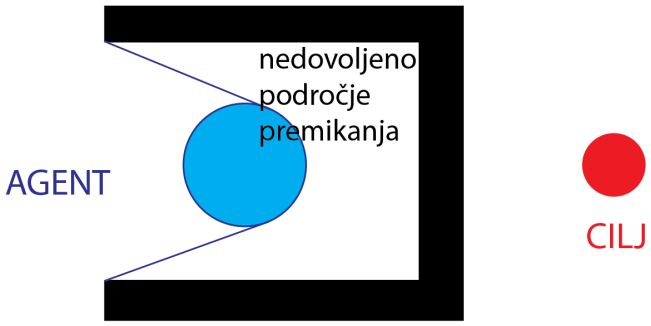
Običajno se iskanje poti v tem primeru izvaja z uporabo algoritmov iskanja poti



Slika 3.1: Primer navigacijske mreže v sceni

po drevesu, pri čemer so vozlišča drevesa točke na površini, kamor je premikanje dovoljeno, povezave med vozlišči pa pomenijo možnost premikanja med paroma vozlišč, pri čemer ima vsaka povezava svojo ceno, ki izraža težavnost premika (hitrost premika, tveganje pri premiku itd.). Najpogosteje uporabljan algoritem je algoritem A*.

Tu pridejo v poštev navigacijske mreže (angl. *navigation mesh*) [3], ki predstavljajo dvodimenzionalno površino, po kateri se lahko agenti gibljejo. Navigacijska mreža je shranjena v obliki grafa ali drevesa, ki se lahko potem uporablja za iskanje poti po tem grafu. Implementacije navigacijskih mrež zato pogosto vključujejo tudi algoritme preiskovanja poti po grafu oz. iskanja poti do cilja po področju, kjer se agenti lahko premikajo. Izgradnja navigacijske mreže je večinoma avtomatska, saj bi v nasprotnem primeru za to potrebovali zelo veliko dela že ob manj zapletenih situacijah. Pri izgradnji moramo definirati prehodne površine ter objekte na njih, ki jih želimo upoštevati kot nepremične ovire. Ker imajo premikajoči agenti navadno določene radije (svojo širino), moramo definirati tudi enoten radij, zaradi katerega bodo navigacijske mreže kot neprehodno označile vso površino, za radij



Slika 3.2: Agent, ki je pri uporabi zgolj lokalne navigacije nezmožen premikanja zaradi postavitve ovire

ali manj oddaljeno od vsake točke vsake statične ovire. V splošnem je ta postopek enak morfološki operaciji dilacije.

3.2 Lokalna navigacija

Algoritmi za iskanje poti v drevesu povezanih vozlišč bi sicer delovali za celotno navigacijo, tako da ločena obravnava lokalne navigacije ni nujno potrebna. Vendar pa bi moral algoritem za iskanje poti zaradi premikajočih se ovir vedno znova izračunavati novo rešitev, saj je določena rešitev veljavna samo pri določeni postavitvi ovir. Ker se določene ovire premikajo, se torej postavitev spreminja in pot, ki jo je prej našel algoritem za iskanje po drevesu, morda ni več veljavna. Ob velikem številu dinamičnih ovir bi to pomenilo veliko računsko obremenitev, zato se v takih primerih uporablja kombinacija ločene lokalne in globalne navigacije, v določenih primerih pa zgolj lokalna navigacija (če so statične ovire dovolj majhne in jih je dovolj malo).

V kombinaciji z globalno se lokalna navigacija uporablja med gibanjem po posameznih odsekih med vozlišči. Pri tem velja, da premikanje v odseku med dvema vozliščema poteka zgolj v eni smeri (agent v odseku smeri ne spreminja), zaradi globalne navigacije pa lahko pričakujemo, da pri tem ne bo prišlo do trkov s statičnimi ovirami, ki jih je globalna navigacija vzela v poštev med izračunom

poti (povezanih odsekov med vozlišči).

Lokalna navigacija poskrbi, da med prehodom med dvema vozliščema ne prihaja do trkov z dinamičnimi ovirami in s statičnimi ovirami, ki jih globalna navigacija med izračunom poti ni upoštevala. V splošnem zaradi raznih omejitev poskuša vsaj zmanjšati število trkov in ustvariti bolj naravno gibanje z vtisom, da se agenti zavedajo prisotnosti drug drugega. Manjše statične ovire namreč zgolj povečujejo zahtevnost izračuna poti, zato je pogosto bolje, da se jih obravnava v okviru lokalne navigacije. Poleg tega lahko dopustimo tudi, da se take ovire lahko premikajo, vendar ne na lastno iniciativo; denimo v primeru, ko jih drugi agenti potiskajo ali ko na njih delujejo druge fizikalne sile. Če bi take ovire hoteli zajeti v globalni navigaciji z iskanjem poti po drevesu, bi morali z vsakim premikom ovire na novo sestaviti drevo (in ne samo poiskati nove rešitve poti do cilja), kar je zahtevna operacija, ki pa se v določenih primerih še vedno uporablja - posebno v situacijah, kjer se samo igralno okolje sčasoma spreminja in se pojavljajo in izginjajo večje ovire, s tem pa tudi področje, po katerem je agentom dovoljeno premikanje.

Lokalni navigaciji pogosto pravimo kar izogibanje trkom, saj deluje tako, da zagotovi oz. poskuša zagotoviti premik v enem časovnem koraku, pri katerem ne bo prišlo do trka, v splošnem pa nima plana vnaprej, za časovne korake po trenutnem. Zato je lokalna navigacija sama po sebi v večini izvedb nezmožna najti pot preko več večjih ovir v določenih postavitvah. Z algoritmi lokalne navigacije je praviloma, denimo, nemogoče razrešiti situacije, kjer so na poti do cilja ovire v treh smereh (glej sliko 3.2) in bi se moral agent za dosego cilja nekaj časa premikati v nasprotni smeri.

3.2.1 Trki

Trk ali kolizija je dogodek, ko prideta dve površini (navadno trikotnika) v stik oz. prekrivanje. V splošnem to velja za celotno površino fizičnih objektov v virtualnem svetu, vendar pa se zaradi velike računske zahtevnosti ugotavljanja prekrivanja v takem primeru večinoma uporabljajo bolj enostavne predstavitve modela, ki se uporablja za detekcijo trkov, ti. *collider* ali trkalno telo. Ta telesa so večinoma enostavni objekti kot so krogla, kapsula, kvader ali kocka oziroma poljubna kombinacija več teh teles v prostoru.

V primeru gibanja množic so trki predvsem problematični v primeru, ko se dve telesi prekrivata toliko, da je to zaznavno – primer, ko se zgolj rahlo dotikata, ni problematičen, saj je to normalen pojav v gibanju gosto natlačenih objektov, npr. ljudi. Zaznavanje trkov tukaj pomaga tako, da do prekrivanja ne more priti, saj gibanje kateregakoli telesa v področje drugega telesa ni dovoljeno; za to lahko poskrbi sam pogon ali pa dodatni algoritmi. V mojem primeru agenti uporabljajo trkalna telesa v obliki kapsul, pri katerih je detekcija zelo preprosta in računsko nezahtevna, tako da njihova uporaba nima večjega vpliva na hitrost izvajanja. Uporaba algoritmov za lokalno izmikanje trkov v večinski meri odpravi posamezne trke oz. prekrivanja trkalnih teles, se pa ti občasno še vedno dogajajo, še posebej v gosto natlačenih situacijah.

Razlogov za to je več:

1. Omejitve pri gibanju, ki določajo, kakšna je lahko nova, popravljena hitrost v smislu največjega pospeška, velikosti, kota odmika itd.

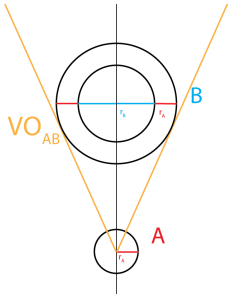
2. Omejene količine podatkov oz. omejenega števila sosednjih agentov, o katerih ima agent podatke (predvsem iz razloga večje zmogljivosti), tudi zaradi tega, ker agenti predvidevajo vzajemno ravnanje vseh ostalih agentov

3. Zmogljivost - ni potrebno, da vsak agent izvaja izmikanje oviram vsak časovni korak, kar pomeni nepopolno delovanje (a v praksi še vedno zadovoljivo)

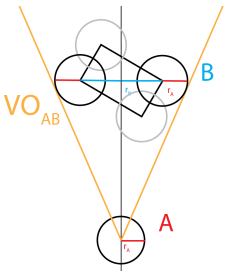
Zaradi omejitev tako v določenih primerih ni možno izbrati vektorja hitrosti, za katerega bi bilo zagotovljeno, da ne bo pripeljal do trka. Zaradi omejenih količin podatkov sosednjih agentov pa pride do tega, da pri določenih parih ni več vzajemnega izmikanja, kar je predpostavka določenih algoritmov izmikanja oviram.

3.2.2 Hitrostne ovire

Med načini za izvedbo izogibanja trkom so precej uporabljene ti. hitrostne ovire, obstajajo pa tudi drugi pristopi, specifični za določene situacije. Ena takih rešitev se uporablja pri gosto stisnjenih množicah, ki jih obravnava kot tok enotnega sestavljenega sistema. Ker so osnovni gradniki tega sistema agenti, ki zavzemajo najmanj neko površino, je narava takega toka takšna, da se lahko njegova gostota prosto širi in krči, vendar se ne more skrčiti pod dovoljeno mejo – podoben primer



Slika 3.3: Hitrostna ovira pri mirovanju

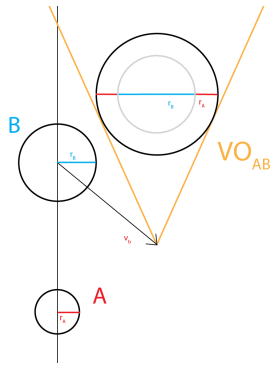


Slika 3.4: Hitrostna ovira pravokotne ovire

v naravi je denimo gibanje zrn peska [9]. Podobni pristopi agente tretirajo kot fizikalne delce, ki poskušajo iz bolj zgoščenih območij uiti v manj zgoščena [1]. Drugi pristopi temeljijo na razdeljevanju agentov v skupine in različni obravnavi vsake skupine, kar ob dobrem načinu grupiranja omogoča večje število agentov oz. daljši radij zaznavanja drugih agentov [17] [4] [19].

Pri uporabi hitrostnih ovir agent potrebuje preko senzorjev pridobiti zgolj informacije o lokaciji premikajoče se ovire, njeni obliki ter njeni hitrosti. To je dobro zato, ker ni potrebna nikakršna posebna komunikacija med agenti. Je pa seveda potrebno upoštevati, da pri meritvi teh količin pride do nenatančnosti, še posebno pri uporabi v resničnosti (v primeru simulacije v virtualnem svetu se nenatančnost pogosto umetno dodaja), ki jo je potrebno opredeliti in izmeriti ter ustrezno izvajati izmikanje oviram tudi z upoštevanjem le-te.

Hitrostna ovira (angl. *velocity obstacle, VO*) nastane ob obravnavi ovir med lokalnim izogibanjem oviram. Eni oviri praviloma pripada ena hitrostna ovira pri vsakemu ločenemu agentu, ki uporablja izogibanje trkom, razen v primeru da je



Slika 3.5: Hitrostna ovira med gibanjem

ovira preveč kompleksna in je zanjo potrebna hkratna obravnava več hitrostnih ovir. Hitrostna ovira je področje na ravnini, v katerem se vektor hitrosti agenta, ki to hitrostno oviro upošteva, ne sme nahajati, če želimo, da vsaj v naslednjem časovnem koraku ne bo prišlo do trka med agentom in oviro. Po matematični definiciji je hitrostna ovira $VO^{\tau}_{A|B}$ (hitrostna ovira agenta A, povzročena s strani agenta B v časovnem oknu τ) nabor vseh relativnih hitrosti v agenta A glede na agenta B, ki privedejo do trka dveh agentov v nekem času t, manjšem od τ.

Zapis hitrostne ovire je potem: $VO^{\tau}_{A|B} = \{v|\exists t \in [0,\tau] :: tv \in D(p_B - p_A, r_A + r_B\}$, kjer je $D(p,r)$ disk s centrom v p in radijem r, v relativna hitrost, t poljuben čas med 0 in τ, p_A in p_B sta poziciji obeh agentov, r_A in r_B pa njihova radija.

Z uporabo hitrostnih ovir torej ugotovimo, kateri vektorji hitrosti ne privedejo do trkov v določenem časovnem koraku. Če hočemo to razširiti na celoten čas izvajanja, moramo v vsakem časovnem koraku hitrostne ovire posodabljati skladno z novim stanjem; predvsem tu pride v poštev oblika in velikost ovire, položaj agenta, ovire ter trenutne hitrosti agenta in ovire.

Pri tem je trenutna hitrost agenta hitrost, ki je bila določena na koncu prejšnjega časovnega koraka in s katero se agent giblje v trenutnem časovnem koraku – ob koncu trenutnega bo spet prejel novo, popravljeno hitrost (če je popravek potreben), s katero se bo gibal v naslednjem časovnem koraku. Vsakemu agentu pripada tudi želena hitrost, ki kaže v smeri naslednjega cilja (ki ga je določila globalna navigacija in na poti do katerega praviloma ni nepremičnih ovir). Velikost vektorja želene hitrosti je večinoma enaka najvišji hitrosti, s katero se agent lahko giblje.

Za najbolj naravno in optimalno gibanje je najbolje, da je na koncu koraka vektor popravljene hitrosti čim bližje vektorju želene hitrosti.

3.2.3 Uporaba

Najpreprostejša oblika hitrostne ovire je, ko obravnavana ovira stoji na mestu, torej ima ničeln vektor hitrosti. Hitrostna ovira v tem primeru je preprosto neskončno dolg enakokrak trikotnik z ogliščem v sredini agenta in neskončno dolgima krakoma, ki sta postavljena pod takim kotom glede na središčno premico med središčema ovire in agenta, da je razdalja med katerokoli točko na obeh krakih in središčem ovire vedno večja ali enaka vsoti radija agenta in radija ovire. Tako je zagotovljeno, da v primeru agentovega gibanja s hitrostjo, ki leži na krakih ali izven njih, do trka ne bo prišlo. Kot med obema krakoma ne more biti večji od 180°, saj bi v nasprotnem primeru veljalo, da se agent in ovira že prekrivata; takrat je razdalja med središčem agenta in središčem ovire manjša od vsote njunih radijev. Za test, ali točka leži znotraj trikotnika je zato dovolj, da preverimo, ali leži desno od levega kraka in levo od desnega kraka.

V splošnem ovire niso popolnoma okroglih oblik. V praksi se jih zato v glavnem aproksimira s krogi, v določenih primerih pa tudi z drugimi oblikami, recimo pravokotniki, ali s kombinacijo oblik. V tem primeru je težavnejše iskanje radija ovire, saj se širina spreminja v odvisnosti od usmeritve ovire glede na agenta. V primeru pravokotne ovire je potrebno za radij izbrati eno izmed štirih stranic ali eno izmed dveh diagonal ter središče pomakniti na sredino izbrane daljice. Če je aproksimacija ovire kompleksnejša kombinacija več oblik, je iskanje radija še zahtevnejše, zato se večinoma uporablja več posameznih hitrostnih ovir, ki se jih potem združi v eno samo. V tem primeru je potrebno pretehtati računsko zahtevnost iskanja radija ter združevanja več hitrostnih ovir v eno.

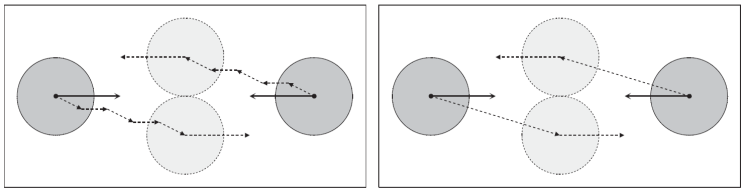
Ker se v splošnem ovire pogosto tudi gibljejo, pa taka predstavitev hitrostnih ovir seveda ni več veljavna, saj bi bilo potrebno v vsakem časovnem koraku ponovno ugotoviti novo predstavitev, kar pripelje do ukrivljenega gibanja agentov. Večino časa pa velja, da je hitrost ovire v določenem časovnem koraku enaka, kot je bila v prejšnjem časovnem koraku (če ne gre drugače, lahko to zagotovimo z umetnimi omejitvami, ki onemogočajo prehitro in prepogosto pospeševanje in zaviranje). Zato zadostuje, da hitrostno oviro zgolj premaknemo v smeri vektorja

hitrosti ovire. To pomeni premik izhodišča trikotnika ter vseh točk na obeh krakih, tako da kot med krakoma ostane enak. S tem zagotovimo, da vektor hitrosti, ki leži izven hitrostne ovire, tudi v tem primeru ne bo privedel do trka.

V taki obliki se hitrostne ovire uporabljajo že več kot stoletje za namen pomorske navigacije v bližini pristanišč (ko je manevrirnega prostora malo), za namen robotskega izmikanja oviram pa so bile pod različnimi imeni izumljene v devetdesetih letih, medtem ko so v prejšnjem desetletju in tudi v trenutnem aktualni predvsem algoritmi, ki uporabljajo izboljšane različice hitrostnih ovir. Sicer se hitrostne ovire v realnosti uporabljajo tudi v avtomobilih za avtomatsko opozarjanje voznika o bližajočem se vozilu, planiranje nalog brezpilotnih letal in še posebej za navigacijo robotov [13].

3.2.4 Težave

Pri uporabi z roboti in še posebej pri uporabi v virtualnih okoljih, torej s samostojnimi agenti, osnoven algoritem za izbiro hitrosti izven navadnih hitrostnih ovir dobro dela, pomanjkljivost pa se pojavi v primeru, da več agentov uporablja isti algoritem za izmikanje drugim – večina premikajočih se ovir se v tem zaveda drugih premikajočih se ovir in se tudi same poskušajo po najboljših močeh izmikati. To privede do tega, da v nekem trenutku (oz. časovnem koraku v primeru simulacije) oba agenta, katerih želene hitrosti bi sicer privedle do trka, popravita svoja vektorja hitrosti, ki zagotavljata, da do trka ne bo prišlo. V naslednjem časovnem koraku, ko bosta agenta popravljena vektorja hitrosti že prevzela, bosta njuni hitrosti izven obeh hitrostnih ovir, zato bosta svoja vektorja hitrosti popravila kar na vektor želene hitrosti. Ta bosta v naslednjem časovnem koraku znova znotraj hitrostnih ovir, zato bodo potrebni vnovični popravki, situacija pa se ponavlja vse dokler se agenta gibljeta drug proti drugem. Rezultat tega je tresoče gibanje, še posebno opazno v daljših časovnih korakih (slika 3.6). Optimalno bi seveda bilo, če bi agenta na začetku srečanja privzela svoja popravljena vektorja hitrosti in se jih držala med celotnim obsegom srečanja. To zagotavljajo algoritmi, ki uporabljajo ti. *vzajemne* hitrostne ovire [16].



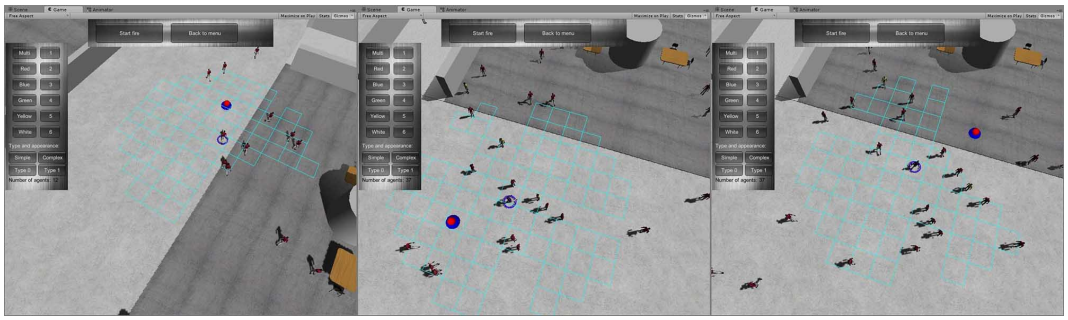
Slika 3.6: Prikaz tresočega gibanja pri uporabi VO (levo) in RVO (desno) [16]

3.3 ClearPath

Eden takih algoritmov je ClearPath [5], razvit leta 2009 na univerzi v Severni Karolini. Algoritem uporablja koncept vzajemnih hitrostnih ovir (angl. *reciprocal velocity obstacle, RVO*)[16], predstavljenih leta 2008, ki rešujejo problem tresočega gibanja. Predvideva vzajemno izmikanje vsakega para agentov oz. ovir, zato ta koncept velja samo pri ovirah, ki se po prostoru ne gibljejo pasivno (brez lastnega izmikanja drugim). Predpostavka je torej, da bo vsaka druga ovira tudi sama ravnala enako, zato agentu in drugemu agentu (oz. oviri) vsakemu pripada polovična odgovornost za izmikanje. Namesto, da bi vsak agent izbral hitrost, ki leži izven hitrostne ovire nekega drugega agenta, raje izbere povprečje med trenutno hitrostjo in neko hitrostjo, ki leži izven hitrostne ovire drugega agenta. Hitrostna ovira je zato iz središča agenta pomaknjena za vektor $\frac{v_a+v_b}{2}$.

Ovir je v splošnem več kot le ena, zato mora vsak agent sestaviti RVO trikotnike za vsako oviro. Zaradi razloga zmogljivosti vsak agent ignorira ovire, ki so preveč oddaljene ali niso vidne, tako da mora v vsakem časovnem koraku najprej ugotoviti seznam relevantnih ovir glede na lastno lokacijo. Ugotavljanje najbližjih vidnih agentov ni trivialno, če nočemo prevelike računske zahtevnosti, zato se navadno uporablja iskanje po drevesih, kjer se hitro ugotovijo bližnji sosedi, ali pa iskanje sosedov v dvodimenzionalni mreži, ki ima shranjene lokacije in hitrosti drugih ovir. Pri tem je potrebno poskrbeti, da bo drevo ali mreža ustrezno posodobljena ob koncu vsakega časovnega koraka in da bo branje posameznih ploščic ali listov potekalo šele, ko so vsi agenti svoje lokacije in hitrosti že poslali skripti, ki skrbi za posodabljanje.

Če imajo ovire svoje trkalnike, pa je za ta namen moč uporabiti tudi fizikalne



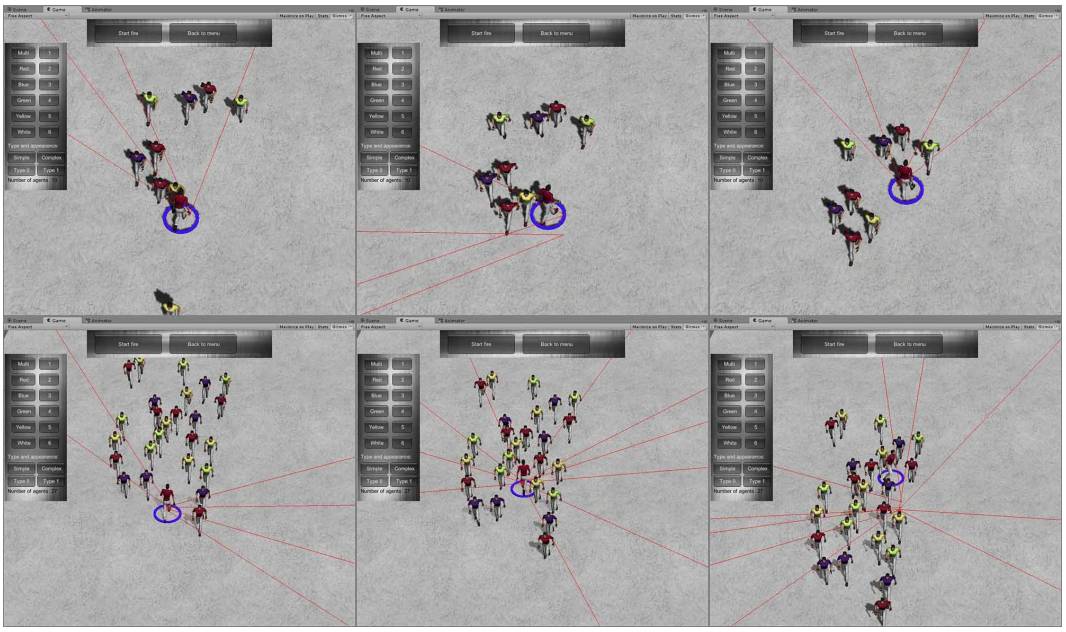
Slika 3.7: Prikaz zaznavanja sosedov v mreži

knjižnice pogona Unity. Tako se lahko, denimo, iz agentove lokacije z določeno funkcijo ustvari kroglo nekega radija, potem pa pogon poskrbi za hitro iskanje vseh trkalnikov znotraj te krogle. Preko teh trkalnikov dobimo reference na objekte ovir. Enako lahko s fizikalnimi funkcijami ugotovimo, ali so ovire vidne, tako da »pošljemo« kroglo v smeri iz agentove lokacije proti lokaciji ovire, katere vidnost želimo preveriti, potem pa nam pogon vrne vse trkalnike, ki se znajdejo na poti poslane krogle. Če je ovira vidna, seveda vmesnih trkalnikov na poti ne sme biti.

3.3.1 Implementacija

V svoji implementaciji sem izvedel dva načina iskanja najbližjih relevantnih sosedov. Prvi koristi fizikalno knjižnico in je uporaben v primeru, da imajo vse ovire lastne trkalnike (nad njimi namreč delujejo funkcije te knjižnice), deluje pa z že omenjenim pregledovanjem notranjosti krogle in pošiljanjem krogel iz agentove sredinske točke. Drugi način deluje tudi v primeru, da trkalnikov ni in koristi za ta namen ustvarjeno mrežo, v kateri so v ustreznih poljih shranjene pozicije in hitrosti vseh agentov. Mreža je shranjena v glavni skripti, agenti pa na koncu vsakega koraka (ko je mreža oz. podatki o vseh agentih v celoti posodobljena) do nje dostopajo z bralnimi dostopi samo do določenih polj ali ploščic.

Izbira ploščic je lahko preprosta, tako da se denimo vzame zgolj ploščice, ki obdajajo ploščico, v katerem se nahaja agent (sredinska ploščica), ali pa bolj kompleksna. Jemanje sosednjih osem ploščic je najbolj smiselno v zgoščinah. Ko zgoščin ni, pa lahko jemljemo tudi bolj oddaljene ploščice, a samo tiste, pred kate-

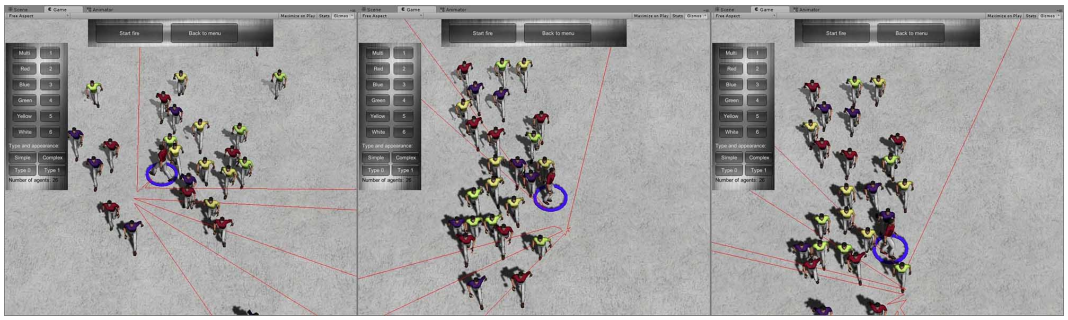


Slika 3.8: Prikaz RVO ovire za relevantne sosednje agente

rimi v smeri proti agentu ni drugih ploščic, kjer bi tudi bile ovire. To dosežemo, da najprej z Bresenhamovim algoritmom za risanje krogov [24] generiramo ploščice na določenem radiju od sredinske, nato pa z Bresenhamovim algoritmom za risanje črt [21] sledimo črti iz sredinske ploščice proti vsaki izmed generiranih ploščic v obliki krožnice. Ko dosežemo prvo, na kateri se nahaja vsaj ena ovira, ovire dodamo na seznam sosedov in prenehamo s sledenjem črti.

Ko so trikotniki RVO sestavljeni za vsako relevantno oviro, jih je potrebno združiti v en sam površinski lik, ki je navadno kompliciranih oblik in nesklenjen. Tej površini se pravi kombiniran RVO ali kombinirana vzajemna hitrostna ovira, predstavljena pa je z naborom robov, ki jo omejujejo. Vektor hitrosti agenta, ki v naslednjem časovnem koraku ne bo privedel do kolizije, leži izven te površine ali na njenih robovih.

Problem predstavitve kombiniranega RVO postane problem iskanja njegovih robov. Ti vedno ležijo na robovih RVO-jev posameznih ovir, je pa potrebno ugotoviti, kateri robovi posameznih RVO-jev so v celoti na robu kombiniranega RVO, kateri le delno in kateri so znotraj. Prvi korak pri tem je, da ugotovimo, katere



Slika 3.9: Prikaz kombinirane RVO ovire oz. področja

točke na posameznih krakih RVO trikotnikov se sekajo z kraki drugih RVO-jev. To je računsko zahteven postopek, saj ima zahtevnost $O(n * n)$ (kjer je n število relevantnih ovir ali sosedov). Vse točke presečišč je potrebno shraniti in v naslednjem koraku ugotoviti, katere ležijo znotraj in katere zunaj kombiniranega RVO. Za vsako točko je potrebno preveriti, ali leži znotraj kateregakoli RVO trikotnika tako, da pogledamo, ali leži deno od levega kraka in levo od desnega kraka. To je računsko zahteven postopek, saj ima zahtevnost $O(n * n * P)$, kjer je P povprečno število točk vsakega RVO, zato so v tem koraku smiselne razne pohitritve. Točke, ki ležijo znotraj omejene površine, se odstrani. Za končno določitev zunanjega roba površine kombiniranega RVO se uporabi postopek, ki iz nabora točk vsakega kraka izmenično jemlje odseke ali daljice. Če krak predstavljajo točke $ABCD$, potem bo daljica AB robna daljica kombiniranega RVO, BC bo znotraj, CD pa spet zunaj kombiniranega RVO. To velja za vse krake vseh RVO-jev, v kolikor so vse notranje (zakrite) točke že izbrisane.

Popravljen vektor hitrosti je enak želenemu vektorju hitrosti, če le-ta leži izven te površine. V nasprotnem primeru leži na enem izmed robov te površine, če vzamemo optimalno možnost, ko je razdalja med želenim vektorjem hitrosti in popravljenim vektorjem hitrosti najmanjša. Katerikoli točki znotraj te omejene površine bo vedno bližje neka točka na robu površine kot pa točka izven te površine. Zato s hevrističnim načinom iskanje popravljenega vektorja izvajamo kar robovih te površine.

V svojem delu sem napisal lastno implementacijo, podobno algoritmu ClearPath, ki za vsakega agenta v vsakem časovnem koraku izvede naslednjih šest

korakov:

0. Iskanje najbližjih sosedov
1. Izgradnja RVO-jev za vsakega izmed sosedov
2. Iskanje točk presečišč v kombiniranem RVO
3. Klasificiranje presečiščnih točk po tem, ali so v ali izven kombiniranega RVO
4. Sortiranje presečiščnih točk na vsakem kraku po urejenem vrstnem redu
5. Klasificiranje posameznih odsekov na krakih v zunanje in notranje in iskanje točke na zunanjih segmentih, ki je najbližja želenemu vektor hitrosti
6. Izvajanje dodatnega ravnanja v primeru, da prej določen popravljen vektor hitrosti ne zadostuje določenim omejitvam

Teh šest korakov je v osnovi enakih algoritmu ClearPath, vendar pa je izvedba posameznih precej drugačna. Moj algoritem je namreč namenjen preprostemu razhroščevanju, vizualizaciji posameznih korakov in podpori dodajanju novih tipov ovir (denimo nepremične ovire ali dinamične ovire, ki se ne odzivajo na okolico).

Algoritem ima računsko zahtevnost $O(A * N * N * C)$, kjer je A število vseh agentov, N povprečno število relevantnih sosednjih ovir, C pa ne-konstantni faktor, odvisen od povprečnega števila segmentov v kombiniranem RVO. Algoritem je zato močno občutljiv na število relevantnih sosednjih ovir, ki pripadajo vsakemu agentu.

Dobra plat tega je, da lahko z dobrim načinom iskanja teh ovir njihovo število dovolj zmanjšamo (navadno na manj kot 10 sosedov) tako v primerih velike zgostitve kot v primerih, ko so agenti (in ovire) razporejene redkeje po površini. Tudi v tem primeru lahko močno zmanjšamo število kolizij, saj, vsaj v teoriji, vedno pravočasno med relevantne ovire zajamemo tudi tisto, s katero bi lahko prišlo do trka. Če takrat tudi ta ovira ravna enako, torej tudi sama zazna agenta, bo ravnanje pravilno in predpostavka pri uporabi RVO-jev – vzajemno ravnanje obeh v paru – bo izpolnjena.

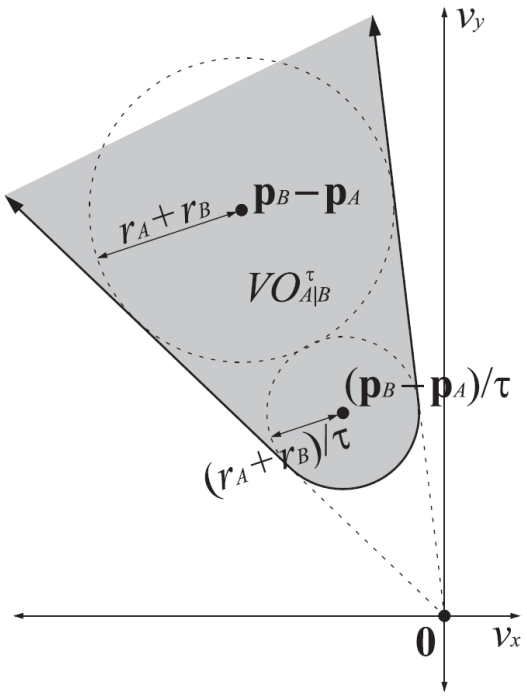
Dejanski algoritem ClearPath ima seveda vključenih mnogo dodatnih optimizacij, ki zmanjšajo odvisnost od števila ovir in omogočajo višje število simuliranih agentov pri določeni računski sposobnosti. Njegova računska zahtevnost za vsakega agenta je $O(A*N(N+M))$, kjer je M število vseh odsekov na kombiniranem

RVO. Ker je število N in M za vsakega agenta praviloma spremenljivo, je pri fiksnih računskih zmogljivostih največje število agentov, ki jih je mogoče simulirati v realnem času (torej ob določenem največjem času izvajanja posameznega časovnega koraka), odvisno prav od razporeditve agentov ter načina iskanja relevantnih sosednjih ovir.

3.4 ORCA

Nato sem uporabil še javno dostopno različico algoritma, imenovanega Optimal Reciprocal Collision Avoidance [15], ki tudi uporablja ovire RVO, in ga priredil za uporabo v svoji aplikaciji. Za uporabo različnih algoritmov izogibanja oviram je potrebno poskrbeti za ustrezno iskanje sosedov ali sosednjih ovir ter zagotoviti pravočasno posodabljanje vektorjev hitrosti agentov. ORCA zagotavlja gibanje več robotov ali agentov brez trkov. Naloga algoritma je, da vsakemu agentu neodvisno in vzporedno dodeli hitrost, za katero bo vsaj τ-časa zagotovljeno, da ne bo privedla do trka. Prisotna je tudi sekundarna zahteva, naj bodo vektorji hitrosti karseda blizu želenim.

Za hitro izvajanje algoritem uporablja linearno programiranje, da v majhnem številu operacij najde najustreznejši vektor hitrosti na dovoljenem območju. Najustreznejša hitrost je tako rezultat maksimizacije spremenljivke v funkcijah, ki jih določajo omejitve in zahteve gibanja brez trkov in brez prekomerno tresočega gibanja. Algoritem omogoča dobro izrabo večprocesorskega izvajanja, posebno pri večjem številu agentov, ter zelo hitro izvajanje pri manjšem povprečnem številu sosednjih ovir. Opaziti je, da pri uporabi tega algoritma v implementaciji večjo omejitev kot lokalna navigacija predstavlja preobsežna komunikacija med agenti ter prekomerna uporaba fizikalnih testov trkov pri senzorjih. To še posebej trži pri razmeroma nizkih radijih zaznavanja sosednjih agentov, medtem ko postane algoritem zelo zahteven v primeru zgoščin in večjih, zgoščinam neprirejenih radijih zaznavanja.



Slika 3.10: Hitrostna ovira pri ORCA z zaobljenim robom, odvisnim od parametra τ [15]

Poglavje 4

Model agenta

V viru [25] avtorji predstavijo splošni osnovni model agentov, primernih za uporabo v različno zapletenih in visoko spremenljivih situacijah. Okvir vsakega agenta zajema več modularnih delov, ki jih je mogoče dodajati, odstranjevati in spreminjati za potrebe različnih situacij. Simulacijo v greobem sestavljajo glavni kontrolni sistem, iz katerega uporabnik določa agente, okolje in razne spremenljivke v njih, sistem okolja, ki upravlja z virtualnim svetom (v katerem prebivajo in delujejo agenti), sistem obnašanja, lasten vsakemu agentu, sistem za motoriko, ki skrbi za premikanje agenta po okolju in morebitne animacije pri premikanju in izvajanju dejanj, ter izrisovalni sistem, ki skrbi za izris posameznega agenta. Problem tukaj predstavlja dobro uravnotežena izbira spremenljivosti ali heterogenosti agentov ter časovne kompleksnosti simulacije agentov. Heterogenost agentov omogoča, da so agenti sposobni delovati oz. se obnašati v različnih situacijah, ne le v eni sami, in da se znajo prilagajati spreminjajoči se situaciji. Prilagajanje pomeni spremenljivo notranje stanje in zmožnost predvidevanja sprememb, kar za vsakega agenta dodatno zviša potrebno računsko delo.

Podobno arhitekturo predstavlja [10], kjer je vsak agent z okolico povezan s senzorji in pogonom – senzorji dovajajo tok podatkov v agenta, pogon pa tok podatkov iz agenta ter njegovo fizično delovanje na okolico. Agenta sestavlja ≫motor≪ obnašanja, kjer planer s pomočjo zunanjega sistema (sama simulacija) nadzoruje motorični krmilnik, ta pa samo gibanje. Vsi trije deli so z informacijskim tokom povezani s predstavitvijo notranjega stanja agenta, ki določa njegovo obnašanje na najvišjem nivoju. V principu taka, na modulih temelječa arhitek-

tura, omogoča definicijo obnašanja na način, ki gre precej globlje od specifičnih scenarijev (denimo evakuacije). V tem smislu je zato precej bolj fleksibilna kot klasične implementacije, ki se uporabljajo primarno za simuliranje evakuacij.

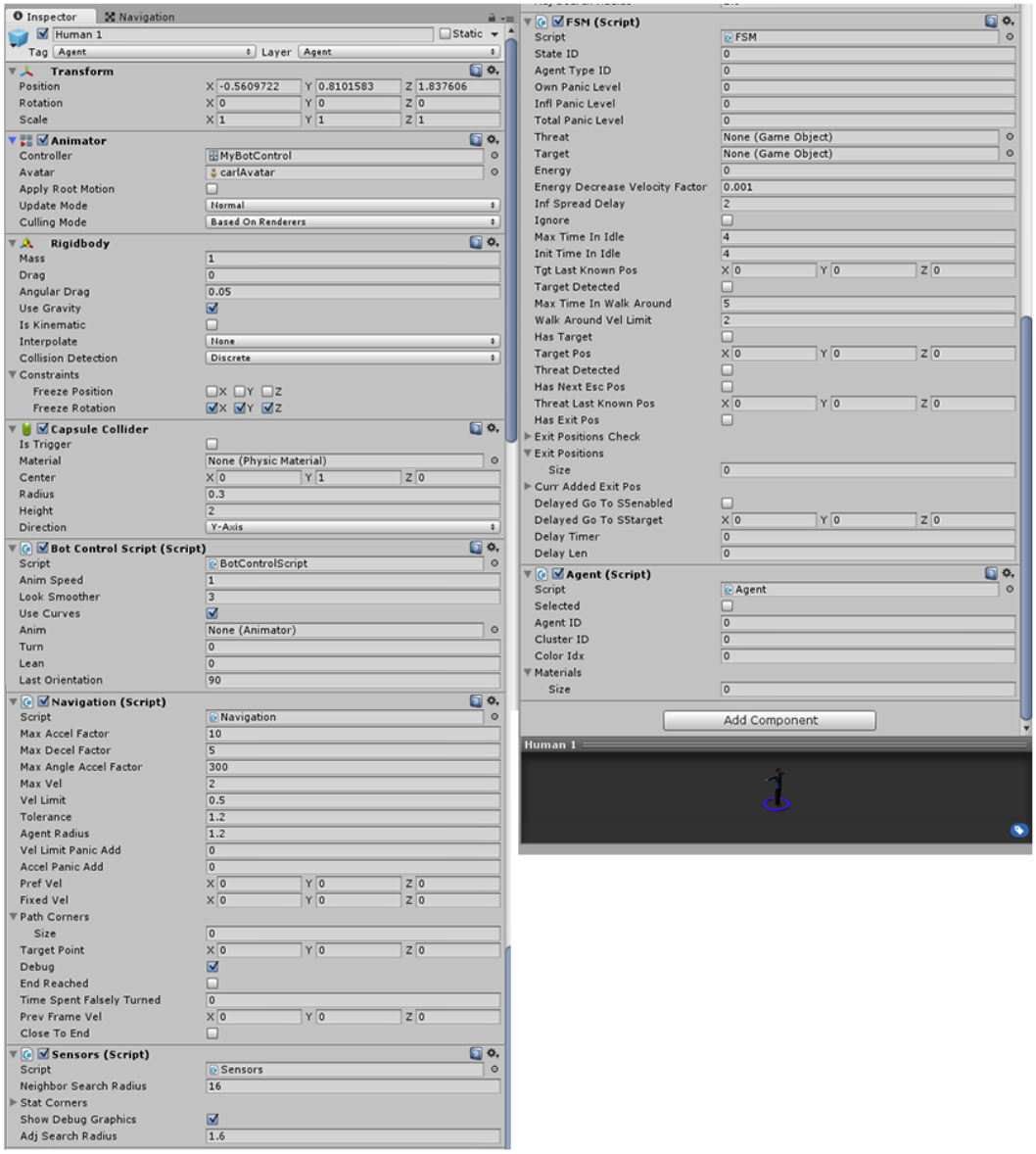
Najpomembnejši je seveda sistem obnašanja, ki predstavlja vsakemu agentu lastno notranje stanje, kjer so shranjene razne spremenljivke, pravila, ki določajo kdo in kdaj se mora na kaj odzvati in kako se na to odzvati ter prioriteto akcij pri odzivanju, spominski modul, v katerem so shranjene pretekle akcije oz. njihovi rezultati in morebitne akcije, ki se še morajo izvesti, ter gibalni modul, ki skrbi za globalno iskanje poti do cilja preko nepremičnih ovir in lokalno izmikanje dinamičnim oviram. Samo obnašanje je v primeru večje kompleksnosti sestavljeno iz več preprostejših obnašanj v hierarhični strukturi, kjer osnovni načini obnašanja direktno vplivajo na določena fizična dejanja agentov. Na najnižjem nivoju so različni načini obnašanja izvedeni kot različna stanja v končnem avtomatu.

4.1 Implementacija agenta

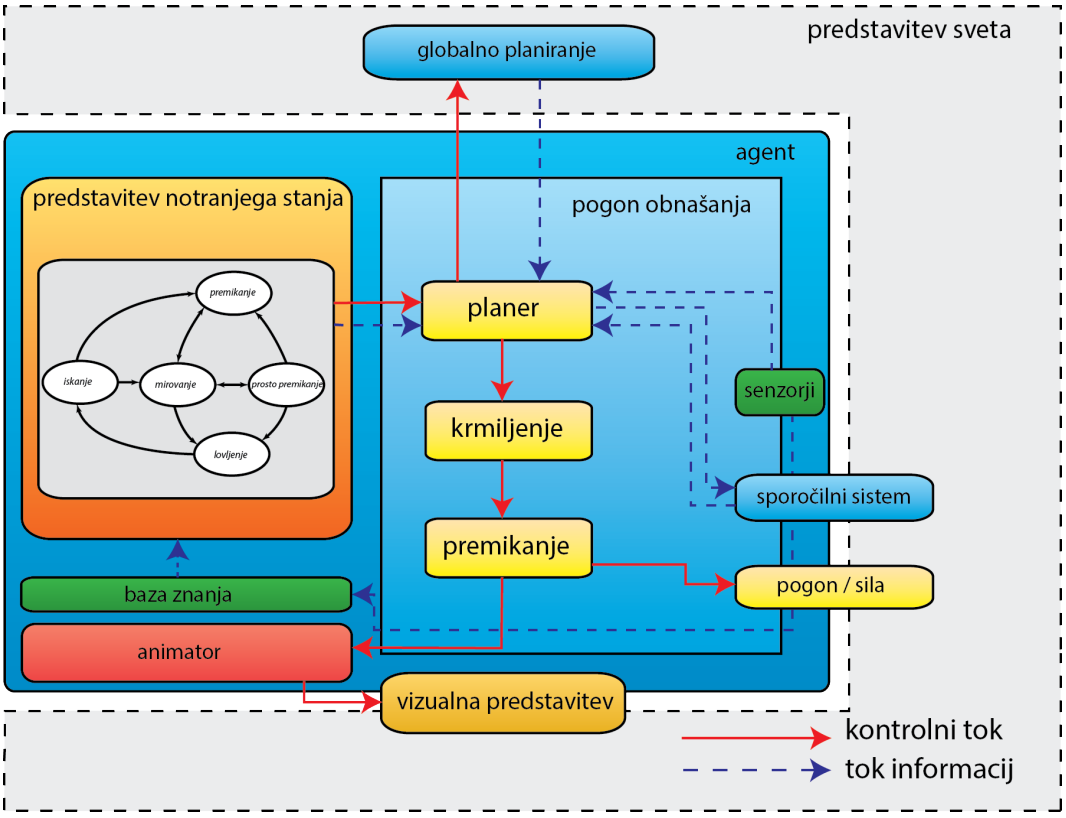
Podoben pristop sem ubral lastni aplikaciji, kjer sem poskušal doseči visoko heterogenost in hkrati nizko računsko zahtevnost. V aplikaciji je vsak agent Unity-jev igralni objekt, na katerega je nameščenih več komponent. Med njimi so skripte – skripta za globalno in lokalno navigacijo, skripta za izvajanje končnega avtomata, skripta za senzorje, skripta za animacijo in še nekaj drugih, poleg njih pa še trkalno telo v obliki kapsule, mrežni model s teksturo, model okostja za potrebe animiranja, ti. *rigidbody* oz. fizikalno telo za zmožnost apliciranja fizikalnih sil, hitrosti ipd. in drugo. Kljub povezanosti nekaterih komponent, še posebno skript (v smislu koriščenja medsebojnih storitev in večinoma bralnih dostopov do zunanjih spremenljivk), je komponente enostavno odstranjevati in menjavati.

4.1.1 Komponente

Arhitektura agenta je podobna [10]. Zunanji sistem oz. simulacija, v katero spadajo tudi ročni vnosi uporabnika, ima nadzor nad predstavitvijo notranjega stanja posameznega agenta. Ta sicer večji del časa deluje samostojno, upravlja pa s planerjem, ki določa želeno pot agenta in ki koristi zunanjo storitev globalne navigacije (iskanja poti po drevesu statičnih ovir). Planer nadzira delovanje krmiljenja,



Slika 4.1: Predstavitev agenta v okolju Unity, z vsemi skriptami in navzven vidnimi spremenljivkami



Slika 4.2: Prikaz implementirane arhitekture agenta

ki vsebuje izmikanje oviram in upoštevanje fizikalnih omejitev pri gibanju, ta pa direktno določa agentovo gibanje. Krmiljenje pošilja vhodne podatke v animator vsakega agenta, ki sicer nima lastnega vpliva na samo gibanje. Agenti imajo svoje vizualne predstavitve v obliki tridimenzionalnega modela in tekstur, samo izrisovanje pa ločeno izvaja pogon Unity. Pogon na nižjem nivoju izvaja tudi naloge, ki jih zahtevajo sporočilni sistemi (komunikacija med objekti), izvajanje fizikalnih funkcij in druge.

Agenti so si med seboj podobni v smislu, da si delijo vse komponente oz. module, je pa njihovo obnašanje najbolj odvisno od spremenljivke, ki določa tip. V implementaciji obstajata dva, ki pa bi jih bilo moč enostavno razširiti na več sorodnih tipov. Prvi tip predstavlja klasično osebo ali bitje, ki drugim agentom istega tipa ne predstavlja grožnje, z njimi komunicira in jih obravnava v okviru izmikanja ovir. Drugi tip predstavi ti. »plenilca«, ki išče agente prvega tipa in jih poskuša ujeti (in onesposobiti), pri tem pa jih agenti prvega tipa zaznavajo in bežijo, pri čemer širijo stopnjo panike na druge. Agenti prvega tipa so uporabni tako za simuliranje gibanja množic po prostoru v običajnih razmerah kot za simulacijo evakuacije oz. bežanja pred nečim v splošnem. Agenti drugega tipa so uporabni za bolj specifične situacije, kjer se agenti osredotočajo zgolj na določene stvari v prostoru in se večinoma ne pojavljajo v večjih skupinah. Zaradi načina delovanja v lokalni navigaciji agenti določenega tipa obravnavajo samo druge agente enakega tipa, pri čemer lahko agenti drugega tipa, v kolikor ni posebnih dinamičnih ovir in je takih agentov dovolj malo (in se zato ne zaletavajo med sabo), lokalno navigacijo kar izpustijo.

4.1.2 Obnašanje

Na obnašanje agenta vpliva vrsta spremenljivk, s katerimi je moč doseči heterogenost agentov. Ena izmed spremenljivk je sam tip agenta, ki določa, kateri končni avtomat se bo pri njem izvajal, na kaj bo pozoren pri uporabi senzorjev itd. Nekatere spremenljivke določajo vrednosti v stanjih, denimo največji čas, ki ga agent preživi v posameznih stanjih in stopnjo panike agenta, druge pa določajo samo gibanje in fizikalno predstavitev agenta: maso, ki ima vpliv predvsem pri kolizijah, velikost trkalnika, največjo določeno hitrost, največji dovoljeni pospešek in kotni pospešek, utrujenost, ki se veča med pospeševanjem in ohranjanjem hitrosti ter

manjša med mirovanjem, in druge.

Vsakemu agentu določenega tipa pripada stopnja panike, ki jo sestavlja lastna panika ter panika, pridobljena od drugih agentov preko senzorjev. Obe stopnji imata spremenljivo stopnjo vpliva na končno vrednost, s čimer je moč simulirati različno hitrost širjenja panike med sosednjimi agenti. V primeru velike vrednosti posamezne stopnje panike pa lahko ta prevzame večinski ali tudi celotni vpliv na končno vrednost celotne stopnje panike. Na lastno stopnjo panike vplivajo le dogodki, ki jih agent sam zaznava, recimo detekcija grožnje. Po drugi strani je pridobljena stopnja panike odvisna samo od lastne panike sosednjih agentov, in sicer od povprečja vrednosti lastne panike sosednjih agentov, pri katerih je ta večja od nič, ter najvišje zaznane stopnje lastne panike sosednjih agentov.

Vloga stopnje panike je oponašanje njenega širjenja v situacijah velike gostote agentov, pri posameznemu agentu pa ima vpliv na hitrost prehajanja med določenimi stanji, največjo hitrost premikanja in na druge spremenljivke stanj. V vsakem časovnem koraku se izračuna končna stopnja panike, nato pa se njen vpliv doda na ustrezne vrednosti.

4.1.3 Senzorji

Vsak agent ima svoje senzorje, ki svojo nalogo izvajajo v njim namenjeni skripti. Praviloma sicer v simulaciji ne bi potrebovali namenskega, agentu lastnega zaznavanja, saj bi lahko vse potrebne informacije agentom priskrboval sam simulator, ki ima poln dostop do vseh informacij. Vendar pa bi v tem primeru prišlo do prevelikega odstopanja s podobnimi situacijami v realnem življenju, kjer agenti oz. roboti ali osebki samostojno dojemajo informacije iz okolice preko svojih čutil, ki so omejena, nenatančna in zmotljiva. Agenti v simulaciji zato lahko zaznavajo samo tiste druge agente oz. ovire, ki so vidne in od njih pridobivajo samo tiste podatke, ki so zaznavni preko zunanjega sveta, denimo hitrost in velikost, pa še za te obstaja določena stopnja nedoločenosti. V ta namen so denimo prilagojeni algoritmi lokalne navigacije, ki glede na stopnjo nedoločenosti hitrosti prilagajajo hitrostne ovire. Globalna navigacija oz. iskanje poti po grafu sicer predvideva, da agent popolnoma pozna celoten teren, kar v določenih primerih ni res in je zato nerealistično, da vsak agent takoj najde komplicirano pot do cilja, je pa agentovo nevednost mogoče posnemati z naključnim gibanjem po terenu ali preiskovanjem

terena.

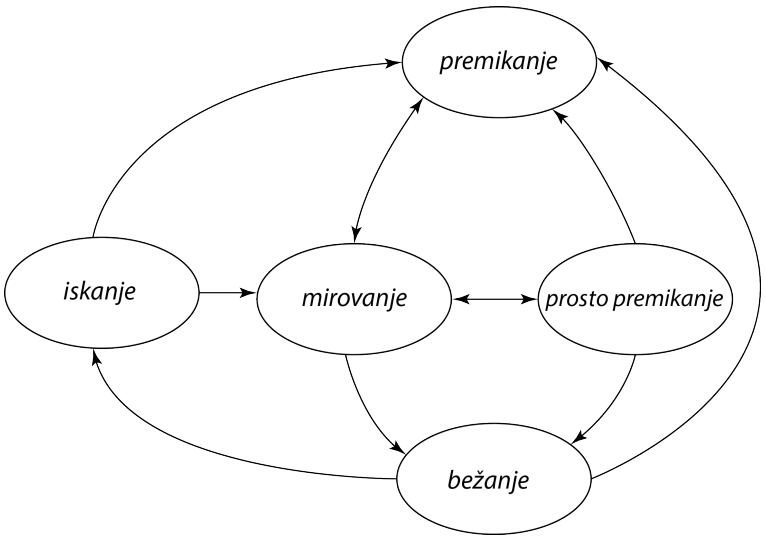
S senzorji agent pridobi podatke, ki jih druge komponente potrebujejo iz okolice. Na nižjem nivoju je vse skupaj izvedeno s sporočilnimi sistemi in uporabo fizikalnih knjižnic okolja Unity. Senzorji tako drugim komponentam dobavljajo informacije o okoliških dinamičnih ovirah, ki so potrebne za lokalno navigacijo, pri čemer morajo ugotavljati najbližje relevantne ovire in pri določenih ne-trivialnih ovirah ugotoviti pravilno obliko hitrostne ovire. Poleg tega senzorji oskrbujejo sistem stanj, ki potrebuje druge navzven vidne informacije o ostalih agentih, denimo nivo panike, tip agenta ipd.

S svojim delovanjem agenti vplivajo na okolico in povzročajo druge spremembe stanja celotnega sistema. Najosnovnejše delovanje je preprosto premikanje po prostoru. Ker imajo agenti svoje fizične predstavitve in ti. *rigidbody* (telo za fizikalno obnašanje), jim pripada določena masa in s hitrostjo tudi kinetična energija, zato delujejo tudi kot fizikalna telesa z gibanjem po lastni iniciativi, ki v primeru trkov delujejo na druga fizikalna telesa. S tem, da jih drugi agenti zaznavajo, pa tudi neposredno vplivajo na njih, denimo v širjenju panike ali s komunikacijo, s katero širijo lastno znanje na druge agente v bližini.

4.2 Končni avtomati

Končni avtomat [22] je računski model, ki se uporablja za digitalna vezja ali za uporabo v računalniških programih. V abstraktnem smislu je to naprava ali sistem, ki je ob nekem trenutku lahko v enem izmed stanj, ki jih je končno število. Stanje, v katerem je sistem ob določenem trenutku, se imenuje trenutno stanje. Veljati mora, da so med stanjem definirani prehodi; to so dogodki, ko ob izpolnjenih pogojih sistem preide v drugo stanje.

Končni avtomati se uporabljajo, ker lahko modelirajo širok spekter problemov, med njimi pa so tudi obnašanje inteligentnih sistemov, denimo virtualnih ali fizičnih agentov z določeno umetno inteligenco. Z različnimi stanji in pogoji, pri katerih prihaja do prehodov, lahko modeliramo več različnih tipov obnašanja različnih sistemov. S povečevanjem števila stanj in večjo kompleksnostjo prehajanja med njimi (v smislu, da je prehodov več) lahko določen sistem modeliramo do poljubne natančnosti, če je seveda obnašanje samih stanj dovolj kvalitetno. Iz

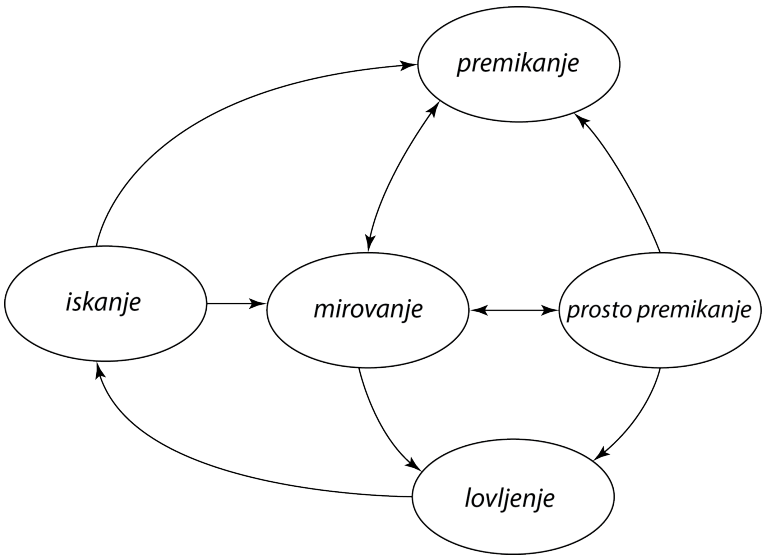


Slika 4.3: Shema končnega avtomata agenta prvega tipa

vidika uporabe v računalniških aplikacijah za modeliranje agentovega obnašanja so končni avtomati dobra praksa predvsem zaradi tega, ker zapišejo dejanja, odvisna od konteksta, v končno velik nabor stanj. So tudi enostavna za razvoj, odstranjevanje napak in v splošnem sama po sebi ne postavljajo velikih računskih zahtev pri izvajanju. Morda največja prednost pa je v intuitivnosti pri povezavi končnega avtomata s sistemom, ki ga modelira ter v fleksibilnosti njegove uporabe [10].

4.3 Implementacija stanj

V implementaciji je vsak agent v vsakem časovnem koraku v točno enem stanju. Za to, v katerem stanju bo določen agent v nekem časovnem koraku, odloča stavek *switch* v funkciji *Update()*. V primeru prehoda med stanjema bo trenutno stanje spremenilo spremenljivko identifikatorja trenutnega stanja, agent pa bo v naslednjem časovnem koraku prešel v naslednje stanje. Pri prehodu mora trenutno stanje poskrbeti za to, da so vse spremenljivke stanja, v katerega agent prehaja, nastavljene na ustrezne vrednosti. To je doseženo kar s klici posebnih funkcij za prehode, kjer vsakemu stanju pripada ena funkcija, zadolžena za ustrezno nastavitev stanju lastnih spremenljivk. Ločene funkcije za prehode omogočajo tudi



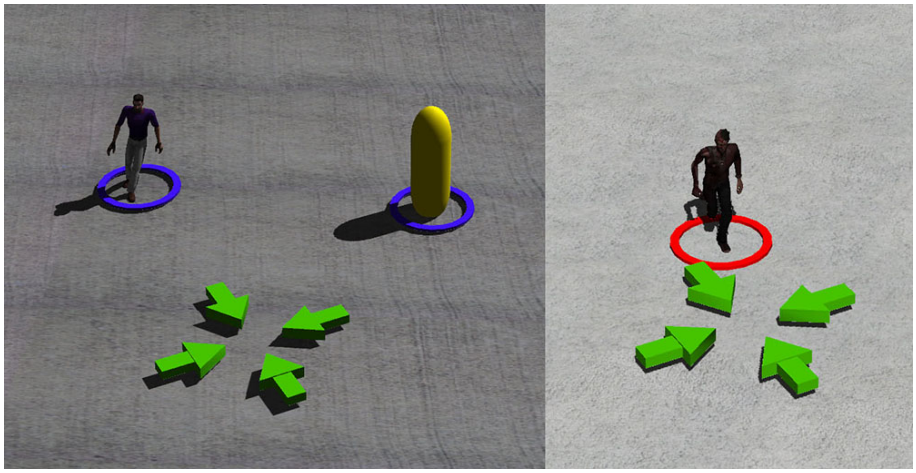
Slika 4.4: Shema končnega avtomata agenta drugega tipa

manjšo sklopljenost posameznih stanj.

Stanja so realizirana v obliki funkcij, kjer se vsaka izvede največ enkrat na vsak časovni korak. Vsaki funkciji pripada ločen nabor spremenljivk, tudi če so določene izmed njih take, da bi se lahko uporabljale za več različnih funkcij ali stanj. S tem je dosežena modularnost, saj so take funkcije s svojimi spremenljivkami podobne modulom, ki jih je enostavno dodajati in odstranjevati.

V implementaciji je 6 stanj, ki modelirajo pogoste načine obnašanja človeku-podobnih agentov, ki se znajdejo v različnih situacijah. Stanja modelirajo različne načine premikanja po prostoru, ki se pojavljajo tudi v določenih resničnih situacijah, ter pojave kot so širjenje panike in informacij med bližnjimi agenti. Izdelal sem dva tipa agentov, ki uporabljata različno kombinacijo stanj v različnih končnih avtomatih, s katerima je doseženo drugačno obnašanje. Agent prvega tipa predstavlja običajnega osebka, ki se umika grožnjam, medtem ko agent drugega tipa predstavlja osebka, ki je drugim grožnja in jih lovi, nekako v smislu fantazijskih pošati ali *zombijev*.

V samih stanjih je poudarek na vejitvah, odvisnih od tega, ali je to prvi vstop v to stanje po prehodu, zadnji vstop (pred prehodom v naslednje stanje) ali pa katera izmed vmesnih možnosti, ki jih je treba ločiti in ločeno izvajati. Po sami



Slika 4.5: Različni tipi obnašanja in predstavitve agentov

vejitvi za vsa stanja velja, da je količina računskih operacij tisti trenutek majhna ali vsaj konstantno velika, s čimer poskrbimo za minimalen vpliv na končno hitrost izvajanja simulacije. Poleg tega za več stanj velja, da je časovni korak po prvem vstopu računsko najbolj zahteven, večina nadaljnjih korakov do izstopa iz stanja pa zahteva zelo malo dejanskega računanja.

4.3.1 Mirovanje

Mirovanje je osnovno stanje vsakega agenta, v katerem se agent po lastni iniciativi ne premika. V tem stanju je agent pozoren na spremembe v okolju, ki jih dobi preko senzorjev oz. na ukaze, ki jih dobi od uporabnika. Med mirovanjem agent nima neposrednih vplivov na okolje po lastni iniciativi, se pa med tem vseeno izvajajo vsi drugi postopki, ki tečejo vsak časovni korak v drugih komponentah. Če tako agent zazna premikajočo oviro, s katero mu grozi trk, bo tako lokalna navigacija sama poskrbela za ustrezno reakcijo. V primeru, da ni zaznanih sprememb v okolju, ki bi privedle do prehoda v drugo stanje, ali ukazov uporabnika, po določenem času agent preide v stanje prostega premikanja, sicer pa se zgodi prehod v stanje, ki ustreza zaznani spremembi.

4.3.2 Prosto premikanje

V tem stanju se agent naključno premika po prostoru. Pri tem sproti generira malo oddaljene vidne točke v bližini v deloma naključni smeri in se premika proti njim, ob prispetju do vsake točke pa ponovi postopek. Vsaka generirana točka ima določeno stopnjo naključnosti, v določeni meri pa sta nov kot odmika in usmeritev levo/desno odvisna od prej izbranih kotov in usmeritev, da se doseže bolj realistično naključno premikanje. Tudi v tem stanju je agent pozoren na spremembe v okolju, zaznane preko senzorjev, ali na ukaze uporabnika. Po določenem času v tem stanju agent preide v stanje mirovanja, če medtem ni zaznal posebnih sprememb iz okolice.

4.3.3 Premikanje

Agent je v tem stanju, ko mu je bila dodeljena ciljna točka brez posebnosti. To lahko sproži drugo stanje ali pa uporabnik sam v primeru ročne nastavitve ciljne točke izbranim agentom. Ob prvem vstopu v stanje je poskrbljeno za vse potrebno v lokalni in globalni navigaciji, pri čemer se izkoristi lokalno navigacijo za končno ciljno točko in želen vektor hitrosti, v primeru da med trenutno agentovo lokacijo in ciljno točko ni nepremičnih ovir, oziroma globalno navigacijo, ki generira set vmesnih točk na poti do cilja, v primeru da se vmes nahajajo nepremične ovire. Ko agent doseže ciljno točko, se vrne v stanje mirovanja. Med gibanjem do ciljne točke oz. vsake vmesne točke lokalna navigacija poskrbi za izmikanje oviram na poti.

4.3.4 Iskanje

V stanje iskanja agent preide, ko za svoje nadaljnje delovanje potrebuje lokacijo določene stvari, saj te agent nima, ali pa je prej znano lokacijo dosegel, ne da bi se na njej iskana stvar tudi nahajala. V stanju agent prečesava celotni prostor in iskano stvar išče preko lastnih senzorjev. Iskana stvar je lahko drug agent določenega tipa, lokacija varne cone oz. izhoda ali pa drug agent, ki ima o slednjem lastno informacijo (ki jo pri tem širi na sosednje agente). Po prostoru agent najprej generira določeno število ciljnih točk, ki se nahajajo na veljavnih položajih po prostoru (ne smejo biti izven meja prostora in v preveliki bližini nepremičnih ovir). Potem naključno izbira eno izmed njih in jo poskuša doseči z lokalno navigacijo,

v primeru da vmes ni nepremičnih ovir, oziroma z globalno navigacijo, če temu ni tako. Vmes za označene označuje vse točke, ki so med potjo do izbrane točke vidne z agentovimi senzorji. Če v vmesnem času iskano stvar najde, preide v drugo stanje, ki je odvisno od tipa iskane stvari, tipa agenta in drugih spremenljivk v tem trenutku. Ko so pregledane vse točke in torej celotni prostor, agent preide v drugo stanje, ki je prav tako odvisno od agentovega tipa in drugih spremenljivk v tem trenutku.

4.3.5 Bežanje

V tem stanju je agent določenega tipa takrat, ko zazna grožnjo. Grožnja je zaznana preko senzorjev in je točkaste oblike, lahko pa je statična (npr. ogenj) ali premikajoča (ko agenta nekaj lovi). Ko je grožnja zaznana, agent poskuša uiti tako, da se premika v nasprotni smeri tako, da v tej smeri ponavljajoče generira ciljne točke. V primeru, da je pot v nasprotni smeri ovirana, se agent premika proti bližnji točki, ki je med veljavnimi najbolj oddaljena od grožnje. S tem ne preneha, dokler grožnje s senzorji ne zazna več in ko se je od zadnjega znanega položaja grožnje dovolj oddaljil. Takrat agent preide v stanje premikanja proti lokaciji varne cone, v primeru da zanjo ve, sicer pa v stanje prečesavanja, v katerem išče lokacijo varne cone ali druge agente, ki imajo informacijo o varnih conah. Med izvajanjem tega stanja lahko agent naleti na drugega agenta, ki mu sporoči lokacije izhodov in nanj razširi stopnjo panike. V tem stanju se stopnja lastne panike agenta postavi na ustrezno vrednost, odvisno od vidnosti grožnje in njene oddaljenosti.

4.3.6 Napadanje/lovljenje

V stanju napadanja je agent določenega tipa takrat, ko preko senzorjev zazna tarčo. Ko je tarča zaznana, jo zasleduje, dokler je ne doseže, pri čemer se sproži ustrezna akcija in agent preide v osnovno stanje za določen čas, ki ga nastavi v funkciji za prehod. Če je tarča prej bila zaznana, potem pa ne več, poskuša agent doseči njeno zadnjo znano lokacijo. Če je ta lokacija dosežena in tarča še vedno ni zaznana, agent preide v stanje iskanja, v katerem tarčo išče po celotnem prostoru. Stanje modelira obnašanje agenta, ki lovi druge agente, denimo divjo zver ali fantazijsko

pošast v računalniških igrah.

4.4 Izvajanje stanj in zahtevnost

Stanja koristijo razne pomožne funkcije, ki jih večinoma kličejo enkrat ali s konstantno pogostostjo na vsak časovni korak, da je znižan vpliv na računsko zahtevnost. Izmed njih so denimo funkcija, ki generira naključne veljavne točke po površini, naključne in direktno dostopne (brez uporabe globalne navigacije) točke za vtis naključnega premikanja po prostoru ter veljavne točke, čim bolj oddaljene od podane točke, uporabne za situacije bežanja.

Omenjene funkcije imajo zgoraj omejeno časovno zahtevnost in se v intervalu časovnih korakov izvajanja določenega stanja kličejo v majhnem deležu korakov – navadno med klicem vsake take funkcije preteče nekaj deset do nekaj sto časovnih korakov. Ker so agenti neodvisni in je neodvisno tudi njihovo preklapljanje med stanji, so taki, računsko bolj intenzivni koraki, razporejeni približno enakomerno čez celoten čas izvajanja stanja, zato ni večjih nihanj v končnem času trajanja vsakega časovnega koraka. V primeru naključnega premikanja je recimo časovnih korakov, kjer stanje izvaja dejansko računanje, manj kot 2 odstotka. Zato kljub temu, da lahko modelirajo prilagodljivo in fleksibilno ravnanje, izvajanje računskih operacij v samih stanj ni zahtevno in je v določenih primerih tudi zanemarljivo v primerjavi z računskimi operacijami, ki jih zahtevajo druge komponente.

Poglavje 5

Izvajanje implementacije

V implementaciji se vzporedno izvajajo skripte vseh objektov, ki skripte imajo. Največ izvajanja opravijo skripte posameznih agentov, ostale skripte pa poleg glavne, ki upravlja z vsemi agenti, določajo obnašanje okoliških objektov in njihovo odzivanje na agente. Objekti sproti posodabljajo svoje lastnosti, kjer je to potrebno, poleg tega pa izvajajo vejitve glede na trenutno situacijo, komunicirajo z drugimi objekti in dostopajo do njihovih podatkov ter izvajajo poizvedbe in storitve izvajalnega pogona. Hkratno s tem se izvaja sam pogon, ki skrbi za vso podporo izvajanju skript igralnih objektov, izrisovanje objektov v sceni itd.

5.1 Pohitritve

V splošnem se v simulacijah izračuni izvajajo najpogosteje vsak časovni korak. V pogonu Unity se časovni korak izvede v funkciji *Update()*, ki ga ima lahko vsaka skripta, odvisen pa je od najdaljšega izvajanja vseh izračunov v tej funkciji in od časa izrisovanja scene v tem koraku. Ta odvisnost lahko privede do tega, da je na časovno enoto korakov premalo, da bi lahko dosegli dovolj veliko pogostost izračunov, ali pa preveč in se bo veliko izračunov izvedlo prepogosto. Dobro je ugotoviti, koliko lahko znaša najmanjša pogostost izvajanja določenih izračunov, da bo natančnost izvedbe še vedno zadovoljiva, saj s tem vplivamo na dolžino samega časovnega koraka oz. pogostost izrisovanja. V ta namen je na voljo funkcija *FixedUpdate()*, ki se kliče s konstantno pogostostjo in je namenjena predvsem fizični manipulaciji predmetov. Še manjšo pogostost lahko dosežemo z selektiv-

	GLAVNA SKRIPTA	SKRIPTE AGENTOV	
		lokalna navigacija	FSM
START	inicializacija sprejemanje zahtev za nastavitev spremenljivk	inicializacija pošiljanje zahteve po izračunu naslednje ciljne točke	inicializacija
UPDATE (redni)	sprejemanje zahtev za posodabljanje informacij o agentih procesiranje uporabniškega vhoda	sprejemanje podatkov o sosedih izračun popravljene hitrosti z upošt. omejitev pri animaciji posredovanje navodil skripti za animacijo	izbira in izvajanje stanja posredovanje navodil navigaciji (lokalni in globalni)
UPDATE (redek)		izračun popravljene hitrosti (opcijsko)	
LATE UPDATE	broadcast sporočila o posodobitvi informacij ponastavljanje spremenljivk	posodobitev informacij	

Slika 5.1: Prikaz nalog, ki jih skripte objektov izvajajo vzporedno

	SKRIPTE AGENTOV		DRUGE SKRIPTE	
	senzorji	skripta za animacijo	drugi objekti, kamera itd.	globalna navigacija
START	inicializacija	inicializacija	inicializacija	inicializacija izračun začetne poti
UPDATE (redni)	iskanje sosedov pošiljanje podatkov lokalni navigaciji	izbira in izvajanje stanja sprejemanje, procesiranje navodil lokalne navigacije posredovanje navodil animatorju	ugotavljanje, ali je objekt izbran itd. posodobitve informacij, relevantnih za lok. navigacijo	izračun nove poti posredovanje poti lokalni navigaciji posodabljanje cen plasti
UPDATE (redek)	medsebojna komunikacija		medsebojna komunikacija	
LATE UPDATE	posodobitev informacij			

Slika 5.2: Prikaz nalog, ki jih skripte objektov izvajajo vzporedno (nadaljevanje)

nim klicanjem funkcij zgolj v določenih klicih *FixedUpdate()*; tako lahko nekatere izračune izvajamo denimo zgolj 5-10-krat na sekundo. Če poskrbimo, da je več teh redkih klicev razporejenih čim bolj enakomerno, lahko tako še dodatno zmanjšamo dolžino povprečnega časovnega koraka.

5.2 Večnitno izvajanje

Izvajanje na več nitih lahko močno zmanjša potrebno delo vsakega procesorskega jedra, če je teh več in če je možna ustrezna razdelitev naloge na več vzporedno izvajajočih se niti. Žal pogon Unity ne omogoča izvajanje prevedenih skript na več nitih v trenutni verziji, bo pa to mogoče v verziji 5.0.

Kljub temu so skripte v simulaciji napisane tako, da delujejo čim bolj neodvisno od drugih, saj ima vsak agent svoj nabor skript, ki v skripte drugih agentov ne dostopajo s pisalnimi dostopi, glavna skripta (ki se ne more izvajati v več nitih) pa ima v primerjavi z delom vseh skript agentov zanemarljivo malo dela med samim izvajanjem. Tudi algoritmi za lokalno navigacijo so zasnovani tako, da se na posameznem agentu izvajajo neodvisno od drugih. Taka zasnova med drugim omogoča tudi to, da posamezne naloge agentov razredčimo, da je pogostost izvajanja nalog pri vsakem agentu manjša od pogostosti časovnih korakov in da so naloge enakomerno razporejene po času.

Poglavje 6

Primeri uporabe in rezultati

6.1 Situacije

Izdelal sem več scen, ki prikazujejo različne primere uporabe agentov v različnih situacijah. Scene je moč izbirati preko glavnega menija, kamor lahko uporabnik pride preko gumba v uporabniškem vmesniku, če se že nahaja v eni izmed scen. V uporabniškem vmesniku je za vsako sceno postavljenih nekaj gumbov, s katerimi se sproža razne dogodke, poleg tega pa je v ločenem delu vmesnika moč vplivati na to, kako velika skupina agentov bo ustvarjena in njen izgled in tip agentov. Vse agente je mogoče označevati, jim nastavljati cilj in izgled. Pri ustvarjanju novih agentov ima uporabnik možnost dveh različnih vizualnih predstavitev agentov, ki sta sicer po obnašanju enakovredni. Razlikujeta se v odsotnosti kompleksnega trodimenzionalnega modela ter odsotnosti animatorskih komponent pri preprostejši predstavitvi, zaradi česar je mogoče simulirati večje število agentov. Po sceni se je s kamero mogoče prosto premikati v vseh treh dimenzijah ali pa zgolj po ravnini na določeni višini, kar se izbira z ustreznim gumbom.

6.1.1 Klasična evakuacija

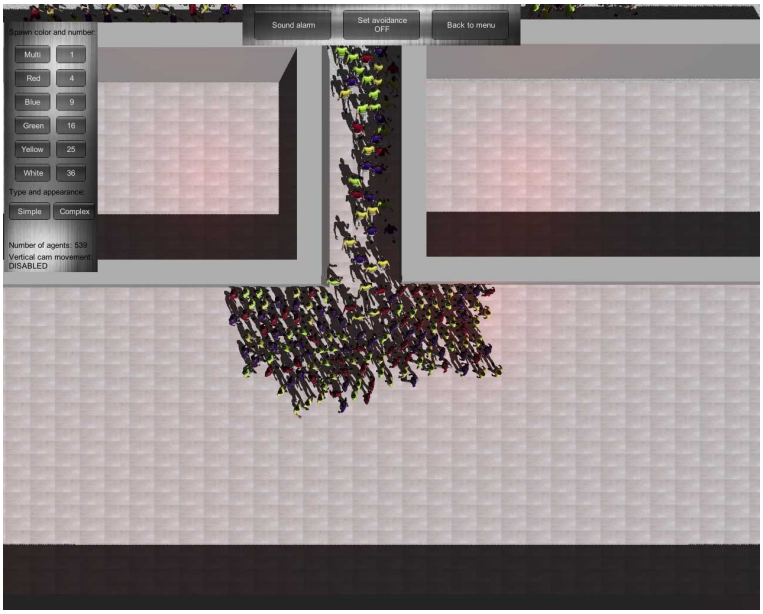
Prva scena prikazuje zgradbo z več prostori, po katerih se sprva naključno giblje srednje veliko število agentov (100-900). Ko se sproži alarm za evakuacijo, agenti pričnejo teči proti dvema zunanjima zavetiščema. V zgradbi so prostori povezani z ozkimi prehodi in hodniki, med zgradbo in zunanjima zavetiščema pa je cesta,

Slika 6.1: 580 agentov pri evakuaciji iz stavbe

na kateri iz obeh smeri poteka promet. Agenti v tej situaciji vedo za lokacijo obeh zunanjih zavetišč, zato na začetku ni zmede in iskanja zavetišč. Promet je predstavljen z več tovornjaki, ki so vsi realizirani kot dinamične ovire, ki okoliških agentov ne upoštevajo pri svojem premikanju, zato morajo pri prečkanju ceste vso potrebno delo za izmikanje prometu opraviti agenti. Ker algoritmi, ki temeljijo na konceptu vzajemnih hitrostnih ovir, predpostavljajo vzajemno delovanje ovire, je potrebno uvesti popravek za take dinamične ovire tako, da se center hitrostnih ovir premakne za vektor $-\frac{v_a - v_b}{2}$. Dinamične ovire so sicer pravokotnih oblik in jih je mogoče obravnavati temu primerno glede na orientacijo glede na agenta, moč pa jih je tudi posplošiti kot ovire okroglih oblik, kar ne zahteva posebne obravnave.

V več zaporednih zagonih simulacije z različnim številom agentom sem zabeležil, da 540 agentov zgradbo skozi dva vhoda zapusti v okoli 120 sekundah, 400 agentov v okoli 90 sekundah, 275 v okoli 60 sekundah, 175 agentov pa v okoli 43 sekundah. Med simulacijo je mogoče opaziti več pojavov, ki so značilni tudi za resnične situacije gibanja množic. Prvi je formacija lokov, ki sledi gnetenju pred ozkimi izhodi, ko je pred njimi dovolj veliko število agentov, katerih cilj je, da pridejo skozi. Proti zunanjim robovom zgoščin agentov je moč opaziti vedno višje hitrosti, saj ima plast agentov na samem robu prosto zunanjo polovico prostora, plasti agentov proti notranjosti pa se lahko nekoliko počasneje gibljejo vzporedno z zunanjo, saj v tem primeru ne pride do kolizij. Podobno se občasno v zgoščinah formirajo tokovi, če se večje število agentov poskuša prebiti proti cilju v isti smeri. Tokovi se dinamično spreminjajo in izginjajo glede na število in želeno hitrost drugih agentov.

Z gumbom na uporabniškem vmesniku je moč vklopiti ali izklopiti uporabo



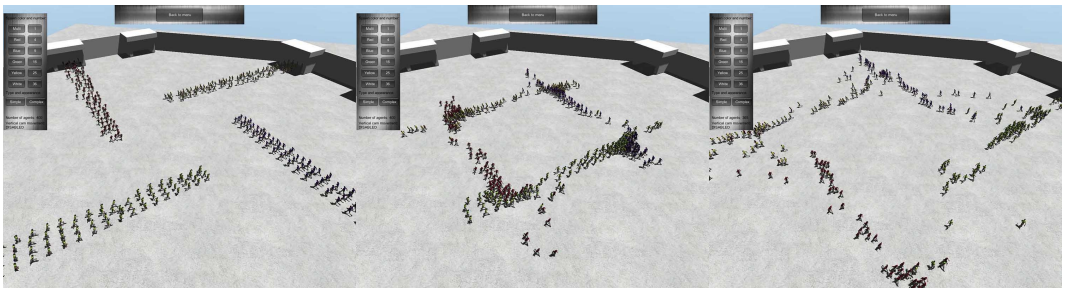
Slika 6.2: Formacija lokov pred zožanimi prehodi

izmikanja oviram. Izklop povzroči precej bolj agresivno vedenje agentov, izrivanje agentov s strani večjih skupin ter občutek gibanja agentov, podobnega gibanju fizikalnih delcev.

6.1.2 Križanje poti v odprtem prostoru

V tej sceni se sreča več skupin agentov, kjer se vsaka skupina želi pomakniti proti svojem cilju na drugi strani prostora. Poti do cilja različnih skupin se križajo, kar povzroči zgoščine na mestih križanja. Zgoščine se izkažejo kot zelo omejujoče za gibanje agentov v njih, vendar v njih ne prihaja do kolizij. Poskušal sem z več načini znižanja vpliva takih zgoščin. Prvi je, da so agenti v skupinah že na začetku v večjih razmakih, kar omogoča, da se velik del dveh skupin seka brez potrebe po medsebojnem izmikanju. Drugi je ustrezna postavitev skupin, da med prečkanjem nastanejo tokovi agentov, ki vodijo v ciljem ustrezne smeri.

V drugem primeru scene se ustvarijo štiri skupine agentov, ki se poskušajo premakniti na drugo stran prostora, pri čemer imata po dve skupini vzporedno pot želenega gibanja. Pri tem se pot gibanja posamezne skupine ukrivlja in na



Slika 6.3: 400 agentov v štirih skupinah pri prečkanju v odprtem prostoru

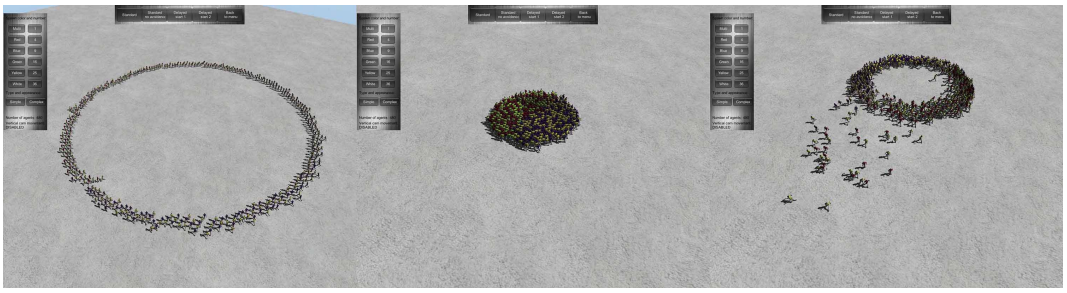
štirih lokacijah srečanja dveh skupin nastanejo posebni vzorci.

6.1.3 Križanje poti v krožni postavitvi

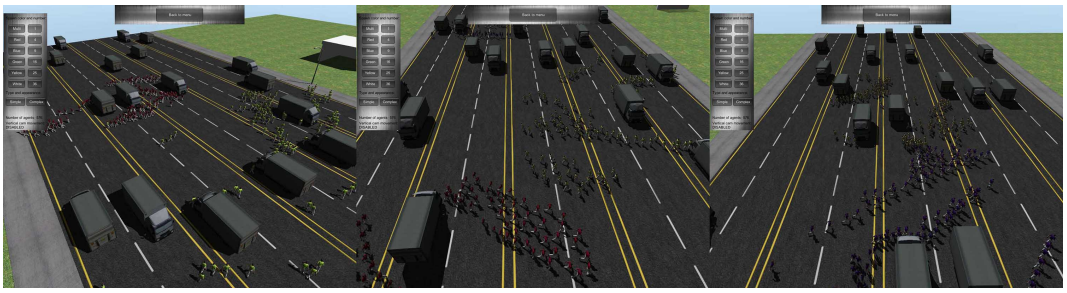
V tej sceni so agenti postavljeni v krožnem vzorcu z nastavljivim radijem in številom agentov. Vsak agent poskuša priti na njemu nasproti ležečo lokacijo na krožnici. Neposredne poti vseh agentov se torej vse križajo v središču krožnice. Izdelal sem več načinov križanja. V prvem agenti ne uporabljajo algoritma za izmikanje oviram in uporabljajo zgolj trkalna telesa za preprečevanje kolizij. V središčni točki se najprej pojavi okrogla zgoščina, potem pa se zgoščina zaradi skupnega delovanja sil vseh agentov v njej začne sukati. Sukanje nastane zaradi netočnosti pri premikanju in zaznavanju kolizij, sicer bi zgoščina od nekega trenutka naprej ostala fiksna. Po ustreznem zasuku vsak agent neovirano hitro pride na svojo ciljno točko. V primeru uporabe izmikanja oviram se podobna stvar zgodi po nekoliko daljšem času v primeru ne prevelikega števila agentov, sicer pa se zgoščina zelo počasi premika in traja več minut, da še zadnji agenti dospejo na ciljne točke. Zadnja dva primera uporabljata zakasnjen začetek premikanja agentov – najprej naključno, potem pa še urejeno v parih. Ob ustrezni nastavitvi zamika do sredinske zgoščine ne pride in agenti hitro dosežejo cilje.

6.1.4 Križanje poti dinamičnim oviram

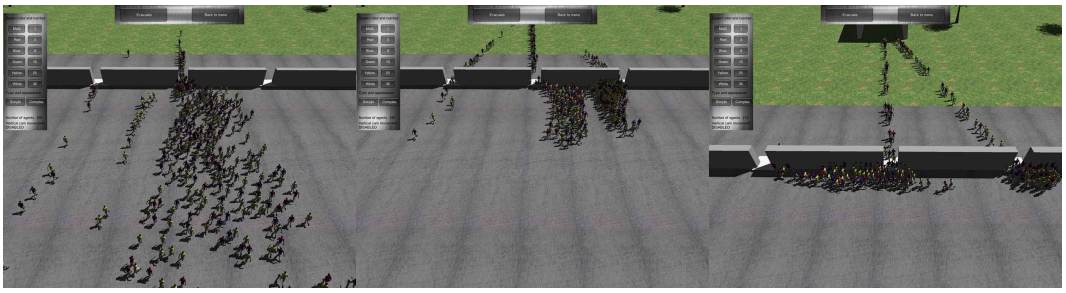
Scena vsebuje večpasovno cesto, na kateri promet poteka v obeh smereh v izmeničnih pasovih. Iz obeh strani jo skušajo prečkati štiri skupine agentov, ki se izmikajo tako ostalim agentom kot prometu. Opazno je upočasnjevanje agentov v



Slika 6.4: 480 agentov v različnih stopnjah pri prečkanju poti v krožni postavitvi



Slika 6.5: 580 agentov v štirih skupinah pri prečkanju večpasovne ceste



Slika 6.6: Evakuacija skozi tri izhode in popravljanjem cen plasti

primeru, da v tistem trenutku ne morejo prečkati trenutnega pasa, in pasovi agentov, ki stečejo, ko so ovire njihovo pot že prečkale ali ko je mogoče prometni pas prečkati z ukrivljenim premikanjem in se oviri izogniti. Pri tem agenti upoštevajo hitrosti prometa in lastne največje hitrosti, zato je malo primerov, ko je ovira prehitra in ji agent zaradi omejene hitrosti ne more ubežati (se pa zaradi dodatnih omejitev še vedno dogajajo).

6.1.5 Evakuacija skozi več vzporednih izhodov

V tej sceni se 300-600 agentov sprva nahaja v pravokotnem prostoru, ki ima na eni izmed stranic tri izhode, ki vodijo do zunanjega zaklonišča. Cilj evakuacije vseh agentov proti temu zaklonišču je, da ga zapustijo z uporabo vseh treh izhodov, če je eden izmed njih preveč zaseden (v smislu velikega števila agentov, ki se skozenj giblje in ki še čaka pred njim). Večina agentov bo ob začetku evakuacije izračunala pot preko enega izmed izhodov, saj je pot skozenj najkrajša. Zaradi postavitve izhodov bo večina agentov izbrala pot skozi isti izhod.

Smiselno je, da agenti sproti ponovno izračunavajo pot in upoštevajo popravljene cene plasti v drevesu navigacijske mreže, na kateri obstajajo ločene plasti v okolici vsakega izhoda. Če je pred določenim izhodom dovolj veliko število agentov, se glede na to število popravi cena plasti okoli izhoda, agenti, ki niso v njegovi neposredni bližini, pa dobijo navodilo za ponoven izračun poti. Navodila za ponovni izračun se potem periodično ponavljajo na določen časovni razmik, pri čemer so razporejena na vse časovne korake, zaradi česar ne pride do velike preobremenjenosti, ko bi vsi agenti morali ponovno izračunati pot v istem časovnem

Slika 6.7: Prikaz panike ob požaru

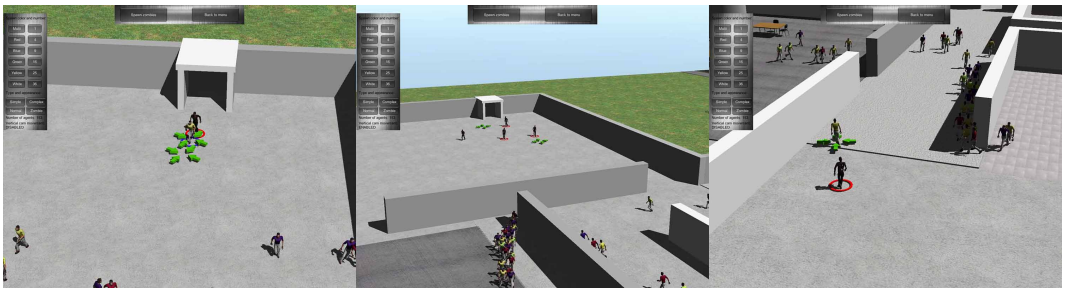
koraku.

Potrebna je natančna nastavitev spremenljivk, da se doseže ustrezno ravnanje agentov, predvsem nastavljanje cen plasti izhodov v odvisnosti od števila agentov v njihovi bližini, ugotavljanje izhodom bližnjih agentov in pogostost ponovnega izračunavanja poti.

6.1.6 Širjenje panike - ogenj

V tej sceni imajo agenti omejeno znanje o več različnih izhodih iz stavbe in se sprva po njej naključno premikajo v mirnem stanju. Med tem zaznavajo vidne izhode in informacijo o njih delijo s sosednjimi agenti, ti pa jo delijo naprej. Med prenosom informacije med dvema agentoma je prisoten določen zamik, da se prepreči takojšnja razširitev informacij na vse agente (v primeru, ko so vsi v sosedski povezanosti z drugimi). S pritiskom na gumb se na določenih mestih znotraj zgradbe začne požar, ki ga bližnji agenti zaznajo. S tem se panika iz teh agentov prenese na druge, agenti pa poskušajo stavbo zapustiti, v primeru, da vedo za vsaj en izhod, sicer pa ga začnejo iskati. Tudi širjenje panike ima določen zamik med posameznima agentoma. V tem primeru je opazno, da agenti izberejo pot skozi tisti izhod, o katerem imajo informacijo, ta pa ni vedno njim najbližji – odvisen je od postavitve in gibanja drugih agentov, od katerih prejmejo informacijo o lokaciji izhoda.

Z nastavljanjem vrednosti spremenljivk, kot so zamik širjenja panike in informacij, razdalja širjenja panike in informacij ter učinkovanje panike na premikanje agentov, je moč spreminjati celoten potek evakuacije. V določenih primerih



Slika 6.8: Prikaz panike in lovljenja zombijev v sceni

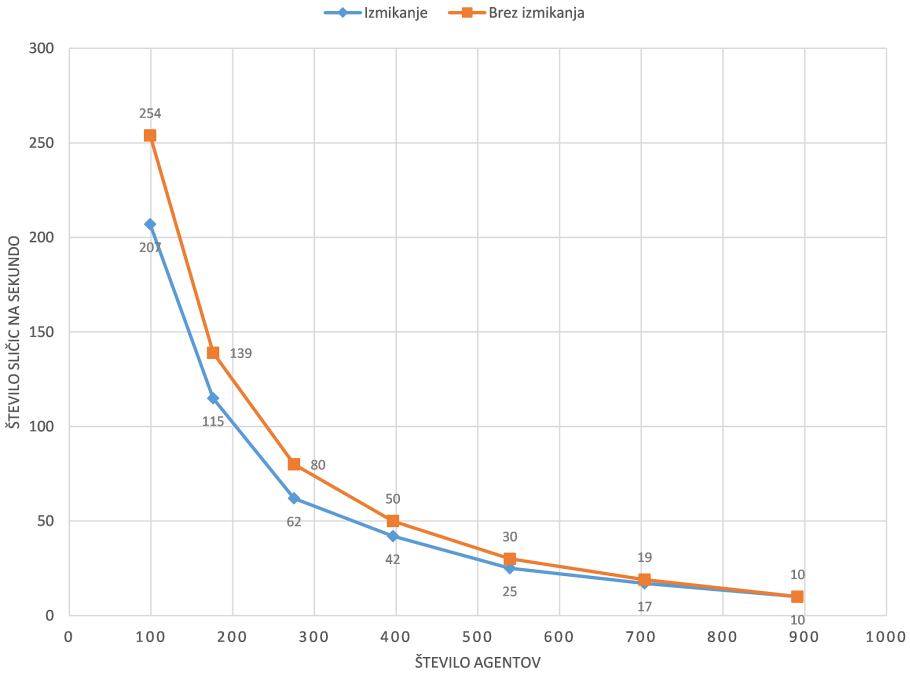
tako večina agentov uide skozi en ali dva (neoptimalna) izhoda, v drugih primerih ostajajo v zgradbi agenti, na katere se panika ni razširila in se še naprej po njej naključno premikajo, dokler tudi sami ne zaznajo nevarnosti.

6.1.7 Širjenje panike - zombiji

Scena je podobna kot zgornja, le da nevarnosti agentom ne predstavljajo izbruhi ognja, temveč drugi agenti (*zombiji*), ki v sceno vstopijo skozi njim namenjene vhode. Ti agenti začno loviti druge, če jih zaznajo. Medtem jim drugi agenti skušajo ubežati, če so v njihovi neposredni bližini, ali pa začno iskati izhod iz stavbe, če niso. Ko zombi ujame agenta, se ta po določenem času sam spremeni v zombija in se začne obnašati temu primerno. Z nastavljanjem sposobnosti zombijev in drugih agentov, hitrostjo spreminjanja ujetega agenta v zombija, radija zaznavanja in drugih spremenljivk je mogoče spreminjati izid poteka situacije; predvsem je od tega odvisno število agentov, ki uide v zatočišča, število ujetih agentov in število agentov, ki panike ne zaznajo in se še naprej naključno premikajo po stavbi, dokler tudi sami ne naletijo na nevarnost.

6.2 Hitrost izvajanja

Hitrost izvajanja aplikacije se meri v številu izrisanih sličic na sekundo, angl. *frames per second, FPS*. Zaželeno je, da je to število dovolj veliko, da človeku še da vtis gladkega gibanja. Ker je število sličic na sekundo odvisno od veliko spremenljivk, sem meritve izvajal v sceni klasične evakuacije z enakimi parametri in



Slika 6.9: Odvisnost števila sličic na sekundo od števila agentov v sceni klasične evakuacije

v trenutku, ko je največ agentov pred določenim izhodom. Izvedel sem meritve pri sedmih različnih vrednostih števila agentov v sceni, pri vsaki pa sem izmeril vrednost z uporabo izmikanja oviram in brez nje. Pri uporabi izmikanja oviram so agenti zaznavali druge v oddaljenosti osmih lastnih radijev, kar je srednje velika vrednost. Dinamične ovire brez vzajemnega izmikanja so zaznavali na nekajkrat večji razdalji, vendar se lahko njihov vpliv na računsko zahtevnost zaradi manjšega števila zanemari. Grafične nastavitve so bile nastavljene na najnižjo vrednost in zaradi uporabe sposobne grafične procesne enote lahko z gotovostjo trdim, da je bil vpliv zahtevnosti izrisa samih agentov minimalen in celo zanemarljiv v primerjavi z zahtevnostjo izvajanja njihovega obnašanja; delo centralne procesne enote je zato predstavljalo ti. *ozko grlo*.

Uporabljena strojna oprema je bila:

- CPE: Intel i5 4670K, 4.0GHz (4 fizična jedra, 1 nit na jedro)
- GPE: AMD Radeon R9 280X
- RAM: 8GB, DDR3

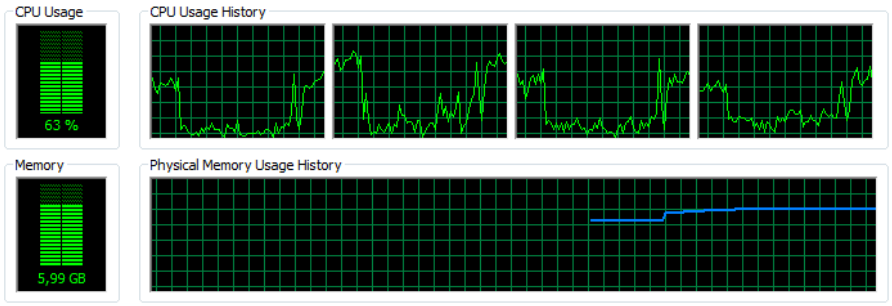
Kot je razvidno, ima uporaba izmikanja z naraščajočim številom agentov v sceni majhen vpliv na samo hitrost izvajanja. Razlog v tem je naraščajoč vpliv drugih stvari, ki so tudi odvisne od števila agentov, denimo izvajanje njihovega obnašanja, komuniciranja, fizikalnih testov in stvari, ki jih v ozadju počne sam pogon, med njimi tudi pošiljanje ukazov za izris grafični procesni enoti (ki sama sicer nima veliko dela).

Poglavje 7

Zaključek

Predstavil sem področje simulacije množic in heterogenih agentov ter kombinacije uporabe teh dveh pristopov. Pri tem sem obravnaval potrebe in omejitve pri realno-časovnem izvajanju in izrisovanju ter predstavil primere uporabe v praktične namene. Opisal sem razvojno okolje in pogon Unity, njegove sestavne dele ter način izgradnje in izvajanja aplikacij v njem. Podrobneje sem razdelal področje globalne in lokalne navigacije agentov, napisal lasten algoritem izmikanja oviram in predstavil njegovo delovanje ter kot alternativno aplikaciji priredil še prosto-dostopen isto-namenski algoritem ORCA. V drugem delu sem predstavil zgradbo oz. arhitekturo heterogenega agenta, ki se v drugih virih pogosto uporablja, ter napisal in podrobneje predstavil implementacijo podobne arhitekture v lastni aplikaciji. Predstavil sem končni avtomat in vsakega izmed stanj obnašanja v njem ter opisal izvajanje aplikacije. Za konec sem komentiral in analiziral rezultate v vsaki izmed scen, ki prikazujejo različne situacije.

Področje simulacije množic agentov, ki so sposobni določene stopnje lastnega odločanja, se razvija zelo hitro in za pričakovati je vedno večjo uporabo v praktičnih aplikacijah, pa naj bodo to računalniške igre, nadomestki za statiste v filmski industriji ali zaradi takih simulacij izboljšani javni objekti in infrastruktura za namene čim hitrejše evakuacije. Na napredek je računati v algoritmih izmikanja oviram, ki se bodo še bolj osredotočali na učinkovito izvajanje in na inteligentno ravnanje za izogibanje zgoščin v povezavi z globalno navigacijo. Prav tako napredek teče v naprednosti, "inteligentnosti"posameznega agenta in tako kaže na večjo splošno uporabnost takih simulacij.



Slika 7.1: Unity izrablja več procesorskih sredic, vendar se skripte izvajajo zgolj na eni in zato izraba več CPE ni optimalna

7.1 Nadaljnje delo

Zaradi širine področja in neprestanega napredka v njem bi bile tudi v moji aplikaciji mogoče izboljšave v različnih smereh. Ena izmed njih je deljeno izvajanje skript agentov na ločenih procesnih enotah, kar trenutno ni zvedljivo zaradi omejitev trenutne različice pogona Unity, bo pa mogoče z njegovo naslednjo verzijo (5.0), saj je vsa koda napisana s tem v mislih in se vsak agent dejansko izvaja ločeno in neodvisno od drugih.

Prav tako obstajajo druge pomanjkljivosti pogona Unity, ki se bolj odrazijo pri velikem številu agentov. Ena izmed njih je sporočilni sistem, ki lahko precej upočasni izvajanje, zato sem moral v implementaciji minimizirati količino podatkov, ki se sproti pošiljajo. Še ena pomanjkljivost je nezmožnost dinamičnega prilagajanja navigacijske mreže v primeru spreminjanja položajev nepremičnih ovir. Obe možnosti deloma rešujejo neuradne, pogosto plačljive razširitve, ki pa so večinoma tudi težavnejše za implementacijo.

Še en primer bi bilo recimo koristenje informacij o skupinah agentov, kot jih koristijo določeni avtorji za potrebe navigacije in obnašanja na splošno. V svoji aplikaciji sem spisal algoritem k-means [23], ki ima dobre rezultate za iskanje več različnih skupin agentov. Agentom v skupinah se priredi določena barva, pripadnost skupinam pa se spreminja dinamično v odvisnosti od lokacije agentov. Identifikator skupine bi se lahko uporabil za potrebe lokalne navigacije na daljše razdalje, kjer se celotna skupina aproksimira kot velika premikajoča se ovira. Ven-

Slika 7.2: Avtomatično razpoznavanje in dodeljevanje skupin (prikazane z barvami agentov) glede na lokacijo

dar je zahtevna tako koristna uporaba informacije o pripadnosti skupinam kot tudi ustrezna pridobitev pripadnosti, saj je potrebno dinamično spreminjanje števila skupin glede na prostorsko postavitev in gostoto agentov, kriteriji za določitev skupin pa morajo upoštevati tudi druge lastnosti agentov. Na sliki 7.2 je prikazan algoritem k-means, ki dinamično dodeljuje skupine in barve agentom v sceni.

7.2 Sklep

Z rezultati sem zadovoljen, saj mi je uspelo izdelati ciljno arhitekturo agenta in doseči želeno gibanje in obnašanje množic v testnih primerih. Omejitve pogona, predvsem v smislu večnitnega izvajanja in počasnosti določenih metod za komuniciranje in fizikalne teste, preprečujejo realno-časovno izvajanje večjega števila (nekaj tisoč) agentov, je pa z uporabo enostavnejšega vizualnega prikaza agentov vseeno mogoče izvajanje okoli 1000 agentov, medtem ko se značilni pojavi pri gibanju množic pojavijo že pri dosti manjših številih agentov v sceni. Agenti se obnašajo situacijam primerno in se spremembam dobro prilagajajo, izgled animacij pa je dovolj realističen in skladen z dejanskim premikanjem.

Literatura

[1] Eric Bouvier, Eyal Cohen, and Laurent Najman. From crowd simulation to airbag deployment: particle systems, a new paradigm of simulation. *Journal of Electronic imaging*, 6(1):94–107, 1997.

[2] J Demange, S Galland, and A Koukam. Towards the agentification of a virtual situated environment for urban crowd simulation. 2011.

[3] Epic. Unreal development kit - navigation mesh reference, 2014. [Online; accessed 17-August-2014].

[4] Abhinav Golas, Rahul Narain, and Ming Lin. Hybrid long-range collision avoidance for crowd simulation. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 29–36. ACM, 2013.

[5] Stephen J Guy, Jatin Chhugani, Changkyu Kim, Nadathur Satish, Ming Lin, Dinesh Manocha, and Pradeep Dubey. Clearpath: highly parallel collision avoidance for multi-agent simulation. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 177–187. ACM, 2009.

[6] Tomoki Hamagami and Hironori Hirata. Method of crowd simulation by using multiagent on cellular automata. In *Intelligent Agent Technology, 2003. IAT 2003. IEEE/WIC International Conference on*, pages 46–52. IEEE, 2003.

[7] Hubert Ludwig Kluepfel. *A cellular automaton model for crowd movement and egress simulation.* PhD thesis, Universität Duisburg-Essen, Fakultät für Physik, 2003.

[8] Jiang Li-jun, Chen Jin-chang, and Zhan Wei-jie. A crowd evacuation simulation model based on 2.5-dimension cellular automaton. In *Virtual Environments, Human-Computer Interfaces and Measurements Systems, 2009. VECIMS'09. IEEE International Conference on*, pages 90–95. IEEE, 2009.

[9] Rahul Narain, Abhinav Golas, Sean Curtis, and Ming C Lin. Aggregate dynamics for dense crowd simulation. In *ACM Transactions on Graphics (TOG)*, volume 28, page 122. ACM, 2009.

[10] Jurgen Rossmann, N. Hempe, and P. Tietjen. A flexible model for real-time crowd simulation. In *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pages 2085–2090, Oct 2009.

[11] Sharad Sharma, Stephen Otunba, and Jingxin Han. Crowd simulation in emergency aircraft evacuation using virtual reality. In *Computer Games (CGAMES), 2011 16th international conference on*, pages 12–17. IEEE, 2011.

[12] Barry G Silverman, Michael Johns, Jason Cornwell, and Kevin O'Brien. Human behavior models for agents in simulators and games: part i: enabling science with pmfserv. *Presence: Teleoperators and Virtual Environments*, 15(2):139–162, 2006.

[13] Jamie Snape, Jur van den Berg, Stephen J Guy, and Dinesh Manocha. The hybrid reciprocal velocity obstacle. *Robotics, IEEE Transactions on*, 27(4):696–706, 2011.

[14] Avneesh Sud, Russell Gayle, Stephen Guy, Erik Andersen, Ming Lin, and Dinesh Manocha. Real-time simulation of heterogeneous crowds. Technical report, Tech. rep., University of north california, 2007.

[15] Jur Van Den Berg, Stephen J Guy, Ming Lin, and Dinesh Manocha. Optimal reciprocal collision avoidance for multi-agent navigation. In *Proc. of the IEEE International Conference on Robotics and Automation, Anchorage (AK), USA*, 2010.

[16] Jur Van den Berg, Ming Lin, and Dinesh Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *Robotics and Automation, 2008.*

ICRA 2008. IEEE International Conference on, pages 1928–1935. IEEE, 2008.

[17] Guillermo Vigueras, Miguel Lozano, JM Ordua, and Francisco Grimaldo. Improving the performance of partitioning methods for crowd simulations. In *Hybrid Intelligent Systems, 2008. HIS'08. Eighth International Conference on*, pages 102–107. IEEE, 2008.

[18] Yongwei Wang, Michael Lees, and Wentong Cai. Grid-based partitioning for large-scale distributed agent-based crowd simulation. In *Proceedings of the Winter Simulation Conference*, page 241. Winter Simulation Conference, 2012.

[19] Yongwei Wang, Michael Lees, Wentong Cai, Suiping Zhou, and Malcolm Yoke Hean Low. Cluster based partitioning for agent-based crowd simulations. In *Winter Simulation Conference*, pages 1047–1058. Winter Simulation Conference, 2009.

[20] Xing Wei, Muzhou Xiong, Xuguang Zhang, and Dan Chen. A hybrid simulation of large crowd evacuation. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 971–975. IEEE, 2011.

[21] Wikipedia. Bresenham's line algorithm — wikipedia, the free encyclopedia, 2014. [Online; accessed 17-August-2014].

[22] Wikipedia. Finite-state machine — wikipedia, the free encyclopedia, 2014. [Online; accessed 17-August-2014].

[23] Wikipedia. K-means clustering — wikipedia, the free encyclopedia, 2014. [Online; accessed 17-August-2014].

[24] Wikipedia. Midpoint circle algorithm — wikipedia, the free encyclopedia, 2014. [Online; accessed 17-August-2014].

[25] He Xiao and Chunlin He. Behavioral animation model for real-time crowd simulation. In *Advanced Computer Control, 2009. ICACC'09. International Conference on*, pages 250–254. IEEE, 2009.

[26] Xinxin Zhao, Yong Zhang, Dehui Kong, and Baocai Yin. Comparision of real-time crowd simulation methods based on parallel architecture. In *Digital Home (ICDH), 2012 Fourth International Conference on*, pages 146–150. IEEE, 2012.