UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Kosta Đurišić

ODLUČIVANJE U POTEZNIM TAKTIČKIM VIDEO IGRAMA ZASNOVANO NA TEHNIKAMA VEŠTAČKE INTELIGENCIJE

master rad

Beograd, 2021.

Mentor:

dr Filip Marić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Vesna Marinković, docent
Univerzitet u Beogradu, Matematički fakultet

dr Predrag Janičić, redovni profesor
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Naslov master rada: Odlučivanje u poteznim taktičkim video igrama zasnovano na tehnikama veštačke inteligencije

Rezime: Rad predstavlja pregled pojedinih algoritama i tehnike veštačke inteligencije za odlučivanje u poteznim taktičkim igrama. Razmatraju se tehnike odlučivanja na osnovu teorije korisnosti(engl. utility AI) i minimaks pretrage (engl. minimax algorithm). Ovi algoritmi su ilustrovani kroz implementaciju jedne taktičke borbene igre, osmišljene u cilju demonstracije i poređenja ovih algoritama. Igra dolazi uz rad u vidu Unity projekta i implementirana je u jeziku C#. Prvo poglavlje posvećeno je kratkom istorijatu veštačke inteligencije u igrama i Unity okruženju za razvoj igara. Tehnika zasnovana na teoriji korisnosti je opisana u poglavlju 2, a minimaks algoritam u poglavlju 3. Pravila igre implementirane za potrebe rada predstavljena su u poglavlju 4. Poglavlje 5. će sadrži detalje implementacije izabranih tehnika, a poglavlje 6. poređenje dva pristupa. Zaključak teze je izložen u poglavlju 7.

Ključne reči: video-igre, veštačka inteligencija, agent, sistem ponašanja agenta zasnovan na korisnosti, pretraga Minimax

Sadržaj

Glava 1

Uvod

U ovom poglavlju biće dat pregled razvoja video igara, veštačke inteligencije, njihovog odnosa i zajedničke istorije, a nakon toga i kratak pregled okruženja za kreiranje igara Unity, u kom je napravljena igra kroz koju će biti ilustrovane tehnike veštačke inteligencije koje su u fokusu ovog rada.

1.1 O video-igrama

Video-igre su igre koje se igraju uz pomoć odgovarajuće tehnologije (analognih ili digitalnih računara, igračkih konzola, mobilnih telefona) sve vreme pružajući korisniku vizuelne povratne informacije. Video-igre su danas značajna grana IT i zabavne industrije. Svoje početke nalazi 1950-tih godina u načnim istraživanjima vezanim za računarstvo (igre „Tennis for Two" 1958. i „Spacewar" 1961.). Popularnost i komercijalni razvoj kreće 1970-ih i 1980-ih, sa pojavljivanjem prvih konzola i arkadnih igara. Nagli razvoj i popularnost doživljavaju i s pojavom kućnih računara. S pojavom mobilnih i pametnih telefona njihova popularnost se dodatno povecala. Razvoj industrije je sa sobom doneo razvoj raznih alata i tehnologija za kreiranje video igara.

Razvoj industrije doveo je i do pojave raznih igara raznolikih sadržaja i implementacija. Danas postoji mnoštvo žanrova igara. Kao neki od tih žanrova izdvajaju se avanture, „pucačke" igre, igre uloga (engl. Role playing games, skr. RPG) i strateške video-igre [6], kojima ćemo se baviti u ovom radu.

O taktičkim poteznim igrama

Strateške video-igre predstavljaju žanr igara koji se fokusira na taktičko razmišljanje i planiranje načina da se dođe do pobede. Akcenat je na strateškim, taktičkim i ponekad i logističkim izazovima (sakupljanje resursa, istraživanje i širenje teritorije, razvoj i nadogradnja tehnologija i upravljanje borbenim jedinicama). Strateške video-igre obično delimo na četiri pod-žanra, u zavisnosti od toga da li je igra potezna ili se igra u realnom vremenu, kao i da li je fokus startegija ili taktika. Potezne igre (engl. turn-based games) podrazumevaju da igrači naizmenično izvode poteze, pri čemu u svakom potezu agent može da izvrši jednu ili više akcija. Samo agent koji je na potezu može da vrši akcije. U igrama u realnom vremenu agenti delaju istovremeno, što često može da dovede do nepreglednog stanja igre za igrača. Zato se uvodi mogućnost pauze, kojom igrač može da pauzira igru, i dá sebi vremena da odabere željene akcije. U opštem smislu, strategija definiše dugoročne planove, a taktike predstavljaju deo strategije i to su konkretni načini kako se neki manji deo plana, kratkoročni cilj, može ostvariti. U kontekstu video igara, glavna karakteristika taktičkih igara je to što je u njima fokus na upravljanju borbenim jedinicama, dok je uloga ostalih aspekata prisutnih u strateškim igrama umanjena ili potpuno zanemarena. Taktičke i strateške igre se često razlikuju i po razmeri sukoba: u strateškim igrama igrač kao borbene jedinice kontroliše armije u toku ratne kampanje, dok u taktičkim pojedinačne likove u manjim okršajima. U ovom radu je fokus na taktičkim poteznim igrama [5].

1.2 Veštačka inteligencija u video igrama

Neformalno, može se reći da je veštačka inteligencija (skr.VI) grana računarstva koja se bavi izučavanjem i oblikovanjem metoda koje omogućavaju prividno inteligentno ponašanje računara. Inteligentna mašina je predstavljena u vidu fleksibilnog pametnog agenta koji može da sagleda svoje okruženje i preduzme akcije da bi maksimizovao šanse za ispunjavanje nekog cilja. Neformalno, pojam veštačka inteligencija se primenjuje kada mašina podražava spoznajne funkcije ljudi kao što su učenje i rešavanje problema. Veštačka inteligencija je opšti pojam koji se primenjuje na različite skupove problema i u zavisnosti od toga razlikuje se i njena implementacija i načini primene. Neke od oblasti u kojima se primenjuje veštačka inteligencija su obrada prirodnog jezika, predstavljanje znanja, metode

upravljanja, računarski vid itd.

Agenti i stanja

Kada se govori o veštačkoj inteligenciji u video-igrama dobro je opisati sledeća dva pojma.

1. *Agenti* predstavljaju entitete koji mogu da obrade informaciju o okruženju u kom se nalaze i da na osnovu tih informacija donesu odluku o svojim akcijama.

2. *Stanje* je jedinstvena konfiguracija okruženja u kojem se agent nalazi. Stanje se može promeniti kada agent (ili igrač) izvrši neku radnju tj. akciju. Za agenta, skup mogućih stanja naziva se *prostor stanja*. Ovo je važan pojam jer je osnovna ideja većine metoda veštačke inteligencije pretraživanje prostora stanja, da bi se pronašao najbolji način delovanja u skladu sa trenutnom situacijom. Karakteristike prostora stanja utiču na prirodu metoda VI koje se mogu koristiti.

Kratka istorija veštačke inteligencije i igara

Veliki deo istraživanja veštačke inteligencije u igrama se bavi pravljenjem agenata koji su sposobni da igraju igru (sa ili bez učenja). I pre nego što je veštačka inteligencija postojala kao formalno polje nauke, ljudi su pisali programe za igranje igara. Alan Tjuring i Klod Šenon su 1950. godine osmislili pristupe kojima bi računar mogao da igra šah, a prvi program koji je uspešno igrao igru napisao je Aleksandar Daglas. Taj program se zvao *OXO* i igrao je igru iks-oks. Nekoliko godina kasnije Artur Samjuel je prvi izumeo oblik mašinskog učenja koji se danas naziva učenje potkrepljivanjem, koristeći program koji je naučio da igra igru Dame protiv sebe. Pojavom video-igara deo istraživanja veštačke inteligencije se usmerio i na njih. Neki od najinteresantnijih i istorijski najznačajnijih rezultata tih istraživanja su sledeći:

1. Krajem 1970. prve igre koje imaju neki vid VI su „Space Invaders” (1978) i „Pacman” (1979).

2. Neke sportske igre iz 80-tih su imale VI koja je pokušavala da simulira tehnike slavnih menadžera i trenera. Najpoznatija takva igra je „Earl Weaver Baseball" (1987).

3. Tokom 80tih i borilačke igre dobijaju prve likove kojima upravlja VI. Među najranijim primerima takvih igara je igra „Karate Champ" (1984).

4. Prva akciona RPG igra koja je implementirala VI agente za računarski vođene likove je „First Queen" (1988).

5. „Dragon Quest IV" (1990) uvodi sistem kojima igrači mogu da menjaju neka ponašanja računarski vođenih likova.

6. Pronalaženje puta usavršeno je u igri „Warcraft" (1994).

7. Sistem senzora, princip simuliranja čula kod računarskih likova, pojavljuje se prvi put u igri Goldeneye (1997), a kasnije je unapređen u igri Thief (1998).

8. VI zasnovana na potrebama (engl. Need-based AI) [8], iz koje će se kasnije razviti sistemi za na korisnosti koja je u fokusu ovog rada, razvijena je u igri „Sims" (2000).

9. „Black and White" (2000) je jedan od najranijih primera igra koja je uspešno iskombinovala više VI pristupa. U igri postoje računarski vođeni likovi koji uče na osnovu dobijenih nagrada i kazni, što podseća na učenje potkrepljivanjem. Takođe za izbor akcija koristila se kombinacija senzora i VI zasnovane na potrebama.

Uloge veštačke inteligencije u video igarama

Jedna od glavih uloga veštačke inteligencije u video igrama je *kreiranje ponašanja agenata*, tj. da *„nauči" agente da igraju igru*. Igra treba da stvori iluziju inteligentnog razmišljanja kod računarski vođenih likova i učini interakciju sa njima što zabavnijom. Nije potrebno da VI potpuno verno simulira misaoni proces ljudi, dovoljno je da reaguje na promene koje igrač napravi u igri potezom koji neće razbiti igračev doživljaj igre. Primeri za ovo ulogu obuhvataju pronalaženje najkraćeg puta, planiranje i izbor akcija itd.

Osim ove uloge, veštačka inteligencija se koristi i za *proceduralno kreiranje sadržaja* za video-igre, a od skoro i za *analiziranje i modelovanje ponašanja igrača*

igara, što pruža dizajnerima igara bolji uvid u želje i reakcije igrača. Proceduralno kreiranje sadržaja podrazumeva mogućnost programa da na osnovu nekakvog algoritma kreira sadržaj bez dalje podrške programera. Primeri u video igrama su generisanje nivoa (npr. „Elder Scrolls: Daggerfall", 1992.), muzike („Portal 2", 2011.), raznih tipova agenata („No Man's Sky", 2016., generiše $18 \cdot 10^{18}$ planeta, gde svaka od njih ima sopstvene obrasce vremenskih prilika, strukturu, floru i faunu koji su takođe proceduralno generisani). Analiziranje i modeliranje igrača podrazumeva korišćenje raznih tehnika veštačke inteligencije zarad konstruisanja računarskih modela ponašanja, spoznaje i emocije igrača, na osnovu interakcije igrača i igre. Ono predstavlja multidisciplinarni presek polja istraživanja veštačke inteligencije, afektivnog računarstva i psihologije [7].

Neke tehnike veštačke inteligencije u video igrama

Zbog kompleksnosti video-igara poželjno je da implementacija VI bude jednostavna, tj. da se u slučaju da implementacija sadrži neke greške, te greške mogu lako pronalaziti i otklanjati. Takođe, poželjno je da VI bude modularna i omogući dizajnerima igara veliki stepen slobode i kontrole u kreiranju određenih ponašanja i menjanju postojećih. U video igrama najčešće se koriste sledeće tehnike:

1. *Ad-hoc kreiranje ponašanja* (engl. Ad-hoc behaviour authoring) je verovatno najčešći pristup za implementaciju VI u video igrama. Sam izraz „VI u video igrama" i dalje se uglavnom odnosi na ovaj pristup. Ovaj pristup podrazumeva upotrebu sistema u kojima se ručno definiše skup pravila koja će se koristiti za definisanje ponašanja agenata kojima upravlja VI. Ovde spadaju VI zasnovane na *konačnim automatima* (engl. finite state machines, skraćeno FSM), *drvetima odlučivanja* (engl. behaviour trees, skraćeno BT) i *teoriji korisnosti* (engl. Utility AI). Pregled ovih pristupa je dat u glavi 2 na strani 13, pri čemu je poseban akcenat stavljen na sisteme ponašanja zasnovanih na teoriji korisnosti.

2. Razne *tehnike pretrage* (grafova i stabala). Neinformisana (pretraga po dubini, engl. depth first search, DFS, pretraga po širini, engl. breadth first search, BFS) i informisana pretraga (pretraga prvi najbolji, engl. best first search) mogu da se koriste za problem traženja najkraćeg puta. Najbolje rezultate obično daje varijanta algoritma prvi najbolji koja se naziva $A*$. Ima primera gde se algoritam A* koristi za planiranje poteza u video-igrama. Jedan od

njih je prikazan na slici 1.1, i dolazi iz igre Mario. Crvene linije ilustruju moguće buduće putanje koje algoritam A* razmatra za Maria, uzimajući u obzir dinamičku prirodu igre.

Minimax i *Monte Karlo pretraga* se koriste u igrama od dva (ili više) igrača za izbor optimalnog poteza. Detaljni opis algoritma Minimax biće dat u glavi 3 na strani 26.

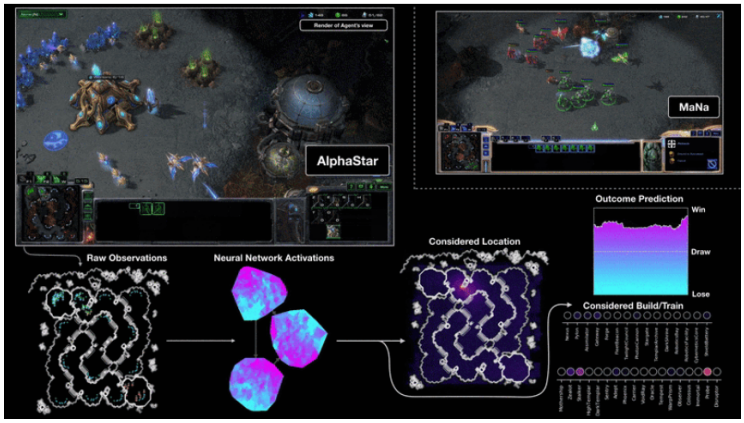
Slika 1.1: Kontroler igrača zasnovan na A* algoritmu, „Mario AI Competition" iz 2009.

3. Iako su u poslednje vreme sve popularnije u kreiranju veštačke inteligencije u video igrama, razne tehnike *mašinskog učenja* (engl. machine learning) ređe imaju primenu u kreiranju veštačke inteligencije u tradicionalnim video-igrama zbog svoje trenutne složenosti. Koriste se za kreiranje ponašanja agenata u igri, kao i za proceduralno generisanje sadržaja. Pošto pravljenje agenata koji mogu da igraju igre predstavlja svojevrsan izazov možemo da izdvojimo nekoliko značajnih dogstignuća.

   Tim naučnika „DeepMind" razvio je 2015. godine VI agenta AlphaGo za igranje tradicionalne kineske igre „Go" koji je uspešno pobedio tadašnjeg evropskog šampiona. Kasnije je agent unapređen (AlphaGo Zero), a potom i uobličen u agenta za igranje igara sa dva suprotstavljena igrača (AlphaZero).

   Još jedan primer, koji je razvila ista kompanija „DeepMind", je VI AlphaStar, koja je naučila da igra igru „Starcraft 2". AlphaStar je VI zasnovana na neuronskoj mreži, koja je trenirana na podacima iz igre, pomoću tehnika nadgledanog učenja i učenja potkrepljivanjem. Uspela je da pobedi jednog od najboljih profesionalnih igrača ove igre u decembru 2018. Rad na ovoj

VI je započet u decembru 2016. Na slici 1.2 je prikaz vizuelizacije agenta, i ekran ljudskog protivnika, koji je bio skriven od agenta.



Slika 1.2: Vizuelizacija agenta AlphaStar.

1.3 Razvojno okruženje Unity

Unity predstavlja platformu i integrisano razvojno okruženje za razvoj 2D i 3D interaktivnih multimedijalnih aplikacija i igara za računare, konzole, mobilne uređaje i veb sajtove, sa mogućnošću izvršavanja na preko 20 podržanih platformi. Unity razvija kompanija Unity Technologies od 2004. godine, sa vizijom da olakša i približi razvoj interaktivnih aplikacija što većem broju ljudi. Danas se može reći da je vizija ostvarena, s obzirom na podatak iz 2019. godine, da Unity ima preko 1,5 miliona aktivnih i oko 4,5 miliona registrovanih korisnika mesečno. Prvu verziju okruženja Unity objavila su trojica osnivača kompanije (David Helgason, Joachim Ante i Nicholas Francis) još davne 2005. godine. Jezgro sistema Unity napisano je u programskom jeziku C/C++, dok je vizuelno okruženje *Unity UI Editor* napisano u jeziku C#. Za pristup najnižem sloju, odnosno jezgru i funkcijama okruženja Unity, dostupan je API za korišćenje u radnom okviru .NET Framework, i jezicima C#, Boo, ali i JavaScript. Za pisanje programskog kôda se obično koristi integrisano okruženje Monodevelop, ali je moguće korišćenje bilo kog drugog okruženja (npr. Microsoft Visual Studio).

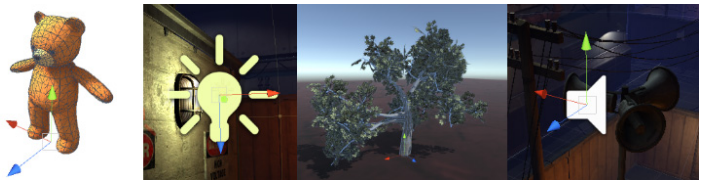
Osnovni koncepti tehnologije Unity

U ovom poglavlju, prvo će biti dat pregled osnovnih koncepata okruženja Unity, a nakon toga i pregled njihove organizacije na korisničkom interfejsu.

1. *Scena* predstavlja virtuelni prostor na kome se igra odvija. Njen prikaz je dat u prozoru scene korisničkog interfejsa (Unity UI Editor, stavka 2). Svi elementi igre se postavljaju na scenu i postavljanjem pravila kako se ovi elementi odnose međusobno se kreira svet igre. Jedna igra može imati više scena u sebi. Na primer, u igri „Super Mario" svaki nivo predstavlja drugu scenu, dok u igri „Tetris" imamo samo jednu scenu (ako izuzmemo početni ekran).

2. Sve entitete kojima korisnik raspolaže prilikom razvoja projekta i objekata igre (zvukovi, slike, teksture, materijali, animacije, modeli, ali i nadogradnje Unity okruženja u vidu ekstenzija, dodataka i skripti) možemo nazivati *resursima* (engl. asset). Oni su grupisani u odvojeni prozor interfejsa (Unity UI Editor, stavka 1).

3. Elementi na sceni se predstavljaju jednim od osnovnih koncepata okruženja Unity, *objektom igre* (engl. GameObject). Korišćenjem ovog objekata, igra se može podeliti na delove kojima se lako upravlja i koji se jednostavno povezuju i podešavaju. Na slici 1.3 dat je primer nekih objekata igre, objekat sa 3D modelom koji će verovatno predstavljati lika u igri, objekat osvetljenja, objekat sa modelom drveta i objekat za kreiranje zvuka. Primer pregleda komponenti biće dat tokom pregleda interfejsa inspektora (Unity UI Editor, stavka 5).



Slika 1.3: Primeri raznih objekata igre.

Objekti igre se sastoje od *komponenti*. Komponente su zapravo funkcionalnosti koje objekat može da ima. Svaki objekat može imati veliki broj komponenti i, dakle, isto toliko funkcionalnosti i osobina. Komponente se podešavaju *promenljivama*. To su osobine ili podešavanja koja datu komponentu kontrolišu. Podešavanjem ovih promenljivih dobijamo potpunu kontrolu nad efektima koje data komponenta ima nad objektom. Moguće je napraviti i prazan objekat igre, ali čak i on mora sadržati komponentu koja

određuje njegovu poziciju, rotaciju i veličinu na sceni kao i odnos sa drugim objektima igre (engl. transform component).

4. Nakon dodavanja objekata na scenu i komponenti koje određuju njihov izgled, poziciju i druge osobine, potrebno je ugraditi logiku, čime se određuje uloga, funkcija i ponašanje tih objekata. Ovo se postiže pisanjem *skripti* i njihovim pridruživanjem objektima u vidu komponenti.

5. Unity omogućava pamćenje kreiranog objekta igre kao gotovog resursa, sa svim komponentama, osobinama i logikom, koje ga čine onim što jeste. Ovo je koncept okruženja Unity, koji se naziva *prefabrikovan objekat* (engl. prefab), a ovakvi objekti su u potpunosti spremni za kasnije korišćenje u istom projektu (npr. kod dinamičkog kreiranja više istih objekata programabilno), ili u drugim projektima jednostavnim uvozom.

Primera radi uzmimo 2D verziju igre „Tetris". Slika jedne kocke (npr. `Kocka.png`) bi prilikom uvoza u Unity okruženje postala resurs. Prevlačenjem tog resursa na scenu dobili bismo objekat igre koji možemo da nazovemo `KockaObjIgre`. Nakon toga tom objektu možemo dodavati razne komponente. Ako planiramo da ovaj objekat iskoristimo ponovo, možemo da napravimo od njega prefabrikovani objekat, prevlačenjem sa scene ili hijerarhije u prozor projekta.

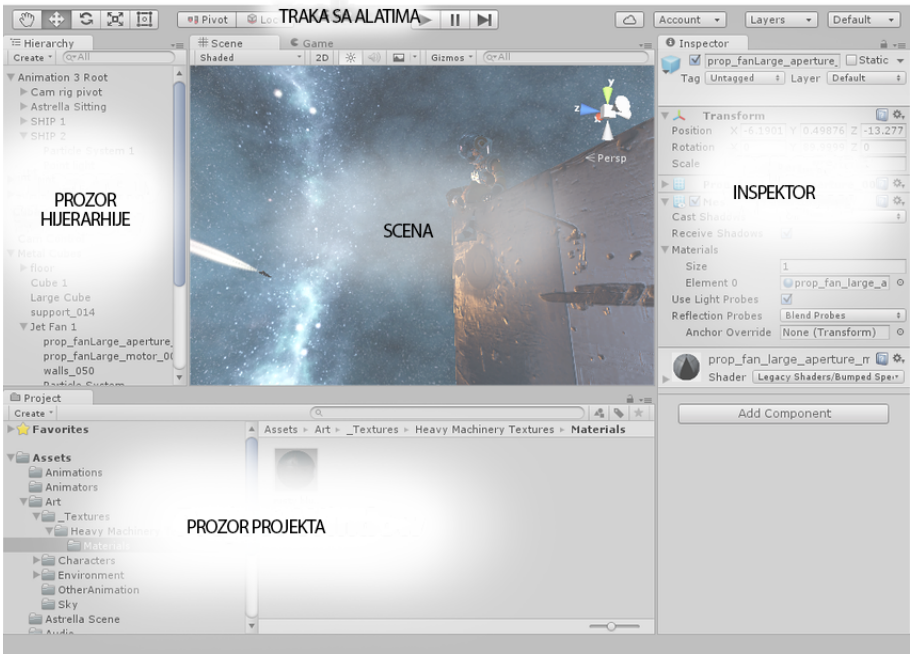
Osim navedenih, Unity pruža podršku i drugim konceptima vezanih za rad sa 2D i 3D objektima, simuliranje fizike, vektorima itd. [3]

Radno okruženje i glavni korisnički interfejs Unity okruženja je *Unity UI Editor* (izgled ovog okruženja prikazan je na slici 1.4).

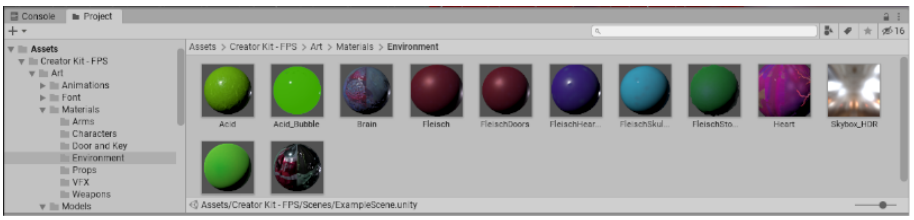
Kada se otvori u svom standardnom rasporedu okruženje *Unity UI Editor* pored trake sa podešavanjima (eng. main menu) sadrži 5 prozora:

1. *Prozor projekta* sadrži u sebi sve resurse (i prefabrikovane objekte) koji se koriste za razvoj igre. Resursi se uključuju u projekat korišćenjem uvoznih funkcionalnosti koje Unity okruženje obezbeđuje. Svi resursi moraju biti u fascikli `Assets`, a korisnik potom ima slobodu da resurse organizuje kako želi. Na slici 1.5 je dat primer kako organizacija resursa može da izgleda. Sadrži prozor sa stablom fascikli i prozor za pregledanje pojedinačne fasikle iz stabla.

2. *Prozor scene* korisniku pruža pogled na scenu i vizuelni pregled igre. U dve razdvojene kartice sadrži pregled igre u stanjima razvoja (engl. scene view)



Slika 1.4: Pregled radnog okruženja *Unity UI Editor* sa standardnim rasporedom prozora.



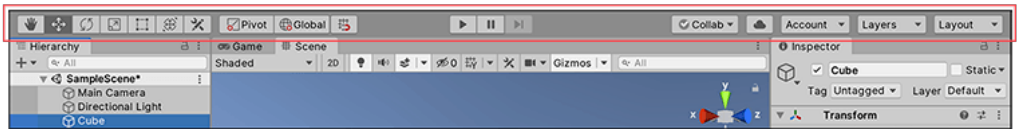
Slika 1.5: Primer prozora projekta.

i igranja (engl. game view). Dok je u stanju razvoja, korisnik prevlačenjem raznih resursa iz prozora projekta na scenu kreira objekte igre, što se potom može ispratiti u prozoru hijerarhije.

3. *Prozor hijerarhije* prikazuje sve objekte scene organizovane hijerarhijski, u obliku stabla. U korenu tog stabla nalazi se objekat scene. U početnom stanju scene, kao prvo „dete", uvek postoji i objekat igre *glavna kamera*, koja određuje pogled igrača dok igra igru. Svaki objekat igre može u sebi sadržati druge objekte igre. Svaki objekat igre na sceni će biti potomak scene ili drugog objekta igre.

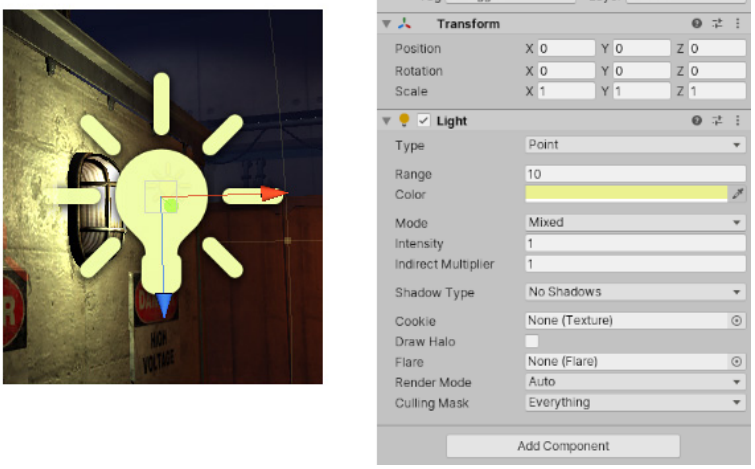
4. *Traka sa alatkama* predstavlja skup softverskih alatki za podešavanje i prikaz objekata na sceni. Alatke su podeljene na alatke za podešavanje objekata,

alatke za prikazivanje pomoćnih elemenata na sceni, alatke za kontrolu toka igre i alatke za podešavanje projekta.



Slika 1.6: Prikaz trake sa alatima.

5. *Inspektor* ili prozor za pregledanje, obezbeđuje detaljni pregled komponenti nekog objekta igre ili podešavanja nekog resursa. Na slici 1.7 je prikazan objekat osvetljenja. Ovaj objekat sadrži komponentu sa podacima o poziciji, rotaciji i veličini na sceni (engl. transform component) i specijalnu komponentu za rad sa osvetljenjem na sceni. Komponenta transformacije u prostoru sadrži promenljive kojima korisnik može da utiče na pozicioniranje objekta igre, dok komponenta osvetljenja sadrži opcije za podešavanje jačine i boje svetla, tipa senke itd.



Slika 1.7: Primer pregleda komponenti objekata igre u inspektoru.

Tagovi (engl. tags), predstavljaju način identifikovanja objekata u Unity-u. Nalaze se ispod naziva objekata igre u inspektoru. Jedan od načina identifikovanja objekta je, naravno, korišćenje imena tog objekta. U pojedinim situacijama korisno je imati načina za zajedničko identifikovanje sličnih ili istih objekata. Na primer, igra može sadržati objekte poput tenkova, aviona,

vojnika, a svi oni mogu biti pod istim tagom (na primer, *Neprijatelj*). Korišćenjem ovog taga, programskim kôdom možemo lako pristupiti i proveriti sve neprijateljske objekte.

Slojevi (engl. layers) ukazuju na neke funkcionalnosti koje su zajedničke različitim, pa i nesrodnim, objektima. Na primer, slojevi mogu da ukazuju na to koji će objekti biti iscrtani, ili koji će biti sakriveni, ili na koje će moći da se puca, ili jednostavno koji će objekti imati neku specifičnu osobinu. Grupisanje objekata u slojeve je jednostavno i vrši se odabirom opcija iz padajuće liste prikazane na prozoru inspektor.

Glava 2

Ad-hoc kreiranje ponašanja

Ovaj pristup podrazumeva upotrebu sistema u kojima programeri i dizajneri igara ručno definišu skup pravila koja će se koristiti za definisanje ponašanja VI tj. za prolazak kroz prostor stanja. Njihova primena je stoga ograničena na agente koji se suočavaju sa prilično ograničenim brojem mogućih situacija, što je čest slučaj kod računarski vođenih likova. Za razliku od algoritama za pretraživanje poput A*, ad-hoc metode se možda ne uklapaju u klasičnu definiciju VI, ali ih industrija video igara smatra metodama VI od svog početka.

2.1 Sistem za kreiranje ponašanje agenta zasnovan na konačnim automatima

Konačni automati i njihove varijante (na primer, hijerarhijski konačni automati) bili su najčešći pristup za kreiranje ponašanja agenata do sredine prve decenije 21. veka. Sistemi sa konačnim automatima su široko rasprostranjeni zbog relativno lake implementacije, ali povećavanjem složenosti automata, održavanje može da postane problem.

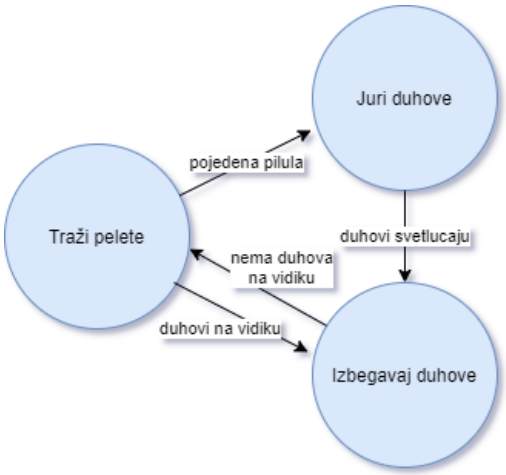
U igrama koje koriste konačne automate, ponašanje agenta tj. zadatak koji agent trenutno izvršava je određeno trenutnim stanjem u kom se nalazi konačni automat. Zadatak određuje niz konkretnih akcija koje agent izvršava, sve dok ne dođe do promene stanja automata. Konačni automat definisan je sa tri glavne komponente:

1. određenim brojem stanja, od kojih svako sadrži informaciju o trenutnom zadatku koji agent izvršava,

2. skupom akcija koji agent izvršava dok je u nekom stanju,

3. prelazima između stanja koji su opisani uslovom koji treba da se ispuni da bi se prešlo u drugo stanje.

Agenti u igri kreću od početnog stanja i zatim razmatraju događaje i pravila koji će izazvati prelazak u neko drugo stanje. U svakom trenutku mogu da se nalaze samo u jednom stanju.



Slika 2.1: Primer konačnog automata za igru Pacman.

Na slici 2.1 vidimo prikaz konačnog automata agenta u igri Pacman. Čvorovi sadrže zadatke *„Traži pelete”*, *„Izbegni duhove”*, *„Juri duhove”*, a prelazi između čvorova pokazuju uslove pod kojima se dešava prelaz u odgovarajuće stanje. Na primer agent je u stanju *„Traži pelete”*, nailazi na „pilulu” i nakon toga prelazi u stanje *„Juri duhove”*. Nakon nekog vremena duhovi kreću da trepere što znači da je efekat „pilule” prošao i agent prestaje da ih juri i beži od njih, tj. ulazi u stanje *„Izbegni duhove”* [7].

2.2 Sistem za ponašanje agenta zasnovan na stablima ponašanja

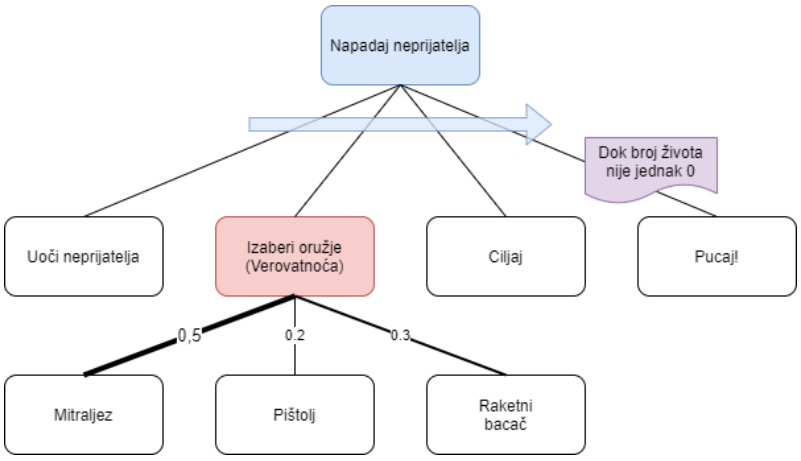
Stablo ponašanja je sastavljeno od povezanih čvorova (zadataka, ponašanja) u kojima se donose odluke o akcijama koje agent izvršava. Za svaku odluku, počevši od korena proverava se određen skup uslova. Razlikujemo tri tipa čvora:

1. *Niz zadataka* je čvor koji za direktne potomke ima druge, manje zadatke poređane u određeni redosled. Ukoliko se jedan od tih zadataka uspešno završi, prelazi se na sledeći zadatak. Niz je uspešan, ako su svi njegovi zadaci uspešno završeni.

2. *Selektor zadataka.* Čvorovi ovog tipa se dele na dva podtipa: *selektore sa verovatnoćom* i *selektore sa prioritetom.* U prvom slučaju svaki čvor potomak ima unapred definisanu verovatnoću sa kojom može biti izabran. Ukoliko izabran zadatak ne bude uspešno završen, neće biti uspešan ni selektor zadatak. U drugom slučaju svi potomci imaju dodeljen prioritet na osnovu kog su sortirani. Ukoliko ne uspe odabrani zadatak, prelazi se na sledeći po prioritetu. Ako se neki zadatak uspešno završi, uspešno se završava i selektor.

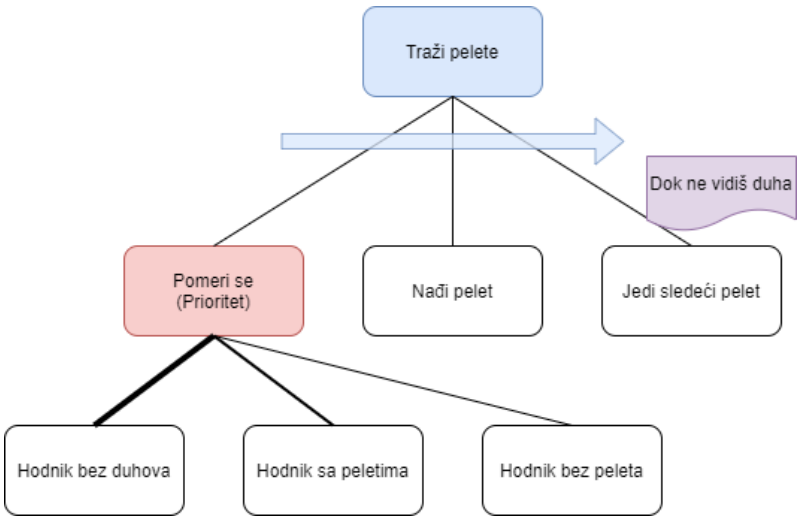
3. *Dekorator.* Ovi čvorovi doprinose usložnjavanju zadataka potomaka. Stvaraju dodatne uslove koji se trebaju ispuniti da bi se zadatak izvršio (na primer, određuju vreme za koje zadatak mora da se izvrši ili koliko puta se neki zadatak mora ponoviti da bi bio uspešan).

Ovaj pristup je fleksibilniji od pristupa sa konačnima automatima. U stablima ponašanja uvek se koristi isti skup odluka i može da se dođe do bilo koje akcije u stablu. Kod konačnih automata razmatraju se samo prelazi u trenutnom stanju, pa ne može da se dođe do svake akcije.



Slika 2.2: Primeri stabla ponašanja. Plavi čvorovi predstavljaju niske ponašanja, crveni selektore verovatnoće, a ljubičasti dekoratore.

Na slici 2.1, prikazano je jedno stablo ponašanja. Početni čvor *„Napadni neprijatelja"* je niz zadataka, što znači da će se svi potomci tj. podzadaci izvoditi u zadatom redosledu. Agent će prvo probati da *„Uoči neprijatelja"*. Ako to uspe, nastavlja sa daljim izvršavanjem zadataka i prelazi na zadatak *„Izaberi oružje"*, koji je selektor sa verovatnoćom, i nasumično uzima jedno od tri oružja. Nakon toga agent vrši zadatak *„Ciljaj"* i ukoliko je taj zadatak uspešno izvršen, prelazi na zadatak *„Pucaj"* koji je ukrašen dekoratorom i u ovom primeru označava da agent neće prestati da puca dok „ne ubije" protivnika.



Slika 2.3: Primer stabla ponašanja. Plavi čvorovi predstavljaju niske ponašanja, crveni selektore prioriteta, a ljubičasti dekoratore.

U drugom primeru 2.3 prikazano je stablo ponašanja *„Traži pelete"* za igru Pacman, sa čvorom *„Pomeri se u"*, koji je selektor sa prioritetom.

2.3 Sistem za ponašanje agenta zasnovan na teoriji korisnosti

Zadnjih decenija izdvaja se još jedan pristup, koji je započeo kao *veštačka inteligencija zasnovana na potrebama* (engl. need-based AI), koju je za potrebe igre „The Sims" osmislio Ričard Evans. U toj igri, trenutna „potreba" agenta za nečim (odmor, hrana, društvena aktivnost) je kombinovana sa vrednošću objekta ili akcije koju bi tu potrebu zadovoljio. Kombinacijom ovih vrednosti dobija se rezultat na osnovu koga agent zna koju akciju treba da preduzme. Ovaj pristup

je nastavio da se razvija i kasnije je uobličen kao *sistem zasnovan na teoriji korisnosti* (engl. utility-based system). Uopšteno govereći, u ovom sistemu se svakoj mogućoj akciji dodeljuje vrednost, nakon čega se izabere jedna ili nekoliko akcija sa najvišim vrednostima. Ovaj jednostavan pristup iskoristio je Dave Mark za pravljenje sistema *IAUS* (engl. Infinite Axis Utility System), čiji će pregled biti izložen u ovom poglavlju.

Pojam korisnosti

Teorija korisnosti kao koncept postoji još pre video igara i računara, i koristi se u teoriji igara, ekonomiji i brojnim drugim poljima. Glavna ideja iza ove teorije je da se svaka moguća akcija ili stanje u datom modelu, može opisati jednom vrednošću, koja predstavlja korisnost date akcije u odnosu na relevantni kontekst. Pritom, korisnost nije fiksna vrednost, već ona nastaje razmatranjem raznih faktora, meri koliko nam je nešto potrebno, i menja se iz trenutka u trenutak u zavisnosti od trenutnog konteksta. Na primer, korisnost flastera je relativno mala dok nemamo povrede, međutim u slučaju neke posekotine, ona bi se naglo povećala.

Kada računamo vrednosti korisnosti, važno je da budemo konzistentni. Pošto se ove vrednosti porede da bi se došlo do konačne odluke, važno je da korisnosti svih objekata u celom sistemu budu izražene na istoj skali. Vrednosti korisnosti se često mogu kombinovati u novu vrednost i zato se, najčešće, koriste normalizovane vrednosti (vrednosti u granicama između 0 i 1, međutim, naravno, vrednosti mogu biti i u nekim drugim fiksnim granicama).

Princip maksimalne očekivane korisnosti

Ključ za donošenje odluka kada se koristi ovaj pristup je da se izračuna korisnost svake akcije koju agent može da izvrši, a zatim izabere ona sa najvećom korisnošću. Naravno, igre su u najvećem broju nedeterminističke tako da računanje korisnosti ponekad nije moguće. Teško je znati koja akcija je poželjnija ako se ne može odrediti rezultat te akcije. Na primer, kada bismo mogli da izračunamo drvo igre partije šaha, ocenjivanje poteza ne bi bilo potrebno, jednostavno bismo izabrali put koji nas vodi do pobede, ali tu mogućnost ne možemo da imamo. Sistem ponašanja agenta zasnovan na korisnosti može doneti „najbolje odluke", ukoliko je način ocenjivanja korisnosti dovoljno dobar. Najčešća tehnika je inspirisana pojmom matematičkog očekivanja slučajne promenljive i podrazumeva da

pomnožimo vrednosti korisnosti sa verovatnoćom svakog ishoda akcije, i te proizvode sumiramo. Ovo nazivamo *očekivana korist* date akcije i obeležavamo sa EU (engl. expected utility).

$$EU = \sum_{i=1}^{n} D_i P_i \tag{2.1}$$

U ovoj jednakosti, D_i predstavlja korisnost ishoda i, a P_i je verovatnoća da će se taj ishod desiti. Za datu akciju, suma verovatnoća svih ishoda je normalizovana i jednaka 1. Ovaj proces se primenjuje na svaku moguću akciju, da bi se na kraju izabrala ona sa najvećom očekivanom korisnošću. Ovoj je princip maksimalne očekivane korisnosti. Na primer zamislimo situaciju da u igri postoje dva moguća ishoda za akciju napada. Neka agent ima 85% šanse da udari metu napada, i neka uspešni napad ima korisnost $0,6$, očekivana korisnost za tu akciju bi bila $0,85 \cdot 0,6 = 0,51$. Ukoliko bismo pretpostavili da postoji akcija napada drugim oružjem, sa verovatnoćom 60% ali sa korisnošću $0,90$, njena očekivana korisnost bi bila $0,6 \cdot 0,9 = 0,54$. Uprkos manjoj verovatnoći bila bi izabrana druga akcija jer je očekivana korisnost veća.

Princip očekivane korisnosti sličan je principu očekivane vrednosti:

$$EV = \sum_{i=1}^{n} V_i P_i \tag{2.2}$$

samo ovde V_i predstavlja vrednost ishoda i i zapravo ga je definisao švajcarski matematičar Danijel Bernuli da bi objasnio *paradoks Sankt Petersburg*. Paradoks je bio vezan za princip očekivane vrednosti, tj. pretpostavku da će prilikom donošenja odluka ljudi želeti da maksimizuju očekivanu vrednost i postavio ga je Bernulijev rođak, Nikolas Bernuli 1713. godine. On je osmislio hipotetičku igru, tj. zanimalo ga je koliko bi novca ljudi platili da igraju igru koja ima sledeća dva pravila: (1) novčić se baca dok ne padne na "pismo" i (2) igrač dobija 2$ ako pismo padne pri prvom bacanju, 4$ na drugom, 8$ na trećem itd. Većina ljudi je spremna da plati najviše nekoliko dolara. Paradoks se ogleda u tome što je očekivana vrednost igre (prosečna isplata koju bi mogla da se očekuje ako bi se igra igrala nebrojeno mnogo puta) beskonačna:

$$EV = 1/2 \cdot 2 + 1/4 \cdot 4 + 1/8 \cdot 8 + 1/16 \cdot 16 + ... = 1 + 1 + ... = \infty \tag{2.3}$$

Danijel Bernuli je dao rešenje problema 25 godina kasnije: ljudi ne maksimiziraju očekivanu (monetarnu) vrednost, već očekivanu korisnost [1].

Faktori odlučivanja

Retko kad će se donošenje odluke zasnivati na proceni samo jedne informacije, tj. samo jednog faktora odlučivanja. Primera radi, posmatrajmo naručivanje proizvoda putem interneta. Prilikom odlučivanja o tome koji ćemo proizvod naručiti, razmatramo obično cenu proizvoda, ali i mogućnost i trajanje dostave, ocene korisnika, kao i razne druge faktore. Svaki od ovih faktora će uticati na odluku da li da poručimo neki proizvod ili ne. Ti faktori mogu biti modifikovani, tj. može im biti dodeljena težina (korisnost), koja određuje koliko utiču na konačnu odluku. Jedan način da se ovo postigne je da se izračuna očekivana korisnost iz jednačine 2.1 za svaki od faktora. Pod pretpostavkom da su vrednosti normirane, možemo izračunati njihovu prosečnu vrednost da bismo došli do konačne korisnosti date akcije (naručivanja konkretnog proizvoda). Ovo nam omogućava da definišemo odluku kao neku kombinaciju njenih faktora. Svaki faktor je izolovan, i za svaku odluku možemo kombinovati koliko faktora želimo.

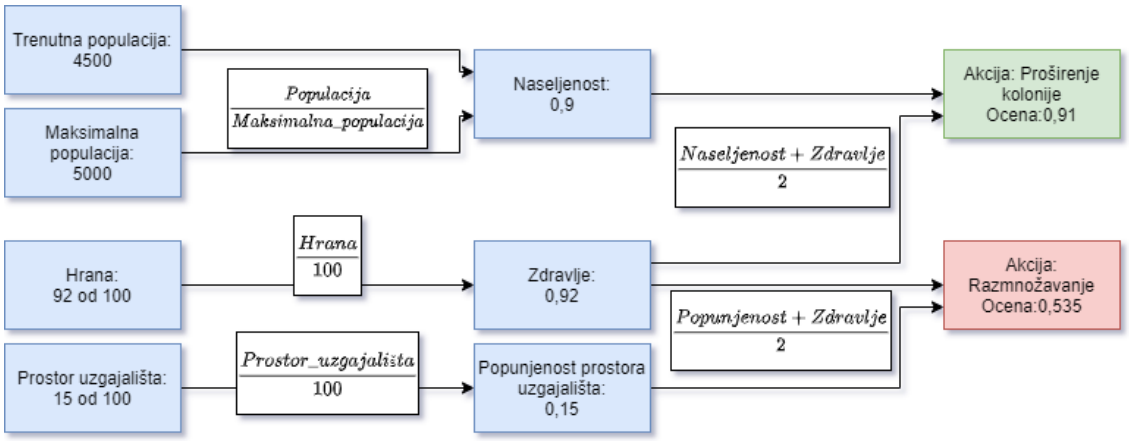
Na slici 2.4 je dat primer simulacije kolonije mrava. VI treba da odluči da li kolonija treba da se: (a) proširi na još teritorije ili da se (b) namnoži tj. da uveća broj jedinki. Za ove akcija razmatramo tri faktora:

1. Naseljenost — ukoliko ima previše mrava u koloniji, treba zauzeti još teritorije.

2. Zdravlje kolonije — je zasnovano na stanju tj. količini skladištene hrane kolonije. Zdravlje je lošije (manje) ukoliko ima manje hrane.

3. Prostor uzgajališta — jaja mrava moraju da budu čuvana na specifičnoj temperaturi i zato postoji poseban prostor za uzgajalište. Da bi kolonija mogla da se množi, potrebno je da ima dovoljno ovog prostora.

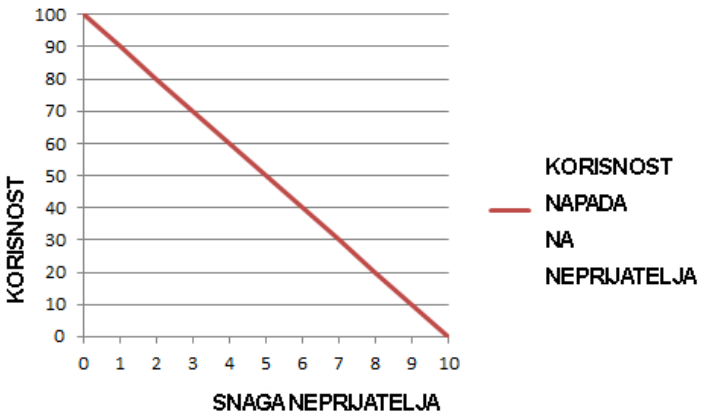
Sva tri navedena faktora se mogu oceniti vrednostima od 0 do 1. Vrednovanje akcije proširivanja zavisi od vrednosti faktora naseljenosti i zdravlja kolonije, a vrednost akcije razmnožavanja od vrednosti zdravlja i prostora uzgajališta. Naseljenost zavisi od trenutne populacije i maksimalne populacije. Zdravlje predstavlja procenat dostupne hrane (količina hrane se meri u rasponu od 0 do 100, procenti popunjenosti kapaciteta za hranu). Prostor uzgajališta predstavlja koliki je procenat uzgajališta trenutno popunjen.



Slika 2.4: Primer računanja korisnosti i izbora akcija na osnovu razmatranja korisnosti raznih faktora.

Pronalaženje odgovarajuće funkcije korisnosti

Korisnost neke akcije možemo izračunati ili ako znamo njene ishode ili kombinovanjem raznih faktora odlučivanja. Sledeći korak je da proizvoljnu vrednost iz igre pretvorimo u vrednost korisnosti. Proces dodeljivanja vrednosti korisnosti faktorima odlučivanja je jako subjektivan, jer dve različite osobe mogu isti faktor različito oceniti. Važno je razumeti vezu između ulaznog podatka i njegove korisnosti. Zato je potrebno za ulazni podatak pronaći odgovarajuću krivu tzv. *funkciju korisnosti*. Definisanje ove funkcije tako da dobro modeluje procenu nekog podatka za donošenje odluke je glavni zadatak ovog pristupa. Primeri nekih funkcija korisnosti dati su na slikama 2.5, 2.6 i 2.7.



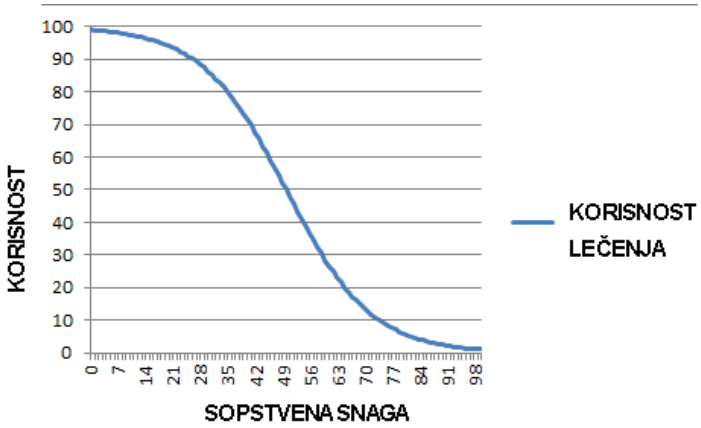
Slika 2.5: Funkcija korisnosti napada.

Na slici 2.5 vidimo primer funkcije korisnosti za akciju *„napad protivnika”*, za

faktor *„snaga protivnika”*. Eksplicitni oblik ove funkcije je:

$$y = 100 - 10 \cdot x \tag{2.4}$$

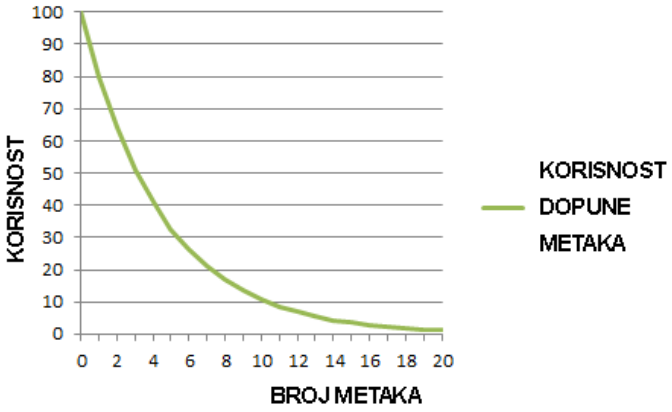
„Snaga protivnika” se meri na skali od 0 do 10, s tim da je protivnik sve slabiji što je bliži 0, a sve jači što je bliži vrednosti 10. Agent će na osnovu procene ovog faktora doneti ocenu ove akcije. Na ovom primeru ocena akcije će biti veća što je protivnik slabiji.



Slika 2.6: Funkcija korisnosti lečenja.

Na slici 2.6 vidimo primer funkcije korisnosti za akciju *„lečenja”*. Ovde se procenjuje faktor *„sopstvena snaga”* za agenta koji razmatra akciju.

$$y = 50 \cdot (1 + \cos(\frac{x}{100} \cdot \pi)) \tag{2.5}$$



Slika 2.7: Funkcija dopune municije.

Na slici 2.7 je dat primer procene akcije *„dopune municije"* a faktor odlučivanja je *„broj metaka"*. Primećujemo da vrednost akcije naglo raste nakon što ostanemo bez polovine metaka (tj. 10 u primeru).

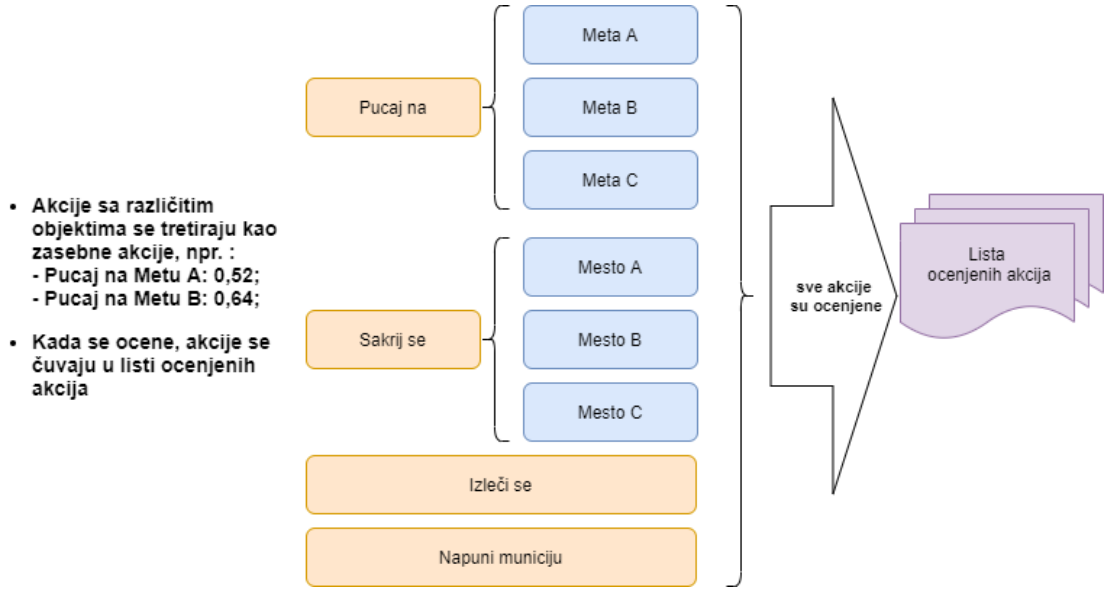
Ovo su proizvoljni primeri, i svaki od njih je vrednovao akciju na osnovu jednog faktora. Na primer, mogli smo akcije *„napada protivnika"* i *„dopune municije"* kombinovati sa procenom faktora *„udaljenost protivnika"*.

IAUS

Sistem *Infinite axis utility system, IAUS* razvio je Dave Mark, za potrebe igara „Guild Wars 2" i „Red Dead Redemption". Sistem je pokušaj standardizacije sistema ponašanja zasnovanih na korisnosti, kojim se samo daje predlog arhitekture, ali ne i sama implementacija koda. Neki od problema koje je autor pokušao da reši ovim sistemom su:

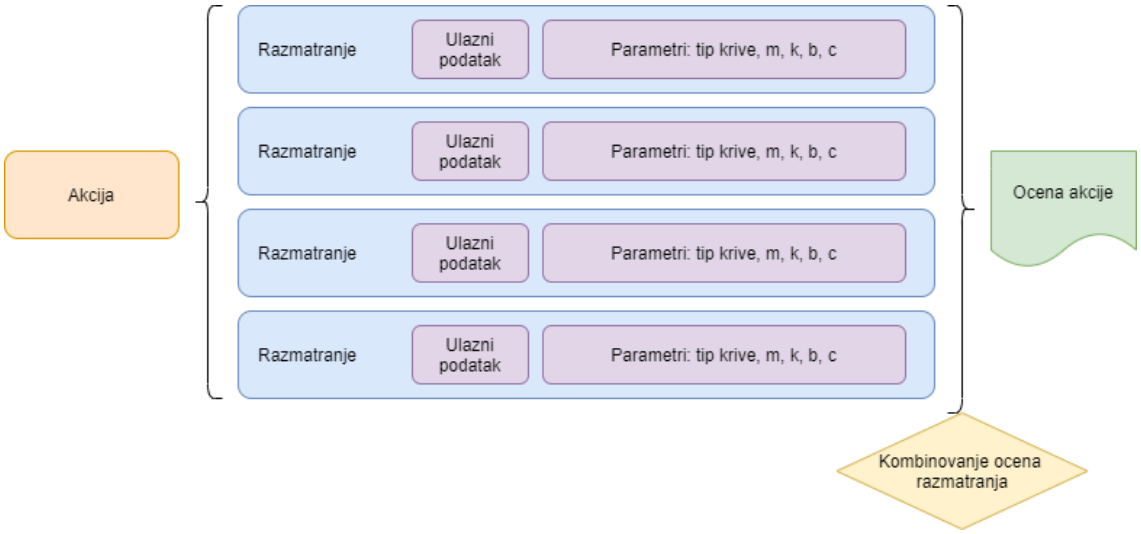
1. standardizovanje ulaznih podataka,
2. definisanje funkcija korisnosti,
3. modularnost sistema, tj laka mogućnost menjanja VI,
4. dostupnost alata za menjanje ponašanja dizajnerima igara.

Slike u ovoj sekciji rada su preuzete sa predavanja koje je autor održao na ovu temu [2]. U ovom sistemu Dave Mark, definiše *akcije* (engl. actions), od kojih se svaka sastoji od proizvoljnog broja *razmatranja* (engl. considerations). Svako od tih razmatranja bi znalo da za dati ulazni kontekst vrati odgovarajuću vrednost. Vrednost akcije bi se računala kao kombinacija vrednosti raznih razmatranja. Svaka akcija ima svoj *kontekst* (engl. context), npr. agenta koji tu akciju razmatra i objekat koji je „meta" te akcije ili neki drugi podatak iz stanja igre, poput vremena koje je prošlo, broj neprijatelja itd. Svaka akcija bi se ocenjivala u zavisnosti od svog konteksta. Na slici 2.8 je dat primer kako lista akcija nekog agenta može da izgleda. U ovom primeru, agent bi drugačije ocenio napad (akcija *„Pucaj na"*) za tri različita neprijatelja (A, B i C). Akcija napada je razdvojena po metama zato što će se neki njihovi podaci koristiti u razmatranjima te akcije, npr. preostali broj života, udaljenost od agenta itd. Akcije lečenja i dopune municije (engl. heal i reload) će uzimati vrednosti sa agenta, pa nisu raslojene po meti.



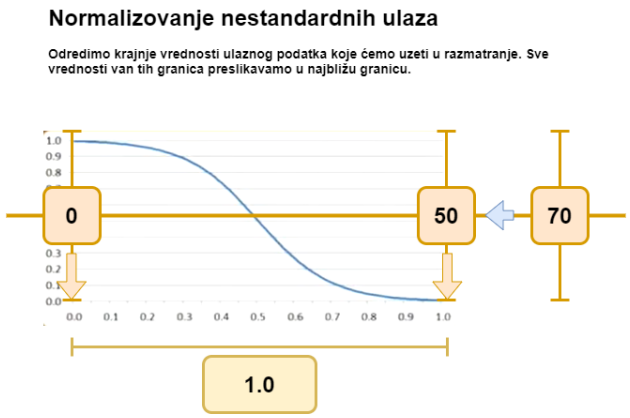
Slika 2.8: Primer akcija koje agent dobija na razmatranje.

Na slici 2.9 vidimo prikaz akcije kao *liste razmatranja*. Svako razmatranje bi sadržalo normalizovani ulazni podatak, tip i parametre funkcije. Za svako razmatranje neki podatak iz konteksta bi se normalizovao, a zatim uzimao kao *ulazna vrednost*. Predlog je da se od tih podataka formira *enumeracija tipa ulaznih podataka*. Onda bi se formirala metoda koja za vrednosti te enumeracije vraća odgovarajući ulazni podatak iz konteksta. Ova metoda se naziva *sortirnica* (engl. clearing house), i u njoj bi se ujedno radila normalizacija podataka. Zatim bi se od tipa i parametara funkcije dobila odgovarajuća vrednost korisnosti. U ovom sistemu se standardizuje nekoliko tipova funkcija koje se parametrizuju tako da jednostavnom promenom parametra moguće dobiti veliki broj različitih grafika, i time ubrzati proces pravljenja prototipa ponašanja.



Slika 2.9: Prikaz razmatranja jedne akcije.

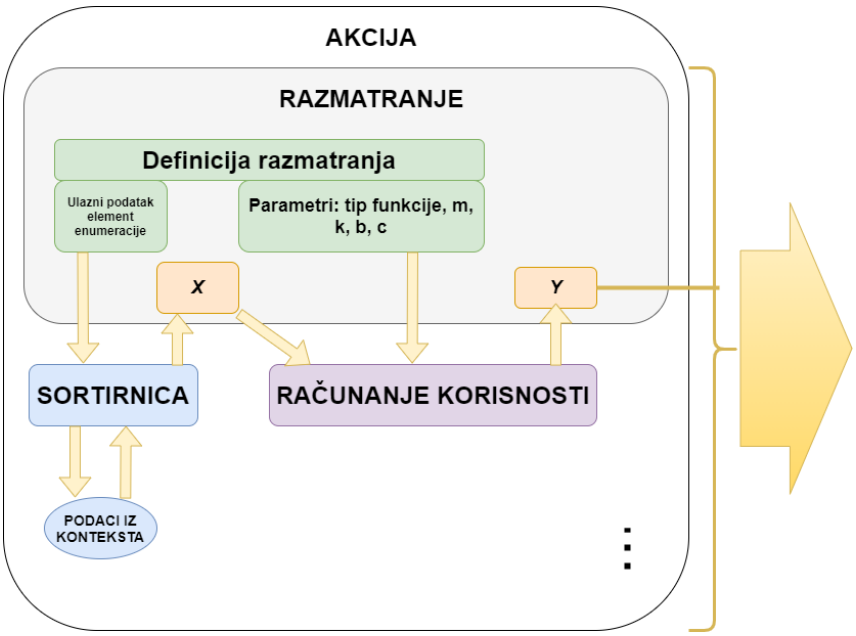
Na slici 2.10 vidimo primer kako možemo normalizovati ulazni podatak udaljenosti od protivnika. Na primeru je dat proizvoljni grafik funkcije korisnosti za faktor udaljenosti. Ideja je da se napravi preslikavanje udaljenosti na raspon od 0 do 1 i da se odrede krajnje vrednosti ulaznog podatka nakon kojih će se sve vrednosti tretirati isto. U primeru sve udaljenosti veće od 50m (ili koja god jedinica mere za koju smo se odlučili) se ne razmatraju, tj. biće tretirane kao maksimalna udaljenost od 50m.



Slika 2.10: Prikaz normalizovanja udaljenosti

Na slici 2.11 možemo videti kako izgleda proces procene jednog razmataranja na nekoj akciji. Za jedno razmatranje šalje se jedan ulazni podatak iz konteksta metodi sortirnici, koja zatim vraća prilagođenu vrednost. Ova vrednost se za-

tim uvrštava u promenjivu funkcije korisnosti, i time se dobija izlazna vrednost razmatranja. Ovaj proces se ponavlja za svako razmatranje akcije.



Slika 2.11: Prikaz toka razmatranja prilikom ocenjivanja akcije.

Ovako definisan sistem bi bilo relativno lako implementirati i dodatno, napraviti dizajnerske alate za kreiranje novih akcija i razmatranja.

Glava 3

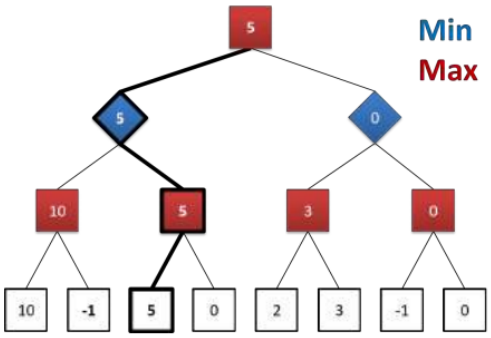
Algoritam Minimax

Algoritmi pretrage stabla se u video igrama najčešće koriste za probleme pronalaženja puta (pretraga po dubini, pretraga po širini, A^*) ili za pretrage i generisanje stabla igre (Minimaks i Monte Karlo pretraga). Monte Karlo pretraga se pokazala kao uspešniji algoritam od Minimaksa i uspešno se koristila u igrama Go, Šah, Dame, ali i u igrama sa nepotpunim informacijama poput igara Podmornica, Poker i Bridž, kao i u nedeterminističkim igrama Monopolu i Tavli. U ovom radu ćemo se detaljnije pozabaviti algoritmom Minimax sa α - β odsecanjem.

3.1 Osnovni Minimax algoritam

Jedan od osnovnih algoritama pretrage, koji se može koristiti za izbor akcija, u igrama u kojima imamo dva suprotstavljena igrača, gde akcije jednog igrača utiču na akcije drugog, jeste *algoritam Minimaks*. Kao jedno od ponuđenih rešenja za problem automatizacije igranja šaha, ovaj algoritam se pojavio se u članku Kloda Šenona 1950. godine. Uspešno je iskorišćen za simuliranje igrača u igrama poput dama i šaha. Algoritam naizmenično simulira poteze dva suprotstavljena igrača. Za svakog igrača svi mogući potezi su istraženi (simulirani), za svako od rezultujućih stanja svi mogući potezi drugog igrača se takođe simuliraju, i taj proces se u teoriji ponavlja dok se ne dođe do stanja u kom se igra završava. Ovako se generiše čitavo stablo igre od korena do listova. U listovima stabla poziva se funkcija evaluacije, koja procenjuje koliko je dobro to stanje za početnog igrača. Zatim algoritam se vraća od listova ka korenu, i na svakom nivou određuje koju bi akciju igrač odigrao optimalno, na osnovu vraćenog rezultata. Ovo znači da će igrač koji je bio na potezu u korenu stabla gledati da izabere akciju sa najboljom ocenom,

dok će protivnik gledati da odigra poteze koje bi minimizovali ocenu. Ovaj proces naizmeničnog maksimizovanja i minimizovanja se naziva *minimaksizacija*. Za igre sa velikim brojem stanja, najefikasnije je ograničiti pretragu na relativno malu dubinu. Kada se dođe do izabrane dubine procenjuje se stanje pomoću funkcije evaluacije. Ovako se ne dobija najbolji potez, ali je aproksimacija dovoljno dobra. Ovo ograničenje dubine dovodi do javljanja efekta horizonta, tj moguće je da će na nekoj od sledećih dubina odabrani potez dovesti do lošijeg ishoda za igrača.



Slika 3.1: Primer stabla igre i osnovnog Minimax algoritma

Pseudokôd

Ulaz: f - funkcija evaluacije čvora, *cvor* - čvor za koji se radi minimaksizacija, *dubina* - dubina do koje se stablo pretražuje, $igraMax$ - argument koji nam govori koji igrač je na potezu.

Izlaz: vrednost najboljeg poteza

Inicijalni poziv: $MINIMAX(f, koren, dubina, TRUE)$

Algorithm 1 pseudokod Minimax algoritma

```
function MINIMAX(f, cvor, dubina, igraMax)
    if (dubina = 0 ili cvor je terminalni cvor) then
        return f(cvor)
    end if
    if (igraMax) then
        najboljaVrednost := −∞
        for each dete cvora do
            pomVrednost := MINIMAX(f, dete, dubina − 1, FALSE)
            najboljaVrednost := MAX(najboljaVrednost, pomVrednost)
        end for
        return najboljaVrednost
    else
        najboljaVrednost := +∞
        for each dete cvora do
            pomVrednost := MINIMAX(f, dete, dubina − 1, TRUE)
            najboljaVrednost := MIN(najboljaVrednost, pomVrednost)
        end for
        return najboljaVrednost
    end if
end function
```

3.2 Minimax algoritam sa α - β odsecanjem

Jedan od glavnih problema Minimax pretrage je broj stanja koji eksponencijalno zavisi od dubine stabla. Veliki broj stanja je moguće eliminisati iz razmatranja *metodom odsecanja*. U praksi se obično razmatra efikasan algoritam koji je zasnovan na osnovnom Minimax algoritmu i heuristikama α i β, pomoću kojih se vrši odsecanje.

1. α predstavlja trenutno najbolji izbor (najvišu vrednost) na koji smo do sada naišli u toku pretrage za igrača koji maksimizuje.

2. β predstavlja trenutno najbolji izbor (najmanju vrednost) na koji smo do sada naišli u toku pretrage za igrača koji minimizuje.

Osnovni postupak ocenjivanja čvorova je minimaks tipa: funkcijom evaluacije ocenjuju se samo čvorovi na nekoj odabranoj dubini, a zatim se rekurzivnim postupkom (minimaksizacijom) ocenjuju čvorovi prethodnici. Postupak α-odsecanja biće opisan pretpostavljajući da funkcija evaluacije za igrača A koji je na potezu

ima pozitivan smisao (bolje su veće ocene). Postupak β-odsecanja je analogan tome. Neka je ocenjeno $n-1$ njegovih legalnih poteza, neka su dobijene ocene $a_1, a_2, ..., a_{n-1}$ i neka je a_k najveća od njih. Razmatramo n-ti legalni potez. Neka je v čvor u koji taj potez vodi i neka je a_n njegova ocena koja tek treba da bude određena kao minimum ocena dece čvora v. Nakon tog poteza, protivnik (igrač B) ima više mogućnosti i traži se ona sa najmanjom ocenom. Za ocenu b_i važi da je ona veća ili jednaka zajedničkom minimumu a_n i onda važi $a_n \leq b_i$. Ako naiđemo na ocenu b_j takvu da važi $b_j \leq a_k$ možemo da zaključimo da je vrednost $a_n \leq a_k$, a pošto se u prethodnom koraku bira najveća vrednost, možemo prekinuti pretragu u čvoru v.

Pseudokod

Ulaz: f - funkcija evaluacije čvora, *cvor* - čvor za koji se radi minimaksizacija, *dubina* - dubina do koje se stablo pretražuje, α,β heuristike, $igraMax$ - argument koji nam govori koji igrač je na potezu.

Izlaz: vrednost najboljeg poteza

Inicijalni poziv: $AlfaBeta(f, koren, dubina, -\infty, +\infty, TRUE)$

Algorithm 2 pseudokod Minimax algoritma sa α - β odsecanjem

```
function AlfaBeta(f, cvor, dubina, α, β, igraMax)
    if (dubina = 0 ili cvor je terminalni cvor) then
        return f(cvor)
    end if
    if (igraMax) then
        pomVrednost := −∞
        for each dete cvora do
            pomVrednost := max(pomVrednost,
                    AlfaBeta(f, dete, dubina − 1, α, β, FALSE))
            α := max(α, pomVrednost)
            if (β ≤ pomVrednost ) then
                break
            end if
        end for
        return pomVrednost
    else
        pomVrednost := +∞
        for each dete cvora do
            pomVrednost := min(pomVrednost,
                    AlfaBeta(f, dete, dubina − 1, α, β, FALSE))
            β := min(β, pomVrednost)
            if (pomVrednost ≤ α ) then
                break
            end if
        end for
        return pomVrednost
    end if
end function
```

Očuvava se osobina Minimaks algoritma da se pronalazi najbolji potez za zadatu dubinu. Ukoliko se u svakom čvoru potezi ispituju od najboljeg ka najgorem, biće najviše odsecanja i samim tim ubrzanja algoritma. Ovaj deo je preuzet mahom iz knjige [4].

Glava 4

Pravila igre

Za potrebe rada implementirana je igra potezne borbe. Pravila su sledeća: dva suprotstavljena igrača igraju igru kontrolišući borbene jedinice, u seriji poteza na tabli šestougaonih (heksagonalnih) polja, dimenzije 10×10. Igrače možemo razlikovati po njihovoj boji, vidljivoj na UI elementima igrača koji ih kontrolišu (crveni i plavi). Svakoj jedinici je pridružen atribut koji se naziva *inicijativa* i redosled igranja borbenih jedinica se utvrđuje na osnovu vrednosti njihove inicijative (na početku igre se sortiraju po toj vrednosti). U toku svog poteza jedinica može da odigra jednu akciju: ili da napadne protivničku jedinicu ili se pomeri na neko drugo, dostižno, polje. Igra je gotova kada jedan igrač izgubi sve svoje jedinice, i pobednik je onaj igrač koji ima još preostalih jedinica.

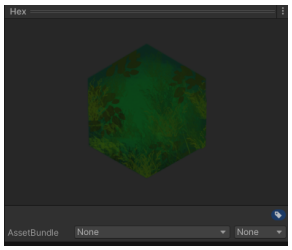
Entiteti u igri su *polja* i *objekti koje se nalaze na poljima*. Na slici 4.1 je prikazano početno stanje igre.

4.1 Polje

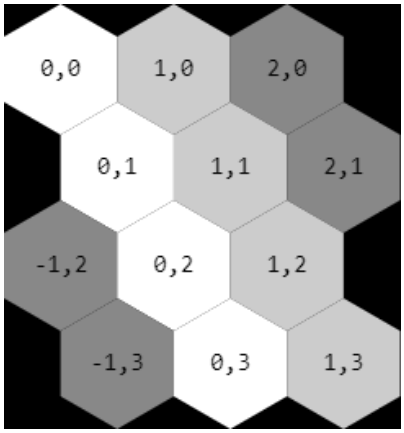
Polja su objekti koji poređani u mrežu čine tablu, prostor po kojem se kreću ostali objekti i koji predstavlja bojište. Polja su šestougaonog oblika, u orijentaciji, poređana u neortogonalnom koordinatnom sistemu (sa uglom od 60° između osa), sa početnim poljem u gornjem levom uglu. Na slici 4.2 dat je prikaz objekta polja.

Svako polje sadrži informacije o tome da li je prohodno ili je zauzeto od strane neke jedinice, sadrži uzvišenje ili neku prepreku. Da bi se jedinica pomerila sa jednog polja na drugo potrebno je da pređe put sastavljen od povezanih prohodnih polja. Primer puta je dat na slici 4.4.

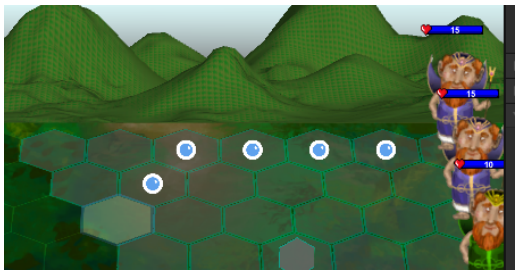
Slika 4.1: Stanje igre nakon incijalizacije, početna scena, pruža pregled svih objekata



Slika 4.2: Objekat polja od kog je sačinjena tabla.



Slika 4.3: Raspored polja na tabli.



Slika 4.4: Generisana putanja jedinice.

4.2 Objekti na poljima

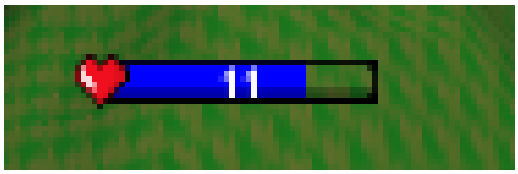
Tokom igre se na poljima nalaze različiti objekti. Među ovim objektima razlikujemo: *borbene jedinice*, *prepreke* i *bonus polja*.

Borbene jedinice

Borbene jedinice su objekti koje igrač kontroliše. One sadrže informacije o broju života, šteti koju nanose, tipu napada, inicijativi, tipu kretanja, dosegu napada i kretanja. Akcije koje borbena jedinica može da preduzme su ili napad ili kretanje. Nakon sto izvrši svoju akciju, potez povlači jedinica sa sledećom najvećom inicijativom. Na početku svakog poteza na osnovu tipa kretanja i dosega kretanja, računa se lista susednih polja po kojima jedinica može da se kreće. Jedinica može da se pomeri na svako prohodno polje u svom dosegu ukoliko do njega postoji postoji put. Neke jedinice mogu da ignorišu prepreke tokom kretanja i da stignu na bilo koje slobodno polje u svom dosegu (to zavisi od njihovog tipa kretanja). Prikaz kretanja dat je na slici 4.5.

Slika 4.5: Primer kretanja jedince po generisanoj putanji.

Svaka jedinica ima početni broj života. Izgled elementa korisničkog interfejsa koji predstavlja broj života dat je na slici 4.6. Broj se smanjuje kada jedinicu ošteti protivnička jedinica. Kada se broj života spusti na 0 jedinica biva uništena i više ne učestvuje u borbi. Šteta je broj koji jedinica svojim napadom oduzima od broja života mete napada.



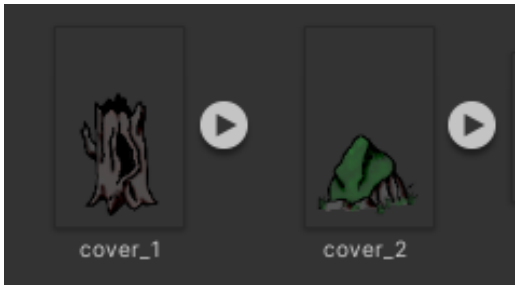
Slika 4.6: UI element za broj života jedinice.

Tip napada određuje način na koji jedinica nanosi štetu protivničkoj jedinici. Definisani su *napad iz blizine* i *napad sa veće udaljenosti*. Jedinica koja napada iz blizine može napasti protivničku jedinicu koja se nalazi na ili susednom polju ili na polju susednom bilo kom dostižnom polju (u tom slučaju prelazi put do mete). Prilikom napada jedinica nanosi štetu protivničkoj jedinici, ali i prima štetu od nje (u visini od polovine vrednosti štete te jedinice). Prikaz jedinica dat je na slici 4.7.

Slika 4.7: Tipovi borbenih jedinica.

Jedinica koja napada sa veće udaljenosti obično nanosi manju štetu, ali ne mora da pređe put do mete, niti trpi štetu. Izuzetak je situacija kada se protivnička jedinica nalazi u susednom polju i tada jedinica koja napada iz daljine takođe trpi štetu.

Prepreke



Slika 4.8: Primer zaklona.

Prepreke su objekti koji otežavaju kretanje po mapi. *Zaklon* je polje koje je neprohodno, a borbenim jedinicama može da pruži zaštitu od napada sa veće udaljenosti. Sadrži vrednost zaštite zaklona, koja se primenjuje na jedinicu u blizini ukoliko je zaklon između napadača i mete napada. Polje označeno kao *opasnost* je prohodno (ne sprečava kretanje), ali ukoliko put prolazi kroz njega može imati loše posledice po jedinicu. Na slici 4.8 prikazani su zakloni.

Bonus polja

Bonus polja pružaju nekakav dobitak jedinicama koje se nalaze na njima. *Uzvišenje* je bonus polje koje poboljšava domet i jačinu napada jedinica koje mogu da napadaju sa veće udaljenosti. Ova polja su prohodna (ne utiču na kretanje jedinica) i grafički se prikazuju kao da su na uzvišenju. Na slici 4.9 dati su primeri uzvišenja i opasnosti.

Slika 4.9: Primer uzvišenja i prepreke, opasnosti.

Glava 5

Implementacija

U ovom poglavlju biće prikazana implementacija igre napravljene za potrebe rada čija su pravila data u prethodnom poglavlju 4. Biće dat pregled klasa koje čine arhitekturu igre i pregled nekih segmenata koda. Igra je implementirana u okruženju Unity, za programski jezik izabran je C#. Pregled će biti razdvojen na pregled implementacija inteligentnih agenata i kraći pregled ostatka koda.

5.1 Pregled koda igre

Kôd demo igre možemo podeliti po njegovoj funkcionalnosti u nekoliko blokova klasa:

1. Klase kontrole toka igre,
2. klase objekata igre,
3. klase za pronalaženje puta,
4. klase agenata veštačke inteligencije.

Kontrola toka igre

Klase koje kontrolišu tok igre su klase *BattlefieldManager* i *ActionManager*.

Klasa *BattlefieldManager* sadrži informacije o veličini pojedinačnog polja, dimenzijama table polja i trenutnom stanju igre u objektu *CurrentStateOfGame*, kao i metode za inicijalizaciju table igre (i svih objekata na njoj) i metode za

selekciju polja i kreiranje puta između dva polja. Primer kôda metode *Start* klase *BattlefieldManager* u kojoj se inicijalizuju objekti na tabli, dat je u sledećem listingu.

```
void Start()
{
  //Creating new stateOfGame object...
    StatesOfGame = new List<StateOfGame>();
    StatesOfGame.Add(new StateOfGame());
    CurrentStateOfGame = StatesOfGame[0];
    //...initializing hexes...
    SetHexSize();
    CalculateInitialPosition();
    CreateGrid();
    //...setup game objects, units and special hex fields...
    SetupSpecialHexes();
    SetupUnits();
    //...initialization of ai agent...
    CreateAIAgent();
    //...setting combat start flag to true once all objects have been setup...
    CombatStarted = true;

  //...setup first playing player and initialize first unit's turn
    CurrentStateOfGame.currentPlayerId = 0;
    var unitBehaviour =
        CurrentStateOfGame.InstantiatedUnits[0].GetComponent<UnitBehaviour>()
    ActionManager.Instance.StartCurrentlyPlayingUnitTurn(unitBehaviour);
}
```

U *Start* metodi najpre se kreira prazan objekat stanja igre, koji se popunjava u metodama koje slede. Metoda *SetHexSize* čuva veličinu polja, koja se kasnije koristi u izračunavanju pozicije početnog polja table u metodi *CalculateInitialPosition*. *CreateGrid* kreira tablu polja i smešta ih u rečnik *CurrentStateOfGame.Board*. Metode *SetupSpecialHexes* i *SetupUnits* kreiraju redom, prepreke, bonus polja i borbene jedinice, ređaju ih na polja table i čuvaju ih u listama *CurrentStateOfGame.InstantiatedBattlefieldObjects* i *CurrentStateOfGame.InstantiatedUnits*. Borbene jedinice imaju definisan atribut *initiative* kojim se određuje redosled poteza jedinice. Borbene jedinice su predstavljene klasom *UnitBehaviour* koja je opisana u segmentu 5.1. Krerianje agenta se odigrava u *CreateAIAgent* metodi. Da bi se obezbedilo naizmenično igranje igrača, u objektu stanja čuva se informacija o identifikatoru igrača koji trenutno igra. Na kraju *Start* metode selektuje se prva borbena jedinica iz liste inicijalizovanih jedinica i otpočinje se njen potez.

Glavni zadatak klase *ActionManager* je kontrolisanje redosleda poteza borbenih jedinica u toku igre, kao i prepuštanje kontrole agentima ili igraču. U nastavku

je dat primer metode koja vrši prepuštanje poteza.

```
public void EndCurrentPlayingUnitTurn()
    {
        //don't know if this is necessary
        BattlefieldManager.ManagerInstance.StartingHexBehaviorTile.
    ChangeHexVisualToDeselected();

        if (BattlefieldManager.ManagerInstance.DestinationTile!=null)
        {
            BattlefieldManager.ManagerInstance.DestinationTile.
    ChangeHexVisualToDeselected();
        }

        //...reseting starting and destination tiles in order to destroy path...
        BattlefieldManager.ManagerInstance.StartingHexBehaviorTile = null;

        BattlefieldManager.ManagerInstance.DestinationTile = null;

        //...this destroys path when the unit reaches destination...
        BattlefieldManager.ManagerInstance.GenerateAndShowPath();

        CurrentlySelectedPlayingUnit.CurrentState = UnitState.Idle;

        //detarget targeted unit
        if (TargetedUnit != null)
        {
            TargetedUnit = null;
        }

        //...selects next playing unit...
        SelectNextPlayingUnit();
        StartCurrentlyPlayingUnitTurn();
    }
```

U metodi *EndCurrentPlayingUnitTurn* najpre se briše prethodno generisana putanja(menjanjem vizuelizacije polja, brisanjem izgenerisanih objekta puta i krajnih polja puta). Zatim se borbena jedinica postavlja u stanje „Idle", u kom su sve jedinice koje nisu trenutno na potezu. Stanja se koriste za kontrolu animacija borbenih jedinica i realizaciju kretanja jedinica. Nakon toga, ukoliko je neka jedinica bila napadnuta deselektuje se kao meta. Metoda *SelectNextPlayingUnit* iz liste inicijalizovanih jedinica bira sledeću jedinicu na potezu i postavlja je kao vrednost polja *CurrentlySelectedPlayingUnit*. Nakon toga *StartCurrentlyPlayingUnitTurn* priprema potez jedinice, tj. selektuje polja table po kojima je moguće kretanje i prepušta kontrolu igraču ili računaru.

Objekti igre

Objekte igre su predstavljeni sledećim klasama: polje je predstavljeno klasom *HexBehaviour*, borbena jedinica sa *UnitBehaviour*, a prepreka i bonus polje klasom *BattlefieldSpecialHex*.

HexBehaviour sadrži informaciju o objektu koji se nalazi na njoj (objektima koji implementiraju interfejs *IISOnHexGrid*, a to su borbene jedinice, prepreke i bonus polja), kao i o objektu klase *HexTile* koji se koristi kao čvor prilikom pretraživanja najboljeg puta. Takođe, sadrži i metode koje osluškuju događaje prelaska i klika miša, kao i metode za promenu vizuelizacije.

UnitBehaviour određuje logiku borbenih jedinica. Sadrže informacije o atributima borbenih jedinica (poput broja života, jačine i tipa napada, daljine i tipa kretanja), kao i metode za kontrolu ponašanja u različitim stanjima borbene jedinice. Kretanje i napad se realizuju pomoću sistema komponeneti. *UnitBehaviour* sadrži vrednost enumeracije *AttackType* i *MovementType*, na osnovu kojih se koristi određena klasa komponenta(klase koje implementiraju, redom interfejse *IAttackComponent* i *IMovementComponent*). Takođe sadrži pomoćne metode za povezivanje sa klasama komponenti korisničkog interfejsa.

BattlefieldSpecialHex sadrži logiku za implementiranje bonus polja i prepreka. Jedino bonus polje je uzvišenje, i ukoliko je na njemu jedinica koja napada iz daljine dobija bonus na jačinu napada. Prepreke su opasnost i zaklon. Zaklon ima vrednost *cover* koja se koristi u izračunavanju štete sa daljine. Ukoliko je polje sa zaklonom između napadača i mete, šteta napada će biti pomnožena sa vrednošću $1 - cover$. Opasnost se za sada koristi samo pri izračunavanju puta, tj. ukoliko je moguće, jedinice će izbegavati putanje koje sadrže polja na kojima je opasnost.

Pronalaženje puta

Polja po kojima će se vršiti kretanje poređana su mrežu šestougaonih polja. Koristi se koordinatni sistem gde su ose pod uglom od 60°, čiji je početak u gornjem levom uglu, početno polje ima koordinate (0, 0) (kao što je prikazano na slici **??**). Svaki čvor mreže predstavljen objektom bazne klase *GridObject*, koja sadrži informacije o koordinatama datog objekta. Nju nasleđuje klasa *HexTile*, koja pored informacija o poziciji u mreži, sadrži i podatke o tome da li je polje trenutno zauzeto od strane borbene jedinice, da li polje sadrži neku prepreku ili bonus, da li je u dometu kretanja borbene jedinice itd.

```
public class HexTile : GridObject, IHasNeighbours<HexTile>
{
    public bool Passable;
    public bool IsInRange;
    public bool Occupied;
    public bool Hazadours;
    public bool Cover;
    public bool HighGround;

    public HexTile(int x, int y)
      : base(x, y)
    {
        Passable = true;
        Occupied = false;
        Hazadours = false;
        Cover = false;
        HighGround = false;

    }
    ...
}
```

Klasa *Hex tile* takođe sadrži logiku za pronalaženje susednih polja: listu pomeraja za susedna polja, metodu *FindNeighbours* koja za dato polje pronalazi sva susedna na izgenerisanoj mreži, kao i razne upite za njih.

```
public class HexTile : GridObject, IHasNeighbours<HexTile>
{
    ...
    public IEnumerable<HexTile> AllNeighbours;

    public IEnumerable<HexTile> InRangeNeighbours =>
             AllNeighbours.Where(o => o.Passable && o.IsInRange);
    public IEnumerable<HexTile> ReachableNeighbours =>
             AllNeighbours.Where(o => o.Passable && o.IsInRange && !o.Occupied);

    // fills each tile with data for its neighbours
    public void FindNeighbours(Dictionary<Point, HexBehaviour> board, Vector2 boardSize)
    {
        List<HexTile> neighbours = new List<HexTile>();

        foreach (Point point in NeighbourShift)
        {
            int neighbourX = X + point.X;
            int neighbourY = Y + point.Y;

            int xOffset = neighbourY / 2;

            if (neighbourX >= 0 - xOffset && neighbourX < boardSize.x - xOffset &&
                neighbourY >= 0 && neighbourY < boardSize.y)
            {
```

```
                Point p = new Point(neighbourX, neighbourY);
                neighbours.Add(board[p].OwningTile);
            }

        }

        AllNeighbours = neighbours;
    }
    ...
}
```

Putanja je implementirana pomoću jednostruko povezane liste, klase *Path* koja sadrži informacije o poslednjem čvoru(polju) putanje, pređenom putu pre poslednjeg čvora, kao i ceni putanje. Takođe sadrži i metodu za dodavanje novog čvora u putanju.

```
public class Path<Node> : IEnumerable<Node>
{
    public Node LastStep { get; private set; }
    public Path<Node> PreviousSteps { get; private set; }
    public double TotalCost { get; private set; }

    public Path(Node lastNode, Path<Node> previousSteps, double totalCost)
    {
        LastStep = lastNode;
        PreviousSteps = previousSteps;
        TotalCost = totalCost;
    }

    public Path(Node start) : this(start, null, 0)
    {
    }

  //... method for adding new node, and it's cost, to the path
    public Path<Node> AddStep(Node step, double stepCost)
    {
        return new Path<Node>(step, this, TotalCost + stepCost);
    }

    public IEnumerator<Node> GetEnumerator()
    {
        for (var p = this; p != null; p = p.PreviousSteps)
           yield return p.LastStep;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

Za pronalaženje puta se koristio algoritam A*. Implementacija je data u klasi *Pathfinder* u metodi *FindPath*. Razdvojeni su slučajevi kada je ciljno polje zauzeto i kada je slobodno. Kada igrač kontroliše jedinicu, klikom miša na slobodno polje ono postaje ciljno i otpočinje kretanje jedinice ka njemu. Moguće je kliknuti na zauzeto polje ako je na njemu neprijateljska borbena jedinica. Ako je zauzeto, koristi se varijanta algoritma DFS pretrage, tj. prvo se proverava da li su polazno i ciljno polje susedi i, ako jesu, vraća put koji se sastoji samo od startnog čvora, inače traži se najkraći put od starta do najbližeg suseda ciljnog polja.

```
public static Path<HexTile> FindPath(HexTile start, HexTile destination)
{
  // destination is set to start if it was not reachable originally,
  // and if it's occupied, we return the shortest length path
  if (destination != start && destination.Occupied)
  {
        Path<HexTile> min = null, currPath;

        // if the unit is adjacent to the enemy unit, no path...
        if (destination.AllNeighbours.Contains(start))
             min = new Path<HexTile>(start);
    //...otherwise return the shortest path to the one of the neighbouring hexes...
        else
        {
            foreach (HexTile reachableNeighbour in destination.ReachableNeighbours)
            {
              //...find path to neighbouring hex...
                currPath = FindPath(start, reachableNeighbour);

        //...check if it's shorter than current minimal path
                if (min == null || min.Count() >= currPath.Count())
                    min = currPath;
            }
        }
        return min;
    }
    else
        ...
```

Ukoliko ciljno polje nije zauzeto, koristi se implementacija A*. Ovde se koristi red sa prioritetom. Elemente reda čine putevi, a prioritet je cena puta tj, većeg prioriteta je onaj put koji ima manju cenu. Dakle iz reda se prvo izbacuje put sa najmanjom cenom, proverava se da li je poslednje polje u skupu zatvorenih

polja, ako nije proverava se da li je poslednje polje ciljno polje. Ako jeste vraća se taj put, inače poslednje polje se označava kao zatvoreno, i za svako susedno polje kreira se novi put. Ukoliko neko od tih polja sadrži opasnost, cena tog puta se dodatno uvećava, sa konstantom 1.5. Funkcija procene je ovde implementirana kao konstanta 1. Algoritam je prikazan u kôdu koji sledi.

```
const double HAZARD_PRICE = 1.5;
...
public static Path<HexTile> FindPath(HexTile start, HexTile destination)
{

    // destination is set to start if it was not reachable originally,
    // and if it's occupied, we return the shortest length path
    if (destination != start && destination.Occupied)
    {
       ...
    }
    else
    {
      //initialize a set of closed hex tiles...
        HashSet<HexTile> closed = new HashSet<HexTile>();

    //...create priority queue, and set starting path...
        var queue = new PriorityQueue<double, Path<HexTile>>();
        queue.Enqueue(0, new Path<HexTile>(start));

    //...starting from the current cheapest path, remove it from queue, check if the
    path contains destination hex tile or  if is last step closed...
        while (!queue.IsEmpty)
        {
           Path<HexTile> path = queue.Dequeue();

           if (closed.Contains(path.LastStep))
               continue;

           if (path.LastStep.Equals(destination))
               return path;

       //...If no close current last step, and add new paths to queue...
           closed.Add(path.LastStep);

           foreach (HexTile tile in path.LastStep.ReachableNeighbours)
           {
               double d = Distance(path.LastStep, tile);
               //... new path is old path + neighbour...
               Path<HexTile> newPath =
                   path.AddStep(tile, d + (tile.Hazadours ? HAZARD_PRICE : 0));

               queue.Enqueue(newPath.TotalCost + Estimate(tile, destination), newPath);
           }
        }
```

```
        return null;
    }
}
```

5.2 Implementacija inteligentnog agenta

Sva logika kojom se implementiraju agenti veštačke inteligencije nalazi se u klasi *AIAgent*. Postoji samo jedna istanca ove klase. U sebi sadrži informaciju o borbenoj jedinici čiji potez odlučuje u promenjivoj *CurrentlyControledUnit* tipa *UnitBehaviour*.

Klasa *ActionManager* na početku poteza borbene jedinice, u metodi *SetupCurrentlyOwningUnit*, nakon što uradi selekciju polja po kojima trenutno selektovana jedinica može da se kreće, proverava da li je na potezu igrač ili računar. Ukoliko je kontroliše računar, promenljiva *CurrentlyControledUnit* se postavlja na borbenu jedinicu koja je na potezu, i poziva metodu *ChooseAction* koja će izabrati i odigrati najbolje ocenjen potez (akciju).

```
public void SetupCurrentlyOwningUnit(UnitBehaviour ub)
{
    // ResetTilesInRange resets tiles which were in movement range of previous unit...
    BattlefieldManager.ManagerInstance.ResetTilesInRange();
    BattlefieldManager.ManagerInstance.StartingHexBehaviorTile = ub.CurrentHexTile;
    //...selects movable tiles for the currently playing unit...
    BattlefieldManager.ManagerInstance.SelectTilesInRangeSimple(ub.movementRange);

  //...checks if it's player 1 turn, if yes gives control to agent
    if (ub.PlayerId == 1)
    {
         AIAgent.AIAgentInstanceAgent.CurrentlyControledUnit = ub;
         AIAgent.AIAgentInstanceAgent.ChooseAction();
    }
}
```

Metoda *ChooseAction* će prvo očistiti listu ocenjenih akcija *ScoredActions*, koje su preostale od prethodnog upravljanja borbenom jedinicom. Nakon toga proverava se tip agenta veštačke inteligencije, *AIType*. Ovom proverom se određuje koji algoritam veštačke inteligencije će se koristiti dalje za realizovanje odlučivanja agenta. Ovde se može implementirati i projektni obrazac strategije ne bi li se izbeglo grananje po nabrojivom tipu.

```
public enum AgentType
{
    UtilityAI,
    MinimaxAI
}
```

U narednim sekcijama, biće dat prikaz implementacija dva sistema ponašanja agenta. Prvi je zasnovan na korisnosti, a drugi je njegovo proširenje Minimaks algoritmom sa α - β odsecanjem. U oba sistema, *akcije* će biti iste. Akcije će zapravo biti instance posebnih klasa akcija. Ovo što je zajedničko za ove klase je da nasleđuju interfejs *IAction*. Sledi pregled interfejsa *IAction*.

```
public interface IAction
{
    void DoAction();
    void Print();
    float GetScore();
    UnitBehaviour ActionOwner { get; set; }
    HexBehaviour ChosenTargetHex { get; set; }
    List<IConsideration> Considerations { get; }
    ActionType ActionType { get; }
    float ScoredValue { get; }
    ...
}
```

Trenutno, u igri su implementirane dve klase akcija: *AttackUnitOnHexAction* i *MoveToHexAction*. *Kontekst* akcija je vezan za polja (instance HexBehaviour klase), tj. akcije će biti *„napadni neprijatelja na polju”* i *„pomeri se na polje”*. Vezanost akcije za polje vidi se u pozivu konstruktora akcija, npr:

```
public AttackUnitOnHexAction(UnitBehaviour Owner, HexBehaviour hex)
{
  ActionOwner = Owner;
  ChosenTargetHex = hex;
}
```

Implementacija sistema ponašanja agenta zasnovanog na korisnosti

Ukoliko je izabrani tip agenta bio *AgentType.UtilityAI*, agent će koristiti pristup sistema ponašanja zasnovanog na korisnosti. Ovaj pristup podrazumeva generi-

sanje svih mogućih akcija agenta u trenutnom stanju igre. Svaka akcija će biti ocenjena kombinovanjem ocena njenih *razmatranja*.

```
public void ChooseAction()
{
  //clear all previously scored actions...
    if (ScoredActions.Count > 0)
       ScoredActions.Clear();

  //... if agent is implemented using utility AI...
    if (AIType == AgentType.UtilityAI)
    {
       //...score all possible actions...
       ScoreActions();
       //...order them desc by score and get the first...
       IAction chosenAction =
          ScoredActions.OrderByDescending(x => x.ScoredValue).First();
        //... do the action...
       chosenAction.DoAction();
    }
    else if (AIType == AgentType.MinimaxAI)
    {
       ...
    }
}
```

Poziva se metoda *ScoreActions* koja generiše sve moguće akcije za borbenu jedinicu u trenutnom stanju (*GetAvailableActions*) i smešta ih u listu *ScoredActions*. Nakon što se popuni lista ocenjenih akcija one se (u metodi *ChooseAction*) sortiraju po dodeljenoj oceni korisnosti u opadajućem poretku i izvršava se akcija sa najboljom ocenom, pozivanjem metode *DoAction* koja je implementirana na samoj akciji.

Prilikom generisanja svih akcija, u metodi *GetAvailableActions*, razdvojeno je generisanje akcija po tipu (*Attack* ili *Move*).

```
public enum ActionType
{
    Attack,
    Move
}
```

Metoda *GetMoveActions* nalazi listu polja do kojih agent može doći u jednom potezu i za svako od njih kreira akciju kretanja. Sledi pregled generisanja akcija napada u metodi *GetAttackActions*.

```
private List<IAction> GetAttackActions()
{
  List<IAction> attackActions = new List<IAction>();
  //get all enemies which can be attacked and create attack action on their hex tile...
  List<HexBehaviour> HexesOccupiedbyEnemy =
     BattlefieldManager.ManagerInstance.GetAllEnemiesInRange();
  foreach (HexBehaviour hex in HexesOccupiedbyEnemy)
  {
    AttackUnitOnHexAction iterator =
       new AttackUnitOnHexAction(CurrentlyControledUnit,hex);
    attackActions.Add(iterator);
  }

  return attackActions;
}
```

Prvo se poziva metoda *GetAllEnemiesInRange*, koja pripada klasi *BattlefieldManager*, i ona vraća listu polja koja sadrže neprijateljske borbene jedinice koje je moguće napasti u ovom potezu. Zatim se prolaskom kroz listu tih polja generišu akcije napada za dato polje i borbenu jedinicu koja napada.

Ocena akcije se čuva u polju *ScoredValue* i računa se pomoću metode *GetScore*. U njoj se prolazi kroz listu svih razmatranja, datih u polju *Considerations*, njihove ocene se množe međusobno, i to čini ocenu date akcije.

```
public float GetScore()
{
   float score = 1;
   //we don't have to multiply the scores of consideration, but this is convenient if
    all values are between 0 and 1
   foreach (IConsideration consideration in Considerations)
      score *= consideration.Score();

   return score;
}
```

Polje *Considerations* dobija listu razmatranja pomoću metode *GetConsiderations*. Ukoliko bi se dodavalo novo razmatranje akciji, moralo bi da se uključi u ovu metodu. U primeru koji sledi data je ova metoda za akciju napada, i u njoj su data tri razmatranja: *ConsiderEnemyHealth_Con* koje razmatra trenutni broj života neprijateljske jedinice, *TargetGetsKilled_Con* koje razmatra da li bi napad uništio neprijateljsku jedinicu, i ako jeste, daje bonus akciji i *SelfGetsKilledByRetaliati-*

on koja razmtara da li će jedinica koju kontroliše agent biti uništena povratnom šetotm.

```
//this method will be specific to each action
private List<IConsideration> GetConsiderations()
{
    List<IConsideration> considerations = new List<IConsideration>();
    considerations.Add(new ConsiderEnemyHealth_Con(ActionOwner, ChosenTargetHex));
    considerations.Add(new TargetGetsKilled_Con(ActionOwner, ChosenTargetHex));
    considerations.Add(new SelfGetsKilledByRetaliation(ActionOwner, ChosenTargetHex));
    return considerations;
}
```

Sva razmatranja sadrže informacije o kontekstu, i mogu da sadrže tip funkcije korisnosti kao i parametre te funkcije. Kontekst podrazumeva informacije o borbenoj jedinici koja je na potezu, tipu ulazne vrednosti za funkciju (enumeracija *ConsiderationInputType* koja kasnije metodi sortirnici govori koji podatak iz konteksta treba da se obradjuje) i polje koje je meta akcije (ili ciljno polje kretanja ili polje koje sadrži neprijatelja). Sledi pregled nabrojivog tipa *ConsiderationInputType*:

```
public enum ConsiderationInputType
{
    TargetHealth,
    SelfHealth,
    NearestEnemyDistance,
    NearestAllyDistance,
    EnemyTargetDistance,
    TargetEnemyRetaliationStrike,
    CoverValue
}
```

Razmatranja su klase koje nasleđuju klasu *ConsiderationBase*. U ovoj klasi se nalaze metode: *GetInputFromContext*, *sortirnica*, koja na osnovu tipa ulazne vrednosti vraća ulaznu vrednost sa konteksta; i *Score* koja računa vrednost razmatranja za tu izračunatu ulaznu vrednost. Sledi pregled metoda *GetInputFromContext* i *Score* koje se nalaze u baznom razmatranju, tj. klasi *ConsiderationBase*. Ovde bi se mogao ukloniti polimorfizam, tj. klasa *ConsiderationBase*.

```
const float MINIMIZE_SCORE = -0.00001f;
public static float GetInputFromContext(ConsiderationInputType c, UnitBehaviour Owner,
                                        HexBehaviour targetHexBehaviour)
{
    float inputValue;
```

```
    switch (c)
    {
      //0.5 is a placeholder, a temporary value until proper value is determined
        case ConsiderationInputType.CoverValue:
        inputValue = 0.5f;
        break;
        case ConsiderationInputType.EnemyTargetDistance:
        inputValue = Vector3.Distance(Owner.transform.position,
                                      targetHexBehaviour.UnitAnchorWorldPositionVector);
        break;
        ...
        case ConsiderationInputType.SelfHealth:
        UnitBehaviour ub = Owner;
        inputValue = (float) ub.CurrentHealth / (float) ub.MaxHealth;
        break;
        ...
        case ConsiderationInputType.TargetEnemyRetaliationStrike:
        UnitBehaviour enemyUB1 = (UnitBehaviour)targetHexBehaviour.ObjectOnHex;
        inputValue = (float)enemyUB1.Damage/2;
        break;

        case ConsiderationInputType.TargetHealth:
        UnitBehaviour enemyUB = (UnitBehaviour) targetHexBehaviour.ObjectOnHex;
        inputValue = enemyUB.CurrentHealth;
        break;
        default:
        inputValue = MINIMIZE_SCORE;
        break;
    }

    return inputValue;
}
```

Trenutno nisu definisane sve vrednosti za tip ulaznog podatka. Zato je uvedena konstanta MINIMIZE_SCORE, koja bi trebalo da umanji ocenu akcije u tom slučaju dovoljno da se ona sigurno neće izabrati.

Metoda *Score* na osnovu vrednosti promenljive *GraphType*, i vrednosti 4 parametara K, M, C, B računa vrednost razmatranja za ulaznu vrednost koja je izračunata u metodi *GetInputFromContext*. Uzete su ovakve parametrizacije zato što je lako ispratiti koju jedan parametar ima na grafik funkcije. Na primer za *GraphType.Linear*: K je stepen funkcije, dakle 1, M kontroliše nagib funkcije, a C i B određuju redom horizontalni i vertikalni pomeraj.

```
public virtual float Score()
{
  var x = ConsiderationInputValue;
  switch (GraphType)
  {
```

```
    case GraphType.Exponential:
    return 0f;
    case GraphType.Logistic:
    return K * (1.0f / (1.0f + Mathf.Pow((1000.0f * M * Mathf.Exp(1)), -x + C))) + B;
    case GraphType.Linear:
    return M * (x - C) + B;
    case GraphType.Quadratic:
    return M * Mathf.Pow(x - C, K) + B;
  }

  return -1.0;
}
```

U primeru akcije AttackUnitOnHexAction imamo tri razmatranja *ConsiderEnemyHealth_ Con*, *TargetGetsKilled_ Con*, *SelfGetsKilledByRetaliation*. Za razmatranje *ConsiderEnemyHealth_ Con* funkcija korisnosti biće linearna:

$$f(x) = x + 1; \tag{5.1}$$

Na osnovu poziva konstruktora klase razmatranja postavljaju se svi potrebni parametri za računanje korisnosti. Primer implementacije razmatranja *ConsiderEnemyHealth_ Con* dat je u sledećem segmentu.

```
class ConsiderEnemyHealth_Con : ConsiderationBase
{
    public ConsiderEnemyHealth_Con(UnitBehaviour Owner, HexBehaviour targetHex)
    : this(Owner, ConsiderationInputType.TargetHealth, targetHex,
           1, 1, 0.1f, 0, GraphType.Linear)
    {}

    public ConsiderEnemyHealth_Con(UnitBehaviour Owner, HexBehaviour targetHex,
                                   ConsiderationInputType CI,
                                   float K, float M, float B, float C,
                                   GraphType GraphType)
    : base(Owner, CI, targetHex, K, M, B, C, GraphType)
    {}

    public ConsiderationInputType ConsiderationInputType
    {
        get { return ConsiderationInputType.TargetHealth; }
    }
}
```

TargetGetsKilled_ Con razmatranje proverava da li će neprijateljska jedinica stradati u napadu, i u tom slučaju daje veliki bonus. Slično, razmatranje *SelfGetsKilledByRetaliation* proverava da li će borbena jedinica agenta preživeti povratnu

štetu, i ako ne, znatno smanjuje ocenu. Sledi primer razmatranja *TargetGetsKilled_Con*, ono neće koristiti metode iz bazne klase za računanje ocene:

```
class TargetGetsKilled_Con : ConsiderationBase
{
  public TargetGetsKilled_Con(UnitBehaviour Owner, HexBehaviour targetHex)
  : base(Owner, targetHex, ConsiderationInputType.TargetHealth,
         0, 0, 0, 0, GraphType.Custom)
  {}

  ...
  //...this should be translated to function...
  public override float Score()
  {
    UnitBehaviour ub = (UnitBehaviour)targetHexContex.ObjectOnHex;
    if (OwnerOFConsideration.Damage >= ub.CurrentHealth)
       return 1.5f;
    else
       return (1 - (ub.CurrentHealth - OwnerOFConsideration.Damage)/ub.MaxHealth);
  }
}
```

Svaka akcija će se ocenjivati na ovaj način, prolaskom kroz listu razmatranja, obradom informacija sa konteksta razmatranja i kombinovanjem njihovih ocena.

Implementacija proširena algoritmom Minimax sa α - β odsecanjem

U ovoj implementaciji algoritam Minimax sa α - β odsecanjem se koristio za proširenje implementacije sistema ponašanja agenta zasnovanog na korisnosti. U svakom čvoru stabla igre se nalazi simulirano stanje igre. U korenu stabla je polazno stanje, u kom agent dobija određen broj najbolje ocenjenih akcija (ocenjenih pomoću metoda pristupa zasnovanog na korisnosti). Za svaku od akcija kreira se kopija trenutnog stanja u kom se ta akcija simulira. Zatim se u tako izmenjenom stanju simulira dalji tok igre, tj. prepušta se kontrola sledećoj borbenoj jedinici na potezu, kreira se broj mogućih akcija za nju, a zatim se opet kreiraju kopije stanja. Pretraga se vrši do zadate dubine, ili kada su zadovoljeni uslovi kraja igre (jedan od igrača je ostao bez borbenih jedinica). Dakle ukoliko je agent tipa *AgentType.MinimaxAI*, prilikom poziva metode *ChooseAction*, prvo se poziva metod *StartMinimaxAB* u kojem se, na osnovu ocenjivanja korišćenih u pristupu zasnovanom na korisnosti (*ScoreActions*), dobijaju polazne akcije agenta. Dobi-

jene akcije se smeštaju u listu koja se dalje prosleđuje glavnoj metodi algoritma *MinimaxAB*. Pregled metode *StartMinimaxAB* dat je u primeru kôda koji sledi.

```
private void StartMinimaxAB()
{
  ScoreActions();

  int numberOfActions = Math.Min(ScoredActions.Count, 3);

  List<IAction> chosenActions =
    ScoredActions.OrderByDescending(x => x.ScoredValue).Take(numberOfActions).ToList();

  MinimaxAB(0, true, ActionManager.Instance.CurrentlySelectedPlayingUnit.PlayerId,
    chosenActions, MIN, MAX);

  IAction actionToDo = chosenActions.OrderByDescending(x => x.SimulatedValue).First();
  BattlefieldManager.ManagerInstance.RevertToOriginalGameState();
  actionToDo.DoAction();
}
```

Da bi se ograničio broj simuliranih stanja, broj početnih akcija je 3, dok je dubina ograničena na 4. U pozivu metode *MinimaxAB* vidimo vrednost početne dubine, da li je igrač na potezu maksimizator, identifikator igrača koji je maksimizator, što će se koristiti u funkciji evaluacije, listu akcija koje se ocenjuju kao i konstante *MIN* i *MAX*. Sve akcije će nakon izvršene metode imati postavljene vrednosti polja *SimulatedValue*, na osnovu kojih se sortirati, a potom i izabrati akcija. Sledi prikaz metode *MinimaxAB*:

```
static void MinimaxAB(int depth, bool maximizingPlayer, int currentPlayerID,
                      List<IAction> actions,
                      decimal alpha, decimal beta)
{
    int valueOfEndGame;
    if (EndReached(out valueOfEndGame, currentPlayerID) || depth == 4)
    {
      var state = BattlefieldManager.ManagerInstance.CurrentStateOfGame;
      decimal playerHealth =
         state.GetHealthScoreOfTheStateForPlayer(currentPlayerID);
      decimal enemyHealth =
         state.GetHealthScoreOfTheStateForPlayer(enemyPlayer(currentPlayerID));

      foreach (IAction action in actions)
         action.SimulatedValue =
            action.SimulateScoreForHealth(playerHealth, enemyHealth) + valueOfEndGame;
      return;
    }

    if (maximizingPlayer)
    {
```

```
        decimal bestValue = MIN;

        foreach (IAction action in actions)
        {
            CopyStateAndSimulateAction(action);

            // utility ai scoring
            List<IAction> availableActions =
               AIAgentInstanceAgent.GetAvailableActions().
               OrderByDescending(x => x.ScoredValue).Take(3).ToList();

            MinimaxAB(depth + 1, false, currentPlayerID, availableActions,alpha, beta);

            decimal minimaxedActionValue = availableActions.Max(x => x.SimulatedValue);

            bestValue = Math.Max(minimaxedActionValue, bestValue);
            alpha = Math.Max(alpha, bestValue);

            action.SimulatedValue = bestValue;
            BattlefieldManager.ManagerInstance.ReturnToPreviousState();
            // Alpha Beta Pruning
            if (beta <= alpha)
               break;
        }

        return;
    }
    else
    {
        decimal bestValue = MAX;

        foreach (IAction action in actions)
        {
            CopyStateAndSimulateAction(action);

            //scoring utility ai
            List<IAction> availableActions =
              AIAgentInstanceAgent.GetAvailableActions().
              OrderByDescending(x => x.ScoredValue).Take(3).ToList();

            MinimaxAB(depth + 1, true, currentPlayerID, availableActions, alpha, beta);

            decimal minimaxedActionValue = availableActions.Min(x => x.SimulatedValue);

            bestValue = Math.Min(minimaxedActionValue, bestValue);
            beta = Math.Min(beta, bestValue);

            action.SimulatedValue = bestValue;
            BattlefieldManager.ManagerInstance.ReturnToPreviousState();

            // Alpha Beta Pruning
            if (beta <= alpha)
                break;
```

```
        }

        return;
    }
}
```

U svakom prolasku kroz stablo, u čvoru koje nije list, najpre se prolazi kroz listu akcija (polazna akcija), i za svaku od njih prvo kopira, a zatim i menja stanje igre. U toj promenjenoj kopiji polaznog stanja, bira se sledeća borbena jedinica na potezu, za nju se generišu 3 moguće akcije. Nakon toga za se za to stanje poziva MinimaxAB, po čijem povratku generisane akcije dobijaju simuliranu vrednost. Onda u zavisnosti od toga da li je igrač na potezu maksimizator ili ne, polazna akcija dobija vrednost jedne od ocenjenih akcija. Nakon što je polazna akcija dobila vrednost, briše se izmenjena kopija stanja pomoću metode *ReturnToPreviousState*. Ukoliko smo dostigli završnu dubinu, za svaku od akcija procenjuje se njena simulirana vrednost. Kao parametar procene uzeta je razlika ukupnog broja života polaznog igrača i njegovog protivnika. Ovako će se izabrati akcija koja može odvesti u najbolje stanje u toku 4 sledeća poteza.

Pregled funkcije kopiranja trenutnog stanja igre, funkcija *CopyStateAndSimulateAction*. Unutar funkcije *CreateNewAndChangeCurrentGameStat* kopiraju se objekti polja, borbenih jedinica i specijalnih polja i trenutno stanje se postavlja na to simulirano stanje. Pamti se akcija čijom će simulacijom nastati novo stanje igre iz kopije, i kasnije će procenjena vrednost biti sačuvana u polju *SimulatedValue* ove akcije. Kreira se kopija akcije u kopiranom stanju, koja se nakon toga simulira pozivom *SimulateAction*. Nakon toga prepusta se potez sledećoj jedinici u simuliranom stanju, resetuju se polja po kojima jedinica može da se kreće. Simulirana stanja će kasnije biti obrisana metodom *ReturnToPreviousState*.

```
public static void CopyStateAndSimulateAction(IAction action)
{
  var bfManager = BattlefieldManager.ManagerInstance;
  var actManager = ActionManager.Instance;

  //creates new game state by copying board, units and spec fields...
  bfManager.CreateNewAndChangeCurrentGameStat();
  //... saving action which will be simulated in the new state...
  bfManager.CurrentStateOfGame.actionToDoInState = action;

  //...create new action based on action in old game state...
  IAction newStateAction = CreateActionForNewState(action);
  newStateAction.SimulateAction();
```

```
  //...get next playing unit in the sim...
  bfManager.CurrentStateOfGame.GetNextUnit();

//...set agent controlled unit to newly selected unit in the sim state...
  AIAgentInstanceAgent.CurrentlyControledUnit = actManager.CurrentlySelectedPlayingUnit;

//...reseting movable hexes for the new unit
  bfManager.ResetTilesInRange();
  bfManager.StartingHexBehaviorTile = actManager.CurrentlySelectedPlayingUnit.
    CurrentHexTile;
  bfManager.SelectTilesInRangeSimple(actManager.CurrentlySelectedPlayingUnit.
    movementRange);
}
```

Glava 6

Poređenje implementacija inteligentnih agenata

U radu su prikazana dva pristupa razvoju sistema ponašanja inteligentnih agenta. U prvom od njih je korišćen sistem zasnovan na korisnosti, i prilikom ocenjivanja akcija u ovom sistemu, moguće je da će više akcija biti ocenjeno istom vrednošću i u tom slučaju na razne načine možemo izabrati jednu od njih, na primer slučajnim izborom ili prioritizacijom akcija određenog tipa. Proširenjem ovog sistema algoritmom Minimax sa α - β odsecanjem pokušano je da se ostvari poboljšanje u izboru odluke, simulacijom nekoliko poteza unapred. Implementacije su upoređene po parametrima jednostavnosti implementacije, vremena izračunavanja poteza i kvaliteta izabranih odluka.

6.1 Jednostavnost implementacije

Sistem za kreiranje ponašanja agenata zasnovan na teoriji korisnosti ima relativno jednostavnu implementaciju. Uvedeni su interfejsi *IAction* i *IConsideration*, koji zadaju strukturu klasa akcija i razmatranja. Ukoliko bi se dodavala nova akcija, potrebno je napraviti za nju novu klasu koja implementira *IAction* interfejs, i dodati je u funkciju za generisanje mogućih akcija *GetAvailableActions*. Dodavanje novih razmatranja akciji je moguće uključivanjem određene klase razmatranja u metodi *GetConsiderations* date akcije. Dakle, za menjanje procene akcije u ovom sistemu dovoljno je promeniti parametre funkcije korisnosti razmatranja ili napraviti novo razmatranje. U odnosu na drugi pristup, lakše je razumeti zašto je agent napravio određeni izbor akcije. Da bi se efikasno koristio pristup sa algoritmom

Minimax, potrebno je pojednostaviti strukuru kôda objekata koji čine stanje igre, jer će se pri simulaciji akcija kopirati celo stanje igre. Ovim pojednostavljenjem strukture kôda, bilo bi moguće povećati dubinu pretrage i polaznog broja akcija u drugom pristupu. Pošto se koristio kao nadogradnja sistema zasnovanog na korisnosti bilo je potrebno izmeniti postojeći sistem akcija.

6.2 Vreme izračunavanja poteza

Pošto pristup koji koristi Minimaks algoritam proširuje pristup zasnovan na korisnosti, bilo je očekivano da će za akcije izabrane na ovaj način biti potrebno više vremena izračunavanja, što jeste slučaj. Za izračunavanje vremena poteza koristila se klasa *Stopwatch* imenskog prostora *System.Diagnostics*. Prvo se definiše objekat klase, a zatim otpočinje merenje metodom *Start*. Merenje se zaustavlja pozivom metode *Stop*, a izmereno vreme se čuva u polju *Elapsed*. Primer merenja vremena potrebnog za generisanje i ocenu akcija u sistemu ponašanja agenta zasnovanog na korisnosti da je u sledećem segmentu kôda.

```
...
if (AIType == AgentType.UtilityAI)  {
  Stopwatch stopwatch = new Stopwatch();
  stopwatch.Start();
  ScoreActions();

  IAction chosenAction = ScoredActions.OrderByDescending(x => x.ScoredValue).First();

  stopwatch.Stop();
  Debug.Log("Time elapsed:"+stopwatch.Elapsed.TotalMilliseconds);
  chosenAction.DoAction();
}
...
```

Prosečna partija bila bi gotova između 23-38 poteza, u zavisnosti od početne pozicije borbenih jedinica. Prosečno vreme donošenja odluke korišćenjem pristupa sa Minimax algoritmom iznosi $\sim$ 612.8694333 milisekundi, dok u pristupu zasnovanom na korisnosti iznosi $\sim$ 0.406891304 milisekundi. Treba napomenuti da je prosečno vreme za koje se funkcija kopiranja stanja izvršavala bilo $\sim$ 21.25 milisikundi.

6.3 Kvalitet izabranih odluka

Poređenje kvaliteta izabranih odluka vršeno je tokom niza partija igre, u kojima su bili suprotstavljeni dva igrača vođeni različitim agentima. Izmenjen je deo kôda koji prepušta potez borbenim jedinicama, primer je dat u sledećem segmentu.

```
...
AIAgent.AIAgentInstanceAgent.CurrentlyControledUnit = CurrentlySelectedPlayingUnit;
if (CurrentlySelectedPlayingUnit.PlayerId == 1)
    AIAgent.AIAgentInstanceAgent.ChooseAction(AgentType.UtilityAI);
else
    AIAgent.AIAgentInstanceAgent.ChooseAction(AgentType.MinimaxAI);
...
```

Prosečna partija trajala je oko 31.07 sekundi. U slučaju da igrač koji je prvi na potezu ima prednost, organizovana su dva bloka partija u kojima su različiti igrači igrali prvi. Za svaki blok izabran je neparan broj da bi se moglo odrediti koja metoda je imala više uspeha. Prvo je organizovan blok od 7 partija, u kome je prvi na potezu bio agent zasnovan na prvom pristupu (samo korisnosti). Agent zasnovan na drugom pristupu (minimax) je pobedio u 4 partije. U preostala tri pobedu je odeno agent zasnovan na prvom pristupu. U drugom bloku testiranja, prvi na potezu je igrao agent zasnovan na drugom pristupu. I u ovom slučaju taj agent je pobedio 4 puta, a u preostale 3 partije pobedio je agent zasnovan na prvom pristupu.

Treba napomenuti da se 3 puta dogodilo da meč postane zaglavljen, tj. tri borbene jedinice, od kojih 2 pripadaju agentu drugog pristupa, ostale bi u igri sa jednim životom, i u ovom slučaju pobeda je pripisana agentu drugog pristupa.

Ove tri jedinice bi zatim birale poteze kretanja umesto napada. U ovim slučajevima pobeda je data onom igraču koji ima više preostalih jedinica. Razlog za nastajanje ove situacije je taj što u prvom pristupu, za akciju napada postoji razmatranje *SelfGetsKilledByRetaliation*, koje procenjuje da li će jedinica koja napada biti uništena povratnom štetom. Ovo razmatranje znatno smanjuje ocenu akcije takvog samoubilačkog napada. Rešenje bi bilo uvesti novo razmatranje koje proverava da li bi akcija dovela do pobede igrača koji je kontroliše i tada bi razmatranje bilo ocenjeno nekom relativno velikom vrednošću. U tom slučaju, u primeru gde se meč zaglavio, jedna od dve preostale jedinice drugog igrača bi se žrtvovala.

Primećeno je da u drugom pristupu, pošto se polazne akcije određuju na osnovu ocene u pristupu zasnovanom na korisnosti, izabrane akcije nisu često odudarale od onih izabranih tim pristupom. U drugom pristupu maksimizuje se razlika ukupnih života igrača na potezu i protivnika, dok se u prvom često bolje ocenjuju akcije napada. Ovo je možda razlog zašto su se akcije poklapale u oba pristupa. Možda bi više odudranja u izboru bilo primećeno da je dubina pretrage ili broj stanja bio veći.

Glava 7

Zaključak i dalji razvoj

U ovom radu, u uvodnom delu, dat je pregled istorije VI u video igrama, kratak pregled nekih pristupa u kreiranju inteligentnih agenata, kao i pregled razvojnog okruženja Unity. U daljim poglavljima opisani su i pristupi ad-hoc kreiranja ponašanja i algoritam Minimax, kao i njegovo poboljšanje, Minimax sa α - β odsecanjem. U implementaciji igre, ideja je bila da se uporede ova da pristupa i da se implementacija sistema ponašanja zasnovanog na teoriji korisnosti proširi pomoću Minimax algoritma sa α - β odsecanjem.

Igra se može dodatno proširiti ubacivanjem novih jedinica ili uvođenjem novih pravila, definisanjem više akcija itd. Sistemi za kreiranje ponašanja agenata mogli bi se dalje proširiti i unaprediti. Sistem korisnosti poboljšanjem postojećih i dodavanjem novih akcija i razmatranja, kao i pravljenjem vizuelnih alata za lakše kreiranje akcija i razmatranja. Kako je drugi sistem zamišljen kao nadogradnja sistema zasnovanog na korisnosti, stablo igre koje Minimax procenjuje je uniformno pošto se u svakom čvoru procenjuje b najboljih akcija, procenjenih prvim sistemom. Zbog poretka kojim se prvo procenjuju najbolje ocenjene akcije očekivano je da će se proceniti $\sim b^{\frac{d}{2}}$ čvorova [9]. Međutim zbog neefikasne implementacije metode kopiranja stanja, čije je prosečno trajanje iznosilo ~ 21.25 milisikundi, ne bi bilo isplativo povećavati dubinu i broj procenjenih akcija pretarage. Ukoliko bi se kopiranje stanja implementiralo efikasnije, sa specijalizovanim strukturama podataka, možda bi to bilo isplativo. Imajući u vidu da igre ovog tipa obično sadrže veći broj stanja i mogućih akcija(više jedinica sa raznovrsnijim izborem akcija, više specijalnih polja) da li je isplativo dodavati Minimax, zavisi od složenosti implementacije struktura kojima se predstavlja stanje igre. Pretpostavlja se da bi prvi pristup u takvim igrama bio jednostavniji za implementaciju, sa dovoljno

dobrim izborom poteza.

Literatura

[1] Kaja Damnjanović and Ivana Janković. Normativna i deskriptivna teorija donošenja odluka u uslovima rizika. *Theoria*, pages 25–50, 2014.

[2] David Mark, Luke Dicken, Dino Dini. Architecture Tricks: Managing Behaviors in Time, Space, and Depth, 2013. on-line at: `https://www.gdcvault.com/play/1018040/Architecture-Tricks-Managing-Behaviors-in/`.

[3] Miljan Mijić, Marko D Petković. Unity: Osnovni koncepti i razvoj 3d igre. 2016.

[4] Mladen Nikolić, Predrag Janičić. *Veštačka inteligencija*. Matematički fakultet, 2021.

[5] Wikipedia contributors. Turn-based tactics — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Turn-based_tactics&oldid=1037001426`, 2021. [Online; accessed 5-September-2021].

[6] Wikipedia contributors. Video game — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Video_game&oldid=1041941915`, 2021. [Online; accessed 5-September-2021].

[7] Georgios N Yannakakis and Julian Togelius. *Artificial intelligence and games*. Springer, 2018.

[8] Robert Zubek. Needs-based ai. *Game programming gems*, 8:302–11, 2010.

[9] Vladimir Jocović i Adrian Milaković. Inteligentni sistemi — Teorija igara, Sekvencijalne igre. `http://ri4es.etf.rs/materijali/vezbe/IS_Teorija_igara_sekvencijalne_igre.pdf`, 2020.