Algoritmi umjetne inteligencije u strateškim igrama u stvarnom vremenu

Mešić, Eldin

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni diplomski studij Računarstvo

ALGORITMI UMJETNE INTELIGENCIJE U STRATEŠKIM IGRAMA U STVARNOM VREMENU

Diplomski rad

Eldin Mešić

Osijek, 2024

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za ocjenu diplomskog rada na sveučilišnom diplomskom studiju

Ocjena diplomskog rada na sveučilišnom diplomskom studiju

Ime i prezime pristupnika:	Eldin Mešić
Studij, smjer:	Sveučilišni diplomski studij Računarstvo
Mat. br. pristupnika, god.	D1309R, 07.10.2022.
JMBAG:	0165083266
Mentor:	izv. prof. dr. sc. Časlav Livada
Sumentor:	
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	izv. prof. dr. sc. Alfonzo Baumgartner
Član Povjerenstva 1:	izv. prof. dr. sc. Časlav Livada
Član Povjerenstva 2:	prof. dr. sc. Krešimir Nenadić
Naslov diplomskog rada:	Algoritmi umjetne inteligencije u strateškim igrama u stvarnom vremenu
Znanstvena grana diplomskog rada:	**Umjetna inteligencija (zn. polje računarstvo)**
Zadatak diplomskog rada:	U radu je potrebno napraviti analizu različitih umjetnih inteligencija te kvalitativno ocijeniti uspješnost primjene pojedinih na mogućnost suprotstavljanju korisniku u stvarnom vremenu u strateškim igrama. Rezervirano za studenta: Eldin Mešić

Datum ocjene pismenog dijela diplomskog rada od strane mentora:	11.09.2024.
Ocjena pismenog dijela diplomskog rada od strane mentora:	Izvrstan (5)
Datum obrane diplomskog rada:	03.10.2024.
Ocjena usmenog dijela diplomskog rada (obrane):	Izvrstan (5)
Ukupna ocjena diplomskog rada:	Izvrstan (5)
Datum potvrde mentora o predaji konačne verzije diplomskog rada čime je pristupnik završio sveučilišni diplomski studij:	03.10.2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O IZVORNOSTI RADA

Osijek, 03.10.2024.

Ime i prezime Pristupnika:	Eldin Mešić
Studij:	Sveučilišni diplomski studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D1309R, 07.10.2022.
Turnitin podudaranje [%]:	3

Ovom izjavom izjavljujem da je rad pod nazivom**: Algoritmi umjetne inteligencije u strateškim igrama u stvarnom vremenu**

izrađen pod vodstvom mentora izv. prof. dr. sc. Časlav Livada

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis pristupnika:

SADRŽAJ

1. UVOD

Tema diplomskog rada je proučavanje algoritama umjetne inteligencije korištenih za upravljanje umjetnih protivnika kod strateških igara u stvarnom vremenu (engl. *Real Time Strategy*, RTS). U današnje vrijeme umjetna inteligencija je postala često korišten alat koji se koristi za velik broj raznovrsnih potreba. Potrebno je izdvojiti korisnost umjetne inteligencije u kontekstu aplikacija i igara gdje se njene prednosti mogu najbolje iskoristiti i to na lako prepoznatljiv način.

Kada su u pitanju strateške igre najčešći primjer umjetne inteligencije je kod umjetnih protivnika koji na smislen i logičan način donose odluke veoma slične onima pravih osoba različitih vještina. Najpoznatiji način na koji moderne strateške igre implementiraju umjetnu inteligenciju je preko strojnog učenja koje trenira umjetnu inteligenciju prema prethodno određenim vrijednostima i standardima [1].

No, ovakav način implementacije nije uvijek bio dostupan. Povijest umjetne inteligencije je puna raznih algoritama i tehnika koje su usprkos svojim manama i jednostavnom dizajnu u doba kada je tehnologija bila ograničena, ispunili tražene zahtjeve te se koriste i u današnje vrijeme [1].

Cilj ovog diplomskog rada je implementacija i analiza rezultata algoritama i metoda umjetne inteligencije u kontekstu računalno kontroliranih protivnika s umjetnom inteligencijom (engl. *Artificial Intelligence*, AI) kod strateških igara u stvarnom vremenu u svrhu proučavanja efikasnosti, prednosti i nedostataka svake metode. Ova implementacija i analiza će biti odrađena preko samostalno izrađene i prilagođene aplikacije.

1.1 Zadatak diplomskog rada

Zadatak diplomskog rada je analiza algoritama umjetne inteligencije kod strateških igara u stvarnom vremenu u svrhu proučavanja njihovih prednosti i nedostataka te zaključka kada i zašto ove tehnike koristiti i u današnje vrijeme. Kroz rad će se proći i kroz izradu jednostavnije strateške aplikacije koja će implementirati ove algoritme. Na taj način će se odrediti rezultati uspješnosti svakog algoritma te bolje upoznati s njihovom implementacijom i pretvaranjem koncepta algoritma u razvijenog AI protivnika.

2. PREGLED PODRUČJA RADA

U ovom poglavlju su predstavljene postojeće RTS aplikacije i strateške igre na poteze (engl. *Turn Based Strategy*, TBS) koje su implementirale neke od osnovnih algoritama umjetne inteligencije u svrhu postizanja izazovnog i inteligentnog AI protivnika. Ove aplikacije su također služile kao inspiracija i osnova za izradu samostalnog rješenja zadatka diplomskog rada.

2.1 Warcraft III

Warcraft III, kojeg je razvio studio za razvoj igara *Blizzard Entertainment*, smatra se jednom od najpoznatijih RTS igara koje su promijenile industrijski standard sa svojim sistemom jedinstvenih herojskih jedinica (engl. *Hero Unit*), unikatnim i jedinstvenim rasama te fokusom na mikro menadžment. Sve te značajke i efikasni marketing, kojim su se kroz priču uveli konflikti interesa i ideologija različitih fantastičnih rasa su *Warcraft III* brzo doveli na vrh RTS igara u narednim godinama nakon svojeg izdanja.

Nerazvijena logika i lijen način implementacije napravili su *Warcraft* AI ozloglašenim. Zbog svojeg djelomično unikatnog načina igranja, *Warcraft* AI nije mogao iskoristiti neke od poznatijih AI algoritama i gotovih rješenja, zbog čega je razvoj patio. S obzirom na to da su AI protivnici bili daleko slabiji od pravih igrača, dobili su nepravedne prednosti poput povećane generacija resursa, snažnijih jedinica i povoljnije početne pozicije.

2.2 Age of Empires II

Age of Empires II se smatra jednom od najpopularnijih RTS igara, koju su mnogi ljubitelji žanra proglasili najboljom strateškom igrom. Američka firma *Ensemble Studios*, osnovana 1994. godine, razvila je 1999. godine *Age of Empires II*,njihovu drugu igru i direktnog nasljednika *Age of Empires I*. Koristeći osnove iz prve igre, sve mehanike igre i povijesna točnost su nadograđene i usavršene.

U Age of Empires II igračima se nudi izbor jedne od trinaest različitih civilizacija koje variraju u prednostima i nedostatcima svojih jedinica (engl. *Units*), tehnologijama i stilovima arhitekture, sve inspirirane stvarnom poviješću civilizacije. Uvođenje povijesnih ličnosti (engl. *Historical Figures*) i scenarija povezanih s njima predstavlja jedan od glavnih razloga uspješnosti ove igre.

Age of Empires II je u svakom smislu istinita RTS igra koja usavršava sve aspekte strateških igara. Jedini je problem način rada njene umjetne inteligencije. Iako je sama implementacija dobro izvršena i strateško razmišljanje točno, previše robotsko razmišljanje i logika kojom AI protivnik dolazi do svojih akcija stvara probleme [2]. Nakon više godina izdana je konačna verzija (engl. *Definitive Edition*) ove igre u kojoj je popravljena logika AI protivnika uvođenjem više akcijskih parametara prilagođenih težini protivnika koju korisnik odredi. Tablica 2.1 prikazuje neke od tih parametara, koji se ističu kao bitni.

AI PONAŠANJE	*Najlakša težina*	*Lagana težina*	*Srednja težina*	*Teška težina*	*Najteža težina*
Unaprjeđuju kroz doba	Kada ljudski igrač dođe do tog doba	Polako kao novi igrač	Kao napredni igrač	Što brže	Što brže
Napadaju prvi	Ne	Ne	Povremeno	Da	Da
Grade svjetska čuda	Ne	Ponekad	Da	Da	Ponekad
Napadaju na gradove	Ne	Ponekad	Da	Da	Da
Hoće li trgovati	Da	Da	Povremeno	Ne	Ne
Dodatni resursi	Ne	Ne	Ne	Ne	Da
Rade zajedno	Ne	Ponekad	Povremeno	Da	Da

Tablica 2.1: Dinamično ponašanje AI protivnika prema postavljenoj težini

2.3 Starcraft II

Blizzard Entertainmenta je 2010. godine također razvio i *Starcraft II*, RTS igru koja je svoju slavu stekla u području eSporta. Kao RTS igra koju je razvio studio zaslužan za *Warcraft* franšizu, Starcraft je brzo postao poznat zbog svojeg inovativnog toka igre koji igračima svaki put omogućava novo i uzbudljivo iskustvo.

Ovdje su popravljeni mnogi problemi AI protivnika koje je *Warcraft* imao, a *Starcraft* se još dodatno razvio i proslavio u natjecateljskom području u videoigrama u kojima su kritičko strateško razmišljanje i brza akcija bitni za pobjedu. *Starcraft* je poznat po svojim RTS mehanikama koje zahtijevaju visok stupanj vještine. Strategija *Starcraft*a zahtijeva brzu akciju i znanje mikro menadžmenta i makro menadžmenta, što ukazuje na kompleksnost njenog dizajna i dizajna logike korištenog AI algoritma [3].

2.4 Sid Meier's Civilization V

Sid Meier's Civilization V, poznat kao *Civilization V*, smatra se pseudo RTS igrom koju je *Firaxis Games* studio razvio 2010. godine. Za razliku od prijašnjih RTS igara, *Civilization V* se temelji na konceptu TBS igre, s razlikom u tome što svi igrači svoje poteze rade u isto vrijeme. Na taj se način ograničene akcije TBS igre spajaju s akcijom i interakcijama koje pružaju RTS igre.

Dizajn AI protivnika u *Civilization V* drastično se razlikuje od prethodno navedenih ponašanja. Zbog ograničenih opcija koje pruža sistem poteza, *Civilization V* može detaljnije napraviti akcije koje njezin AI protivnik koristi te također dijeli ponašanje glavnih elemenata na odvojene AI implementacije. Na takav se način odvaja AI razmišljanje planiranja grada, menadžmenta resursa, vojnih pokreta, diplomatskih veza i slijeđenja različitih uvjeta pobjede [4].

2.5 The Battle For Wesnoth

The Battle For Wesnoth, kao i *Civilization V*, smatra se pseudo RTS igrom napravljenom kao projekt otvorenog koda (engl. *Open-Source*) gdje se više ljubitelja i fanova klasičnih RTS i TBS igara udružilo kako bi napravili unikatnu stratešku igru [5].

Godine velikog truda, dobre organizacije i impresivnog talenta sudionika omogućile su da se *The Battle For Wesnoth* uzdigne na status uspješne strateške igre koja umanjuje vrijednost daleko većih i sponzoriranih studija. Zahvaljujući korisnički kreiranim modifikacijama i proširenjima strastvenih fanova i posvećene zajednice, onda se i dalje unaprjeđuje, širi i raste.

Status pseudo RTS igre dodijeljen je iz istih razloga kao i kod *Civilization V*, gdje se runde igre sastoje od poteza u kojima svi igrači istovremeno obavljaju svoje radnje. Prepoznatljivost estetike *The Battle For Wesnotha* smatra se jednim od razloga njezine uspješnosti kod specifičnih igrača.

Teren igre, za razliku od poligonalnih ravnina, korištenih u igrama kao što su *Warcraft* ili *Starcraft,* sastoji se od šesterokutnih polja (engl. *Hextiles*) koji joj daju unikatan izgled prikazan na slici 2.1. Ovo je omogućilo lakšu implementaciju nekih algoritama te lakšu preglednost i predvidljivost, što igračima pomaže u određivanju budućih poteza.

Slika 2.1: *The Battle For Wesnoth* Hextile teren [6]

Zbog Hextile načina izrade terena, AI algoritmi koji se koriste u igri *The Battle For Wesnoth* smatraju se efektivnim i hardverski nezahtjevnim, unatoč njihovom jednostavnom dizajnu i logici izvršavanja. AI protivnika podijeljen je na više dijelova kao što su kupovina jedinica, kretnja jedinica, napadanje jedinica, potreba za resursima, određivanje sljedećeg strateškog koraka i slično. Također, ovaj AI je modificiran kako bi funkcionirao drugačije ovisno o trenutnom scenariju igre.

3. ZAHTJEVI NA PROGRAMSKO RJEŠENJE

Kako bi rezultati algoritama AI protivnika bili efektivno ocijenjeni i analizirani ključno je odrediti koje algoritme treba analizirati, na koji ih način ocijeniti te gdje ih implementirati. Prema većini TBS i RTS aplikacija i igara, uključujući i one navedene u prethodnom poglavlju, mogu se identificirati neki od najčešće korištenih i osnovnih algoritama koji se i danas koriste za razvoj AI protivnika modernih RTS igara, bilo u njihovom originalnom obliku ili kao osnova za razvijenije algoritme. Iz prethodno navedenih RTS igara može se zaključiti da je za kompliciranije RTS igre potrebno razdvojiti logiku AI protivnika na više dijelova, svaki sa svojom logikom razmišljanja i donošenjem odluka. Navedeni dijelovi bi trebali upravljati svojim strateškim područjem RTS aplikacije te, je neophodno definirati zahtjeve idealne RTS aplikacije. Na taj se način algoritmi AI protivnika mogu kvalitetno i efektivno ocijeniti.

Za rješavanje problema ovog diplomskog rada potrebno je razviti prilagođenu RTS aplikaciju koja će imati samo najvažnije dijelove strateških igara. S obzirom na to da će rad biti usmjeren na algoritme AI protivnika, aplikaciji nije potrebno previše informacija pa bi izbor bilo koje od prethodno navedenih RTS igara davao nepotpune ili nekontrolirane rezultate. Kreirana aplikacija će biti procjenjiva prema definiranim standardima algoritama, a sama gradnja aplikacije će omogućavati bolje razumijevanje logike iza implementacije algoritama, kao i uključivanje tih informacija u analizu AI algoritama i konačni rezultat. Menadžment, upravljanje pokretima i varijacija borbenih jedinica su ključni dijelovi RTS igara koji ih čine strateški orijentiranim. Prethodno navedeni primjeri RTS igara su uključivali borbene jedinice od najosnovnijih vojnika do fantazijskih bića i rasa, no što se strategije tiče, najvažniji čimbenik je klasa borbene jedinice. Klase poput strijelaca, konjanika, čarobnjaka, kopljanika i sličnih razlikuju se u prednostima i nedostatcima. S obzirom na to, važan dio strategije predstavlja ispravan odabir klase jedinica koje će se koristiti i njihova kvantitativna distribucija u arsenalu igrača [7].

Iako je glavni cilj strateške igre pobijediti protivnika na nekoliko različitih načina kao što su uništavanje svih borbenih jedinica, zauzimanje protivničkih baza ili diplomatska pobjeda, to ne smije biti jedina stvar na koju se AI fokusira [8]. Kako bi postigao veći uspjeh, AI protivnik trebao bi se najprije usmjeriti na korisne ciljeve koji mu mogu donijeti prednost u širem kontekstu igre. Stoga, prilagođena aplikacija treba uključivati vrste ciljeva koji će AI-u pomoći razviti bolju strategiju i planirati sljedeće poteze.

Prema navedenim uvjetima, mogu se definirati glavni kriteriji za prilagođene RTS aplikacije koji će omogućiti učinkovitu i preciznu evaluaciju različitih AI algoritama protivnika:

1. Aplikacija mora biti dovoljno jednostavna kako bi potezi i odluke AI protivnika bili predvidljivi i usporedivi s ispravnim potezima.

2. Logika AI mora biti podijeljena na više dijelova ili segmenata kao što su kupovina ili novačenje borbenih jedinica i akcije pojedinačnih borbenih jedinica.

3. Potrebno je implementirati barem jedan osnovni cilj koji će prisiliti AI protivnika da ga osvoji kako bi stekao prednost i borbenu prisutnost. Taj cilj može biti nešto jednostavno poput kućice ili tornja koji pruža resurse za izgradnju većeg broja borbenih jedinica.

4. Borbene jedinice se moraju podijeliti na više klasa koje imaju jasno vidljive mane, snage i definiranu jačinu.

5. Uvjet pobjede također mora biti pojednostavljen kako bi se moglo pratiti ponašanje AI protivnika u različitim trenutcima borbe.

Kada prilagođena RTS igra zadovolji sve ove uvjete, moći će se implementirati odabrani AI algoritmi koji najbolje prikazuju kompleksnost i uspješnost RTS igara. U RTS igrama opisanim u prethodnom poglavlju koriste se tri glavna algoritma AI protivnika, bilo u njihovim osnovnim verzijama ili kao kompleksnije verzije prilagođene za rad u željenom obliku. Ti algoritmi su algoritam stabla odluka (engl. *Decision Tree*, DT), algoritam stabla ponašanja (engl. *Behaviour Tree*, BT) i algoritam temeljen na korisnost (engl. *Utility-Based*, UB) [9].

4. DIZAJN RTS APLIKACIJE

Kako bi se efikasno mogli ocijeniti različiti algoritmi AI protivnika u RTS igrama potrebno je napraviti prilagođenu RTS aplikaciju u kojoj će se lagano moći vidjeti razlika i uspješnost u ponašanju različitih AI algoritama. U prošlom poglavlju su bili razmotreni različiti zadatci, uvjeti i zahtjevi koje ova prilagođena RTS aplikacija mora implementirati kako bi se dobili najbolji i najtočniji rezultati.

Prvi korak u stvaranju RTS aplikacije je odabir razvojnog okruženja i alata koji će se koristiti za razvoj i izradu aplikacije. S obzirom na zahtjev za jednostavnošću aplikacije, kako bi se rezultati lakše očitavali, nije potrebno koristiti kompleksne programe poput *Unreal Engine*a. Za ovaj stil jednostavne aplikacije najbolje je koristiti 2D projekt, što su izbor programa za razvoj aplikacija sužava na alate kao što su *Godot* ili *Unity Game Engine*. Za razvoj aplikacije odabran je *Unity Game Engine*, koji koristi široko poznati programski jezik C#, za razliku od *Godot*ovog integriranog jezika *GDScript*.

4.1 Unity Game Engine

Unity Game Engine jedno je od najpopularnijih razvojnih okruženja za izradu web, mobilnih i desktop aplikacija. Prvu verziju, objavljenu 9. kolovoza 2005. godine, razvila je tvrtka *Unity Technologies* i od tada, razvojno okruženje kontinuirano bilježi rast popularnosti. Njegov jednostavan alat za razvoj video igara i računalne grafike omogućio je mnogim programerima ulazak u svijet složenijih aplikacija. Ključni razlog njegovog uspjeha je mogućnost izvođenja na svim platformama dostupnim na računalnom tržištu. Korištenje *Unitya* od strane različitih tipova programera na raznim uređajima osiguralo je njegovu uspješnost i popularnost na tržištu [10].

Unity Game Engine smatra se prvom platformom koja je u potpunosti podržavala sve mogućnosti različitih uređaja i operacijskih sustava. Korisnicima su dostupni svi potrebni alati za razvoj igara, uključujući alate za uređivanje slika, 3D modeliranje i povezivanje s bilo kojom vrstom koda [11]. Osim navedenog, integracija izvornih formata omogućuje uvoz vektorske ili druge grafike iz programa poput *Photoshopa*, te korištenje 3D i 2D modela iz poznatih alata za modeliranje kao što su *Blender* i *Maya*.

Unity Game Engine, jedan od najstarijih i najduže podržanih alata, koristi lako prepoznatljiv i mnogim programerima poznat C# programski jezik. To znači da ima veliku podršku i opsežnu kolekciju korisničkih alata i dodataka na svom raspolaganju.

Što se tiče programskog koda, *Unity Game Engine* dolazi s ugrađenim IDE-om nazvanim *MonoDevelop*, dizajniranim za podršku programskih jezika kao što su Visual Basic .NET, C, C++, C#, F# i Vala. U ovom radu, korišten je za dizajn aplikacije, razmještaj i funkcionalnost različitih objekata i elemenata korisničkog sučelja, povezivanje svih objekata i dijelova s C# skriptama napisanima u Visual Studiu, za upravljanje svim potrebnim resursima, za implementaciju algoritama AI protivnika te za sam rad aplikacije.

4.2 Funkcionalnost RTS Aplikacije

Aplikaciji nije potreban glavni izbornik preko kojeg korisnik može mijenjati razne scene zbog njezine ne komercijalne upotrebe i rada na jednoj sceni. Korisniku bi trebao biti omogućen izbor veličine borbene karte (engl. *Battle Map*) koja je predstavljena šesterokutnom mrežom. Nakon toga će se sva šesterokutna polja proceduralno inicijalizirati. Korisniku će se sa strane prikazati izbornik s opcijama za pokretanje i zaustavljanje bitke, odabira veličine označivača (engl. *Highlighter*) te odabira različitih terenskih polja (engl. Terrain Tiles) koja predstavljaju drukčiji teren pločica kao što su duboka i plitka voda, različite boje trave, planine, dvorci, pijesak i slično.

Korisnik bi trebao moći klikom miša na postojeća terenska polja postaviti novo-odabrana terenska polja i na taj način napraviti svoju borbenu kartu. Nakon toga bi klikom na dugme za pokretanje borbene faze trebao moći kontrolirati jedan dvorac kao igrač ili promatrati bitku između dva ili više AI protivnika od kojih bilo koji može koristiti bilo koji algoritam logike AI protivnika.

Tijekom borbene faze će se svakom sudioniku ili igraču pružiti početni iznos zlata ili resursa za kupovanje jedinica i jedan dvorac koji predstavlja glavnu bazu tog sudionika, nakon čega igrač može klikom na svoj dvorac otvoriti izbornik kupovine različitih jedinica. Kada se kupi, jedinica se pojavi na polju dvorca te je klikom korisnik može pomaknuti na susjedno polje ili napasti jedinicu susjednog protivnika. Nakon bilo koje akcije, jedinica mora čekati neko određeno vrijeme prije nego bi mogla odraditi novu akciju.

Na borbenoj karti bi trebala postojati posebna polja koja će predstavljati ciljeve na koje jedinice mogu stati kako bi ih osvojili za svojeg vlasnika. Posjedovanje tih polja će igraču periodično davati dodatne resurse za kupovinu novih jedinica.

Kada jedna jedinica stane na polje koje predstavlja dvorac protivnika, taj igrač gubi svoj dvorac i u situaciji ako mu je to bio jedini dvorac ispada iz igre. Igrač koji je osvojio novi dvorac ima mogućnost izgradnje jedinica preko novo-osvojenog dvorca. Jedna runda igre traje dok god ne ostane samo jedan igrač, odnosno dok god jedan igrač nije vlasnik svih ostalih dvoraca u igri.

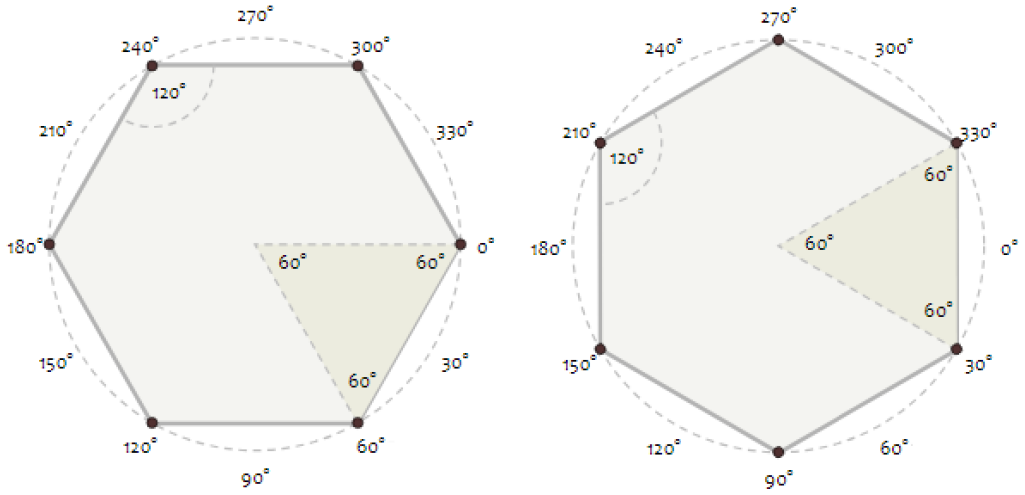
4.3 Borbena Rešetka

Prvi korak izrade RTS aplikacije je određivanje dizajna borbene rešetke na kojoj će se voditi borbe. S obzirom na to da strateške igre zahtijevaju jednako povoljnu poziciju i stanje svakog igrača na početku je igre ključno odrediti oblik polja borbene rešetke. Zbog svojih dijagonalno simetrijskih svojstava, šesterokut je najbolji oblik polja za strateške igre.

Šesterokuti se često koriste u matematičkim i geometrijskim modelima zbog svojih jedinstvenih svojstava koja omogućuju učinkovito popunjavanje prostora bez preklapanja ili praznina, što nije slučaj kod kvadrata. Zbog svoje geometrijske pravilnosti šesterokutne mreže olakšavaju složene proračune u simulacijama, kao što su modeliranje kretanja ili izračun dometa i crte vidljivosti [12]. Osim toga, šesterokutna struktura omogućava prirodniji i estetski ugodniji prikaz terena, što je korisno u razvoju videoigara i simulacija okoliša. Međutim, ovaj pristup donosi i izazove, budući da složenija geometrija šesterokuta može zahtijevati naprednije algoritme i veću računalnu snagu za obradu, za razliku od jednostavnijih kvadratnih mreža [12].

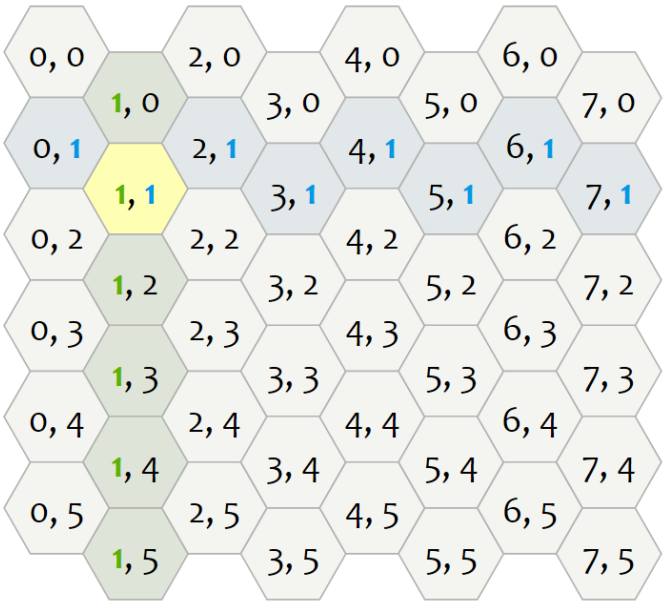
Nakon što se odredio oblik borbene rešetke potrebno je preko Unity C# skripti napraviti upravitelja ili menadžera koji će kontrolirati oblik karte i omogućavati lagani pristup svakom polju preko koordinatnog sustava namijenjenog za šesterokutne oblike. Taj menadžer je skripta HexGridManager.cs. Unutar nje su definirani singleton instanca iste klase za jednostavniji pristup i dictionary tip objekta sa Vector2Int ključem i predstavlja 2D koordinate specifičnog polja kojemu je spremljena vrijednost pod tim ključem. HexGridManager radi na jednostavan način gdje mu se na klik pozove funkcija LayoutGrid u kojoj se prođe kroz sve koordinate i instancira polje na svakoj od njih. Nakon što se novo polje instancira pozicija mu se postavi na točno određenu vrijednost koja ovisi samo o njegovim koordinatama preko metode GetPositionFromCoordinate. GetPositionFromCoordinate je jedna od najvažnijih funkcija ove aplikacije jer ona određuje na koju lokaciju se instancira novo polje.

Način dodjele koordinata šesterokutnoj rešetci nije jednostavan kao kod kvadratne rešetke. Sa šesterokutima postoji više načina kako im pridijeliti koordinate, ali onaj koji će se u ovom radu koristiti je način pomaka koordinata (engl. *Offset coordinates*). Prvi korak u tom načinu je odlučiti hoće li šesterokuti biti ravnog vrha ili šiljastog vrha. Ovo se odnosi na orijentaciju gornjeg vrha šesterokuta. Ako su šesterokuti ravnog vrha okrenuti 0° stupnjeva, onda bi šesterokuti šiljastog vrha bili okrenuti za 30° stupnjeva što možemo vidjeti na slici 4.1. Za ovaj rad koristit će se šesterokuti ravnog vrha.



Slika 4.1: Prikaz šesterokuta ravnog vrha (lijevo) i šiljastog vrha (desno) [12]

Što se načina slaganja ovih šesterokuta u koordinatnu rešetku tiče koristit će se neparni-q način koordinata pomaka u kojem su neparni stupci, označeni sa q, pomaknuti prema dolje kao što se može vidjeti na slici 4.2.



Slika 4.2: Neparni-q vertikalni raspored koordinata [12]

Nakon što je definiran način rasporeda koordinata, može se zapisati kod GetPositionFromCoordinate funkcije koja će koordinate pretvoriti u 2D poziciju u sceni.

```
public static Vector3 GetPositionFromCoordinate(Vector2Int coordinate)
{
    int column = coordinate.x;
    int row = coordinate.y;

    float size = 72f;
    float horizontalDistance = size * (0.75f);
    float verticalDistance = size;

    bool shouldOffset = (column % 2) == 1;
    float offset = (shouldOffset) ? size / 2f : 0;

    float xposition = (column * horizontalDistance);
    float yposition = (row * verticalDistance) + offset;

    return new Vector3(xposition, -yposition, 0);
}
```

Primjer koda 4.1: Funkcija za pretvaranje koordinata šesterokuta u neparnu-q poziciju unutar 2D prostora

4.3 Objekti Polja

S gore opisanim kodom jednostavno može biti napraviti borbena rešetka proizvoljne veličine, sastavljena samo od kolekcije zadanog osnovnog polja koje će u ovom slučaju biti polje zelene trave. Umjesto direktnog referenciranja GreenGrass objekta ili pravljenja posebne skripte GreenGrass.cs instanciranje polja će se odvojiti razinom apstrakcije preko apstraktne klase Tile.cs. Ta klasa će predstavljati osnovnu funkcionalnost objekata polja i olakšati implementaciju svih raznih tipova polja i generaliziranje istih pod sličnim kategorijama.

Svaki objekt polja će se razlikovati po nekoliko vrijednosti i parametara kao što su tip polja, obitelj polja, koordinate, susjedna polja, naziv, opis polja, cijena kretanja po polju, prohodnost polja, lista osnovnih slika polja, lista vjerojatnosti pojavljivanja osnovnih slika polja i lista zadanih slika granica polja. Za lakše identificiranje i kodiranje polja definirati će se dva enum tipa podataka koji će predstavljati granice i obitelj polja. Ovo je prikazano na isječku koda 4.2.

```
public enum border
{
    North, NorthEast, SouthEast, South, SouthWest, NorthWest
};
public enum family
{
    DeepWater, ShallowWater, CoastalReef, Swamp,
    Grassland, Dirt, Road, Sand, Forest, Hills,
    Mountains, Castle
};
```

Primjer koda 4.2: Enum tip podataka za granicu i obitelj polja

Na navedeni način svako će polje ima listu slika koje će predstavljati osnovni izgled polja u situacijama kada jedan tip polja ima više slika. Pretvarajući listu vjerojatnosti pojavljivanja slika polja u težinsku vjerojatnost može se upravljati nasumičnim odabirom osnovnih slika i time izbjeći problem jednakog izgleda istog tipa polja. Svako polje također ima barem jedan tip graničnih slika koje predstavljaju slike prijelaza između svojeg tipa ili obitelji i tipa ili obitelji susjednog polja. U slučaju da jedna obitelj polja ima puno više kolekcija graničnih slika ili svoja unikatna pravila i funkcionalnosti iskoristiti će se obiteljske skripte koje nasljeđuju Tile.cs skriptu i uvode svoja pravila, logiku, funkcije i unikatne atribute. Za primjer obiteljskog nasljednika koji u svrhu dodavanja nasljeđuje više graničnih slika može se koristiti GrasslandTile.cs skripta koja mora nadjačati metodu GetBorderSprites iz osnovne klase sa svojom logikom.

S obzirom na to da su slike polja istog tipa, a ponekada i iste obitelji napravljene s mogućnošću uklapanja iz jedne uz drugu, funkcija GetBorderSprites može vratiti praznu listu što označava nepostojanje graničnih slika između tih polja.

Također, u slučajevima kada neki tipovi ili obitelji polja dijele iste granične slike prema svim drugim tipovima ili obiteljima onda se preko statičnih funkcija mogu dohvatiti prethodno spremljene kolekcije tih graničnih slika i izbjeći problem opetovanja unutar koda ili Unity preglednika.

Primjer takvog tipa bi bili odnosi između suhih polja(engl. *DryTiles*) i planinskih polja gdje više obitelji može imati njihovu suhu varijaciju gdje bi dodavanje istih graničnih slika na svaku od obiteljskih pod klasa bilo nepotrebno, neefikasno i zamarajuće. Način odabira graničnih slika je najjednostavnije vidjeti na primjeru nadjačane GetBorderSprites metode kod GrasslandTile.cs skripte prikazanom na isječku koda 4.3.

```
Public override List<Sprite> GetBorderSprites(Tile neighbour)
{
    if (neighbour.tileName == this.tileName)
    {
        return new List<Sprite>();
    }
    else if (TileUtilities.IsMountainTile(neighbour) &&
this.tileName.ContainsInsensitive(“dry“))
    {
        return TileUtilities.GetDryToMountainBorderTiles();
    }
    else
    {
        return neighbour.tileFamily switch
        {
            family.DeepWater => waterBorderSprites,
            family.ShallowWater => waterBorderSprites,
            family.CoastalReef => waterBorderSprites,
            family.Swamp => new List<Sprite>(),
            family.Grassland => longBorderSprites,
            family.Dirt => mediumLongSprites,
            family.Road => normalBorderSprites,
            family.Sand => mediumLongSprites,
            family.Forest => new List<Sprite>(),
            family.Hills => new List<Sprite>(),
            family.Mountains => new List<Sprite>(),
            family.Castle => new List<Sprite>(),
            _ => new List<Sprite>(),
        };
    }
}
```

Primjer koda 4.3: Način odabira graničnih slika kod polja obitelji *Grassland*

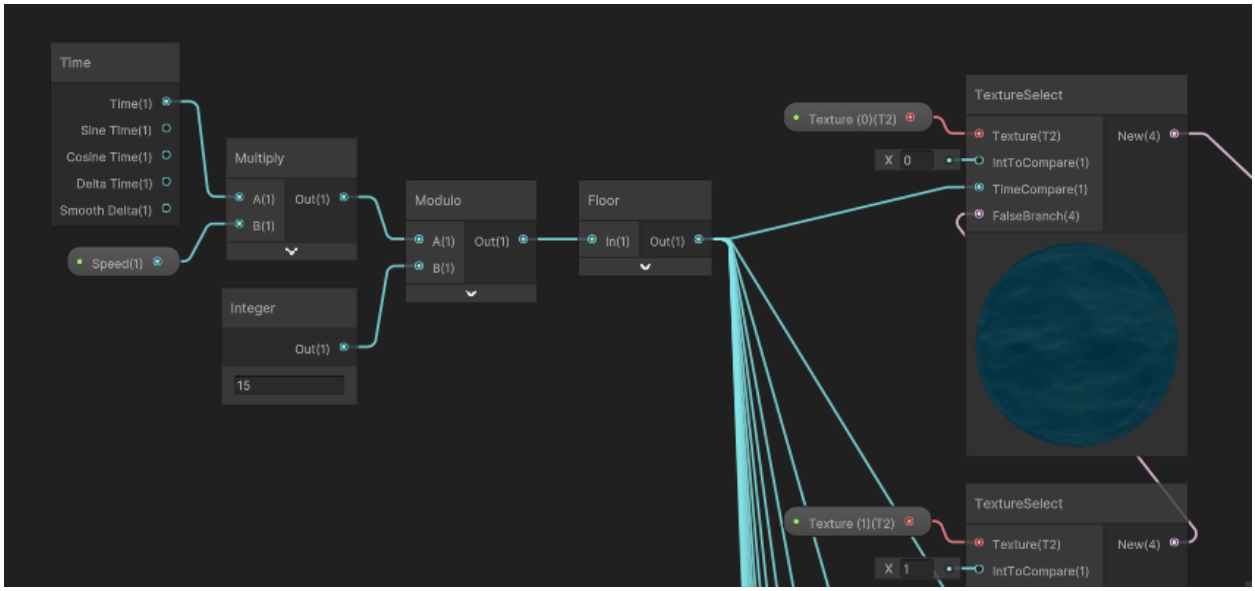
Obitelji polja koje se izdvajaju su vodene obitelji *DeepWater* i *ShallowWater* koje predstavljaju duboka i plitka polja te planinsku obitelj kojoj se izgled mijenja ovisno o uzorcima susjednih planinskih polja.

Kod vodenih obitelji korištena je kombinacija dviju tehnika koje su napravljene koristeći shadere unutar prilagođenog *Unity Shader Graph*a. Shaderi su tip skripti koji određuju način prikazivanja materijala i površina objekata, uključujući boje, teksture i osvjetljenje. *Unity Shader Graph* je poseban pristup kodiranja shadera u kojem se preko vizualnog načina spajanja blokova izbjegava ručno kodiranje. Dvije tehnike koje će biti spojene unutar *Shader Graph*a su tehnika svjetski poravnate teksture i animacije slike po slici.

Tehnika svjetski poravnatih tekstura odnosi se na izradu materijala shadera koji koristi svjetske ili scenske koordinate umjesto lokalnih UV koordinata objekta na kojem je postavljen. Ova metoda omogućava pomicanje tekstura prema globalnoj vrijednosti [4]. Na taj način, tekstura se ponavlja konzistentno u svijetu beskonačno puta na istim globalnim ili svjetskim koordinatama, bez obzira na poziciju ili orijentaciju objekta na koji je primijenjena. Ova tehnika se široko koristi u različitim

vrstama i žanrovima igara za stvaranje iluzije bešavne teksture koja ostaje dosljedna bez obzira na poziciju ili rotaciju objekta.

Druga tehnika koja se koristi je animacija slike po slici unutar shadera, koja koristi globalno vrijeme za stvaranje animacija. Ova tehnika dijeli globalno vrijeme u segmente prema broju slika animacije te pomoću if-else blokova, dodjeljuje odgovarajuću sliku iz svake slike u svakom segmentu. Rezultat je animacija koja izgleda kao da se stranice knjige okreću. Brzina izvođenja ove animacije kontrolira se vanjskom vrijednošću brzine, koja množi globalno vrijeme prije nego se uđe u if-else blok. Način implementiranja ove metode prikazan je na slici 4.3.

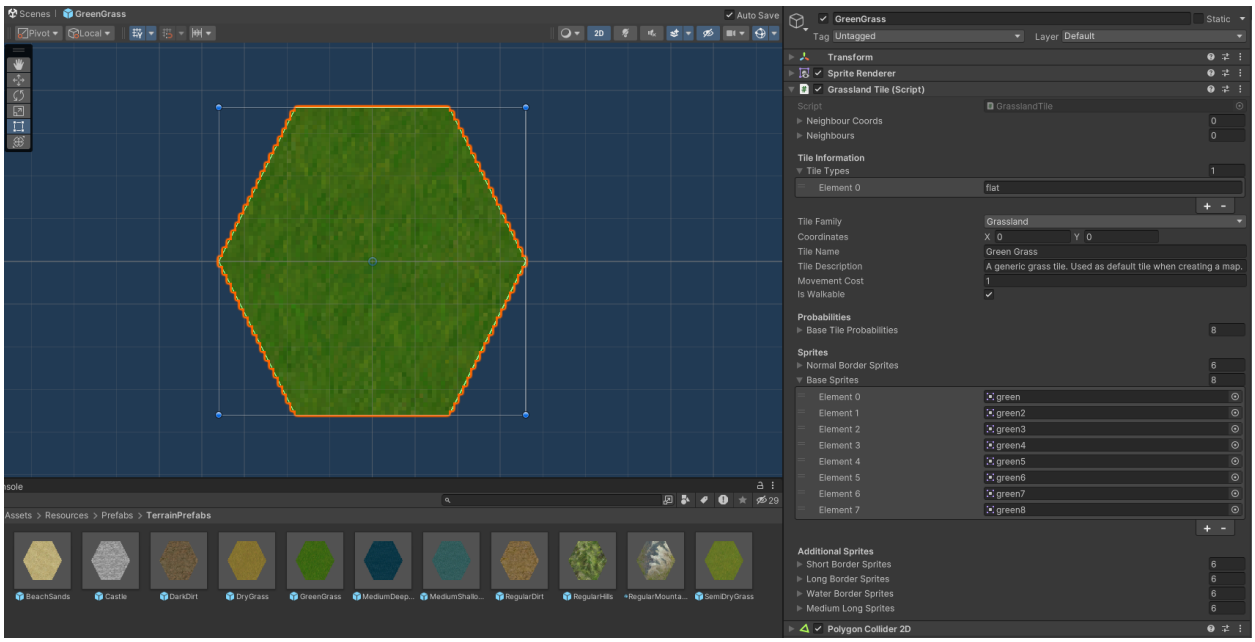


Slika 4.3: Isječak *Shader Graph* logike za tehniku animacije slike po slici

Unity Game Engine olakšava pravljenje kopija objekata putem prefab sistema. Ovaj sistem je ključan za efikasno upravljanje i ponovno korištenje objekata unutar igre. Prefab predstavlja unaprijed definirani objekt koji može sadržavati kompletnu hijerarhiju objekata, komponente, skripte i druge resurse korištene unutar igre. Korištenjem prefaba, moguće je jednostavno napraviti i upravljati višestrukim instancama objekata, što doprinosi organiziranju i optimiziranju *Unity* rada.

Za potrebe ove aplikacije, napravljeni su prefab objekti koji predstavljaju svaki tip ili različite vrste terena. Ovakvim pristupom osigurana je jednostavna upotreba tih objekata tijekom proceduralnog stvaranja i crtanja terena. Umjesto stalnog kopiranja ili ponovnog pravljenja identičnih polja, koristeći spremljene prefab objekte, moguće je jednostavno instancirati polje željenog terena preko referenci. Ovaj postupak je sličan tehnici u *Unity*u gdje se za pohranu čestih podataka koriste skriptabilni objekti (engl. *Scriptable Objects*). Međutim, umjesto samo spremanja informacija,

ovaj pristup omogućava spremanje i izgleda objekta te dodavanje dodatnih skripti koje bi se inače trebale naknadno dodati kreiranom ili instanciranom objektu.



Slika 4.4: Prikaz prefab objekata terenskih polja

4.4 Polja ciljeva

Svaka strateška igra zahtjeva ciljeve na borbenoj karti koji će igrače potaknuti na preuzimanje rizika, reformiranje strategije i mijenjanje planova svojom vrijednošću ili vrijednošću resursa, pogodnosti i prednosti koje pružaju. U većini RTS igara ovi ciljevi su predstavljeni kao dva tipa po načinu na koji ih igrači dobiju; konstruirani i osvojivi.

Konstruirani ciljevi su često prikazani kao unikatne građevine koje igrači mogu napraviti jednom ili više puta, ovisno o njihovoj važnosti i resursima koje pružaju. Igre kao *Civilization* serijal koriste jednokratne konstruirane ciljevi u formi svjetskih čuda ili povijesnih znamenitih građevina koje samo jedan igrač u igri može izgraditi i na taj način dodaju dodatni sloj strategije i odlučivanja. Konstruirani ciljevi nakon njihove izgradnje također mogu postati osvojivi ciljevi te je u većini strateških igara takav slučaj.

Osvojivi ciljevi su napravljeni na borbenoj karti na početku igre i/ili tijekom igre od strane igrača. Svi ciljevi mogu imati više načina dobivanja prihoda, ali u većini slučajeva ti načini su podijeljeni prema načinu na koji igračima daju resurse; jednokratni, višekratni i pasivni. Jednokratni ciljevi igraču koji ih je izgradio pružaju resurse ili pogodnosti samo na trenutku njihovog nastanka, nakon

toga su beskorisni. Višekratni ciljevi se mogu više puta unaprijediti, izgraditi ili aktivirati te tijekom svake od tih aktivacija ponovno pružiti igraču resurse ili pogodnosti. Obilježava ih mogućnost osvajanja od drugih igrača te služe kao glavna tema strategije jer njihovo osvajanje ojačava igrača, a oslabljuje protivnika ili protivnike. Najčešći tip osvojivih ciljeva su pasivnog tipa, koji svaki određeni interval vremena pružaju igraču resurse ili koji samim njihovim postojanjem osnaživanje korisnika na neki način, bilo to osnaživanje njihovih jedinica, povećanje generiranih resursa, oslabljivanje protivnika na neki način ili mijenjanje borbene karte u igračevu korist. Primjeri ovakvih ciljeva su gradovi, sela, kule i slične zgrade koje svaki vremenski interval osvajaču daju konstantnu opskrbu resursa.

Pošto za potrebe ove aplikacije nisu potrebni svi strateški elementi, dovoljno je pravljenje najčešćeg oblika cilja, pasivno osvojivog cilja. U ovoj aplikaciji taj cilj se predstavlja poljima sela čije osvajanje igraču daje resurse za kupovinu jedinica nakon svakog određenog vremenskog intervala. Implementiranje ovog tipa cilja zahtijeva mogućnost postavljanja sela na različita polja, pripadnost različitim igračima i osvajanje na način stajanja jedinice igrača na slobodno polje.

Kako bi se ispunio zahtjev postavljanja navedenih sela na određena polja potrebno je definirati unikatan tip polja za stvaranje koja se mogu postaviti na već postojeća polja kao ukras. Iz tog razloga napravljena je apstraktna klasa ne samo da predstavlja ukrasno polje nego sadrži i funkcionalnost koju proširuje na polje na koje je postavljena. Preko koda 4.4 riješen je problem pravljenja i postavljanja apstraktnog ukrasnog polja te se ta funkcionalnost može prenijeti na bilo koju klasu koja je implementira.

```
protected virtual bool CanBePlacedOn(Tile tile)
{
    if (tile.flair != null && tile.flair.name == name + "(Clone)") return false;
    return !blacklistedTiles.Contains(tile.tileFamily) && tile.unit == null;
}
public virtual Flair PlaceOn(Vector2Int coords, string sortingLayerName = "Flairs")
{
    if (HexGridLayout.Instance.map.TryGetValue(coords, out Tile tile))
    {
        if (!CanBePlacedOn(tile)) return null;
        return CreateFlairOn(tile, sortingLayerName);
    }
    else
    {
        HexGridLayout.Instance.CreateTileAt(coords, defaultTile);
        return CreateFlairOn(HexGridLayout.Instance.map[coords], sortingLayerName);
    }
}
```

Primjer koda 4.4: Metode za postavljanje ukrasnog polja

U prethodnom potpoglavlju navedena je Unity metoda skriptabilnih objekata koja služi za spremanje vrijednosti i jednostavnih funkcionalnosti preko posebnih skriptabilnih objekata koji drže podatke i definicije stavki. Oni su idealni za pohranu konfiguracijskih podataka, postavki igri, definicija stavki, karakteristika likova i drugih vrsta podataka koji ne trebaju biti povezani s određenim objektom u sceni. Pomažu u smanjenju redundantnosti, poboljšavaju organizaciju koda i podataka te olakšavaju upravljanje i ažuriranje globalnih podataka na jednom mjestu.

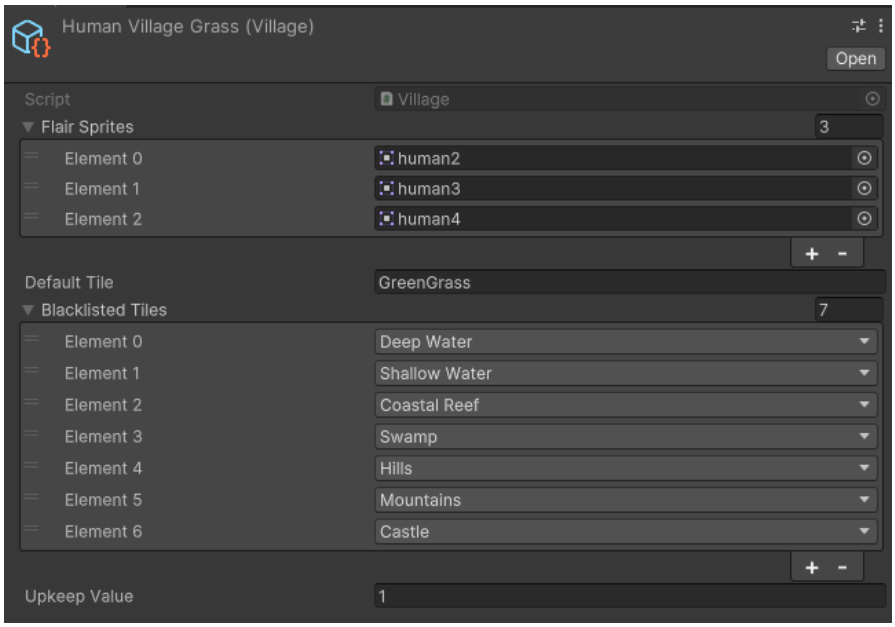
S obzirom na to da polje na koje se želi postaviti ovaj ukras već postoji, nije potrebno stvoriti posebni prefab ukrasnog polja za svaku moguću kombinaciju. Moguće je ukrasna polja napraviti kao jednostavne skriptabilne objekte i time olakšati njihovo stvaranje. Na isječku koda 4.5 može se vidjeti primjere skriptabilnog objekta koja predstavlja cilj tipa selo.

```
[CreateAssetMenu(fileName= "Village", menuName= "ScriptableObjects/Flair/Village", order= 1)]
public class Village : Flair
{
    [SerializeField] int upkeepValue = 1;
    GameObject flag;
    Player owner;
    public Player Owner => owner;
    protected override void OnCreated()
    {
        GameManager.Instance.villages.Add(this);
        tile.OnUnitEnter.AddListener(SetUpkeep);
        GameManager.Instance.OnTurnStart.AddListener(IncreasePoints);
    }
    private void OnDisable()
    {
        if(GameManager.Instance != null)
        {
            GameManager.Instance.OnTurnStart.RemoveListener(IncreasePoints);
            tile.OnUnitEnter.RemoveListener(SetUpkeep);
        }
    }
    private void SetUpkeep(Unit unit)
    {
        if(owner != unit.Player)
        {
            if (owner) owner.villages.Remove(this);
            owner = unit.Player;
            owner.villages.Add(this);
            SetPlayerFlag(owner.PlayerColor);
        }
    }
    private void IncreasePoints()
    {
        if(owner) owner.gold += upkeepValue;
    }
}
```

Primjer koda 4.5: Isječak koda ukrasnog sela sa svojim najvažnijim funkcijama

Iz navedenog isječka koda uočljivo je kako skriptabilni objekti za lakše pravljenje imaju posebnu *CreateAssetMenu* upravljačku crtu koja im olakšava izradu direktno iz Unity sučelja. Preko

UnityEvent funkcionalnosti se na svaki ulaz jedinice u polje na kojem se nalazi cilj sela poziva funkcija *SetUpkeep* koja igrača čija je jedinica ušla na polje mijenja u vlasnika sela. Time se ispunio uvjet osvajanja polja od strane različitih igrača. Pri dnu isječka koda 4.5 može se dodatno i primijetiti funkcija za povećanje resursa koju skripta menadžera igre poziva svaki određeni vremenski interval. Izgled ukrasnog polja sela se također može vidjeti na slici 4.5.



Slika 4.5: Prikaz skriptabilnog objekta ukrasnog sela

Za označavanje vlasnika sela koristi se prefab objekt zastavice koja se instancira samo u situacijama kada igrača osvoji selo čime zastavica poprimi boje osvajača. Ova zastavica koristi najjednostavniji oblik Unity *FlipBook* animacije koja na svakom određenom ključnom kadru (engl. *Keyframe*) mijenja cijelu sliku zastavice na potpuno drugu sliku zastavice, u sitnom pomaku u odnosu na prethodnu sliku zastavice. Time se stvara iluzija kretanje zastavice i kontrole piksel slike dok se u stvarnosti cijela slika zapravo veoma brzo mijenja više puta u sekundi.

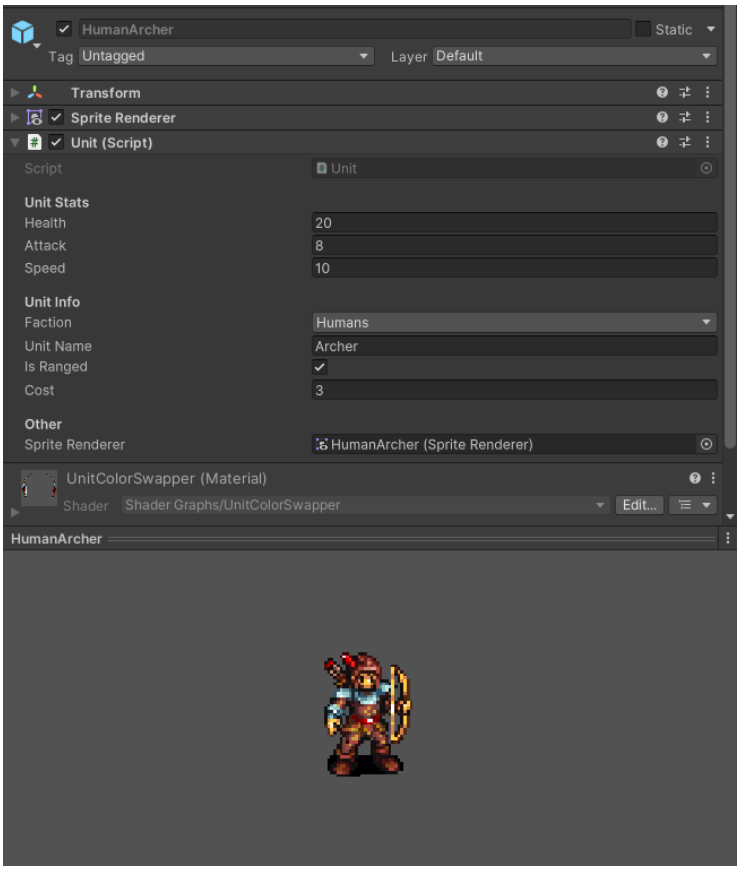
4.5 Objekti jedinica

Menadžment, upravljanje i planiranje akcija borbenih jedinica su glavni aspekti RTS igara koji predstavljaju najveći stupanj strategije. Neki od ključnih razloga zašto su RTS igre poput StarCrafta postale iznimno popularne u eSports okruženju su brzina i težina upravljanja borbenim jedinicama za efikasno izvođenje strateškog plana. Različitost, prednosti i mane ovih jedinica daju RTS igrama dodatni sloj odlučivanja te produbljuju strategiju i ideju igre. Borbene jedinice su temelj svake vojske i svaka ima ključnu ulogu u dinamici bojnog polja. Raznolikost jedinica, od

brzih izviđača do moćnih tenkova, za postizanje pobjede zahtijeva od igrača balansiranje napada, odbranu i upravljanje resursima. Navedene jedinice se uobičajeno dijele na stvarne i fantastične. U grupu stvarnih ubrajaju se vojnici, tenkovi, kopljanici i slično, dok grupu fantastičnih čine čudnovate rase i bića poput zmajeva, goblina, kostura i vanzemaljaca.

S obzirom na to da borbene jedinice predstavljaju glavni uvjet kontrole strategije u RTS igrama, prilagođena RTS aplikacija bi također trebala imati ove jedinice. Prema prethodno opisanoj važnosti borbenih jedinica i njihovim specifičnim razlikama mogu se definirati zahtjevi koje bi jedinice prilagođene RTS aplikacije trebale ispuniti.

Ove jedinice se trebaju razlikovati po prednostima i nedostacima , a prema tome i cijeni kupovanja, treniranja ili izgradnje. Kako bi se ove jedinice lakše napravile ponovno će se koristiti i skriptabilni i prefab objekti. Preko prefab objekata se mogu izraditi pojedinačne jedinice koje će se razlikovati po izgledu, atributima, imenu, cijeni izgradnje, načinu napadanja, frakciji kojoj pripadaju te boji. Na slici 4.6 se može vidjeti *Unit* skripta koja upravlja kontrolom, prednostima i nedostacima pojedine jedinice. Važno je napomenuti kako su sve slike jedinica korištene u ovom radu preuzete s javne kolekcije grafičkih resursa upotrjebljenih u igri *Battle for Wesnoth*, uz službeno odobrenje.



Slika 4.6: Prefab jedinice ljudskog strijelca

Metode za pomak jedinice na drugo polje, akciju izvođenja napada protivničke jedinice, primanja štete od protivničke jedinice, umiranja jedinice i čekanja jedinice za njen sljedeći potez su najvažnije metode unutar navedene skripte. U uobičajenim RTS igrama borbene jedinice se mogu kretati i djelovati bez ikakvog čekanja između svojih pokreta ili akcija. To uvodi element brzine i stupanj vještine u uobičajeno statičke RTS igre. Poteškoću implementiranja navedenog načina kontrole jedinica u ovoj prilagođenoj RTS igri predstavlja brzina izvođenja akcije koja ovisi o ljudskoj vještini, što ju čini suvišnom za ocjenjivanje rada algoritma AI protivnika. Zbog spomenutog razloga, bit će implementirana tehnika čekanja jedinice koja joj ograničava poteze na vremenske periode. Na opisan se način jedinici može dodijeliti atribut brzine koji joj prilagođava vrijeme čekanja između akcija. Metoda koja upravlja opisanim ponašanjem se nalazi unutar skripte za ponašanje jedinica i prikazana je na isječku koda 4.6.

```
IEnumerator UnitCooldown()
{
    canAct = false;
    spriteRenderer.color = Color.gray;
    float time = 0;
    float duration = 100;
    while (time < duration)
    {
        time += Time.deltaTime * (speed / unitTile.movementCost);
        yield return null;
    }
    canAct = true;
    spriteRenderer.color = Color.white;
    yield return null;
}
```

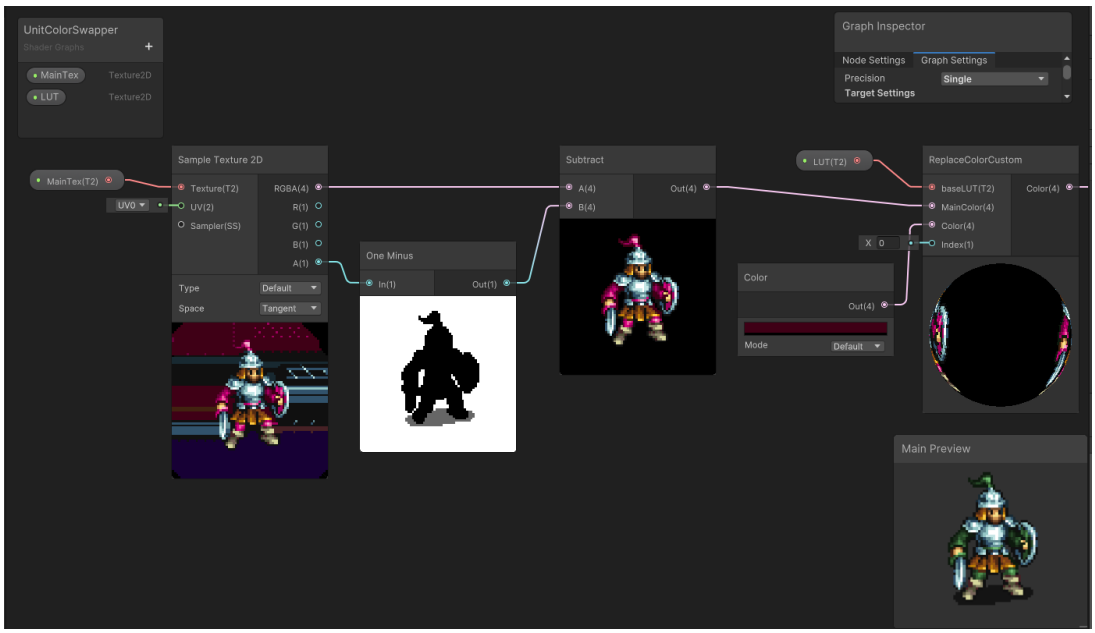
Primjer koda 4.6: Metoda za ograničavanje brzine kretnje jedinice

Uporabom metode skriptabilnih objekata može se napraviti skriptabilni objekt frakcije koji sadržava i pruža informacije o frakciji te navodi i opisuje jedinice unutar nje. Na navedeni se način olakšava pridavanje jedinica igraču, ovisno o odabranoj frakciji. Skriptabilni objekt frakcije sadržava sve ključne podatke o frakciji, poput specifičnih sposobnosti, prednosti i slabosti svake jedinice, kao i njihove uloge unutar vojske. Metoda skriptabilnih objekata omogućava jednostavnije upravljanje i prilagodbu jedinica, unaprjeđujući balans igre i potrebno strateško planiranje. Osim navedenog, pojednostavljuje razvoj i proširenje igre dodavanjem novih frakcija i jedinica bez potrebe za opsežnim promjenama u kodu, čineći proces razvoja učinkovitim i fleksibilnim.

Razlikovnost jednog tipa jedinica od drugog je veoma važna, ali je bitna i razlika između jedinica po igračima. Za takve potrebe korisno je iskoristiti jednostavno uočljivu vizualnu promjenu izgleda jedinica kao što je na primjer njihova boja. U aplikaciji će biti iskorištena posebna tehnika

izmjena paleta (engl. *Palette Swapping*). Navedena tehnika se koristi za promjenu boje cijele teksture pomoću unaprijed definiranih paleta. Svakom pikselu u teksturi dodijeljen je indeks koji upućuje na određenu boju u primarnoj ili temeljnoj paleti. Temeljna paleta se sastoji od nijansi pažljivo odabrane boje koja se ne ponavlja ni u jednoj teksturi. Umjesto stvaranja izmijenjenih slika za sve teksture sa svakom mogućom kombinacijom svih paleta u igri, sve se jedinice mogu modificirati samo za odabranu temeljnu paletu. Označavanjem specifičnih piksela u teksturi s bojama temeljne palete osigurava se promjena boja piksela na boje druge palete tijekom zamjene igračeve palete. Na takav se način mijenjaju samo željeni dijelovi originalne slike ili teksture. Kako bi se dodatno olakšao proces ove izmjene, *Unity Game Engine* omogućava izradu shader grafa koji ove izmjene odrađuje u stvarnom vremenu tijekom trajanja igre.

Primjenom navedene tehnike, razvojni timovi mogu izbjeći potrebu za stvaranjem veće količine varijacija istih tekstura u različitim bojama. Time se smanjuje količina potrebne memorije, a pojednostavljuje proces izrade i održavanja grafičkih resursa. Preuzimajući zadatak izmjene boje, Shader omogućava jednostavno prilagođavanje vizualnog identiteta jedinica bez potrebe za ponovnim crtanjem ili prilagođavanjem osnovnih tekstura. Shader dinamički mijenja paletu boja u stvarnom vremenu, osiguravajući brzu i jednostavnu promjenu izgleda jedinice. Opisana tehnika je bila popularna u starijim igrama s ograničenim hardverskim resursima, ali se zbog učinkovitosti koristi i danas . Na slici 4.7 je prikazan rad shadera koji izmjenjuje nijanse temeljne palete u nijanse bilo koje druge palete.



Slika 4.7: Prikaz rada *UnitColorSwapper* shadera

Na primjeru sa slike je korištena izmjena u blago zelenu paletu. Ovaj shader radi tako što temeljnu paletu postavi na jednodimenzionalnu crtu i pomjeranjem X vrijednosti UVa bira svaku pojedinačnu boju palete te ju zamijeni odgovarajućom bojom željene palete. Tijekom instanciranja prefab objekta borbene jedinice željena paleta u shaderu se zamjenjuje paletom vlasnika jedinice.

U svrhu upravljanja jedinicom, potrebne su nove skripte koje će omogućiti igraču vizualizaciju mogućih akcija i kretnji jedinica pomoću izbornika. Iako navedeno nije neophodno AI protivnicima, u svrhu ispravnog testiranja nužno je implementirati funkcionalnosti preko kojih će se moći provjeriti ispravnost AI poteza prema postavljenim pravilima. Skripta *UnitController* će brinuti o instanciranju pokazivača dozvoljenih akcija borbene jedinice te odabira istih. Jedinice će se moći kretati samo jedno polje, specifično na jedno od svojih susjednih polja koje nije zauzeto i koje je prohodno. U situaciji kada je jedno od tih polja zauzeto od strane protivničke jedinice pomoćni će pokazivač promijeniti svoj izgled kako bi pojasnio izvršavanje akcije napada protivničke jedinice umjesto akcije kretanja do odabranog polja. Navedena funkcionalnost je prikazana na isječku koda 4.7.

```
private void DrawAvailableActionsVisual()
{
    var tiles = selectedTile.neighbouringTiles.Values;
    foreach(var tile in tiles)
    {
        if(tile.unit == null)
        {
            if (tile.isWalkable)
            {
                CreateVisual(moveActionVisual, tile);
            }
            else
            {
                CreateVisual(invalidActionVisual, tile);
            }
        }
        else if(tile.unit.Player.TeamID != player.TeamID){
            CreateVisual(attackActionVisual, tile);
        }
        else
        {
            CreateVisual(invalidActionVisual, tile);
        }
    }
    CreateVisual(selectedUnitActionVisual, selectedTile);
}
```

Primjer koda 4.7: Isječak *UnitController* klase koji kontrolira vizualizaciju akcija jedinice

5. IMPLEMENTACIJA AI ALGORITAMA

Nakon prethodno detaljne razrade RTS igre, u ovom poglavlju će se usmjeriti na implementiranje sofisticiranih algoritama umjetne inteligencije koji će kontrolirati ponašanje AI protivnika unutar razvijene, prilagođene RTS aplikacije. Implementacija algoritama AI protivnika predstavlja korak prema ostvarivanju živopisne i izazovne strategije, pružajući igračima dinamično iskustvo s protivnicima koji donose raznovrsne strategijske odluke. Svrha AI protivnika u RTS igrama je nadvladati poteze stvarne osobe i potaknuti igrača na strateško razmišljanje. S obzirom na to da su strateške igre uvjetovane utjecajem više igrača, omogućavanjem postavljanja jednog ili više AI protivnika se smanjuje potreba za više ljudskih igrača. Navedeno je korisno i u situacijama potrebe uštede resursa ili pravljenja igre za jednog igrača koji će umjesto stvarnim osobama biti izazvan AI protivnicima. U modernom dobu, gdje tržištem dominiraju online igre koje zahtijevaju stalnu internetsku vezu, interes prema igrama za jednog igrača je sve veća. S time proporcionalno raste i potražnja za AI protivnicima te igrama s manje ograničenja.

U ovom poglavlju će biti istražena primjena i implementacija različitih algoritamskih pristupa u stvaranju AI protivnika. Usmjerit će se na dva glavna pristupa; Algoritam stabla odlučivanja (engl. *Decision Tree Algorithm*, DT algoritam) i algoritam stabla ponašanja (engl. *Behaviour Tree Algorithm*, BT algoritam). Oba pristupa se široko koriste u industriji video igara zbog svoje sposobnosti strukturiranja i organiziranja odluka AI protivnika, a osobito zbog prilagodljivog implementiranja u dinamičnim RTS okruženjima.

Kako bi se ovi algoritmi uopće mogli implementirati potrebno je napraviti skripte koje će svi algoritmi AI protivnika koristiti za slanje svojih naredbi i upravljanje jedinicama. Logika AI protivnika se razdvaja na kontrolu kupovanja jedinica, odnosno njihovu izgradnju ili regrutiranje te kontrolu upravljanja, kretnji i borbenih akcija. Zbog poštivanja SOLID načela navedena logika se mora izdvojiti u zajedničko sučelje (engl. *Interface*) ili apstraktnu klasu što je prikazano u kodu 5.1. SOLID načela predstavljaju skup smjernica za pisanje dobro strukturiranog i održivog koda u objektno orijentiranim programskim jezicima. Primjenom navedenih načela izbjegavaju se loše prakse i olakšava održavanje, proširenje i testiranje softverskih sustava.

```
public abstract class Logic : MonoBehaviour
{
    [HideInInspector] public ArtificialIntelligence AI { get; set; }
    public virtual void Initialize(ArtificialIntelligence ai)
    {
        AI = ai;
        StopAllCoroutines();
        StartCoroutine(StartLogicLoop());
    }
    private void OnDisable()
    {
        StopAllCoroutines();
    }
    IEnumerator StartLogicLoop()
    {
        while (true)
        {
            yield return new WaitUntil(() => GameManager.Instance.Mode == GameMode.Battle);
            if (AI)
            {
                yield return new WaitForSeconds(AI.ActionDelayTime);
                LogicLoop();
            }
            yield return null;
        }
    }
    protected abstract void LogicLoop();
}
```

Primjer koda 5.1: Apstraktna klasa za kontrolu logike AI protivnika

Za razliku od sučelja, apstraktna klasa pruža mogućnost programerima da, uz prazne ili virtualne metode, sadrži i definirane, odnosno implementirane metode što znatno smanjuje količinu koda u izvedenim konkretnim klasama.

U kodu 5.1 može se vidjeti kako je opisana glavna logika ponašanje AI algoritama. Pozivanjem inicijalizacijske metode pokreće se asinkrona petlja koja kontrolira kada i koje akcije će AI protivnik odraditi. *Unity Game Engine* za ostvarivanje asinkronosti u kodu koristi korutine koje omogućavaju raspoređivanje zadataka na nekoliko okvira. One mogu pauzirati izvršenje i vratiti kontrolu Unityu, a zatim nastaviti gdje su stale na sljedećem okviru. Međutim, važno je napomenuti da korutine nisu niti (engl. *Threads*) te ih je najbolje koristiti kod dugih asinkronih operacija [13]. Prethodno navedena metoda ponavlja svoj asinkroni dio, a pauzira ga na dva mjesta ovisno o postavljenim uvjetima. Prvi uvjet provjerava prijelaz igre iz faze gradnje borbene karte u fazu borbe, a drugi uvjet čeka određeno vrijeme umjetnog razmišljanja koje je postavljeno na specifičnom AI algoritmu kako bi prevladalo nedostatak predugog ljudskog procesa razmišljanja. O razlogu za ovo i detaljnijem objašnjenu bit će govora u kasnijem dijelu rada.

Jedina apstraktna metoda u ovom kodu je *LogicLoop()* metoda koju će svaka implementacija ove klase ispuniti sa svojom željenom funkcionalnošću kao što je prikazano na isječcima kodova 5.2 i 5.3.

```
public class RecruitLogic : Logic
{
    private PlayerCastle castle;
    private void Awake()
    {
        castle = GetComponent<PlayerCastle>();
    }
    protected override void LogicLoop()
    {
        Unit unitToPurchase = AI.GetUnitToPurchase(castle);
        if (unitToPurchase)
        {
            PurchaseUnit(unitToPurchase, castle.Tile);
        }
    }
    void PurchaseUnit(Unit unit, Tile tile)
    {
        castle.Player.gold -= unit.Cost;
        Unit createdUnit = UnitCreator.Instance.CreateUnit(castle.Player, unit, tile);
        UnitLogic logic = createdUnit.AddComponent<UnitLogic>();
        logic.Initialize(AI);
    }
}
```

Primjer koda 5.2: Klasa za kontrolu logike kupovanja jedinica

Iz koda 5.2 može se vidjeti princip kupovanja borbenih jedinica. Metoda *LogicLoop()* poziva borbenu jedinicu od klase AI koju je umjetna inteligencija odredila za kupnju i ako je ta jedinica dostupna, metoda je kupuje smanjujući resurse igrača i generirajući prefab objekt borbene jedinice. Ova metoda ne sadrži očekivane provjera poput provjere ima li korisnik dovoljno resursa za kupnju jedinice ili smije li je postaviti na željeno mjesto zato što su te provjere zadatak umjetne inteligencije koja je proslijeđena ovoj logici. Pri dnu koda može se vidjeti kako se jedinici kupljenoj od strane AI protivnika pridaje komponenta logike borbene jedinice zato jer ju kontrolira umjetna inteligencija.

```
public class UnitLogic : Logic
{
...
    protected override void LogicLoop()
    {
        if (!unit.CanAct) return;
        UnitAction action = AI.GetUnitAction(unit);
        switch (action.Type)
        {
            case UnitAction.ActionType.MOVE:
                MoveTowards(action.Tile);
                break;
            case UnitAction.ActionType.ATTACK:
                Attack(action.Unit);
                break;
        }
    }
    private void MoveTowards(Tile target)
    {
        if (Pathfinder.TryFindPath(unit.UnitTile, target, out List<Tile> path, out int _))
        {
            Tile tileToMoveTo = path.FirstOrDefault();
            if (tileToMoveTo && !tileToMoveTo.unit && tileToMoveTo.isWalkable)
            {
                unit.MoveUnit(tileToMoveTo);
            }
        }
    }
...
}
```

Primjer koda 5.3: Klasa za kontrolu logike upravljanja borbenih jedinica

Slično kao i kod koda 5.2, klasa s koda 5.3 kojom je opisana kontrola logike upravljanja borbenih jedinica od pridijeljene umjetne inteligencije, dobiva podatke o vrsti akcije koju će borbena jedinica napraviti. U slučaju dobivene akcije, klasa s koda 5.3 pruža i dodatne informacije o toj akciji. Akcija za napad druge jedinice je sama po sebi dovoljno jasna, ali kod akcije pomicanja borbene jedinice se javljaju poteškoće koje će biti pojašnjene i riješene u jednom od narednih poglavlja rada.

S ciljem omogućavanja pomaka jedinica potrebno je iskoristiti jedan od algoritama pronalaženja putova (engl. *Pathfinding Algorithms*) te ga implementirati u aplikaciju. Za potrebe ove aplikacije korišten je A* (A-zvijezda) algoritam. A* je algoritam pretraživanja i planiranja puta koji se koristi za pronalaženje najkraćeg puta između dvije točke u mreži ili grafu. Cijenjen je zbog svoje cjelovitosti, učinkovitosti i optimalnosti. A* algoritam koristi heuristiku koja procjenjuje udaljenost između trenutne točke i cilja. Svakom čvoru dodjeljuje dvije vrijednosti: *g(n)* koja predstavlja trošak putovanja od početnog čvora do trenutnog čvora, i *h(n)* koja predstavlja procijenjenu udaljenost od trenutnog čvora do ciljnog čvora. Kombiniranjem ovih vrijednosti, A* određuje prioritet čvorova za istraživanje. Čvorovi s nižim vrijednostima imaju prednost

istraživanja u odnosu na one s višim vrijednostima. U praksi, A* koristi otvorenu i zatvorenu listu. Otvorena lista sadrži čvorove koji još nisu istraženi, dok zatvorena lista sadrži čvorove koji su već prošli kroz istraživanje. Kada istražuje čvor, A* izračunava sve moguće susjedne čvorove i ažurira njihove *g(n)* i *h(n)* vrijednosti. Ako je novi put do nekog čvora kraći od prethodnog, A* ažurira vrijednosti i dodaje čvor na otvorenu listu. Kada A* pronađe ciljni čvor, rekonstruira put od početnog do ciljnog čvora prateći prethodne čvorove. Zahvaljujući efikasnom korištenju heuristike, prednost A* algoritma leži u njegovoj sposobnosti pružanja optimalnog rješenja za probleme planiranja puta uz relativno malu računalnu kompleksnost.

U prethodno opisanim kodovima koristi se objekt klase AI koji predstavlja umjetnu inteligenciju. Navedena klasa služi kao sloj apstrakcije, isto kao i kod logike AI algoritma, čime omogućuje pristup različitim ponašanjima algoritama AI protivnika na identičan način.

Posljednji korak u pripremi aplikacije za implementaciju algoritama umjetne inteligencije je zajednička apstraktna klasa kojom se objedinjuje objekt protivničke umjetne inteligencije. Zajedničke funkcionalnosti implementiranih algoritama umjetne inteligencije grupiraju se ovom klasom u svrhu održavanja čistog koda, lakšeg pristupa od strane drugih skripti i klasa te postavljanja razine apstrakcije za svaki od algoritama. Apstrakcijom klasa za logiku kupovina i upravljanja jedinice omogućava se jednostavan i identičan način pristupa rezultatima željenih algoritama. Svim algoritmima protivničke umjetne inteligencije ključne su određene pomoćne metode kojima se statički izvršava odabrana funkcionalnost ispunjavanja zadataka i donošenja odluka, odnosno rezultata. Zbog navedenog se te metode nalaze unutar apstraktne klase.

Pomoćne metode imaju istu ulogu pružanja pomoći za odluke i dolazak do rješenja protivničkom AI algoritmu, a mogu se razlikovati po svom načinu rada. Podjela je obavljena u dvije kategorije: dohvatne metode i notifikacijske metode. U dohvatne su metode uvrštene metode koje kroz svoje kalkulacije vraćaju i dohvaćaju tražene vrijednosti. Takve se metode dodatno mogu podijeliti na one koje dohvaćaju vrijednost varijable, na one koje svojim lokalnim vrijednostima izračunavaju i vraćaju vrijednosti te na one koje primaju argumente i onda kroz njih svojim radnim sustavom filtriraju i dohvaćaju vrijednosti. Metode koje su važne za algoritme umjetne inteligencije u kontekstu RTS aplikacija uključuju: metodu za dohvaćanje najjače jedinice, metodu za dohvaćanje najbližeg dvorca, metodu za dohvaćanje svih polja na koja jedinica može putovati, metodu za dohvaćanje vrijednosti umjetnog čekanja za emulaciju ljudskog razmišljanja i slično. Za razliku od njih, notifikacijske su metode zadužene za praćenje, mijenjanje ili ispunjavanje vrijednosti ovisno o događajima te za rukovanje tim događajima. Ove metode su izuzetno važne u kontekstu RTS aplikacija zbog raznih događaja koji mogu promijeniti stanje igre. U ovoj aplikaciji jedna od

tih metoda je *OnHashSetChanged()*, koja se koristi za rukovanje događajem preuzimanja tuđe baze, u ovom slučaju dvorca. Notifikacijske metode omogućuju algoritmima umjetne inteligencije prilagodbu promjenama u stvarnom vremenu, čime se povećava njihova učinkovitost i točnost u donošenju odluka. Sve ove metode mogu se vidjeti na isječku koda u primjeru 5.4, gdje su prikazane definicije notifikacijskih i dohvatnih metoda, preko kojih se osigurava reagiranje protivničke umjetne inteligencije na promjene okruženja na odgovarajući način, čime se unaprjeđuje točnost rezultata.

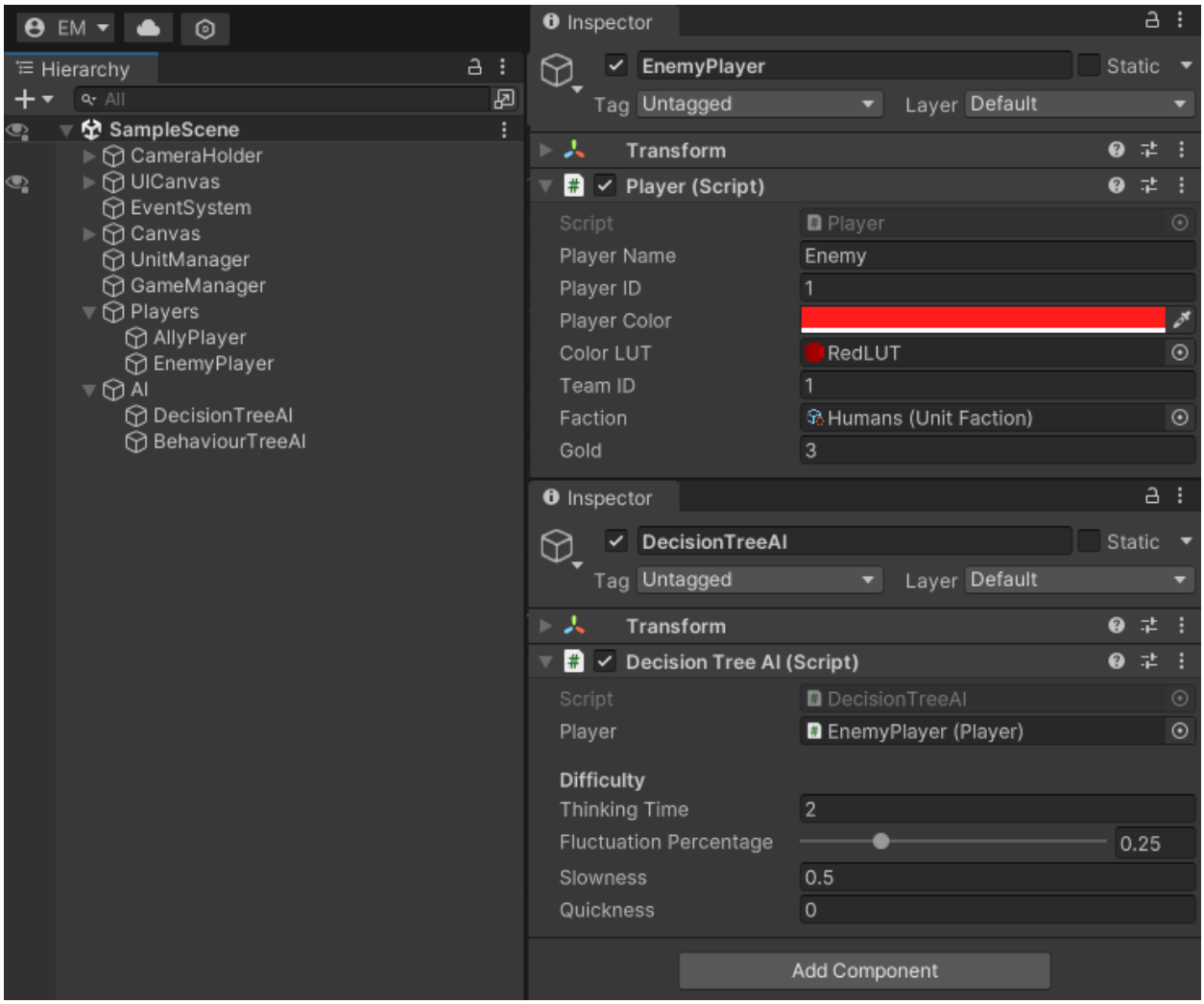
```
public bool IsBattle => gameManager.Mode == GameMode.Battle;
public float ActionDelayTime =>  Mathf.Clamp(...);
private void OnHashSetChanged(object sender, HashSetChangedEventArgs<PlayerCastle> e){...}
public Unit GetFastestUnit(List<Unit> units){...}
public Unit GetMostPowerfulUnit(List<Unit> units){...}
public Tile GetClosestTile(Tile origin, List<Tile> tiles, out int shortestDistance){...}
public Dictionary<Tile, int> GetClaimableVillageTiles(Unit unit){...}
public Dictionary<Tile, int> GetClaimableCastleTiles(Unit unit){...}
public Dictionary<Tile, int> GetTilesWithDistance(Tile origin, List<Tile> tiles){...}
```

Primjer koda 5.4: Definicije dohvatnih i notifikacijskih metoda

Od skripti za kontrolu logike upravljanja i kupovanja jedinica spomenut je način komuniciranja i dobivanja uputa od AI algoritma preko posebnih metoda koje vraćaju važne, ali iznimno pojednostavljene i kompaktne strukture koje predstavljaju rezultat odlučivanja sljedeće akcije. Navedeno obično predstavlja još jedan problem implementiranja AI algoritama u bilo koju aplikaciju, a ne samo u kontekstu RTS igre. Međutim, zbog dodatnog sloja apstrakcije uvedenog u skriptu umjetne inteligencije, ovaj problem postaje trivijalan. Korištenjem dvije glavne metode, od kojih jedna dobavlja rezultat logike kupovine jedinice, a druga rezultat borbenog upravljanja jedinicom, logički dio aplikacije dobiva sve potrebne informacije za svoj rad, neovisno o kojoj se vrsti AI protivničkog algoritma radi. Ove metode klasificirane su kao apstraktne i dodijeljena im je samo definicija, dok će se tijelo koda i način dobivanja rezultata nalaziti unutar pojedinih implementacija apstraktne klase. Kupovinom jedinica se bavi *GetUnitToPurchase()*, prva od spomenutih metoda, koja kao argument prima svojeg pozivatelja, odnosno dvorac koji želi regrutirati borbenu jedinicu. Druga metoda, čiji je zadatak usmjeravanje borbenih jedinica, zove se *GetUnitAction()*, i kao svoj argument prima borbenu jedinicu pozivača.

Kako bi implementacije skripte algoritama umjetne inteligencije bile iskorištene, potrebno ih je postaviti u hijerarhiju scene. No, da bi se ti algoritmi spojili s igračem i time stvorili AI igrača ili protivnika, potrebno je dodatno napraviti objekt igrača koji će sadržavati standardne informacije i postavke potrebne RTS aplikaciji za rad i donošenje poteza. Ova skripta je potpuno jednostavna,

sadrži samo osnovne, ali važne informacije za tijek igre. Neke od tih informacija su identifikacijski broj igrača, ime igrača, boja igrača, LUT nijansa boje za prethodno definiran shader jedinica, identifikacijski broj igračevog tima za timski oblik borbe, početni resursi igrača i skriptabilni objekt borbene frakcije igrača s informacijama o borbenim jedinicama koje igrač može kupiti ili regrutirati. Na slici 5.1 je prikazan izgled skripti kao komponenti u hijerarhiji i pregledniku.



Slika 5.1: Prikaz *Player* i *AI* skripte u hijerarhiji i pregledniku Unity editora

Uspješnost implementacije AI protivnika također se ogleda u iskustvu AI igrača. Jedno od glavnih obilježja RTS igara je brzina igrača. Uz strategiju, brzina igrača je još jedna odlika koja razlikuje iskusnog i uspješnog igrača od lošijeg. Problem koji se javlja kod AI protivnika je taj što je umjetna inteligencija u većini slučajeva više umjetna nego inteligentna. Računalo ne može napraviti grešku, ne treba mu odmor, ima mikroskopsku preciznost i savršeno prati upute. Bez ikakvih dodatnih mjera, AI protivnik je toliko dobar koliko i programer koji ga je napravio. Velika se većina igrača

ne želi boriti protiv savršenog protivnika pa je zadatak programera umjetno slabljenje AI protivnika.

Taj se zadatak može riješiti na više načina, ali najjednostavnija i za potrebe ove RTS aplikacije i analize rezultata najprikladnija metoda je usporavanje poteza AI protivnika. Kao što se može vidjeti na slici 5.1 jedini otvoreni parametri kod skripte umjetne inteligencije su parametri za modificiranje težine AI igrača preko usporavanja njegovih akcija. Formula za ovo usporavanje je jednostavna, a korištenjem nasumičnog elementa dobiva se nepredvidljiv rezultat. Formula 5.1 uzima vrijeme razmišljanja i dodaje mu nasumični broj u modificiranom rasponu ovisnom o negativnom i pozitivnom faktoru brzine te fluktuaciji.

$$f(x) = T_t + Random.Range(-T_t * Fl - T_n, \quad T_t * Fl + T_p) \qquad (5-1)$$

gdje je:

$f(x)$ – umjetno vrijeme razmišljanja

T_t – vrijeme razmišljanja

Fl – faktor fluktuacije

T_n - negativni faktor brzine

T_p – pozitivnih faktor brzine

5.1 Algoritam stabla odlučivanja

Algoritam stabla odlučivanja utemeljen je na razgranatoj strukturi odlučivanja, gdje svaki čvor predstavlja određenu odluku ili uvjet, a grane čvorova predstavljaju moguće akcije ili sljedeće odluke [14]. Ovakav pristup omogućava jasno i lagano definiranje logike odlučivanja AI protivnika na temelju trenutnog stanja igre i ciljeva protivnika. Struktura algoritma stabla odlučivanja počinje od korijenskog čvora (engl. *Root Node*) koji postavlja početno pitanje ili uvjet, a svaka grana koja izlazi iz korijena vodi do sljedećih čvorova koji predstavljaju sljedeće odluke ili uvjete na temelju odgovora [15]. Jednostavnost ovog algoritma polazi iz toga da svaka odluka ili pitanje predstavlja samo dva moguća odgovora; Da, odnosno istina, i ne, odnosno laž. Ovaj

proces se ponavlja sve dok se ne dođe do listova stabla koji predstavljaju konačne akcije AI poduzimanja.

Primjena algoritma stabla odlučivanja u RTS igrama omogućava programerima definiranje kompleksne, zamršene ili složene scenarije odlučivanja na jednostavan i pregledan način. Na primjer, AI protivnik može koristiti stablo odluka za određivanje hoće li borbena jedinica napasti, povući se, izgraditi dodatne jedinice ili prikupljati resurse, ovisno o trenutnoj situaciji na bojnom polju. Na taj način, ako je neprijateljska vojska veća od vojske AI protivnika, on može odlučiti povući svoje borbene jedinice i krenuti prikupljati resurse za jačanje svojih snaga. S druge strane, ako AI protivnik ima nadmoćnu vojsku, može odlučiti izvršiti napad. Jedna od ključnih prednosti algoritma stabla odlučivanja je njegova transparentnost i lakoća razumijevanja. Programeri i dizajneri igara mogu jednostavno i pregledno pratiti logiku odlučivanja AI protivnika pregledom stabla odluka, što olakšava dijagnostiku, analizu i ispravljanje eventualnih problema ili nelogičnosti poteza [16]. Također, algoritmom stabla odlučivanja omogućavaju se jednostavne izmjene i proširenja, budući da novi uvjeti i odluke mogu biti dodani kao dodatni čvorovi bez potrebe za rekonstrukcijom cijeloga stabla.

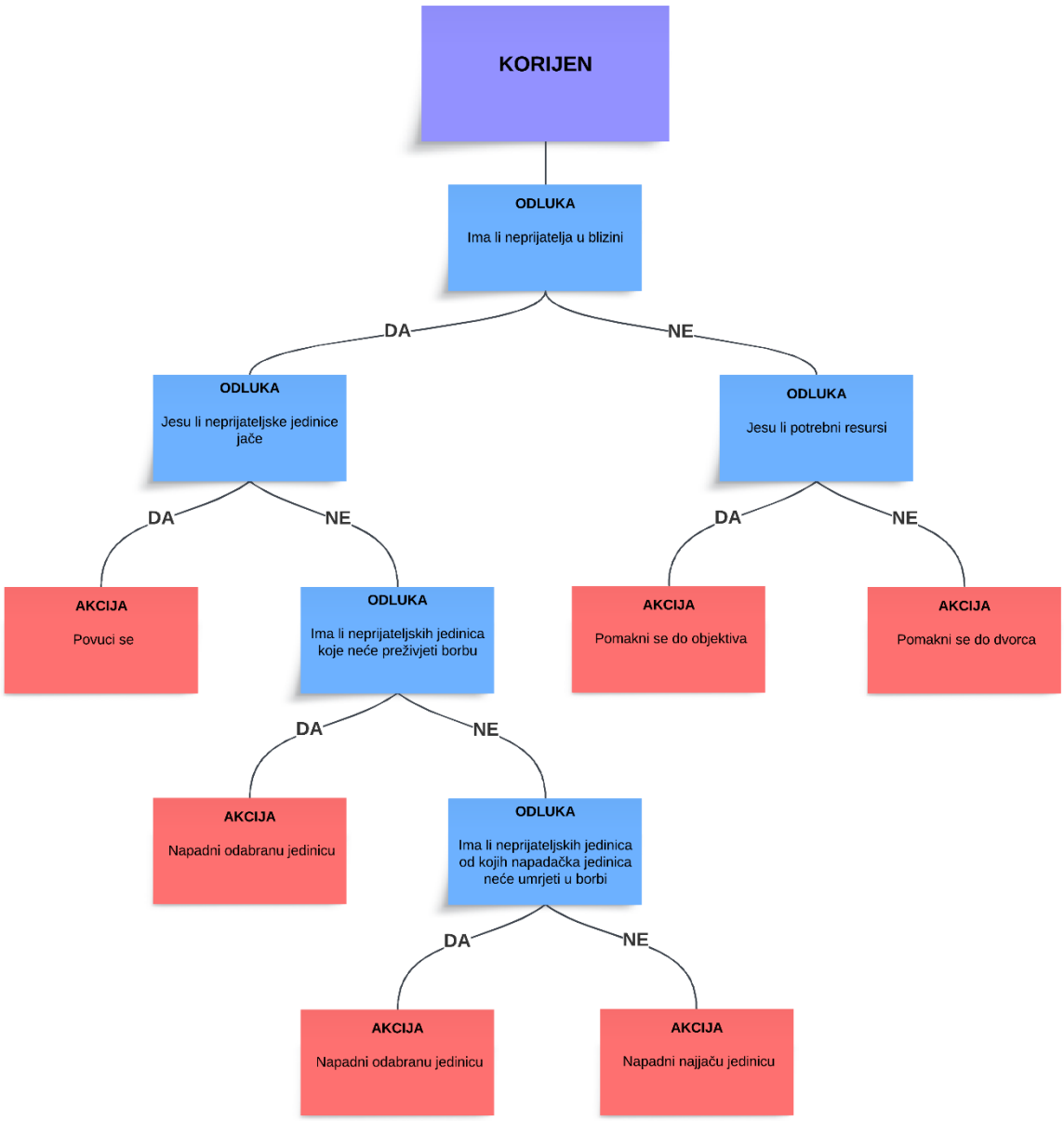
Međutim, jedan od izazova i najvećih nedostataka korištenja algoritma stabla odlučivanja njegova je potencijalna neefikasnost i neučinkovitost u vrlo složenim scenarijima gdje broj mogućih uvjeta i odluka eksponencijalno raste [17]. U takvim slučajevima, stablo može postati vrlo veliko i teško za upravljanje, što može utjecati na performanse igre, kao i na preglednost logike i jednostavnog ispravljanja potencijalnih grešaka. Unatoč tome, zbog svoje jednostavnosti i efikasnosti u donošenju odluka temeljenih na predefiniranim uvjetima, algoritam stabla odlučivanja je još uvijek jedan od najčešće korištenih pristupa u implementaciji AI protivnika za RTS igre, pod uvjetom da je pravilno dizajniran i da je vođeno računa o optimizaciji.

Koristeći prethodno objašnjene definicije, prednosti, nedostatke i izazove, mogu se definirati uvjeti i postupak rada implementacije algoritma stabla odlučivanja. Budući da RTS igre, uključujući i ovu aplikaciju, posjeduju kompleksne mehanike, praćenje više stanja, različite odluke te proširivu i izazovnu strategiju, za efikasnu implementaciju i točnije rezultate potrebno je logiku razmišljanja algoritma stabla odlučivanja podijeliti na više dijelova. Kao što je rečeno u prethodnom poglavlju, ovaj pristup je standard za kompliciranije igre i aplikacije, a strateške igre su jedne od najpoznatijih korisnika ovog pristupa. U mnogim strateškim igrama, svaki ključni dio strategije raspodijeljen je na više različitih stabala odlučivanja, pri čemu svako od njih, sa svojim postavljenim uvjetima i stanjima, donosi informacije o sljedećem potezu za taj strateški sektor, u potpunosti odvojen od ostalih strateških sektora.

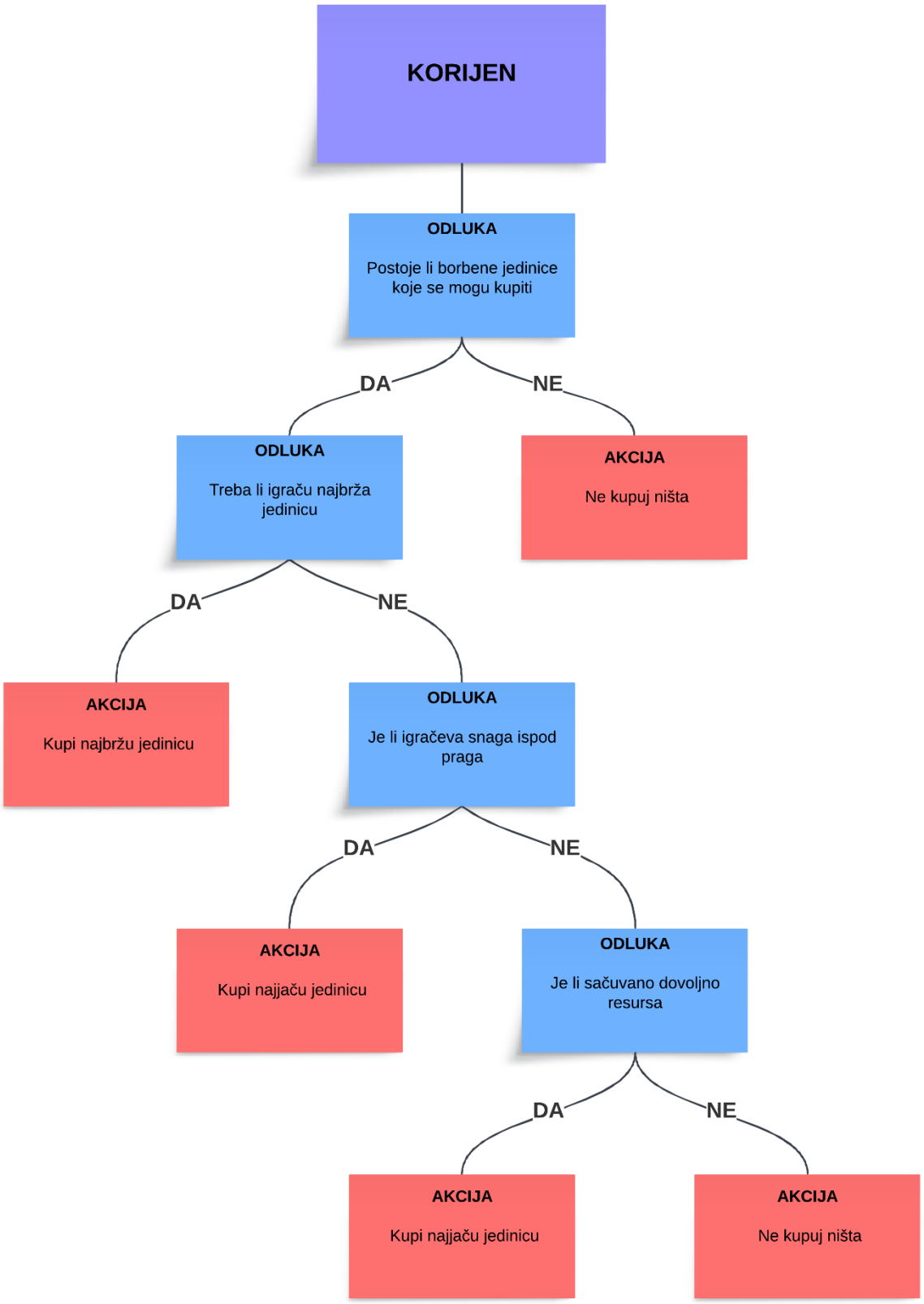
U poglavlju definicije zahtjeva za programskim rješenjem definirano je rješenje ove podjele. Najbolji način podjele logike algoritama umjetne inteligencije za aplikaciju koja može testirati ove algoritme je podjela logike na kontrolu i upravljanje borbenim jedinicama te na odlučivanje i vremensko ograničavanje kupovine borbenih jedinica. Skripte za kontrolu logike upravljanja borbenim jedinicama postavljene su na pojedinačne borbene jedinice, dok su skripte za odlučivanje o kupovini i izgradnji novih borbenih jedinica postavljene na glavne baze igrača i jedine mogu proizvoditi borbene jedinice.

U prethodnom potpoglavlju su provedene sve potrebne pripreme za algoritam stabla odlučivanja i ponašanja. Tom prilikom su, uz dodatne pomoćne skripte, pratitelje varijabli, rukovatelje događaja i brojne druge skripte i Unity objekte, napravljeni i pozivi prema osnovnoj klasi koja predstavlja algoritme umjetne inteligencije. Specifično, ti su pozivi napravljeni dvjema funkcijama. Prva funkcija prolazi kroz određeno stablo i dobiva informaciju o tome koji bi trenutni potez borbene jedinice trebao biti, dok druga funkcija također prolazi kroz svoje stablo i dobiva informaciju o kupovini jedinica za taj vremenski okvir.

Sljedeći korak je dizajniranje stabla odlučivanja prema svim navedenim informacijama. Stablo odlučivanja za algoritam stabla odlučivanja jednostavno je za izradu i nema kompleksnih elemenata koji se mogu pronaći u stablu ponašanja kod algoritma stabla ponašanja, iako su na kraju oba stabla strukturno veoma slična. Stablo odlučivanja dijeli se na grane i čvorove. Grane su predstavljene crtama koje spajaju roditeljski čvor s djecom čvorovima. Glavni uvjet stabla odlučivanja je taj da svaki unutarnji čvor mora imati točno dva čvora potomka, odnosno dječje čvorove. Prijelaz iz roditeljskog čvora u dječje čvorove preko grana predstavljen je odlukom, po čemu je ovo stablo i dobilo naziv. Svaka odluka može biti „DA“ ili „NE“, odnosno „ISTINA“ ili „LAŽ“ (engl. *TRUE/FALSE*), te odgovara na pitanje svojeg pridruženog čvora. Svaki krajnji čvor predstavlja neku akciju ili odluku, dok svaki unutarnji čvor predstavlja pitanje koje razgranava i produbljuje stablo. Na slikama 5.2 i 5.3 mogu se vidjeti strukture stabla odlučivanja koje predstavljaju logiku upravljanja borbenih jedinica i logike kupovanja borbenih jedinica napravljenih prema spomenutim pravilima.



Slika 5.2: Stablo odlučivanja logike borbene jedinice



Slika 5.3: Stablo odlučivanja logike kupovanja borbenih jedinica

Pravljenje i implementacija stabala odlučivanja nije zahtjevna, ne samo zbog mnogih napravljenih priprema u aplikaciji, već i zbog nedostatka kompleksnosti samog algoritma stabla odlučivanja i njegovog dizajna. Pripreme uključuju detaljno razrađene skripte koje su već integrirane u sustav, što značajno smanjuje potrebu za dodatnim kodiranjem i testiranjem. Skripta koja upravlja ovim stablom nasljeđuje osnovnu skriptu umjetne inteligencije i implementira obavezne apstraktne metode sa specifičnim ponašanjem prilagođenim algoritmu stabla odlučivanja. Ova skripta služi kao posrednik između logike algoritma i stvarnih akcija u igri, omogućujući donošenje odluka na temelju postavljenih uvjeta. Budući da algoritam stabla odlučivanja funkcionira na osnovu uvjeta, njegovo ponašanje se jednostavno može emulirati i implementirati u kodu koristeći C# uvjete i if-izjave (engl. *If-Statements*). U isječku koda 5.5 prikazan je dio ove implementacije, gdje se jasno vidi kako se odluke algoritma pretvaraju u konkretne akcije unutar igre. Ovaj pristup ne samo da omogućuje fleksibilnost u razvoju, već i osigurava brzo i efikasno primjenjivanje promjena u logici algoritma bez potrebe za velikim preinakama u ostatku sustava.

```
public override Unit GetUnitToPurchase(PlayerCastle castle)
{
    if (castle.PurchasableUnits.Count <= 0) return null;
    if (castle.Tile.unit) return null;

    List<Unit> purchasableUnits = castle.PurchasableUnits;
    List<Unit> playerUnits = unitManager.GetUnits(player);

    int playerUnitPower = playerUnits.Sum(u => u.Cost);

    if (TryBuyFastestUnit(purchasableUnits, playerUnits, out Unit unitToPurchase))
        return unitToPurchase;
    if (playerUnitPower < 12) return GetMostPowerfulUnit(purchasableUnits);
    else
    {
        if (goldToWaitTo <= 0)
        {
            int averageCost = (int)player.Faction.Units.Average(u => u.Cost);
            int maxCost = player.Faction.Units.Max(u => u.Cost);
            goldToWaitTo = Random.Range(averageCost, maxCost);
        }
        if (player.gold < goldToWaitTo) return null;

        goldToWaitTo = 0;
        return GetMostPowerfulUnit(purchasableUnits);
    }
}
```

Primjer koda 5.5: Metoda algoritma stabla odlučivanja za kupovanje borbenih jedinica

5.2 Algoritam stabla ponašanja

Algoritam stabla ponašanja pristupa odlučivanju AI protivnika kroz hijerarhijski raspored čvorova, gdje se svaki čvor bavi određenim ponašanjem ili zadatkom. Ovakav fleksibilan pristup

omogućava dinamičko prilagođavanje AI ponašanja ovisno o promjenjivim uvjetima u igri, čime se postiže kompleksnije i adaptivnije ponašanje protivnika. Općenito, algoritam stabla ponašanja se sastoji od različitih tipova čvorova kao što su selektori, sekvenceri, dekoratori i zadaci ili akcije. Selektori biraju prvi od svojih uspješnih podčvorova, sekvenceri redom izvršavaju svoje podčvorove dok jedan ne zakaže, odnosno vrati neuspjeh, dekoratori modificiraju ponašanje svojih podčvorova, a zadaci ili akcije predstavljaju konkretne akcije koje AI može poduzeti [18]. Stablo kod ovakvog AI algoritma izgleda izvrnuto, gdje je korijenski čvor na samom vrhu, a grane i listovi se protežu prema dnu. Algoritam stabla ponašanja je poseban po tome što pozornost treba usmjeriti na redoslijed postavljenih čvorova. Ovo je važno zbog posebnog smjera prolaska navedenog algoritma kroz svoje čvorove, od lijeva ka desno. Čvorovi koji biraju između više podčvorova ili ovise o njima, poput selektora i sekvencera, ako im se zamijeni ili pogrešno postavi redoslijed podčvorova, mogu potpuno promijeniti željeno ponašanje AI protivnika. Za razliku od algoritma stabla odlučivanja, svaki od čvorova može imati proizvoljan broj podčvorova, te oni vraćaju jedno od tri stanja: uspjeh (engl. *Success*), neuspjeh (engl. *Failure*) i izvršava se (engl. *Running*) [18].

Primjena algoritma stabla ponašanja u RTS igrama omogućava stvaranje modularnih i ponovo upotrebljivih komponenti ponašanja, što značajno olakšava razvoj i održavanje kompleksnih AI sustava. Kao primjer, jedan podčvor može biti odgovoran za slanje borbene jedinice na izviđanje područja, drugi za obranu baze, dok treći može biti zadužen za napad na neprijateljske jedinice. Ako je glavni dvorac AI protivnika pod napadom, selektor može odabrati obrambeni čvor koji će potom izvršiti niz sekvencijalnih akcija poput postavljanja straža, izgradnje obrambenih jedinica ili jedinica sa velikom efektivnošću odbrane dvorca i slanja borbenih jedinica na obranu. Neke od glavnih prednosti algoritma stabla ponašanja su njegova skalabilnost, proširivost i mogućnost ponovne upotrebe već postojećih modula ponašanja u različitim kontekstima ili situacijama. Navedeno omogućava programerima izradu složenih i prilagodljivih sustava AI protivnika bez potrebe za ponovnim pisanjem koda ispočetka ili od nule za svaki specifičan scenarij. Osim toga, hijerarhijska priroda algoritma stabla ponašanja omogućava lako razumijevanje i praćenje logike AI protivničkog ponašanja, što u nastavku olakšava dijagnosticiranje i ispravljanje potencijalnih problema.

Iako je algoritam stabla ponašanja moćan alat, njegova primjena može biti zahtjevna u pogledu inicijalne i početne postavke i dizajna. Kreiranje optimalnog rasporeda čvorova i definiranje međusobnih odnosa između njih zahtijeva pažljivo planiranje i testiranje od strane programera koji ih žele implementirati. Međutim, jednom kad je struktura uspostavljena i izgrađena, prednosti u pogledu fleksibilnosti, modularnosti i skalabilnosti često nadmašuju sve početne izazove i daju

veoma zadovoljavajuće rezultate [18]. Zbog ovih karakteristika, algoritam stabla ponašanja se široko koristi u industriji video igara za kreiranje sofisticiranih i adaptivnih AI protivnika koji pružaju dinamično i izazovno ponašanje i strategijsko odlučivanje za igrače.

Identično kao i kod algoritma stabla odlučivanja, uvjeti i zahtjevi implementacije algoritma stabla ponašanja ostaju isti. Ovaj algoritam mora na isti način raspodijeliti svoju logiku na više dijelova, što je nužno zbog složenosti strategija i odluka koje se donose u realnom vremenu u RTS igrama. Budući da su testni, mjerni i kvantitativni uvjeti identični, logika će se raspodijeliti na isti način, osiguravajući konzistentnost u ponašanju umjetne inteligencije. Ovaj pristup također osigurava da ključnim aspektima igre, poput upravljanja jedinicama, donošenja strateških odluka i prilagodbe ponašanja prema promjenama u stanju igre, pravilno integriranje u algoritam. Tako se, kroz ovu strukturu, omogućuje reagiranje umjetne inteligencije na različite situacije unutar igre na način koji je predvidiv, ali i dovoljno fleksibilan za prilagodbu na neočekivane promjene u igri. Ovo vrijedi i za sve druge uvjete i zahtjeve koje bi usporedivi algoritmi umjetne inteligencije protivnika u RTS igrama trebali ispuniti. Algoritam stabla ponašanja, baš kao i algoritam stabla odlučivanja, mora ispuniti standarde koji omogućuju učinkovito upravljanje jedinicama i donošenje odluka. Takvim se standardima osigurava preciznost u prepoznavanju uvjeta igre, brzina reakcije na promjene te sposobnost donošenja odluka u realnom vremenu. Sve ove karakteristike izrađeni algoritam stabla odlučivanja već je ispunio u prethodnom potpoglavlju, a sada se isti zahtjevi postavljaju i pred algoritam stabla ponašanja.

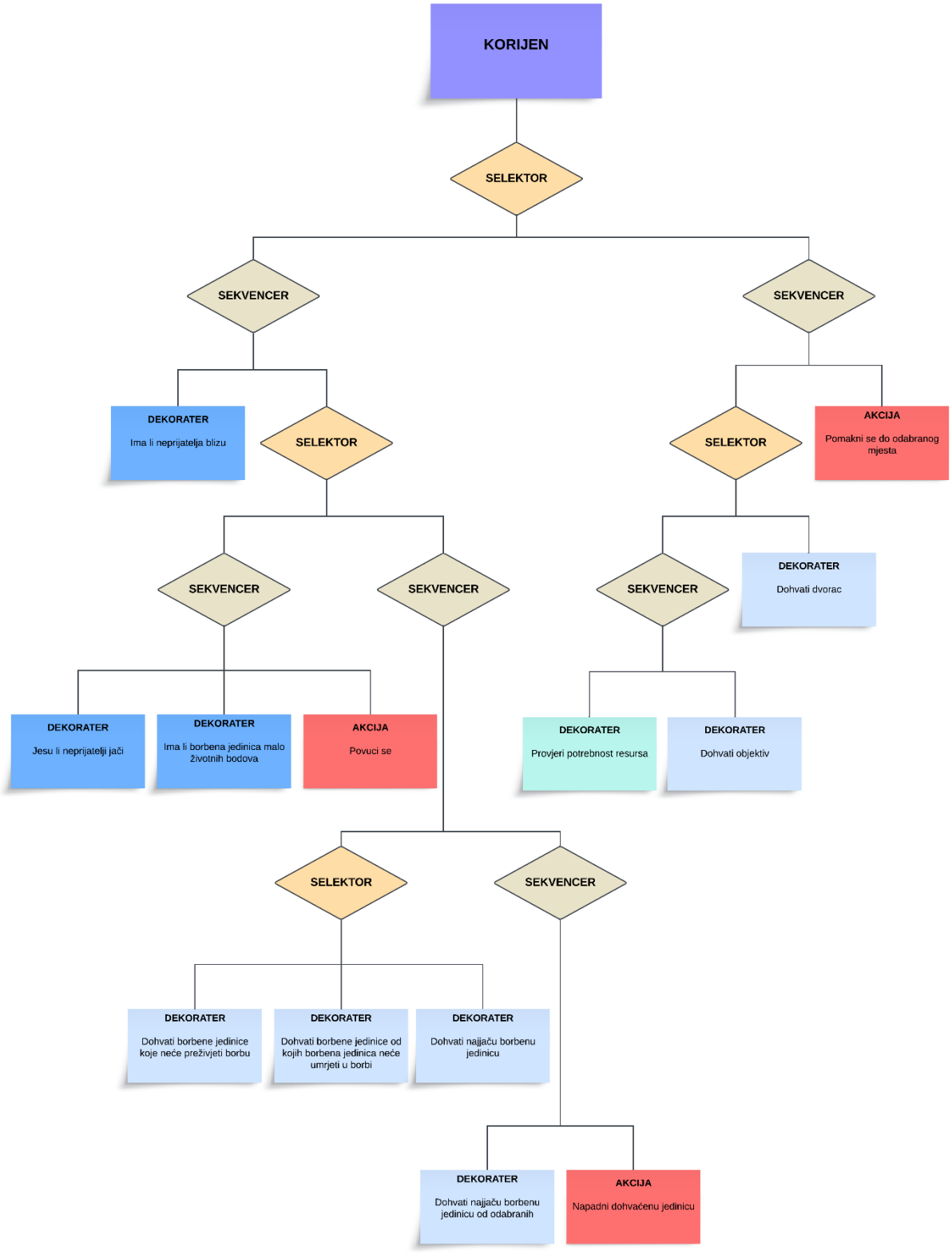
Ako bi se algoritam stabla odlučivanja predstavio kao osnovno mjerilo, algoritam stabla ponašanja bi bio njegova poboljšana, proširena i mogućnostima bogatija verzija. Većina nedostataka algoritma stabla odlučivanja su u algoritmu stabla ponašanja popravljane, zaobiđene ili ublažene. Kompleksnost algoritma stabla ponašanja dolazi iz strukture njegovog stabla. Kao i stablo odlučivanja, ono ima identične glavne elemente, čvorove i grane, ali ih proširuje i koristi na efikasniji i modificirano jednostavniji način. Temeljna i najvažnija razlika između stabala ova dva algoritma je to što stablo ponašanja ne putuje poput stabla odlučivanja samo hijerarhijski, od korijenskog čvora do krajnjih grana, nego dodatno putuje i između čvorova braće ili sestara od lijeva ka desno. Time, u odlučivanje puta odluke, kroz čvorove dodaje dodatni stupanj izražavanja.

Prethodno je istaknuto da svaki čvor posjeduje stanje izvršavanja koje određuje je li trenutno u tijeku, je li uspješno završen ili je došlo do neuspjeha. Ovo se stanje izračunava evaluacijom čvora pomoću funkcije koja se zove *Evaluate()*. Stanje grana se zatim izračunava na temelju stanja djece čvorova te se izmjenjuje na različite načine, ovisno o tipu roditeljskog čvora. Kao i kod algoritma stabla odlučivanja, svaki čvor u algoritmu stabla ponašanja ima svoje specifično stanje izvršavanja

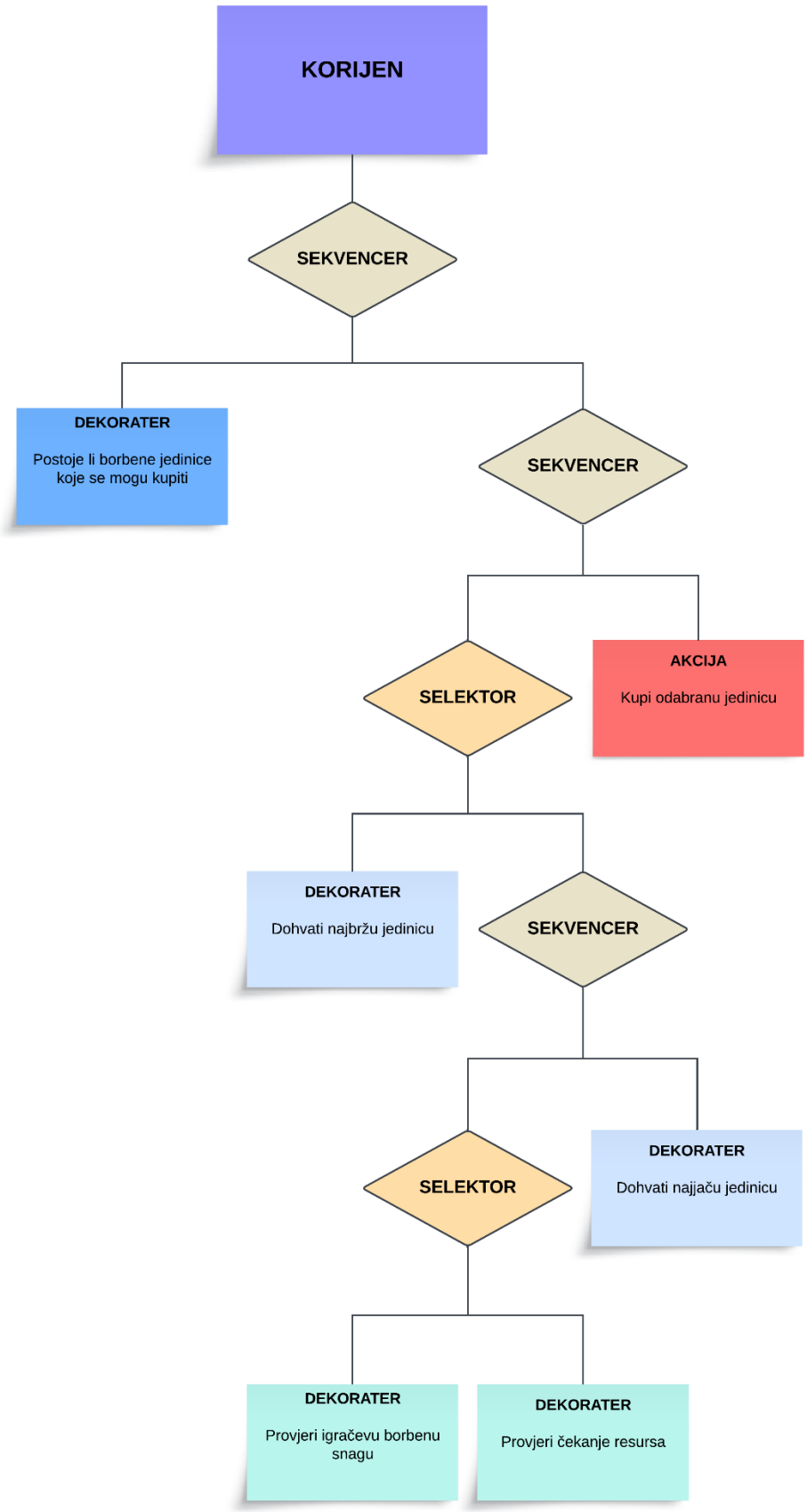
koje mora biti precizno definirano i praćeno tijekom cijelog tijeka igre. Proces evaluacije svakog čvora ključan je za donošenje ispravnih odluka u realnom vremenu, jer on određuje hoće li se određena akcija nastaviti, zaustaviti ili promijeniti smjer. Stanje grana, koje povezuje roditeljske čvorove s njihovim dječjim čvorovima, predstavlja logičku poveznicu između različitih dijelova stabla i omogućuje prenošenje evaluacije rezultata pojedinih čvorova dalje u sustav. Na taj način, algoritam može donositi kompleksne odluke, objedinjujući rezultate evaluacije iz više čvorova i prilagođavajući se promjenama u igri.

Uz navedeno, važno je napomenuti da čvorovi imaju više uloga i mogućnosti od obične "DA" ili "NE" provjere. Općenita podjela, koja se koristi i u ovom projektu, podjela je čvorova na akcije (engl. *Actions*), sekvencere (engl. *Sequencers*) i selektore ili birače (engl. *Selectors*), te izmjenjive i adaptivne dekoratere (engl. *Decorators*).

Akcije predstavljaju krajnje čvorove koji završavaju jedan ciklus prolaza kroz stablo i daju akcije, naredbe ili informacije izvan stabla. To su čvorovi koji "izvode" radnje. Sekvenceri i selektori su glavni čvorovi putovanja kroz stablo. Kako je već navedeno, uobičajeno je da čvor može imati tri stanja: uspjeh, neuspjeh i izvršava se. Također je prethodno pojašnjen i rezultat njihova pojavljivanja u čvoru. Sekvenceri i selektori to ponašanje mijenjaju tako da njihovo stanje ovisi o stanju njihove djece. Sekvenceri će vratiti uspjeh ako i samo ako sva njihova djeca također vrate uspjeh, prateći uvjete od lijeva ka desno. Selektori rade na sličan način, ali oni vraćaju uspjeh ako bar jedno njihovo dijete također vrati rezultat uspjeha. Na njih se može gledati kao na logička "I" i "ILI" vrata. Struktura stabla ponašanja obično je puna kombinacija sekvencera i selektora, koji međusobno mogu davati kompleksnije izraze nego pojedinačno. Dekorateri su najjednostavniji jer predstavljaju običan čvor sa svojim uvjetima i služe kao provjera stanja.



Slika 5.4: Stablo ponašanja logike borbene jedinice



Slika: 5.5: Stablo ponašanja logike kupovanja borbene jedinice

Zbog složenije i kompliciranije upotrebe, programska implementacija stabla ponašanja, započinje dizajnom čvora. Prvi je korak stvaranje osnovne klase čvora koja može biti proširena u složenije čvorove, poput selektora i sekvencera. Ova osnovna klasa mora biti dovoljno fleksibilna da bi mogla omogućiti različitim vrstama čvorova obavljanje svoje specifične funkcije unutar stabla. Jedna od ključnih značajki čvorova stabla ponašanja je njihova sposobnost vraćanja kroz stablo u vertikalnom smjeru prema gore i putovanja horizontalno između čvorova pomoću selektora i sekvencera. Ova dinamičnost znači da čvorovi moraju imati mogućnost prenošenja izračunatih i dohvaćenih informacija kroz cijelo stablo. To se postiže tako da svaki čvor pohranjuje sve dobivene informacije u posebnu mapu koja se nalazi na samom vrhu stabla, odnosno u korijenu stabla. Na taj način, svaki čvor u stablu može pristupiti tim informacijama i koristiti ih u procesu donošenja odluka.

Istaknuti način rada omogućuje stvaranje složenih i međusobno povezanih struktura unutar stabla ponašanja, gdje svaki čvor doprinosi općem tijeku informacija i donošenju odluka. Primjer ovakve implementacije može se vidjeti u isječku koda 5.6, gdje metode *GetData()* i *SetData()* omogućuju pristup i manipulaciju podacima pohranjenima u korijenu stabla.

```
public enum NodeState
{
    RUNNING,
    SUCCESS,
    FAILURE
}
public class Node
{
    public Node(params Node[] children)
    {
        foreach (Node child in children)
            Attach(child);
    }
    public virtual NodeState Evaluate() => NodeState.FAILURE;
    public void SetData(string key, object value)
    {
        dataContext[key] = value;
        parent?.SetData(key, value);
    }
    public object GetData(string key)
    {
        object val = null;
        if (dataContext.TryGetValue(key, out val))
            return val;
        Node node = parent;
        if (node != null)
            val = node.GetData(key);
        return val;
    }
}
```

Primjer koda 5.6: Prikaz čvor klase sa definicijom stanja čvorova

U čvor klasi iz koda 5.6 može se primijetiti virtualna metoda *Evaluate()*, preko koje će svaki zasebni čvor koji naslijedi ovu klasu definirati svoje ponašanje i način evaluacije te mjerenja specifičnih uvjeta. Najbolji primjer toga može se vidjeti u *Evaluate()* metodama sekvencer i selektor čvornih klasa na slici 5.4.

```
public override NodeState Evaluate()
{
    bool anyChildIsRunning = false;

    foreach (Node node in children)
    {
        switch (node.Evaluate())
        {
            case NodeState.FAILURE:
                state = NodeState.FAILURE;
                return state;
            case NodeState.SUCCESS:
                continue;
            case NodeState.RUNNING:
                anyChildIsRunning = true;
                continue;
            default:
                state = NodeState.SUCCESS;
                return state;
        }
    }

    state = anyChildIsRunning ? NodeState.RUNNING : NodeState.SUCCESS;
    return state;
}
```

```
public override NodeState Evaluate()
{
    foreach (Node node in children)
    {
        switch (node.Evaluate())
        {
            case NodeState.FAILURE:
                continue;
            case NodeState.SUCCESS:
                state = NodeState.SUCCESS;
                return state;
            case NodeState.RUNNING:
                state = NodeState.RUNNING;
                return state;
            default:
                continue;
        }
    }

    state = NodeState.FAILURE;
    return state;
}
```

Slika 5.4: Implementacija *Evalute()* metode kod sekvencer (lijevo) i selektor (desno) klasa

Tijekom programske gradnje stabla ponašanja, svaki čvor dekorater i čvor akcija moraju naslijediti osnovnu čvor klasu i na svoj način, koristeći tri čvorna stanja, implementirati *Evaluate()* metode prema njihovim zahtjevima. Akcije čvorovi, za razliku od dekorater čvorova, moraju na sam korijen stabla poslati konkretne i gotove informacije kako bi one mogle biti predane pozivajućem entitetu, bila to pojedinačna borbena jedinica ili dvorac koji kupuje, pravi ili regrutira jedinice.

Jedina važna informacija koju kupac jedinica, odnosno dvorac, treba kod logike za kupovanje jedinica je objekt jedinice koja će biti kupljena, pa je taj dio jednostavan za implementirati. Akcijski čvor nazad u korijen vrati samo objekt te jedinice, koji se onda koristi u ostatku koda.

Problem nastaje kod dobivanja informacija o potezu borbene jedinice. Za razliku od logike kupovanja, jedinica može imati više akcija: napad tuđeg dvorca, napad tuđeg cilja, osvajanje ciljeva, kretnja prema cilju, kretnja prema neprijateljskim jedinicama, kretnja prema neprijateljskim ciljevima, bježanje od bitke i čekanje. Zbog jednostavnosti i sličnosti većine opcija, sve mogu biti svrstane u tri opće grupe: ne radi ništa, pomakni se, napadni (engl. *Nothing, Move, Attack*). Međutim, s obzirom na širok spektar mogućnosti, izuzetno je važno dobro definirati te parametre.

Prednost se očituje u tome što se navedeni parametri mogu smanjiti na samo dva parametra: borbena jedinica i polje. Preko ove tri informacije se svakoj jedinici može zapovjediti bilo koja od gore navedenih akcija. To je stoga jer su ostale informacije suvišne. Ako je jedinici dano polje i zapovijed kretanja, ona počinje micanje prema predanom polju, ne znajući ide li tamo u svrhu osvajanja cilja, povlačenja od bitke ili približavanja protivnicima kako bi ih napala. O navedenom se brine algoritam umjetne inteligencije koji je i davatelj ovih skraćenih minimalnih informacija, spremljenih u strukturu zvanu *UnitAction* prikazanu na kodu 5.7.

```
public class UnitAction
{
    public enum ActionType
    {
        NOTHING,
        MOVE,
        ATTACK,
    }
    public ActionType Type {  get; set; }
    public Tile Tile { get; set; }
    public Unit Unit {  get; set; }

    public UnitAction(
        ActionType type = ActionType.NOTHING,
        Tile tile = null,
        Unit unit = null
    ){
        Type = type;
        Tile = tile;
        Unit = unit;
    }
}
```

Primjer koda 5.7: Struktura akcije koju jedinica treba izvršiti

Posljednji element tijekom implementiranja algoritma stabla ponašanja je izgradnja samog stabla ponašanja. Za razliku od stabla odlučivanja, ovo stablo je kompleksnije i modularnije te ga ne možemo jednostavno ubaciti u kod kao jednu cjelinu. Stoga su čvorovi napravljeni odvojeno kao zasebne klase i na sebe nadovezuju svoju djecu čvorove. Navedena modularnost omogućava lakšu izmjenu i proširenje stabla bez potrebe za potpunim restrukturiranjem. Na kodu sa slike 5.6 može se vidjeti konstruktorska metoda koja rekurzivno prima klasu čvor. Uporabom *params* formata moguće je dodati proizvoljni broj djece čvorova te na pregledniji i jednostavniji način izgraditi željeno stablo, prikazano na isječku koda 5.8. Ovime se postiže i modularnost stabla jer ga lagano možemo izmijeniti i ponovno iskoristiti izrađene čvorove.

```
private Node CreateUnitActionRootNode()
{
    return new Selector(
        new Sequence(
            new CheckEnemyInRangeNode(),
            new Selector(
                new Sequence(
                    new CheckEnemyPowerGreaterNode(),
                    new CheckHealthPercentageNode(0.3f, (x, y) => x <= y),
                    new ActionRetreat()
                ),
                new Sequence(
                    new Selector(
                        new GetWinningCombats(),
                        new GetSurvivableCombats(),
                        new GetStrongestAttackableEnemy()
                    ),
                    new Sequence(
                        new GetStrongestAttackableEnemy(),
                        new ActionAttack()
                    )
                )
            )
        ),
        new Sequence(
            new Selector(
                new Sequence(
                    new CheckGoldStatus(4, 2),
                    new GetVillage()
                ),
                new GetCastle()
            ),
            new ActionMove()
        )
    );
}
private Node CreateRecruitRootNode()
{
    return new Sequence(
        new CheckRecruitableUnits(),
        new Sequence(
            new Selector(
                new GetFastestUnit(),
                new Sequence(
                    new Selector(
                        new CheckPlayerPower(12),
                        new CheckGoldStatus()
                    ),
                    new GetStrongestRecruitableUnit()
                )
            ),
            new ActionRecruit()
        )
    );
}
```

Primjer koda 5.8: Prikaz metoda za izgradnju stabala ponašanja

Iz gornjeg se koda, kod implementiranja algoritma stabla odlučivanja, uočava jednostavnost programske izgrade stabala ponašanja i njihova modularna prednost nad stablima odlučivanja.

6. ANALIZA REZULTATA

Kroz prethodni dio rada su se definirali i implementirali algoritmi stabla odlučivanja i ponašanja za umjetnu inteligenciju u specifičnom kontekstu RTS aplikacija i igara. Navedeni algoritmi, iako poprilično stari i osnovni, služili su kao temelj dizajniranja i stvaranja gotovo svih algoritama za AI protivnike koji se koriste u današnje vrijeme. Njihova dugovječnost i utjecaj u svijetu umjetne inteligencije ukazuju na njihovu učinkovitost i prilagodljivost različitim izazovima u igrama. Kako bi se provjerila uspješnost ovih algoritama te kako bi se njihovi rezultati mogli analizirati i međusobno usporediti, potrebno je testirati ih u različitim situacijama i scenarijima. RTS aplikacija, kreirana za potrebe ovog rada omogućava stvaranje borbenih karti i postavljanje više igrača u borbu, pri čemu svi igrači mogu biti kontrolirani od strane umjetne inteligencije. Konkretno, to su algoritmi AI protivnika implementirani u ovom radu.

Za ostvarivanje što veće točnosti i preciznosti rezultata potrebno je definirati jasne kriterije ocjenjivanja i scenarije kroz koje će se moći bolje vidjeti, zabilježiti i analizirati rezultati implementacije algoritama stabla odlučivanja i stabla ponašanja. Navedeni scenariji će osigurati da testovi obuhvate sve aspekte igre, od strategije akcija do reakcija na neočekivane događaje, čime će se dobiti sveobuhvatan uvid u performanse svakog algoritma. Na tablici 6.1 definirani su različiti uvjeti i uspjesi kao i bodovi kojima se ukazuje na njihovo prisustvo ili uspješno izvršavanje. Neki od navedenih uvjeta poput manjeg broja okolnih ciljeva, udaljenih ciljeva, manje ili veće količine početnih resursa, nepovoljnih pozicija i slično ukazuju na prednost danu jednom algoritmu. Svaki od ovih uvjeta donosi određen broj bodova, što omogućuje matematičku analizu i određivanje uspješnijeg algoritma zadanog scenarija. Na taj će način biti moguće objektivno procijeniti efikasnost svakog algoritma, čime se pružaju značajni uvidi za buduća unaprjeđenja i prilagodbe te jasnije prikazivanje općenitih nedostataka svakog algoritma i njihovog implementiranja kao rezultata pogrešno postavljenih pravila ponašanja i odluka ili sustavnih problema.

MJERILA	BODOVI
Pobjeda	+15
Trajanje borbene runde kao pobjednik	-1/min preko 5 min
Početna pozicija jednostavna za obraniti	-3
Početna pozicija zahtjevna za obraniti	+3
Dobiveni početni resursi	-1/resursu
Okolni ciljevi	-2/cilju
Udaljeni okolni ciljevi	+4
Dobro izvođenje povlačenja	+5
Postavljanje prioriteta na ciljeva	+4
Značajno kretanje borbenih jedinica	+5
Efikasno kupovanje borbenih jedinica	+7
Ispravno biranje borbenih akcija	+7

Tablica 6.1: Uvjeti i zahtjevi ocjenjivanja performansi AI algoritama.

Glavno mjerilo ocjenjivanja tijekom testiranja AI algoritama predstavlja sama pobjeda jer ona pridaje najviše bodova te je primarni cilj svakog AI protivnika upravo postizanje pobjede u borbi. Pobjeda je ultimativni pokazatelj uspješnosti, i stoga se na nju stavlja najveći naglasak. Međutim, osim pobjede, postoje i drugi čimbenici koji doprinose ukupnoj ocjeni AI protivnika. Ovi čimbenici predstavljaju sekundarne ciljeve koje AI protivnici moraju ostvariti tijekom borbe, a ključni su za demonstriranje njihove sposobnosti strateškog razmišljanja i prilagođavanja promjenjivim uvjetima u borbenoj rundi. Sekundarni ciljevi uključuju specifične zadatke i zahtjeve koje AI mora ispuniti, a to su uspješno upravljanje resursima, efikasno korištenje jedinica, zauzimanje ciljeva na mapi i slično. Ovi ciljevi su važni jer odražavaju sposobnost AI protivnika u nošenju s kompleksnim situacijama i dinamičnoj promjeni planova kako bi povećao svoje mogućnosti za pobjedu. Uspješno ispunjavanje ovih ciljeva također se nagrađuje bodovima, no u nešto manjoj mjeri u odnosu na pobjedu.

Iz tablice 6.1 može se uočiti veća podjela bodova zahtjevima koji su direktno povezani s funkcionalnostima implementiranih algoritama umjetne inteligencije. Ovakva raspodjela bodova omogućava detaljniju analizu i ocjenu performansi svakog AI protivnika. Na taj način, AI protivnici ne samo da mogu biti uspoređeni s drugim AI algoritmima, već se može procijeniti i učinkovitost same implementacije pojedinog AI algoritma. Tako se pruža sveobuhvatan uvid u to kako dobro određeni algoritam funkcionira u praksi i gdje postoje mogućnosti za dodatna poboljšanja. Ovakva metoda ocjenjivanja i raspodjela bodova omogućava preciznije razumijevanje kvaliteta i slabosti AI protivnika, čime se osiguravaju sveobuhvatne i za potrebe rješavanja zadatka ovog rada relevantne evaluacije.

Slika 6.1: Borbena karta sa jednakim početnim uvjetima za oba igrača

Na slici 6.1 se također može vidjeti izrađena borbena karta za potrebe testiranja algoritama gdje svaki algoritam ima jednake početne uvjete. Napravljena je i druga borbena karta u kojoj jedna strana ima nepovoljan položaj u usporedbi sa drugom stranom. U slučaju kada se nalazi u lošijoj poziciji, ta druga karta će se koristiti za potrebe detaljnijeg ocjenjivanja AI algoritama.

Za bolje razumijevanje načina ocjenjivanja, potrebno je promotriti jednu borbu između AI protivnika. U tu će se svrhu testiranje izvesti između algoritma stabla odlučivanja i algoritma stabla ponašanja u situaciji kada su oba algoritma u neutralnoj poziciji. Na navedeni način moći će se istaknuti akcije koje oba algoritma uzimaju u raznim situacijama i istim pridijeliti bodovi. Na slici 6.1 je prikaz neutralne borbene karte u kojoj svaki AI algoritam ima identične početne uvjete pa

im nisu pridijeljeni bodovi vezani za nepovoljnu ili povoljnu poziciju niti bodovi za različite borbene resurse i okolne ciljeve. Početak borbe se može vidjeti na slikama 6.2 i 6.3 gdje se prikazuje donošenje prvih odluka AI algoritama.

Slika 6.2: Početne odluke algoritma stabla ponašanja

Slika 6.3: Početne odluke algoritma stabla odlučivanja

Iz navedenih se slika uočava postavljanje prioriteta AI algoritma na okolne ciljeve, zbog čega grade svoje najbrže jedinice te ih usmjeravaju prema najbližem cilju. Opisano ponašanje je primjer dobro implementiranog određivanja prioriteta na ciljeve i efikasnog kupovanja borbenih jedinica. Prema tablici ocjenjivanja performansi AI algoritama, oba AI algoritma dobivaju po 11 bodova.

Nakon nekoliko minuta borbe, jedinice AI algoritama osvajaju sve okolne ciljeve te počinju kretnju ka protivniku, što je prikazano na slici 6.4.

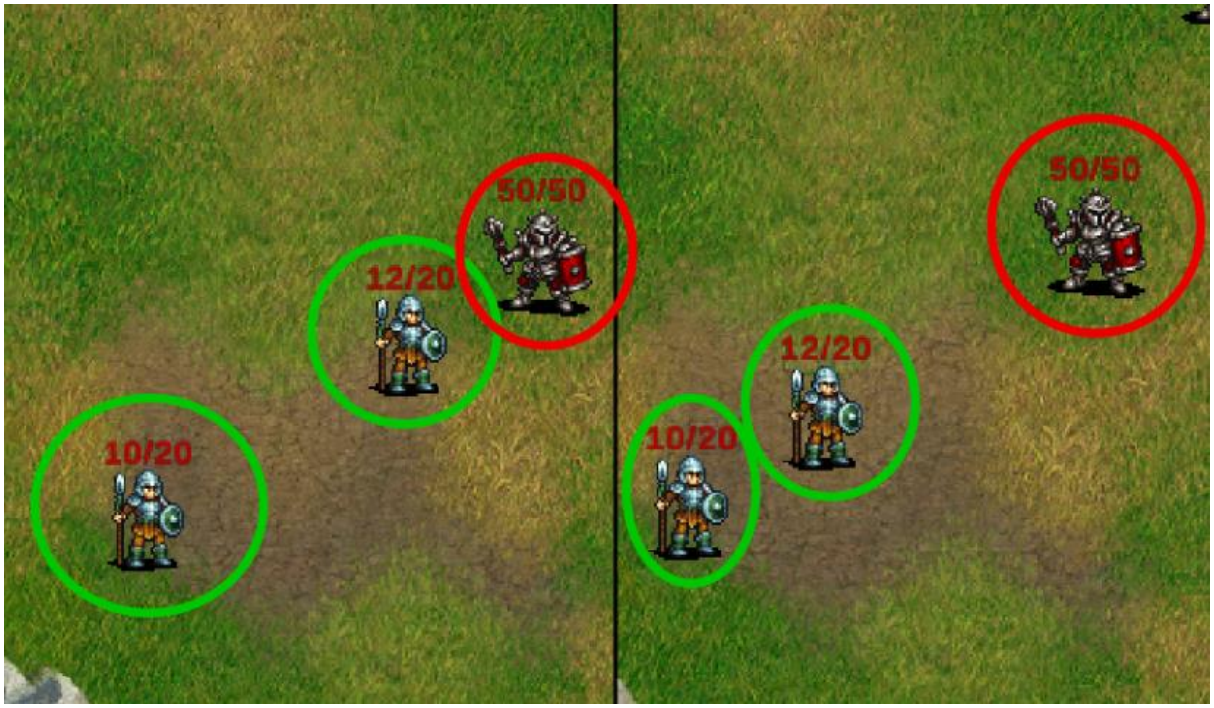
Slika: 6.4: Prikaz kretnje borbenih jedinica jednog AI protivnika ka drugome

Na slici 6.5 se uočava funkcionalnost promjene ponašanja AI algoritma na osnovu borbenog stanja što je jedna od prednosti algoritma stabla ponašanja.

Slika 6.5: Prikaz promjene ponašanja algoritma stabla ponašanja tijekom igre

Na istaknutoj se slici uočava način na koji jedinice algoritma stabla odlučivanja (prikazane crvenom bojom) znatno nadmašuju jedinice algoritma stabla ponašanja (prikazane zelenom bojom). Iz navedenog razloga, algoritam stabla ponašanja mijenja svoju strategiju i ulaže resurse u brže, ali slabije, jedinice kako bi ih što žurnije spojio sa jedinicama kojima je potrebna podrška. Navedeno se ponašanje ubraja u grupu ispravnog odabira borbenih akcija jedinica. To je ponašanje koje algoritam stabla odlučivanja ne može ostvariti, a tu algoritam stabla ponašanja skuplja 7 bodova.

U situaciji kada je jedna borbena jedinica ozlijeđena u borbi protiv znatno jače jedinice, algoritam stabla ponašanja ponovno iskazuje svoju efikasnost, što je prikazano na slici 6.6.



Slika 6.6: Prikaz funkcionalnosti povlačenja jedinica algoritma stabla ponašanja

Na navedenoj se slici također zapaža odluka povlačenja borbene jedinice algoritma stabla ponašanja nakon što je ozlijeđena u borbi protiv daleko jače jedinice. Također, borbena jedinica se povlači prema okolnim prijateljskim jedinicama. Opisanim su predstavljena mjerila dobrog izvođenja povlačenja i značajnog kretanja borbenih jedinica. Time algoritam stabla ponašanja skuplja 12 bodova. Algoritam stabla odlučivanja također izvršava akcije povlačenja, ali za razliku od algoritma stabla ponašanja, to obavlja nasumično i što udaljenije od protivnika, čime ponekad skuplja 5 bodova.

Točno 16 minuta i 48 sekundi, nakon početka borbe, jedinice algoritma stabla ponašanja okružuju te zauzimaju dvorac algoritma stabla odlučivanja, čineći algoritam stabla ponašanja pobjednikom ove runde, što se može vidjeti na slici 6.7.

Slika 6.7: Algoritam stabla ponašanja zauzima dvorac algoritma stabla odlučivanja

Prednosti algoritma stabla ponašanja su prevelike u odnosu na algoritam stabla odlučivanja. Kao glavni razlozi njegove pobjede su se istakli prilagodljivost situacijama i značajno upravljanje borbenim jedinicama. Nakon što se svi bodovi zbroje, ukupni rezultat algoritma stabla ponašanja je 32 boda, dok je algoritam stabla odlučivanja skupio 16 bodova.

S obzirom na to da promjenjivost borbenog stanja i nasumičnih elemenata u kodu mogu uzrokovati drugačije ishode, potrebo je napraviti veći broj testiranja u različitim uvjetima. Svako testiranje će biti prvenstveno podijeljeno prema povoljnosti karte, a zatim prema borbi s istim algoritmom i drugim algoritmom. Zbog sličnosti uvjeta i nasumičnosti u ponašanju algoritama, ukupni bodovi za svaku kategoriju su izvedeni na manjem uzorku od 10 borbi.

Na tablici 6.2 se nalaze ukupni rezultati svake moguće permutacije i ukupni rezultati svakog algoritma. Borbe se analiziraju po stupcima, gdje bodovi prikazuju rezultat algoritma iz stupca kada se suprotstavlja algoritmu iz retka, pri odgovarajućoj poziciji na karti. Promatrajući primjer borbe algoritma stabla ponašanja i odlučivanja u neutralnoj poziciji uočava se velika razlika rezultata bodova. U navedenom scenariju algoritam stabla ponašanja je skupio 308 bodova u

usporedbi sa 111 bodova algoritma stabla odlučivanja, pretežno zbog broja pobjeda, zbog čega se može zaključiti kako je daleko uspješniji u odnosu na algoritam stabla odlučivanja.

KARTE	**BORBE**	*Algoritam stabla ponašanja*	*Algoritam stabla odlučivanja*
Neutralna pozicija	*Algoritam stabla ponašanja*	186	111
	Algoritam stabla odlučivanja	308	215
Povoljna pozicija	*Algoritam stabla ponašanja*	302	219
	Algoritam stabla odlučivanja	317	212
Nepovoljna pozicija	*Algoritam stabla ponašanja*	304	253
	Algoritam stabla odlučivanja	343	271
	UKUPNO BODOVA	1760	1281

Tablica 6.2: Rezultati testiranja uspješnosti algoritama umjetne inteligencije po bodovima

KARTE	**BORBE**	*Algoritam stabla ponašanja*	*Algoritam stabla odlučivanja*
Neutralna pozicija	*Algoritam stabla ponašanja*	5	0
	Algoritam stabla odlučivanja	10	6
Povoljna pozicija	*Algoritam stabla ponašanja*	9	7
	Algoritam stabla odlučivanja	10	7
Nepovoljna pozicija	*Algoritam stabla ponašanja*	1	0
	Algoritam stabla odlučivanja	3	3
	UKUPNO POBJEDA	38	23

Tablica 6.3: Rezultati testiranja uspješnosti algoritama umjetne inteligencije po pobjedama

Iz ovih tablica mogu se uočiti rezultati koji su bili očekivani, ali i oni koji, iako neočekivani, imaju jasno objašnjenje. Među očekivanim rezultatima se jasno izdvajaju borbe između istih algoritama pri neutralnoj poziciji, gdje su prikazani jednaki omjeri pobjeda i poraza. Kod algoritma stabla odlučivanja ta je vrijednost nešto veća, no još uvijek unutar dopuštenog raspona, što se može pripisati manjem broju provedenih testiranja. Slična pojava uočena je i kod algoritma stabla ponašanja u situacijama kada je imao povoljnu poziciju, što potvrđuje predviđanja o njegovoj učinkovitosti.

U situacijama kada jedna strana ima prednost nad drugom, očekivanja su usmjerena prema algoritmu s prednošću, a rezultati testiranja pokazuju da je to bio slučaj s algoritmom stabla ponašanja. Dodatni rezultati iz scenarija u kojem je algoritam stabla ponašanja bio u nepovoljnoj poziciji potvrđuju ove zaključke, jer se pokazalo da je, unatoč nepovoljnoj situaciji, uspio nadmašiti algoritam stabla odlučivanja, što ukazuje na njegove superiorne performanse. Također je važno naglasiti da je algoritam stabla ponašanja ostvario sve pobjede u borbama s neutralnom pozicijom, čime se dodatno potvrđuje njegova snaga i sposobnost prilagodbe različitim situacijama. Ovi rezultati pružaju jasan uvid u učinkovitost algoritma stabla ponašanja i njegovu sposobnost nošenja s različitim izazovima, bez obzira na početne uvjete u kojima se našao. Algoritam stabla odlučivanja pokazao je neočekivanije rezultate u borbama na povoljnoj i nepovoljnoj poziciji. Primjetno je variranje njegovih rezultata, a unatoč povremenoj prednosti, mogućnost gubitka je znatno veća u usporedbi s algoritmom stabla ponašanja u identičnim uvjetima. Ova varijabilnost rezultata sugerira da je algoritam stabla odlučivanja skloniji nepredvidivim i nesigurnim ishodima. Takav zaključak dodatno je potvrđen kroz analizu suprotnih borbi gdje se, također, uočava ista vrsta variranja. Time se jasno ukazuje na njegovu osjetljivost na promjene u uvjetima borbe i na njegovu manju pouzdanost u usporedbi s algoritmom stabla ponašanja.

Ako se analiziraju rezultati bodova, jasnije se uočava da su performanse algoritma stabla ponašanja bolje i predvidljivije u usporedbi s algoritmom stabla odlučivanja. U borbama gdje je algoritam stabla ponašanja bio u povoljnoj i nepovoljnoj poziciji, bodovi su bili veoma slični, što ukazuje na dosljedno visoku razinu performansi, kvalitete i efikasnosti, neovisno o početnim uvjetima. Dobiveni rezultati dodatno potvrđuju optimalno iskustvo borbe koje pruža algoritam stabla ponašanja, bilo u slabijoj ili boljoj poziciji u odnosu na ostale igrače, čime se osigurava stabilnost i pouzdanost tijekom igranja. Još jedan važan podatak koji je sada veoma jasan zbog gornjih rezultata je ekstremno velika razlika u usporedbi algoritma stabla ponašanja i stabla odlučivanja. Kroz zapažanje borbenih rezultata u slučaju neutralne pozicije uočava se da je algoritam stabla

ponašanja svaki put pobijedio algoritam stabla odlučivanja. Iako se zbog manjeg broja uzoraka ne može sa sigurnošću reći da će se taj rezultat ponoviti svaki put, postoji dovoljno dokaza koji sugeriraju značajnu prednost algoritma stabla ponašanja. Ovi rezultati pružaju čvrstu osnovu za zaključak da je algoritam stabla ponašanja, u kontekstu RTS aplikacija i igara daleko uspješniji od algoritma stabla odlučivanja.

Analizom rezultata, možemo primijetiti nekoliko ključnih razloga zašto je algoritam stabla ponašanja učinkovitiji. Kao prvi razlog su njegova fleksibilnost i prilagodljivost koje mu omogućavaju bolju prilagodbu na različite scenarije unutar igre. Dok se algoritam stabla odlučivanja oslanja na fiksne odluke koje su unaprijed definirane, algoritam stabla ponašanja može dinamički reagirati na promjene u okruženju, prilagođavajući svoje odluke na temelju trenutne situacije. Sposobnost brzog odgovora na promjene u igri osigurava mu značajnu prednost, posebno u dinamičnim RTS igrama gdje se situacija na borbenom polju može brzo promijeniti. Drugi ključni razlog učinkovitosti algoritma stabla ponašanja je dosljednost u performansama, bez obzira na uvjete u kojima se nalazi. To je posebno uočljivo u rezultatima borbi u kojima se algoritam stabla ponašanja pokazao iznimno učinkovitim i u povoljnim i u nepovoljnim pozicijama. Njegova sposobnost održavanja visoke razine performansi neovisno o situaciji ukazuje na izdržljivost i otpornost na vanjske čimbenike. S druge strane, algoritam stabla odlučivanja pokazao je veću varijabilnost u svojim rezultatima, što sugerira da je skloniji nepredvidivim ishodima, a to može dovesti do lošijih performansi u većini situacija. Zaključno, rezultati jasno pokazuju da algoritam stabla ponašanja ne samo da nadmašuje algoritam stabla odlučivanja u kontekstu uspjeha u bitkama, već pruža i konzistentniju i pouzdaniju osnovu za donošenje odluka unutar igre. Ovo je posebno važno u RTS igrama, gdje je potrebno brzo donositi odluke i prilagoditi se promjenama kako bi se osigurala pobjeda.

Algoritam stabla ponašanja, sa svojom sposobnošću prilagođavanja i reagiranja na promjene, osigurava prednost koja može biti ključna za postizanje pobjede. Ovi rezultati nedvojbeno ukazuju na to da je algoritam stabla ponašanja bolji izbor za implementaciju u RTS aplikacijama i igrama, osiguravajući bolje performanse i veću prilagodljivost u dinamičnim okruženjima.

7. ZAKLJUČAK

Algoritmi umjetne inteligencije, a time i same usluge za korištenje umjetne inteligencije su napredovali rekordnom brzinom. Zbog velikih prednosti i olakšanja korištenja ovih usluga, AI tehnologija se samo nastavljala razvijati, ali time i ljudsko oslanjanje na nju. Gledajući na strateško stranu priče u kontekstu RTS igara jasno se primjećuje kako svaka strateška igara koristi sve naprednije algoritme time stvarajući vrhunsko korisničko iskustvo. AI protivnici današnjih RTS igara mogu predviđati buduće poteze, dinamički se ojačati ili oslabiti ovisno o borbenom stanju i igračevim željama, organizirati sa drugim stvarnim i AI igračima i više. Usprkos tome, igrači se još uvijek žale na neefikasnost AI protivnika i osuđuju im performanse ovisno o sitnim greškama ili nedostatcima. Cilj ovog rada je bio pogled u prošlost AI protivnika RTS igara, u rane dane umjetne inteligencije, kada su AI algoritmi bili daleko manje napredni i dostupni nego što su danas. U vremenu kada su sofisticirani AI alati lako dostupni i integrirani u razne aplikacije, lako je uzeti zdravo za gotovo složenost i izazove s kojima su se programeri nekad suočavali. Današnji AI sustavi nude besprijekorna rješenja koja su nekada bila predmet intenzivnog istraživanja i eksperimentiranja.

Kroz ovaj rad i njegovu implementaciju i usporedbu algoritama stabla ponašanja i stabla odlučivanja, vratilo se u ta ranija vremena kada je stvaranje učinkovite umjetne inteligencije za RTS igre bilo zastrašujući zadatak i gdje se svaki sitni napredak smatrao revolucionarnim uspjehom. Stablo ponašanja, sa svojom modularnom i prilagodljivom strukturom, omogućilo je uvid u postepenu evoluciju umjetne inteligencije, pružajući fleksibilniji i dinamičniji pristup donošenju odluka. Suprotno tome, algoritam stabla odlučivanja, iako temeljni, istaknuo je ograničenja i krutost s kojima su se programeri morali suočiti pri radu s osnovnim AI strukturama. Rezultati ovog rada pokazali su koliko je izazovno bilo raditi s ovim ranim algoritmima. Algoritam stabla odlučivanja, iako učinkovit u određenim scenarijima, nije imao potrebnu prilagodljivost i dosljednost za složena, dinamična okruženja kao što su ona u RTS igrama. Algoritam stabla ponašanja, iako sofisticiraniji, i dalje je zahtijevao pažljiv dizajn i implementaciju kako bi učinkovito funkcionirao.

Promatrajući navedene izazove, postalo je jasno kako je evolucija umjetne inteligencije u igrama, a osobito u strateškim RTS igrama prošla dug put. Rani algoritmi umjetne inteligencije, unatoč svojim ograničenjima, postavili su temelje za napredne sustave na koje se oslanjaju današnji algoritmi umjetne inteligencije. Ponovnim proučavanjem ovih osnovnih algoritama, stečena je dublja zahvalnost za napredak koji je postignut i za složenost uključenu u stvaranje inteligentnog AI protivnika u RTS igrama.

Na kraju, ovo istraživanje je pokazalo da, unatoč svojoj jednostavnosti, ovi temeljni AI algoritmi i dalje sadrže vrijedne lekcije za današnje programere. Time osvrću na važnost razumijevanja i cijenjenja korijena razvoja umjetne inteligencije. Dok moderni alati olakšavaju implementaciju sofisticiranih AI algoritama, put otkrivanja i inovacija oslanja se na temeljima koje su postavili ti rani, tada iznimno kompleksni i izazovni algoritmi. Kroz ovaj rad, pokazalo se kako ovi temeljni AI algoritmi, unatoč svojoj jednostavnosti, pružaju vrijedne uvide i lekcije koje su relevantne za današnje programere, podsjećajući na važnost ustrajnosti i kreativnosti u stalno promjenjivom području umjetne inteligencije.

LITERATURA

[1] B., Geryk, „A History of Real-Time Strategy Games“, [online] *GameSpot*, 27-tra-2014. Dostupno na https://web.archive.org/web/20110427052656/http://gamespot.com/gamespot/features/all/real_time/ [Pristupljeno: 24.05.2024.].

[2] B., Colayco, „Age of Empires 2: Designer Diary“, [online] *FiringSquad*, 03-sij-2013. Dostupno na: https://archive.ph/20130103073907/http://www.firingsquad.com/games/aoe2diary/ [Pristupljeno: 24.05.2024.].

[3] C., Remo, „The Design of StarCraft II“, [online] *GameDeveloper*, 26-lis-2009. Dostupno na: https://www.gamedeveloper.com/business/the-design-of-i-starcraft-ii-i- [Pristupljeno: 25.05.2024.].

[4] S., Totilo, „Civilization V Preview: Small Changes, Big Differences“, [online] 10-ožu-2010. Dostupno na: https://kotaku.com/civilization-v-preview-small-changes-big-differences-5489814 [Pristupljeno: 25.05.2024.].

[5] W., Russell, „The Battle For Wesnoth“ [online] *Phoronix*, 21-sij-2009. Dostupno na: https://www.phoronix.com/review/890/4 [Pristupljeno: 26.05.2024.].

[6] „Hex grid (for 1.2, semi-urgent)“ [online] *Wesnoth Forums*, 22-pro-2006. Dostupno na: https://forums.wesnoth.org/viewtopic.php?t=14432

[7] S., Cheriyedath, „Designing Real-Time Strategy Games Using AI“, [online], 02-sij-2024. Dostupno na https://www.azoai.com/article/Designing-Real-Time-Strategy-Games-Using-AI.aspx [Pristupljeno: 27.05.2024.].

[8] B. G., Weber, M., Mateas, A., Jhala, „Building Human-Level AI for Real-Time Strategy Games“, *Advances in Cognitive Systems*, 2011.

[9] N. Ahamed, „AI Algorithms in Games“, [online] *DevGenious*, 02-ruj-2022. Dostupno na: https://blog.devgenius.io/ai-algorithm-in-games-db8bf1cc195f . [Pristupljeno: 29.05.2024].

[10] M., Dealessandri, „What is the best game engine: is Unity right for you?“, [online] *Gamer Network*, 16-sij-2020. Dostupno na: https://www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you [Pristupljeno: 02.06.2024.].

[11] J., Brodkin, „How Unity3D Became a Game-Developing Beast“, [online] *Dice Insights*, 19-lis-2018. Dostupno na: https://www.dice.com/career-advice/how-unity3d-become-a-game-development-beast [Pristupljeno: 02.06.2024.].

[12] A., Amit, „Hexagonal Grids“ [online] *Red Blob Games*, ožu-2013. Dostupno na: https://www.redblobgames.com/grids/hexagons/ [Pristupljeno: 04.06.2024.].

[13] Unity dokumentacija. Dostupno na: https://docs.unity3d.com/Manual/index.html

[14] D., von Winterfeldt, E., Ward, „Decision trees“, *Decision Analysis and Behavioural Research. Cambridge University Press*, str. 63-89, 1986.

[15] N. C., Chauhan „Decision Tree Algorithm, Explained“ [online], 09-velj-2022. Dostupno na: https://www.kdnuggets.com/2020/01/decision-tree-algorithm-explained.html. [Pristupljeno: 12.06.2024.].

[16] R., Quinlan, „Learning efficient classification procedures“, *Machine Learning: an artificial intelligence approach*, str. 463-482, 1983.

[17] T., Plapinger, „What is a Decision Tree?“, *Towards Data Science*, srp. 2017.

[18] M., Pêcheux, „How to create a simple behaviour tree in Unity/C#“, [online] *Medium* ,02-stu-2021. Dostupno na: https://medium.com/geekculture/how-to-create-a-simple-behaviour-tree-in-unity-c-3964c84c060e [Pristupljeno: 18.06.2024.].

SAŽETAK

U ovom radu prolazi se kroz načine rada i implementaciju osnovnih algoritama AI protivnika u RTS aplikacijama. Zadatak ovog završnog rada je istraživanje, implementacija i analiza dva osnovna algoritma umjetne inteligencije u kontekstu RTS igara, konkretno algoritam stabla odlučivanja i algoritam stabla ponašanja. U te svrhe izrađena je prilagođena RTS aplikacija koristeći *Unity Game Engine*. Kroz proučavanje, implementaciju i usporedbu ovih algoritama, cilj je prikaz složenost i izazove s kojima su se susretali rani programeri AI sustava te prikazati evoluciju umjetne inteligencije u strateškim igrama. Preko rezultata rada omogućuje se dublje razumijevanje kako su temeljni AI pristupi poslužili kao osnova za napredne sustave koji se danas koriste te su naglasili važnost ustrajnosti i inovacija u ovom polju.

Ključne riječi: AI protivnik, algoritam stabla odlučivanja, algoritam stabla ponašanja, RTS igre, umjetna inteligencija

ABSTRACT

This paper explores the working methods and implementation of basic AI opponent algorithms in RTS applications. The task of this final paper is the research, implementation, and analysis of two basic artificial intelligence algorithms in the context of RTS games, specifically the decision tree algorithm and the behavior tree algorithm. For this purpose, a customized RTS application was created using the Unity Game Engine. Through the study, implementation, and comparison of these algorithms, the aim is to showcase the complexity and challenges faced by early AI system developers and to illustrate the evolution of artificial intelligence in strategy games. The results of this work provide a deeper understanding of how fundamental AI approaches served as the foundation for advanced systems used today, emphasizing the importance of persistence and innovation in this field.

Keywords: AI opponent, artificial intelligence, behavior tree algorithm, decision tree algorithm, RTS games

ŽIVOTOPIS

Eldin Mešić rođen je 05.02.2000. godine u Vinkovcima, Republika Hrvatska. Završio je Osnovnu Školu Orašje u Orašju. Srednju školu je završio u Orašju, smjer elektrotehničar. 2019. godine upisao se na preddiplomski sveučilišni studij elektrotehnike na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku, a 2020. godine prebacio na preddiplomski sveučilišni studij računarstvo na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. 2022. godine se upisao na diplomski studij DRC – Programsko inženjerstvo na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku.