Konačni automat kao model umjetne inteligencije primijenjen na upravljanje umjetnim igračem

Čičak, Tomislav

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Tomislav Čičak

KONAČNI AUTOMAT KAO MODEL UMJETNE INTELIGENCIJE PRIMIJENJEN NA UPRAVLJANJE UMJETNIM IGRAČEM

ZAVRŠNI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Čičak

Matični broj: 46201/06–R

Studij: Informacijski sustavi

KONAČNI AUTOMAT KAO MODEL UMJETNE INTELIGENCIJE PRIMIJENJEN NA UPRAVLJANJE UMJETNIM IGRAČEM

ZAVRŠNI RAD

Mentor:

dr. sc. Bogdan Okreša Đurić

Varaždin, rujan 2021.

Tomislav Čičak

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI Radovi

Sažetak

U ovom završnom radu prikazana je primjena konačnog automata unutar videoigra. Na početku rada opisan je način pripreme i korištenje literature radi upoznavanja sa temom koju obrađujemo. Nakon toga opisani su korišteni alati za izradu konačnog automata: Unity i Microsot visual studio. U nastavku dolazimo do teorijskog dijela rada u kojem detaljnije objašnjavamo što je to umjetna inteligencija te prikazujemo vrste umjetne inteligencije koje su zastupljene u videoigrama. Slijedom toga dolazimo do glavne teme a to je konačni automat. Prikazujemo njegov osnovni izgled i način implementacije unutar programskog jezika c++. Nakon upoznavanja sa osnovama konačnog automata pomoću primjera iz literature, implementiramo simulacijsku videoigru koja je zasnovana na konačnom automatu. U videoigri se nalaze dva tima pčela čiji je zadatak povećati svoju brojnost pomoću raznih sredstava koja se nalaze na mapi. Nakon implementacije prikazan je i rezultat implementirane videoigre. Pri kraju završnog rada prikazana je kreacija još jedne igre, ali u programu Unity. Pomoću alata Unity kreirana je videoigra sa tenkovima. Jednim tenkom upravlja igrač, a drugim konačni automat odnosno umjetna inteligencija. Nakon prikazanih svih postupaka do kreiranja spomenute videoigre prikazan je rezultat koji prikazuje ponašanje neigrajućeg tenka unutar različitih stanja. Na kraju rada nalazi se zaključak i literatura korištena za izradu.

Ključne riječi: Unity, C#, Umjetna inteligencija, Konačni automat, Videoigra

Sadržaj

1. Uvod

Gaming industrija od svojih početaka bilježi kontinuirani rast i predstavlja veoma bitan dio života današnje mladeži i danas bi bilo jako teško naći osobu koja nikada u svome životu nije zaigrala niti jednu videoigru. Usporedimo li videoigre proizvedene danas i prije 20 godina, možemo zaključiti kako je tehnologija napredovala. Prvo što će nam vjerojatno činiti veliku razliku je grafika videoigre i realnost virtualnog svijeta što uključuje fiziku i ponašanje objekata kao da su stvarni. No, jedna od bitnijih i nezaobilaznih stvari koja zapravo ruši granicu između stvarnog i virtualnog svijeta je umjetna inteligencija. Današnje videoigre su nezamislive bez umjetne inteligencije, jer je ona zaslužna za upravljanje raznim likovima kao što su ljudi, životinje koji se zajednički nazivaju neigrajućim likovima (eng. non-player character, NPC) te drugim raznim događajima unutar videoigre.

Umjetna inteligencija kontrolira neigrajuće likove na koje će igrač naići tijekom svog igranja te će mu obogatiti sam virtualni svijet i mogućnosti kontakta s njime. Teško je zamisliti ratne videoigre bez ogromnih vojski, sportske igre bez protivnika protiv kojega se igrač natječe i slične. Upravo zbog takvih značajnih stvari, tema ovog rada odnosi se na početke razvoja umjetne inteligencije i sam način razvoja istih. Kao metoda umjetne inteligencije odabran je konačni automat koji je u ranim fazama bio jako zastupljen zbog svoje jednostavnosti i praktičnosti. Iako je danas sve rjeđe u uporabi, još uvijek pronalazi svoje mjesto kao početni model umjetne inteligencije u videoigrama koje razvijaju pojedinci ili manje kompanije s nižim budžetom, koje se stručno nazivaju indy videoigre.

Ovaj završni rad sam odabrao kako bih produbio svoje znanje o umjetnoj inteligenciji i shvatio kako zapravo ona radi i što je sve potrebno poduzeti kako bi je uopće ostvarili. Izradom igre u poznatom programu Unity i korištenjem Visual Studia za pisanje skripti usvojit ću nova znanja te smatram kako ću ubuduće sve videoigre gledati drugim očima i dati priznanje proizvođačima na izrađenoj umjetnoj inteligenciji.

2. Metode i tehnike rada

Metode i tehnike korištene za izradu ovog rada su prvotno dva alata korištena za izradu videoigre, a to su „Unity“ i „Microsoft Visual Studio“ odnosno programski jezik C#.

Priprema za izradu završnog rada sastojala se od prethodnog znanja rada u programu Visual Studio te programskog jezika C# za koji je bio zaslužan kolegij "Programsko inženjerstvo". Za stjecanje ostalog znanja zaslužan je servis "Udemy" koji nudi razne tečajeve. Za potrebe završnog rada završio sam tečaj koji se dotiče uvoda i općeg znanja o programu "Unity", pisanje skripti u programskom jeziku C# te prikazom raznih tehnika i metoda izrade 3D igara. Drugi tečaj koji sam završio je bio vezan za umjetnu inteligenciju koja je prisutna u "Unityu" što uključuje traženje puta i prikaz kako izraditi novu.

Kako bi se došlo do konačnog rješenja u završnom radu, prvo smo se dotakli literature koja nas je uputila u teorijski dio rada. Naučili smo što je točno konačni automat i kako izleda njegov osnovni model. Gdje ga je dobro primjenjivati i koje su poznate igrice bile zasnovane na njemu. Kroz daljnje napredovanje u literaturi pronašli smo implementirani primjer konačnog automata u sklopu simulacijske videoigre koji smo detaljno proučili. [1, str. 236-255]

Na bazi primjera uz nekoliko preinaka implementirali smo svoju simulacijsku videoigru te trenutno znanje podigli na novu razinu za koju smatram da je bila dovoljna za nastavak izrade završnog rada. Na kraju svo stečeno znanje primjenjeno je za izradu videoigre unutar programa Unity.

Uz sve korištene metode i tehnike, naporan rad i uloženo vrijeme u samostalno proučavanje alata su neke od najbitnijih stvari koje su rezultirale izradom finalnog programskog rješenja.

2.1. Opis korištenih alata

Kako je već spomenuto za izradu praktičnog dijela rada korišteni su alati Unity i Microsoft Visual Studio te ćemo ih detaljnije opisati u nastavku.

2.1.1. Unity

Ukratko, Unity je alat za izradu videoigra, te svatko tko bi se htio okušati u izradi videoigra svoj fokus bi trebao dati Unity-u. Unity je moćan alat, lagan za korištenje i besplatan ukoliko pojedinac ne zarađuje velike novce od izrađene igrice. Neke od poznatijih igrica izrađene u Unity-u su: Temple Run[2], Angry Birds[3], Super Mario Run[4], Subnautica[5]. Pošto je Unity zapravo game engine (hrv. motor za igre) programerima je jako olakšan rad jer neke stvari ne moraju sami implementirati što im skraćuje vrijeme izrade par mjeseci, pa čak i godina. Za nezavisne programere (eng. indie developers) ovo daje mogućnost natjecanja s većim kompanijama.[6]

2.1.2. Microsoft visual studio

Stvoren od strane Microsofta, Visual Studio je integrirano razvojno okruženje (eng. integrated development environment, IDE) koje se koristi za razvoj računalnih programa za Windows operacijski sustav. Moguće ga je koristiti i za izradu internet stranica, internet aplikacija i internet servisa.[7] No, uz sve to preporučeni je alat za pisanje skripti koje su potrebne za videoigre unutar Unity-a, a razlog toga je njegova mogućnost predviđanja koda (IntelliSense) koja ubrzava pisanje i smanjuje gramatičke pogreške. Brzo ispravljanje pogrešaka, postavljanje točaka prekida te procjena varijabli i kompleksnih izraza su samo neke od značajki koje olakšavaju programerima njihov rad.[8]

3. Umjetna inteligencija u videoigrama

Kako bi krenuli pričati o konačnom automatu, moramo se upitati što on predstavlja. Već znamo da je on jedan od modela umjetne inteligencije igre[1], ali što bi to uopće umjetna inteligencija trebala biti i što to ona predstavlja.

U svom najjednostavnijem obliku, umjetna inteligencija je područje koje kombinira računalnu znanost i robusne skupove podataka kako bi omogućilo rješavanje problema[9]. Također, obuhvaća potpodručja strojnog učenja i dubokog učenja, koja se često spominju zajedno s umjetnom inteligencijom. Ove se discipline sastoje od AI (eng. artificial intelligence) algoritama koji nastoje stvoriti stručne sustave koji predviđanja ili klasifikacije temelje na ulaznim podacima. Kako bi olakšali razumijevanje ove definicije najlakše bi bilo reći da umjetna inteligencija koristi računala i strojeve kako bi oponašala sposobnosti ljudskog uma za rješavanje problema i donošenje odluka.[9]

U nastavku rada više ćemo se orijentirati na umjetnu inteligenciju unutar videoigra (eng. Game AI), te spomenuti kakvi sve modeli postoje i koje su njihove razlike, te što očekivati od umjetne inteligencije igre (hrv. UI igre) u budućnosti.

3.1. Što je umjetna inteligencija igre?

U najširem značenju nekakva vrsta umjetne inteligencije (hrv. UI) je postojana u većini videoigra. Za primjer možemo uzeti jednu od poznatijih starih videoigara Pac Man[10]. Raznobojni duhovi koji postoje u videoigri su primjer umjetne inteligencije, a možemo navesti i umjetne igrače u videoigrici Call of Duty[11]. Poneki programeri smatraju da i algoritmi za pronalaženje puta, pa i sama detekcija kolizije spada u umjetnu inteligenciju videoigre. Vođeni time, umjetnu inteligenciju igre možemo shvaćati kao širok pojam koji uključuje jednostavno praćenje i bježanje od igrača, pa sve do neuronskih mreža i genetskih algoritama [1, str. 21-22], te se ona smatra "slabom" UI. Slaba UI uključuje širi raspon namjena i tehnologija kako bi strojevi dobili specijalizirane inteligentne kvalitete, dok jaka UI zahtijeva samoučenje, prilagođavanje, svijest pa čak i emocije te se ne ubraja u UI igre.[1, str. 22]

Za kraj možemo zaključiti da je definicija UI igre prilično široka i fleksibilna. Sve što daje iluziju inteligencije na prihvatljivoj razini, čineći videoigru obuzimajućom, izazovnijom i, što je najvažnije, zabavnom, može se smatrati UI igre. Baš poput upotrebe stvarne fizike u igrama, dobar UI dodaje dubinu videoigrici, privlačeći igrača i suspendirajući njegovu stvarnost na neko vrijeme.[1, str. 22]

3.1.1. Deterministička i nedeterministička

Tehnike UI igre dolaze u dvije verzije [1, str. 24]:

- Determinističke
- Nedeterminističke

Deterministički način je specifičan i predvidljiv. Čisti primjer determinističke metode je algoritam za praćenje igrača. Kodiramo umjetnog lika da se kreće prema poziciji igrača, pomicanjem po x i y-osi sve dok njegova pozicija nije jednaka onoj od igrača.

Nedeterministički je suprotnost determinističkom. Postoji neizvjesnost i nepredvidivost, a za primjer možemo uzeti umjetnog lika u videoigrici šah. Kroz igru on proučava igračeve pokrete i pomoću tih podataka prilagođava se i uči.

Determinističke metode su standard za izradu UI igre. Brzina, predvidljivost, lakoća implementacije, testiranja i otklanjanja pogreški su stvari koje su većini programera primamljive. Iako postoji mnogo pozitivnih značajki, jedan od problema je taj što svaki scenarij i svako ponašanje mora biti predviđeno i kodirano sa strane programera. Dodatni problem je mogućnost učenja i evoluiranja, jer bez tih značajki nakon nekog igranja, deterministično ponašanje postaje predvidljivo što utječe na dugovječnost igranja videoigre.

Nedeterminističke metode olakšavaju učenje i nepredvidivo igranje [1, str. 24]. Programeri imaju lakši posao pošto ne moraju direktno svako stanje predvidjeti i isprogramirati. Takve metode imaju mogućnosti samoučenja i ekstrapolacije te mogu promovirati takozvano "emergentno ponašanje" (eng. emergent) iliti ponašanje koje se pojavljuje bez izričitih uputa.

Iako zasad deterministične tehnike i dalje stoje na vrhu i češće se koriste, programeri se polagano okreću prema nedeterminističkim. Razlog sporoj promjeni je nepredvidivost razvijene metode. Pitanje je kako uz kratak rok za razvoj videoigre testirati sve moguće varijacije na koje će igrač naići tijekom igranja, a da ne dođe do nekih suludih rezultata.

3.1.2. Budućnost umjetne inteligencije

Sljedeća velika stvar u UI igre je učenje. Umjesto da su svi umjetni likovi unaprijed kodirani do vremena dok se videoigra isporuči, videoigra bi trebala evoluirati, učiti i prilagođavati se sve više što je igrana[1, str. 28]. Cilj je napraviti videoigru u kojoj videoigra prati igrača i gdje igrač ne može predvidjeti sljedeći događaj, čime se zapravo produljiva vrijeme života videoigre. Ako je videoigra predvidiva, ona postaje dosadna. No, ipak je točno ta nepredvidiva priroda učenja i evoluiranja videoigre utjecala na programere UI da prilaze tehnologijama sa zdravom dozom straha.

Nekoliko popularnih igara, poput Creatures[12], Black & White[13], Battlecruiser 3000AD[14], Dirt Track Racing[15] i Heavy Gear[16], koristile su nedeterminističke metode UI-ja. [1, str. 28] Njihov uspjeh izazvao je ponovno zanimanje za učenje UI metoda kao što su stabla odlučivanja, neuronske mreže, genetski algoritmi i vjerojatnosne metode.[1, str. 28]

Ove videoigre koristile su nedeterminističke metode u kombinaciji s tradicionalnim determinističkim metodama, te su ih koristile samo tamo gdje su najprikladnije. Ovakav pristup je preporučen jer je moguće izolirati dio UI koji je nepredvidljiv, što ga čini težim za razvijanje, testiranje i popravljanje grešaka, a uz sve to većina UI ostaje u tradicionalnoj formi.

3.2. Konačni automat

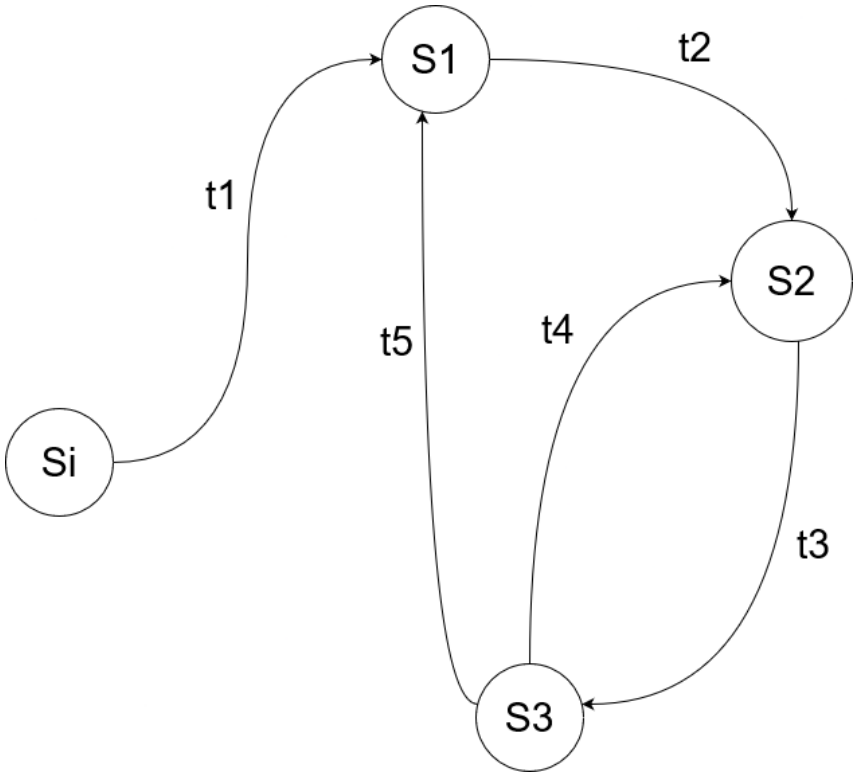
Konačni automat je apstraktni stroj koji može postojati u jednom od više različitih i predefiniranih stanja [1, str. 228]. Konačni automat definira potrebne uvjete koji trebaju biti ispunjeni da bi se stanje konačnog automata promijenilo. Stanje određuje kako će se točno konačni automat ponašati.

"Konačni automat koncept je koji je opisan ulaznom abecedom Σ, izlaznom abecedom Γ, konačnim nepraznim skupom stanja S dio kojeg je i početno stanje $S_0 \in S$, funkcijom prijelaza $\delta : S \times \Sigma \to S$ te izlaznom funkcijom $\omega : S \times \Sigma \to \Gamma$. Formalno, konačni automat je šestorka: $(\Sigma, \Gamma, S, S_0, \delta, \omega)$" [17]

Konačni automat datira sve do najranijih početaka programiranja videoigra. Jedan od svima poznatih konačnih automata je videoigra Pac Man, odnosno duhovi koji su prisutni u toj videoigrici. Za svoj rad koriste tri stanja: hvatanje, istraživanje i bježanje. Ovisno o igračevim postupcima njihova se stanja mijenjaju. Ako igrač pojede plavu pilulu, stanje se prebacuje u bježanje [1, str. 228].

Konačni automat je UI metoda koja je dominirala kontrolom i donošenjem odluka umjetnih likova do sredine 2000-ih[18, str. 52]. Iako je već dugo vremena u uporabi i dalje se koristi u modernim videoigrama. Sama činjenica što je lagan za razumjeti, implementirati i popraviti pogreške u kodu, doprinosi njegovoj čestoj uporabi u razvijanju videoigra.

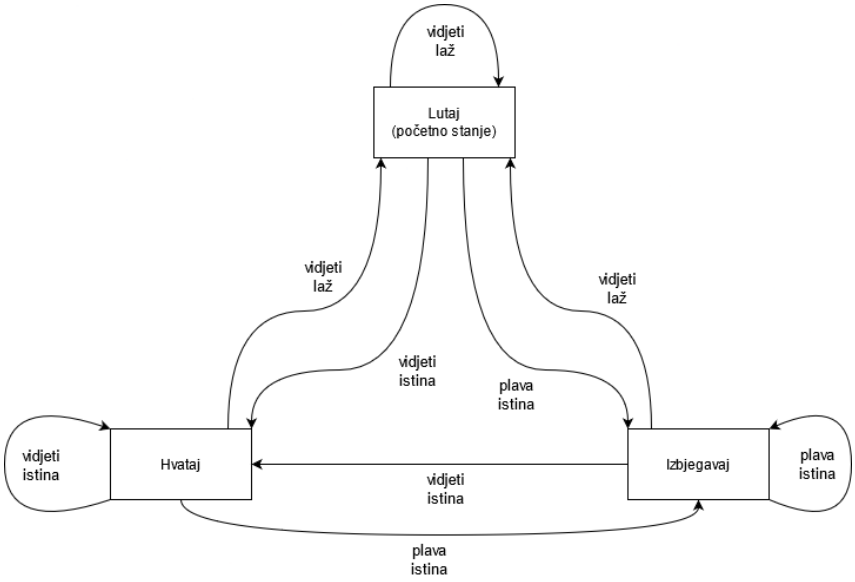
3.2.1. Osnovni model konačnog automata



Slika 1: Osnovni model konačnog automata

Na slici 1, svako moguće stanje je prikazano krugom i trenutno imamo četiri moguća stanja (Si, S1, S2, S3). Ako se prisjetimo, konačni automat može mijenjati svoja stanja ovisno o određenim uvjetima. Uvjeti i moguće promjene na slici su ilustrirani pomoću strelica (t1, t2, t3, t4, t5). Konačni automat počinje u stanju Si. Kako bi se njegovo stanje promijenilo potrebno je ispuniti uvjet t1 kako bi se stanje promijenilo u S1. Naravno, možemo vidjeti da je prijelaz iz stanja S1 u stanje S3 nemoguć pošto ne postoji veza. Stanje S2 i S3 imaju povratne veze pa zbog toga se mogu međusobno izmjenjivati.

Sada ćemo pokazati Pac Man model konačnog automata za duhove.



Slika 2: Pac Man model konačnog automata, prema uzoru na [1, str. 230]

Na slici 2 vidljivo je da duhovi mogu biti samo u 3 stanja: lutaj (eng. Roam), hvataj (eng. Chase), izbjegavaj (eng. Evade). Stanje `lutaj` je početno stanje prilikom pokretanja videoigre. Ako prilikom lutanja vidimo igrača, stanje se prebacuje u `hvataj`, što je prikazano strelicom i uvjetom `vidjeti istina` (eng. see true).

Ako tijekom hvatanja igrač pojede plavu pilulu i duhovi postanu plavi, stanje se mijenja u `izbjegavaj`. Tranzicija je prikazana strelicom `plava istina` te možemo vidjeti da se ista situacija dešava ako je duh u stanju `lutaj`.

Kada je duh u stanju `izbjegavaj`, za tranziciju `plava istina` možemo vidjeti da stanje ostaje isto, što znači da sve dok duhovi imaju plavu boju odnosno efekt plave pilule je postojan, duhovi ostaju u stanju `izbjegavaj`.

Jedini način da duh promjeni svoje stanje je pričekati da efekt plave pilule prođe te onda ovisno o uvjetu da li je igrač vidljiv promijeniti stanje u ono odgovarajuće, `lutaj` ako igrač nije vidljiv te `hvataj` ako je.

Sada kada postoji vizualni prikaz kako bi jedan konačni automat trebao izgledati, biti će pokazan način kako ga programski izvesti u programskom jeziku C++. Radi lakšeg povezivanja sadržaja s isječcima koda u nastavku će biti korišteni engleski nazivi stanja, funkcija i uvjeta.

Isječak kôda 1: Primjer koda za konačni automat duha, preuzeto od [1, str. 230- 231]

```
switch (currentState)
{
    case kRoam:
        if (imBlue==true) currentState=kEvade;
        else if (canSeePlayer==true) currentState=kChase;
        else if (canSeePlayer==false) currentState=kRoam;
        break;
    case kChase:
        if (imBlue==true) currentState=kEvade;
        else if (canSeePlayer==false) currentState=kRoam;
        else if (canSeePlayer==true) currentState=kChase;
        break;
    case kEvade:
        if (imBlue==true) currentState=kEvade;
        else if (canSeePlayer==true) currentState=kChase;
        else if (canSeePlayer==false) currentState=kRoam;
        break;
}
```

Iako ovaj primjer možda nije najbolja prezentacija konačnog automata, moguće je vidjeti da kod predstavlja sliku 2. Program provjerava trenutno stanje duha pomoću varijable `currentState` putem naredbe `switch`. Svatko stanje unutar sebe ima tri uvjeta i ovisno o njima govori koje će stanje biti sljedeće. Vidljivo je da `imBlue` uvjet ima najbitniju važnost i ako je on pozitivan stanje odmah prelazi u `kEvade` neovisno o ostalim uvjetima. Ako je uvjet `imBlue` neistinit ovisno o vidljivosti igrača uvjetovano varijablom `canSeePlayer` stanje se mijenja u `kChase` ili `kRoam`.

3.2.2. Dizajn konačnog automata u C++

U ovom dijelu će biti prikazan jednostavan dizajn izrade konačnog automata u videoigrici koristeći programski jezik C++. Za početak će posao biti razdvojen na dvije komponente. Prva komponenta sadrži vrste struktura koje će biti korištene za spremanje podataka vezanih za UI lika. Nakon toga prikazano je kako implementirati funkcije koje će biti korištene za tranziciju između stanja konačnog automata.

Videoigre napravljene u programskim jezicima visoke razine kao što su C ili C++ većinom sve podatke vezane za UI lika spremaju u strukture ili klase. Takva struktura može sadržavati razne detalje o liku kao što su: životni bodovi, jačina, brzina, vitalnost, razne moći i inventar lika. Osim takvih detalja strukture sadrže i trenutno stanje lika, a to je zapravo ono što na kraju najviše utječe na njegovo ponašanje.

Primjer koda 2 prikazuje kako izgleda jedna klasa koja sadrži sve informacije o liku.

Isječak kôda 2: Primjer koda klase lika, preuzeto od [1, str. 232 - 233]

```
class AIEntity
{
    public:
        int type;
        int state;
```

```
6           int row;
7           int column;
8           int health;
9           int strength;
10          int intelligence;
11          int magic;
12  };
```

Varijabla `type` sadržava tip lika, na primjer čovjek ili vilenjak. Sljedeća varijabla je najbitnija. Ona sadrži trenutno stanje lika, točnije stanje konačnog automata. Ostale varijable predstavljaju tipične stvari koje se nalaze u videoigrici.

Samo stanje je tipično definirano globalno i dodavanje dodatnih stanja je jednostavno definiranje novog među postojećim. Primjer koda 3 prikazuje način definiranja stanja.

Isječak kôda 3: Primjer koda definiranja stanja, preuzeto od [1, str. 233]

```
1   #define kRoam 1
2   #define kEvade 2
3   #define kAttack 3
4   #define kHide 4
```

Sada kada je struktura uspostavljena, kreće se na drugi dio, a to su funkcije koje služe za prijelaz stanja. Na primjeru koda 4 vidljive su novo dodane funkcije u postojeću klasu.

Isječak kôda 4: Primjer koda klase lika, preuzeto od [1, str. 233 - 234]

```
1   class AIEntity
2   {
3       public:
4           int type;
5           int state;
6           int row;
7           int column;
8           int health;
9           int strength;
10          int intelligence;
11          int magic;
12          Boolean playerInRange();
13          int checkHealth();
14  };
```

Za ovaj primjer dodane su samo dvije nove funkcije kako bi se prikazalo kako promijeniti stanje lika, no u pravoj videoigrici njih bi bilo znatno više. U sljedećem primjeru prikazano je kako se korištenjem dodanih funkcija može utjecati na promjenu stanja.

Isječak kôda 5: Primjer koda promjene stanja, preuzeto od [1, str. 234]

```
1
2       if ((checkHealth()<kPoorHealth) && (playerInRange()==false))
3       state=kHide;
4       else if (checkHealth()<kPoorHealth)
5       state=kEvade;
6       else if (playerInRange())
```

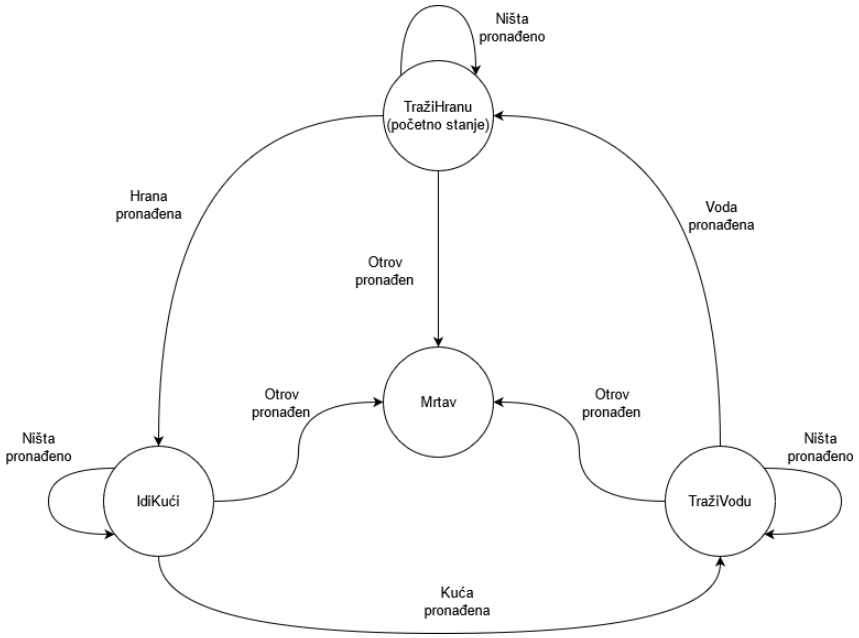
```
7       state=kAttack;
8       else
9       state=kRoam;
```

Prvi uvjet provjerava jesu li životni bodovi UI lika niski i je li igrač u blizini. Ako su oba uvjeta ispunjena lik odlazi u stanje `skrivanja` i u njemu ostaje sve dok su mu životni bodovi niski ili dok igrač nije u blizini. Drugim uvjetom i dalje se testira da li su životni bodovi niski, ali ako je igrač vidljiv onda se stanje mijenja u `izbjegavaj`. U tom stanju ostaje sve dok su mu životni bodovi niski. Trećim uvjetom provjerava je li igrač vidljiv te ako je, stanje prelazi u `napad`. U tom stanju ostaje sve dok je igrač u vidokrugu UI lika. Naposljetku, ako niti jedan uvjet nije ispunjen, točnije ako su životni bodovi visoki i igrač nije vidljiv, onda se događa prijelaz stanja u `lutaj` i u njemu ostaje sve dok se ne ispuni jedan od uvjeta.

3.2.3. Primjer izrade videoigre sa konačnim automatom u C++

U ovom dijelu rada će sve prije spomenute stvari biti primijenjene i izraditi će se simulacijska videoigra u kojoj su prisutna dva tima pčela. Cilj pčela je pronaći cvjetove i njihovu pelud donesti nazad u košnicu. Naravno, videoigra će imati zapreke i uvjete koje pčele moraju ispuniti. Kada pčela nađe cvijet i njegov pelud donese nazad u košnicu, stvara se nova pčela s istim zadatkom. Pčela koja je dostavila pelud tada kreće u potragu za vodom i luta po mapi sve dok ju ne nađe te onda opet nastavlja potragu za cvjetovima, odnosno peludi.

Hrana, voda i otrov su nasumično postavljeni po mapi, a otrov je spomenuta zapreka koja naravno ubija pčelu ako ga ona pronađe.



Slika 3: Model konačnog automata pčela, prema uzoru na [1, str. 237]

Kao što slika 3 prikazuje, svaka pčela počinje u stanju `TražiHranu` (eng. Forage). Iz tog stanja moguće su dvije tranzicije, ako je hrana pronađena prijelaz je u stanje `IdiKući` (eng. GoHome), a ako je pronađen otrov onda je stanje `Mrtav` (eng. Dead). Prelaskom u stanje `IdiKući` ponovo postoji izbor između dva stanja ovisno o uvjetima. Ako je pronađen otrov, pčela umire, točnije njezino stanje prelazi u `Mrtav` ili ako je košnica pronađena, stanje prelazi u `TražiVodu` (eng. Thirsty). U stanju `TražiVodu` postoje dvije moguće solucije. Prijelaz u stanje `Mrtav` ako je pronađen otrov te prijelaz u početno stanje `TražiHranu` ako je voda pronađena.
Ovaj konačni automat formalno možemo zapisati:
$\Sigma =$ {pronađi hranu , pronađi kuću, pronađi vodu, pronađi otrov, ništa pronađeno}
$\Gamma =$ {istražuj, odnesi kući, umri}
$S =$ {Traži hranu, Idi kući, Traži vodu, Mrtav}
$S_0 =$ {Traži hranu}

Tablica 1: δ

	Traži hranu	Idi kući	Traži vodu	Mrtav
pronađi hranu	Idi kući	...	...	...
pronađi kuću	...	Traži vodu	...	...
pronađi vodu	...	...	Traži hranu	...
pronađi otrov	Mrtav	Mrtav	Mrtav	...
ništa pronađeno	Traži hranu	Idi kući	Traži vodu	...

Tablica 2: ω

	Traži hranu	Idi kući	Traži vodu	Mrtav
pronađi hranu	odnesi kući	...	...	...
pronađi kuću	...	istražuj	...	...
pronađi vodu	...	...	istražuj	...
pronađi otrov	umri	umri	umri	...
ništa pronađeno	...	...	...	...

Kako je vidljivo da pčela kroz svoj posao mijenja više stanja i svako stanje ima svoj način ponašanja, sada je potrebno definirati ta stanja u programskom jeziku, što je prikazano na isječku koda 6.

Isječak kôda 6: Stanja pčela, prema uzoru na [1, str. 237]

```
1 #define kForage 1
2 #define kGoHome 2
3 #define kThirsty 3
4 #define kDead 4
```

Prvi uvjet simulacije je da pčela traži hranu, što je definirano stanjem `kForage`. Svaka pčela u tom stanju nasumično hoda po mapi u potrazi s hranom. Prilikom nailaska na hranu mijenja stanje u `kGoHome` i traži kuću zanemarujući dodatnu hranu koju pronađe na putu. Ako

pronađe kuću bez nailaženja na polje koje sadrži otrov, stanje se mijenja u `kThirsty` te sličnim načinom umjesto hrane traži vodu. Kada pronađe vodu, stanje se iz `kThirsty` mijenja u `kForage` i tako dolazi opet do početka, odnosno ponavlja se ciklus života pčele.

3.2.3.1. Klase i strukture konačnog automata

Kako je cilj konačnog automata opisan i prikazan, sada je potrebno kreirati klase i funkcije u koje će se podaci spremati.

Isječak kôda 7: Klasa pčela, prema uzoru na [1, str. 238]

```
1  #define kMaxEntities 200
2  class ai_Entity
3  {
4      public:
5      int type;
6      int state;
7      int row;
8      int col;
9      ai_Entity();
10     ~ai_Entity();
11 };
12
13 ai_Entity entityList[kMaxEntities];
```

Isječak koda 7 prikazuje kako klasa ima 4 varijable. Prva varijabla `type` odnosi se na tim kojem pčela pripada. Zbog toga je potrebno definirati te timove, što je prikazano u isječku koda 8.

Isječak kôda 8: Timovi pčela, prema uzoru na [1, str. 238]

```
1  #define kHoneyBee 1
2  #define kBumbleBee 2
```

Druga varijabla `state` služi za spremanje trenutnog stanja pčele. Moguća stanja su spomenuta prije, a to su: kForage, KGoHome, kThirsty, kDead.

Posljednje 2 varijable su `row` i `col`, a one služe za spremanje pozicije pojedine pčele na polju.

Izvan klase vidljivo je kreiranje polja u koje se spremaju podaci o pojedinoj pčeli, a maksimalan broj pčela je ograničen konstantom koja je definirana pri vrhu.

3.2.3.2. Definiranje popločenog svijeta

Kako je prije rečeno, ova simulacija će se odvijati na popločenom svijetu što znači da imamo retke i stupce, poput ploče za šah. Isječak koda 9 prikazuje definiranje konstanti i deklaraciju polja.

Isječak kôda 9: Definiranje svijeta, prema uzoru na [1, str. 239]

```
1  #define kMaxRows 15
```

```
2  #define kMaxCols 17
3  int terrain[kMaxRows][kMaxCols];
```

Svaki element u dvodimenzionalnom polju `terrain` sadrži vrijednost koja predstavlja neki tip ploče u svijetu. Definirane vrijednosti mogućih tipova ploča su prikazani u isječku 10.

Isječak kôda 10: Definiranje polja, prema uzoru na [1, str. 239]

```
1  #define kGround 1
2  #define kWater 2
3  #define kHoneyHome 3
4  #define kBumbledHome 4
5  #define kPoison 5
6  #define kFood 6
```

Zadana vrijednost svih ploča u svijetu je `kGround`, što znači da je to samo prazna lokacija u svijetu. Sljedeća konstanta je `kWater` i to je tip ploče koju pčela traži kada je u stanju `kThirsty`. Sljedeće dvije konstante predstavljaju pločice na kojima se nalazi košnica koju pčele traže kada se nalaze u stanju `kGoHome`. Pločica s vrijednosti `kPoison` ubija pčelu odnosno pri njezinom stajanju na tu ploču njezino stanje prijelazi u `kDead`. Posljednja konstanta je `kFood` i ploče s tom vrijednošću su odredište za pčele sa stanjem `kForage`.

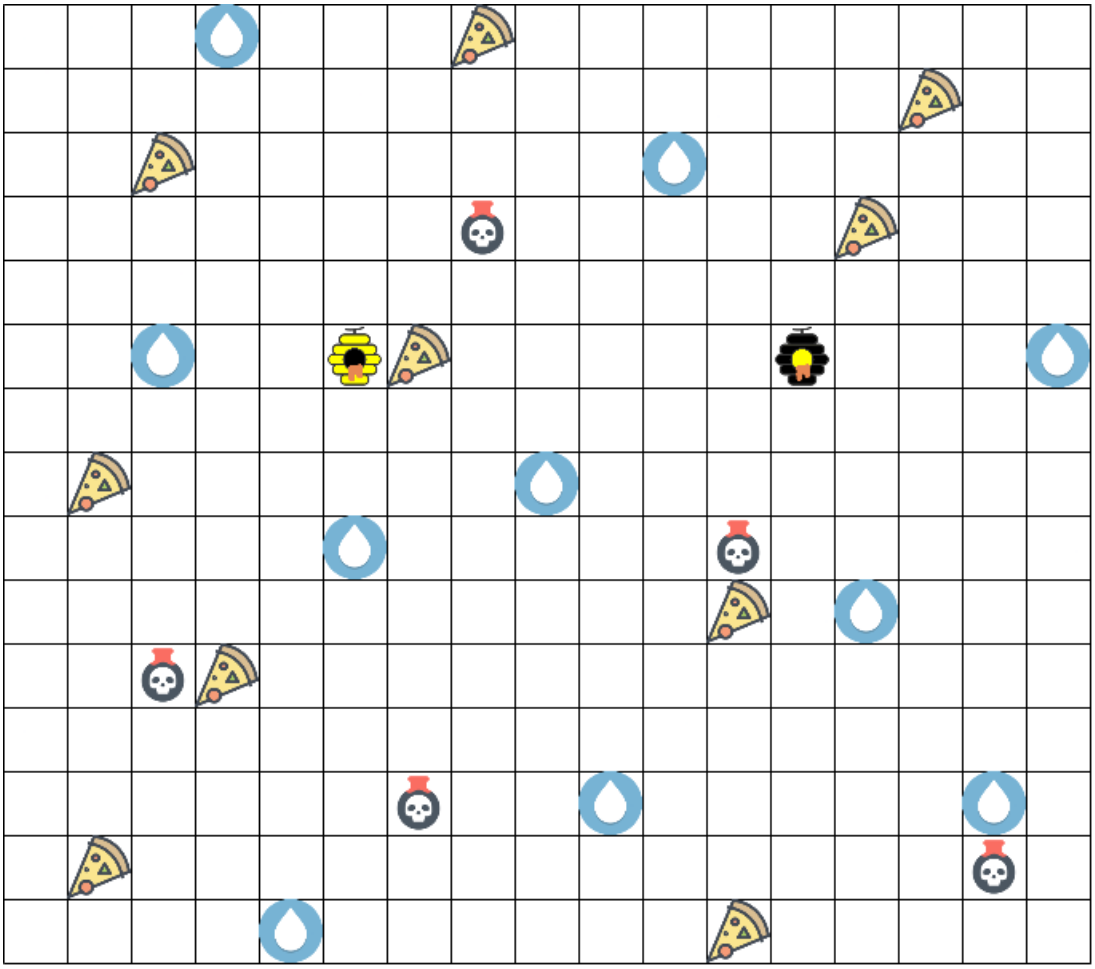
Kada su sve varijable i konstante deklarirane, nastavlja se s inicijalizacijom svijeta pomoću isječka koda 11.

Isječak kôda 11: Definiranje pozicije košnica i postavljanje polja, prema uzoru na [1, str. 240]

```
1  #define kHoneyHomeRow 5
2  #define kHoneyHomeCol 5
3  #define kBumbleHomeRow 5
4  #define kBumbleHomeCol 12
5
6  for (i=0;i<kMaxRows;i++)
7      {
8      for (j=0;j<kMaxCols;j++)
9          {
10             terrain[i][j]=kGround;
11         }
12     }
13 terrain[kHoneyHomeRow][kHoneyHomeCol]=kHoneyHome;
14 terrain[kBumbleHomeRow][kBumbleHomeCol]=kBumbleHome;
15
16 for (i=0;i<kMaxWater;i++)
17     terrain[Rnd(2,kMaxRows)-3][Rnd(1,kMaxCols)-1]=kWater;
18 for (i=0;i<kMaxPoison;i++)
19     terrain[Rnd(2,kMaxRows)-3][Rnd(1,kMaxCols)-1]=kPoison;
20 for (i=0;i<kMaxFood;i++)
21     terrain[Rnd(2,kMaxRows)-3][Rnd(1,kMaxCols)-1]=kFood;
```

Kako je vidljivo u kodu, prvo je inicijalizirano dvodimenzionalno polje kojemu je svaka njegova vrijednost tipa `kGround`. Nakon toga postavljaju se košnice na njihove koordinate koje su definirane konstantama `kHoneyHomeRow`, , `kBumbleHomeRow`, `kBumbleHomeCol`. Zadnje tri for petlje nasumično postavljaju vrijednosti `kWater`, `kPoison` i `kFood` po svijetu.

Slika 4 predstavlja jedan od mogućih rezultata inicijalizacije svijeta koji je zasad bez ijednog konačnog automata.



Slika 4: Izgled svijeta

Sada kada je svijet spreman, preostaje dodati konačne automate odnosno pčele.

3.2.3.3. Naseljavanje svijeta

Za kreiranje nove pčele dodana je nova funkcija u klasu.

Isječak kôda 12: Klasa pčela, prema uzoru na [1, str. 241 - 242]

```
1  class ai_Entity
2      {
3          public:
4              int type;
5              int state;
6              int row;
7              int col;
8              ai_Entity();
9              ~ai_Entity();
10             void New (int theType, int theState, int theRow, int theCol);
11     };
```

Funkcija `New` poziva se svaki put kada je potrebno dodati novu pčelu u svijet. Isječak koda 13 prikazuje definiranu funkciju.

Isječak kôda 13: Funkcija New, prema uzoru na [1, str. 242]

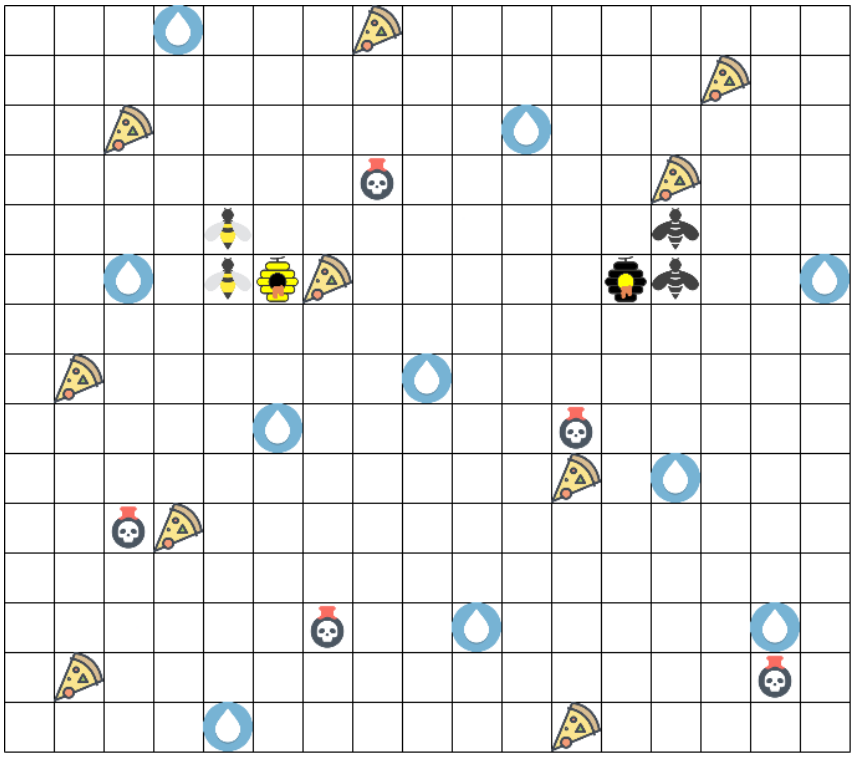
```
1 void ai_Entity::New(int theType, int theState, int theRow, int theCol)
2     {
3         type=theType;
4         state=theState;
5         row=theRow;
6         col=theCol;
7     }
```

Funkcija `New` inicijalizira 4 vrijednosti koje su zadane prilikom pozivanja funkcije, a to su `type`, `state`, `row` i `col`. Sljedeći isječak koda prikazuje dodavanje pčela u konačni automat simulacije.

Isječak kôda 14: Dodavanje pčela u svijet, prema uzoru na [1, str. 242]

```
1     entityList[0].New(kHoneyBee,kForage,4,4);
2     entityList[1].New(kHoneyBee,kForage,5,4);
3     entityList[2].New(kBumblebee,kForage,4,13);
4     entityList[3].New(kBumblebee,kForage,5,13);
```

U svaki tim dodane su po 2 pčele, zadano je početno stanje `kForage` te koordinate pčela su definirane s 2 zadnje vrijednosti koje predstavljaju red i stupac u svijetu. Prikaz trenutnog stanja svijeta je prikazan na Slici 5.



Slika 5: Izgled svijeta

Sada kada je svijet napučen, potrebno je pokrenuti simulaciju. Ako se prisjeti prijašnjeg

dijela rada, konačni automat se sastoji od više stanja koja sadržavaju razne informacije i ponašanja. To je dio koda koji će upravljati pčelama i njihovim ponašanjem ovisno o stanju u kojem se nalaze.

Isječak kôda 15: Provjera trenutnog stanja pčela, prema uzoru na [1, str. 244]

```
for (i=0;i<kMaxEntities;i++)
{
    switch (entityList[i].state)
    {
        case kForage:
        entityList[i].Forage();
        break;

        case kGoHome:
        entityList[i].GoHome();
        break;

        case kThirsty:
        entityList[i].Thirsty();
        break;

        case kDead:
        entityList[i].Dead();
        break;
    }
}
```

U isječku 15 vidljivo je da se pomoću naredbe `for` prolazi kroz polje pčela, te putem naredbe `switch` provjerava trenutno stanje svake pčele i ovisno u stanju dodjeljuje odgovarajuće ponašanje. U isječku koda 16 prikazano je da su dodane dodatne 4 funkcije u klasu pčele.

Isječak kôda 16: Klasa pčela, prema uzoru na [1, str. 244 - 245]

```
class ai_Entity
{
    public:
        int type;
        int state;
        int row;
        int col;
        ai_Entity();
        ~ai_Entity();
        void New (int theType, int theState, int theRow, int theCol);
        void Forage(void);
        void GoHome(void);
        void Thirsty(void);
        void Dead(void);
};
```

3.2.3.4. Funkcija Forage

Funkcija `Forage` upravlja ponašanjem pčela u stanju `Forage`. U ovom stanju pčele nasumično lutaju po svijetu u potrazi za hranom. Do promjene stanja može doći na 2 načina, ako pčela pronađe hranu onda mijenja stanje u `GoHome` ili ako naiđe na otrov, njeno stanje prelazi u `Dead`. Ovo ponašanje je implementirano u funkciji `Forage` te je prikazano u isječku koda 17.

Isječak kôda 17: Funkcija Forage, prema uzoru na [1, str. 245 - 246]

```
1  void ai_Entity::Forage(void)
2  {
3      int rowMove;
4      int colMove;
5      int newRow;
6      int newCol;
7      int foodRow;
8      int foodCol;
9      int poisonRow;
10     int poisonCol;
11
12     rowMove=Rnd(0,2)-1;
13     colMove=Rnd(0,2)-1;
14     newRow=row+rowMove;
15     newCol=col+colMove;
16
17     if (newRow<1) return;
18     if (newCol<1) return;
19     if (newRow>=kMaxRows-1) return;
20     if (newCol>=kMaxCols-1) return;
21
22     if ((terrain[newRow][newCol]==kGround) ||
23     (terrain[newRow][newCol]==kWater))
24     {
25         row=newRow;
26         col=newCol;
27     }
28
29     if (terrain[newRow][newCol]==kFood)
30     {
31         row=newRow;
32         col=newCol;
33         terrain[row][col]=kGround;
34         state=kGoHome;
35
36         do {
37             foodRow=Rnd(2,kMaxRows)-3;
38             foodCol=Rnd(2,kMaxCols)-3;
39         } while (terrain[foodRow][foodCol]!=kGround);
40
41         terrain[foodRow][foodCol]=kFood;
42     }
43
```

```
44      if (terrain[newRow][newCol]==kPoison)
45      {
46          row=newRow;
47          col=newCol;
48          terrain[row][col]=kGround;
49          state=kDead;
50
51          do {
52              poisonRow=Rnd(2,kMaxRows)-3;
53              poisonCol=Rnd(2,kMaxCols)-3;
54          } while (terrain[poisonRow][poisonCol]!=kGround);
55
56          terrain[poisonRow][poisonCol]=kPoison;
57      }
58  }
```

Prvo što je vidljivo je deklariranje varijabli. Prve dvije varijable sadrže vrijednost za koju se pčela pomiče. Sljedeće dvije (`newRow` i `newCol`) služe za pohranu pozicije pčele u svijetu. Posljednje četiri predstavljaju nove pozicije hrane i otrova nakon što su konzumirani. U početku funkcije nasumično se generiraju brojevi između -1 i 1 koji se dodjeljuju varijablama `rowMove` i `colMove`. Nakon kalkulacije zbrajamo nove brojeve i trenutnu lokaciju pčele u varijable `newRow` i `newCol`. `NewRow` i `newCol` predstavljaju novu lokaciju na koju će pčela otići, no prvo podaci moraju biti provjereni da li se nalaze unutar svijeta. Sljedeći blok `if` funkcija je zadužen za provjeru valjanosti te ako se pronađe koordinata koja nije moguća izlazi se iz funkcije.

Sada kada je pozicija valjana, provjerava se njezin tip. U slučaju da je tip trenutne lokacije `kWater` ili `kGround` pčela staje na njega no ne dolazi do promjene stanja pošto je pčela u potrazi za hranom.

Ako polje na koje pčela treba stati je tipa `kFood`, njezine koordinate postaju koordinate tog polja, to polje postaje tipa `kGround` (pošto je pčela pojela hranu), pčelino stanje prelazi u `kGoHome` te u `do-while` se generira nasumično novo polje hrane u svijetu (polje mora biti tipa `kGround`).

Druga mogućnost je nailazak pčele na polje s otrovom. Ako je polje otrovno, pčeline koordinate postaju jednake onima od polja, tip polja prelazi u `kGround` (pčela je otrovana) i pčelino stanje postaje `kDead`. U `do-while` petlji nasumično se generira novo otrovno polje u svijetu (polje mora biti tipa `kGround`).

3.2.3.5. Funkcija GoHome

Nakon pronađene hrane pčela leti prema svojoj košnici i ostaje u ovom stanju sve dok ne dođe kući ili ako stane na polje s otrovom. Isječak koda 18. prikazuje implementirano ponašanje pčele u stanju `kGoHome`.

Isječak kôda 18: Funkcija GoHome, prema uzoru na [1, str. 248 - 249]

```
1  void ai_Entity::GoHome(void)
2  {
3      int rowMove;
```

```
4       int colMove;
5       int newRow;
6       int newCol;
7       int homeRow;
8       int homeCol;
9       int index;
10      int poisonRow;
11      int poisonCol;
12
13      if (type==kHoneyBee)
14      {
15          homeRow=kHoneyHomeRow;
16          homeCol=kHoneyHomeCol;
17      }
18      else
19      {
20          homeRow=kBumbleHomeRow;
21          homeCol=kBumbleHomeCol;
22      }
23
24      if (row<homeRow)
25          rowMove=1;
26      else if (row>homeRow)
27          rowMove=-1;
28      else
29          rowMove=0;
30
31      if (col<homeCol)
32          colMove=1;
33      else if (col>homeCol)
34          colMove=-1;
35      else
36          colMove=0;
37          newRow=row+rowMove;
38          newCol=col+colMove;
39
40      if (newRow<1) return;
41      if (newCol<1) return;
42      if (newRow>=kMaxRows-1) return;
43      if (newCol>=kMaxCols-1) return;
44
45      if (terrain[newRow][newCol]!=kPoison)
46      {
47          row=newRow;
48          col=newCol;
49      }
50      else
51      {
52          row=newRow;
53          col=newCol;
54          terrain[row][col]=kGround;
55          state=kDead;
56
```

```
57          do {
58              poisonRow=Rnd(2,kMaxRows)-3;
59              poisonCol=Rnd(2,kMaxCols)-3;
60          } while (terrain[poisonRow][poisonCol]!=kGround);
61
62          terrain[poisonRow][poisonCol]=kPoison;
63      }
64
65      if ((newRow==homeRow) && (newCol==homeCol))
66      {
67          row=newRow;
68          col=newCol;
69          state=kThirsty;
70
71          for (index=0; index < kMaxEntities; index ++)
72              if (entityList[index].type==0)
73              {
74                  entityList[index].New(type,
75                  kForage,
76                  homeRow,
77                  homeCol);
78                  break;
79              }
80      }
81  }
```

Početne deklarirane varijable su jednake onima u funkciji `kForage`, no vidljivo je da su dodane dvije nove `homeRow` i `homeCol`. To su koordinate kojima će se odlučiti je li pčela uspješno stigla kući. Varijabla `index` će biti od pomoći prilikom dodavanja nove pčele u svijet.

U početku dodjeljuje se vrijednosti varijablama `homeRow` i `homeCol` ovisno o vrsti pčele o kojoj se radi bila ona `HoneyBee` ili `BumbleBee`. Kada se dohvate koordinate košnice iz globalno zadanih varijabli, pomoću jednostavnog algoritma traži se put prema košnici. Pomoću naredbi `if`, `else if` i `else` provjerava se odnos trenutne pozicije pčele prema poziciji njene košnice. Ako je njezin trenutni redak manji od redka košnice, pčela će se pomaknuti za 1 red više, što znači da se varijabli `rowMove` dodjeljuje vrijednost 1. Ako je pčelin redak veći od redka košnice, `rowMove` dodjeljuje se vrijednost -1, te ako niti jedan uvjet od ranije nije ispunjen ostaje zaključiti da su pčela i košnica u istom redu te se varijabli `rowMove` dodjeljuje vrijednost 0. Sličan algoritam je primjenjen na računanje i vrijednosti za stupce, odnosno `colMove` varijablu.

Nakon što su `rowMove` i `colMove` varijable dodijeljene, pčelinim trenutnim koorinatama se dodavaju `rowMove` i `colMove` i dobiva se nova pozicija pčele (`newCol` i `newRow`). Kao i u prijašnjoj funkciji, provjerava se jesu li nove koordinate valjane i da ne odstupaju od veličine svijeta, a to se odrađuje pomoću poznatog bloka `if` naredbi. U sljedećoj `if` naredbi provjerava se treba li pčela naići na otrovno polje. Sve dok pčelino sljedeće polje nije otrovno ona se kreće. Ako ipak naiđe na otrovno polje, ono postaje tipa `kGround`, pčelino stanje prelazi u `kDead` te se u `do-while` petlji nasumično generira novo otrovno polje u svijetu.

Posljednji `if` uvjet provjerava jesu li koordinate pčele i košnice jednake. Ako je uvjet ispunjen, pčeline koordinate postaju jednake koordinatama košnice, njezino stanje se mijenja

u `kThirsty`, te se u for petlji stvara nova pčela koja je istog tipa, stanje joj je `kForage` i ima koordinate od košnice.

3.2.3.6. Funkcija Thirsty

Sljedeća funkcija je zaslužna za stanje `kThirsty`. U ovo stanje pčela dolazi kada se je uspješno vratila kući. Sada umjesto hrane pčela nasumično leti svijetom i traži vodu te pronalaskom vode nastavlja tražiti hranu. Isječak koda 19 prikazuje funkciju `Thirsty`.

Isječak kôda 19: Funkcija Thirsty, prema uzoru na [1, str. 251 - 252]

```
1  void ai_Entity::Thirsty(void)
2  {
3      int rowMove;
4      int colMove;
5      int newRow;
6      int newCol;
7      int foodRow;
8      int foodCol;
9      int poisonRow;
10     int poisonCol;
11
12     rowMove=Rnd(0,2)-1;
13     colMove=Rnd(0,2)-1;
14     newRow=row+rowMove;
15     newCol=col+colMove;
16
17     if (newRow<1) return;
18     if (newCol<1) return;
19     if (newRow>=kMaxRows-1) return;
20     if (newCol>=kMaxCols-1) return;
21
22     if ((terrain[newRow][newCol]==kGround) ||
23     (terrain[newRow][newCol]==kFood))
24     {
25         row=newRow;
26         col=newCol;
27     }
28
29     if (terrain[newRow][newCol]==kWater)
30     {
31         row=newRow;
32         col=newCol;
33         terrain[row][col]=kGround;
34         state=kForage;
35
36         do {
37             foodRow=Rnd(2,kMaxRows)-3;
38             foodCol=Rnd(2,kMaxCols)-3;
39         } while (terrain[foodRow][foodCol]!=kGround);
40
41         terrain[foodRow][foodCol]=kWater;
```

```
42      }
43
44      if (terrain[newRow][newCol]==kPoison)
45      {
46          row=newRow;
47          col=newCol;
48          terrain[row][col]=kGround;
49          state=kDead;
50
51          do {
52              poisonRow=Rnd(2,kMaxRows)-3;
53              poisonCol=Rnd(2,kMaxCols)-3;
54          } while (terrain[poisonRow][poisonCol]!=kGround);
55
56          terrain[poisonRow][poisonCol]=kPoison;
57      }
58  }
```

Vidljivo je da su dijelovi koda jednaki onima iz funkcije `kForage` i `kGoHome` i nisu potrebni daljnjeg opisivanja. Nakon provjere valjanosti koordinata unutar svijeta nailazi if uvjet koji provjerava je li sljedeća lokacija tipa `kGround` ili `kFood` te ako je pčela se samo pomiče na nju. Ako je lokacija tipa `kWater` pčela se pomiče na nju, tip lokacije se mijenja u `kGround`, stanje pčele prelaz i u `kForage` te pomoću `do-while` petlje generira se novo polje vode. Ako je lokacija tipa `kPoison` pčela umire i generira se novo polje što je već poznato iz prije spomenutih funkcija.

3.2.3.7. Funkcija Dead

Prilikom doticaja pčele s otrovom njeno stanje prelazi u `kDead` i prijelaz u ovo stanje je moguć iz svih drugih stanja. Isječak koda 20 prikazuje funkciju `Dead`.

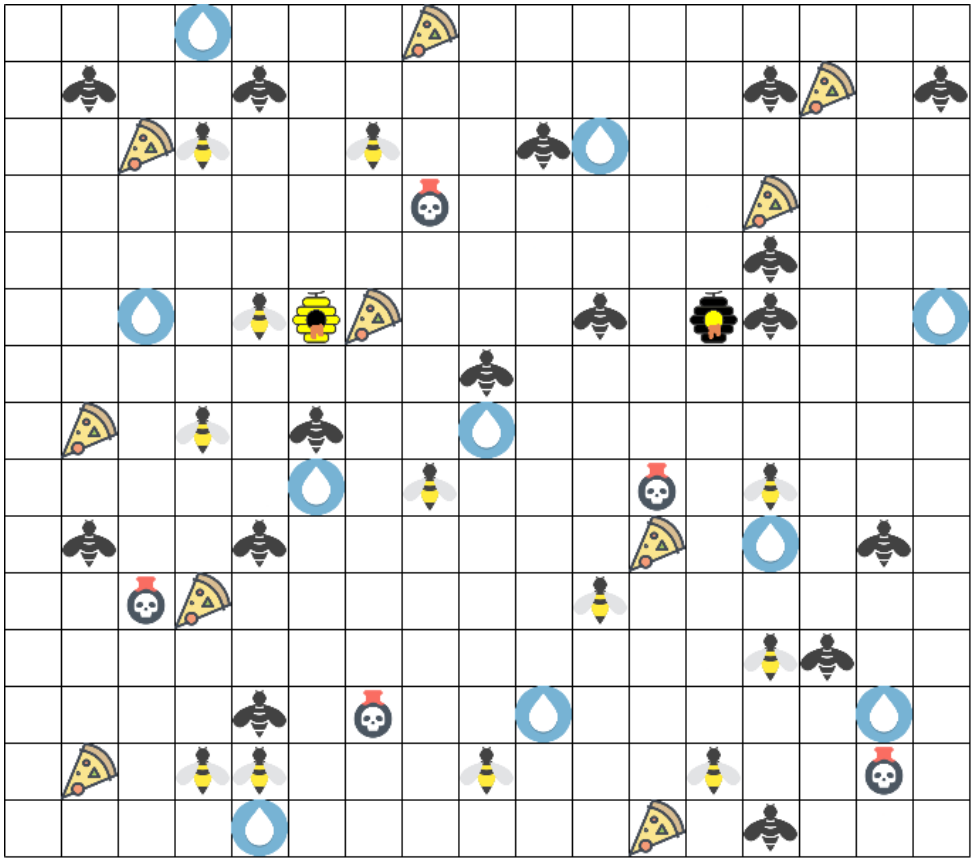
Isječak kôda 20: Funkcija Dead

```
1  void ai_Entity::Dead(void)
2  {
3   type=0;
4  }
```

Funkcija `Dead` služi za brisanje pčele iz polja kako bi se na njeno mjesto mogle upisati nove. Pošto se u funkciji `GoHome` prilikom stvaranja nove pčele u polju pčela provjerava je li sljedeći zapis `type` jednak 0, sve što je potrebno da se oslobodi novo mjesto je mrtvoj pčeli varijablu `type` postaviti na 0.

3.2.3.8. Rezultat

Kako su sada definirane sve četiri funkcije, sve što ostaje je pokrenuti simulaciju i vidjeti napredak pčela u svijetu. Slika 6 prikazuje jedan od mogućih rezultata simulacije nakon nekog vremena.



Slika 6: Izgled svijeta

4. Izrada videoigre sa konačnim automatom u programu Unity

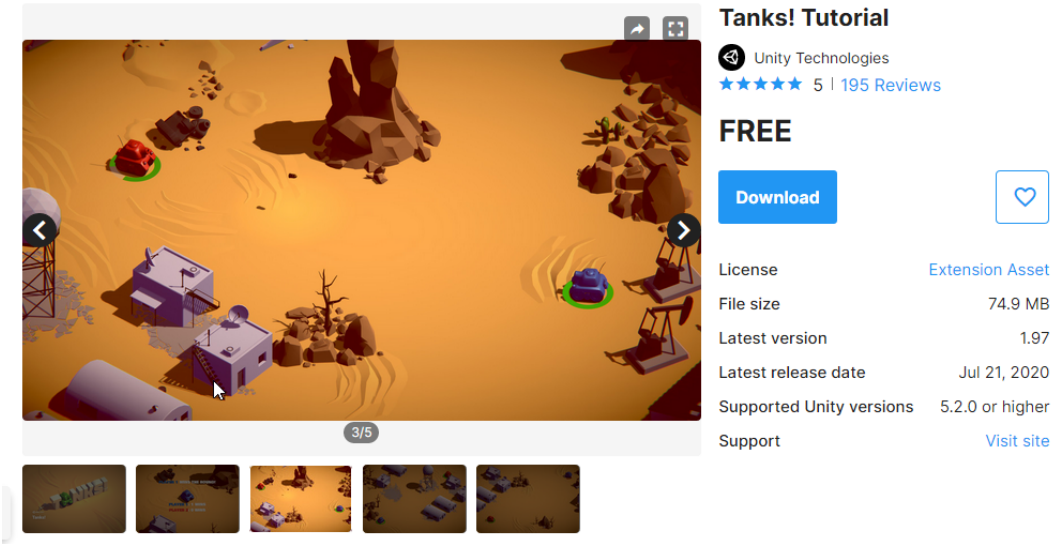
U ovom djelu rada sve prijašnje spomenute informacije će se primjeniti za izradu videoigre u programu Unity. Zadatak ovog dijela rada je prikazati jednostavan model konačnog automata i način njegovog razvijanja u spomenutom programu te na kraju vizualno promatrati i njegov rad. videoigra će se sastojati od dva tenka koja se nalaze u pustinjskoj mapi od kojih je jedan upravljan sa strane igrača, a drugi od strane umjetne inteligencije odnosno konačnog automata.

Kao polazni zadatak, prvo će se izraditi model mape na kojoj će se tenkovi kretati, a samim time i same modele tenkova. Pomoću navigacijskog agenta koji je dio programa Unity, upravljati će se kretanjem neigrajućeg lika i biti će pokazani svi potrebni postupci kako bi se takav agent uopće izradio. Naravno, više o tome će biti spomenuto u nastavku rada pod naslovom "Izrada svijeta i navigacijske mreže".

Nakon izrađenih modela i uspostave navigacijske mreže koja je potrebna za korištenje navigacijskog agenta, dolazi se do dijela programiranja konačnog automata tj. programiranja stanja, ponašanja i uvjeta neigrajućeg tenka. Program Unity u svom sastavu sadrži alat koji se zove `Animator` i pomoću njega će se definirati stanja koja tenk može imati i uvjete koji se moraju zadovoljiti za prijelaz u drugo stanje. Sve ovo će biti prikazano i detaljnije objašnjeno u nastavku rada pod naslovom "Izrada konačnog automata i skripti".

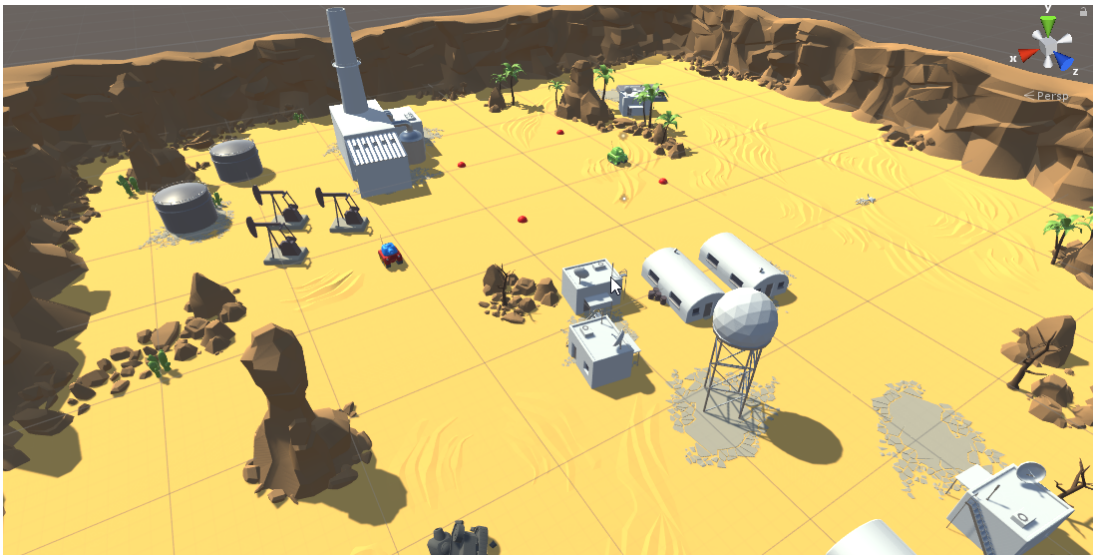
4.1. Izrada svijeta i navigacijske mreže

Kao što je prije navedeno, prvo što je potrebno napraviti je vizualni prikaz videoigre, točnije mapa na kojoj će se nalaziti tenkovi te sami tenkovi. Za lakšu izradu korišten je besplatni paket modela koji je moguće nabaviti na `asset store-u` unutar programa Unity.



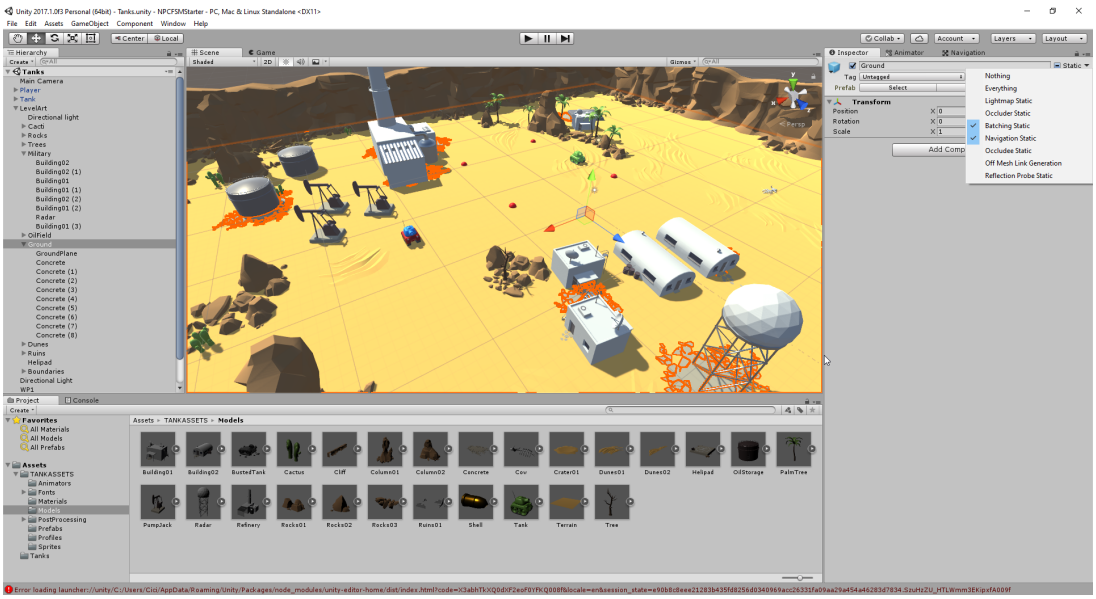
Slika 7: Prikaz asset store-a

Pomoću gotovih modela napravljena je mapa i tenkovi, te time je završen prvi korak u izradi videoigre.



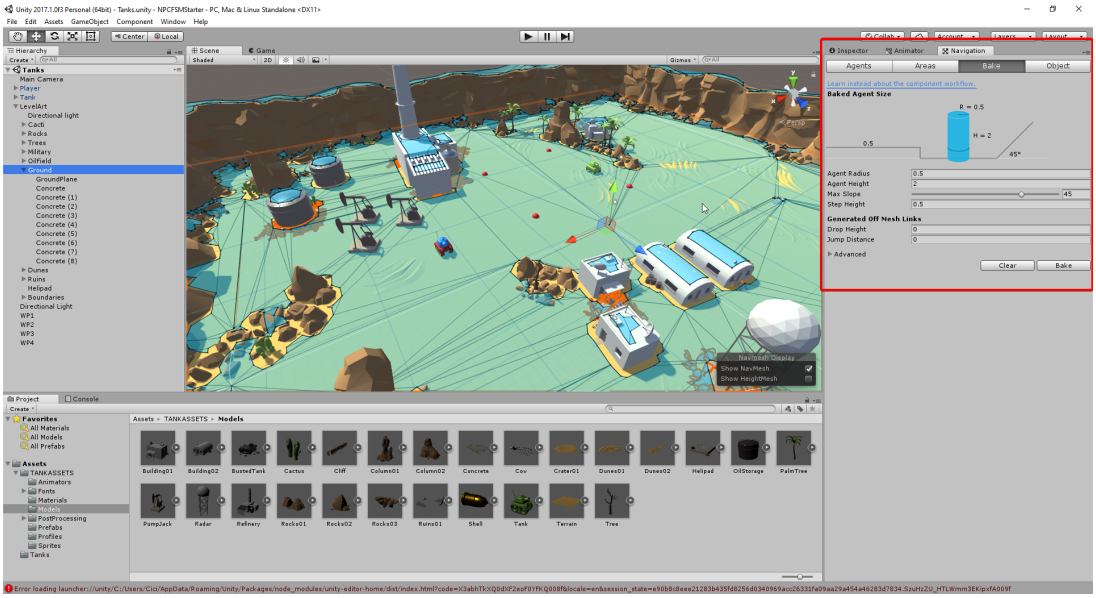
Slika 8: Prikaz svijeta

Pošto se koristi navigacijski agent za kontroliranje kretnji neigrajućeg lika, prvo je potrebno izraditi navigacijsku mrežu koja predstavlja dozvoljeno područje koje navigacijski agent može koristiti. Prvi korak je odabrati sve objekte unutar videoigre koji će biti uključeni u navigacijsku mrežu te im dodjeliti opciju `navigation static`.



Slika 9: Dodjela navigation static opcije

Sada kada program Unity zna koji su objekti uključeni u kreiranje navigacijske mreže pomoću kartice `navigation` i podkartice `bake` postavljaju se svojstva agenta koji će služiti kao primjer za izračun svih dostupnih mjesta na uključenim objektima. Klikom na gumb `bake` kreira se navigacijska mreža na svim uključenim objektima ovisno o agentovim svojstvima koje smo ranije zadali. Sve što je obojano plavo u prikazu svijeta predstavlja navigacijsku mrežu, odnosno poziciju na koju agent može biti pozicioniran.



Slika 10: Dodjela navigation static opcije

Ovim koracima je završena priprema svijeta i modela koji su potrebni za daljni razvoj videoigre. Sada slijedi programski dio u kojem tenkovima dajemo "život". Kreiranje osnovnih kontrola za igrača, te izrada konačnog automata za neigrajućeg tenka.

4.2. Izrada konačnog automata i skripti

U ovom dijelu rada prikazan je način izrade konačnog automata u programu Unity. Za primjer će se uzeti zeleni tenk koji je ranije ubačen u videoigru. Konačni automat će se sastojati od 3 stanja: `Patroliraj` (eng. Patrol), `Hvataj` (eng. Chase), `Napadaj` (eng. Attack). Tenk će imati označene `međutočke` (eng. waypoints) koje predstavljaju put patrole, te ovisno o blizini igrača stanje će se mijenjati. Opisani konačni automat formalno je zapisan:
$\Sigma =$ {u dometu, izvan dometa, u vidokrugu, izvan vidokruga}
$\Gamma =$ {pucaj, približavaj se, prati put}
$S =$ {patroliraj, hvataj, napadaj}
$S_0 =$ {patroliraj}

Tablica 3: δ

	Patroliraj	Hvataj	Napadaj
u vidokrugu	Hvataj	...	...
izvan vidokruga	...	Patroliraj	...
u dometu	...	Napadaj	...
izvan dometa	...	...	Hvataj

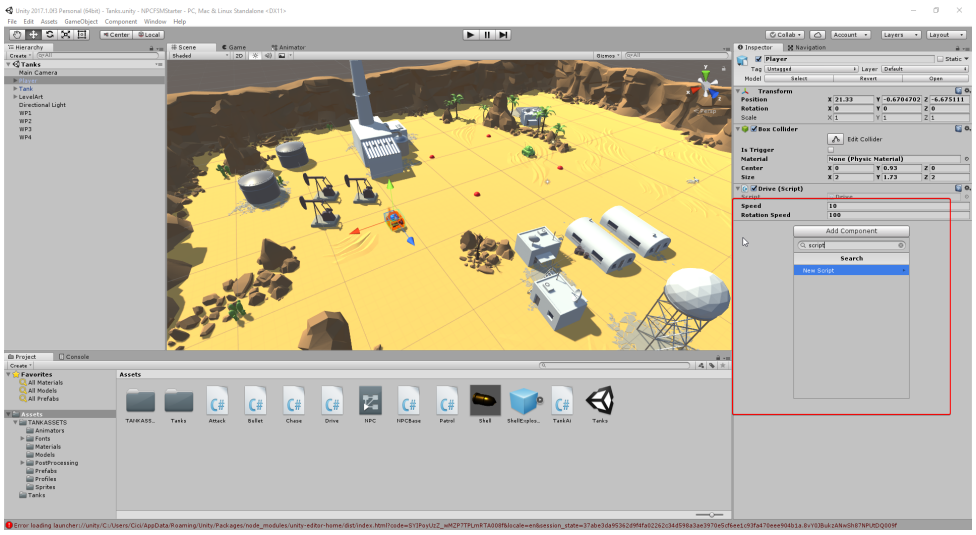
Tablica 4: ω

	Patroliraj	Hvataj	Napadaj
u vidokrugu	Približavaj se	...	...
izvan vidokruga	...	Prati put	...
u dometu	...	Pucaj	...
izvan dometa	...	...	Približavaj se

Prije početka izrade konačnog automata prvo će se dodjeliti kontrole kretanja igraču kako bi se kasnije mogao testirati kod konačnog automata.

4.2.1. Kontrole igrača

Kao što je vidljivo na slici 11, klikom na željeni objekt (u ovom slučaju tenk igrača) u kartici `inspector` postoji gumb `Add Component` koji služi za dodavanje raznih stavki kao što su `fizika` (eng. Physics), `efekti` (eng. Effects) i druge. Tako ovdje postoji i opcija dodavanja skripti. Upisivanjem imena koje nije zauzeto u programu Unity (nije nekakva stavka), kao što je u primjeru `Drive`, program automatski daje prijedlog kreiranja skripte sa tim imenom. Sada kada je skripta dodana na objekt, potrebno je dodati i kod koji će govoriti što taj objekt radi. Pisanje koda se odvija u programu Visual Studio, no radi pregledivosti kod će biti prikazan kao tekst umjesto slike.

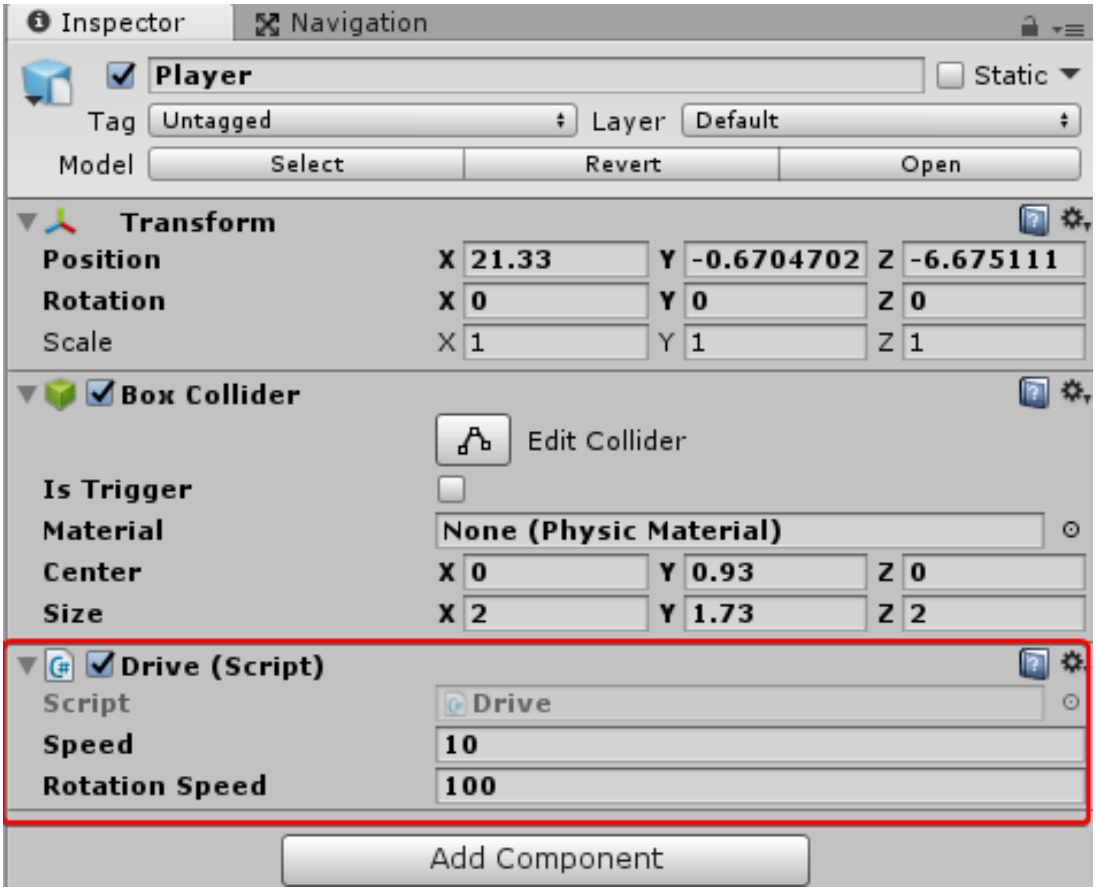


Slika 11: Dodavanje skripti objektu

Isječak kôda 21: Skripta Drive

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Drive : MonoBehaviour {

        public float speed = 10.0F;
    public float rotationSpeed = 100.0F;

    void Update() {
        float translation = Input.GetAxis("Vertical") * speed;
        float rotation = Input.GetAxis("Horizontal") * rotationSpeed;
        translation *= Time.deltaTime;
        rotation *= Time.deltaTime;
        transform.Translate(0, 0, translation);
        transform.Rotate(0, rotation, 0);
        }
}
```

U isječku koda 21 prikazana je skripta `Drive`, u početku se nalaze inicijalizirane varijable `speed` i `rotationSpeed`, dodjeljivanjem pristupa public omogućuno je da se te vrijednosti mijenjaju unutar programa Unity što je prikazano na slici 12.



Slika 12: Promjena vrijednosti varijabli unutar Unity-a

Nakon varijabli nalazi se funkcija `Update` tipa `void`. Funkcija `Update` je slična funkciji `While(true)` pošto se ona izvodi neograničeno puta. Unutar funkcije prvo se nalaze dvije varijable: `translation` i `rotation`. Varijabla `translation` sadrži vrijednost `speed`, koja je ranije inicijalizirana, pomnoženu sa vrijednosti `Input.GetAxis("Vertical")` koja sadrži vrijednost između -1 i 1. Pošto je u Unity-u početno zadano da su strelice `up` i `down` zaslužne za `vertical`, to znači da pritiskom `up` dobivamo vrijednost 1, a `down` -1. Varijabla `rotation` sadrži vrijednost `rotationSpeed` pomnoženu sa vrijednosti `Input.GetAxis("Horizontal")` (-1,1). Slično kao i kod `Input.GetAxis("Vertical")` vrijednost ovisi o pritisnutoj strelici na tipkovnici, no pošto sada koristimo horizontalnu os strelice `left` i `right` su te o kojima ovisi vrijednost.

Pošto se funkcija `update` ponovno pokreće svaku novu sličicu (eng. frame) vrijednosti `translation` i `rotation` treba pomnožiti sa vrijednosti `time.deltaTime` kako bi dobili ujednačenu vrijednost neovisno o količini sličica koje računalo može proizvesti u sekundi (eng. FPS-Frames Per Second). Sada na kraju jedino što preostaje je pomaći tenk unutar svijeta. Pomoću `transform` dohvaćamo svojstvo `transform` tenka te pomoću `translate` pomičemo poziciju ili `rotate` ga rotiramo. Unutar svake funkcije potrebno je definirati kojim osima dodajemo te vrijednosti (x, y, z). Ovime smo završili kreaciju kontrola za igračev tenk.

4.2.2. Neigrajući tenk

Sada je na redu izraditi sva moguća stanja dostupna konačnom automatu. Imati ćemo 3 stanja: Patrol, Chase i Attack. No prije pisanja koda za pojedina stanja, prvo će se napraviti osnovna klasa kako bi se neke repetitivne stvari izbjegle. Kreira se skripta imena `NPCBase` te njen kod je prikazan u isječku koda 22.

Isječak kôda 22: Skripta NPCBase

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class NPCBase : StateMachineBehaviour {
6
7          public GameObject NPC;
8          public GameObject opponent;
9          public UnityEngine.AI.NavMeshAgent agent;
10         public float accuracy = 3.0f;
11
12     public override void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo
       , int layerIndex)
13     {
14                 NPC = animator.gameObject;
15                 opponent = NPC.GetComponent<TankAi>().getplayer();
16                 agent = NPC.GetComponent<UnityEngine.AI.NavMeshAgent>();
17     }
18 }
```

Prvo što treba uočiti je bazna klasa `StateMachineBehaviour`, ona se koristi svagdje

gdje imamo korištenje skripti u stanjima. Sljedi deklariranje javnih varijabli, dvije `GameObject` varijable `NPC` i `opponent`. Tim varijablama ćemo unutar programa Unity dodjeliti objekte, točnije tenk igrača i neigrajući tenk. Sljedeća varijabla će biti zaslužna za dohvaćanje agenta za navigaciju koji smo spominjali i varijabla `accuracy` kojom određujemo preciznost odnosno dozvoljeni odmak neigrivog tenka od međutočke. Funkcija `OnStateEnter` se pokreće prilikom ulaska u stanje. Varijabli `NPC` dodjeljujemo `gameObject` na koji je animator komponenta spojena (neigrajući tenk). Varijabla `opponent` dohvaća igračev tenk preko vanjske skripte TankAi koja je komponenta neigrajućeg tenka te će biti ubrzo implementirana. Varijabli `agent` dodjeljujemo komponentu Nav Mesh Agent koja je dio neigrajućeg tenka.

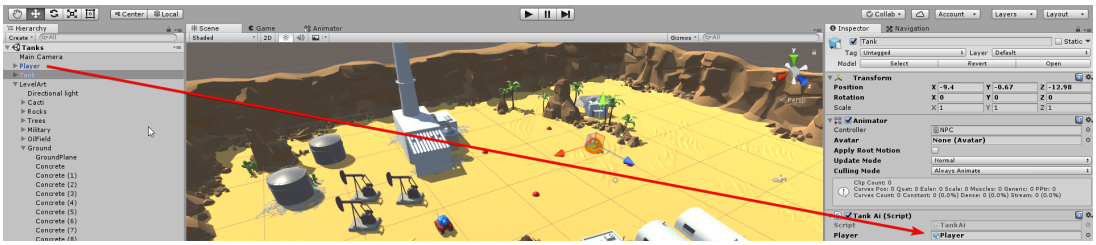
Kako bi ovaj kod bio valjan, sada se dodaje skripta TankAi, te je njezin kod prikazan u isječku koda 23.

Isječak kôda 23: Skripta TankAi

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TankAi : MonoBehaviour {

        public GameObject player;
        public GameObject getplayer()
    {
                return player;
    }


        // Use this for initialization
        void Start () {

        }

        // Update is called once per frame
        void Update () {

        }
}
```

Javnom deklaracijom varijable `player` omogućeno pridruživanje igračevog tenka unutar Unity-a, te jednostavnom funkcijom `getplayer()` omogućen je dohvat igračevog tenka unutar drugih skripti. Prikaz pridruživanja gameObject-a `player` na skriptu TankAi vidljiv je na slici 13.

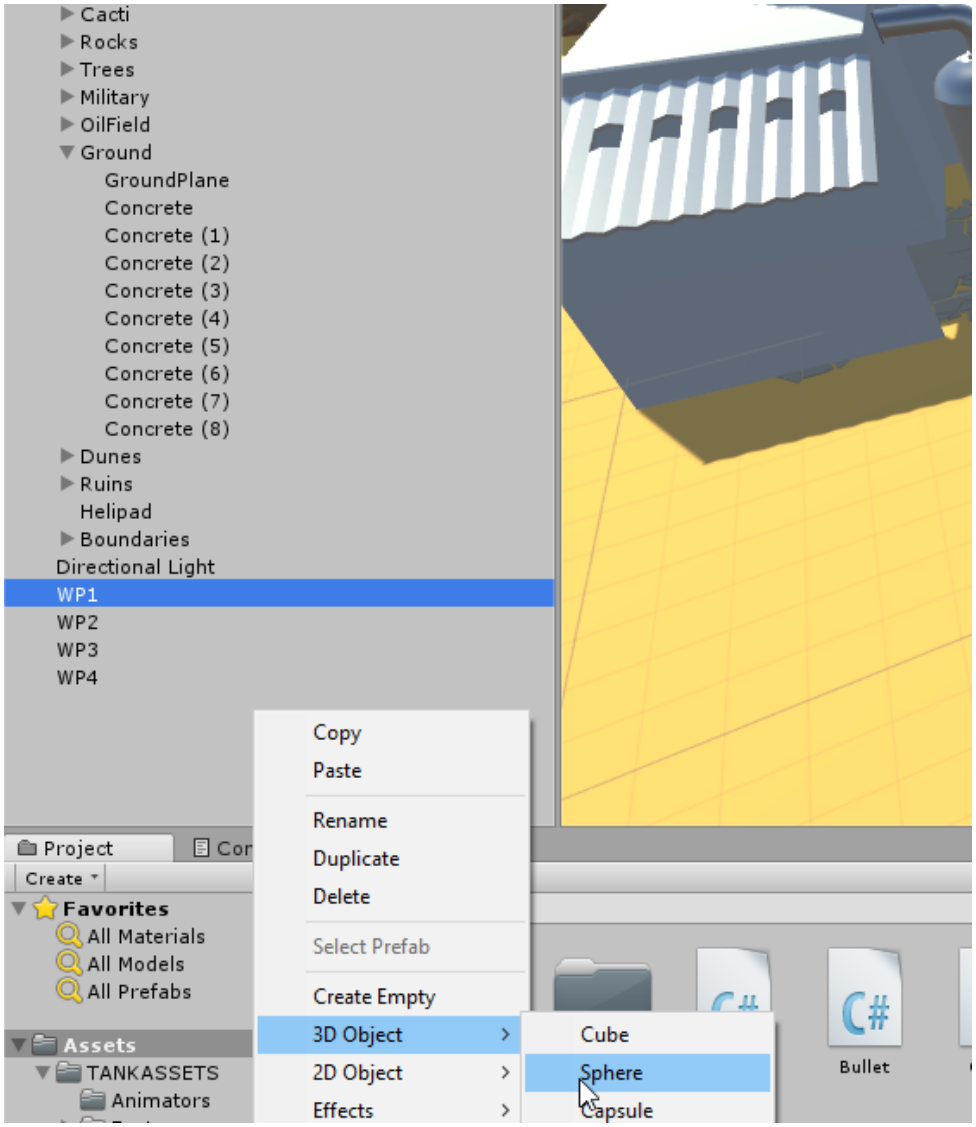
Sada kada su osnovne skripte implementirane, kreće izrada stanja unutar kartice Animator. Animator je sredstvo unutar Unity-a koje služi za kreiranje animacija pomoću raznih stanja i ponašanja unutar stanja. Upravo su animacije prikaz rada metode umjetne inteligencije (u ovom slučaju konačni automat) koliko god one bile kompleksne ili jednostavne. To će biti prikazano pomoću 3 stanja u nastavku.



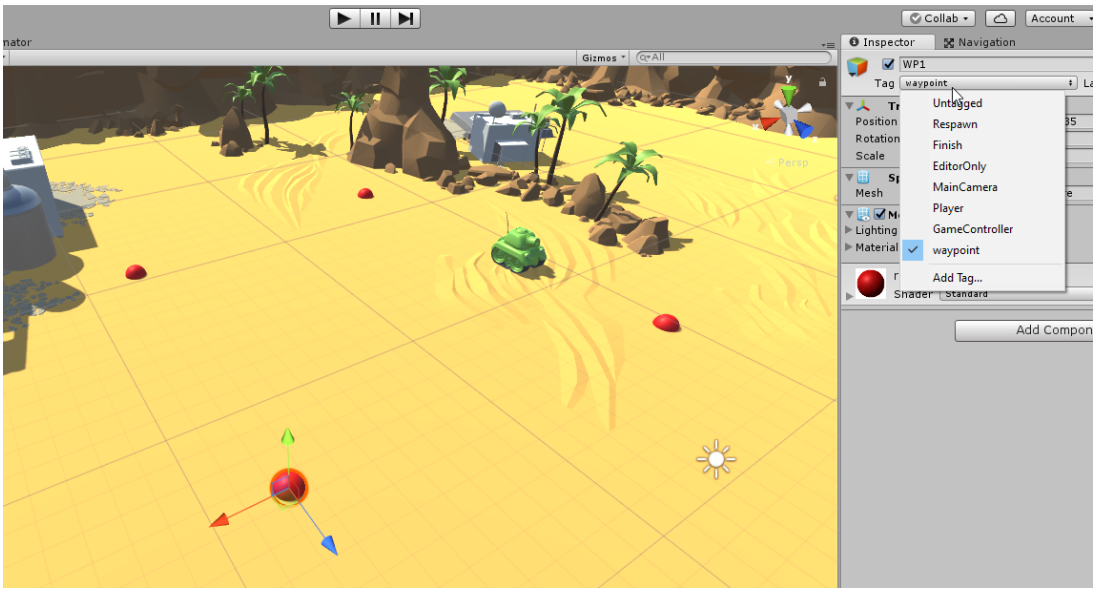
Slika 13: Pridruživanje gameObject-a

4.2.2.1. Funkcija Patrol

U početku videoigre prvotno stanje neigrajućeg tenka je `patrol`. Pošto je tenk u stanju `patrol`, njegovo ponašanje unutar tog stanja će biti vidljivo tako što će se on kretati po međutočkama označene crvenom sferom na prikazu mape. Prvi zadatak je ubaciti te sfere u videoigru te im dodjeliti nekakvu oznaku (eng. tag) kako bi ih kasnije bilo moguće dohvatiti unutar samog koda. Kreiranje sfere i dodjelivanje oznake vidljivo je na slikama 14 i 15.

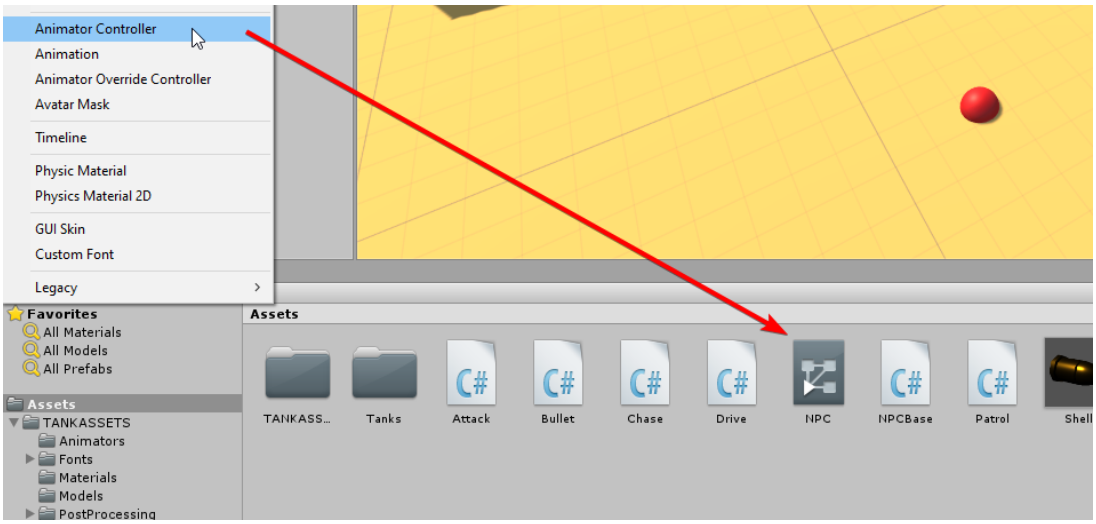


Slika 14: Kreiranje sfere



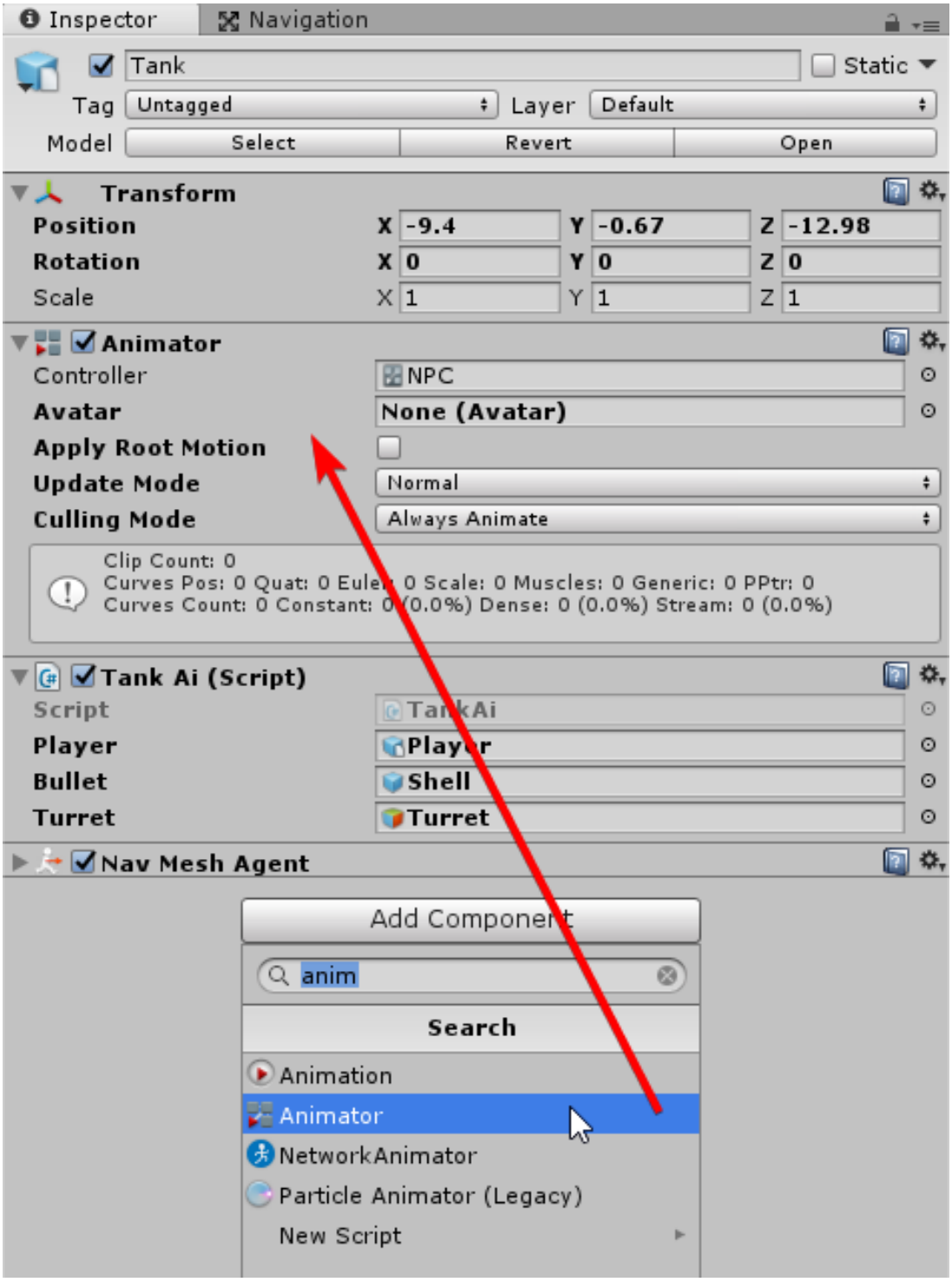
Slika 15: Dodjeljivanje oznake

Sada kada je ovaj dio gotov, potrebno je kreirati novo stanje unutar animatora i tom stanju dodati skriptu koja će njime upravljati. Kako bi u krajnjem dijelu kreirana stanja bila dodijeljena neigrajućem tenku, potrebno je prvo kreirati animator controller koji će sadržavati spomenuta stanja, te samom tenku dodati novu komponentu animator na koji će se povezati kreirani controller. Primjer izrade controllera, stanja i skripti prikazan je na sljedećim slikama.



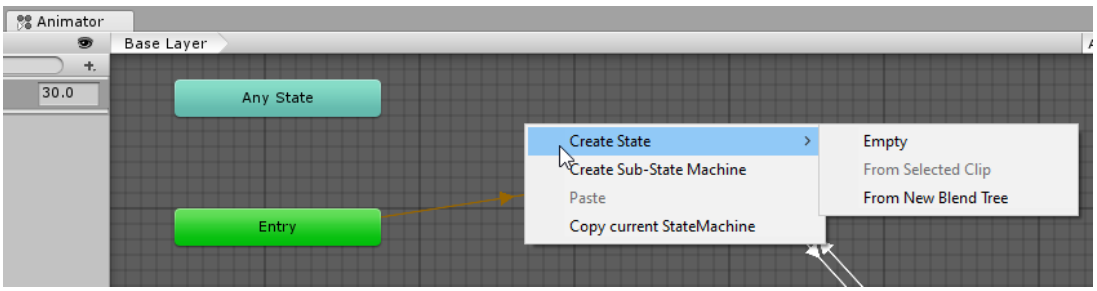
Slika 16: Kreiranje animator controllera

Kao što je vidljivo na slici 16, kreirani controller je nazvan NPC. Nakon dodavanja animator komponente neigrajućem tenku, dodjijeljen je i kreirani NPC controller unutar animator prozora što je vidljivo na slici 17.

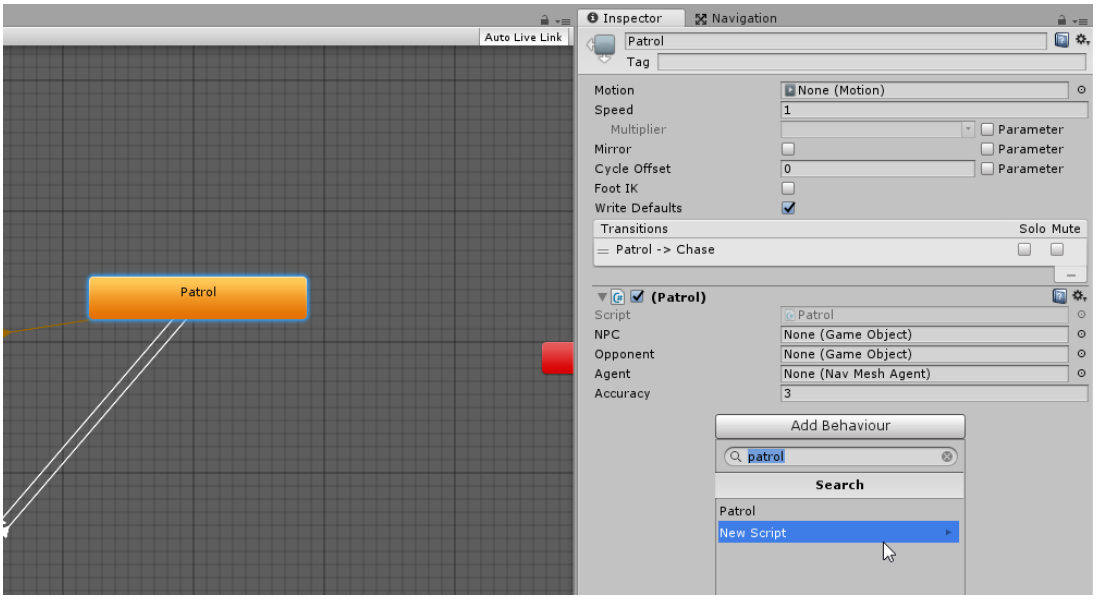


Slika 17: Dodavanje komponente animator

Sada otvaranjem controllera NPC dolazi se do kartice animator u kojoj se kreiraju stanja, te im se dodaje ponašanje (eng. behaviour) odnosno skripta koja njime upravlja. Prikaz kreiranja stanja i skripti na slikama 18 i 19.



Slika 18: Dodavanje stanja



Slika 19: Dodavanje ponašanja

Kod unutar kreirane skripte prikazan je u isječku koda 24.

Isječak kôda 24: Skripta Patrol

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Patrol : NPCBase {
6
7          GameObject[] waypoints;
8          int currentWP;
9
10         void Awake()
11     {
12                 waypoints = GameObject.FindGameObjectsWithTag("waypoint");
13     }
14         override public void OnStateEnter(Animator animator, AnimatorStateInfo
            stateInfo, int layerIndex)
```

```
15          {
16                  base.OnStateEnter(animator, stateInfo, layerIndex);
17                  currentWP = (int)Random.Range(0,4);
18          }
19
20          override public void OnStateUpdate(Animator animator, AnimatorStateInfo
             stateInfo, int layerIndex)
21          {
22                  if (waypoints.Length == 0) return;
23
24          if (Vector3.Distance(waypoints[currentWP].transform.position,NPC.transform.
             position) < accuracy)
25          {
26                          currentWP++;
27              if (currentWP >= waypoints.Length)
28              {
29                                  currentWP = 0;
30              }
31          }
32
33                  agent.SetDestination(waypoints[currentWP].transform.position);
34          }
35
36          override public void OnStateExit(Animator animator, AnimatorStateInfo
             stateInfo, int layerIndex)
37          {
38
39          }
40  }
```

U početku koda može se primjetiti da klasa `Patrol` nasljeđuje klasu `NPCBase` koja je ranije spomenuta, radi toga osnovne stvari definirane tamo ovdje su prisutne i nije ih potrebno opet definirati. U nastavku vidljivo je deklarirano polje `waypoints` tipa `GameObject` u koje će se spremati sfere koje smo ranije kreirali. Isto tako kreirana je varijabla `currentWP` tipa `int` u koje će biti spremljen podatak pozicije trenutačne međutočke unutar polja. Funkcijom `Awake` osigurava se izvršavanje koda prije početka rada skripte te unutar funkcije vidljivo je dohvaćanje svih objekata videoigre koji sadrže oznaku `waypoint` i njihovo spremanje unutar polja `waypoints`. Prilikom ulaska u stanje funkcijom `base.OnStateEnter(animator, stateInfo, layerIndex)` pokreće se funkcija `OnStateEnte` naše bazne klase `NPCBase` kako bi se inicijalizirale sve varijable koje su tamo navedene, te nakon toga se odabire nasumična međutočka i taj podatak se sprema u varijablu `currentWP`.

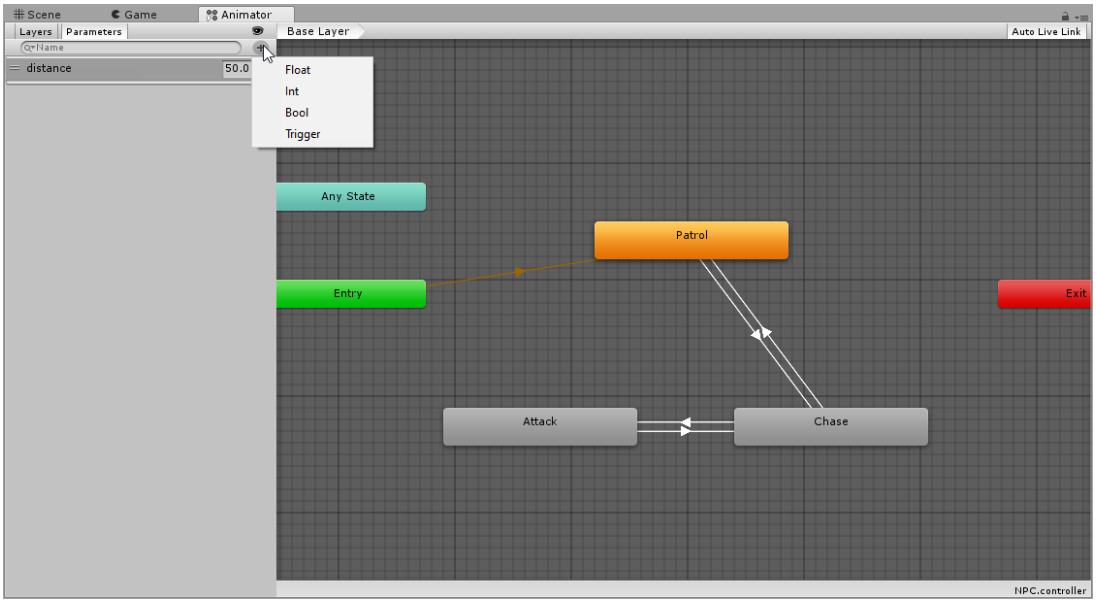
U funkciji `update` koja se stalno iznova izvršava, prvo se provjerava da li je `waypoints` polje prazno, te ako je kod se prekida. U suprotnom nastavlja se na provjeru da li je udaljenost između trenutne međutočke (currentWP) i neigrajućeg tenka manja od vrijednosti varijable `accuracy` (accuracy = 3.0f) te ispunom tog uvjeta prelazi se na novu točku u nizu polja. Sljedeći `if` uvjet provjerava da li je pozicija trenutne međutočke veća ili jednaka od ukupnog broja međutočaka, točnije da li je trenutna međutočka posljednja u polju, te ako je, za trenutnu međutočku postavlja opet prvu i time se osigurava da tenk ne ostane bez međutočki. Na kraju se na-

lazi funkcija `agent.SetDestination(waypoints[currentWP].transform.position)` koja navigacijskom agentu zadaje koordinate prema kojima se treba kretati, a te koordinate predstavljaju poziciju trenutne međutočke unutar videoigre. Nakon toga agent sam računa najoptimalniji put pomoću navigacijske mreže koja je ranije kreirana.

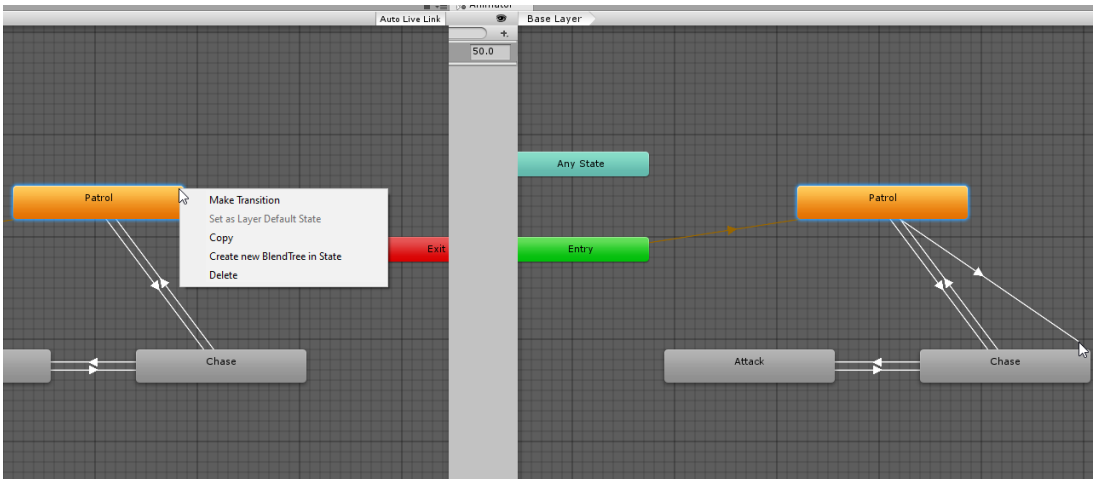
4.2.2.2. Funkcija Chase

Prilikom patrole neigrajućeg tenka ako igrač svojim tenkom dođe na određenu udaljenost, neigrajući tenk prelazi u stanje `chase` i pokušava smanjiti udaljenost kako bi došao u dovoljan domet da ga napadne.

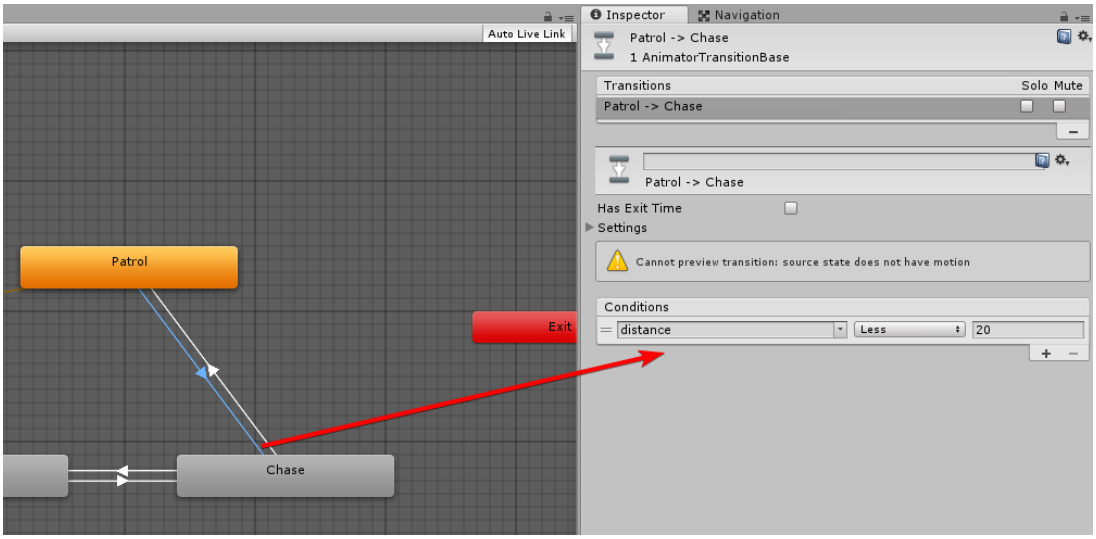
Kako bi se rješila ova funkcionalnost mora se kreirati novo stanje i njezina skripta, te uvesti uvjet koji će aktivirati prijelaz iz jednog u drugo stanje. Za uspostavljanje uvjeta potreban je parametar koji je u ovom slučaju udaljenost između dva tenka. Slika 20 prikazuje kreaciju parametra, a slika 21 i 22 prijelaze i njihove uvjete.



Slika 20: Kreiranje parametra



Slika 21: Kreiranje tranzicijske veze



Slika 22: Dodavanje uvjeta

U nastavku je prikazan isječak koda unutar skripte Chase.

Isječak kôda 25: Skripta Chase

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Chase : NPCBase {

        override public void OnStateEnter(Animator animator, AnimatorStateInfo
         stateInfo, int layerIndex) {
                base.OnStateEnter(animator, stateInfo, layerIndex);
                agent.stoppingDistance = 2;
        }

        override public void OnStateUpdate(Animator animator, AnimatorStateInfo
         stateInfo, int layerIndex) {
                agent.SetDestination(opponent.transform.position);

        }

        override public void OnStateExit(Animator animator, AnimatorStateInfo
         stateInfo, int layerIndex) {
        }
}
```

Prilikom ulaska u stanje pokreće se bazna klasa radi inicijalizacije agenta i dohvaćanja protivnika (igračev tenk) i agentov zaustavni put (eng. stopping distance) se postavlja na vrijednost 2 (prihvatljivi radius zaustavljanja od zadane lokacije), te pošto ostatak posla prilikom hvatanja obavlja navigacijski agent sve što ostaje je u funkciji `update` pomoću funkcije `SetDestination` agentu zadati lokaciju protivnika prema kojoj se treba kretati.

Kako bi došlo do prijelaza iz jedno u drugo stanje potrebno je parametar `distance` konstantno ažurirati. Tu funkcionalnost će se dodati u skriptu TankAi. Isječak koda 26 prikazuje ažurirani kod skripte TankAi.

Isječak kôda 26: Skripta TankAi

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TankAi : MonoBehaviour {

        public GameObject player;
        public GameObject getplayer()
    {
                return player;
    }

    Animator anim;

        // Use this for initialization
```

```
16          void Start () {
17              anim = GetComponent<Animator>();
18          }
19
20          // Update is called once per frame
21          void Update () {
22                  anim.SetFloat("distance", Vector3.Distance(transform.position,
                     player.transform.position));
23          }
24  }
```

Ispod funkcije `getplayer` nalazi se deklarirana varijabla `anim` tipa `animator`, u funkciji `start` inicijalizira se varijabla `anim` tako što joj se pridružuje komponenta `animator` koja je dio neigrajućeg tenka. Na kraju u funkciji `update` postavlja se vrijednost parametra `distance` pomoću funkcije `SetFloat` koja za argumente uzima koji parametar mjenjamo (distance) i koju vrijednost zadajemo, a za vrijednost se koristi funkcija `Distance` koja vraća vrijednost udaljenosti između pozicije neigrajućeg tenka i tenka igrača.

4.2.2.3. Funkcija Attack

Kada neigrajući tenk prilikom hvatanja igračevog tenka njihovu međusobnu udaljenost dovoljno smanji njegovo stanje prelazi u `attack`. Ulaskom u stanje `attack` tenk staje i kreće pucati na igrača. Ukoliko igrač dovoljno poveća međusobnu udaljenost tenk se vraća u stanje `chase`. Isječak koda prikazuje izgled skripte `Attack`.

Isječak kôda 27: Skripta Attack

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Attack : NPCBase {
6
7          override public void OnStateEnter(Animator animator, AnimatorStateInfo
            stateInfo, int layerIndex) {
8                  base.OnStateEnter(animator, stateInfo, layerIndex);
9                  NPC.GetComponent<TankAi>().StartFiring();
10                 agent.stoppingDistance = 10;
11         }
12
13         override public void OnStateUpdate(Animator animator, AnimatorStateInfo
            stateInfo, int layerIndex) {
14                 NPC.transform.LookAt(opponent.transform.position);
15         }
16
17         override public void OnStateExit(Animator animator, AnimatorStateInfo
            stateInfo, int layerIndex) {
18                 NPC.GetComponent<TankAi>().StopFiring();
19         }
20 }
```

Kao i u prijašnjim primjerima prvo se inicijalizira bazna klasa radi dohvaćanja potrebnih varijabli. Nakon toga poziva se funkciju `StartFiring` koja se nalazi u skripti TankAi i agentov zaustavni put postavlja se na vrijednost 10 kako bi prilikom malog odmaka igrača od njega tenk i dalje nastavio pucati.

U funkciji `update` pomoću funkcije `LookAt` rotiramo neigrajući tenk prema igraču kako bi metci letjeli prema njemu. Na kraju skripte nalazi se funkcija `OnStateExit` koja govori da pri izlasku, odnosno promjeni u drugo stanje tenk prestane sa pucanjem pozivajući funkciju `StopFiring` koja se nalazi unutar skripte TankAi. U nastavku je prikazan isječak koda ažurirane skripte TankAi.

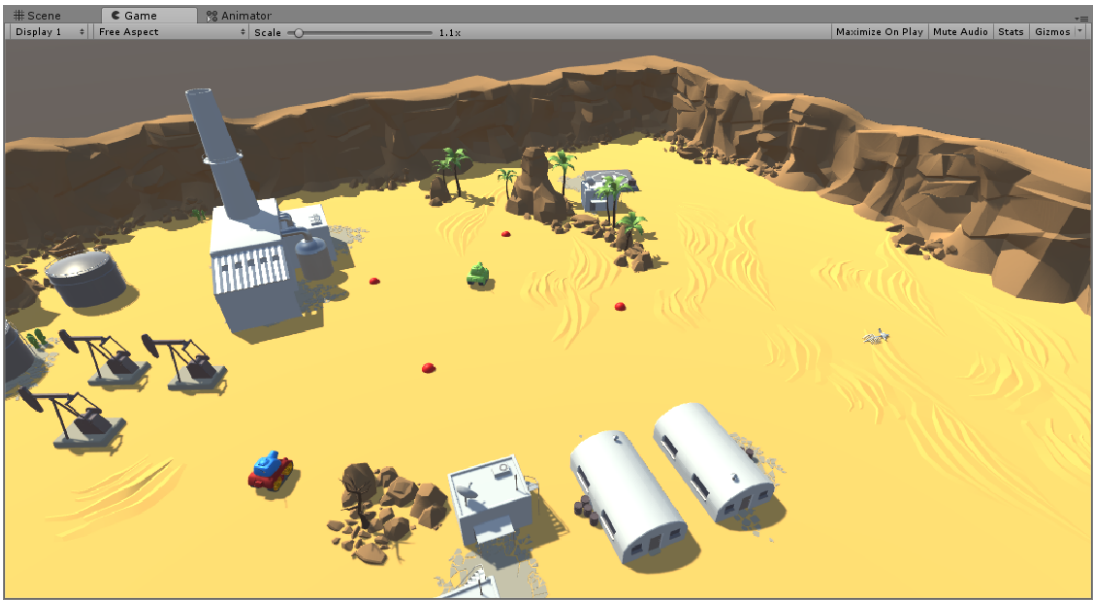
Isječak kôda 28: Skripta TankAi

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class TankAi : MonoBehaviour {
6
7          public GameObject player;
8          public GameObject bullet;
9          public GameObject turret;
10         public GameObject getplayer()
11     {
12                 return player;
13     }
14
15         Animator anim;
16
17         void Fire()
18     {
19                 GameObject b = Instantiate(bullet, turret.transform.position, turret
                    .transform.rotation);
20                 b.GetComponent<Rigidbody>().AddForce(turret.transform.forward * 500)
                    ;
21     }
22
23         public void StopFiring()
24         {
25                 CancelInvoke("Fire");
26         }
27         public void StartFiring()
28         {
29                 InvokeRepeating("Fire", 0.5f, 0.5f);
30         }
31
32
33         // Use this for initialization
34         void Start () {
35                 anim = GetComponent<Animator>();
36         }
37
```

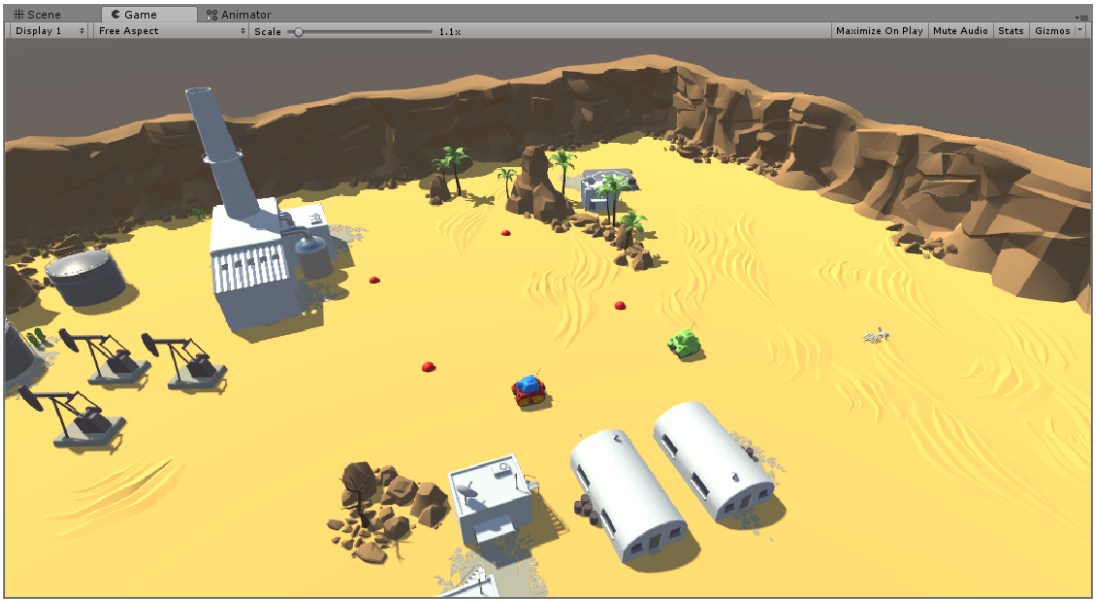
```
38          // Update is called once per frame
39          void Update () {
40                  anim.SetFloat("distance", Vector3.Distance(transform.position,
                     player.transform.position));
41          }
42  }
```

Unutar skripte TankAi dodane su nove javne deklaracije tipa `GameObject bullet` i `turret` kojima će se pridružiti njihovi objekti unutar Unity-a. U funkciji `Fire` inicijalizira se varijabla `b` tipa `GameObject` kojoj se instancira objekt `bullet`, sa pozicijom i rotacijom objekta `turret`, te u sljedećoj liniji koda tom objektu pomoću njegove komponente `rigidbody` (fizika objekta) dodaje silu koja ga propelira naprijed. U funkciji `StartFiring` pomoću funkcije `InvokeRepeating` poziva se funkcija `Fire` na način da se nakon pola sekunde krene izvršavati funkcija `Fire`, te se ona opet poziva nakon pola sekunde i tako u nedogled. Kako bi tenk prestao pucati koristi se funkcija `StopFiring` koja u sebi sadrži funkciju `CancelInvoke` koja zaustavlja pozivanje funkcije `Fire`.

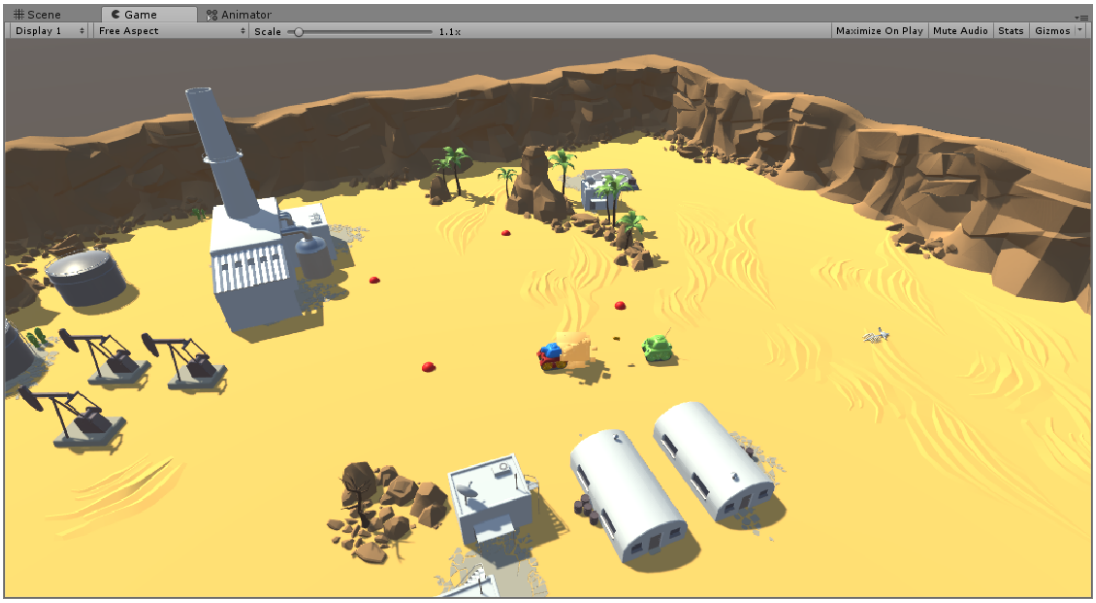
4.3. Rezultat



Slika 23: Tenk u stanju Patrol



Slika 24: Tenk u stanju Chase



Slika 25: Tenk u stanju Attack

5. Zaključak

U početku ovog rada spominjala se umjetna inteligencija koju je većina ljudi susretala u videoigrama i kao takvu nije ni bilo prepoznato da je ona zadužena za kvalitetu videoigre koju neprekidno igraju. Rečeno je kako kroz ovaj rad želim steći bolje razumijevanje umjetne inteligencije kako bih u budućim interakcijama lakše ju prepoznao i dobio dojam koliki je trud potreban za kreaciju iste. Kroz detaljno proučavanje metoda koje su prisutne i koje se koriste u izradi videoigara mogu reći da sada posjedujem dovoljno opće znanje da bih razumio kolika kompleksnost stoji iza jedne umjetne inteligencije koju ljudi uzimaju zdravo za gotovo.

Razvojem svoje videoigre i implementacijom konačnog automata, koji je jedan od početnih modela umjetne inteligencije korišten u videoigrama, shvatio sam koliko je čak i jednostavna izvedba umjetne inteligencije potrebna za izgradnju videoigre koja će obuzeti igrača, izolirati ga od realnog svijeta i na kraju biti uspješna.

Korištenjem programskog jezika C++ pokazan je način izrade konačnog automata kroz programski kod i potrebni koraci povezivanja raznih struktura kako bi na kraju zapravo simulirali njegov rad. U krajnjem rezultatu je vidljivo kako bi taj kod funkcionirao kada bi se ubacio grafički dio simulacije. Kombinacijom C# programskog jezika i alata Unity kreirana je videoigra u kojoj je implementiran konačni automat koji upravlja neigrajućim tenkom. Prednost ove implementacije je vizualni prikaz rada konačnog automata u realnom vremenu i način kreacije stanja i veza naspram ručnog načina objašnjenog pomoću primjera u programskom jeziku C++. Unutar Unity-a kreirana su stanja i uvjetovane veze konačnog automata, te je bilo potrebno samo implementirati ponašanje unutar pojedinog stanja pomoću programskog jezika C#. Može se zaključiti da je posao programera olakšan u odnosu na prijašnji način rada i da, iako već stara metoda i dalje drži svoje mjesto kao početni oblik umjetne inteligencije koja se kasnije nadograđuje ako to videoigra zahtijeva.

Nadam se da ću se nastaviti baviti područjem umjetne inteligencije i doći u kontakt s višim odnosno jačim izvedbama i proučavati njihov rad i sposobnosti samoučenja jer je to ipak najbitnija funkcionalnost koju današnji programeri pokušavaju dostići. Pomoću samoučenja mogućnosti umjetne inteligencije nemaju granica, jer kao i ljudi uče kroz svoje postojanje i ne prestaju se razvijati, a računala u tom spektru sposobnosti su puno brža. Za kraj nam ne ostaje ništa drugo nego čekati daljnji razvoj i razne primjene umjetne inteligencije u svijetu bio on stvaran ili virtualan.

Popis literature

[1] D. M. Bourg i G. Seemann, *AI for Game Developers: Creating Intelligent Behavior in Games*, English, 1st edition. Sebastopol, CA: O'Reilly Media, 8. 2004., ISBN: 978-0-596-00555-9.

[2] W. contributors, *Temple Run — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Temple_Run&oldid=1038126779, [Online; accessed 17-September-2021], 2021.

[3] ——, *Angry Birds (video game) — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Angry_Birds_(video_game)&oldid=1041809958, [Online; accessed 17-September-2021], 2021.

[4] ——, *Super Mario Run — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Super_Mario_Run&oldid=1041557667, [Online; accessed 17-September-2021], 2021.

[5] ——, *Subnautica — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Subnautica&oldid=1043191393, [Online; accessed 17-September-2021], 2021.

[6] A. Sinicki. (3. 2021.). „What is Unity? Everything you need to know." en-US, adresa: https://www.androidauthority.com/what-is-unity-1131558/ (pogledano 28. 8. 2021.).

[7] J. Nunns. (4. 2017.). „What is Visual Studio?" en-US, adresa: https://techmonitor.ai/what-is/what-is-visual-studio-4959054 (pogledano 28. 8. 2021.).

[8] Microsoft. (). „Unity Games Development Tools | Visual Studio," adresa: https://visualstudio.microsoft.com/vs/unity-tools/ (pogledano 28. 8. 2021.).

[9] I. C. Education. (8. 2021.). „What is Artificial Intelligence (AI)?" en-us, adresa: https://www.ibm.com/cloud/learn/what-is-artificial-intelligence (pogledano 28. 8. 2021.).

[10] W. contributors, *Pac-Man — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Pac-Man&oldid=1041557570, [Online; accessed 17-September-2021], 2021.

[11] ——, *Call of Duty — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Call_of_Duty&oldid=1041977571, [Online; accessed 17-September-2021], 2021.

[12] ——, *Creatures (video game series) — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Creatures_(video_game_series)&oldid=1036987656, [Online; accessed 17-September-2021], 2021.

[13] ——, *Black & White (video game) — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Black_26_White_(video_game)&oldid=1037106956, [Online; accessed 17-September-2021], 2021.

[14] ——, *Battlecruiser 3000AD — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Battlecruiser_3000AD&oldid=1036986080, [Online; accessed 17-September-2021], 2021.

[15] ——, *Dirt Track Racing (video game) — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Dirt_Track_Racing_(video_game)&oldid=963443401, [Online; accessed 17-September-2021], 2020.

[16] ——, *Heavy Gear (video game) — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Heavy_Gear_(video_game)&oldid=1016864460, [Online; accessed 17-September-2021], 2021.

[17] M. Schatten i B. Okreša Đurić, *Višeagentni sustavi: Formalizacija agenata, nastavni materijali*, 2020.

[18] G. N. Yannakakis i J. Togelius, *Artificial Intelligence and Games*, English, 1st ed. 2018 edition. Cham: Springer International Publishing, 2. 2018., ISBN: 978-3-319-63519-4.

Popis slika

Popis tablica

Popis isječaka koda

Prilozi

1. Prilog 1

2. Prilog 2