Modeliranje i implementacija neprijateljskih likova kao konačnih automata na primjeru igre žanra 3D shooter

Blažević, Zdravko

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Zdravko Blažević

MODELIRANJE I IMPLEMENTACIJA NEPRIJATELJSKIH LIKOVA KAO KONAČNIH AUTOMATA NA PRIMJERU IGRE ŽANRA 3D SHOOTER

ZAVRŠNI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Zdravko Blažević

Matični broj: 0016156019

Studij: Informacijske tehnologije i digitalizacija poslovanja

MODELIRANJE I IMPLEMENTACIJA NEPRIJATELJSKIH LIKOVA KAO KONAČNIH AUTOMATA NA PRIMJERU IGRE ŽANRA 3D SHOOTER

ZAVRŠNI RAD

Mentor :

Doc. dr. sc. Bogdan Okreša Đurić

Varaždin, rujan 2024.

Zdravko Blažević

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

Ovaj završni rad bavi se modeliranjem i implementacijom neprijateljskih likova kao konačnih automata na primjeru igre žanra 3D shooter. Glavni cilj rada bio je prikazati kako se neprijateljski likovi u video igrama mogu modelirati kao konačni automati (eng. Finite State Machines), s ciljem prikazivanja jednostavnost i učinkovitost njihove primjene u dizajnu ponašanja neprijateljskih likova. Rad istražuje osnovne teorijske postavke umjetne inteligencije u igrama i konačnih automata, uz detaljnu analizu njihovih primjena kroz povijest razvoja video igara. Konačni automat se koristi za definiranje različitih stanja neprijateljskih likova i prijelaza između tih stanja, ovisno o situaciji u igri. Na ovaj način omogućuje stvaranje neprijateljskih likova s realističnim ponašanjem, koje oponaša ljudsku interakciju i donosi odluke na temelju ulaznih podataka u stvarnom vremenu. U praktičnom dijelu rada razvijena je videoigra u kojoj su kreirani neprijateljski likovi čije je ponašanje definirano korištenjem konačnih automata. Njihove reakcije na određene situacije unutar igre ovise o njihovim stanjima i prijelazima, čime se postiže izazovnost igre. Također, igraču je omogućeno korištenje različitih oružja te eliminacija neprijateljskih likova, što dodatno doprinosi dinamici i zabavnosti same igre.

Ključne riječi: konačni automat, neprijateljski likovi, 3D shooter, video igre, umjetna inteligencija.

Sadržaj

1. Uvod

Umjetna inteligencija (UI, prema eng. akronimu AI, Artificial intelligence), dio računalstva koji se bavi razvojem sposobnosti računala da obavljaju zadaće za koje je potreban neki oblik inteligencije; također oznaka svojstva neživog sustava koji pokazuje inteligenciju (inteligentni sustav). [1]. Sama umjetna inteligencija donosi sposobnost učenja, razumijevanja i mogućnost nošenja sa nepoznatim situacijama pred koje je postavljena, te zapravo posjeduje sposobnost da reproducira razmišljanje čovjeka, ljudske akcije i racionalno razmišljanje. Sposobnost umjetne inteligencije da analizira ogromne količine podataka i na temelju njih donosi odluke s visokim stupnjem točnosti, donijeli su revoluciju u raznim granam industrije. Ova tehnologija je drastično promijenila način na koji živimo i radimo. Primjerice umjetna inteligencija obuhvaća sve od raznih preporuka u online trgovinama, do postavljanja medicinskih dijagnoza, pa čak i upravljanje vozilima. Umjetna inteligencija je donijela bitne promjene u svijet video igara, gdje je korištena za stvaranje prilagodljivih neprijateljskih likova, realistične interakcije i kreiranje realističnih svjetova, pružajući igračima nezaboravno iskustvo igranja videoigara.

Konačni automat (eng. FSM - Finite State Machine) je model koji se koristi za prikazivanje ponašanja određenog sustava kroz niz različitih stanja (eng. state) i prijelaza među njima. Svaki automat može biti u jednom od preddefiniranih stanja u bilo kojem trenutku, dok se prijelaz iz jednog stanja u drugo događa kada je ispunjen određeni uvjet. Konačni automati se primjenjuju u raznim granama informatičkih znanosti, zbog svoje jednostavnosti i učinkovitosti. Jedan od primjera korištenja konačnih automata, van svijeta videoigara zasigurno je i kontrola sustava, gdje se konačni automat koristi za kontrolu jednostavnih sustava poput dizala ili perilice rublja gdje se operacije definiraju raznim stanjima i prijelazima između njih. [2, str. 33].

Kako bi koncept konačnog automata bio pobliže objašnjen može se prikazati na primjeru dizala. Dizalo ima dva stanja "Prizemlje" i "Prvi kat". Dizalo u određenom trenutku može biti u bilo kojem od ta dva stanja. Ako je dizalo u prizemlju, a pritisne se gumb za podizanje, konačni automat prelazi u stanje "Prvi kat" te podiže osobu na prvi kat. Ako je dizalo na prvom katu i pritisnut je gumb za spuštanje prizemlje, konačni automat se vraća na stanje prizemlje. Svako stanje ima definirane akcije, poput paljenja odgovarajućeg svjetla (crveno za prizemlje, zeleno za prvi kat). Konačni automat osigurava da dizalo uvijek pravilno reagira na ulazne signale (pritisak na gumb), te se kreće između stanja u skladu s tim. [3].

Umjetna inteligencija i koncept konačnog automata važni su elementi u razvoju računalnih igara, iako nisu jedini pristup koji se koristi. Umjetna inteligencija omogućava stvaranje složenih neprijateljskih likova i kompleksnih simulacija, gdje se konačni automati često koriste za upravljanje ponašanjem neigrajućih likova, ali i svijeta unutar igara, omogućujući definiranje stanja i prijelaza između stanja. Iako korištenje konačnog automata nije nužno za stvaranje dobre igre, konačni automat predstavlja učinkovit način za modeliranje ponašanja u različitim situacijama.

2. Metode i tehnike rada

Za istraživanje teme koncepta konačnog automata korištena je razna literatura, uključujući knjige, online znanstvene članke, te druge web stranice koje se bave temom ovog završnog rada, te detaljno objašnjavaju koncept konačnog automata. Za proučavanje igara koje koriste slične mehanike, korištena je platforma YouTube, a sama videoigra razvijena je u alatu Godot. Godot je alat otvorenog koda (eng. open-source) pomoću kojeg je moguće jednostavno kreirati 2D ili 3D igara. Godot koristi vlastiti skriptni jezik, koji se naziva GDScript, koji je dosta sličan Pythonu.

Za uređivanje samih animacija neprijateljskih likova korišten je alat Blender, koji je također alat otvorenog koda te se koristi za 3D animacije, modeliranje i renderiranje. Za izradu videoigre bilo je potrebno pronaći odgovarajuće resurse (eng. assets), pri čemu su korištene platforme Itch.io, Mixamo i Kenney, dok su zvučni efekti preuzeti s platforme Pixabay

3. Povijest i razvoj umjetne inteligencije u igrama

Video igre i umjetna inteligencija imaju dugu zajedničku povijest. Rani pokušaji razvoja umjetne inteligencije sežu na početak 20. stoljeća, kada su programeri radili na ugradnji jednostavnih algoritama kako bi stvorili suparnika (neprijatelje) unutar igara. Prije nego što je umjetna inteligencija službeno prepoznata u ovom polju, rani pioniri računalne znanosti pisali su programe za igranje računalnih igara kako bi testirali mogu li računala riješiti zadatke koji zahtijevaju određenu razinu "inteligencije". [2, str. 8.-10.].

Jedna od prvih igara koja je koristila jednostavan oblike umjetne inteligencije bila je igra zvana "OXO" koja je razvijena 1952. godine, te je predstavljala digitaliziranu verziju igre križić-kružić (eng. Tic-Tac-Toe) koju je osmislio Alexander S. Douglas na Sveučilištu Cambridge. [2, str. 8.-10.]. U samoj igri umjetna inteligencija je predstavljala suparničkog igrača, u kojoj igra na osnovu trenutnog stanja na igraćoj ploči donosi odluke o sljedećem potezu na osnovu prethodno definiranih pravila i logike igre. [4].

Jednako tako važnu ulogu u razvoju umjetne inteligencije u računalnim igrama imao je Arthur Samuel koji je zaslužan za razvoj jedne od prvih primjena strojnog učenja, poznatog kao poticano učenje (eng. reinforcement learning). Njegov rad na igri dame (eng. Checkers) tijekom 1950-ih i 1960-ih godina postavio je temelje za moderni razvoj algoritama poticanog učenja. Samuelov program koristio je osnovne principe poticanog učenja. Program je igrao tisuće partija protiv samog sebe, i na taj način istraživao različite strategije i poteze. Program je koristio heurističku analitičku funkciju za procjenu snage trenutnog stanja igre na ploči. Na temelju rezultata svake partije (pobjeda, poraz ili neriješeno), program bi nadopunio svoje procjene o tome koji su potezi i strategije bili najuspješniji, koji nešto manje uspješni. Kroz neprestano igranje i procjenu dobivenih rezultata, program je polagano poboljšavao svoje strategije, učeći koje poteze treba favorizirati u određenim situacijama. [5, str. 210.-229.].

Nakon napora na igri dame od strane Arthura Samuela, istraživanje umjetne inteligencije u igrama na ploči nastavilo se proširivati, ulazeći u sve složenije igre i sofisticiranije algoritme. Šah je postao jedno od glavnih područja istraživanja zbog same složenosti igre. Zbog toga su mnogi programeri usmjerili svoj rad na stvaranje programa koji bi mogli igrati šah. Puno ranih istraživanja fokusirala su se na algoritme za pretraživanje stabala, poput minimax i alfa-beta rezanja, koji su omogućavali računalima da analiziraju moguće poteze i njihov učinak na rezultat igre.[2, str. 8.-10.].

Važan trenutak u razvoju umjetne inteligencije zasigurno je bilo IBM-ovo računalo Deep Blue. Deep Blue je bio poznat po svojim sposobnostima računanja i preko milijuna poteza u sekundi, a to je postizao kombiniranjem snažnog hardvera za svoje vrijeme i algoritama. Deep Blue je koristio minimax algoritam s izmjenama napravljenim specifično za šah, u što su bile uključene vrlo precizne funkcije za analizu ploče koje su mogle procijeniti vrijednost pozicija na temelju faktora poput materijalne prednosti, kontrole ploče i sigurnosti kralja. [6].

1997. godine Deep Blue je postao prvi računalni sustav koji je pobijedio slavnog svjetskog prvaka u šahu, Garryja Kasparova u meču koji je bio podosta medijski popraćen. Ova pobjeda značila je veliki trenutak u povijesti umjetne inteligencije, u kojem je pokazano da su

računalni programi postigli razinu vještine koja je mogla nadmašiti najbolje ljudske igrače. Garry Kasparov je bio poznat po svom briljantnom taktičkom razmišljanju i sveobuhvatnom razumijevanju igre, bio je nadmašen zahvaljujući velikoj računalnoj snazi i sofisticiranim algoritmima koje je koristio Deep Blue. [7].

Nekoliko godina prije uspjeha Deep Bluea također je bio razvijen sustav TD-Gammon, od strane Geralda Tesauria 1992. godine u IBM-u. Ovaj program je postao krucijalan zbog upotrebe naprednih metoda umjetne inteligencije, a naručito neuronske mreže u učenju vremenskih razlika (eng. temporal difference learning) što je omogućilo programu da postigne razinu vještine usporedivu s najboljim ljudskim igračima backgammona. Glavna inovacija TD-Gammona bila je upotreba učenja vremenskih razlika (eng. temporal difference learning, TD-learning). Ova metoda omogućava programu da ažurira svoje procjene na temelju razlike između predviđenih i stvarnih ishoda. TD-Gammon koristi algoritam, koji kombinira trenutne procjene sa stvarnim rezultatima kako bi kontinuirano poboljšavao svoje strategije. Kako bi naučio optimalne strategije, TD-Gammon je odigrao milijune partija protiv sebe. Kroz ovaj proces, program je bio u stanju istražiti različite strategije i poteze, te prilagoditi svoje procjene kako bi poboljšao ukupnu igru. [2, str. 8.-10.].

IBM-ova sljedeća uspješnica bio je Watson koji je bio napredni računalni sustav umjetne inteligencije, te je bio prepoznatljiv po svojoj sposobnosti obrade prirodnog jezika i analize velikih količina podataka, Watson je nazvan po osnivaču IBM-a, Thomasu J. Watsonu. Razvijen je kao dio IBM-ove inicijative za stvaranje računalnih sustava koji mogu razumjeti, učiti i odgovarati na pitanja postavljena na prirodnom jeziku. Godine 2011. Watson se natjecao u TV igri Jeopardy! te je osvojio milijun dolara protiv bivših pobjednika igre. [2, str. 8.-10.].

Razvoj igara pogonjenih umjetnom inteligencijom nije stao samo na razvoju igara na ploči, koje su imale diskretne mehanike za poteze i gdje je kompletno stanje igre vidljivo svim igračima, nego se također proširio na svijet videoigara. U posljednjih nekoliko desetljeća uloženi su veliki napori u razvoj umjetne inteligencije koja može igrati igre što učinkovitije oponašajući ljudsko ponašanje. Jedna od velikih prekretnica u svijetu videoigara bio je Googleov algoritam DeepMind, koji je naučen igrati razne Atari igre, od kojih su mnoge bile vrlo izazovne. Jedna od najzahtjevnijih igara za savladavanje bila je Ms. Pac-Man, koju je kompanija Namco razvila 1982. godine, a poznata je po složenim neprijateljskim ponašanjima i nepredvidivim situacijama. Microsoftov tim Maluuba je u lipnju 2017. uspio pobijediti ovu igru koristeći hibridnu arhitekturu za upravljanje agentima unutar igre, čime je postignuto super-ljudsko igranje.[2, str. 8.-10.].

U ranim 1970-ima, kada su se prvi put pojavile igre sa statičkom grafikom, ponašanje i mehanike neigrajućih likova (engl. non-player characters) bili su strogo skriptirani ili su se oslanjali na jednostavna pravila. Umjetna inteligencija kakvu danas poznajemo nije postojala, što je dijelom bilo zbog ograničene računalne snage i nedovoljno razvijenog hardvera, a dijelom zbog nedostatka znanja na području umjetne inteligencije. Kako se umjetna inteligencija razvijala u akademskim zajednicama, tako je rasla i njezina složenost u videoigrama. Prekretnica u svijetu videoigara dogodila se s igrom Creatures, koju je 1996. godine razvila tvrtka Millennium Interactive. Ova igra se istaknula zbog korištenja naprednih tehnika umjetne inteligencije, osobito

upotrebom neuronskih mreža za oblikovanje ponašanja virtualnih bića zvanih Nornsi. Ova bića su, zahvaljujući neuronskim mrežama, kroz vrijeme razvijala svoja ponašanja, oponašajući procese učenja slične onima kod živih bića. Neuronske mreže omogućile su Nornsima (virtualna bića u igri) da uče iz svog okruženja putem pozitivnih i negativnih povratnih informacija. [8].

Još jedan od bitnih predstavnika korištenja umjetne inteligencije u igrama bio je Doom, predstavljen 1993. godine, u kojem su neprijateljski likovi koristili konačni automat (FSM - Finite State Machines). Konačni automat je funkcionirao tako da je neprijatelj u svakom trenutku bio u određenom stanju (eng. state), a svako stanje definiralo je njegovo ponašanje, poput napada, kretanja ili povlačenja. Promjena stanja odvijala se na temelju uvjeta kao što su blizina igrača ili primljena šteta, omogućujući neprijateljima da reagiraju na različite situacije u igri. [9].

Sljedeća velika prekretnica u razvoju umjetne inteligencije u video igrama je bila Halo koja je znanstveno-fantastična pucačka igra u prvom licu koju je razvio Bungie. Bila je jak poznata po svojima naprednom značajkama umjetne inteligencije koje su mogle upravljati ponašanjem neprijatelja i saveznika. Neprijateljski igrači u igri imaju visoki stupanj koordinacije, te koriste timske taktike kako bi nadjačali samog igrača. neprijatelji su imali mogućnost međusobnog komuniciranja, te su se mogli postaviti u razne formacije i strategije ovisno o stanju na borbenom polju [10]. 11. studenog izašao je novi nastavak igre zvan Halo 2 koji je bio posebno specifičan po korištenju stabla ponašanja (eng. behavior trees). Igra je zapravo i popularizirala korištenje stabala ponašanja u igrama, a ona su omogućavala kreatorima igara kreiranje složenih hijerarhijski ponašanja za negirajuće likove. Na ovaj način su neprijatelji u igru mogli donosi odluke na temelju više faktora, na temelju čega su neprijateljski likovi prirodnije i prilagodljivije reagirati na akcije igrača. [2, str. 8.-10.].

Kako su tekle godine tako su se razvijale i video igre, ali i sam hardver koji je iz godine u godinu postajao sve snažniji i brži, te na taj način poboljšao računalnu snagu, umjetna inteligencija nije samo poboljšala ponašanje neprijatelja, nego je omogućila i dinamičko generiranje sadržaja, gdje je algoritam mogao stvarati razne zadatke i svjetove unutar samih igara. Jedan od glavnih predstavnika te dimenzije umjetne inteligencije u video igrama bila je No Man's Sky, koja koristi proceduralno generiranje svjetova, koja omogućila da igrači svakim igranjem igre osjete nešto novo, bez potrebe za ručnim dizajniranjem svjetova u igri.[2]

Povijest razvoja umjetne inteligencije ima vrlo zanimljiv razvojni put, od jednostavnih algoritama do kompleksnih sustava koji danas čine temelj modernih videoigara. Umjetna inteligencija ne samo da je transformirala način na koji doživljavamo videoigre, već je otvorila vrata budućim inovacijama na području tehnologije i interaktivne zabave.

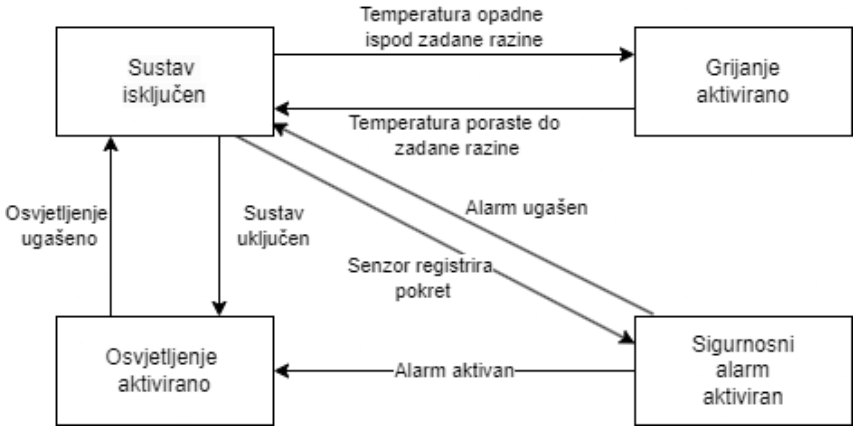
4. Koncept konačnog automata

Koncept konačnog automata (eng. FSM-Finite State Machine) predstavlja temeljni matematički model koji se koristi za opisivanje sustava kroz niz stanja. Konačni automati se naširoko koriste u raznim područjima, uključujući umjetnu inteligenciju, računalne znanosti i teoriju jezika, ali su posebno važni u razvoju videoigara za upravljanje ponašanjem likova, neprijatelja i simulacijama. [2, str. 33.]

Konačni automat se može predstaviti kao grafički model sastavljen od stanja (eng. states) i prijelaza (eng. transitions) između tih stanja. Svako stanje pohranjuje informacije o određenom zadatku ili ponašanju, dok prijelazi definiraju uvjete pod kojima se prelazi iz jednog stanja u drugo [2, str. 33.]. Primjerice, u videoigri koja simulira preživljavanje u divljini, lik može biti u **"stanju traženja hrane"** dok se kreće kroz šumu. Ako lik naiđe na divlju životinju, automat ga prebacuje u **"obrambeno stanje."**. U svakom trenutku, lik može biti samo u jednom stanju ili traži hranu ili se brani. Promjena stanja događa se ako se susretne s određenom situacijom, poput pojave životinje ili opasnosti u okolini, što služi kao signal za prijelaz u novo stanje.

Kako bi koncept konačnog automata bio što jasnije objašnjen, prikazat će se kroz primjer pametnog doma, koji je vidljiv na dijagramu i tablici stanja, koji se nalazi na slici 1.i tablici 1. Ovaj konačni automat prikazuje osnovni sustav upravljanja pametnom kućom s četiri glavna stanja: **"Sustav isključen"**, **"Grijanje aktivirano"**, **"Osvjetljenje aktivirano"** i **"Sigurnosni alarm aktiviran"**. Automat kontrolira prelazak iz jednog stanja u drugo na temelju različitih ulaznih signala. U stanju **"Sustav isključen"** sustav je neaktivan, a promjena stanja može se dogoditi ako temperatura opadne ispod zadane razine, što prelazi sustav u stanje **"Grijanje aktivirano"**. Ako senzor registrira pokret, aktivirat će se stanje **"Sigurnosni alarm aktiviran"**, te će se oglasiti sigurnosni alarm i upaliti svjetla. Također, sustav prelazi u stanje **"Osvjetljenje aktivirano"** ako se ručno uključi. Kada je grijanje aktivirano, ono će ostati aktivno dok temperatura ne poraste do zadane razine, nakon čega se vraća u stanje **"Sustav isključen"**. Slično tome, osvjetljenje ostaje aktivirano dok se sustav ne isključi, nakon čega se vraća u prvobitno stanje. Sigurnosni alarm se aktivira kada senzor detektira pokret i ostaje aktivan sve dok se ne isključi alarm, nakon čega se sustav ponovno vraća u stanje isključenosti. Ovaj primjer automata prikazuje kako pametni sustav reagira na različite ulazne signale, kao što su promjene temperature, pokreti ili ručna aktivacija sustava, i na temelju njih prelazi u odgovarajuća stanja.

Konačni automati su jednostavni za dizajn i implementaciju te omogućuju jasan prikaz sustava, što ih čini popularnim u svijetu igara. Međutim, njihova struktura postaje složena kada broj stanja raste, što može otežati održavanje i proširenje sustava. Iako konačni automati mogu upravljati složenim ponašanjem unutar igre, ponekad su ograničeni u prilagodljivosti i evoluciji nakon što su dizajnirani. Za ublažavanje ovog problema, prijelazi između stanja mogu biti definirani pomoću neizrazite logike (eng. fuzzy logic), što povećava fleksibilnost sustava. [2, str. 33.]



Slika 1: Dijagram stanaja za prikaz scenarija u pametnom domu

Tablica 1: Prikaz tablice stanja na primjeru sustava pametnog doma

Trenutno stanje	**Ulazni uvjet**	**Sljedeće stanje**
Sustav isključen	Temperatura opadne	Grijanje aktivirano
Sustav isključen	Senzor registrira pokret	Sigurnosni alarm aktiviran
Grijanje aktivirano	Temperatura poraste	Sustav isključen
Sigurnosni alarm aktiviran	Alarm isključen	Sustav isključen
Sigurnosni alarm aktiviran	Alarm aktiviran	Osvjetljenje aktivirano
Osvjetljenje aktivirano	Sustav uključen	Sustav isključen
Sustav isključen	Svjetlo ugašeno	Osvjetljenje aktivirano

4.1. Klasifikacija konačnih automata

Postoje različite vrste konačnih automata, od kojih se najčešće spominju prihvatitelji (priznavatelji) i transduktori, koji imaju različite uloge i primjene u modeliranju sustava. Prihvatitelji automati prepoznaju određene nizove ulaznih simbola i na temelju njih odlučuju je li niz prihvaćen ili odbijen. S druge strane, transduktori ne samo da prepoznaju nizove, već i generiraju izlazne simbole na temelju ulaznih podataka. Unutar kategorije transduktora razlikuju se dvije osnovne vrste: Mooreovi i Mealyjevi automati. Osim tih osnovnih skupina, postoje i mješoviti automati, koji kombiniraju najbolje značajke Mooreovih i Mealyjevih modela, čime omogućuju smanjenje broja stanja i poboljšavaju učinkovitost. U nastavku će biti prikazane razlike između modela i njihove primjene.

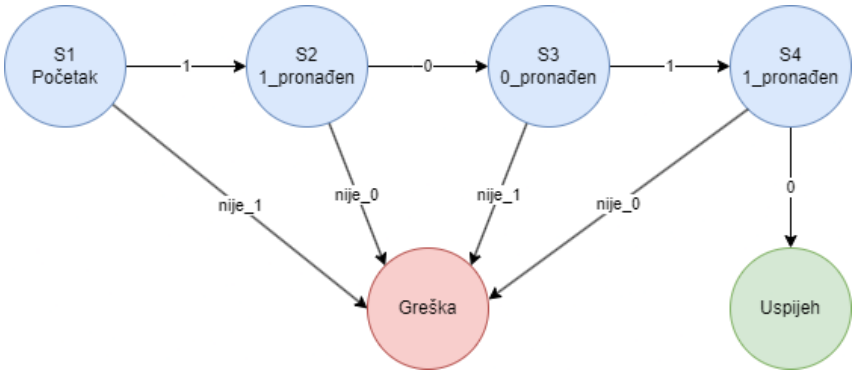
4.1.1. Prihvatiteljski automati (priznavatelji)

Prihvatiteljski konačni automat (eng. finite state acceptor) je vrsta konačnog automata koja nema izlaznih vrijednosti [11]. Ovaj automat služi za prepoznavanje ili odbijanje nizova ulaznih simbola na temelju zadatih pravila ili obrazaca. Korisniku automata je važan samo konačni rezultat, ako se automat završi u "prihvatljivom" stanju nakon obrade niza simbola,

smatra se da je ulazni niz prihvaćen; ako se završi u "neprihvatljivom" stanju, niz je odbijen. [11]

Jedan od najčešćih primjera korištenja prihvatiteljskih automata je provjera da li niz znakova zadovoljava određeni regularni izraz. Na primjer, ako imamo regularni izraz koji opisuje niz brojeva s jednim velikim slovom u sredini, konačni automat će pratiti prijelaze između stanja kako bi provjerio da li se ulazni niz podudara s tim obrascem. Automatom se upravlja tako da se postavlja u početno stanje, zatim prolazi kroz niz stanja prema ulaznim simbolima, i na kraju se provjerava je li završio u prihvatljivom stanju. Ako je, ulazni niz se prihvaća, inače se odbija.

Prihvatiteljski automati često se koriste u programima za provjeru regularnih izraza, gdje ulazni niz mora zadovoljiti određena pravila za prepoznavanje. Ovaj tip automata jednostavan je za implementaciju i koristan je u mnogim aplikacijama koje zahtijevaju prepoznavanje obrazaca.



Slika 2: Konačni automat prihvatitelj: parsiranje niza "1010"

Slika 2. prikazuje primjer korištenja prihvatiteljskog konačnog automata koji se koristi za prepoznavanje niza "1010". Automat započinje u početnom stanju označenom kao "Početak" i čeka da primi prvi znak. Ako primi znak "1", prelazi u stanje `1_pronađen`. Ako prvi znak nije "1", automat prelazi u stanje "Greška", što znači da niz nije prepoznat.

Nakon prepoznavanja "1", automat prelazi u stanje `1_pronađen` i očekuje znak "0". Ako primi "0", prelazi u sljedeće stanje `0_pronađen`. Ako bilo koji drugi znak bude primljen umjesto "0", automat se prebacuje u stanje "Greška". Kada je "0" prepoznato, automat prelazi u stanje `0_pronađen` i očekuje sljedeći znak "1". Ako je "1" prepoznato, prelazi u stanje `1_pronađen`, dok svaki drugi znak vodi do stanja "Greška". Ako je "1" prepoznato, automat očekuje znak "0". Ako je "0" prepoznato, automat prelazi u stanje "Uspjeh", što znači da je cijeli niz "1010" ispravno prepoznat. Ako bilo koji drugi znak bude primljen, automat prelazi u stanje "Greška", što znači da niz nije ispravan.

4.1.2. Transduktori

Transduktori su vrsta konačnih automata koji, za razliku od prihvatiteljskih automata, imaju izlaznu funkciju koja generira izlazne simbole dok se obrađuju ulazni podaci. U kontekstu konačnih automata, transduktori su strojevi koji uzimaju nizove ulaznih simbola i pretvaraju ih u izlazne nizove, koristeći unutarnju logiku stroja koja se temelji na trenutačnom stanju i ulaznom simbolu. Funkcija izlaza može ovisiti o stanju stroja i ulaznom simbolu, a ova struktura omo-

gućava transduktorima da ne samo prepoznaju određene ulazne sekvence, već i da proizvode odgovarajuće izlazne sekvence na temelju tih ulaza. [12, str. 480.-501.]

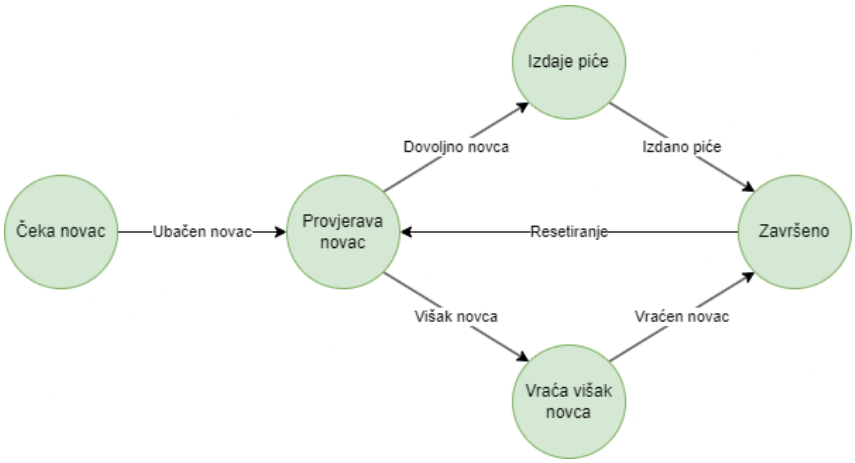
Jedan od ključnih elemenata transduktora je njihova sposobnost da, za razliku od prihvatiteljskih automata, rade s ulaznima i izlaznima, čime se omogućava neposredna reakcija na ulaz. U transduktorskim strojevima, izlaz nije vezan samo uz prihvaćanje ili odbijanje niza, već je povezan s prijelazima između stanja [12, str. 480.-501.]. Ovo čini transduktore posebno korisnim u aplikacijama poput signalne obrade ili procesiranja podataka u realnom vremenu, gdje je potrebno ne samo prepoznati ulaz, već i odmah generirati odgovarajući odgovor.

Dva najpoznatija tipa transduktora su Mealyjev i Mooreov automat. Mealyjev automat generira izlaz na prijelazu između stanja, dok Mooreov automat generira izlaz kada se stigne u određeno stanje.

4.1.2.1. Mooreov automat

Mooreov model transduktora predstavlja vrstu konačnog automata u kojem izlaz ovisi isključivo o trenutnom stanju, a ne o ulaznim signalima. Svaki put kada automat uđe u novo stanje, automatski izvršava radnje povezane s tim stanjem. Ova jednostavna struktura temelji se na "akcijama pri ulasku u stanje", što znači da se svaka promjena stanja automatski pokreće radnje specifične za to stanje [13].

Ovaj model je lakši za razumijevanje jer radnje ovise samo o stanju, a ne o prijelazima između stanja. Prednost Mooreovog automata je u tome što je jednostavniji za implementaciju i održavati jer je ponašanje automata jasno definirano kroz stanje u kojem se nalazi [13].



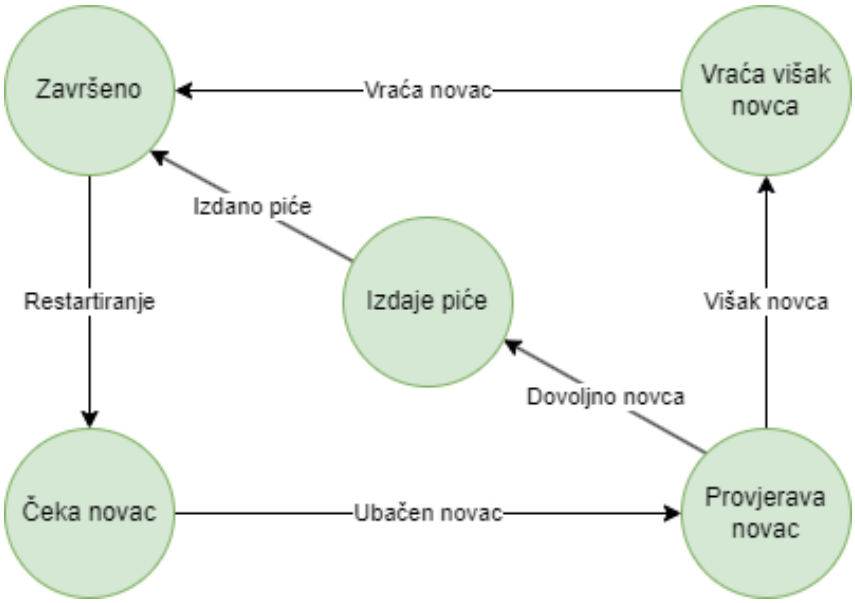
Slika 3: Primjer Moorov automat

Primjer Mooreovog automata za pića prikazan na slici 3. prikazuje proces koji aparat prolazi prilikom izdavanja pića. Automat započinje u stanju „Čeka novac“, gdje se čeka da korisnik ubaci novac. Kada se novac ubaci, aparat prelazi u stanje „Provjerava novac“, gdje se utvrđuje je li iznos dovoljan za izdavanje pića. Ako je ubačeno dovoljno novca, automat prelazi u stanje „Izdaje piće“, gdje se piće isporučuje korisniku. Nakon izdavanja pića, automat prelazi u završno stanje „Završeno“, gdje se proces završava i aparat se resetira. Ako je ubačeno previše novca, aparat umjesto toga prelazi u stanje „Vraća višak novca“, gdje se vraća višak novca

korisniku, nakon čega također prelazi u stanje „Završeno“. U završnom stanju se vrši resetiranje sustava kako bi se vratio na početno stanje „Čeka novac“ i ponovio proces za sljedećeg korisnika. Ovaj automatski proces prikazuje kako Mooreov automat funkcionira s obzirom na prijelaze između stanja i jasno definirane radnje unutar svakog stanja.

4.1.2.2. Mealyjev automat

Mealyjev model transduktora je konačnog automata gdje su izlazni signali povezani s prijelazima između stanja, a ne sa samim stanjima kao u Mooreovom modelu. U ovom modelu, reakcije automata, njegove izlazne akcije, određene su ne samo trenutnim stanjem, već i ulaznim signalima koji uzrokuju prijelaze između stanja. [13]. Mealyjev model je nešto teži i složeniji za razumijevanje u usporedbi s Mooreovim modelom, ali omogućava bolju uporabu stanja, te zahtjeva manje stanja za postizanje istih funkcionalnosti.



Slika 4: Primjer Mealyjevog automata

Na primjeru Mealyjev automat prikazanog na slici 4. koji predstavlja aparat za piće i prikazuje proces kupnje pića u kojem izlaz automata ovisi o ulaznim podacima i prijelazima između stanja, a ne samo o trenutnom stanju, što je karakteristika Mealyjevog modela.

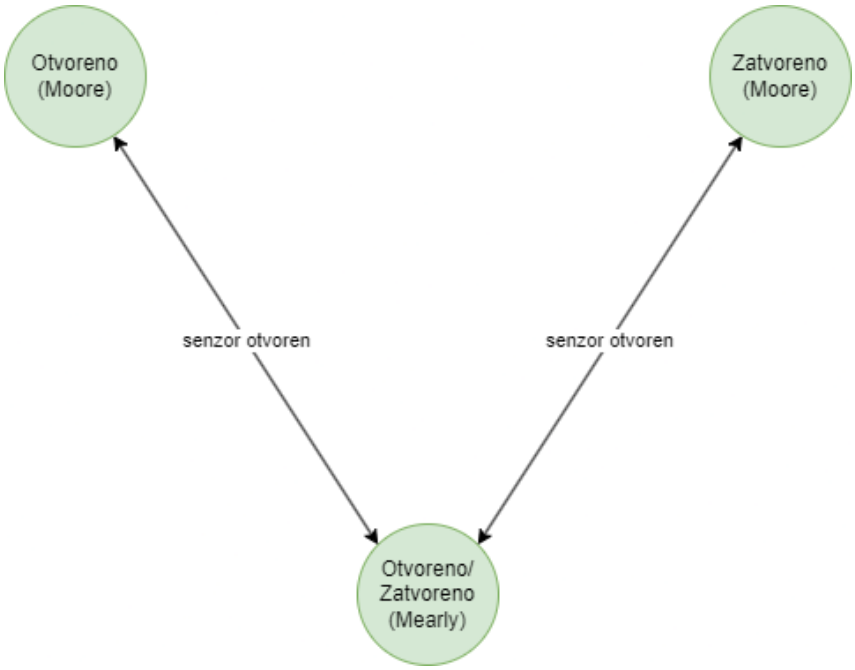
Automat počinje u stanju "Čeka novac", gdje čeka na unos novca. Kada je novac ubačen, automat prelazi u stanje "Provjerava novac", gdje se provjerava je li ubačena dovoljna količina novca. Ako je novca dovoljno, automat prelazi u stanje "Izdaje piće", gdje izdaje piće i generira odgovarajuću akciju. Ako je ubačen višak novca, automat prelazi u stanje "Vraća višak novca". Nakon vraćanja viška novca, ili izdavanja pića, automat završava u stanju "Završeno", gdje vraća novac i vraća se u početno stanje putem prijelaza Restartiranje.

Razlika između Mealyjevog i Mooreovog automata je u načinu na koji se generiraju izlazi. U Mealyjevom automatu izlaz ovisi o prijelazima između stanja i ulaznim uvjetima. U ovom primjeru, akcije poput "Izdaje piće" ili "Vraća višak novca" aktiviraju se prilikom prijelaza između stanja. Nasuprot tome, kod Mooreovog automata, izlazi su vezani uz stanja, a to znači

da izlaz (akcija) nastupa samo kada automat uđe u određeno stanje, neovisno o prijelazima.

4.1.2.3. Mješoviti automat

Mješoviti model konačnog automata kombinira najbolje karakteristike Mooreovog i Mealyjevog modela. U mješovitom modelu, određena stanja koriste pristup tipičan za Mooreov model, gdje su akcije povezane s ulaskom u određeno stanje, dok druga stanja kombiniraju ulazne akcije karakteristične za Mealyjev model. [13]. Na ovaj način omogućeno je da automat ima manje stanja, ali i da zadrži strukturu i preglednost sustava.



Slika 5: Primjer Miješovitog automata

Slika 5. prikazuje primjer mješovitog automata za kontrolu vrata dizala. U ovom slučaju postoje Mooreova stanja "Otvoreno" i "Zatvoreno", gdje automat signalizira stanje vrata (otvorena ili zatvorena) kada se u njih uđe. Mealyjevo stanje "Otvaranje/Zatvaranje" koristi se tijekom prijelaza između otvorenih i zatvorenih vrata, a akcije poput pokretanja motora ovise o ulaznim signalima ("senzor zatvoren" ili "senzor otvoren"). Korištenjem mješovitog automata omogućeno je smanjuje broj stanja i osigurano je učinkovitije ponašanje automata.

4.2. Matematički model konačnog automata

Konačni automat je matematički model koji se koristi za prikazivanje i analizu sustava u određenim stanjima. Postoji nekoliko različitih vrsta konačnih automata, od kojih su najpoznatiji prihvatitelji konačni automat i transduktori konačni automat. [14]

Prihvatitelji konačni automat koji se koristi za prepoznavanje određenih nizova ulaznih simbola, to jest prepoznavanje obrazaca unutar njih, a definira se kao petorka $(\Sigma, S, s_0, \delta, F)$. Ova struktura opisuje kako automat obrađuje ulazne simbole i prolazi kroz različita stanja, za-

vršavajući u konačnom stanju ako je ulaz prihvaćen. [14]

U ovoj petorki: [15]

- Σ je ulazna abeceda (konačan neprazan skup simbola).
- S je konačan neprazan skup stanja.
- s_0 je početno stanje, element skupa S. U nedeterminističkom konačnom automatu, s_0 je skup početnih stanja.
- δ je funkcija prijelaza: $\delta : S \times \Sigma \to S$.
- F je skup konačnih stanja, (potencijalno prazan) podskup od S.

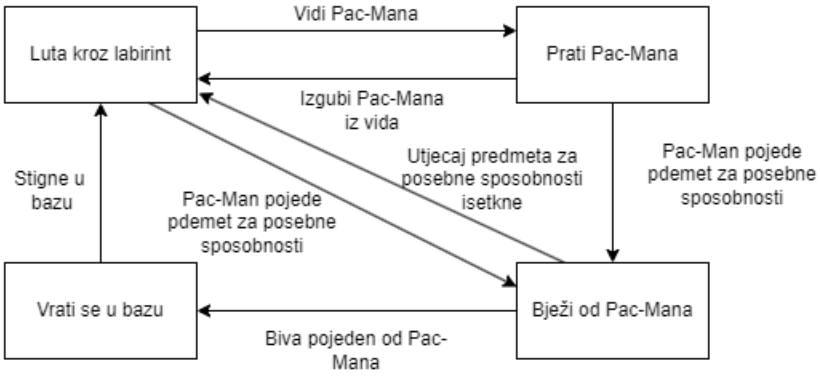
Transduktorski konačni automati koji također prepoznaje nizove simbola , ali u isto vrijeme generiraju izlazne nizove na temelju ulaznih podataka, definira se kao šestorka $(\Sigma, \Gamma, S, s_0, \delta, \omega)$. Ovaj model prolazi kroz stanja na temelju ulaznih simbola, ali također generira odgovarajuće izlazne simbole, što omogućuje složenije operacije poput obrade signala ili kodiranja podataka. [16]

U ovoj šestorki: [15]

- Σ je ulazna abeceda (konačan neprazan skup simbola).
- Γ je izlazna abeceda (konačan neprazan skup simbola).
- S je konačan neprazan skup stanja.
- s_0 je početno stanje, element skupa S. U nedeterminističkom konačnom automatu, s_0 je skup početnih stanja.
- δ je funkcija prijelaza: $\delta : S \times \Sigma \to S$.
- ω je izlazna funkcija.

4.3. Pac-Man model konačnog automata

Pac-Man, kao jedan od pionira u svijetu igara, ostavio je značajan trag u povijesti videoigara. Igra je objavljena još davne 1980. godine i bila je posebna po tome što je koristila različite modele ponašanja za svoje neprijatelje, duhove. Svaki duh imao je jedinstven obrazac kretanja [17]. Blinky, crveni duh, uvijek je izravno pratio Pac-Mana i postajao brži kad bi ostalo malo bodova na ploči. Pinky, ružičasti duh, pokušavao je presjeći put Pac-Manu, ciljajući polje četiri mjesta ispred njega. Inky, plavi duh, kombinirao je svoju poziciju s Blinkyjevom kako bi odabrao ciljnu ploču i zbog toga je imao najnepredvidljivije ponašanje. Clyde, narančasti duh, povlačio se u kut kad bi se previše približio Pac-Manu, ali ga je inače pratio kad je bio dovoljno udaljen [18]. Neki od ovih obrazaca ponašanja mogli su biti modelirani pomoću konačnih automata, ali nisu svi duhovi nužno koristili isti tip automata ili ga uopće koristili. Na slici 6. prikazan je globalni koncept konačnog automata za videoigru Pac-Man.



Slika 6: Dijagram stanja za igru Pac-Man, prema [17]

Dijagram stanja predstavlja ponašanje duhova u igri Pac-Man. Svako stanje u dijagramu predstavlja različitu aktivnost duha, dok strelice između njih prikazuju prijelaze između aktivnosti na temelju određenih uvjeta. [17]

U početnom stanju duhovi se kreću kroz labirint kako bi pronašli Pac-Mana. Ovo stanje naziva se **"Luta kroz labirint"**. Kada duh uoči Pac-Mana, prelazi u stanje **"Prati Pac-Mana"**, gdje pokušava uloviti Pac-Mana, ako Pac-Man nestane iz vidokruga duha, duh se vraća u prvobitno stanje i nastavlja lutati labirintom.

Ako Pac-Man pojede predmet za posebne sposobnosti, duhovi mijenjaju svoje ponašanje i prelaze u stanje **"Bježi od Pac-Mana"**, jer Pac-Man sada može pojesti duhove. U ovom stanju, duhovi bježe od Pac-Mana dok traje učinak posebne sposobnosti. Kada učinak predmeta za posebne sposobnosti istekne, duhovi se vraćaju u stanje **"Prati Pac-Mana"** i ponovno počinju loviti Pac-Mana.

U slučaju da duh bude pojeden od strane Pac-Mana, prelazi u stanje **"Vrati se u bazu"**. U ovom stanju duh se vraća u svoju bazu da bi se oporavio. Kada duh stigne u bazu i oporavi se, vraća se u početno stanje i ponovno počinje lutati labirintom.

Upotreba konačnog automata omogućuje definiranje svih prijelaza na nepredvidljiv način, što donosi izazovnosti same igre, što čini Pac-Man jednom od najutjecajnijih igara u povijesti videoigara.

5. Primjena konačnog automata u igri City Siege

Produkt praktičnog dijela ovog završnog rada je videoigra u kojoj su neprijateljski likovi modelirani i kontrolirani s pomoću konačnih automata. U nastavku će se detaljno razmotriti funkcionalnosti igre, poput upravljanja oružjem, pucanja, prikupljanja oružja i municije, te način na koji su konačni automati primijenjeni na neprijateljske likove kako bi definirali njihovo ponašanje i interakciju s igračem. Primjena konačnih automata omogućava realističniju simulaciju ponašanja neprijatelja, čime igra postaje izazovnija i zanimljivija za igranje.

5.1. Igra City Siege i njene funkcionalnosti

Sama igra smještena je u urbano okruženje s perspektivom iz prvog lica, gdje igrač ima ulogu protagonista koji mora eliminirati neprijateljske likove. Inspiracija za razvoj ovog projekta dolazi iz popularnih 3D pucačkih igara u prvom licu, poput Call of Duty i Dooma. Cilj same igre je pobijediti neprijatelja i doći do cilja.

U igri igraču je omogućeno slobodno kretanje po mapi, a osim standardnog hodanja, igrač može puzati ili trčati. Na početku igre dostupan je samo jedno oružje, no tijekom igranja igrač može prikupiti još dva dodatna oružja. Za svako oružje potrebno je prikupljati dodatnu municiju kako bi igrač mogao napredovati kroz izazove u igri. Također, igra omogućava bacanje oružja. Ako igrač pokupi oružje koje već posjeduje, umjesto da dobije novo oružje, bit će mu dodana samo metci u rezervu. U igri su prisutni i neprijateljski likovi koji pokušavaju eliminirati igrača ispaljujući svjetleće projektile prema njemu.

5.2. Resursi i dodatci korišteni u igri

Za izgradnju mape i građevina u igri korišten je Cyclops Level Builder, alat koji je omogućio jednostavno i efikasno kreiranje okruženja u igri. Pomoću ovog alata izgrađene su građevine poput kuća, zidova i puteva, te je tako stvoren prostor unutar igre.

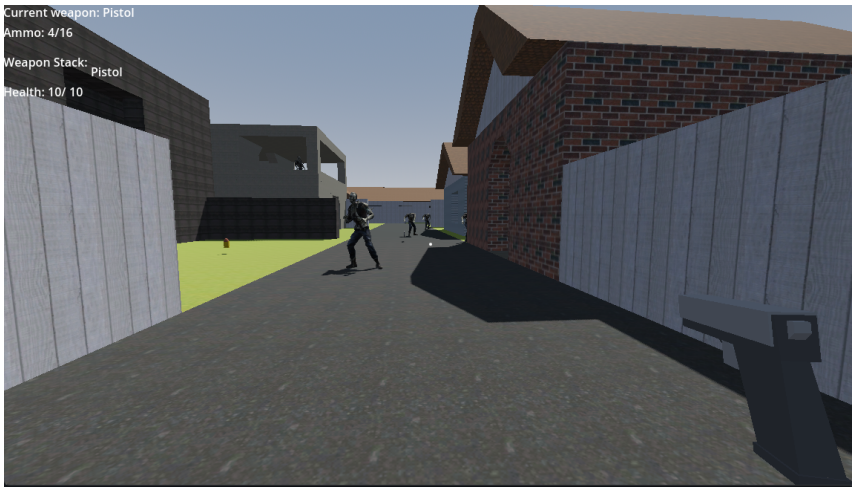
Za izradu igre korišteni su različiti resursi (eng. assets). Teksture korištene u igri preuzete su sa stranice Itch.io, gdje je korišten Classic 64 Asset Pack za kreiranje vizualnog izgleda okoline i objekata. Resursi za oružja preuzeti su sa stranice Kenney, gdje je korišten 3D Weapon Kit za modele i animacije oružja u igri. Ovi resursi značajno su poboljšali kvalitetu grafike i izgelda same igre.

5.3. Funkcionalnosti i mogućnosti igre

5.3.1. Početni zaslon igre

Pri samom ulasku u videoigru vidljiva je perspektivu iz prvog lica, gdje igrač drži pištolj, te je okružen kućama. U gornjem lijevom kutu zaslona vidljive su osnovne informacije

o trenutnom statusu igrača, uključujući naziv trenutnog oružja (eng. Current weapon), preostalu količinu metaka (eng. Ammo), oružja u arsenalu (eng. Weapon Stack), te zdravlje igrača (eng. Health). Na samoj početnoj točci videoigre se nalaze neprijateljski likovi, iz čega se može pretpostavit da će igrač uskoro stupiti u sukob s njima.

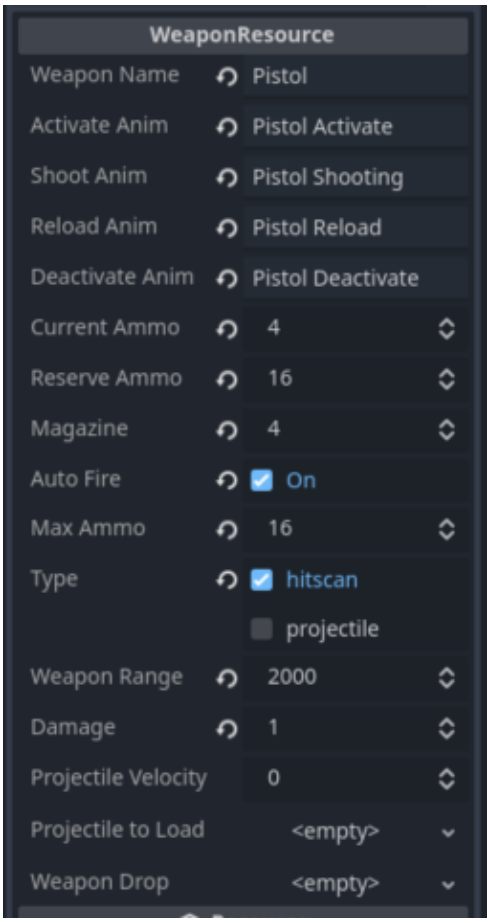


Slika 7: Početni prizor videoigre

Igrač se kreće kroz igru pomoću tipki WASD, pri čemu tipka W omogućuje kretanje naprijed, tipka S kretanje unatrag, dok tipke A i D omogućuju pomicanje ulijevo i udesno. Pucanje se izvodi pritiskom na lijevu tipku miša, dok kotačić miša (eng. scroll) omogućuje promjenu oružja. Tipka R služi za ponovno punjenje trenutno odabranog oružja, a tipkom Q igrač može baciti oružje koje trenutačno koristi. Pritiskom na tipku Control igrač može puziti, dok se držanjem tipke Shift omogućuje trčanje. Pritiskom na tipku Escape igrač može izaći iz igre i potpuno je zatvoriti.

5.4. Upravljanje oružijem

Kako bi se olakšalo dodavanje oružja u igru, kreiraj je resurs za oružje (eng. weapon resource) koji omogućava jednostavno upravljanje oružjima u igri, olakšavajući njihovo dodavanje i prilagodbu bez potrebe za izmjenom koda. Svako oružje definirano je kroz različite parametre, kao što su animacije aktivacije, pucanja, punjenja i deaktivacije, te broj metaka, kapacitet spremnika i rezervna municija. Oružje može imati različite tipove paljbe, uključujući automatsku ili pojedinačnu paljbu, te može biti kategorije "hitscan" za trenutačne pogotke ili "projectile" za ispaljivanje projektila s određenom brzinom i dometom. Ako je oružje tipa projectile, u resursu je omogućeno i unos modela projektila koji će biti ispaljen. Za svaki projektil može definirati i njegov izgled. Na taj način, svaki projektil može imati jedinstveni model koji odgovara tipu oružja koje ga ispaljuje. U igru je implementiran bacač raketa (eng. rocket laucher).Na slici 8. prikazan je primjer Weapon Resource za pištolj dok su na slici 9. prikazana oružja koja su koriste u videoigri.



Slika 8: Prikaz resursa za oružije

Slika 9: Prikaz oružja korištenih u igri

5.4.1. Mehanika pucanja

Budući da je žanr igre shooter, ključan aspekt je omogućiti igraču da precizno pogodi i eliminira neprijatelje. Smjer pucanja određuje se kursorom miša, kojim igrač cilja i usmjerava metak ili projektil. Brzina pucanja, odnosno koliko brzo oružje ispaljuje metke, određena je trajanjem animacije specifične za svako oružje. U nastavku se nalazi kod za pucanje i igri:

```
func shoot():
        if current_weapon.current_ammo != 0:
                if !animation_player.is_playing():
                        animation_player.play(current_weapon.shoot_anim)
                        current_weapon.current_ammo -= 1
                        emit_signal("update_ammo", [current_weapon.current_ammo,
                            current_weapon.reserve_ammo])
                        $"../../Sounds/ShootSound".play()

                        var camera_collision = get_camera_collision()
```

```
                        match current_weapon.type:
                                NULL:
                                        print("tip oruzija nije odabran")

                                HITSCAN:
                                        hitscan_collision(camera_collision)

                                PROJECTILE:
                                        launch_projectile(camera_collision)
        else:
                reload()

  func hitscan_collision(collision_point):
        var bullet_direction = (collision_point - bullet_point.get_global_transform
            ().origin).normalized()
        var new_intersection = PhysicsRayQueryParameters3D.create(bullet_point.
            get_global_transform().origin, collision_point + bullet_direction * 2)
        var bullet_collision = get_world_3d().direct_space_state.intersect_ray(
            new_intersection)

        if bullet_collision:
                var hit_indc = BulletShot.instantiate()
                var world = get_tree().get_root().get_child(0)
                world.add_child(hit_indc)
                hit_indc.global_translate(bullet_collision.position)
                hitscan_damage(bullet_collision.collider, bullet_direction,
                    bullet_collision.position)

func hitscan_damage(collider, direction, collider_position):
        if collider.is_in_group("Target") and collider.has_method("hit_successful"):
                collider.hit_successful(current_weapon.damage, direction,
                    collider_position)

func launch_projectile(point : Vector3):
        var direction = (point - bullet_point.global_transform.origin).normalized()
        var projectile = current_weapon.projectile_to_load.instantiate()
        var projectile_RID = projectile.get_rid()
        collision_exclusion.push_back(projectile_RID)
        projectile.tree_exited.connect(remove_exclusion.bind(projectile.get_rid()))

        bullet_point.add_child(projectile)
        projectile.damage = current_weapon.damage
        projectile.set_linear_velocity(direction * current_weapon.
            projectile_velocity)

func get_camera_collision() -> Vector3:
        var camera = get_viewport().get_camera_3d()
        var viewport = get_viewport().get_size()

        var ray_origin = camera.project_ray_origin(viewport/2)
        var ray_end = ray_origin + camera.project_ray_normal(viewport/2) *
            current_weapon.weapon_range
```

```
        var new_intersection = PhysicsRayQueryParameters3D.create(ray_origin,
            ray_end)
        new_intersection.set_exclude(collision_exclusion)


        var intersection = get_world_3d().direct_space_state.intersect_ray(
            new_intersection)

        if not intersection.is_empty():
                var col_point = intersection.position
                return col_point

        else:
                return ray_end
```

Ovaj kod prikazuje mehaniku pucanja u igri. Kada igrač pritisne gumb za pucanje, prvo se provjerava je li trenutno oružje napunjeno. Ako ima metaka, pokreće se animacija pucanja, smanjuje se broj metaka u spremniku, emitira se signal za ažuriranje metaka, te se reproducira zvuk pucnja. Nakon toga, kod provjerava tip oružja. Ako je oružje hitscan tipa, koristi se metoda za trenutačno prepoznavanje kolizije pomoću zrake (raycast) iz kamere, kako bi se simulirao pogodak metka. Ako je metak pogodio metu, šteta se oduzima od neprijateljevog zdravlja (eng. helth). Ako je oružje projectile tipa, ispaljuje se projektil prema smjeru cilja, ovisno o putanji koju kamera detektira. Na taj način, kod podržava različite vrste oružja, bilo da koriste trenutačno pogotke (hitscan) ili ispaljuju projektile.

5.4.2. Punjenje oružja metcima

Jedna od bitnijih funkcionalnosti u igri je punjenje oružja metcima. Kada oružje ostane bez metaka, automatski će se napuniti ukoliko igrač ima dovoljno rezervne municije. Međutim, ako više nema metaka u rezervi, oružje neće moći biti napunjeno. U nastavku se nalazi prikaz koda koji implementira ovu mehaniku punjenja:

```
func reload():
        if current_weapon.current_ammo == current_weapon.magazine:
                return
        elif !animation_player.is_playing():
                if current_weapon.reserve_ammo != 0:
                        animation_player.play(current_weapon.reload_anim)

                        var reload_amount = min(current_weapon.magazine -
                            current_weapon.current_ammo, current_weapon.magazine,
                            current_weapon.reserve_ammo)
                        current_weapon.current_ammo += reload_amount
                        current_weapon.reserve_ammo -= reload_amount
                        emit_signal("update_ammo", [current_weapon.current_ammo,
                            current_weapon.reserve_ammo])
                        $"../../Sounds/Reload".play()
                else:
                        $"../../Sounds/NotEnoughtAmmo".play()
```

Kod prikazuje funkcionalnost punjenja oružja u igri. Prvo provjerava je li spremnik već pun, i ako jest, prekida proces. Ako spremnik nije pun i oružje trenutno ne izvodi drugu animaciju, te postoji rezervna municija, pokreće se animacija punjenja. Zatim se izračunava koliko metaka može biti napunjeno, oduzima se taj broj iz rezerve i dodaje u spremnik, te se emitira signal za ažuriranje prikaza municije. Ako nema dovoljno rezervne municije, reproducira se zvuk koji obavještava igrača da nema više metaka u rezervi.

5.4.3. Prikupljanje oružija i metaka

Jedan od bitnih aspekata pucačkih igara je prikupljanje oružja i metaka, što omogućuje igraču da proširuje svoj arsenal i osigurava dovoljno resursa za borbu protiv neprijatelja. Kada igrač naiđe na novo oružje, ono se automatski dodaje u njegov arsenal oružja, omogućujući mu korištenje različitih tipova naoružanja. S druge strane, prikupljanje metaka omogućava igraču da puni svoja oružja i osigurava kontinuiranu borbenu sposobnost.

```
func _on_pickup_area_body_entered(body):
        if body.is_in_group("PickupGroup"):
                if body.type == "Weapon" and body.pickup_ready:
                        var weapon_in_stack = weapon_stack.find(body.weap_name, 0)
                        if weapon_in_stack == -1:
                                var getRef = weapon_stack.find(current_weapon.
                                    weapon_name)
                                weapon_stack.insert(getRef, body.weap_name)
                                emit_signal("update_weapon_stack", weapon_stack)
                                exit(body.weap_name)
                                body.queue_free()
                        elif add_ammo(body.weap_name, weapon_list[body.weap_name].
                            magazine):
                                body.queue_free()
                elif body.type == "Ammo" and body.pickup_ready:
                        if add_ammo(body.weap_name, weapon_list[body.weap_name].
                            magazine):
                                body.queue_free()

func add_ammo(_name, ammo) -> bool:
        var required = weapon_list[_name].max_ammo - weapon_list[_name].reserve_ammo
        if required > 0:
                var remaining = max(ammo, required, 0)
                weapon_list[_name].reserve_ammo += min(ammo, required)
                emit_signal("update_ammo", [current_weapon.current_ammo,
                    current_weapon.reserve_ammo])
        return required > 0
```

Kod prikazuje funkcionalnost prikupljanjem municije i oružja, te dodavanjem municije igračevim oružjima. Funkcija `add_ammo` izračunava koliko municije je potrebno oružju, temeljem razlike između maksimalne količine municije i trenutne rezerve. Ako je potrebno više municije, dodaje onoliko metaka koliko je moguće, emitira signal za ažuriranje prikaza municije i vraća logičku vrijednost koja označava je li igrač mogao dodati municiju. Funkcija, `_on_pickup_area_body_entered`, prepoznaje ulazak igrača u područje prikupljanja oružja

ili municije. Ako igrač pokupi oružje koje još nema, ono se dodaje u arsenal oružja, te se ažurira prikaz oružja, a objekt se uklanja iz igre. Ako igrač već ima to oružje, dodaje se municija. Slično tome, kada igrač pokupi municiju, funkcija dodaje metke u odgovarajuće oružje i uklanja objekt iz igre.

5.4.3.1. Objekt za prikupljanje oružja i metaka

Kako bi igrač mogao prikupiti oružje ili metke u igri, potrebno je implementirati sustav koji omogućava prikupljanje tih resursa. Na temelju odabranog oružja (pištolj, strojnica ili raketni bacač), kod učitava odgovarajući 3D model oružja. Iako je u kodu prikazan postupak za učitavanje oružja, isti princip vrijedi i za metke.

```
extends RigidBody3D
class_name Example
var type := "Weapon"
@export_enum("Pistol", "Machinegun", "Rocketlauncher") var weap_name   = "Pistol"
@onready var mesh_instance_3d = $MeshInstance3D
var pickup_ready := false
func  _ready():
        match weap_name:
                "Pistol":
                        mesh_instance_3d.set_mesh(load("res://3dModels/Weapon3dModel
                            /pistol.obj"))

                "Machinegun":
                        mesh_instance_3d.set_mesh(load("res://3dModels/Weapon3dModel
                            /machinegun.obj"))

                "Rocketlauncher":
                        mesh_instance_3d.set_mesh(load("res://3dModels/Weapon3dModel
                            /rocketlauncher.obj"))
        await get_tree().create_timer(1).timeout
        pickup_ready = true
```

5.4.4. Bacanje oružja iz arsenala

Još jedna od mogućnosti u igri je izbacivanje oružja iz arsenala, što igrač može učiniti pritiskom na tipku Q na tipkovnici. Ova funkcionalnost omogućava igraču da se riješi nepotrebnog oružja, što može biti korisno u situacijama kada igrač preferira korištenje specifičnog tipa oružja.

```
func drop(_name : String):
        if weapon_stack.size() > 1:
                var weapon_ref = weapon_stack.find(_name, 0)
                if weapon_ref != -1:
                        weapon_stack.pop_at(weapon_ref)
                        emit_signal("update_weapon_stack", weapon_stack)
                        var weapon_dropped = WP_Pickup.instantiate()
                        if weapon_dropped.has_method("set_weapon_name"):
                                weapon_dropped.set_weapon_name(_name)
```

```
else:
        weapon_dropped.weap_name = _name
var world = get_tree().get_root().get_child(1)
world.add_child(weapon_dropped)
weapon_dropped.global_transform = bullet_point.
    global_transform
var getRef = weapon_stack.find(current_weapon.weapon_name)
getRef = max(getRef - 1, 0)
exit(weapon_stack[getRef])
```

Funkcija `drop` omogućava igraču da izbaci oružje iz svog arsenala. Prvo provjerava ima li igrač više od jednog oružja, kako bi spriječila bacanje zadnjeg preostalog oružja. Ako igrač ima više oružja, funkcija pronalazi poziciju oružja koje treba biti izbačeno iz arsenala i uklanja ga iz popisa oružja. Nakon toga, kreira se novi objekt za izbačeno oružje u svijetu igre, postavlja se ime tog oružja, te se ono dodaje na odgovarajuću poziciju u sceni. Oružje se pojavljuje na mjestu gdje se igrač nalazi, a igrač automatski prelazi na korištenje sljedećeg dostupnog oružja u svom arsenalu.

5.4.5. Nanošenje štete igrač i neprijatelju

Jedan od ključnih elemenata igre je mehanizam nanošenja štete igraču i neprijatelju. U nastavku će biti prikazan primjer koda koji oduzima zdravlje (eng. health) neprijatelju kada je pogođen. Iako se ovdje fokusiramo na neprijatelja, isti princip vrijedi i za igrača. Ovaj sustav oduzimanja zdravlja osigurava da se, nakon primljene štete, smanjuje količina zdravlja (eng. health) neprijatelja ili igrača, a kad zdravlje padne na nulu, lik biva eliminiran iz igre.

Prikaz koda za nanošenje štete igrači i neprijatelju:

```
func hit_successful(damage, _Direction := Vector3.ZERO, _Position := Vector3.ZERO):
        var hit_position = _Position - global_transform.origin
        health -= damage
        if health <= 0:
                queue_free()
```

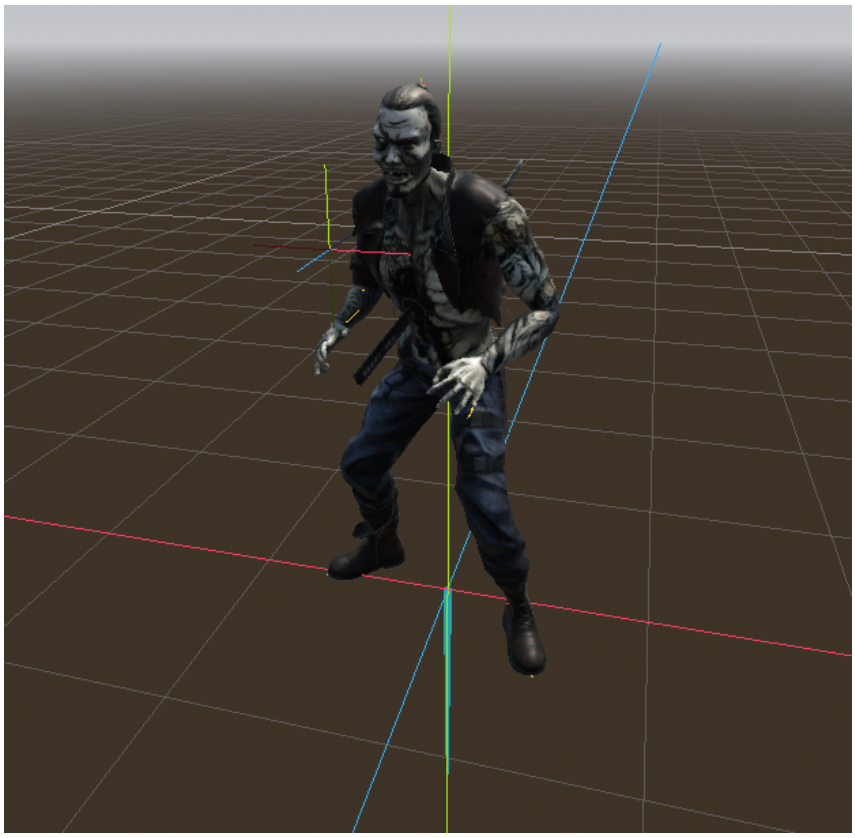
5.5. Neprijateljski likovi kao konačni automati

Konačni automat u igri koristi se za upravljanje ponašanjem neprijateljskog lika. Kako bi se taj automat kreirao, bilo je potrebno definirati različita stanja i prijelaze između tih stanja. Svako stanje predstavlja specifično ponašanje neprijatelja, dok prijelazi određuju kada i kako neprijatelj prelazi iz jednog stanja u drugo, ovisno o određenim uvjetima, poput detekcije igrača ili dolaska do ciljane točke u igri.

5.5.1. Neprijateljski lik u igri

Na slici 10. prikazan je neprijateljski lik u igri, čiji je glavni cilj eliminirati igrača. Neprijatelj napada ispaljivanjem svjetlećih projektila koji igraču oduzimaju zdravlje (eng. health). Svaki

projektil predstavlja prijetnju, a igrač mora precizno izbjeći projektil kako bi izbjegao gubitak zdravlja i nastavio napredovati kroz igru.



Slika 10: Prikaz neprijateljskog igrača

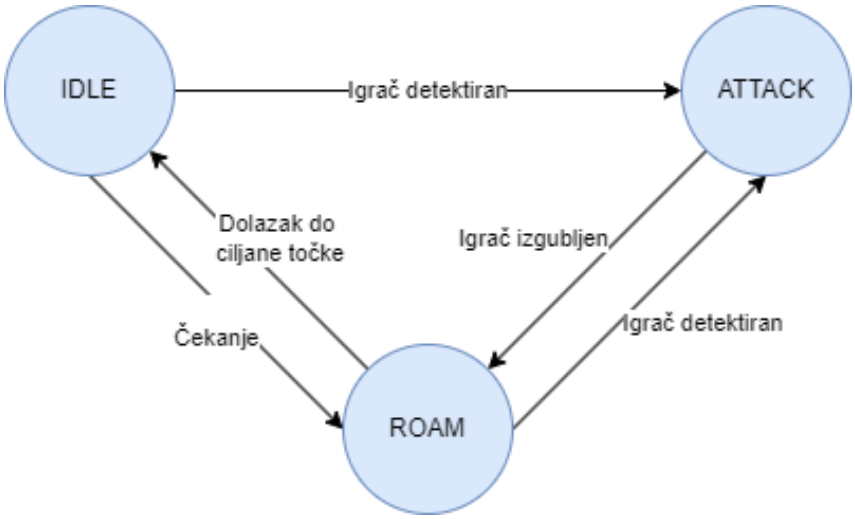
Model neprijateljskog lika u igri preuzet je sa stranice Mixamo, a radi se o modelu pod nazivom Yaku J Ignite. Mixamo je platforma s koje je moguće preuzeti 3D modele s gotovim animacijama, što je olakšalo implementaciju neprijateljskog lika. Za dodatnu prilagodbu i uređivanje animacija korišten je alat Blender, pomoću kojeg su uređeni pokreta neprijatelja, od njegovih borbenih animacija do kretanja unutar same igre.

5.5.2. Model ponašanja neprijatelja putem konačnog automata

Konačni automat prikazan na slici 11. prikazuje dijagram ponašanja neprijateljskog lika kroz tri stanja: čekanje (IDLE), lutanje (ROAM) i napad (ATTACK). U stanju čekanja, neprijatelj miruje dok ne prođe određeno vrijeme, nakon čega prelazi u stanje lutanja. U stanju lutanja, neprijatelj se nasumično kreće po mapi, a kada dosegne ciljnu točku, vraća se u stanje čekanja. Ako u bilo kojem trenutku detektira igrača, prelazi u stanje napada. Kada napada igrača, neprijatelj se zaustavlja i počinje ispaljivati svjetleće projektile prema igraču. Ako izgubi igrača iz vidokruga, vraća se u stanje lutanja kako bi ga ponovno tražio. Na ovaj način je osigurano da neprijatelj reagira na promjene u okolini i igračevim akcijama.

Kod konačnog automata za neprijateljskog igrača nalazi se u nastavku:

```
enum {IDLE, ROAM, ATTACK}
```



Slika 11: Grafički prikaz konačnog automata za igru

```
var currentState = IDLE:
        set(value):
                currentState = value
                match value:
                        IDLE:
                                animation_player.play("Idle")
                                speed = 0.001
                                $moveTimer.start()
                        ROAM:
                                animation_player.play("Walk")
                                speed = 5.5

                        ATTACK:
                                var direction = (detectedPlayer.global_position -
                                    $enemyExample/bulletPoint.global_position).
                                    normalized()
                                $enemyExample/Armature.rotation.y = atan2(direction.
                                    x, direction.z)
                                animation_player.play("Attacking")
                                await animation_player.animation_finished

                                var nBullet = enemy_prijectile.instantiate()
                                get_parent().add_child(nBullet)
                                nBullet.global_position = $enemyExample/bulletPoint.
                                    global_position

                                nBullet.damage = damage
                                nBullet.set_linear_velocity(direction *
                                    projectile_velocity)
                                nBullet.look_at(detectedPlayer.global_position,
                                    Vector3.UP,true)
                                $DetectPlayerArea/CollisionShape3D.disabled = true
                                $DetectPlayerArea/CollisionShape3D.disabled = false
                                $moveTimer.start()
```

```
func _on_detect_player_area_body_entered(body):
        if body is Player:
                print("player detected")
                $moveTimer.stop()
                detectedPlayer = body
                currentState = ATTACK


func _on_detect_player_area_body_exited(body):
        if body is Player:
                currentState = ROAM
                _on_move_timer_timeout()
                var sphere_point = get_random_pos_insphere(roamDistance)
                move_to(sphere_point)

func _on_navigation_agent_3d_navigation_finished():
        currentState = IDLE

func _on_move_timer_timeout():
        var sphere_point = get_random_pos_insphere(roamDistance)
        move_to(sphere_point)

func move_to(target_pos):
        currentState = ROAM
        var clossest_pos = NavigationServer3D.map_get_closest_point(get_world_3d().
            get_navigation_map(), target_pos)
        $NavigationAgent3D.set_target_position(clossest_pos)
```

Ovaj kod implementira ponašanje neprijateljskog lika koristeći konačni automat s tri glavna stanja: čekanje (IDLE), lutanje (ROAM) i napad (ATTACK). Neprijatelj se u stanju čekanja miruje i pokreće timer koji određuje koliko dugo će ostati u tom stanju. Kada timer istekne, neprijatelj prelazi u stanje lutanja, gdje se nasumično kreće po mapi prema odabranoj točki unutar određenog radijusa. Brzina kretanja u tom stanju je postavljena na višu vrijednost kako bi neprijatelj simulirao patroliranje. Nakon što dosegne ciljnu točku, vraća se u stanje čekanja, ali ako tijekom lutanja detektira igrača, prelazi u stanje napada.

U stanju napada neprijatelj se prestaje kretati, te se rotira prema igraču i ispaljuje projektil u njegovom smjeru. Projektil je instanciran, postavljen na neprijateljevu poziciju, a brzina ispaljivanja određena je unaprijed definiranom vrijednošću. Nakon ispaljivanja projektila, neprijatelj nastavlja napadati igrača sve dok je on u njegovom detekcijskom području. Ako igrač napusti to područje, neprijatelj prestaje s napadom i vraća se u stanje lutanja, nasumično birajući novu točku prema kojoj će se kretati. Navigacija neprijatelja po mapi kontrolira se navigacijskim agentom (eng.NavigationAgent), a kada neprijatelj završi navigaciju prema točki, prelazi natrag u stanje čekanja. Cijeli sustav ponašanja temelji se na prijelazima između ovih stanja, ovisno o prisutnosti igrača i stanju kretanja.

5.5.3. Kretanje neprijateljskog lika

Kako bi neprijateljski likovi u igri djelovali što realističnije, bilo je važno implementirati sustav koji omogućava njihovo kretanje prema određenoj točki odakle igrač započinje igru. Pritom je ključno osigurati da neprijatelji tijekom kretanja ostanu pravilno orijentirani prema cilju. Kada neprijatelj detektira igrača, zaustavlja se, rotira prema njemu i prelazi u stanje napada. U nastavku je prikazan kod koji implementira kreiranje neprijateljskog lika u igri, uključujući njegovo kretanje, detekciju i napad na igrača.

```
func _physics_process(delta):
        if !is_on_floor():
                velocity.y -= 100
                move_and_slide()
                return

        vel = Vector3.ZERO

        var target = $NavigationAgent3D.get_next_path_position()
        var pos = global_transform.origin
        var n = $RayCast3D.get_collision_normal()
        if n.length_squared() < 0.001:
                n = Vector3.UP

        vel = (target - pos).slide(n).normalized() * speed

        if currentState == ATTACK:
                var attack_direction = (detectedPlayer.global_position -
                    global_position).normalized()
                $enemyExample/Armature.rotation.y = lerp_angle(
                $enemyExample/Armature.rotation.y,
                        atan2(attack_direction.x, attack_direction.z),
                        delta * 10
                )
        else:
                if vel != Vector3.ZERO and vel.length() > 0.1:
                        var move_direction_angle = atan2(vel.x, vel.z)
                        $enemyExample/Armature.rotation.y = lerp_angle(
                                $enemyExample/Armature.rotation.y,
                                move_direction_angle,
                                delta * 10
                        )
        if currentState != ATTACK:
                velocity = vel
                move_and_slide()
        else:
                $NavigationAgent3D.set_velocity(Vector3.ZERO)
                move_and_slide()

func _on_navigation_agent_3d_velocity_computed(safe_velocity):
        set_velocity(safe_velocity)
```

```
func _on_navigation_agent_3d_navigation_finished():
        currentState = IDLE
```

Prikazani kod upravlja kretanjem neprijateljskog lika koristeći navigacijskog agenta (NavigationAgent3D), koji određuje sljedeću ciljnu točku na mapi prema kojoj se neprijatelj kreće. Tijekom kretanja, neprijatelj se prilagođava terenu i rotira u smjeru kretanja kako bi se njegovo ponašanje činilo prirodnijim. Ako je neprijatelj u stanju napada, rotira se prema igraču kako bi ga napadao. Funkcija `move_and_slide()` upravlja fizičkim kretanjem neprijatelja, omogućavajući mu nesmetano kretanje po terenu. Na neprijatelja se primjenjuje gravitacijska sila ukoliko neprijateljski igrač nije na podu. Nakon što neprijatelj stigne do zadane točke, završava navigaciju i prelazi u stanje mirovanja (IDLE), čekajući nove naredbe ili događaje koji će ga vratiti u kretanje ili napad.

5.5.4. Generiranje lokacije kretanja

Kako bi se omogućilo kretanje igrača u igri, bilo je potrebno implementirati sustav generiranja nasumičnih pozicija za kretanje neprijatelja. Ovaj sustav omogućava neprijateljima da se kreću po mapi. U nastavku će biti prikazan kod za generiranje nasumičnih pozicija igrača:

```
func get_random_pos_insphere(radius: float) -> Vector3:
        var x1 = randi_range(-1, 1)
        var x2 = randi_range(1, -1)

        while x1*x1 + x2*x2 >= 1:
                x1 = randi_range(-1, 1)
                x2 = randi_range(1, -1)

        var random_pos_on_unit_sphere = Vector3(
                1 - 2 * (x1*x2 + x2*x2),
                0,
                1 - 2 * (x1*x2 + x2*x2))

        random_pos_on_unit_sphere.x *= randi_range(-radius, radius)
        random_pos_on_unit_sphere.z *= randi_range(-radius, radius)

        return random_pos_on_unit_sphere

 func move_to(target_pos):
        currentState = ROAM
        var clossest_pos = NavigationServer3D.map_get_closest_point(get_world_3d().
            get_navigation_map(), target_pos)
        $NavigationAgent3D.set_target_position(clossest_pos)

func _on_move_timer_timeout():
        var sphere_point = get_random_pos_insphere(roamDistance)
        move_to(sphere_point)
```

Ovaj kod omogućava neprijatelju nasumično kretanje po mapi, koristeći funkcije za generiranje pozicija unutar sfere i navigaciju prema tim točkama. Funkcija `get_random_pos_insphere` generira nasumičnu poziciju unutar definiranog radijusa sfere. Prvo se nasumično generiraju

dvije vrijednosti koje se koriste za provjeru jesu li unutar jedinice sfere, a zatim se te vrijednosti koriste za stvaranje vektora koji predstavlja nasumičnu poziciju u 2D prostoru, pri čemu je y-os postavljena na 0. Pozicija se potom skalira prema zadanom radijusu, kako bi neprijatelj mogao lutati unutar ograničenog područja. Funkcija `move_to` koristi ovu generiranu poziciju i postavlja neprijatelja u stanje lutanja (ROAM), a pomoću navigacijskog sustava igre određuje najbližu valjanu točku na mapi, osiguravajući da neprijatelj ne pokušava doći do nepristupačnih područja. Nakon toga, postavljena točka postaje ciljna pozicija neprijatelja. Kada istekne moveTimer, poziva se funkcija `_on_move_timer_timeout`, koja generira novu nasumičnu poziciju i šalje neprijatelja prema toj točki, simulirajući prirodno lutanje neprijatelja po mapi.

5.5.5. Gađanje igrača projektilima

Gađanje neprijatelja projektilima igra važni ulogu u igri. Ako neprijatelj pogodi igrača, njegovo zdravlje (health) će se smanjiti. Nakon što neprijatelj pogodi igrača određeni broj puta, igrač će biti eliminiran iz igre.

Iako je prethodno bilo govora o dijelu konačnog automata koji se odnosi na napad, odnosno ispaljivanje projektila prema igraču, u ovom dijelu bit će detaljnije i preciznije objašnjen taj proces.

```
ATTACK:
        var direction = (detectedPlayer.global_position - $enemyExample/bulletPoint.
            global_position).normalized()
        $enemyExample/Armature.rotation.y = atan2(direction.x, direction.z)
        animation_player.play("Attacking")
        await animation_player.animation_finished

        var nBullet = enemy_prijectile.instantiate()
        get_parent().add_child(nBullet)
        nBullet.global_position = $enemyExample/bulletPoint.global_position

        nBullet.damage = damage
        nBullet.set_linear_velocity(direction * projectile_velocity)
        nBullet.look_at(detectedPlayer.global_position, Vector3.UP, true)

        $DetectPlayerArea/CollisionShape3D.disabled = true
        $DetectPlayerArea/CollisionShape3D.disabled = false
        $moveTimer.start()
```

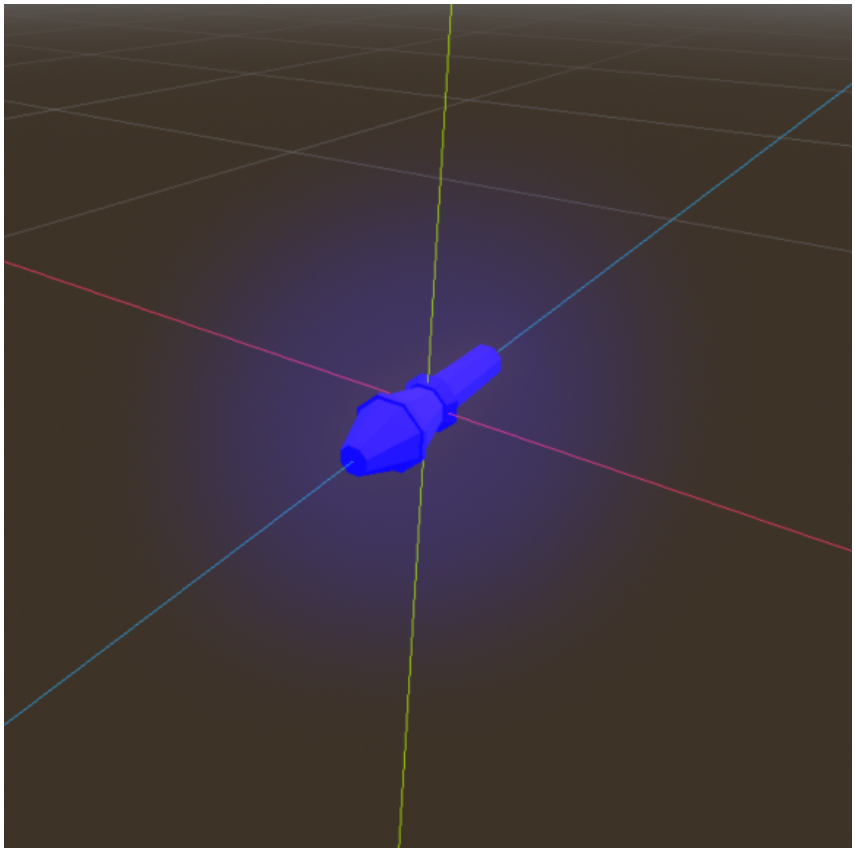
Izračunavanje smjera prema igraču obavlja se tako što se uzima vektor razlike između pozicije igrača i pozicije neprijatelja, pri čemu se koristi `detectedPlayer.global_position` i `bulletPoint.global_position`. Taj vektor se koristi kako bi se dobio jedinični vektor koji točno određuje pravac prema igraču. Nakon toga, neprijatelj se rotira prema igraču pomoću funkcije atan2, koja računa kut između neprijatelja i igrača na temelju njihovih pozicija, osiguravajući da je neprijatelj uvijek usmjeren prema svom cilju. Animacija napada se pokreće putem `animation_player.play("Attacking")`, a kod čeka da animacija završi prije nego što neprijatelj ispali projektil.

Potom se stvara novi projektil iz unaprijed učitanog resursa `enemy_prijectile`. Pro-

jektil se postavlja na poziciju oružja neprijatelja i dodaje u igračevu roditeljsku scenu. Projektilu se dodjeljuje određena vrijednost štete(eng. damage), definirana varijablom damage, te se postavlja njegova brzina koristeći funkciju `set_linear_velocity()`. Ova brzina se temelji na prethodno izračunatom smjeru prema igraču i unaprijed definiranoj vrijednosti brzine projektila (`projectile_velocity`). Projektil se postavlja da gleda prema igraču koristeći funkciju `look_at()` kako bi njegovo ispaljivanje bilo u smjeru igrača. Nakon što je projektil ispaljen, detekcija igrača se privremeno onemogućava i zatim ponovno omogućava kako bi se resetiralo detekcijsko područje. Na kraju se ponovno pokreće timer kako bi neprijatelj mogao nastaviti s napadom ili kretanjem.

5.5.5.1. Neprijateljski projektil

Kako bi neprijatelj uopće mogao eliminirati igrača, bilo je potrebno implementirati sustav neprijateljskih projektila. Ovi projektili nanose štetu igraču kada ga pogode, a nakon sudara ili određenog vremena, automatski nestaju iz igre kako bi se održala čistoća same igre.



Slika 12: Prikaz neprijateljskog projektila

Prikaz koda neprijateljskog projektila:

```
ATTACK:
        var direction = (detectedPlayer.global_position - $enemyExample/bulletPoint.
            global_position).normalized()
        $enemyExample/Armature.rotation.y = atan2(direction.x, direction.z)
        animation_player.play("Attacking")
        await animation_player.animation_finished
```

```
var nBullet = enemy_prijectile.instantiate()
get_parent().add_child(nBullet)
nBullet.global_position = $enemyExample/bulletPoint.global_position

nBullet.damage = damage
nBullet.set_linear_velocity(direction * projectile_velocity)
nBullet.look_at(detectedPlayer.global_position, Vector3.UP, true)

$DetectPlayerArea/CollisionShape3D.disabled = true
$DetectPlayerArea/CollisionShape3D.disabled = false
$moveTimer.start()
```

Ovaj kod upravlja ponašanjem neprijateljskog projektila koji napada igrača. Varijabla damage određuje količinu štete koju projektil nanosi kada pogodi igrača. Funkcija `_on_timer_timeout()` briše projektil iz igre kada istekne timer, osiguravajući da projektil ne ostane na ekranu predugo ako ne pogodi cilj. Funkcija `_on_body_entered(body)` detektira sudar projektila s bilo kojim objektom. Ako projektil pogodi igrača i igrač posjeduje metodu `hit_successful()`, ona se poziva kako bi igraču bila nanesena šteta. Nakon toga, projektil se uklanja iz igre pomoću `queue_free()` kako bi se spriječilo daljnje postojanje nakon sudara.

6. Zakjučak

U ovom završnom radu obrađena je tema modeliranja i implementacije neprijateljskih likova kao konačnih automata na primjeru 3D shootera. Kroz teorijski dio rada, objašnjeni su osnovni pojmovi vezani uz umjetnu inteligenciju, s posebnim naglaskom na koncept konačnih automata, njihovu primjenu u računalnim igrama i povijesni razvoj. Prikazani su ključni matematički modeli te podjele konačnih automata, uključujući prihvatitelje, transduktore te primjere Mealyjevih i Mooreovih automata. Kroz povijesni pregled razvoja umjetne inteligencije u video igrama, prikazni su razni važni trenutci u razvoju umjetne inteligencije u igrama, a samim tim prikazan je razvoj ponašanja negirajući likova.

Konačni automat je jedna od najosnovnijih metoda umjetne inteligencije, koja se može primijeniti u raznim sferama računalnih znanosti, a koja omogućava modeliranje složenih sustava kroz jednostavne i jasno definirane prijelaze između stanja. Ova metoda koristi se za simuliranje raznih ponašanja, omogućavajući jednostavno upravljanje sustavima koji reagiraju na određene ulazne uvjete.

U kreiranoj igri korišteni su samo neki primjeri upotrebe konačnih automata, poput upravljanja ponašanjem neprijateljskih likova. Međutim, to nije jedini način na koji se konačni automati mogu koristiti, njihova primjena može biti dosta šira i složenija. Na primjer, konačni automati mogu upravljati dinamikom cijelog svijeta igre, regulirati interakcije između objekata u igri, pa čak i kreirati složene scenarije u igri.

Praktični dio rada bavi se implementacijom 3D shootera igre, u kojoj su neprijateljski likovi modelirani pomoću konačnih automata. Opisane su funkcionalnosti igre, poput kretanja i borbe, gdje su konačni automati omogućili prirodnije ponašanje neprijatelja. Neprijateljski likovi u igri kreću se, detektiraju prisutnost igrača, napadaju ga projektilima te se vraćaju u stanje čekanja. Osim toga, igraču je omogućeno korištenje oružja, s mehanikama pucanja, prikupljanja dodatnih oružja i metaka za ta ista oružja.

Rezultati rada pokazali su da se primjenom konačnih automata može postići realna simulacija ponašanja neprijatelja, a također osigurat jednostavnost kodiranja, što je važno u razvoju video igara. Time se omogućuje stvaranje interaktivnih likova koji učinkovito reagiraju na promjene u okolini i akcije igrača, i na taj način čine samu igru izazovnijom i zanimljivijom za igranje. Ovaj rad predstavlja osnovni model za razvoj složenijih ponašanja i mehanizama umjetne inteligencije.

Popis literature

[1] *umjetna inteligencija - Hrvatska enciklopedija — enciklopedija.hr*, https://enciklopedija.hr/clanak/umjetna-inteligencija, [Accessed: 10.08.2024.], n.d.

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

[3] *How To Design A Finite State Machine*, https://www.cs.princeton.edu/courses/archive/spr06/cos116/FSM_Tutorial.pdf, [Accessed: 10.08.2024.], n.d.

[4] *Timeline of Computer History*, https://www.computerhistory.org/timeline/1952/169ebbe2 [Accessed 20-08-2024.], 2022.

[5] A. L. Samuel, „Some studies in machine learning using the game of checkers," *IBM Journal of research and development*, sv. 3, br. 3, 1959.

[6] *Deep Blue*, https://stanford.edu/~cpiech/cs221/apps/deepBlue.html, [Accessed: 06.08.2024.]

[7] *Garry Kasparov*, https://en.wikipedia.org/wiki/Garry_Kasparov, [Accessed 20-08-2024.], n.d.

[8] *Creatures (video game series)*, https://en.wikipedia.org/wiki/Creatures_(video_game_series), [Accessed: 06.08.2024.]

[9] *The AI of DOOM (1993)*, https://www.gamedeveloper.com/game-platforms/the-ai-of-doom-1993, [Accessed 20-08-2024.], 3-2022.

[10] P.-I. Evensen, H. Stien i D. H. Bentsen, „Modeling battle drills for computer-generated forces using behavior trees," *Interservice/industry training, simulation, and education conference (I/ITSEC), Orlando, Florida, November 2018*, 2018.

[11] *A Simple Finite State Acceptor*, http://osr600doc.sco.com/en/SDK_c++/_A_Simple_Finite_State_Acceptor.html, [Accessed 05-09-2024.], 2.06.2005.

[12] R. M. Keller, *Computer science: Abstraction to implementation*. Harvey Mudd College, 2001.

[13] *StateWORKS: Moore or Mealy model?* http://www.stateworks.com/technology/TN10-Moore-Or-Mealy-Model/, [Accessed: 15.08.2024.], n.d.

[14] *Finite State Machine*, https://medium.com/smartive/what-state-machines-are-and-why-we-use-them-5ea55183be09, [Accessed: 15.08.2024.], n.d.

[15] *Finite-state machine*, https://en.wikipedia.org/wiki/Finite-state_machine/, [Accessed: 15.08.2024.], n.d.

[16] E. Udofia, *Are finite state machines defined by 6-tuple?* https://eitca.org/cybersecurity/eitc-is-cctf-computational-complexity-theory-fundamentals/finite-state-machines/examples-of-finite-state-machines/are-finite-state-machines-defined-by-6-tuple/, [Accessed: 15.08.2024.], n.d.

[17] *Artificial Intelligence 1: Finite State Machines*, https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/previousinformation/artificialintelligen [Accessed: 05.09.2024.], 2024.

[18] *The Pac-Man Dossier Holenet.info*, https://pacman.holenet.info/#Chapter_4, [Accessed 05-09-2024.]

Popis slika

Popis tablica