Izrada 3D natjecateljske FPS igre u Godotu

Duka, Marko

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

Sveučilišni studij Računarstva, smjer Programsko Inženjerstvo

Izrada 3D FPS natjecateljske igre u Godotu

Završni rad

Marko Duka

Osijek, 2023.

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

Obrazac Z1P - Obrazac za ocjenu završnog rada na preddiplomskom sveučilišnom studiju

Osijek, 12.09.2023.

Odboru za završne i diplomske ispite

Prijedlog ocjene završnog rada na preddiplomskom sveučilišnom studiju

Ime i prezime Pristupnika:	Marko Duka
Studij, smjer:	Programsko inženjerstvo
Mat. br. Pristupnika, godina upisa:	R4484, 27.07.2020.
OIB Pristupnika:	97437242531
Mentor:	izv. prof. dr. sc. Časlav Livada
Sumentor:	,
Sumentor iz tvrtke:	
Naslov završnog rada:	Izrada 3D natjecateljske FPS igre u Godotu
Znanstvena grana rada:	**Obradba informacija (zn. polje računarstvo)**
Zadatak završnog rad:	U radu je potrebno izraditi 3D FPS igru koristeći Godot Game Engine. Igra treba sadržavati podršku za više igrača, ali i za natjecanje s računalom kontroliranim likovima uz podršku osnovne umjetne inteligencije. Rad rezerviran za: Marko Duka
Prijedlog ocjene završnog rada:	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 2 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 2 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	12.09.2023.
Datum potvrde ocjene od strane Odbora:	

Potvrda mentora o predaji konačne verzije rada:	*Mentor elektronički potpisao predaju konačne verzije.*
	Datum:

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

IZJAVA O ORIGINALNOSTI RADA

Osijek, 24.09.2023.

Ime i prezime studenta:	Marko Duka
Studij:	Programsko inženjerstvo
Mat. br. studenta, godina upisa:	R4484, 27.07.2020.
Turnitin podudaranje [%]:	4

Ovom izjavom izjavljujem da je rad pod nazivom**: Izrada 3D natjecateljske FPS igre u Godotu**

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

i sumentora ,

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

Potpis studenta:

SADRŽAJ

1. UVOD

Tema je završnog rada izrada 3D online FPS igre koristeći Godot razvojno okružje uz implementaciju neprijatelja koristeći strojno učenje. Zadana je tip igre 1 na 1 arena, borba dvaju igrača.

Problemi ovog tipa igre nalaze se u stvaranju okružja koje podržava uspješno umrežavanje i interakciju igrača. Umrežavanje je bitan zadatak rada riješen korištenjem *Godot Multiplayer API*, dok interakcija igrača riješena je korištenjem RPC poziva.

Budući Godot ne podržava strojno učenje, javljaju se problemi implementacije strojnog učenja. Strojno učenje implementirano je korištenjem jezika Python i lokalnog UDP servera kako bi igra i Python skripte mogle komunicirati.

Cilj rada prikazati je izradu igre navedenog tipa, objasniti proces stvaranja svih ključnih objekata igre i rješenja navedenih problema.

Poglavlja redom analiziraju sljedeće sadržaje i probleme:

1. poglavlje sadržaja analizira i prikazuje igre sličnih žanrova
2. poglavlje prikazuje i analizira Godot razvojno okružje s problemima stvaranja igrača, neprijatelja i menadžera igre
3. poglavlje rješava probleme umrežavanja i komunikacije sa eksternim programima potrebnim za strojno učenje
4. poglavlje objašnjava probleme strojnog učenja te prikazuje dva konkretna primjera modela koja uspoređuje sa skriptiranim ponašanjem neprijatelja.

1.1. Zadatak završnog rada

Zadaci rada su opisati i izraditi igru žanra 1 na 1 arena FPS u Godot razvojnom okružju i prikazati problem, rješenje i implementaciju neprijatelja pomoću strojnog učenja.

2. ANALIZA ZNAČAJNIH IGARA

Kako bi se lakše prepoznali problemi projekta, potrebno je analizirati druge projekte sličnog tipa.

2.1. Wolfenstein 3D

Wolfenstein 3D je jedna od prvih „3D“ FPS igra izdana 1992. godine. Za prikaz okružja ne koristi 3D modele nego slike prikazano slikom 2.1.

Sl.2.1. Wolfestein 3D [15].

Budući da igra ne koristi 3D modele, a prikazuje 3D svijet, svaki piksel ekrana je zasebno izračunat. Prikazivanje igre na ekranu realizirano je računanjem udaljenosti i dodjeljivanju boje svakog piksela [1]. Pikseli su se nalazili u u nizu. Nakon kalkulacija piksela, niz je konvertiran u oblik slike koja se prikazuje na ekran igrača.

Upravljanje igrača je odrađeno „tenkovskim upravljanjem“ (engl. *Tank Controls*). Takav tip upravljanja osmišljen je za igrače palice (engl. *Joystick*) koji ograničava igračevu sposobnost gledanja uokolo. Igrač se mogao okretati isključivo lijevo i desno. Hodati je mogao u dva smjera, naprijed i nazad, dok skakanje nije bilo omogućeno. Kretanje će u radu djelomično implementirati takav tip kretanja igrača restrikcijom skakanja.

Zadaća je neprijatelja u igri ubijanje igrača. Neprijateljevo ponašanje opisuje se dvjema funkcijama – hodanjem i pucanjem. Neprijatelji nisu mogli izvoditi obje funkcije istovremeno te su ovisno o igračevoj poziciji i mogućnosti pogotka izvodili samo jednu funkciju. Za razliku od Wolfenstein 3D, neprijatelji u radu imat će sposobnost izvođenja obje funkcije istovremeno.

2.2. Quake 3 Arena

Quake 3 Arena jedna je od najpoznatijih Online 3D Arena FPS igra. Igru su razvili ID Software i izdali 1999. godine. Za razliku od Wolfenstein 3D, igra koristi 3D modele sa slikama tekstura prikazano slikom 2.2.

Sl.2.2. Quake [16].

Kretanje u igri koristi današnji standard upravljanja igrača. Igrač može slobodno gledati, kretati se i skakati bez restrikcija. Neprijatelji se također kreću slobodno te mogu izvoditi funkcije hodanja i pucanja istovremeno. Nadalje, neprijatelj, za razliku od igrača, ne koristi stvarne upravljačke jedinice poput miša i tipkovnice, nego slično digitalno sučelje [2]. Sučelje neprijatelja u radu realizirat će se pomoću „kosturske“ konstrukcije, kojom će se neprijateljevo ponašanje imitirati igračevo i ovisiti o drugim ulazima.

Igračevo korisničko sučelje prikazuje minimalan broj informacija potreban igraču. Njegove trenutne živote i preostali broj metaka. Korisničko sučelje igrača u radu također će prikazivati iste informacije.

Quake 3 Arena namijenjena za veći broj igrača, iz tog razloga okružja su kompleksna [3]. Radom prikazani projekt predstavlja format igre 1 na 1 te za razliku od Quake 3 Arene, ne zahtijeva visoku kompleksnost okružja.

3. RAZVOJNO OKRUŽJE I KLJUČNI OBJEKTI

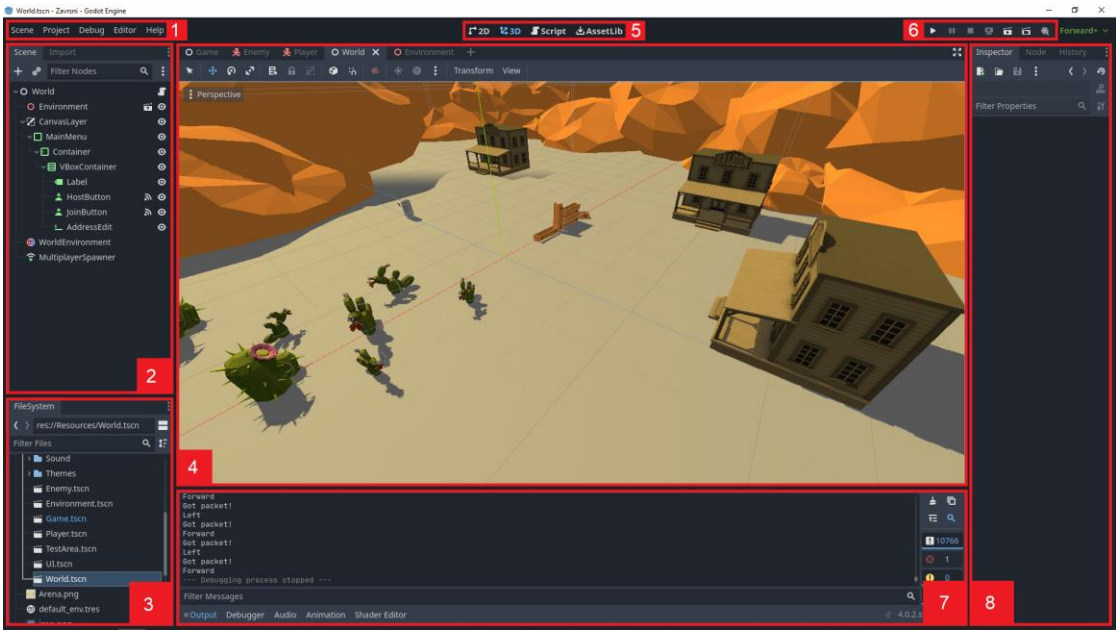
3.1. Razvojno okružje Godot

Godot je besplatno razvojno okružje namijenjeno za izradu 2D i 3D videoigara otvorenog koda, originalno izdano 2014. godine. Autori su alata Juan Linietsky i Ariel Manzur koji su objavili okružje pod MIT licencom [18].

3.1.1. Uređivač

Uređivače (engl. *Editor*) je na osam dijelova, prikazano na slici 3.1.:

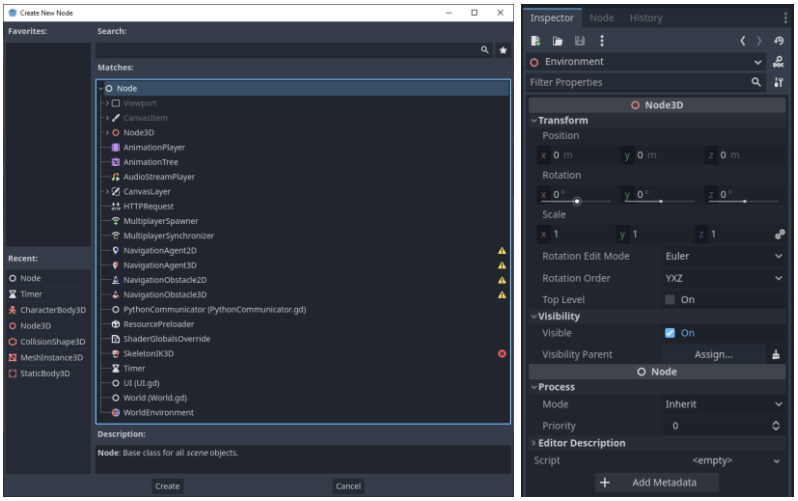
1. dio sučelja koristi se za upravljanje projektom
2. dio prikazuje trenutnu scenu
3. dio omogućuje upravljanje datotekama
4. dio koristi se za vizualni prikaz trenutne scene ili uređivanje koda
5. dio omogućuje promjenu perspektive gledanja svijeta ili prebacivanje između prikaza scene i uređivanja koda
6. dio koristi se za pokretanje igre
7. dio predstavlja konzolu
8. dio prikazuje značajke i grupu označenog čvora.



Sl.3.1. Grafičko sučelje Godota

3.1.2. Struktura objekata

Za razliku od drugih okružja, Godot objektima pristupa strukturom stabla. Svaki je element u stablu samostalan objekt. Svaki objekt spremljen je u obliku scene sa ekstenzijom“.tscn“. Kako postoji veliki broj čvorova, ne pregledavaju se svi, nego se fokusira na najkorištenije čvorove. Za ovaj zadatak najbitniji tipovi čvorova su prostorni čvorovi (engl. *Spatial Nodes*) koji se nalaze u obitelji 3D čvorova (engl. *Node3D*), obični čvorovi (engl. *Nodes*) i posebni čvorovi za umrežavanje (engl. *Multiplayer Nodes*).



Sl. 3.2. Prikaz različitih tipova čvorova i njihovih značajki

Primjer značajki 3D čvora prikazan je na desnoj strani slike 3.2. Prikazani čvor korijen je objekta svijeta. Svijet je 3D objekt, a njegove najbitnije značajke su pozicija i rotacija koje se nalaze unutar *Transform* sekcije. Sljedeća je bitna značajka objekta priložena skripta. Objekt svijeta ne koristi skriptu što je vidljivo unutar *Script* sekcije koja se nalazi pri dnu slike.

3.1.3. Osnovne funkcije okružja

Dvije osnovne funkcije okružja su *_ready()* i _process(). Funkcija _ready() pozvana je točno jedan put nakon stvaranja objekta, dok se funkcija *_process()* poziva svaku sličicu u sekundi.

3.2. Ključni objekti

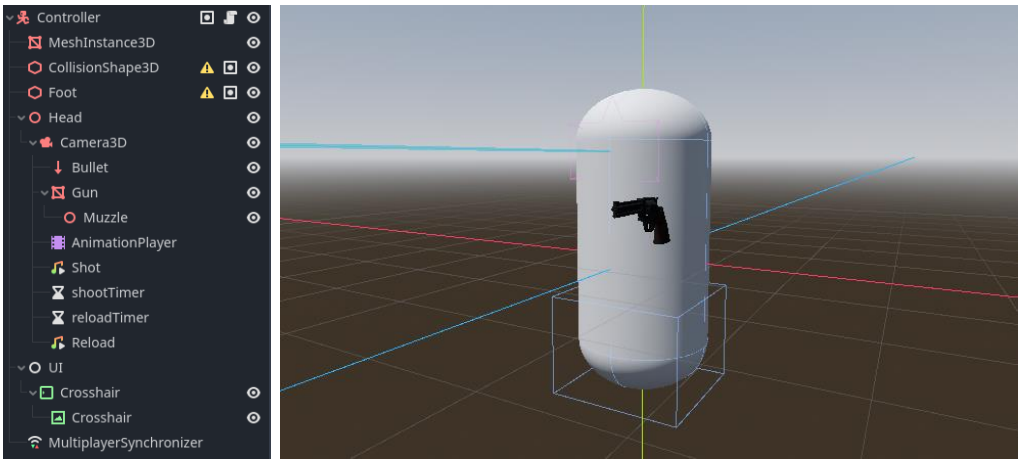
Na početku projekta bitno je definirati ključne objekte i mehanike potrebne za prvobitnu implementaciju. U ovom projektu najbitniji objekti su igrač, neprijatelj i menadžer igre.

3.2.1. Igrač

Objekt igre je igrač. Njegova glavna zadaća je interakcija s okolinom u kojoj se nalazi. Problemi koje je potrebno riješiti za izradu igrača kretanje su i pucanje.

3.2.2. Konstrukcija igrača

Godot tretira objekte kao stabla čvorova te je potrebno odrediti korijen stabla igrača. Na lijevoj strani slike 3.3. prikazana je konstrukcija objekta igrača, dok je na desnoj izgled igrača. Korijen je igrača čvor tipa *CharacterBody3D*. Potrebno mu je dodati model igrača. Model igrača stvoren je čvorom *MeshInstance3D* koji igraču daje oblik kapsule. Nije potrebno rotirati cijeli objekt na svim osima pri gledanju uokolo, dovoljno je za to napraviti čvor glave..



Sl. 3.3. Konstrukcija igrača

Čvor glave tip je *Node3D* kako bi se glava mogla rotirati zajedno sa svojim pod-čvorovima. Također, kamera iz koje igrač gleda, tip *Camera3D*, nalazi se unutar glave.

Nakon stvaranja glave igrača, preostaje riješiti problem sudaranja igrača s drugim objektima. Taj se problem rješava dodavanjem zonama pogotka koje su nevidljivi 3D modeli koji se koriste za fizičku interakciju s okružjem (engl. *Hitboxes*). Ti modeli dodani su kao čvorovi tipa *CollisionShape3D*, gdje je jedan oblika kapsule, kao i igrač, dok je drugi kutija koja reprezentira igračeve noge.

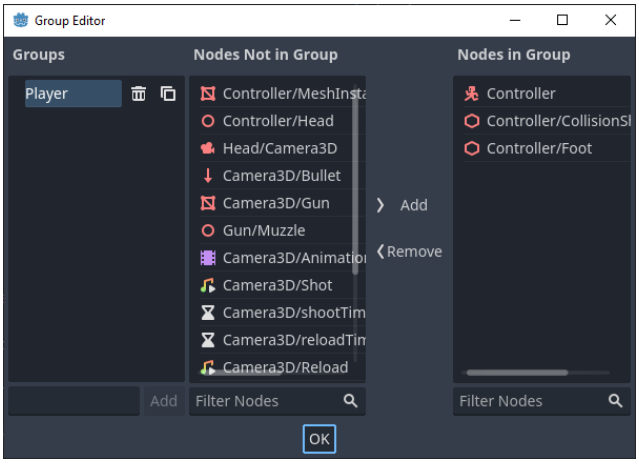
Unutar glave, kako bi igrač mogao pucati, stvoreni su čvorovi koji predstavljaju pištolj, njegov vrh i virtualna zraka koja provjerava u koji objekt igrač gleda. Animacija pucanja i proizvedenih zvukova igrača rješava se dodavanjem posebnih čvorova za te zadaće.

Virtualna zraka (engl. *RayCast*) termin je koji se koristi u razvoju videoigara kako bi se detektirala kolizija ili sudar između objekata. Virtualna zraka emitira se iz određene točke u igri kako bi se provjerilo susreće li se s drugim objektima. Ova se tehnika često koristi za detekciju sudara, interakciju s objektima i druge funkcionalnosti u igrama.

Za svaki zvuk koristi se zasebni čvor te su iz tog razloga stvorena dva čvora *AudioStreamPlayer*. Jedan čvor pušta zvukove pucnja pištolja, a drugi punjenja pištolja metcima. Unutar glave igrača nalazi se *AnimationPlayer* čvor na koji se vezuju navedeni *AudioStreamPlayer* zvukovi.

Projekt zahtijeva umreženje igrača, stoga je potrebno dodati čvor čija je zadaća sinkronizacija igrača na serveru. Čvor *MultiplayerSynchronizer* rješava taj problem. Pomoću njega sinkronizirani su igračeva pozicija, rotacija, ukupan život i trenutni broj metaka.

Grupiranje objekata bitna je značajka čvorova. Kako bi igra imala mogućnost prepoznavanja igrača, dodaju se grupe. Grupe se objekata označavaju unutar značajki čvorova pod prozorom čvora. Stvaranje i upravljanje grupa izvršeno je unutar prozora za uređivanje grupa (engl. *Group Editor*). Slika 3.4. prikazuje stavljanje čvorova igrača unutar grupe igrač (engl. *Player group*)



Sl. 3.4. Postavljanje igrača u grupu

3.2.3. Kretanje igrača

Stvaranjem objekata zona pogotka riješen je problem sudaranja s preprekama, ali i dalje ostaje problem kretanja igrača. Kretanje igrača rješava se mijenjanjem igračeve pozicije unutar funkcije *_process()*. Budući da igrač ima fizičke interakcije sa svijetom, potrebno je koristiti izvedenu funkciju *_physics_process()* koja ima jednaku zadaću kao i *_process()*, ali se obazire na fiziku

igre. Na kodu 3.1. prikazan je relevantan kod korišten za kretanje igrača. Svakom sličicom briše se vrijednost smjera kretanja unutar linije 112, te ovisno o pritisnutoj tipki vektor smjera kretanja zbraja se s bazičnim vektorom pritisnutog smjera.

Linija ***Kod***

```
8:      var speed = 17.5

97:     func _physics_process(_delta):
112         direction = Vector3()
114:        if Input.is_action_pressed("moveForward"):
115:            direction -= transform.basis.z
117:        elif Input.is_action_pressed("moveBack"):
118:            direction += transform.basis.z
121:        if Input.is_action_pressed("moveLeft"):
122:            direction -= transform.basis.x
124:        elif Input.is_action_pressed("moveRight"):
125:            direction += transform.basis.x
128:        direction = direction.normalized
130:        set_velocity(direction * speed)
131:        set_up_direction(Vector3.UP)
132:        move_and_slide()
```

Kod. 3.1. Controller.gd

Nakon određivanja smjera kretanja, za pokretanje igrača, potrebno je normalizirati vektor linijom 128 te postaviti igračevu brzinu linijom 130, to se postiže množenjem normaliziranog vektora smjera i konstante koja predstavlja igračevu brzinu unutar linije 8. Kretanje omogućuje postavljanje orijentacije vektora okomitog na igrača unutar linije 113, te korištenje metode za kretanje igrača unutar linije 132.

3.2.4. Igračevo pucanje

U ovom je trenutku igrač grupiran i ima sposobnost kretanja. Nadalje, potrebno mu je implementirati drugu ključnu mehaniku, pucanje. Pomoću virtualne zrake igrač provjerava u što puca. U slučaju pogađanja igrača, potrebno je smanjiti ukupan broj života pogođenog igrača. Kod 3.2. prikazuje relevantne linije koda unutar kojeg linije 13 i 14 predstavljaju konstante ukupnog života i broja metaka u pištolju.

Funkcija pucanja, s početnom linijom 67, poziva se unutar *_physics_process()* funkcije pri pritisku tipke pucanja što se nalazi na liniji 104. Kako nije dopušteno konstantno pucanje, implementirano odbrojavanje vremena u kojem igrač ne smije pucati. Isto grananje koje provjerava status tipke pucanja, provjerava odbrojavanje vremena nedozvoljenog pucanja i punjenja pištolja metcima.

Ako su svi uvjeti grananja zadovoljeni, igrač će pogoditi metu u koju gleda i počet će se odbrojavati vrijeme do sljedećeg mogućeg pucanja.

Linija ***Kod***

```
13:     var damage = 25
16      var health = 100

60:     func reload():
61:         if(!animationPlayer.is_playing()):
62:             animationPlayer.play("Reload")
64:         ammo = 6

135:    func receiveDamage():
136:        health -= damage
137:            if health <= 0:
138:                health = 100
139:                position = Vector3.ZERO

67:     func shoot():
68:         if ammo > 0:
69:             ammo = ammo - 1
71:             if !animationPlayer.is_playing():
72:                 animationPlayer.play("Kick")
74:                 shootSound.play()
77:             if bulletRaycast.is_colliding():
78:                 var bullet = get_world_3d().direct_space_state
79:                 var query = PhysicsRayQueryParameters3D
                                .create(muzzle.transform.origin,
                                 bulletRaycast.get_collision_point())
82:                 if collision:
84:                     if(target.is_in_group("Player") && target != self):
86:                         var targetPlayer = target
87:                         targetPlayer.receiveDamage()
88:                 elif(target.is_in_group("Enemy")):
90:                     var targetEnemy = target
91:                     targetEnemy.receiveDamage()
93:         else:
94:             reload()
```

Kod 3.2. Controller.gd

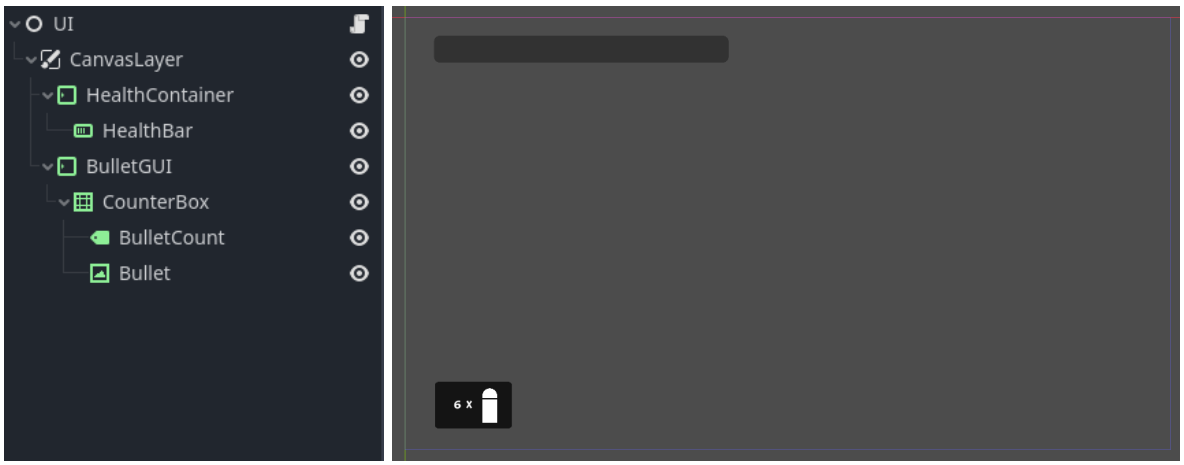
Prilikom pucanja, prva značajka koju skripta provjerava broj je metaka. Ako je broj metaka 0 (ili u teoriji moguće i manje), potrebno je napuniti pištolj metcima, postupak čega je prikazan grananjem linija 68 i 93, dok je punjenje pištolja prikazano pozivanjem funkcije za punjenje pištolja unutar linije 94.

Funkcija punjenja pištolja s početkom unutar linije 60 pušta animaciju pomoću linija 61 i 62, dok unutar linije 64 postavlja ukupan broj metaka na maksimum od 6. U slučaju da igrač ima dovoljno metaka za pucanje, prvo se pušta animacija pucanja unutar linija 71 i 72, nakon čega se izvodi provjera objekta u koji igrač u tom trenutku gleda pomoću linije 77. Nakon dohvaćanja objekta u koji igrač gleda pomoću linije 78, stvorena je nova virtualna zraka koja spaja vrh pištolja i pogođenu točku te ima ulogu metka jer očitava pogođeni objekt. Tek nakon stvaranja metka, potrebno je provjeriti upucani objekt. Linija 83 uzima referencu na pogođeni objekt, nakon čega se vrši provjera grupe pogođenog objekta. Ako je pogođeni objekt grupiran kao igrač ili neprijatelj, prima štetu pomoću funkcije definirane linijom 135.

3.2.5. Korisničko sučelje

Cilj igračevog korisničkog sučelja prikaz je ključnih informacija o igraču koje čine broj njegovih života i broj raspoloživih metaka. Problem izvedbe korisničkog sučelja broj je objekata igrača. Pri umreženom igranju prisutna su točno dva igrača. Ako se korisničko sučelje nalazi unutar igrača, oba će korisnička sučelja biti iscrtana jedno preko drugog.

Način na koji se ovaj problem rješava odvajanje je korisničkog sučelja od igrača te instanciranje jednog korisničkog sučelja koje se spaja na lokalnog igrača. Konstrukcija i izgled korisničkog sučelja prikazani su slikom 3.5. Svi korišteni čvorovi, osim korijena, pripadaju obitelji čvorova za korisničko sučelje (engl. *UI Nodes*), dok je korijen prazan čvor. Stanje trenutnog života igrača prikazano je pomoću trake za napredak (engl. *Progress bar*) koja se nalazi unutar kutije s



Sl. 3.5. Korisničko sučelje igrača

marginama (engl. *Margin container*). Broj raspoloživih metaka prikazan je oznakom (engl. *Label*) smještenom unutar kutije koja sadrži sliku metka.

Korisničko sučelje sadržava funkcije za prikaz broj metaka i trenutne živote prikazane kodom 3.3. Funkcija linije 12 mijenja traku napretka, dok funkcija linije 15 mijenja broj preostalih metaka.

Linija ***Kod***

```
12:     func updateHealthBar(healthValue):
13:         healthBar.value = healthValue

15:     func updateAmmoCount(ammoValue):
16:         bulletCount.text = str(ammoValue) + " x"
```

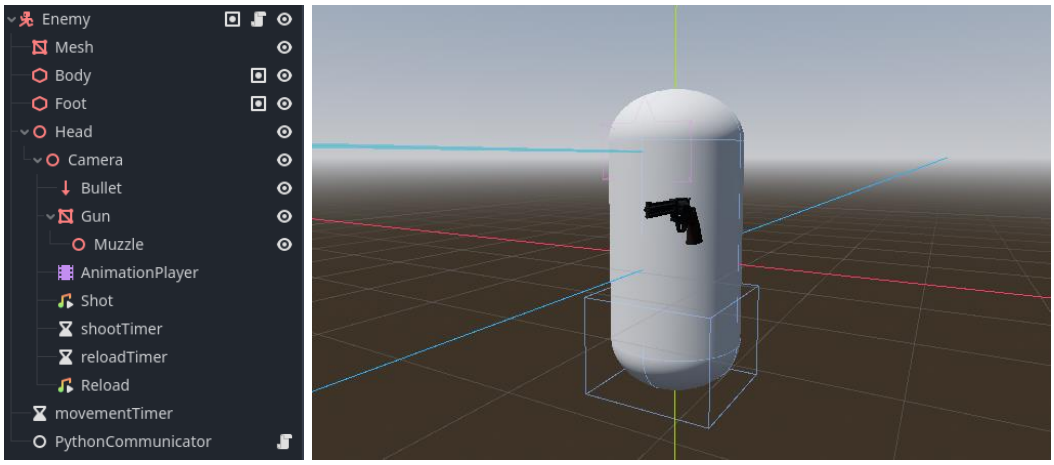
Kod. 3.3. UI.gd

3.2.6. Neprijatelj

Glavna je zadaća neprijatelja oponašati igrača. Cilj neprijatelja napraviti je kostura spremnog za strojno učenje.

3.2.7. Konstrukcija neprijatelja

Konstrukcija vidljiva na slici 3.6. kopira konstrukciju igrača. Na isti su način postavljeni fizički temelji objekta, uključujući odbrojavanje vremena za pucanje. Glavna je razlika manjak stvarne kamere i dodatak brojača vremena za hodanje. Brojač vremena za hodanje dodana je funkcija kako bi budući korisnici igre imali vremena razmisliti o sljedećem potezu, jer je odlučivanje i reagiranje u sklopu od jedne sličice po sekundi ljudskim osjetilima neizvedivo.



Sl. 3.6. Konstrukcija neprijatelja

3.2.8. Kretanje neprijatelja

Kretanje neprijatelja realizira se kao kretanje igrača. Bitna razlika u ovom mjestu koda jest ta da umjesto čitanja korisnikovih ulaza, funkcija kretanja prihvaća riječ smjera kao argument. Ovisno o smjeru kretanja izračunavaju se neprijateljevi vektori smjera kretanja. Prikaz kostura nalazi se na kodu 3.4. Prikazani je kod funkcija s početkom u liniji 64 koja prikazuje implementaciju koncepta kostura. Ako je uvjet brojača vremena zadovoljen unutar grananja linije 66, ovisno o željenom smjeru kretanja uz pomoć grananja određuje se vektor smjera kretanja.

Linija ***Kod***

```
64:     func move(direction : String):
66:         if(MovementTimer.is_stopped()):
67:             if(direction == "Backward"):
68:                 movementDirection -= transform.basis.z
69:             elif(direction == "Right")
70:                 movementDirection -= transform.basis.x
71:             elif(direction == "Left"):
72:                 movementDirection += transform.basis.x
73:             elif(direction == "Forward")
74:                 movementDirection += transform.basis.x
77:             elif(direction == "Stop")
79:                 movementDirection = Vector3.ZERO
80:         MovementTimer.start()
```

Kod 3.4. Enemy.gd

3.2.9. Neprijateljevo pucanje

Prije rješavanja problema pucanja potrebno je riješiti problem usmjeravanja pogleda igrača. Svijet igre dizajniran je tako da su svi igrači međusobno vidljivi. Iako su igrači vidljivi, mogu se sakrivati iza prepreka. Nakon toga rješava se implementacija gledanja igrača. Kod 3.5. prikazuje funkciju na liniji 64 pomoću koje neprijatelj gleda u igrača. Iako neprijatelj gleda u smjeru igrača, još uvijek ne znači da ga može u potpunosti vidjeti i pogoditi. Stoga se dodaje funkcija linije 68 koja na način na koji igrač provjerava što puca, provjerava može li neprijatelj pogoditi igrača.

Linija Kod

```
64:     func rotateHead():
66:         Head.look_at(player.position)
67:
68:     func doesSeePlayer():
69:         var bullet = get_world_3d().direct_space_state
            var query = PhysicsRayQueryParameters3D
                                .create(muzzle.transform.origin,
70:                                 bulletRaycast.get_collision_point())
            var collision = bullet.intersect_ray(query)
71:         if collision:
72:             if(collision.collider.is_in_group("Player")):
73:                 return true
74:         else:
77:             return false
```

Kod 3.5.. Enemy.gd

Kada je implementirano gledanje, implementacija pucanja je značajno se pojednostavljuje. Jedina je razlika pozivanje funkcije pucanja i gađanje igrača. Na kodu 3.6. prikazana je implementacija pucanja. Prije pozivanja funkcije pucanja neprijatelj provjerava može li pogoditi igrača na liniji 127. Pri pucanju, nakon što neprijatelj uočava igrača, a kako se igra ne bi otežala, dodan je nasumičan broj koji određuje uspješnost neprijateljevog pogotka igrača. Nasumičan broj se generira unutar linije 62, provjerava linijom 63, te ako je zadovoljen uvjet uspješno pogađa igrač.

Linija Kod

```
50:     func shoot():
51:         if ammo > 0:
52:             shootTimer.start()
57:             ammo -= 1
62:             var rng = RandomNumberGenerator.new()
63:             if rng.randf_range(0, 1) > 0.6:
64:                 player.receiveDamage()
65:         else:
66:             reloadTimer.start()
69:              ammo = 6


116:    func _process(_delta):
127:        if(doesSeePlayer() and reloadTimer.is_stopped()
                               and shootTimer.is_stopped()):
128:            shoot()
```

Kod. 3.6. Enemy.gd

3.2.10. Menadžer igre

Problem spajanja objekata igre zahtijeva implementaciju *singleton* objekta čija je dužnost menadžment i spajanje drugih objekata igre. Menadžer igre (engl. *Game Manager*) implementira se pomoću skripte prikazane kodom 3.7. Dužnost menadžera u ovoj fazi spremanje je referenci lokalnih objekata.

Linija ***Kod***

```
1:      extends Node
2:      var player : Player
3:      var enemy : Enemy
4:      var ui : UI
```

Kod. 3.7. GameManager.gd

Spremanje referenci implementirano je pomoću funkcije *_ready()*. Pri instanciranju objekata neprijatelja, igrača i korisničkog sučelja, vanjski objekti sami spremaju svoju referencu u menadžera. Primjer spremanja referenci prikazuje kod 3.8. Objekt neprijatelja pri instanciranju, linijom 114 sprema referencu samog sebe unutar menadžera.

Linija ***Kod***

```
1:      func _ready():
2:          GameManager.enemy = self
```

Kod 3.8.. Enemy.gd

4. SERVERI I KOMUNIKACIJA

4.1. UDP server

UDP server je vrsta servera koji koristi UDP protokol za komunikaciju sa klijentima. UDP (*engl. User Datagram Protocol*) je komunikacijski protokol koji pruža procedure programima za slanje poruka drugim programima sa minimalnim protokolnim mehanizmom [4]. Za razliku od TCP (engl. *Transmission Control Protocol*), UDP ne provjerava uspješnost slanja paketa što ga čini nesigurnim ali bržim [5]. Iz tog razloga najčešća primjena UDP-a su streaming servisi.

Za uspostavu servera potrebno je odrediti nekoliko parametara:

- Adresu servera
    - IP adresa
    - PORT
- Buffer size

Adresu servera čine IP adresa i port. IP adresa ovisi o lokaciji servera, a ako se server nalazi na lokalnoj mreži, zadana adresa je „127.0.0.1“. Port predstavlja logičku spojnicu na mreži koja služi kao krajnja točka za komunikaciju. Buffer size označava duljinu poruka koje se šalju mrežom.

4.1.1. Određivanje porta

Ukupan broj mogućih portova UDP servera je:

$$2^{16} - 1 = 65535$$

Taj broj čine rezervirani, zauzeti i slobodni portovi. Problem određivanja porta najviše čine rezervirani portovi. Popis svih rezerviranih portova nalazi se u izvoru [6].

4.2. Game server

4.2.1. ENet

Enet (engl. *Extremely Reliable Networking*) otvorena je biblioteka namijenjena izradi servera za online igre. Temelji se na UDP protokolu i pruža jednostavan sloj za pouzdanu mrežnu komunikaciju [7]. Prednost korištenja UDP-a manja je latencija koja je bitna značajka servera online igara.

Godot koristi ENet biblioteku koja je jednostavna i efikasna za implementaciju te je zbog toga odabrana kao optimalno rješenje problema umreženja.

Linija ***Kod***

```
10:     const Player = preload("res://Resources/Player.tscn")
11:     const PORT = 5555
12:     var enetPeer = ENetMultiplayerPeer.new()
```

Kod. 4.1. World.gd

Kod 4.1. prikazuje skriptu u kojoj se nalazi server. U liniji 10 se nalazi referenca na objekt igrača, dok se u liniji 11 specificira port servera. U liniji 12 kreira se instanca objekta *ENetMultiplayerPeer()*, koji ima zadaću stvaranja servera ili povezivanja na server kao klijent.

Linija ***Kod***

```
14:     func _on_HostButton_pressed():
17:         enetPeer.create_server(PORT)
18:         multiplayer.multiplayer_peer = enetPeer
20:         multiplayer.peer_connected.connect(addPlayer)
21:         multiplayer.peer_disconnected.connect(removePlayer)
23:         addPlayer(multiplayer.get_unique_id())
```

Kod. 4.2. World.gd

Funkcija prikazana kodom 4.2. u liniji 17 kreira server na specificiranom portu. Linije 18 do 20 koriste Godotov multiplayer API kako bi postavile odgovarajuće vrijednosti, ovisno o tome je li igrač spojen ili odspojen s tog servera. Nakon toga, linijom 23 poziva se funkcija *addPlayer()* koja instancira objekt igrača za svaku osobu koja je spojena na server.

Linija ***Kod***

```
25:     func _on_JoinButton_pressed():
28:         enetPeer.create_client("localhost", PORT)
29:         multiplayer.multiplayer_peer = enetPeer
```

Kod. 4.3. World.gd

Prilikom spajanja na server, poziva se funkcija prikazana kodom 4.3. Na liniji 28 kreira se klijent koji se povezuje na specificiranu IP adresu i port. Nakon toga, linija 29 sprema *ENetPeer* objekt koji omogućuje klijentu komunikaciju sa serverom.

.

4.2.2. RPC

RPC (engl. *Remote Procedure Calls*) je mrežni komunikacijski protokol koji je korišten kao temelj konstrukcije server-klijent aplikacija. Ideja RPC-a je uspješno slanje informacija klijenta serveru koji će ih obraditi i poslati poruku klijentu.

Projekt primjenjuje RPC protokol kako bi se omogućila interakcije između igrača na serveru. Kod 4.4. prikazuje relevantan kod implementacije protokola. Igračeva funkcija primanja štete je postavljana na RPC tip funkcije koju može pozvati bilo koji klijent, što je prikazano linijom 134. Pri pozivu funkcije primanja štete, potrebno je definirati ID igrača koji prima štetu što realizirano linijom 87. Nakon implementacije koda umreženi igrači mogu biti uspješno pogođeni i primiti štetu.

Linija ***Kod***

```
134:    @rpc("any_peer")
135:    func receiveDamage():
136:        health -= damage
137:        if health <= 0:
138:            health = 100
139:            position = Vector3.ZERO

87:     targetPlayer.receiveDamage.rpc_id(
                                  targetPlayer.get_multiplayer_authority())
```

Kod. 4.4. Controller.gd

4.3. Komunikacija s Python skriptama

4.3.1. Problem s Godotom

Godot, za razliku od Unity-a, ne podržava strojno učenje što čini veliku prepreku za zadani problem. Jedan od načina implementacije strojnog učenja u Godot je korištenje eksternih skripti i programa, poput Pythona. Python je interpretirani i objektno-orijentirani jezik koji se često koristi za strojno učenje. Zbog svoje fleksibilnosti može poslužiti kao most za rješavanje trenutnog problema.

4.3.2. Pokretanje Python skripti unutar Godota

Prilikom pokretanja skripti koristi se Godotova klasu OS. OS klasa zadužena je za funkcije operacijskog sustava poput pokretanja i zaustavljanja procesa. Pokretanje procesa odrađuje funkcija *OS.execute()*, dok zaustavljanje procesa vrši funkcija *OS.kill()*. Za ispravno pokretanje procesa, potrebo je funkciji predati putanju procesa koji pokreće i njegove argumente, tj. za pokretanje Python skripti potrebno je predati putanju interpretera i skripte koja se pokreće.

Linija ***Kod***

```
11:     var DIR = OS.get_executable_path().get_base_dir()
12:     var interpreterPath =  3
            DIR.path_join("Resources/Python/env/Scripts/python.exe")
13:     var scriptPath =  DIR.path_join("Resources/Python/PathDrawer.py")
        …
23:     func drawPath():
24:         var output = []
25:         var PID = OS.execute(interpreterPath,
                                 [scriptPath], output, false, false)
26:         processPIDs.append(float(PID))
```

Kod 4.5. GameManager.gd

Kod 4.5. prikazuje linija 11 koja dohvaća putanju direktorija projekta. Iako joj je moguće predati apsolutnu putanju, pri kompilaciji projekta putanja će se promijeniti te je istu iz sigurnosnih razloga potrebno implicitno odrediti.

Linije 12 i 13 određuju putanje interpretera i skripte koje se nalaze unutar projektnih podataka. Linija 25 pokreće skriptu, funkcija *execute()* vraća ID procesa koji se u liniji 26 dopiše u listu svih

pokrenutih procesa. Pomoću te liste, pri gašenju igre, poziva se funkcija *killProcesses()* koja je prikazana kodom 4.6.

Linija ***Kod***

```
29:     func killProcesses():
30:         for pid in processPIDs:
31:             OS.kill(pid)
```

Kod. 4.6. GameManager.gd

4.3.3. Implementacija UDP servera i klijenta

Nakon pokretana skripti i dalje postoji problem komunikacije s Godotom. Problem se rješava uspostavom UDP servera u Godot-u i UDP klijenta u Python skripti.

Linija ***Kod***

```
4:      const UDP_IP = "127.0.0.1"
5:      const UDP_PORT = 6666
6:      var server := UDPServer.new()

28:     func _ready():
29:         server.listen(UDP_PORT)
```

Kod. 4.7. PythonCommunicator.gd

Kod 4.7. prikazuje kod implementacije. Linije 4 i 5 postavljaju konstante potrebne za server, IP adresu i port, dok linija 6 instancira objekt *UDPServer*. Unutar funkcije *_ready()* na liniji 29 server se postavlja na zadani port.

U Python skripti prikazanoj kodom 4.8. adresa servera postavljena je na liniji 4, buffer size na liniji 5, i instanciranje *socket()* koji preuzima službu klijenta na 6. liniji.

Linija ***Kod***

```
4:      serverAddressPort = ("127.0.0.1", 6666)
5:      bufferSize = 1024
6:      UDPClientSocket = socket.socket(family=socket.AF_INET,
                                        type=socket.SOCK_DGRAM)
```

Kod 4.8. Communicator.py

4.3.4. Potreba višenitnosti

Godot za igru koristi jednu nit. Pri pozivanju funkcije *OS.execute()*, pokrenuta Python skripta također koristi tu istu nit. Pokretanjem skripti na istoj niti kao i igru donosi izvedbu programa s naglim padom broja sličica u sekundi što usporava izvedbu igre.

Kako bi problem bio riješen potrebno je koristiti klasu *Thread* koja je zadužena za stvaranje i korištenje više niti što omogućuje višenitnost (engl. *Multithreading*).

Linija ***Kod***

```
18:     func startDrawPathThread():
19:         var thread = Thread.new()
20:         thread.start(Callable(self, "drawPath"))
```

Sl.4.9. GameManager.gd – Korištenje višenitnosti

Kod 4.9. prikazuje funkciju koja koristi metodu *drawPath()* sa koda 4.5. U liniji 19 instancira se objekt klase *Thread*, a u liniji 20 poziva se funkcija *start()* za pokretanje predane funkcije u argumentu na zasebnoj niti.

5. STROJNO UČENJE NEPRIJATELJA

5.1. Problem neprijatelja

Trenutno neprijatelj nema implementirano nikakvo ponašanje, ali se može smatrati spremim za implementaciju ponašanja. Kao temelj, može se implementirati skriptirano ponašanje kako bi postojala početna točka.

5.1.1. Skriptirano ponašanje

Budući da je žanr igre 1_na_1 arena FPS, poželjno kretanje neprijatelja igraču treba biti neprepoznatljivo i nejasno kako bi neprijatelj izbjegavao opasnost. Implementacija je te nepredvidivosti pseudo-nasumično kretanje. Korištenjem funkcije *randi_range()*, koja vraća pseudo-nasumičnu cjelobrojnu vrijednost, neprijatelj se može nasumično kretati.

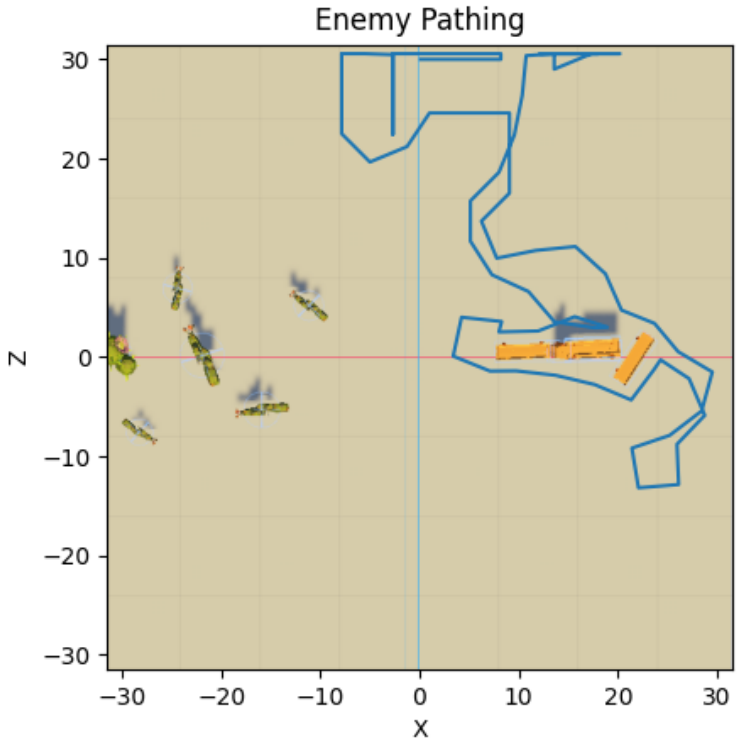
Linija ***Kod***

```
23      var movementDictionary = { 1 : "Forward", 2 : "Left", 3 : "Right",
                                   4 : "Backward", 5 : "Stop"}
        …
103:    func _process(_delta):
105:        var currentDirection = movementDictionary[randi_range(1, 5)]
106:        move(currentDirection)
```

Kod. 5.1. Enemy.gd

Kod 5.1. prikazuje implementaciju neprijateljevog kretanja. Mapiranje cjelobrojne vrijednosti s konkretnim smjerom omogućuje rječnik kretanja koji se nalazi na liniji 23. Unutar funkcije *_process()*, na liniji 105, spremit će se pseudo-nasumičan smjer, koji će se unutar linije 106 proslijediti kao smjer željenog kretanja.

Rezultat takvog kretanja vidljiv je na slici 5.1.



Sl. 5.1. Primjer skriptiranog kretanja nepriatelja

5.2. Strojno učenje

Strojno je učenje pristup programiranju temeljen na korištenju podataka ili prošlih iskustava kako bi se optimizirao određeni kriterij uspješnosti [19]. Cilj je strojnog učenja rješavanje problema koji se ne mogu riješiti eksplicitno definiranim koracima.

Strojno učenje dijeli se na tri osnovna tipa:

- Nadzirano učenje
- Nenadzirano učenje
- Podržano učenje

5.3. Nadzirano učenje

Cilj nadziranog učenja generiranje je funkcije koja mapira ulazne varijable na željene izlazne. Za klasifikacijske probleme, nadzirano učenje čest je pristup zato što je cilj problema učenje računala određenih klasifikacijskih sistemima koje smo osmislili [8]. Za razliku od podržanog, nadzirano i nenadzirano učenje za svoje procese zahtijeva skup podataka.

5.3.1. Prikupljanje skupa podataka

Prepreka nadziranog učenja prikupljanje je podataka. Prije nego što se skup podataka prikupi, važno je razumjeti koji su parametri potrebni za sam proces učenja. Budući je izlazna varijabla smjer kretanja, npr. „Naprijed“, problem se može prezentirati kao klasifikacijski. Potrebne su ulazne varijable pozicije igrača i neprijatelja. Koncept trenutnog skupa podataka bilježenje je igračevih odluka o kretanju ovisno o vlastitoj i neprijateljevoj poziciji. Ulazne varijable mogu biti pozicije igrača i neprijatelja na X i Z osima zato što u igri skakanje nije dopušteno.

Linija ***Kod***

```
97:     func _physics_process(_delta):
99:         currentDirectionString = "Stop"
114:        if Input.is_action_pressed("moveForward"):
116:            currentDirectionString = "Forward"
117:        elif Input.is_action_pressed("moveBack"):
119:            currentDirectionString = "Backward"
121:        elif Input.is_action_pressed("moveLeft"):
123:            currentDirectionString = "Left"
124:        elif Input.is_action_pressed("moveRight"):
126:            currentDirectionString = "Right"
```

Kod 5.2. PlayerMovement.gd

Kod 5.2. prikazuje skriptu objekta igrača unutar koje vrijednost currentDriectionString s linije 99 obilježava trenutan smjer kretanja. Varijabla smjera prima vrijednost ovisno o igračevom trenutnom smjeru kretanja. Primanje vrijednosti implementirano je pomoću dodavanja mogućnosti zapisa trenutnog smjera u trenutku igračeva donošenja odluke. Ako se igrač trenutno ne kreće, zapisana riječ bit će „Stop“.

Kako neprijatelj već posjeduje sposobnost spremanja svoje pozicije tijekom igre, sam će spremati informacije o igraču. U liniji 21 koda 5.3. instancirana je lista i zapisuje se prvi red koji definira stupce skupa podataka. Unutar funkcije *move()* koja se nalazi u liniji 64 nakon svakog ciklusa brojača za kretanje, u liniji 81 spremaju se potrebni podaci u listu koja se pomoću funkcije s linije 33 sprema u datoteku zvanu „*lastPlayerPathing.txt*“.

Linija ***Kod***

```
21:     var playerPath = [["Player_positon", "Enemy_position",
                           "Player_direction"]]


64:     func move(direction : String):
80:         if(MovementTimer.is_stopped()):
81:             if GameManager.player:
                    playerPath.append[
                    [GameManager.player.position.x,
                     GameManager.player.position.z],
                    [self.position.x, self.position.z],
                     GameManager.player.currentDirectionString])


33:     func savePlayerPathing():
34:         var pathFile = FileAccess.open("res://lastPlayerPathing.txt",
                                           FileAccess.WRITE)
35:         pathFile.store_string(str(playerPath))
```

Kod. 5.3.Enemy.gd

5.3.2. Obrada skupa podataka

Prije rješavanja problema učenja potrebno je odraditi predobradu podataka. Za obradu podataka korištene su Python biblioteke Pandas i Numpy. Pandas je biblioteka zadužena za tablični zapis podataka i operacije vezane uz tablice. Numpy je biblioteka zadužena za obradu multidimenzionalnih vektora i matrica.

U kodu 5.4. prikazana je implementacija učitavanja podatkovnog skupa u Pandas tablicu. Linije 9 i 10 otvaraju datoteku skupa podataka i spremaju vrijednosti u varijablu. Nakon učitavanja podataka, u linijama 12 i 13, varijabla je podijeljena u dva skupa. Prvi je skup vektor imena

stupaca, a drugi skup redova tablice. U liniji 16 instanciran je objekt tablice u kojoj su podaci spremljeni.

Linija ***Kod***

```
9:      with open('Data.txt', 'r') as file:
10:         fileData = file.read()
12:     fileData = file.read()data = eval(fileData)
13:     columnNames = data[0]
14:     rows = data[1:]
16:     df = pd.DataFrame(rows, columns=columnNames)
```

Kod 5.4.Enemy.py

Tablica 5.1. prikazuje učitane podatke u Pandas tablici:

Tablica 5.1. Primjer prva 4 elementa neobrađene tablice

Red	*Player_position*	*Enemy_position*	*Player_direction*
0	[0, -30]	[-26.53, -4.18]	Stop
1	[0, -30]	[-26.53, -4.18]	Stop
2	[-0.06, -30.59]	[-26.54, 2.90]	Forward
3	[-0.25, -30.59]	[-26.54, 2.90]	Stop

Nakon učitavanja tablice svi su stupci tipa *Object* što je potrebno prebaciti u druge tipove podataka. Skup podataka stvoren je za rješavanje klasifikacijskog problema. Potrebno je preraditi stupac „*Player_driection*“ u kategoričke vrijednosti te stupce „*Player_position*“ i „*Enemy_position*“ u brojčane. U kodu 5.5. linija 22 prebacuje tip stupca „*Player_direction*“ iz objektnog u kategorički. Od linije 24 do 26 stupac „*Player_position*“ razdvaja se u dva brojčana stupca koji predstavljaju njegove koordinate. Isti postupak prerade primjenjuje se na „*Enemy_position*“ stupcu od linije 30

Linija ***Kod***

```
22:     df['Player_direction'] = pd.Categorical(df['Player_direction'])
24:     df[['Player_X', 'Player_Y']] = pd.DataFrame(
                                        df['Player_position'].tolist(),
                                        columns=['Player_X', 'Player_Y'])
25:     df['Player_X'] = df['Player_X'].astype(float)
26:     df['Player_Y'] = df['Player_Y'].astype(float)
30:     df[['Enemy_X', 'Enemy_Y']] = pd.DataFrame(
                                      df['Enemy_position'].tolist(),
                                      columns=['Enemy_X', 'Enemy_Y'])
31:     df = pd.DataFrame(rows, columns=columnNames)
32:     df['Enemy_X'] = df['Enemy_X'].astype(float)
33:     df['Enemy_Y'] = df['Enemy_Y'].astype(float)
```

Kod 5.5. Enemy.py

do 32. Nakon obrade podataka, iz tablice se brišu stupci „*Player_position*“ i „*Enemy_position*“ unutar linije 34 jer su zamijenjeni drugim stupcima.

Nakon obrade podataka tablica poprima oblik tablice 5.2.

Tablica 5.2. Primjer prva 4 elementa obrađene tablice

Red	*Player_X*	*Player_Y*	*Enemy_X*	*Enemy_Y*	*Player_direction*
0	0	-30	-26.53	-4.18	Stop
1	0	-30	-26.53	-4.18	Stop
2	-0.06	-30.59	-26.54	2.90	Forward
3	-0.25	-30.59	-26.54	2.90	Stop

Jedan od načina dodatne obrade tablice zrcaljenje je svih elemenata po Y osi, uključujući i stupac igračevog smjera kretanja. Takva obrada zahtijeva kopiranje trenutne tablice (vidljivo u liniji 40 koda 5.6.) i množenje svih numeričkih vrijednosti koje predstavljaju Y koordinatu s -1 što je implementirano linijom 43. Razlog zrcaljena podatka po Y-osi je pozicija prepreka u svijetu.

Linija Kod

```
40:     dfReversed = df.copy()
42:     numericColumnNames = ["Player_Y", "Enemy_Y"]
43:     dfReversed[numericColumnNames] * -1
45:     reversedMapping = {"Forward" : "Backward"}
46:     dfReversed['Player_direction'] = dfReversed['Player_direction']
                                           .map(reversedMapping)
48:     df = pd.concat([df, dfReversed], ignore_index=True)
```

Kod. 5.6. Enemy.py

Za promjenu smjera kretanja u stupcu „*Player_direction*“ potrebno je mapirati preslikavanje klasifikacija. Mapiranje je realizirano linijom 45 pomoću rječnika, te operacija zamjene koja se nalazi u liniji 46. Nakon transformacija, nova tablica poprima oblik tablice 5.3. Nakon rotacije potrebno je dodati nove redove u izvornu tablicu što je realizirano linijom 48.

Tablica 5.3. Primjer prva 4 elemenata rotirane tablice

Red	*Player_X*	*Player_Y*	*Enemy_X*	*Enemy_Y*	*Player_direction*
0	0	30	-26.53	4.18	Stop
1	0	30	-26.53	4.18	Stop
2	-0.06	30.59	-26.54	-2.90	Backward
3	-0.25	30.59	-26.54	-2.90	Stop

5.3.3. Podjela podataka na skupove za učenje i testiranje

Kako bi skup bio spreman za implementaciju, potrebno ga je podijeliti na ranije definirane ulazne i izlazne podatke. Ulazni podaci su igračeva i neprijateljeva pozicija, dok je izlaz smjer igrača.

Nakon podjele podataka na ulaz X i izlaz Y, skupovi podataka dodatno se dijele na skupove za treniranje modela i za njegovo testiranje. Nadalje, potrebno je odrediti postotak ukupnog skupa podataka koji će biti korišteni u skupu za testiranje.

Implementacija postupka prikazana je kodom 5.7. Linije 50 i 51 dijele cijeli skup na ulaz i izlaz,

Linija ***Kod***

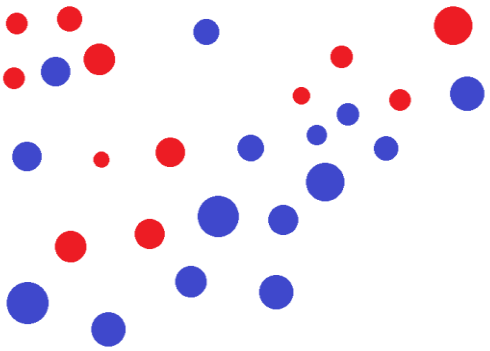
```
50:     X = df[['Player_X', 'Player_Y', 'Enemy_X', 'Enemy_Y']]
51:     y = df['Player_direction'].values
        X_train, X_test, y_train, y_test = train_test_split(X, y,
53:                                        test_size=0.2, random_state=42)
```

Kod 5.7. Enemy.py

dok linija 53 dodatno dijeli skup na dva dijela. U ovom se slučaju 20% skupa podataka koristi za testiranje, dok se 80% koristi za trening.

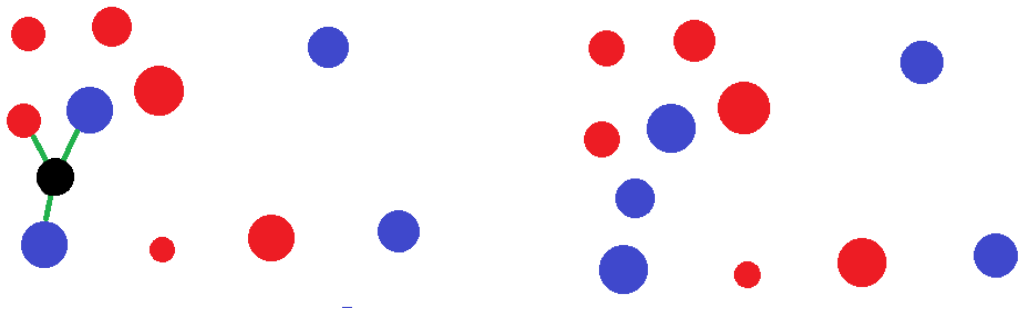
5.3.4. KNN

K-najbližih-susjeda (*engl. k-Nearest-Neighbors*) nadzirani je algoritam strojnog učenja koji se primjenjuje kod rješavanja problema klasifikacije. Ideja kNN-a lijena je metoda učenja koja sprema sve naučene podatke kako bi klasificirala nove redove u tablici računajući udaljenosti između točaka [9]. Svaki red tablice reprezentiran je točkom u n-dimenzionalnoj ravnini vidljiv je iz slike 5.2. koja prikazuje problem dvije klase unutar 2 dimenzije.



Sl. 5.2. kNN – Primjer dvije klase

Pri dodavanju novih elemenata, kNN najprije ulazne vrijednosti pretvori u točnu, nakon čega nova točka prima većinsku klasu svojih k susjeda. Primjer na lijevoj strani slike 5.3. prikazuje dodavanje



Sl. 5.3. kNN – Primjer dodavanja točke i njena klasifikacija

točke u sustav kojemu je konstanta k jednaka 3, te označava najbliže susjede. Desna strana slike 5.3. prikazuje sustav nakon određivanja klase nove točke.

5.3.5. Implementacija

Za implementaciju algoritma kNN korištena je Python biblioteka Scikit-learn. Scikit-learn je biblioteka namijenjena za izradu nadziranih i nenadziranih modela za strojno učenje. Klasa „KNeighborsClassifier“ predstavlja kNN model. Određivanje parametra k zasebni je problem. Pri radu sa skupom podataka koji ima 2500 redova, potrebno je odrediti optimalan k. Način na koji je

Linija ***Kod***

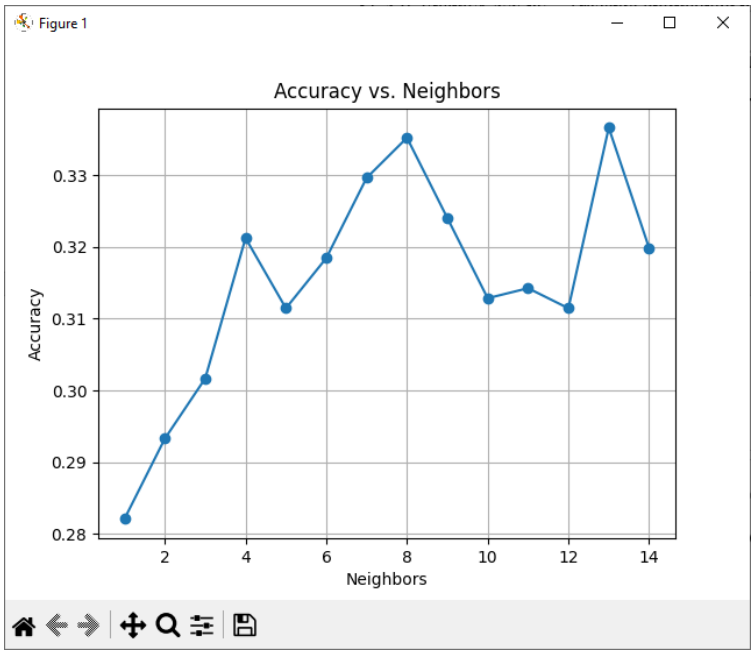
```
65:     Neighbors = range(1, 15)
66:     accuracy = []
67:     for k in Neighbors:
68:         knn = KNeighborsClassifier(n_neighbors = k).fit(X_train,y_train)
69:         Pred_y = knn.predict(X_test)
70:         accuracy.append(metrics.accuracy_score(y_test, Pred_y))
```

Kod 5.8. EnemyKNN.py

najjednostavnije odrediti optimalan parametar treniranje je N modela te uspoređivanje preciznosti svih modela.

Kod 5.8. prikazuje potragu za optimalnim k. Unutar liniji 65 moguće vrijednosti k spremljene su u varijablu. Linija 66 instancira prazno polje u koje će se upisivati preciznost svih modela. Unutar for-petlje, koja prolazi kroz sve moguće vrijednosti k, linija 66 trenira novi model. Linija 67 klasificira skup podataka za testiranje, a linija 68 sprema vrijednost preciznosti testiranog modela

u polje. Pomoću grafičkog prikaza odlučuje se optimalan parametar k, što je u trenutnom slučaju 8, kao što je vidljivo na slici 5.4.



Sl. 5.4. Grafički prikaz potrage za optimalnim parametrom k

Nakon što je pronađen parametar k, prikazano kodom 5.9. potrebno je napraviti i trenirati model.

Linija ***Kod***

```
58:     knn = KNeighborsClassifier(n_neighbors = 8).fit(X_train,y_train)
```

Kod 5.9. EnemyKNN.py

Za konkretno predviđanje kretanja u stvarnom vremenu potrebno je koristiti ranije spomenuti UDP server za komunikaciju s Python skriptama. Nakon inicijalizacije UDP servera u Godotu i UDP klijenta u Python skripti potrebno je uspješno slati, primati i obraditi pakete.

Linija ***Kod***

```
96:     while(True):
98:         data, address = UDPClientSocket.recvfrom(6666)
99:         if len(data) == struct.calcsize('ffff'):
100:            unpackedData = struct.unpack('ffff', data)
101:            reshapedData = np.reshape(unpackedData, [1, 4])
102:            dfData = pd.DataFrame(reshapedData, columns =
                            ['Player_X', 'Player_Y', 'Enemy_X', 'Enemy_Y'])
103:            prediction = knn.predict(dfData)
105:            UDPClientSocket.sendto(str(prediction).encode(),
                                       serverAddressPort)
106:        else:
107:            print("Received data length does not match expected format")
```

Kod. 5.10. EnemyKNN.py

Kodom 5.10. prikazano je primanje paketa koji sadrži koordinate igrača i neprijatelja, obradu predanog paketa, predviđanje modela i vraćanje paketa Godot serveru.

Linija 98 prihvaća poslani paket Godota. Linija 99 provjerava format poruke koja je stigla kako bi skripta mogla reagirati na netočno poslanu poruku. Unutar linije 100 bytecode pretvara se u četiri realna broja koja predstavljaju pozicije igrača i neprijatelja.

Obzirom da se skup podataka sastoji od igračevih odluka o smjeru kretanja, pri predviđanju je potrebno zamijeniti stupce koordinata igrača i neprijatelja. Linija 102 formira tablicu s jednim redom koja će se koristiti u liniji 103 za predviđanje trenutnog stanja koje će u liniji 105 biti poslano Godotu kako bi se neprijatelj kretao u stvarnom vremenu.

Implementacija koda u Godotu prikazana je kodovima 5.11. i 5.12. Kod 5.11. prikazuje pakiranje, slanje i primanje paketa, dok kod 5.12. prikazuje implementaciju dobivenog paketa u neprijatelju.

Linija ***Kod***

```
45:     for i in range(0, peers.size()):
46:         if(GameManager.enemy and GameManager.player):
47:             var packet = peers[i].get_packet()
49:             if(packet):
51:             Packet = processPacketToDirection(packet
                                                  .get_string_from_utf8())
54:         packetToSend.append(Vector2(GameManager.enemy.position.x,
                                        GameManager.enemy.position.z))
55:         packetToSend.append(Vector2(GameManager.player.position.x,
                                        GameManager.player.position.z))
56:         peers[i].put_packet(packetToSend.to_byte_array())
57:         packetToSend.clear()
        …
22:     func getPacket():
23:         if Packet == "Forward" or Packet == "Backward" or
               Packet == "Left" or Packet == "Right" or Packet == "Stop":
24:             return Packet
25:         else:
26:             Return "Stop"
```

Kod 5.11. PythonCommunicator.gd

Linija ***Kod***

```
101:    func _process(_delta):
106         move(GameManager.pythonCommunicator.getPacket())
```

Kod 5.12.. Enemy.gd

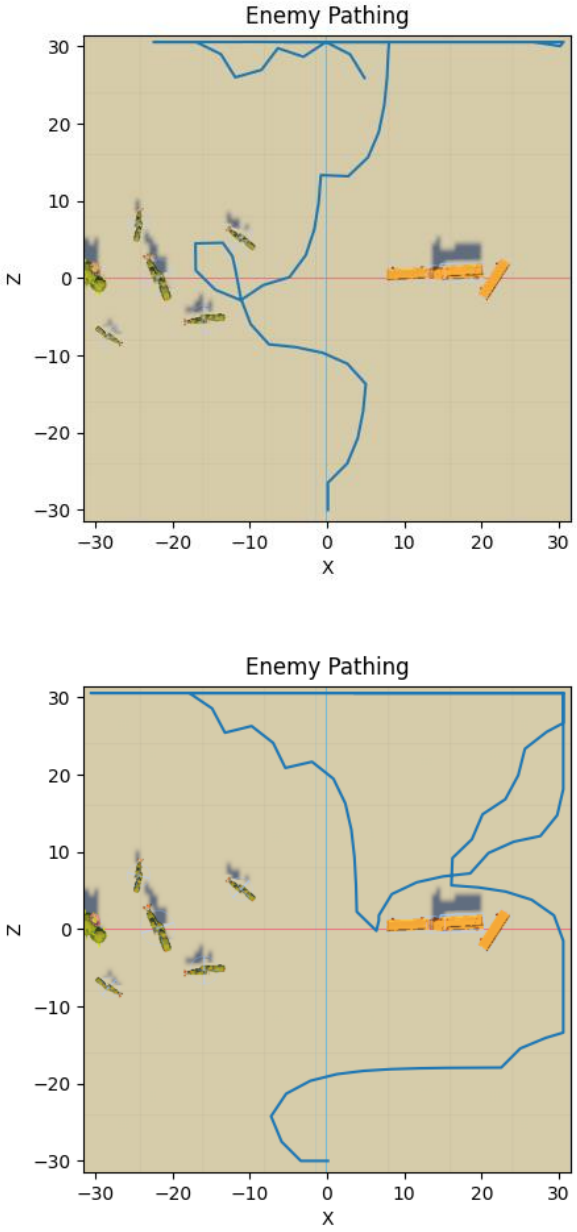
Kod 5.11. prikazuje for-petlju linije 45 koja prolazi kroz sve spremljene klijente UDP servera. Ako trenutno postoje neprijatelj i igrač, odnosno, ako se igrač trenutno bori protiv protivnika, pomoću funkcije *get_packet()* s linije 51 server prima poruku Python skripte. Nakon što poruka stigne paket

je procesiran i spremljen. Kada je paket procesiran, funkcija getPacket()pozvana je unutar neprijatelja kako bi mogao koristiti dobiveni rezultat predikcije kao smjer kretanja.

Paketi koje server šalje Python skripti formiraju se unutar linija 54 i 55 gdje se u pakete spremaju pozicije igrača i neprijatelja. Paket se šalje funkcijom put_packet() te pretvara u bytecode funkcijom to_byte_array() unutar linije 56.

5.3.6. Evaluacija

Radi točne evaluacije modela, isti se uspoređuje sa skriptiranim ponašanjem neprijatelja i komentiraju rezultati. Slikom 5.5. prikazana su dva primjera kretanja pomoću kNN algoritma.



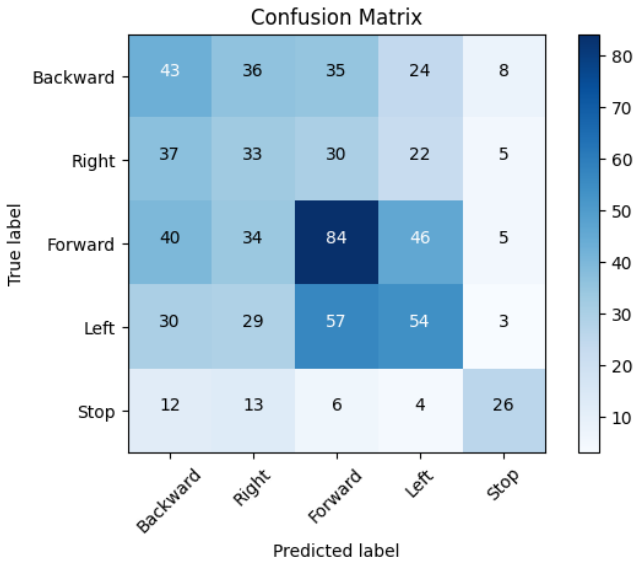
Sl. 5.5. Dva primjera kretanja neprijatelja kNN modelom

Slika 5.5 prikazuje neprijateljeve putanje kroz dvije runde. Iako se neprijatelj nije kretao besmisleno poput skriptiranog modela, kNN model nije predstavljao ideju nepredvidljivosti.

Ideja idealnog kretanja proizlazi iz ideje nepredvidivosti. Problem neprijateljevog kretanja u zadanom primjeru „grljenje“ je zidova. Iako se skriptirani neprijatelj zabijao u zidove i besmisleno kretao, bolje je prenio koncept idealnog kretanja. Zbog upornog približavanja zidovima, kNN model je iznimno predvidljiv, čak i ako se neprijatelj ne zabija u druge objekte i kreće se smislenije, predvidljivost ostaje problem.

Evaluacijom modela, osim usporedbom sa skriptiranim ponašanjem neprijatelja, potrebno je prikazati matricu zabune. Matrica zabune prikazuje uspješnost predviđanja test skupa podataka klasifikacijskih problema.

Prikazano slikom 5.6. Y os tablice prikazuje stvarne vrijednosti tablice, a X os predviđene. Analizom matrice zabune, idealan slučaj prikazuje sve brojeve na sporednoj dijagonali što znači da je model sve ulaze uspješno predvidio. Model trenutno prikazuje poteškoće predviđanja točnih klasa vidljivo iz broja pogrešnih predikcija.



Sl. 5.6. Matrica zabune kNN modela

5.4. Podržano učenje

Cilj je podržanog učenja, za razliku od nadziranog i nenadziranog, koristiti pravila i okružje (engl. *Environment*) za proces učenja. Podržano je učenje problem s kojim se suočava agent čija je zadaća naučiti određeno ponašanje interakcijom s dinamičkim okružjem [13].

5.4.1. Model

Model podržanog učenja čine:

- Agent
- Okružje

Agent je meta podržanog učenja, a okružje definira prostor učenja sa njegovom interakcijom i sistem nagrađivanja svakog koraka. U standardnom modelu podržanog učenja agent je spojen sa okružjem pomoću percepcije i pokreta (engl. *Action*) unutar okružja [13].

Interakcija agenta sa okružjem dijeli se na korake:

1. Agent prima ulaz koji prikazuje trenutno stanje okružja
2. Agent odabere pokret unutar okružja
3. Okružje procesuira pokret i agentu daje novo stanje
4. Agent prima nagradu od okružja

Zbog nedostatka podržanog učenja u Godot-u, potrebno je definirati arenu igre, igrača i neprijatelja u Python-u pomoću biblioteke Gym. Ulogu neprijatelja ima agent, dok okružje učenja oponaša svijet igre i igrača.

Kodom 5.13. prikazana je klasa čija je uloga okružje, CowboyEnv, inicijalizacija stanja varijablom self.state koja se nalazi na liniji 39. Osim varijable stanja, inicijalizirane su sve pomoćne varijable poput 2D prostora čija je uloga svijet igre i koordinate igrača i neprijatelja definirane su linijama

35, 37 i 38. Svi pokreti okružja definirani su linijom 34 kao broj. Rječnik linije 16 pridodaje riječ smjera kretanja brojevima pokreta.

Linija ***Kod***

```
16:     stepToDirection = {
17:         0 : "Stop",
18:         1 : "Forward"
19:         2 : "Backward"
20:         3 : "Left"
21:         4 : "Right" }

32:     class CowboyEnv(Env):
33:         def __init__(self):
34:             self.action_space = Discrete(5)
35:             self.observation_space = Box(
                    low=np.array([-32, -32, -32, -32]),
                    high=np.array([32, 32, 32, 32]))
36:             self.startPosition = np.array([0, 30])
37:             self.position = np.copy(self.startPosition)
38:             self.playerPosition = np.copy(self.startPosition) * -1
39:             self.state = [self.position[0], self.position[1],
                    self.playerPosition[0], self.playerPosition[1]]
```

Kod 5.13. ReinforcementEnv – Inicijalizacija

Kod 5.14. prikazuje funkciju *step()* pomoću koje agent utječe na okružje. Ulaz funkcije jedna je od mogućih pokreta agenta. Funkcija *checkObstacle()* provjerava je li se agent zabio u prepreku, te mu ne dopušta da se pomake u tom smjeru.

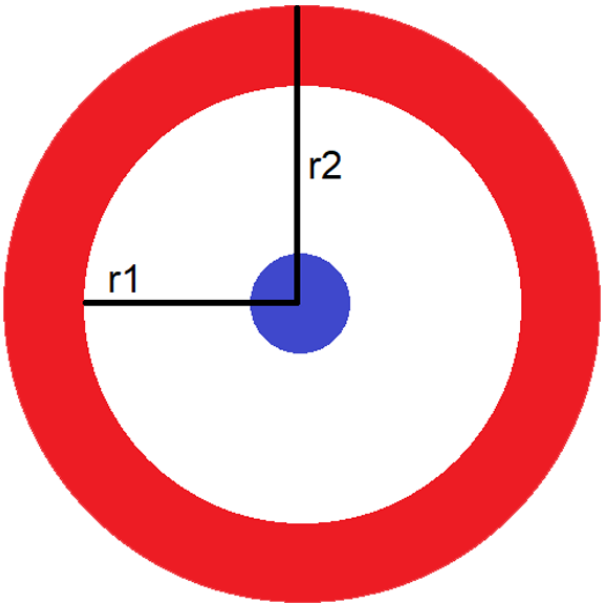
Linija ***Kod***

```
47:     def step(self, action):
49:         previousDistance =  abs(np.linalg.norm(
                self.position - self.playerPosition))
50:         previousPosition = self.position
52:         if stepToDirection[action] == "Forward":
53:             self.position += np.array([movementConstant, 0])
54:         elif stepToDirection[action] == "Backward":
55:             self.position += np.array([-movementConstant, 0])
56:         elif stepToDirection[action] == "Right":
57:             self.position += np.array([0, movementConstant])
58:         elif stepToDirection[action] == "Left":
59:             self.position += np.array([0, -movementConstant])
61:         if self.position[0] > 31:
62:             self.position[0] = 31
63:         elif self.position[0] < -31:
64:             self.position[0] = -31
66:         if self.position[1] > 31:
67:             self.position[1] = 31
68:         elif self.position[1] < -31:
69:             self.position[1] = -31
71:         if not self.checkObstacle(self.position[0], self.position[1]):
72:             self.position = previousPosition
```

Kod 5.14. ReinforcementEnv – Agentova interakcija

Nakon svake odluke, agent prima nagradu u obliku cijelog broja koja predstavlja ocjenu; što je veći broj, nagrada je veća. Budući da je neprijateljev cilj oponašati igrača, ocjena svakog koraka ovisi o njegovoj udaljenosti od igrača. Radi dinamike igre, agent je kažnjen ako stoji na mjestu.

Zbog preciznosti, interes je igrača držati dovoljnu udaljenost od svog protivnika gdje je dovoljno precizan i gdje će moći izbjeći protivnikov napad. Na slici 5.7. plavom točkom prikazan je igrač, crvenim obručem prihvatljiva udaljenost, a radijusima r1 i r2, donja i gornja granica prihvatljive udaljenosti. Princip prihvatljive udaljenosti temelj je sistema za nagrađivanje agenta.



Sl. 5.7. Vizualni prikaz prihvatljive udaljenosti

Sistem nagrađivanja agentu dijeli ocijene tablicom 5.4, a implementiran je u funkciji *step()* kodom 5.15:

Tablica 5.4. Nagrađivanje agenta

Udaljenost od igrača	*Nagrada*
Agent se približio bližem radijusu	1
Agent se udaljio od bližeg radijusa	-2
Agent se nalazi unutar pojasa	3
Agent se nije pomaknuo	-1

Linija ***Kod***

```
77:     if currentDistance >= optimalDistance[0] and
           currentDistance <= optimalDistance[1] and
           previousDistance != currentDistance:
78:         reward = 3
80:     elif previousDistance > currentDistance and
             currentDistance > optimalDistance[1]:
81:         reward = 1
83:     elif previousDistance < currentDistance and
             currentDistance < optimalDistance[0]:
84:         reward = 1
86:     elif previousPosition == self.position:
87:         reward = -1
89:     else:
90:         reward = -2
```

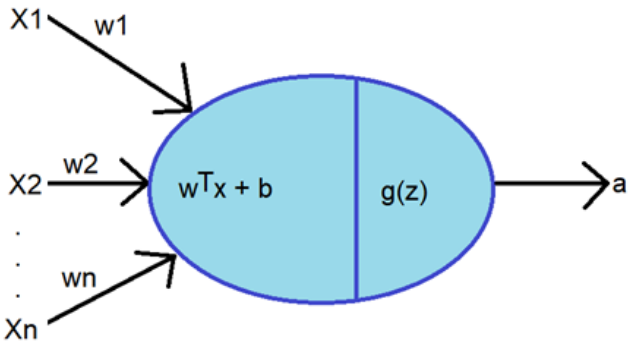
Kod 5.15. ReinforcementEnv – Sistem nagrade

Varijable *currentDistance* i *previousDistance* korištene su za bilježenje trenutne i prijašnje udaljenosti od igrača, a varijabla *previousPosition* za ilježenje prijašnje pozicije agenta.

5.4.2. Neuronske mreže

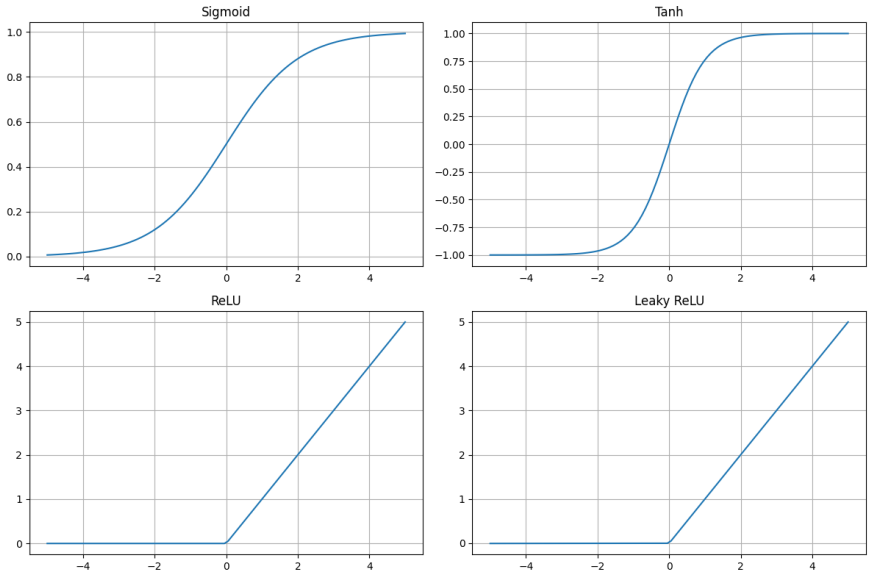
Umjetne neuronske mreže nastale su po uzoru ranih modela senzorne obrade ljudskog mozga. Sastavni element mreže umjetni su neuroni [14]. Umjetni neuroni, kao i mreže, osmišljeni su po uzoru biološkog neurona. Biološki neuron prima ulazne signale od drugih neurona nakon čega će ovisno o ulazu proizvesti izlazni signal.

Umjetni neuron, prikazan slikom 5.8., prima ulaz od broja drugih neurona ili eksternih elemenata, svaki ulazni signal množi se sa svojom težinom nakon čega se signali zbroje [14]. Ako je suma signala iznad određene granica, izlaz signala je jedan, u suprotnom, izlaz je nula.



Sl. 5.8. Umjetni neuron

Granicu neurona definira aktivacijska funkcija. Najčešće korištene funkcije prikazane su slikom 5.9.



Sl. 5.9. Aktivacijske funkcije umjetnih neurona

Neuronska mreža implementirana je kodom 5.16. Za realizaciju korištena je Tensorflow-ova biblioteka Keras. Implementirana mreža se sastoji od 6 slojeva:

- Ulazni sloj sastoji se od 4 neurona, jedan za svaku koordinatu igrača i neprijatelja
- Izlazni sloj sastoji se od 5 neurona, jedan za svaki smjer kretanja, uključujući stajanje te se aktivira linearnom funkcijom prikazano slikom 5.9.
- Mreža se također sastoji od 3 skrivena sloja koja sadrže 24 neurona sa aktivacijskom funkcijom ReLu, prikazano slikom 5.9.

Linija ***Kod***

```
18:     actions = env.action_space.n

22:     model = Sequential()

25:     model.add(Flatten(input_shape=(1, 4)))
26:     model.add(Dense(24, activation="relu"))
27:     model.add(Dense(24, activation="relu"))
28:     model.add(Dense(24, activation="relu"))
29:     model.add(Dense(actions, activation="linear"))
```

Kod 5.16. EnemyReinforcement.py – Postavljanje neuronske mreže

Varijabla actions linije 18 dohvaća ukupan broj različitih pokreta koje agent koristi unutar okružja koja je korištena za broj neurona izlaznog sloja mreže.

5.4.3. Implementacija treniranja

Učenje agenta implementirano je Tensorflow-ovom bibliotekom RL2 koja je namijenjena za podržano učenje koristeći DQM algoritam za učenje agenta.

Duboka Q Mreža (engl. *Deep Q Network*) je algoritam osmišljen 2015. godine od strane DeepMind-a. Algoritam rješava širok spektar Atari igara kombiniranjem podržanog učenja i dubokih neuronskih mreži [17]. Algoritam je nastao poboljšanjem algoritma Q-učenja.

Q učenje temeljen je na Q-funkcijama. RL2 definira Q-funkciju kao maksimum koji može biti dohvaćen počevši sa opservacijom okružja (s) i agentovim pokretom (a) prateći optimalnu politiku učenja. Optimalna Q-funkcija prati Bellmanovu optimalnu jednadžbu prikazanu formulom 5.1.

$$Q^*(s,a) = E[r + \gamma\, max_{a'} Q^*(s',a')] \quad (5\text{-}1)$$

U idealnom slučaju Q-funkcija je tablica svih mogućih kombinacija stanja (s) i pokreta (a). U realnom slučaju, ideja je trenirati funkciju koja aproksimira neuronsku mrežu sa određenim parametrima i težinama. Q-učenje primarno koristi pohlepnu politiku prikupljanja rezultata, dok koristi različite politike ponašanja u okružju.

Kod 5.17. prikazuje stvaranje DQM agenta koji koristi Boltzmannovu politiku. Boltzmannova politika normalizira konačne Q-vrijednosti koristeći softmax funkciju kako bi rezultat prikazao kao skup vjerojatnosti.

Linija ***Kod***

```
33:     policy = BoltzmannQPolicy()
34:     memory = SequentialMemory(limit=5000, window_length=1)
35:     dqn = DQNAgent(
            model=model,
            memory=memory,
            policy=policy,
            nb_actions=actions,
            nb_steps_warmup=10,
            target_model_update=1e-3)
```

Kod 5.17. EnemyReinforcement.py – Postavljanje DQM Agenta

Agentu su predani argument:

- Stvorena neuronska mreža argumentom model
- Memorija koju koristi za učenje argumentom memory
- Bolzmannova politika argumentom policy

- Broj mogućih pokreta argumentom nb_actions
- Broj koraka za zagrijavanje argumentom nb_steps_warmup
- Stopa ažuriranja mreže argumentom target_model_update

Nakon stvaranja mreže potrebno je odrediti metrike učenja prikazane kodom 5.18. Tensorflow-ov algoritam Adam prikazan linijom 39. korišten je za optimizaciju uz stopu učenja određenu argumentom learning_rate. Metrika po kojoj je evaluirano učenje je srednja kvadratna pogreška (engl. *Mean Squared Error*) određena argumentom metrics.

Linija ***Kod***

```
39:     dqn.compile(Adam(learning_rate=1e-3), metrics=['mae'])
43:     dqn.fit(env, nb_steps=10000, verbose=1)
45:     dqn.save_weights('BotWeights.h5f', overwrite=True)
```

Kod 5.18. EnemyReinforcement.py – Učenje DQM Agenta

Učenje je prikazano linijom 43 pomoću funkcije *fit()*, argument *nb_steps* određuje broj ponavljanja. Nakon učenja, linijom 45 spremljene su težine naučene mreže.

5.4.4. Evaluacija

Evaluacija modela određena je na dva načina. Prvi način je pomoću funkcije test() koja sprema ukupne rezultate svake testirane epizode zajedno sa prikazom rezultata kodom 5.19.

Linija ***Kod***

```
49:     scores = dqn.test(env, nb_episodes=5, visualize=True, verbose=2)
50:     print(np.mean(scores.history['episode_reward']))

        Episode 1: reward: 112.000, steps: 60
        Episode 2: reward: 94.000, steps: 60
        Episode 3: reward: 161.000, steps: 60
        Episode 4: reward: 156.000, steps: 60
        Episode 5: reward: 28.000, steps: 60
```

Kod 5.19. EnemyReinforcement.py – Evaluacija naučenog modela

Ukupni rezultati svih epizoda su pozitivni brojevi što naglašava da je treniranje, čak i u najgorem slučaju, izvršilo svoju zadaću. Iako određeni rezultati znatno odstupaju od središnje vrijednosti ukupni rezultati su zadovoljavajući.

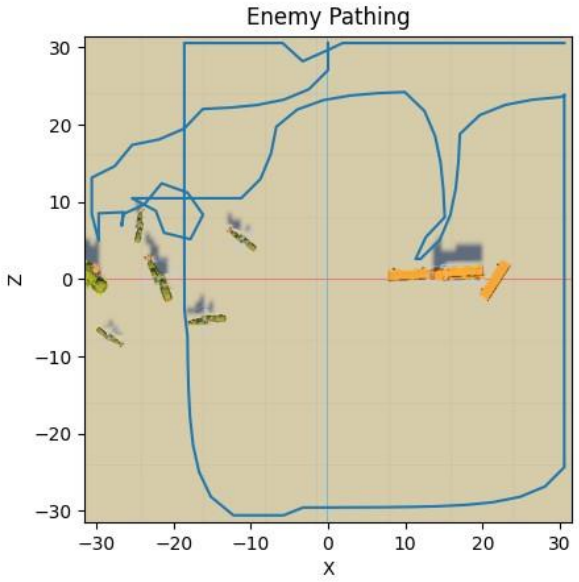
Drugi način evaluacije je igra protiv naučene mreže. Spajanje naučene mreže i igre odrađeno je na sličan način spajanja KNN modela. Glavna razlika, prikazana kodom 5.20, je učitavanje spremljenog naučenog modela i priprema agenta prije komunikacije sa igrom, prikazano od linije 9 do 17. Druga razlika je procesiranje paketa. Potrebno je dodati provjeru ponavljaju li se paketi zbog brzine predviđanja smjera kretanja, što je prikazano linijom 42.

Linija ***Kod***

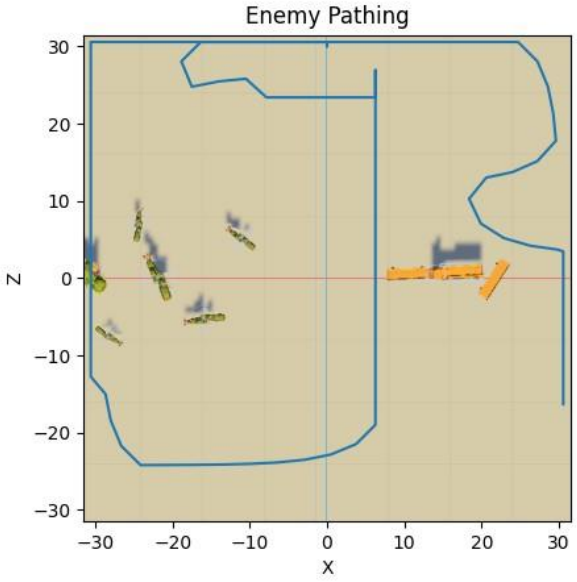
```
9:      env = testingReinfrocementEnv.CowboyEnv()
11:     model = keras.models.load_model('RLmodel.keras')
12:     memory = SequentialMemory(limit=10000, window_length=1)

14:     agent = DQNAgent(
            model=model,
            nb_actions=env.action_space.n,
            memory=memory)

16      agent.compile(Adam(learning_rate=1e-3), metrics=['mae'])
17:     agent.load_weights(filepath='BotWeights.h5f')

35:     while(True):
36:         data, address = UDPClientSocket.recvfrom(6666)
37:         if len(data) == struct.calcsize('ffff'):
38:             unpackedData = struct.unpack('ffff', data)
40:             if unpackedData[0] == float(55):
41:                 exit()
42:             elif oldPackage != None or oldPackage != unpackedData:
43:                 env.setPosition(unpackedData[0], unpackedData[1])
44:                 env.setPlayer(unpackedData[2], unpackedData[3])
46:                 prediction = agent.forward(env.getCurrentState())
48:                 predictionDirection= stepToDirection[prediction]
51:                 UDPClientSocket.sendto(
                        str(predictionDirection).encode(),
                        serverAddressPort)
52:                 previousPrediction = predictionDirection
53:                 oldPackage = unpackedData
54:         else:
55:             print("Received data length does not match expected format")
```

Kod 5.20. RLnetworking.py – Spajanje Godot-a i naučenog agenta

Rezultati kretanja neprijatelja pomoću naučenog agenta prikazani su slikama 5.10. i 5.11.



Sl. 5.10. Prikaz naučenog modela



Sl. 5.11. Prikaz naučenog modela

Rezultati prikazuju dva tipa slučaja. Lijeva slika prikazuje zadovoljavajuće kretanje, dok desna nepoželjno. U prvom primjeru, neprijatelj se kretao oko prepreka i držao svoju udaljenost od igrača, dok u drugom primjeru se držao zidova i udaljenost koju drži nije konzistentna.

Model naučenog kretanja predstavlja isti problem kao i KNN model, predvidljivost. Ideja nepredvidljivog kretanja teško je naučena, svi igrači nakon nekog vremena postanu predvidljivi svojim protivnicima. Budući da je model osmišljen po autorovom ponašanju, prisutna je mogućnost predvidljivosti ljudskog ponašanja koje model pokušava oponašati. Iako model nije imao problem sa zabijanjem u prepreke svijeta, problem predvidljivosti je značajniji. Jedini način promijene ponašanja je osmišljanje novog modela i sistema nagrade.

Rezultati inicijalnog testiranja igre prikazali su isti problem. Preferirano kretanje je skriptirano jer je predstavljalo veći izazov.

6. ZAKLJUČAK

Rad prikazuje proces stvaranja igre žanra online 1 na 1 arena FPS u razvojnom okružju Godot s naglaskom na ključne objekte igre i proširivanje mogućnosti Godota uz vanjsku suradnju Python skripti.

Problem umrežavanja igrača riješen je stvaranjem ENetMultiplayer servera uz korištenje RPC funkcija koje dopuštaju igračima međusobnu interakciju poput primanja štete.

Zadaća neprijatelja je oponašati igrača, iz tog razloga imaju sličnu konstrukciju, jedina razlika je manjak neprijateljeve kamere. Temelj neprijateljevog kretanja je nasumičnost koja je realizirana korištenjem nasumičnog generatora broja ili korištenjem strojno naučenih modela.

Budući da Godot ne posjeduje mogućnost strojnog učenja, proširen je putem Python skripti koje igra pokreće u pozadini. Strojno učenje realizirano je bibliotekama Scikit-Learn za nadzirani model, a bibliotekama Tensorflow i Gym za podržani model.

Prilikom izrade igre uočeni su problemi izrade i treniranja različitih modela strojnog učenja čije su zadaće kretanje neprijatelja u igri. Moguće je izraditi bolje modele koje preslikavaju ideju nasumičnosti. Realizirani modeli koriste autorovo ponašanje kao temelj te ponašanje drugačijeg igrača može pridonijeti kompleksnosti i uspješnosti modela.

Prisutan je problem pri izvozu igre, potrebna je rekonstrukcija Python okružja i instalacija potrebnih biblioteka jer Godot ne može spremiti Python datoteke u igru. Slanje okružja zajedno s igrom također nije izvedivo zbog količine datoteka koje sadrži. Pakiranje i raspakiranje Python okružja u .zip datoteku traje dvoznamenkast broj sati. Igra je u potpunosti funkcionalna bez Python skripti, neprijatelj će se skriptirano kretati ako nisu pokrenute Python skripte.

LITERATURA

[1] F. Sanglard, „Game Engine Black Book: Wolfenstein 3D“, ruj. 2018, url: https://books.google.hr/books?id=hel6DwAAQBAJ&lpg=PA3&ots=03SeejQIzP&dq=Game%20Engine%20Black%20Book%3A%20Wolfenstein%203D&lr&hl=hr&pg=PP1#v=onepage&q=Game%20Engine%20Black%20Book:%20Wolfenstein%203D&f=false

[2] J.M.P. van Wavern, „The Quake III Arena Bot“, *University of Technology Delft Faculty ITS*, lip. 2001, url: http://www.quadronyx.org/mirror/Quake%20III%20Arena%20Bot%20thesis%20paper%20(2001).pdf

[3] D. Stefyn, „Quake III Arena Game Structures“, *CAIA Technical Report 110209A*, velj. 2011, url: https://researchbank.swinburne.edu.au/file/78a06386-6a1c-4a97-9bb9-c7e9f8d2a06f/1/PDF%20%28CAIA-TR-110209A%29.pdf

[4] J. Postel, „User Datagram Protocol“, kol. 1980., doi: 10.17487/RFC0768

[5] J. Postel, „Transmission Control Protocol“, ruj. 1981, doi: 10.17487/RFC0793

[6] J. Touch, E. Lear, K. Ono, W. Eddy, B. Trammell, J. Iyengar, M. Scharf, M. Tuexen, E. Kohler i Y. Nishida, „Service Name and Transport Protocol Port Number Registry“, *Internet Assigned Numbers Authority*, lip. 2023, https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt

[7] J. Ylönen, „Multiplayer networking with an open source library: implementation plan for HactEngine“, *Turun University of Applied Sciences*, 2016, url: https://www.theseus.fi/bitstream/handle/10024/113538/Ylonen_Johannes.pdf

[8] T. Oladipupo, „Types of machine learning algorithms“, *University of Portsmouth United Kingdom*, 2010, str. 19–48, url: https://pdfs.semanticscholar.org/c4ae/802491724aee021f31f02327b9671cead3dc.pdf

[9] G. Guo, H. Wang, D. Bell, Y. Bi , K. Greer, „KNN Model-Based Approach in Classification“, *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE: OTM Confederated*, stu. 2003, str. 986–996, url: https://www.researchgate.net/profile/David-Bell-32/publication/221549123_KNN_Model-Based_Approach_in_Classification/links/55927ef908ae15962d8e937d/KNN-Model-Based-Approach-in-Classification.pdf

[10] T. Salmela, „Game development using the open-source Godot Game Engine“, *Tampare University of Applied Science*, svi. 2022, url: https://www.theseus.fi/bitstream/handle/10024/746943/Salmela_Tero.pdf?sequence=3

[11] E. Beeching, J. Debangoye, O. Simonin, C. Wolf, „Godot Reinforcement Learning Agents“, *arXiv preprint arXiv:2112.03636.*, 2021

[12] A. D. Birrell, B. J. Nelson, „Implementing Remote Procedure Calls“, *ACM Transactions on Computer Systems*, 1984, str. 39–59, url: https://dl.acm.org/doi/pdf/10.1145/2080.357392

[13] L.P. Kaelbling M. L. Littman, A. W. Moore, „Reinforcement Learning: A Survey“, *Journal of Artificial Intelligence Research 4*, 1996, str. 237–285, url: https://www.jair.org/index.php/jair/article/download/10166/24110/

[14] L.P. Kaelbling M. L. Littman, A. W. Moore, „What are artificial neural networks?“, *Nature biotechnology*, 2008., str. 195-197, url: https://people.binf.ku.dk/~krogh/publications/pdf/Krogh08.pdf

[15] Steam, Wolfenstein 3D, Pristupljeno 30.6.2023, url: https://store.steampowered.com/app/2270/Wolfenstein_3D/

[16] Steam, Quake, Pristupljeno 30.6.2023, url: https://store.steampowered.com/app/2310/Quake/a/

[17] Tensorflow, Introduction to RL and Deep Q Networks, Pristupljeno 4.9.2023., url: https://www.tensorflow.org/agents/tutorials/0_intro_rl

[18] Godot, Lisence, Pristupljeno 8.9.2023., url: https://godotengine.org/license/

[19] E. Alpaydin, „*Introduction to Machine Learning*“, *MIT Press*, 2014., url: https://ds.amu.edu.et/xmlui/bitstream/handle/123456789/6943/Introduction%20to%20Machine%20Learning.pdf?sequence=1&isAllowed=y

Sažetak

Rad opisuje postupak izrade 3D FPS natjecateljske igre u Godot razvojnom okružju. Rezultat je rada izrađena igra u kojoj se igrač bori protiv neprijatelja, čije je kretanje strojno naučeno, ili drugog umreženog igrača. Cilj stvorene igre igračeva je pobjeda napadanjem i obranom od neprijatelja.

Strojno učenje neprijatelja realizirano je proširivanjem mogućnosti Godota koristeći skripte napisane u jeziku Python. Komunikacija između Godot i Python skripti realizirana je putem lokalnog UDP servera. Također se prikazuju tehnologije koje su bile potrebne pri izradi igre.

Ključne riječi: FPS igra, Godot, Online igra, Strojno učenje

Abstract

Creating a 3D FPS competitive video game in Godot

The project assignment was to design a multiplayer FPS arena video game in the Godot game engine. The result is a game which consists of the player fighting against an enemy, which was taught using machine learning, or another connected player. The main goal is for the player to attack the enemy and avoid its attack.

Godot's lack of machine learning was an issue which is resolved by implementing Python scripts. The bridge used for communication between Godot and Python scripts is a local UDP server. In addition, this project highlights different technologies used in the creation process of said game.

Keywords: FPS game, Godot, Machine learning, Online game

Prilog

Poveznica za preuzimanje igre: https://dukathebig.itch.io/untitled-bean-shooter