## Sadržaj

# Uvod

Uz napredak umjetne inteligencije nalazi se i sve više i više njenih primjena. Jedno područje koje često istražuje primjenu umjetne inteligencije jesu računalne igre. Nerijetko dolaze novosti kako je napravljen robot koji igra određenu igru i pobjeđuje najbolje ljudske igrače. No ja ću u ovom radu primijeniti umjetnu inteligenciju u računalnoj igri, ali na malo drugačiji način.

Frka u Svemiru (engl. Fuss in Space) je naziv računalne igre koja je tema ovog rada. Igra je osmišljena za više igrača. Svaki igrač prvo treba ući u neku čekaonicu (engl. Lobby). Ako nema postojećih čekaonica, igrač ju može i stvoriti. Kada se skupi dovoljno ljudi (2-4), može se započeti igra. Svaki igrač ima svog lika i može se tući s drugima, cilj mu je spustiti životne bodove (engl. Hit points) od ostalih igrača na nulu. Udarce stvara tako da mišem nacrta neki simbol na ekranu (krug, kvadrat ili trokut) i ovisno o nacrtanom obliku, aktivirat će se drugačiji udarac.

Dio programa koji prepoznaje koji lik je nacrtan je jako teško algoritamski osmisliti, pa tu dolazi na red strojno učenje. Model strojnog učenja koristi metodu preneseno učenje (engl. transfer learning), što znači da je model već istreniran na ogromnom skupu podataka i naučen da bi prepoznao što je na slikama i onda je istom modelu postavljen drugačiji (manji) skup podataka na kojem je treniran dodatno. To nam omogućuje da model bude jako precizan uz relativno malen skup podataka.

Dio programa sa strojnim učenjem već je napravljen u sklopu predmeta „Projekt“ u programskom jeziku Python.

Kako je igra osmišljena za više igrača, treba postojati klijent - server arhitektura. Pošto jedan dio igre mora biti u Pythonu zbog strojnog učenja i prepoznavanja simbola, odlučio sam se da će to biti server. Klijent će biti u C++u.

Također ideja ovog rada je primijeniti i pokazati znanje naučeno s ostalih predmeta na FER-u. Tako će se obilno koristiti objektno orijentirano programiranje na klijentu, a komunikacija između servera i klijenta koristit će TCP protokol. Ovo su samo neka od tehnologija koje ću koristiti, naučena na predmetima OOP i Komunikacijske mreže. Osim sintaktičkih rješenja na neke probleme, stvaranja mrežnog sučelja i neke osnove neuralnih

mreža, čitatelj bi iz ovog rada trebao naučiti osnove arhitekture za stvaranje igara za jednog ili više igrača. Ovim radom želim pokazati da stvaranje igara ne zahtijeva samo poznavanje sintakse programskog jezika za brže programiranje, već se treba stvoriti potrebna infrastruktura da bi se cijela igra u budućnosti lakše unaprjeđivala.

U ovom radu je igra razvijena do razine koncepta.

# 1. Potrebna znanja

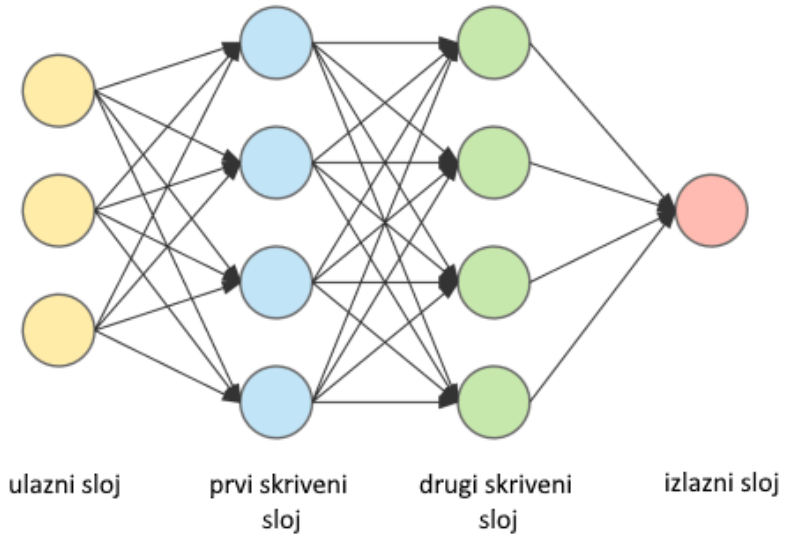
Prije nego uđemo duboko u izradu rada, potrebna su nam neka znanja. U ovom poglavlju ćemo istraživati koncepte vezane uz strojno učenje koje se koristi za prepoznavanje simbola, i vezane općenito za izradu računalnih igara.

## 1.1. Strojno učenje

Strojno učenje je jedna od tehnologija koja je enormno napredovala u zadnjih 20 godina. Pošto bih se u ovom radu htio fokusirati više na temu izrade računalnih igara i većinu programa vezanog sa strojnim učenjem sam odradio u sklopu predmeta „Projekt“, strojno učenje ću objasniti samo bazično.

### 1.1.1. Neuralne mreže

Osnovna stavka svakog strojnog učenja jest neuralna mreža. Neuralna mreža jest skup težina i algoritama koji pokušavaju prepoznati temeljnu vezu u danom skupu podataka kroz proces koji oponaša način na koji ljudski mozak djeluje.




Slika 1.1 – Neuralna mreža s dva skrivena sloja

Kao svaka funkcija, neuralna mreža ima određeni broj ulaza i određeni broj izlaza. Na slici 1.1, to predstavljaju ulazni sloj i izlazni sloj. Svaka strelica na slici uz sebe ima pridružen

jedan broj, taj broj zovemo težina. Svaka kuglica na slici, izuzev kuglica na ulaznom sloju također ima uz sebe pridružen jedan broj, taj broj nazovimo sklonost (engl. bias). Da bi izračunali vrijednost funkcije koju neuralna mreža aproksimira radimo sljedeće:

```
Za svaki sloj neuralne mreže (osim ulaznog sloja):
      Za svaku kuglicu1 u trenutnom sloju:
            vKuglice1 = 0
            Za svaku kuglicu2 u prijašnjem sloju
                  //označimo s w težinu koja je dodijeljena
                  //strelici koja spaja kuglicu1 i kuglicu2
                  vKuglice1 += w * vKuglice2
            //označimo s b vrijednost broja sklonosti na kuglici1
            vKuglice1 += b
            //sigm(x) je funkcija sigmoide opisana kasnije
            Ako trenutni sloj nije izlazni:
                  vKuglice1 = sigm(vKuglice1)
```

Kod 1.2 – Kod za izračunavanje neuralne mreže

Nakon izračunatog – `vKuglice` na kuglicama izlaznog sloja će biti vrijednosti dobivene funkcije koju neuralna mreža aproksimira.

$$f(x) = \frac{1}{1 + e^{-x}}$$

Slika 1.3 – Sigmoidna funkcija

Točnost neuralne mreže jako ovisi o težinama, zato postoje metode da se te težine podese na najbolji mogući način da bi naša neuralna mreža što bolje oponašala željenu funkciju.

### 1.1.2. Povratna propagacija

Povratna propagacija (engl. back propagation) je jedna od metoda za poboljšanje neuralne mreže. Još jedna metoda jest genetički algoritam koji je bolje opisan na predmetu „Uvod u umjetnu inteligenciju“ na trećoj godini studija.

Povratna propagacija funkcionira pomoću gradijenta. Za rad ovog procesa potrebno je imati skup podataka za treniranje, po mogućnosti što veći. Procesom se gleda gradijent (vektor najvećeg rasta funkcije) za cijelu mrežu i pridodaje se trenutnim težinama mreže. Gradijent se gleda za cijeli skup podataka. Proces prilagođavanja mreže za skup podataka se ponavlja dok ne dobijemo željenu točnost neuralne mreže.

### 1.1.3. Preneseno učenje

Preneseno učenje je metoda koja se koristi kad imamo jedan model koji je već jako dobro istreniran na velikom skupu podataka pa mu damo jedan manji skup podataka na kojem se ponovno trenira. Ako su zadaci koje želimo da neuralna mreža radi slični kod manjeg skupa i velikog skupa, model će se uspješno prilagoditi. Pošto je većina težina već ispravno podešena za veliki skup podataka, kada bismo ih htjeli prilagoditi za manji skup, neće biti potrebne velike promjene, s toga će vrijeme treniranja biti kraće i broj podataka ne treba biti opsežan.

## 1.2. Računalna igra za jednog igrača

Postoji puno različitih arhitektura koje se koriste u izradi igara za jednog igrača. Većina ima neke zajedničke stavke, pa ćemo sada proći kroz neke najbitnije.

### 1.2.1. Igraća petlja

Temeljna stavka svake igre, bilo za jednog ili više igrača koristi jest game loop. Igraća petlja (Game loop) je petlja koja se ponavlja stalno dok igra nije završena. Unutar petlje se nalaze tri glavna koraka:

- Upravljanje ulazima (engl. Input handling)
- Funkcija osvježavanja (engl. Update function)
- Funkcija oslikavanja (engl. Render)

#### Upravljanje ulazima

Ideja je da igra prvo treba pokupiti ulazne podatke igrača, odnosno tipke koje je igrač pritisnuo ili pokreti mišem koji su relevantni za igru. Naprimjer, želimo li provjeriti je li igrač stisnuo neki gumb na ekranu, relevantni podaci su pozicija miša i je li bio pritisnut desni klik miša. Ako je, provjerava se jesu li u tom trenutku koordinate miša unutar pravokutnika koji označava gumb na ekranu.

Ovisno o tome u kojem programskom jeziku radimo i kojem operacijskom sustavu, drugačije ćemo prikupljati podatke. Za operacijski sustav Windows naprimjer postoji API pomoću kojeg možemo dobiti te podatke.

### Funkcija osvježavanja

Nakon što je program pokupio ulazne podatke i napravio određene pripreme, treba to nekako iskoristiti za unaprjeđenje stanja igre. U ovom bi se koraku dogodile sve promjene objekata na ekranu, naprimjer pomicanje metka kojeg je ispalio igrač ili provjera sudara (engl. Collision detection) – je li taj metak pogodio jednog igrača – ako je: oduzmi tom igraču životne bodove.

### Funkcija oslikavanja

U ovom koraku se objekti u igrici crtaju igraču na ekran. Koriste se različite funkcije (naravno) ovisno o operacijskom sustavu. Opengl i Vulkan su jedni od popularnih aplikacijskih sučelja za crtanje vektora na ekran korisnika.

### Limitiranje brzine petlje (engl. frame – limiting)

Ideja je da se ovi koraci ponavljaju unutar svake iteracije petlje. Prirodno, neke iteracije će se izvršiti brže od drugih. Ako su u jednom video-okviru (engl. Frame) više tipka pritisnute treba se više podataka obraditi i jednostavno će trajati dulje. To će dovesti do neujednačenosti igre. Ponekad će metak proći veću a ponekad manju udaljenost u određenom vremenu, a željeli bismo da pomak u vremenu bude konstantan za metak i slične objekte. Osim što igra neće biti tečna, procesor će cijelo vrijeme biti zauzet, a to nije dobro. No to se može jednostavno popraviti. Uvodimo pojam limitiranje brzine petlje.

Umjesto da se igraća petlja ponavlja neograničeno puta u sekundi. Ograničit ćemo ju da odradi određeni broj iteracija unutar sekunde, često se ovaj broj postavlja na 60. To će značiti da je FPS (frames per second) 60. Ako se broj iteracija ograniči s obzirom na sekundu, može se lagano izračunati duljina trajanja određene iteracije.

```
T = 1s / FPS
```

Za 60 FPS, to bi bilo:

```
T = 1s / 60 = 16ms
```

Svaki put kada je dovršena jedna iteracija igraće petlje, ako je duljina trajanja te iteracije kraća od 16ms, treba učiniti da procesor miruje za razliku između `T` i duljine trajanja te iteracije. Na primjer, jedna iteracija petlje je trajala 10ms, kada je gotova iteracija, procesor će spavati (odnosno ne raditi ništa) 6ms.

Možemo pretpostaviti da ne radimo nikakve pre složene operacije unutar igre. Većina operacija za igru su trivijalne, a prosječna duljina trajanja jedne iteracije će biti i manja od 2-3 ms. To znači da će većina, ako ne i svaka iteracija petlje doći do dijela gdje čeka.

Ograničavanje video-okvira unutar sekunde ne samo da će učiniti igru tečnijom, nego i reaktivnijom te je česta praksa unutar stvarno-vremenskih (engl. real time) igara.

### 1.2.2. Igraći objekt

Sve što se prikazuje na ekranu unutar igrice ima neka posebna svojstva koja su međusobno drugačija. Na primjer, metak će se uvijek pomicati pravocrtno, a stvorit će se na ekranu tek kad ga neki igrač ispali. Lik (karakter) se neće pomicati pravocrtno, nego ovisno o pritisnutim tipkama igrača će se pomicati lijevo, desno ili skakati. Možemo unutar igre imati blok koda jedan ispod drugoga za svaki takav objekt, ali to će jako brzo postati nepregledno i teško za održavati. Zato uvodimo pojam igraći objekt (engl. Sprite).

Igraći objekt je sučelje za lakše stvaranje novih značajki unutar igre. Svaki igraći objekt je nešto što će se u jednom trenutku prikazati na ekranu. Isto kao i igraća petlja, igraći objekt će imati tri osnovne funkcije:

- Upravljanje ulazima
- Osvježavanje objekta
- Oslikavanje objekta

Ove funkcije su gotovo iste kao i za igraću petlju, zato ih neću toliko detaljno objašnjavati. Navesti ću samo neke osnovne razlike. Za upravljanje ulazima, objekt će provjeravati samo sebi relevantne ulazne podatke, na primjer ako hoćemo pomaknuti „Glavnog lika“, nećemo promatrati je li pritisnuta tipka „M“, nego samo tipke „W“, „A“, „S“ i „D“. Ovisno koja je od tih tipki stisnuta promijenit ćemo mu vektor kretanja u zadanom smjeru, a u idućem će se koraku (osvježavanje objekta) liku promijeniti koordinate.

U koraku oslikavanje objekta bismo htjeli nacrtati samo taj objekt na ekranu pomoću grafičkih funkcija. Naravno moramo imati neke podatke o tome gdje ga želimo nacrtati i kako treba izgledati. Zato su nam potrebne x, y koordinate, širina i duljina objekta te njegova tekstura. Tekstura se najčešće iz trajne memorije učita u radnu memoriju pri instanciranju objekta, to je čisto iz praktičnih razloga, odnosno da ne moramo pri svakom crtanju objekta ponovno učitavati njegovu teksturu.

Svaki novi objekt će naslijediti roditeljsku klasu igraći objekt i prepisati (engl. override) metode koje želi promijeniti.

### 1.2.3. Skup igraćih objekata

S obzirom na to da ćemo imati relativno velik broj tih objekata da svakog ručno održavamo, cilj nam je da ih nekako stavimo u zajednički skup koji će se brinuti o svim objektima unutar skupa. Ovo će također riješiti problem objekata za koje ne znamo kada će nastati i koliko će ih biti, na primjer metak.

Skup igraćih objekata sadrži vektor objekata koje će slijedno posluživati. Skup će također moći sahranjivati nove skupove u taj vektor. To će pomoći u kreiranju izbornika za opcije. Skup će također imati 3 metode: upravljanje ulazima, osvježavanje objekata i oslikavanje objekata. Kada se pozove jedna od tih metode, skup će automatski pozvati tu metodu na svim objektima unutar vektora.

### 1.2.4. Animacije (dodatno)

Za većinu objekata na ekranu koji nisu statični je potrebno dodati neku vrstu animacije, kako bi igra izgledala reaktivno i prirodno. To uključuje pomicanje udova glavnog lika, povećanje i smanjenje gumba dok se prelazi mišem preko njega i slično. Neispravno bi bilo zaključiti da ako je broj video-okvira po sekundi 60, da će se i animacije mijenjati 60 puta u sekundi. Takav zaključak bi vodio do nekoliko problema. Trebalo bi se nacrtati 60 različitih slika animaciju koju želimo izvest, a to je ogromna količina posla. Čak je taj broj veći ako želimo da neki lik drugačije izgleda dok trči, dok hoda ili dok izvodi neku akciju (pucanje metka). Zato treba striktno odvojiti broj video-okvira po sekundi i mijenjanje animacija objektima. Postoji nekoliko načina za rješavanje ovog problema, ja ću ovdje prikazati dva.

#### Animacije izvedene brojačem

Jednostavan brojač unutar objekta može pamtiti koliko je iteracija igraće petlje prošlo, nakon određenog broja iteracija promjeni animaciju i resetira brojač. Želimo li 20 animacija po sekundi, a 60 FPS-a: svake 3 iteracije (60 / 20) bi objekt promijenio animaciju koju crta na ekran. A svakim pozivom funkcije za osvježenje objekta bi objekt povećao unutarnji brojač za 1. Ovakvo rješenje je brzo i jednostavno, ali ima nekoliko

mana. Što kad bismo željeli da se vrši 24 animacije po sekundi. Logično, 60 nije djeljivo s 24, pa se ovo čini nemoguće.

### Animacije izvedene pamćenjem vremena

Taj problem se rješava pamćenjem vremena. Želimo da se animacija mijenja određeni broj puta u sekundi, označimo taj broj s APS (animations per second). To znači da se svakih 1 / APS s treba promijeniti animacija.

Označimo početni vremenski trenutak (proizvoljno trenutak kada je započeta igrica) s T. Imajmo unutarnji brojač x koji se povećava za 1 svakim osvježenjem objekta. Svaki put kada se pozove funkcija osvježenja se odredi vremenski trenutak t – trenutak u kojem je ta funkcija započeta. Provjeravamo sljedeće:

$$t - T > x / APS$$

Ako ovaj izraz vrijedi, znači da se treba promijeniti animacija. Svaku minutu recimo (također proizvoljno), možemo resetirati početni vremenski trenutak T i unutarnji brojač x da ne bismo dobivali pre velike brojke koje ne možemo više mjeriti.

## 1.3. Računalna igra više igrača

Računalna igra za više igrača mnogostruko je kompleksnija od igre za jednog igrača. Glavna komplikacija je komuniciranje između igrača. Za komunikaciju se najčešće koriste TCP ili UDP protokoli. Neću ići u detalje jer komunikacijske mreže nisu predmet ovog rada, ali potrebno je pokazati prednosti i mane svakog protokola (u sljedećim poglavljima), naravno gledajući samo u okviru izrade računalnih igara. U igri za jednog igrača, sve promjene (funkcija osvježavanja) se događaju lokalno, u igri za više igrača nemamo tu privilegiju jer kada bi svaki igrač imao svoju funkciju osvježavanja, također bi imao i drugačije interpretacije stanja igre, to bi dovelo do toga da igra nije jednaka svima. Ovaj problem se rješava klijent – server arhitekturom. Ona se može ostvariti na dva glavna načina.

- Glavni server
- Peer to peer

Svaki će biti opisan u svojim poglavljima. Također podaci bi se između klijenta i servera trebali slati što češće, da bi se održala reaktivnost igre, ali koliko vremena je potrebno za

prijenos tih podataka limitira brzina interneta. Također neke podatke je bespotrebno slati, a nema ni smisla slati podatke više puta u sekundi od potrebnog. Treba se stvoriti neka petlja koja će određeni broj puta u sekundi slati i primati podatke. Ovisno je li to server ili klijent, implementacija petlje će biti drugačija.

### 1.3.1. UDP protokol

UDP protokol spada u grupu bespojnih protokola i ne garantira gotovo ništa, u smislu pouzdanosti i sljednosti podataka. S druge strane, UDP je znatno brži protokol od TCP-a. Ovaj protokol se zato najčešće koristi za slanje podataka koji se trebaju jako često slati između servera i klijenta, kao što je stanje objekata koji se trebaju trenutno klijentu prikazati na ekranu, ili stanje tipkovnice klijenta koje se šalje serveru da bi ih server obradio.

Pretpostavimo da klijent šalje stanje tipkovnice i prima podatke o objektima na ekranu 30 puta u sekundi. Ako jednom u tih 30 puta ne uspije primiti ispravno podatke i ne primi svaki objekt koji bi trebao, nije velik problem, jer igrač vrlo vjerojatno neće ni primijetiti da baš u tom trenutku taj objekt nije postojao. Također, postoji puno načina za provjeru podataka koje je klijent primio, pa i ako u jednom trenutku ne primi neki objekt za koji on pretpostavlja da ga je trebao primiti, može pretpostaviti njegovu lokaciju.

### 1.3.2. TCP protokol

TCP protokol spada u grupu spojnih protokola i garantira pouzdanu i slijednu isporuku podataka. Za igru, ovo je dobro jer znamo da što god pošaljemo na server ili sa servera će stići točno u tom redoslijedu i stići će u cjelini. Jedina mana je što je naravno sporiji iz istog razloga. Zato se ovaj protokol koristi u slanju nekih podataka koji se ne šalju toliko često, a bitno nam je da stignu u cjelini, kao što su neke funkcijske obavijesti na primjer da je igra završena, započeta ili da je stvorena nova čekaonica.

### 1.3.3. Glavni server

Ova metoda za komunikaciju klijenata se bazira na postojanju nekog glavnog servera kojem svi klijenti šalju podatke, on ih obrađuje i svima natrag šalje povratnu informaciju. Dobra je jer striktno držimo odvojene server i klijent i klijenti nikad ne komuniciraju

međusobno nego samo sa serverom, pa imamo kontrolu nad kompletnim protokom informacija.

Pouzdana je metoda i vjerojatno lakša za izvest od druge metode.

### 1.3.4. Peer to peer

Peer to peer je metoda za komunikaciju klijenata gdje ne postoji glavni server, i zapravo jedan ili više klijenata „glumi“ da je server. Klijent uključi opciju da glumi server (engl. hosting) i ostali se klijentu onda spajaju na njegovu adresu. Sve kalkulacije koje bi inače vršio glavni server sada vrši taj klijent. Ovo dovodi do problema gdje taj jedan klijent koji glumi server vjerojatno nema pre jako računalo i često ne može držati korak sa svim informacijama koje dobiva i treba slati ostalim klijentima. Zato se često napravi da se opcija za glumiti server automatski uključi svima i svaki klijent odrađuje dio posla. Ovo je jako teško za izvesti, ali ima nekoliko prednosti u odnosu na metodu „Glavni server“. Glavni server izveden prošlom metodom se može srušiti i onda nikome neće raditi igra. Ovako se to izbjegava.

### 1.3.5. Petlja komunikacije

S obzirom na to je u igraćoj petlji u prijašnjem poglavlju sve operacije vršilo jedno računalo, u igri za više igrača to neće raditi. Ali možemo napraviti neka unaprjeđenja za sličnu funkcionalnost. Klijent će imati sljedeću petlju:

```
while True:
      upravljajUlazima()
      pošaljiUlazeServeru()
      primiPodatkeSaServera()
      obradiPodatke()
      oslikajEkran()
```

Kôd 1.4 – Pseudokod za petlju klijenta

Pojedine funkcije unutar ove petlje možemo limitirati na sličan način kao što smo limitirali animacije. Iz već spomenutih razloga nema potrebe da šaljemo i primamo podatke češće nego što internetska mreža to može podnijeti.

Zato funkcije `pošaljiUlazeServeru()` i `primiPodatkeSaServera()` ne moramo izvršiti pri svakoj iteraciji petlje.

Petlja na serveru bi mogla izgledati ovako:

```
while True:
      primiPodatkeSvihKlijenata()
      obradiPodatke()
      pošaljiPodatkeSvimKlijentima()
```

Kôd 1.2 – Pseudokod za petlju servera

Primijetimo da petlja servera nema funkciju `oslikajEkran()`, to je jer server služi jedino za obradu podataka i organizaciju komunikacije klijenata.

### 1.3.6. Stvaranje čekaonica

Prijašnji kodovi i ideje iz prijašnjih kodova bi odlično radile kad bi svi klijenti igrali zajedno. Isto kao što više ljudi može igrati šah na istom stolu, ali na više ploča, mi moramo osigurati da više ljudi može igrati različite partije iste računalne igre istovremeno. Pojam čekaonica je točno ono što rješava ovaj problem. Kada netko upali igru postoje dvije opcije:

- Stvaranje čekaonice
- Ulaz u čekaonicu

Server kod sebe ima popis svih čekaonica koje trenutačno postoje. Kada netko želi stvoriti novu čekaonicu, potreban je naziv čekaonice koju će stvoriti. Server onda sprema da je čekaonica nekog naziva stvorena i pamti koji su sve igrači trenutno u čekaonici. Kada se drugi igrač želi spojiti u neku čekaonicu, treba upisati naziv te čekaonice, ako ona postoji server će mu odgovoriti da se uspješno spojio s tom čekaonicom i dodati ga u popis igrača te čekaonice. Ako je igra namijenjena za dva do četiri igrača, igrači mogu započeti igru tek kada ima bar dvoje igrača u čekaonici.

Čekaonica će biti izvedena ovako:

```
while True:
      primiPodatkeSvihKlijenataUČekaonici()
      obradiPodatke()
      pošaljiPodatkeSvimKlijentimaUČekaonici()
```

Kôd 1.3 – Pseudokod za petlju čekaonice

Svaki put kada igrač stvori novu čekaonicu, server će stvoriti novu dretvu (engl. thread) koja će vršiti Kôd 1.3 i brinuti se o svim igračima unutar te čekaonice. Ako u nekom trenutku svi igrači izađu iz čekaonice, čekaonica prestaje postojati.

Ovim poglavljem završili smo potrebno znanje (izuzev sintaktičkog znanja programskog jezika izbora) za rad igre, možemo početi s implementacijom.

# 2. Izrada rada i korištene tehnologije

U ovom poglavlju ću proći kroz implementaciju rada i sve korištene tehnologije, počevši s dijelom programa za klijent, a nastaviti s onim dijelom za poslužitelja.

## 2.1. Frontend

Frontend se odnosi na onaj dio programa koji klijent vidi i treba imati na svom računalu da bi igra radila. Pošto se radi o igri za više igrača, klijent bez da se spoji na server (backend), samo pomoću frontend-a neće moći igrati igru.

### 2.1.1. Postavljanje okoline

Zbog prethodnog iskustva, kao programski jezik za Frontend sam odlučio koristiti C++. Od vanjskih biblioteka, za prikazivanje grafike će nam trebati SDL.h biblioteka.

Za mrežno povezivanje sa serverom koristim Windows API.

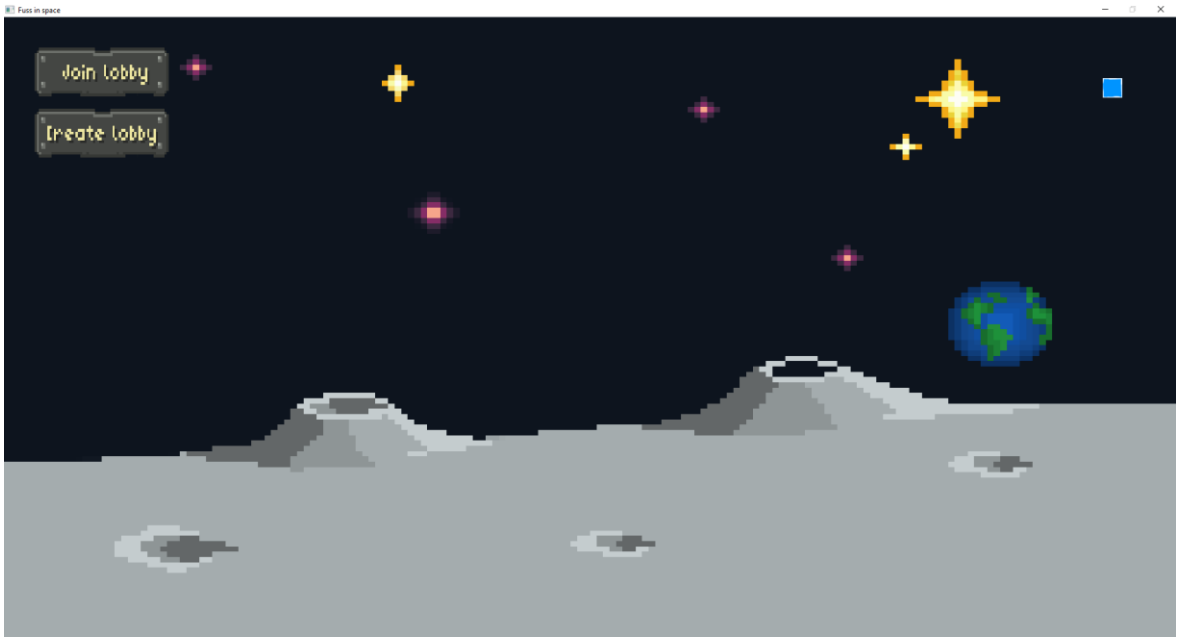
Compiler koji ću koristiti je Microsoft Visual C++ jer ima jako dobre opcije za otklanjanje pogrešaka u programu (engl. debug).

### 2.1.2. Funkcionalni zahtjevi

Klijent kada pokreće igru treba u naredbeni redak iz kojeg pokreće upisati opciju [-a *adresa*]. S time da je *adresa* adresa servera koji je pokrenut (192.0.0.1 za localhost na primjer). Također postoji opcija [-p *port*] i [-f *fps*] za mijenjanje porta na koji se spaja, po zadanom 27015, i za mijenjanje broja iteracija komunikacijske petlje u sekundi. Igra je namijenjena i testirana da radi sa 60 fps.

Nakon što klijent pokrene igru, otvori mu se prozor gdje ima 3 gumba – „JOIN LOBBY“, „CREATE LOBBY“ i gumb za opcije (plavi gumb na slici 1.1), kojim se otvara novi izbornik gdje može upisati i mijenjati svoje korisničko ime. Ovisno koji gumb od prva dva navedena korisnik pritisne, otvara se različit izbornik s poljem za upis imena čekaonice (prazan gumb na slici 1.2) i gumbom „JOIN“, odnosno „CREATE“. Izbornik za uključenje

u postojanu čekaonicu je prikazan na slici 1.2. Kada je korisnik uspješno kreirao ili ušao u čekaonicu ulazi u novi prozor gdje mu piše popis korisnika koji se trenutačno nalaze u istoj čekaonici i piše ime čekaonice. Također postoji gumb s kojim može pokrenuti igru. Kada se pokrene igra, korisniku se prikazuju likovi svih igrača, 3 platforme po kojoj mogu skakati i životni bodovi od svih igrača.




Slika 2.1 – Početni zaslon

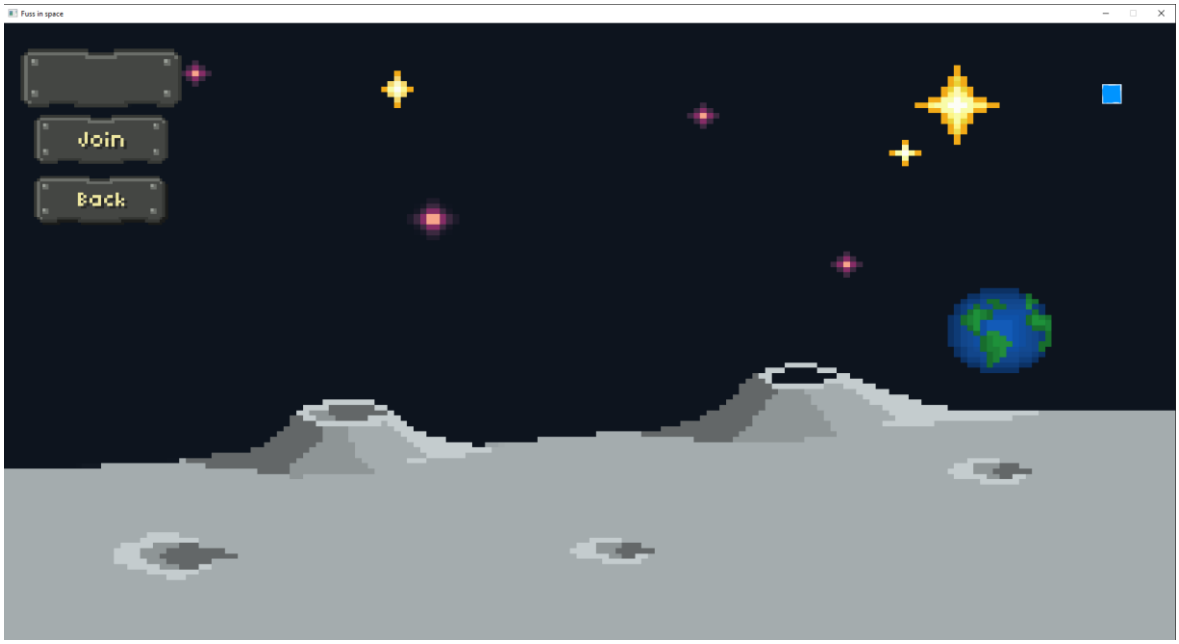
Kroz ovaj opis igre možemo vidjeti nekoliko stvari koji dijele funkcionalnost, stoga ćemo implementirati:

- Klasa Igraći objekt
- Klasa gumb (nasljeđuje igraći objekt)
- Klasa za unos teksta (nasljeđuje gumb)
- Klasa za skup igraćih objekata (nasljeđuje igraći objekt)

Također ćemo imati pomoćne klase koje će nam pomagati oko organizacije projekta:

- Klasa za klijent (šalje i prima poruke sa servera)
- Klasa za ocrtavanje slova
- Klasa za praćenje pokreta miša

Klasa za skup igraćih objekata će također sama biti igraći objekt, to će nam kasnije omogućiti da novu instancu te klase možemo staviti u drugi skup igraćih objekata. Klasa za ocrtavanje slova će nam pomoću u kreiranju klase za unos teksta, a klasa za praćenje pokreta miša će pratiti pokrete miša kako bi ih slala na server kasnije.




Slika 2.2 – Izbornik za uključiti se u čekaonicu

### 2.1.3. Igraći objekt

Prva stvar koju sam implementirao nakon postavljanja okoline i testiranja da svi moduli ispravno rade jest igraći objekt. Implementiran je jednako kao u poglavlju 1.2.2. Svaki igraći objekt sadrži polje tekstura i dstDr. Polje tekstura je tipa pokazivača na SDL_Texture. SDL_Texture je struktura iz biblioteke SDL2.h. DstDr je varijabla koja prikazuje na koje mjesto na ekranu će se taj objekt nacrtati, s obzirom na to da svaki igraći objekt ima predviđeno mjesto na ekranu. DstDr je tipa pokazivača na SDL_Rect što je struktura iz biblioteke SDL2.h i sadrži polja x, y, w, h (width, height). Također svaki igraću objekt sadrži funkcije:

- handleEvents()
- update()
- render()

Kada se pozove funkcija render, zadana tekstura se nacrta na zadano mjesto na ekranu pomoću funkcije SDL_RenderCopy() iz modula SDL2.h. Ostale funkcije nemaju nikakvu funkcionalnost u ovoj klasi jer ona ima različite namjene ovisno o klasi izvedenici.

S obzirom na to da programiramo u C++-u moramo prevoditelju (engl. compiler) nekako naznačiti da se ove funkcije mogu mijenjati u klasi izvedenici (engl. override), stoga ćemo dodati ključnu riječ `virtual` na početku definicija svake od navedenih funkcija.

### 2.1.4. Klasa gumb

Klasa za gumb treba odraditi neku funkciju kada je gumb pritisnut. Stoga trebamo tu funkciju nekako pohraniti. Radimo u C++-u i spremanje funkcija se može lagano odraditi sa sljedećom anotacijom: `void (*func)();` Svaki put kada stvaramo novu instancu gumba, u ovo polje ćemo spremiti jedinstveni pokazivač na tu funkciju (jer želimo da različiti gumbi imaju različite funkcionalnosti). Na svakom pozivu nadjačane funkcije handleEvents() bismo željeli provjeriti je li gumb pritisnut. Stoga ćemo unutar te funkcije dodati provjeru:

```
Je li miš pritisnut, ako je:
      Jesu li koordinate miša unutar ovog gumba, ako da:
            Izvrši funkciju *func
```

Kôd 2.4 – Pseudokod za provjeru je li gumb pritisnut

Ako bismo željeli dodati funkcionalnost da se gumb poveća dokle god pokazujemo mišem preko njega – potrebno je dodati jednu varijablu koja pamti je li trenutno miš pokriva gumb. S toga trebamo dodati jednu funkcionalnost u funkciju handleEvents() za svaki put kada je miš pomaknut:

```
Je li miš pomaknut, ako je:
      Jesu li koordinate miša unutar ovog gumba, ako da:
            Pritisnut = True;
```

Kôd 2.5 – Pseudokod za provjeru prelazi li miš preko gumba

Također bi funkcija render() trebala biti promijenjena, jer treba prikazivati drugačiji gumb ovisno o tome je li pritisnut ili nije. Pseudokod nije potreban zbog trivijalnosti.

### 2.1.5. Klasa za unos teksta

Kada pritisnemo na instancu ove klase, trebali bi moći upisivati tekst koji se sprema unutar klase. S toga ova klasa ima sličnu funkcionalnost kao klasa gumb u smislu da se mora pamtiti je li ona pritisnuta ili nije. Ima i dodatnu funkcionalnost upisivanja teksta. Logično je zaključiti da ćemo ovu klasu izvesti iz klase gumb.

Prepisat ćemo metodu handleEvents(), jer u ovom slučaju trebamo pratiti i sve tipke na tipkovnici koje je igrač stisnuo. Dodat ćemo polje content koje pamti sadržaj teksta u ovoj klasi. Također render() funkcija treba biti drugačija jer neće biti stalna slika koja se prikazuje, nego ovisi o tekstu koji je zasad unesen. Za to će nam pomoći funkcije pomoćne klase za ocrtavanje slova. Klasa za ocrtavanje slova radi pomoću fonta koji sam samostalno nacrtao.

### 2.1.6. Klasa za praćenje pokreta miša

Ova klasa služi kako bismo (dok je igra pokrenuta) mogli dobiti sliku koja prikazuje potez mišem koji igrač želi nacrtati. Ista slika će se kao bitovni niz poslati na server koji će odraditi prepoznavanje simbola i ovisno o nacrtanom simbolu odraditi drugačiju akciju. Ista klasa je korištena za crtanje početnog skupa podataka na kojem je istreniran model.

### 2.1.7. Predložak s varirajućim brojem argumenata

~(engl. varArgs template)

U ovom radu ne postoji poglavlje za spremnik za igraće objekte jer je implementiran gotovo isto kao u poglavlju 1.2.3. Naravno sintaktički je dosta kompleksno za implementirati sve zbog složenosti programskog jezika C++, pa se nude neke komplikacije.

Nakon implementacije prve verzije klase Container (skup igraćih objekata) je bilo jasno da se neki igraći objekti nakon što ih se doda u taj skup neće nigdje više referencirati osim u tom skupu, a neki drugi objekti se hoće referencirati. Na primjer gumb za pokretanje igre se neće više spominjati nakon što smo ga dodali u njegov odgovarajući izbornik, a polje za unos imena se na primjer koristi kasnije jer nas zanima njen sadržaj kada se serveru šalju podatci o korisničkom imenu igrača. U C++-u bi inače trebalo otprilike 3 linije koda za svaki novi igraći objekt koji bismo htjeli dodati u program što nakon nekog vremena jasno

uzrokuje nepreglednost koda i teže održavanje. Stoga sam stvorio funkciju koja ujedno stvara igraći objekt (poziva njegov konstruktor), dodaje ga u vektor odgovarajućeg spremnika i kao povratnu vrijednost vraća pokazivač na taj objekt u slučaju ako ga planiramo referencirati kasnije za dodatne funkcionalnosti. Slijedi isječak koda direktno iz programa koji služi upravo tome:

```
template<typename T, typename... Args >
      T* addSprite(Args&&... args)
      {
            T* s = new T(std::forward<Args>(args)...);
            sprites.push_back(std::unique_ptr<Sprite> {s});
            return s;
      }
```

Kôd 2.6 – Predložak s varirajućim brojem argumenata

`Sprites` je naziv vektora igraćih objekata unutar klase za skup igraćih objekata. `Sprite` je naziv klase za igraći objekt, što znači da bi `Sprite(...)` bio konstruktor iste klase. Ostatak sintakse iz koda 1.6 je prirodno C++-u i stoga nepotrebno za objašnjavati u kontekstu ovog rada.

## 2.2. Backend

Backend se odnosi na dio programa koji prima i šalje podatke klijentima, služi za međusobnu komunikaciju uređaja i trebao bi biti cijelo vrijeme upaljen da se igrica može na klijentu pokrenuti u bilo kojem trenutku. Naravno to zahtjeva da neko računalo cijelo vrijeme upaljeno, a za moju igru koja još nije puštena u pogon (engl. deployment) nije potrebno. Server se pali po potrebi, naravno na računalu koji ima pristup internetu.

### 2.2.1. Postavljanje okoline

Za izradu servera ću koristiti Python jer je jednostavan programski jezik i upravo ovim projektom se može pokazati koliko Python za neke stvari ima elegantno rješenje. Koristit ću Anacondu da postavi sve što Python-u treba za rad. Nakon toga instalirati sve module koji mi trebaju za rad neuralnog modela. Anakonda koristi verziju Python-a 3.9.7.

### 2.2.2. Opis arhitekture

S obzirom na to da je ovaj dio programa rađen u programskom jeziku Python, mnogostruko je jednostavniji od klijenta koji je rađen u C++-u. Stoga će i opis biti poprilično kraći jer jako sliči opisu servera u poglavlju 1.3.5.

Postoji glavna petlja koja cijelo vrijeme hvata nove klijente koji se žele spojiti na server, ako se neki klijent spoji, stvorit će novu dretvu koja će se brinuti o svim podatcima koje taj klijent šalje na server, a trenutna petlja će nastaviti hvatati nove klijente. Dretva za primanje informacija od klijenta je također petlja koja se vrši dokle god je taj klijent spojen sa serverom. Postoji nekoliko različitih vrsta poruka koje server može primiti, a sve počinju određenim kodom (1 oktet), ovisno o kodu slijedi različit broj podataka nakon kodnog okteta. Postojeći kodovi:

- S_CREATE_LOBBY = 41
- S_JOIN_LOBBY = 42
- S_LEAVE_LOBBY = 43
- S_START_GAME = 50
- S_SCREENSHOT_TAKEN = 120
- S_PLAYER_INPUT = 60

Kada server od klijenta primi oktet S_CREATE_LOBBY, zna da taj klijent želi stvoriti novu čekaonicu pa će nakon toga primiti još nekoliko okteta podataka, među kojima će prvih 10 biti naziv čekaonice koju želi stvoriti, a drugih 10 naziv igrača koji ju želi stvoriti. Za uključenje igrača u postojeću čekaonicu vrijedi slična stvar. Svaki put kada je nova čekaonica stvorena, pokreće se nova dretva koja se brine o stvarima vezanim za tu čekaonicu.

Kada server primi poruku koda 43, zna da taj igrač želi izići iz čekaonice i ako on trenutačno jest u nekoj čekaonici, iz iste će ga izbaciti. Kada server primi kod S_START_GAME, to znači da je igrač trenutačno u čekaonici i da želi pokrenuti igru. Server će napraviti određene pripreme. Ako server primi kod S_SCREENSHOT_TAKEN, zna da nakon toga slijede 4 okteta koje predstavljaju dimenzije bitovnog niza koji predstavlja sadržaj poteza mišem koji je igrač nacrtao na ekranu. Ovo će se proslijediti direktno u neuralnu mrežu i pretpostaviti koji simbol je igrač želio nacrtati.

S_PLAYER_INPUT predstavlja kod nakon kojeg slijede podaci o igračevoj tipkovnici koji su relevantni za igru, na primjer jesu li pritisnute tipke „W“, „A“, „S“ ili „D“.

Petlja za čekaonicu se vrši 60 puta u sekundi i svaku iteraciju šalje trenutno stanje čekaonice svim sudionicima. Ovisno je li igra pokrenuta ili igrači i dalje čekaju druge igrače, slat će se drugačiji podaci. Ako igrači i dalje čekaju, šalje se bitovni niz koji predstavlja naziv svih igrača trenutačno u čekaonici. Ako je igra pokrenuta, slat će se jedan oktet koji predstavlja broj igraćih objekata, i nakon tog okteta, toliko bitovnih nizova koji sadrže informacije relevantne za taj igraći objekt.

## 2.3. Problemi s kojima sam se susreo

U ovom poglavlju slijede neki problemi s kojima sam se susreo, neki problemi i dalje nisu riješeni, dok za druge postoje elegantna rješenja.

### 2.3.1. Windows protiv Linuxa

Otkad koristim računalo, koristim uglavnom operacijski sustav Windows. U međuvremenu sam naučio koristiti i operacijski sustav Linux, ali čisto iz navike nastavljam koristiti Windows. Stoga sam i ovaj projekt počeo programirati u Windowsu. Isprva se sve činilo glatko. U jednom trenutku kada sam imao osnovni klijent i server, sam počeo testirati program i otkrio da je ekstremno sporo. Negdje u komunikaciji je došlo do ogromnom kašnjenja, a to nije imalo smisla jer sam sve testirao lokalno na istom računalu. Nakon pretjeranog testiranja i provjeravanja točno gdje u programu dolazi do kašnjenja, otkrio sam da je klijent sasvim u redu, ali da određene funkcije na serveru ponekad kasne.

Dvije funkcije kada se pozivaju, iz Python-ove biblioteke socket, (sock.send() i sock.recv()) kasne u višekratnicima od 15 ms ili ne kasne uopće. A pošto postoji nekoliko poziva tih funkcija unutar komunikacijske petlje koja se treba izvršiti oko 60 puta u sekundi za tečnu igru, to je vremenski kritično.

Činilo se da je problem u implementaciji tih funkcija, a ne u mom programskom kodu. Kasnije je isti Python kod isproban na operacijskom sustavu Linux i sve je radilo. Očito su implementacije drugačije ovisno o operacijskom sustavu.

Nakon otkrivanja ove greške sam razvoj servera nastavio isključivo u operacijskom sustavu Linux, a razvoj frontenda nastavio na operacijskom sustavu Windows. To je

izazvalo vlastite nepogode (uglavnom hardverske) jer sam trebao koristiti dva računala umjesto jednog. Na laptopu sam pokretao backend, a na PC-u frontend. Međusobno su morali biti spojeni na lokalnoj mreži da bi klijentu bila dostupna adresa servera. Postoji i opcija postavljanja javnog servera koja bi bila dostupna svakom klijentu, ali u svrhu testiranja programa i dok igrica ne bude puštena u pogon, ovo nije potrebno.

### 2.3.2. Nemoguće spajanje

Jedan od neotkrivenih problema bilo bi nemoguće pinganje računala na mreži. Ponekad kad želim testirati program na lokalnoj mreži, računala se ne mogu međusobno otkriti i samim time se klijent ne može spojiti na server, a testiranje igre se ne može nastaviti.

Ako pokušam pingati ip adresu računala servera iz naredbenog retka, izbacuje se destination host unreachable. Ovo me dovelo do toga da zaključim da je problem u internetskoj mreži (ili jednom od računala na kojem su testiranja izvođena), a ne u samom programu.

Kasnije (naizgled) ničim izazvano se računala međusobno mogu pingati, a program radi ispravno.

Rješenje problema nije otkriveno, plan je isprobati server na drugom računalu s operacijskim sustavom Linux i vidjeti ishod.

# Zaključak

Na kraju rada htio bih napomenuti da je ovo i dalje projekt u napredovanju i da bih htio jednog dana pustiti cijelu igru u pogon. Mislim da ima puno potencijala jer koristi jednu kreativnu mješavinu tehnologija koja dosad nije puno korištena na ovakav način u industriji video igara.

# Literatura


[1] Simple DirectMedia Layer. *Official WIKI page*. Poveznica: https://wiki.libsdl.org/FrontPage;

[2] Howard, J., Thomas R., *Practical deep learning for coders*. Poveznica: https://www.fast.ai/;

[3] Let's Make Games, *SDL2/C++ Tutorial*. Poveznica: https://www.youtube.com/watch?v=QQzAHcojEKg&list=PLhfAbcv9cehhkG7ZQK0nfIGJC_C-wSLrx;

[4] Codergropher, *C++/SDL2 RPG Platformer Tutorial for Beginners*. Poveznica: https://www.youtube.com/watch?v=KsG6dJlLBDw;

[5] Tech with Tim, *Python Socket Programming Tutorial*. Poveznica: https://www.youtube.com/watch?v=3QiPPX-KeSc;

[6] Nepoznati autor, *Slika neuralne mreže*. Poveznica: https://i1.wp.com/francescolelli.info/wp-content/uploads/2019/05/NeuralNetworks-input-layer-hidden-layer-output-layer.png?resize=448%2C293&ssl=1;

[7] Sigmoidna funkcija. Wiki stranica. Poveznica: https://en.wikipedia.org/wiki/Sigmoid_function;

[8] Ivanković, L., *Fuss in space produkcijska verzija*. Poveznica: https://github.com/Ivkalu/Fuss-in-space;

[9] Ivanković, L., *Fuss in space razvojna verzija za klijent*. Poveznica: https://github.com/Ivkalu/Fuss-in-space-CLIENT;

[10] Ivanković, L., *Fuss in space razvojna verzija za server*. Poveznica: https://github.com/Ivkalu/Fuss-in-space-SERVER;

# Sažetak

Ovaj rad prikazuje proces izrade računalne igre Frka u Svemiru. Igra se bazira na tučnjavi između više igrača koji su skupa u igri. Cilj igre je spustiti protivničke životne bodove na nula. Igrači rade udarce tako da nacrtaju neki simbol na ekranu (trokut, zvijezda, krug) i ovisno o nacrtanom simbolu pokrene se drugačija vrsta napada. Prepoznavanje simbola odrađuje neuralna mreža. Arhitektura igre sastoji se od servera i klijenta. Server neprekidno prima i obrađuje podatke od spojenih klijenata i šalje trenutačno stanje igre svim klijentima. Server je rađen na programskom jeziku Python, a klijent na programskom jeziku C++.

# Summary

This final work is showing process of making computer game Fuss in Space. The game is based of attacking between multiple players in same lobby. The goal of the game is to lower the hit points of all other players to zero. Players make attacks by drawing a sort of symbol on their screen (triangle, star, circle) and based on the drew symbol – different attack will be activated. Neural network is recognising which symbol is drawn. Architecture of game is made of server and clients. Server is continuously receiving and processing data from clients and is sending current game state to all clients. Server is made on Python programming language, and client on C++.