SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2401

RAČUNALNA IGRA AGAR U OKRUŽENJU VIŠE IGRAČA

Dana Dodigović

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2401

RAČUNALNA IGRA AGAR U OKRUŽENJU VIŠE IGRAČA

Dana Dodigović

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 12. ožujka 2021.

DIPLOMSKI ZADATAK br. 2401

Pristupnica:	**Dana Dodigović (0036501185)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	prof. dr. sc. Željka Mihajlović

Zadatak: **Računalna igra Agar u okruženju više igrača**

Opis zadatka:

Proučiti računalnu igru Agar te ostvariti implementaciju korištenjem modernog OpenGL-a i sjenčara pisanih u jeziku GLSL. Proučiti problematiku prisutnu u kontekstu više igrača. Posebice obratiti pažnju na problematiku kašnjenja odziva (engl. lag). Implementirati klijent-server arhitekturu te razmotriti mogućnosti predikcije na strani klijenta kako bi se umanjio učinak kašnjenja odziva (engl. lag compensation). Analizirati i ocijeniti ostvarene rezultate. Diskutirati upotrebljivost ostvarenih rezultata kao i moguća proširenja. Izraditi odgovarajući programski proizvod. Koristiti programski jezik C++17, jezik za sjenčare GLSL i programsko grafičko sučelje OpenGL. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 28. lipnja 2021.

$R\heartsuit$

Sadržaj

1 Uvod **1**

1.1 Igra Agar 2

2 Mrežna komunikacija **3**

2.1 Socket 3

2.2 Transportni sloj 3

2.2.1 TCP i UDP 3

2.3 Posebnosti u igrama u okruženju više igrača 5

2.3.1 Mrežni koncepti u Unityju 5

3 Odabrani koncepti iz igara u okruženju više igrača **7**

3.1 Podjela arhitektura 7

3.1.1 Peer-to-peer arhitektura 8

3.1.2 Klijent-poslužitelj arhitektura 8

3.2 Predikcija na strani klijenta 10

3.3 Usuglašavanje (engl. *Reconciliation*) 12

3.4 Vremenski korak poslužitelja 13

3.5 Mrtvi obračun (engl. *Dead reckoning*) 13

3.6 Interpolacija ostalih entiteta 14

3.7 Kompenzacija kašnjenja 15

4 Implementacija **17**

4.1 Implementacija mrežnog dijela 17

4.1.1 Igra s jednim igračem 17

4.1.2 Dodavanje klijent-poslužitelj arhitekture 18

4.1.3 Asinkrona inačica igre 20

4.1.4 Predikcija na strani klijenta 23

4.1.5 Vremenski korak poslužitelja 24

4.2 Implementacija grafičkog dijela 24

4.2.1 Grafički prikaz scene 24

4.2.2 Grupno iscrtavanje 25

4.2.3 Kamera 26

4.3 Logika igre 26

4.3.1 Stanje igre 26

4.3.2 Detekcija kolizije 29

4.3.3 Generiranje hrane 29

4.3.4 Sustav događaja 29

5 Korištene tehnologije i alati **31**

5.1 Korištenje programa 32

5.2 OpenGL 32

5.2.1 Vertex Buffer Object (VBO) 32

5.2.2 Index Buffer Object (IBO) 32

5.2.3 Vertex Array Object (VAO) 32

5.2.4 Sjenčar vrhova 33

5.2.5 Sjenčar fragmenata 33

5.3 ImGui 34

5.4 Serijalizacija i deserijalizacija 34

5.5 Boost.Asio 35

5.5.1 Povezivanje i prihvaćanje veze 35

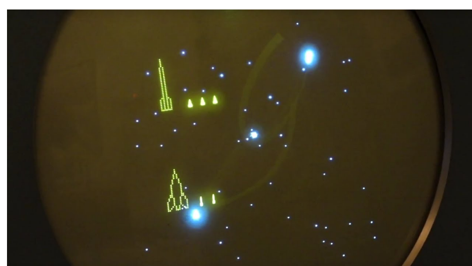
5.5.2 Pisanje i čitanje s mreže 36

5.5.3 Timeri 36

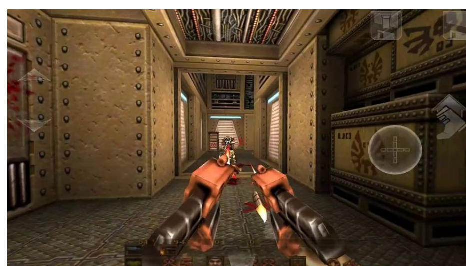
A Dodatni tekstovi programa **40**

1. Uvod

Igre u okruženju više igrača (engl. *multiplayer games*) prvi puta su se pojavile šezdesetih godina prošlog stoljeća (Nim na računalu NIMROD, OXO, Spacewar!...) [1]. No, do pojave igara koje su uključivale komunikaciju preko različitih računala proteklo je još dva desetljeća. Prvom takvom igrom smatra se Maze War koja je objavljena 1973. godine [2]. Mogućnost dva igrača dodana je povezivanjem računala putem serijskog kabla. Prva igra koja je koristila Internetski protokol je prikazivala jednostavnu simulaciju upravljanja avionom, no, preko mreže su se slali višeodredišni (engl. broadcast) paketi što je igru ograničilo na jedan mrežni segment (paketi nisu mogli ići preko usmjernika). Tek 1989. godine dodana je IP multicast funkcionalnost čime je igra postala dostupna i za igranje u okruženju više igrača preko mreže u punom smislu te riječi.



(a) Spacewar!



(b) Quake

Slika 1.1: Primjer igara u okruženju više igrača

Vrlo veliko značenje u razvoju online igara u okruženju više igrača imala je igra Quake u kojoj su prvi puta predstavljeni koncepti koji su ostali i dan danas temelj za sve moderne igre takvog tipa [3]. Prelazak s jednog igrača koji lokalno računa sva stanja na online igre u okruženju više igrača otvorio je novi niz problema koje je bilo potrebno riješiti. Postavilo se pitanje kako sinkronizirati sve igrače uz prividno minimalno kašnjenje na svakom računalu na nesigurnoj mreži (mogućnost gubitka podataka, raznih napada, varanja...). Koncepti koji su to omogućili su detaljnije objašnjeni u sljedećim poglavljima, a samo neki od njih su: predikcija na strani klijenta (engl. *client side prediction*), usuglašavanje (engl. *reconciliation*), kompenzacija kašnjenja (engl. *lag compensation*) i interpolacija entiteta (engl. *entity interpolation*).

1.1. Igra Agar

Agar je online igra u okruženju više igrača razvijena 2015. godine od strane brazilskog programera Matheusa Valadaresa [4].

Igra se sastoji od proizvoljnog broja igrača, od kojih je svaki predstavljen jednim krugom (stanicom) zadane početne veličine kojoj je cilj što više narasti. Igrač to može postići na dva načina: sakupljanjem hrane i tako da pojede protivnike. Hrana je također predstavljena kružićima i pokupljanjem svaka donosi jedan bod. Igrač se kreće kroz mapu korištenjem miša (u smjeru strelice) ili strelica tipkovnice. Kako igrač raste, postaje inertniji kako bi manji igrači mogli zadržati konkurentnost u igri.

Slika 1.2: Isječak iz igre Agar

Osim igrača i hrane, u originalnoj inačici igre podržano je još nekoliko funkcionalnosti. Prva od njih je dijeljenje igrača na više manjih kružića (dioba stanice) kako bi se bržim kretanjem ipak omogućilo jedenje manjih igrača. Također, u igri postoje virusi iza kojih se manji igrači mogu sakriti, dok veće dijele na mnogo manjih dijelova. Navedeno je dodano u igru kako bi se povećao prostor za razvoj strategija i time smanjila repetitivnost igre. U sklopu rada ove funkcionalnosti su samo spomenute, no nisu implementirane jer je naglasak na mrežnim konceptima specifičnim za igre u okruženju više igrača, a ne na razvoju komercijalne igre.

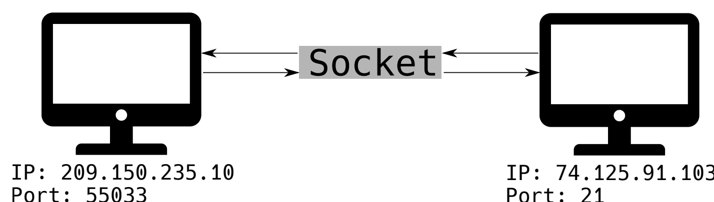
Cilj ovog rada je ostvariti online igru Agar u okruženju više igrača koja podržava navedene koncepte, a bez korištenja gotovog pokretača igara (engl. *game engine*). Također, u sklopu rada je ostvarena pojednostavljena inačica 2D pokretača igara (bez grafičkog sučelja) koja omogućuje lakše rukovanje grafičkim primitivama, pretplatu na različite događaje u sustavu i slično.

2. Mrežna komunikacija

Za razvoj online igre u okruženju više igrača potrebno se upoznati s osnovnim konceptima mrežne komunikacije. Kako bi računala uspješno razmjenjivala podatke potrebne za izvršavanje igre, potrebno je između njih uspostaviti neki način komunikacije. Jedan od čestih načina je komunikacija preko socket-a.

2.1. Socket

Mrežni socket (slika 2.1) je krajnja točka komunikacije u računalnoj mreži (npr. na Internetu). On je jedinstveno identificiran s IP adresom, brojem porta i komunikacijskim protokolom (npr. TCP ili UDP).



Slika 2.1: Komunikacija putem socket-a

2.2. Transportni sloj

Transportni sloj je četvrti od sedam slojeva u OSI modelu i treći od četiri u TCP/IP mrežnom modelu [5]. Osnovna funkcionalnost transportnog sloja je prihvaćanje podataka od sloja iznad, podjela u manje jedinice te slanje na mrežni sloj, a u ovisnosti o tome radi li se o pouzdanom ili nepouzdanom protokolu, dodatnim mehanizmima osigurati da dijelovi stignu ispravno na drugu stranu.

2.2.1. TCP i UDP

TCP je konekcijski orijentirani protokol koji se koristi za komunikaciju putem Interneta. TCP osigurava detekciju i ispravljanje pogreške pri slanju/primanju paketa [5], pouzdano dostavljanje podataka i dolazak paketa u istom redoslijedu kao što su bili poslani. UDP je, s druge strane, protokol kojim se ne uspostavlja veza između računala. Iako je po ulozi sličan TCP-u, on ne detektira i ne radi korekciju pogrešaka. Umjesto toga, paketi se kontinuirano šalju prema primatelju bez obzira na to jesu li već primljeni

ili ne. To omogućava uređajima da komuniciraju brže i s manje dodatnih informacija koje nisu dio sadržaja paketa.

U nastavku je dana kratka usporedba ovih protokola prema različitim svojstvima.

Brzina

UDP je brži od TCP-a zato što se za jedno slanje/primanje mora manje toga napraviti. Dok TCP treba uspostaviti vezu, baratati s pogreškama i garantirati da će svi podaci doći u ispravnom redoslijedu, UDP ne osigurava ništa od navedenog i gotovo da izravno šalje podatke.

Tok podataka i izbjegavanje zagušenja

TCP i mehanizam izbjegavanja zagušenja (engl. *Congestion control*) osigurava da pošiljatelj u niti jednom trenutku ne preopterećuje primatelja odašiljanjem prevelikog broja paketa u prekratkom vremenu [6]. UDP ne nudi tu mogućnost zato što su podaci poslani u kontinuiranoj sekvenci ili su odbačeni. Kod TCP-a se na ovaj način smanjuje mogućnost raznih napada kao što je primjerice napad uskraćivanja resursa (engl. *Denial of Service*).

Konekcijski i beskonekcijski protokol

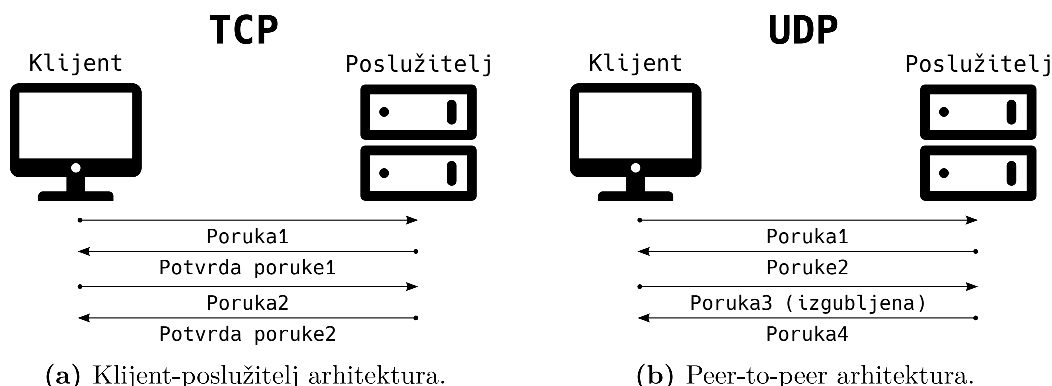
Kao što je ranije napomenuto, UDP je beskonekcijski protokol dok je TCP konekcijski orijentirani. U TCP-u je konekcija uspostavljena između primatelja i pošiljatelja prije slanja podataka i traje za cijelo vrijeme komunikacije. U fazi uspostave je korišteno rukovanje (engl. *three-way handshake*) koje uključuje izmjenu SYN i ACK paketa. UDP ne zahtijeva eksplicitnu vezu da bi poslao podatke.

Pouzdanost

TCP je pouzdani protokol - kada se podaci šalju, garantirano je da će doći na odredište bez pogrešaka. Čak i ako su podaci promijenjeni ili izgubljeni pri slanju, TCP će ih oporaviti i poslati ponovno. TCP također prati nedostaje li koji paket i jesu li stigli u pogrešnom redoslijedu. UDP je za razliku od toga, nepouzdan protokol koji ne garantira dostavu podataka. Zbog toga se datagrami mogu izgubiti tijekom slanja. Također, UDP ne prati pakete između pošiljatelja i primatelja.

Nakon što su navedene razlike između ovih protokola, može se uočiti da je UDP prikladan za aplikacije gdje su efikasnost i brzina važniji od pouzdanosti. Neki od primjera gdje je to slučaj su TFTP (engl. *Trivial File Transfer Protocol*), DNS (engl. *Domain Name System*), VPN tuneliranje, VoIP (engl. *Voice over Internet Protocol*), online igre i prijenos (engl. *streaming*) multimedije. TCP je prikladniji za

aplikacije gdje je pouzdanost važnija od brzine. To uključuje primjenu u elektroničkoj pošti (POP, SMTP, IMAP), SSH-u (engl. *Secure Shell*), pretraživanju Internetom (HTTP/S) i prijenosu datoteka (FTP).



(a) Klijent-poslužitelj arhitektura. **(b)** Peer-to-peer arhitektura.

Slika 2.2: Glavna podjela arhitektura

2.3. Posebnosti u igrama u okruženju više igrača

Najčešći pristup koji se koristi pri implementaciji mrežne komunikacije u igrama u okruženju više igrača je izgradnja mrežnog sloja povrh UDP-a [3]. Vrlo je važno da igrač kod komunikacije može primiti najnovije stanje igre (a poslužitelj posljednju igračevu akciju) bez da se čeka na to da se izgubljeni paketi ponovno šalju. Ako bi se čekalo na ponovno slanje, cijela komunikacija staje. Čak i ako pristignu novi podaci, oni se stavljaju u red i ne može im se pristupiti sve dok izgubljeni paket nije ponovno poslan. U tom slučaju, potrebno je najmanje $round-trip$ (RTT) vremena da se podaci ponovno pošalju, a često je to i $2RTT$ i još pola RTT da paket stigne do primatelja [3]. Dakle, u slučaju ping-a od 125ms, u najboljem slučaju će čekanje trajati $\frac{1}{5}$ sekunde, a u najgorem i pola sekunde ili više.

Zbog navedenog, ali i još mnogih razloga, za razvoj igara koristi se protokol UDP. No, u sklopu implementacije je korišten protokol TCP zato što je komunikacija napravljena samo preko jednog računala, a svo kašnjenje između klijenata i poslužitelja je simulirano kao što je kasnije opisano u poglavlju 5.5.3. Prelazak na UDP protokol u slučaju nadogradnje ove igre ne bi mijenjao sučelje između klijenta i poslužitelja (`read` i `write` metode također opisane u poglavlju 5.5), tako da proširenje ne bi zahtijevalo velike preinake. No, u svrhu demonstracije, TCP radi dovoljno dobro i prikazuje na zoran način željene koncepte. Za budući rad se ostavlja prelazak na UDP i izgradnja sloja povrh njega.

2.3.1. Mrežni koncepti u Unityju

U ovom poglavlju će za usporedbu biti opisana mrežna komunikacija na primjeru gotovog pokretača igara - Unityja. U nastavku rada korištena je knjižnica Boost.Asio

koja služi kao podloga za apstrakciju mrežne komunikacije. Uz mrežno sučelje visoke razine (engl. *high level API*), Unity također nudi sučelje niske razine koje se naziva Transportni sloj (engl. *Transport Layer*) [7]. Transportni sloj omogućava izgradnju vlastitog mrežnog sustava sa specifičnijim ili naprednijim zahtjevima za razvoj igre.

Transportni sloj je tanki sloj iznad mrežnog sloja operacijskog sustava na razini socket-a. On može slati i primati poruke u obliku niza okteta, a naglasak mu je na fleksibilnosti i performansama. Osnovne funkcionalnosti za mrežnu komunikaciju koje pruža su: uspostavljanje konekcije, kontrola toka (engl. *flow control*), dohvaćanje statistike... Ovaj sloj može koristiti dva protokola: UDP za generičku komunikaciju i web sockete za WebGL za slučaj rada s igrama u pregledniku. Za izravno korištenje transportnog sloja, tipični tijek izgleda ovako:

1. Inicijaliziraj mrežni transportni sloj.
2. Konfiguriraj mrežnu topologiju.
3. Stvori poslužitelja.
4. Započni komunikaciju (rukovanje konekcijama, slanje/primanje,...).
5. Zatvori knjižnicu nakon uporabe.

Osim sučelja niske i visoke razine, u Unityju je dostupna i knjižnica srednje razine apstrakcije (engl. *mid-level api*) (MLAPI) koja također služi za apstrakciju mrežne komunikacije. Dok se u sučelju niske razine bilo potrebno fokusirati na najniže protokole i mrežne okvire (engl. *framework*), MLAPI pruža jednostavniji pristup za programera [7]. Ova knjižnica je trenutno i preporučeno programsko ostvarenje pri korištenju Unityja [7] zbog dovoljne fleksibilnosti, ali i jasnijeg sučelja prema programeru.

```
using MLAPI;

NetworkManager.Singleton.NetworkConfig.ConnectionData =
      System.Text.Encoding.ASCII.GetBytes("room password");
NetworkManager.Singleton.StartClient();
}
```

Isječak 2.1: Glavna petlja za singleplayer igru.

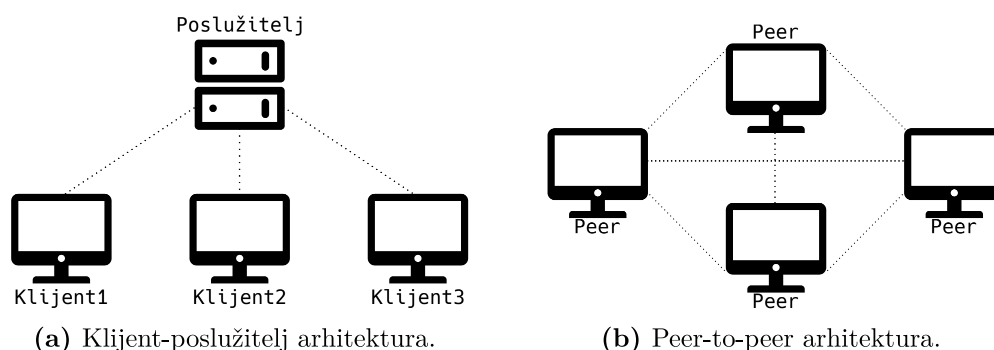
Isječak koda 2.1 prikazuje spajanje klijenta na poslužitelj. Parametar `ConnectionData` predstavlja proizvoljne podatke koje klijent šalje poslužitelju. Najčešće ti podaci predstavljaju neku vrstu ulaznice, lozinke za sobu ili slično na temelju koje se zaključuje treba li veza biti uspostavljena ili ne.

3. Odabrani koncepti iz igara u okruženju više igrača

Razvoj online igara u okruženju više igrača otvara brojne probleme od kojih su glavni priroda mrežne komunikacije (kašnjenje, gubitak podataka, nepouzdanost,...) i varanje sudionika u igri. U igrama s jednim igračem obično nije važno hoće li igrač pokušati varati jer to neće utjecati na nikog osim na njega, no kod više igrača je iznimno važno osigurati da igrači igraju prema određenim pravilima. Kako bi to osigurali, pretpostavit će se da će igrač pokušati varati, a zadaća programera je da to onemogući (ugrađivanjem različitih koncepata u igru).

3.1. Podjela arhitektura

Komunikacija se dijeli na dvije glavne arhitekture: Peer-to-peer i klijent-poslužitelj. Potonja je danas i dominantna pa je tako i uzeta kao temelj za programsko ostvarenje ovog rada.



(a) Klijent-poslužitelj arhitektura. **(b)** Peer-to-peer arhitektura.

Slika 3.1: Glavna podjela arhitektura

Prednosti Peer-to-peer arhitekture u odnosu na klijent-poslužitelj su to što za nju nije potreban centralni poslužitelj, vrlo se lako skalira na veći broj igrača, pogodna je za distribuciju podataka i stabilnija je (po pitanju napada uskraćivanjem usluga). Nedostaci su to da je implementacija generalno složenija, vrlo je teško spriječiti varanje, teško je postići sigurnost te je kašnjenje sustava (engl. *latency*) mnogo veće [8].

3.1.1. Peer-to-peer arhitektura

Peer-to-peer mreža je mreža u kojoj su sva računala međusobno povezana putem Interneta. Takva mreža nema središnjeg poslužitelja, pa svako računalo može dijeliti bilo kakve podatke s bilo kojim drugim računalom u mreži. U tom slučaju svako računalo istovremeno ima ulogu i klijenta i poslužitelja.

Vrlo česta se primjenjuje [8] u:

– dijeljenju datoteka (engl. *File sharing*) (Bit Torrent protokol),
– razmjenjivanju poruka (engl. *Instant messaging*),
– glasovnoj komunikaciji (Skype)
– prijenosu multimedije (engl. *Streaming*)

U peer-to-peer igrama ipak najčešće i dalje postoji jedan poslužitelj koji služi za prihvaćanje novih konekcija te obavještava postojeće klijente o dolasku novog klijenta (kako bi se osiguralo to da se ostali povežu sa njime). Najjednostavniji način za ostvarenje takvog povezivanja je korištenje sobe za razgovor (engl. *chat room*) u kojoj igrači čekaju do kad se svi ne spoje u igru. Kada su svi spremni, igrači pritišću tipku za početak igre i ulaze u igru u istom trenutku. Kompliciraniji pristup omogućava izlaske i ponovne ulaske u igru no to znatno otežava implementaciju. Problem koji se javlja pri korištenju peer-to-peer komunikacije je to što je brzina svakog igrača jednaka brzini najsporijeg igrača. Zbog toga je kašnjenje u takvim slučajevima često neprihvatljivo visoko.

Peer-to-peer komunikacija je prisutna u velikom broju igara, uključujući većinu strategija, sportskih igara i simulacija upravljanja vozilom. Također, većina konzolnih igara na uređajima poput Xbox360 i PlayStation3 koristi takav način komunikacije dok se klijent-poslužitelj arhitektura koristi najčešće u pucačinama u prvom licu (engl. *first person shooter, FPS*) i u igrama u okruženju s velikim brojem igrača (engl. *massively multiplayer online game, MMO*).

3.1.2. Klijent-poslužitelj arhitektura

Za razliku od peer-to-peer arhitekture gdje računala međusobno komuniciraju izravno, u klijent-poslužitelj arhitekturi se svi igrači povezuju na jedinstveno računalo koje se naziva još i poslužitelj. Poslužitelj održava dijeljeno stanje igre i koordinira rad klijenata. Poslužitelj izvršava različiti program od klijenata i ne koristi se izravno od strane nekog igrača. Njegova je svrha da služi ostalim klijentima (trebao bi imati brzu internetsku vezu, biti dostupan u bilo kojem trenutku i ne bi smio pasti usred igre).

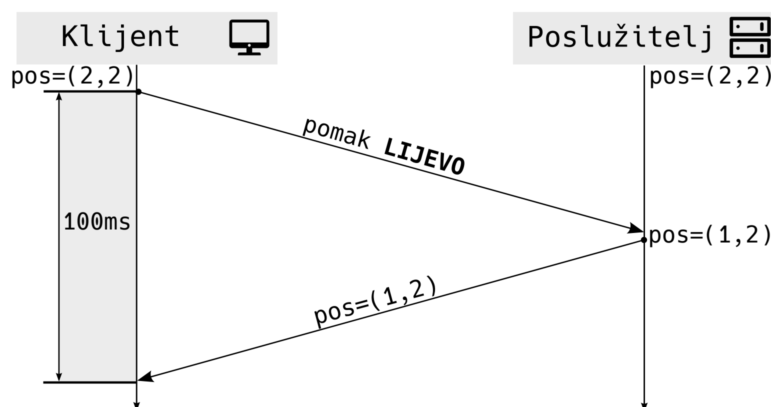
Autoritativni poslužitelj

Autoritativni poslužitelj osigurava igru u kojoj je nemoguće varati. To je ostvareno tako da klijent poslužitelju šalje akciju koju želi izvršiti (npr. na pritisak lijeve strelice šalje poruku s pomakom (-1, 0)), a poslužitelj prima klijentov ulazni podatak i na temelju trenutnog stanje igre primjenjuje klijentovu akciju i računa novo stanje igre. Poslužitelj nakon toga šalje izračunato stanje klijentu koji ga zatim i primjenjuje. Na ovaj način, klijent nikada poslužitelju ne šalje kako bi igra u nekom trenutku trebala izgledati i time ne može utjecati na generalno stanje igre (slika 3.2).

$$Stanje_n + Ulaz_n = Stanje_{n+1} \tag{3.1}$$

Slika 3.2 prikazuje tijek igre uz ovakvu implementaciju uz pretpostavku da je kašnjenje između slanja dvije poruke jednako 100ms (50ms od klijenta do poslužitelja i obrnuto).

U ovakvom načinu igre, klijent ne izvršava nikakvu logiku već se njegov program sastoji od slanja ulaznih podataka, primanja novog stanja i iscrtavanja igre na zaslon. No, iako ovakav pristup rješava problem varanja, istovremeno ne uzima u obzir prirodu mrežne komunikacije i zbog toga ovo dovodi do kašnjenja između igračeve akcije (npr. pritiska tipke) i promjene na zaslonu. Primjerice, igrač pritisne lijevu strelicu, a lik na zaslonu promjeni svoju poziciju tek nakon pola sekunde. To se događa zato što klijentov ulazni podatak (”pomak za jedno mjesto u lijevo”) prvo mora putovati do poslužitelja, poslužitelj treba izračunati novo stanje igre na temelju ulaznog podatka i ažurirano stanje treba ponovno doći do klijenta.



Slika 3.2: Komunikacija putem socket-a

Klijent uz izvršavanje koda poslužitelja

Da bi se doskočilo prethodno opisanom problemu, uvodi se klijent koji osim pasivnog primanja i slanja podataka te iscrtavanja izvršava isti kod kao i poslužitelj uz dodatne provjere (kako bi se osiguralo da klijent i dalje ne može podmetnuti poslužitelju lažno

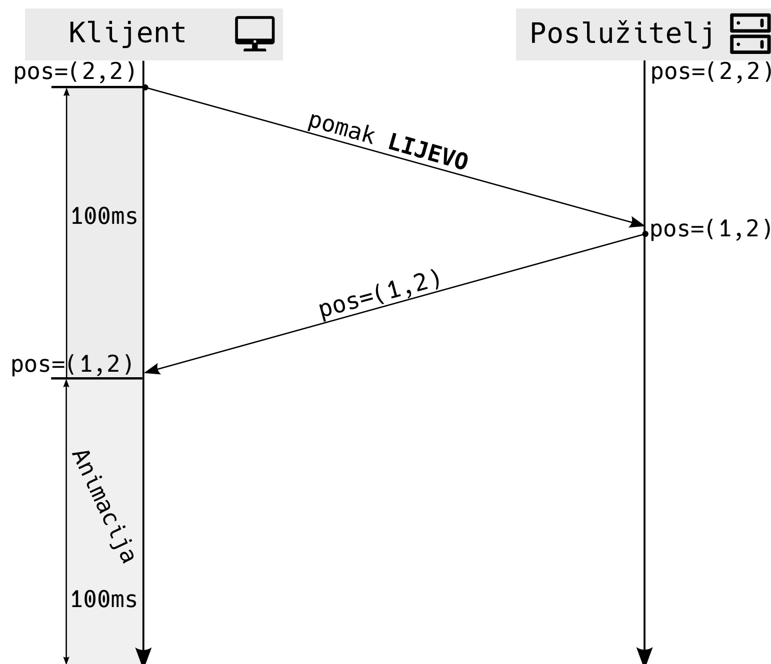
stanje, ali i da pri primanju stanja od poslužitelja ažurira lokalno stanje ako je to potrebno). Prvi koncept koji se javlja pri takvom unaprjeđenju klijenta je predikcija.

3.2. Predikcija na strani klijenta

Većinu vremena će poslužitelj primati ispravne zahtjeve (od igrača koji ne varaju i od onih koji ne varaju baš u tom trenutku). To znači da će većina primljenih ulaznih akcija također biti ispravna i ažurirat će stanje igre na očekivani način. Odnosno, ako je igrač na poziciji (2, 2) i pritisne strelicu prema lijevo, završit će na poziciji (1, 2). Ova pretpostavka je korisna pri definiranju predikcije.

Kašnjenje između igračeve akcije i pripadajućeg vizualnog efekta na ekranu stvara neprirodan osjećaj i onemogućava precizno kretanje kroz mapu te ciljanje drugih igrača (slika 3.3). Predikcija na strani klijenta je način na koji je moguće izbjeći takav efekt i učiniti igračeve akcije vidljive odmah [9].

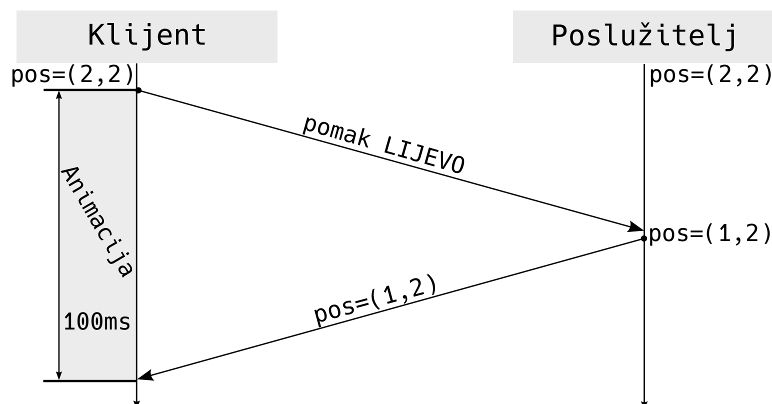
Umjesto da čeka poslužitelja da ažurira njegovu poziciju, lokalni klijent predviđa rezultat vlastite akcije. Sada klijent primjenjuje ista pravila kao i poslužitelj da bi procesirao vlastite naredbe. Nakon što je predikcija završila, igrač se lokalno pomiče na novu poziciju dok ga poslužitelj još uvijek vidi na starom mjestu (kao i drugi igrači).



Slika 3.3: Ažuriranje stanja bez korištenja predikcije.

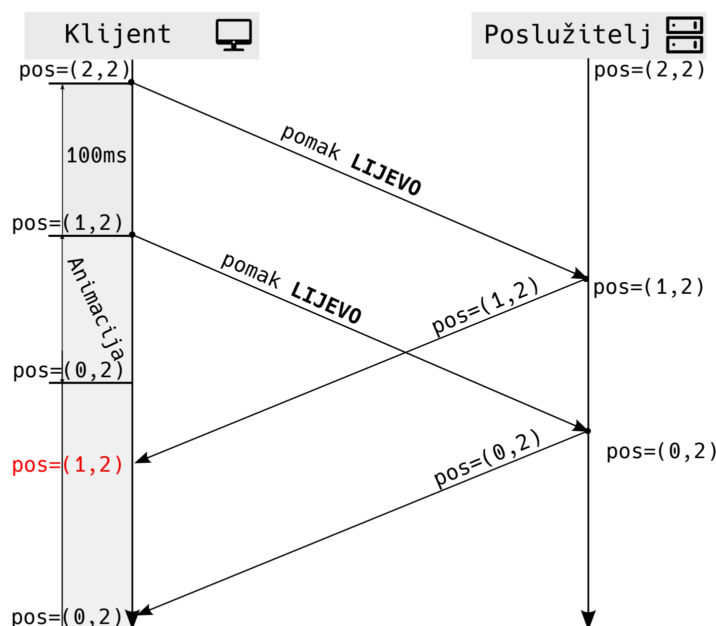
Ovaj koncept odnosi se isključivo na vizualno poboljšanje na lokalnoj razini. Slika 3.4 prikazuje kako se uz dodavanje predikcije više neće trebati čekati da se stanje počne iscrtavati 100ms, nego će to početi odmah. Sada je u potpunosti uklonjeno kašnjenje između akcije i rezultata na zaslonu uz zadržavanje autoritativnog poslužitelja i tih

100ms koje će biti bez novih poruka od poslužitelja se mogu iskoristiti za prikaz animacije sljedećeg koraka.



Slika 3.4: Korištenje predikcije na strani klijenta.

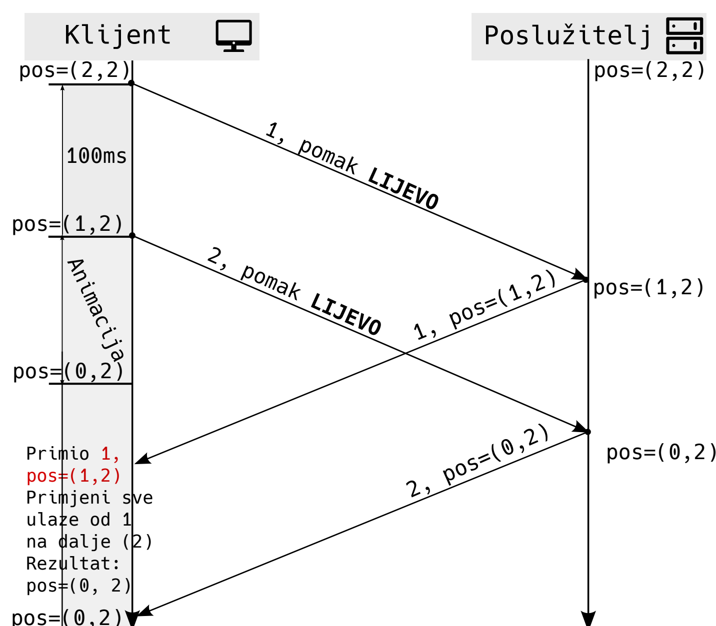
Iako je u gornjem primjeru sve radilo bez ikakvih problema, u stvarnosti je dovoljno malo promijeniti kašnjenje u sustavu i ova metoda će postati neiskoristiva (bez dodatnih poboljšanja). Takav scenarij je prikazan na slici 3.5. U ovom slučaju je kašnjenje do poslužitelja 250ms i prikaz animacije traje 100ms, a igrač je pritisnuo dva puta strelicu prema lijevo. U trenutku t=250ms klijent lokalno ažurira novo stanje igre i računa da se nalazi na poziciji (0, 2), ali poslužitelj je autoritativan pa šalje klijenta natrag na poziciju (1, 2) (i klijent to mora izvršiti). Klijent će ovo na zaslonu vidjeti kao dva pomaka u lijevo, pa pomak u desno i na kraju jedan korak u lijevo, umjesto, onoga što je htio (dva pomaka u lijevo). Takav je prikaz, naravno, pogrešan i neprihvatljiv.



Slika 3.5: Problemi s jednostavnim algoritmom predikcije.

3.3. Usuglašavanje (engl. *Reconciliation*)

Da bi se riješio problem ponovnog vraćanja uvodi se jednostavna nadogradnja prethodnog algoritma. Na svaku igračevu akciju koja se šalje na poslužitelj se dodaje sekvencijalni identifikator (ID) [10]. Nakon slanja akcije, klijent može odmah napraviti predikciju svoje nove pozicije dok poslužitelj procesira akciju i računa autoritativno stanje igre. Kada poslužitelj završi, klijentu šalje odgovor koji sadrži stanje igre i ID zadnje akcije koju je procesirao. Na primanju odgovora, klijent ne validira odmah svoje lokalno stanje (ne uspoređuje jesu li njegovo lokalno i pristiglo stanje isti) nego prvo na svaku akciju koja se javila od akcije s poslanim ID-om primjenjuje poslužiteljev odgovor i onda ju validira. Samo ako stanja nisu jednaka (razlika je manja od neke male predefinirane vrijednosti ϵ), igrač se pomiče na poziciju koju mu je poslužitelj poslao. U slučaju da se stanja razlikuju, najjednostavnije je da se igrač pomakne na točnu poziciju u jednom koraku. Za velike razlike između izračunatog i lokalnog to djeluje previše skokovito pa je poželjno dodatno interpolirati sva stanja između.



Slika 3.6: Usuglašavanje klijenta i poslužitelja

Na ovaj način se postiže da klijent uvijek na zaslonu iscrtava najnoviju akciju, ali i dalje bez mogućnosti slanja pogrešnih informacija klijentu. Prethodni primjeri su implicirali primjenu usuglašavanja stanja pri ažuriranju igračeve pozicije, no isti princip se može primijeniti i na razne druge elemente igre. Primjerice, kada jedan igrač napadne drugog, mogu se prikazati razni vizualni efekti koji predstavljaju štetu koja je učinjena, ali se vitalnost igrača ne bi smjela mijenjati do kada to poslužitelj ne odobri.

Zbog kompleksnosti stanja igre (koje nije uvijek reverzibilno), poželjno je izbjegavati

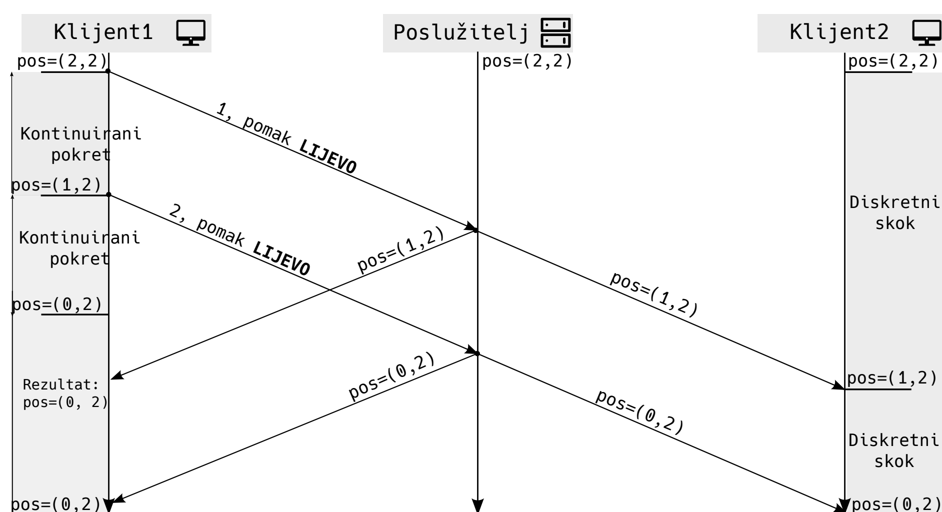
pogađanje igrača do kad to poslužitelj ne kaže, čak iako količina vitalnosti padne ispod nule u lokalnom stanju igre klijenta (što ako je drugi igrač koristio paket prve pomoći prije smrtonosnog napada i poslužitelj to još nije dojavio?). Ovo dovodi do zanimljivog zaključka. Iako je svijet potpuno deterministički i klijenti ne pokušavaju varati, još uvijek je moguće da predviđeno stanje i stanje koje je poslao poslužitelj budu različiti.

3.4. Vremenski korak poslužitelja

Umjesto da poslužitelj na svaku ulaznu akciju ažurira klijenta i šalje novo stanje igre, češće se koristi pristup u kojem poslužitelj ažurira stanje igre fiksni broj puta u sekundi (to se naziva (engl. *tickrate*)). Sada se ulazne akcije svakog klijenta spremaju u jedan red koji se njihovim primjenjivanjem na globalno stanje igre prazni na svaki tickrate. Na ovaj način se osim broja poruka koje poslužitelj mora poslati također smanjuje i opterećenje poslužitelja.

3.5. Mrtvi obračun (engl. *Dead reckoning*)

Kada je broj ažuriranja na strani poslužitelja mali (klijent dobiva rijetke informacije o novim stanjima - u velikom razmaku), klijent zbog usuglašavanja i predikcije i dalje vidi svoju poziciju onako kako bi trebao i bez kašnjenja, no sada problem stvara ono što vidi o ostalim igračima. Ako poslužitelj šalje svakih pola sekunde novo stanje igre, ostali igrači se na strani svakog klijenta miču vrlo skokovito (sve ono što su trebali proći u prethodnih pola sekunde se svodi na jedan skok) (slika 3.7).



Slika 3.7: Ažuriranje stanja s velikim vremenskim korakom.

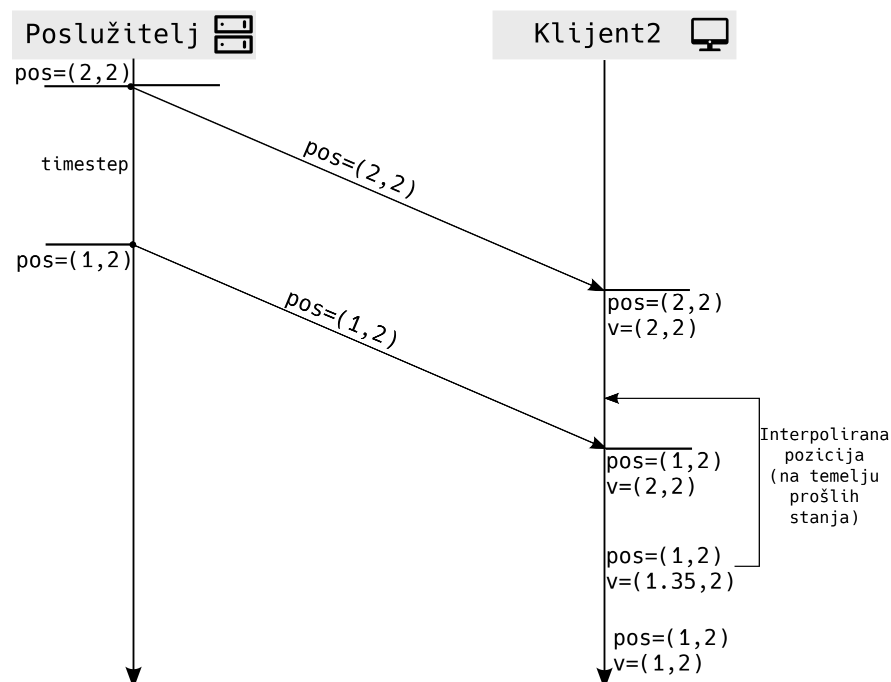
Jedan pristup kako ovo riješiti je, nakon što smo primili stanje igre, predviđati sva stanja do sljedećeg primanja poruke. To je moguće napraviti samo u igrama u kojim se između dva primljena stanja igrač miče gotovo predvidljivom linijom. Primjerice, ako

prikazujemo utrku i znamo da se automobil kreće jednoliko ubrzano te da je krenuo s točke a, u slučaju da se kretao samo ravno, možemo točno znati kroz koje točke je prošao do primanja iduće poruke od poslužitelja. U slučaju da je malo zakretao, predikcija neće biti točna, ali će u većini ovakvih slučajeva biti dovoljno dobra. U ovu metodu se možemo pouzdati zato što znamo da automobili ili neko drugo vozilo ne može raditi vrlo nagle pokrete (npr. potpuni zaokret) u malim jedinicama vremena.

3.6. Interpolacija ostalih entiteta

Iako prethodno opisani postupak radi vrlo dobro u situacijama u kojima je primjenjiv (kada je kretanje gotovo očekivano), u nekima je u potpunosti neuporabljiv. To je slučaj kod pucačina u prvom licu gdje se pozicija, smjer i brzina mogu promijeniti u trenutku. Kada bismo primijenili tehniku mrtvog obračuna, poslužitelj bi stalno slao neko stanje koje nije blizu predviđenog stanja i to bi rezultiralo stalnim teleportiranjem igrača (predviđeno je jedno, a stvarno stanje je nešto sasvim drugo). Takvo ponašanje, naravno, nije prihvatljivo.

Rješenje za ovaj problem je prikazati druge igrače u prošlosti u odnosu na korisnikovog igrača. Recimo da je primljeno stanje igre u t = 1000ms i da se prima svakih 100ms (slika 3.8). To znači da su već dostupni podaci od t = 900ms (odnosno znamo gdje je igrač bio u ta dva trenutka). Tako da se od t = 1000ms do 1100ms, prikazuje ono što je igrač radio od t = 900ms do t = 1000ms. Na ovaj način se prikazuje što je igrač stvarno radio u tim trenucima, samo uz 100ms kašnjenja (nema predikcije).

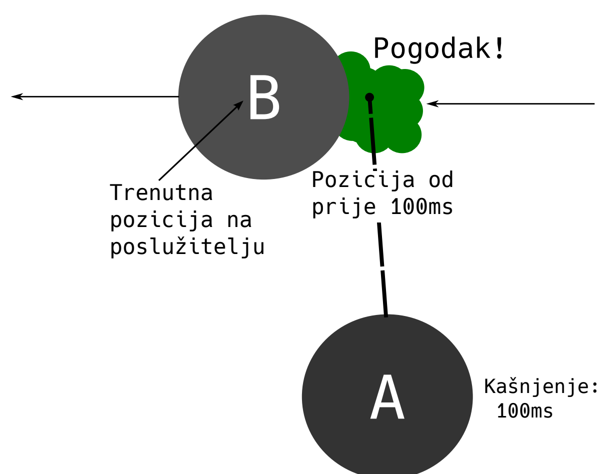


Slika 3.8: Interpolacija.

U većini slučajeva, interpolacija će raditi dovoljno dobro, no, za dodatno poboljšanje je moguće da poslužitelj umjesto novog stanja, šalje i niz stanja između (npr. od svakih 10ms između). Također, važno je za primijetiti da uvođenjem ove tehnike sada svaki igrač vidi malo različiti svijet, zato što sebe vidi u sadašnjosti, a ostale igrače u prošlosti. Čak i za igre s brzim tempom (engl. *fast-paced*) kašnjenje od 100ms još uvijek uglavnom nije primjetno [11]. No, u nekim slučajevima, kada je preciznost izrazito važna, i to ipak pravi veliku razliku. Jedan od primjera je pucanje u metu koja se miče i slično. Kako i to učiniti najpravednijim za sve igrače opisano je u sljedećoj cjelini.

3.7. Kompenzacija kašnjenja

Posljednji u nizu koncepata u igrama u okruženju više igrača je kompenzacija kašnjenja (engl. *Lag compensation*). Razmotrimo sljedeći problem: igrač A mirno stoji i cilja igrača B koji trči prema desno. Igrač A prekrije mišem preko igrača B i puca (pretpostavimo da će ispucati odmah). Poruka koja indicira da igrač puca putuje do poslužitelja npr. 100ms. Za to vrijeme, igrač B se nastavlja kretati prema desno. Poslužitelj prima poruku nakon 100ms i testira koliziju između igrača B i mete igrača A, no ispada da je igrač promašio jer se igrač B nastavio kretati. Iako je igrač A vidio da je igrača B direktno pogodio, nije uzrokovao nikakvu štetu [12].



Slika 3.9: Kompenzacija kašnjenja.

Općenito, poslužitelj vidi akcije svakog igrača sa zakašnjenjem zbog kašnjenja na mreži i igrači završavaju tako da promaše mete u pokretu. Na taj način, da bi igrači pogodili metodu, trebali bi pucati u prazan prostor ispred igrača koji se miče. To nije realistično niti pošteno, a kompenzacija kašnjenja je tehnika kojom se to izbjegava. Pri pucanju, klijent šalje ovaj događaj poslužitelju sa sljedećim informacijama: točni vremenski trenutak pucnja i točnu metu oružja [11]. Zbog toga što poslužitelj dobiva sve akcije s točnim vremenskim trenutkom, može rekonstruirati stanje igre u bilo kojem

trenutku u prošlosti, uključujući i trenutak kada je igrač pucao. To znači da poslužitelj može točno znati u što je igrač u tom trenutku ciljao. To je bila pozicija neprijateljeve glave u prošlosti, ali poslužitelj zna da je za trenutnog igrača to bilo ono što je on vidio u tom trenutku. Poslužitelj procesira pucanj u tom trenutku i ažurira klijente. Na ovaj način igrač više neće promašiti kada ne bi trebao, ali se može dogoditi da se neprijatelj zakloni iza pucnja, a igrač ga vidi u prošlosti (prije zaklona) i neprijatelj strada iako je iz svoje perspektive bio sakriven. To je kompromis, ali i dalje je najbolja alternativa koja se može postići u svijetu u kojem su klijenti desinkronizirani.

4. Implementacija

4.1. Implementacija mrežnog dijela

Nakon upoznavanja s teorijskim konceptima, u sklopu ovog rada implementiran je dio koncepata iz igara u okruženju više igrača. Implementacija je započela sa ostvarenjem grafičkog prikaza i logike igre s jednim igračem (poglavlje 4.1.1) koji se kreće po polju fiksne veličine i može sakupljati hranu čime se povećava . Nakon toga, igra je proširena na dva igrača, ali unutar jednog programa kako bi se implementirala detekcija kolizije (poglavlje 4.3.2). Optimizacija prikaza scene ostvarena je dodavanjem grupnog iscrtavanja (engl. *batch rendering*) što je vezano isključivo uz grafičku komponentu rada i opisano je u poglavlju 4.2.2.

Nakon uspješne mogućnosti rada unutar jednog programa, ostvareno je odvajanje klijenta i poslužitelja u različite izvršne cjeline (poglavlje 4.1.2). Kako bi se ostvarila komunikacija između poslužitelja i klijenata, uspostavljen je format poruka preko kojih međusobno razmjenjuju informacije i napravljena je serijalizacija i deserijalizacija podataka korištenjem knjižnice Boost.Serialization (poglavlje 5.4). Osim formata, osmišljena je glavna petlju autoritativnog poslužitelja i klijenta koji samo prima nova stanja sustava. To je ostvareno uz sinkronu komunikaciju (sve se izvršava slijedno, a pozivi za čitanje i pisanje poruka s mreže su blokirajući, dakle, do kada poruka nije poslana/primljena, ne izvršava se nova naredba). Takav način izvršavanja se pokazao neadekvatnim iz nekoliko razloga koji su opisani kasnije u poglavlju 4.1.3. Asinkrona komunikacija je uvedena kako bi se omogućilo simuliranje kašnjenja od klijenta do poslužitelja i obrnuto. To je napravljeno da bi se mogli demonstrirati različiti scenariji kašnjenja (veliki/mali ping, broj ažuriranja poslužitelja u sekundi,...) u uvjetima koji su laki za reproducirati u bilo kojoj situaciji.

Uz trenutnu arhitekturu klijenta i poslužitelja i dodavanje fiksnog broja ažuriranja poslužitelja u sekundi (engl. *tickrate*), implementirana je i predikcija na strani klijenta (poglavlje 4.1.4). Ostali koncepti su samo teorijski pokriveni.

4.1.1. Igra s jednim igračem

Kao polazna točka implementacije napravljena je igra Agar sa samo jednim igračem (engl. *singleplayer*). Isječak koda 4.1 prikazuje glavnu petlju u takvom slučaju. U svakoj iteraciji igrač očitava proteklo vrijeme od prošle iteracije do sadašnjeg trenutka,

nakon čega se ažurira pozicija igrača na poziv funkcije `on_update(timestep)`. Ta funkcija prima proteklo vrijeme tako da može ažurirati poziciju za pomak koji je proporcionalan upravo tom vremenu. Na taj način se osigurava da se igrač pomakne do istog mjesta u jednakom vremenu bez obzira na brzinu računala na kojem se igra. Primjerice, ako jedan igrač ažurira stanje igre 60 puta u sekundi, onda će se pomaknuti 60 puta po jednu jedinicu udaljenosti, a ako drugi igrač sa sporijim računalom ažurira 5 puta u sekundi, onda će se on svaki put pomaknuti po 12 jedinica udaljenosti, što će ih nakon jedne sekunde smjestiti na istu poziciju. Osim veličine, u sljedećem koraku se ažurira i cijelo stanje igre, a to u ovom jednostavnom slučaju uključuje povećanje igrača i stvaranje nove hrane, dok bi u slučaju više igrača uključivalo i provjeru kolizije, dodavanje inercije igrača i slično.

```
while (running) {
   Timestep timestep = current_timepoint - last_timepoint;

   player_controller.on_update(timestep);
   game.on_update(timestep);
   camera_controller.set_position(player);

   renderer.clear();
   renderer.begin_scene(camera_controller.get_camera());
   renderer.begin_batch();
   game_renderer.render();
   renderer.end_batch();

   window.on_update();
}
```

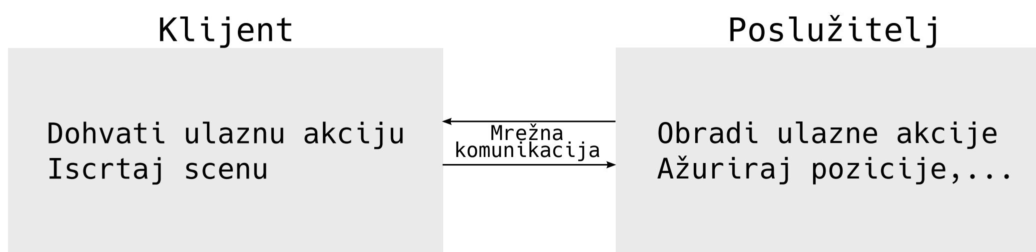
Isječak 4.1: Glavna petlja za singleplayer igru.

U ovoj inačici igre, kao i u pravoj igri Agar, kamera prati centar igrača, pa se tako u sljedećem koraku ažurira njezina pozicija. Kasnije je to uklonjeno zbog demonstracije na različitim klijentima i poslužitelja kako bi se lakše moglo pratiti željeno ponašanje.

Grafički dio obuhvaća brisanje trenutnog prikaza na zaslonu, početak nove scene, grupno iscrtavanje scene na zaslon (`game_renderer.render()`) i na kraju zamjenu međuspremnika (poziv OpenGL funkcije `glfwSwapBuffers()`).

4.1.2. Dodavanje klijent-poslužitelj arhitekture

Kao temelj za izgradnju igre u okruženju više igrača odabrana je klijent-poslužitelj arhitektura. Sada se cijela igra sastoji od proizvoljnog broja klijenata i jednog poslužitelja kao što je prikazano u isječcima koda 4.2 i 4.3.



Slika 4.1: Komunikacija klijenta i autoritativnog poslužitelja.

Klijentski dio arhitekture

Klijent u svojem tijeku programa prvo uspostavlja vezu s poslužiteljem na zadanoj IP adresi i portu. U ovom slučaju korištena je lokalna IP adresa (127.0.0.1), odnosno, poslužitelj i klijent se nalaze na istom računalu, no to ne mijenja implementaciju. Prva poruka koju klijent šalje nakon spajanja je njegovo ime (`player_name`) kako bi poslužitelj to mogao spremiti i koristiti kasnije kroz igru.

U svakoj iteraciji petlje klijent dohvaća trenutnu naredbu za pomak (`movement command`). Ona je u ovom slučaju samo dvodimenzionalni vektor (x, y) koji predstavlja smjer kretanja igrača. Ako se koristi tipkovnica onda je to za pomak u desno $(+1, 0)$, a kod miša je to bilo koja vrijednost iz skupa [-1, 1]x[-1, 1]. Ta vrijednost se prije slanja treba serijalizirati u tekst (`string`) kako bi se poslala u odgovarajućem formatu. Nakon toga klijent čeka da mu poslužitelj pošalje novo stanje igre, deserijalizira ga i sprema. Na temelju nove pozicije postavlja i poziciju kamere. Posljednji korak je iscrtavanje cijele scene što uključuje prikaz svih igrača, hrane, trenutnog broja bodova i slično.

```
connection = connect(io_context, ip_address, port);
connection->write(player_name);
while (running) {
   auto movement_command = get_movement_command();
   connection->write(Serialization::serialize(movement_command));
   game->state =
      Serialization::deserialize<Game::State>(connection->read());

   camera_controller.set_position(
      game->state.get_player(configuration.player_name).center);

   game_renderer->draw_scene(
      camera_controller, window, configuration.player_name);
   window.on_update(); // Poll events.
}
```

Isječak 4.2: Glavna petlja klijenta.

U ovoj petlji se možemo uvjeriti da se radi o klijentu koji samo šalje ulazne podatke, prima novo stanje i iscrtava scenu na zaslon (kao što je ilustrarirano na slici 4.1). Dakle, radi se o potpuno autoritativnom poslužitelju.

Poslužiteljski dio arhitekture

Na poslužiteljskoj strani se na početku uspostavlja veza s unaprijed određenim brojem klijenata (funkcija `accept` u isječku koda A.3). S uspješnim završetkom uspostavljanja veze kreće glavna petlja poslužitelja (isječak koda 4.3). U svakoj iteraciji poslužitelj računa proteklo vrijeme od prošle iteracije, prima i deserijalizira ulaznu akciju za svakog klijenta i na temelju toga računa novu poziciju. Sljedeći je korak ažurirati globalno stanje igre, što uključuje generiranje nove hrane, detekciju kolizije, računanje nove veličine igrača i dodavanje inercije. Zadnji je korak to stanje serijalizirati i putem mreže ga poslati klijentu kako bi mogao na svojoj strani ažurirati cijelu scenu. U komercijalnim igrama ovo stanje bi se dodatno kompresiralo kako bi se smanjila količina podataka koja se šalje preko mreže.

```
while (running) {
   Timestep timestep = current_timepoint - last_frame_timepoint;

   // Receive movement command from each player and update position.
   for (auto& [name, connection] : connections) {
      auto movement_command =
         Serialization::deserialize<glm::vec2>(connection->read());
      game->update_player_position(name, timestep, movement_command);
   }

   game->on_update_state(); // Update global game state.

   for (auto& [name, connection] : connections) { // Send new game state.
      connection->write(Serialization::serialize(game->state));
   }
}
```

Isječak 4.3: Glavna petlja poslužitelja.

4.1.3. Asinkrona inačica igre

Iako prethodno opisane glavne petlje poslužitelja i klijenta vrlo jasno prikazuju rad igre, one same po sebi nisu dovoljne da osiguraju kontinuirano slanje podataka od klijenta do poslužitelja i obrnuto. Način rada koji je korišten do sada se naziva sinkronim, a u ovom poglavlju će biti zamijenjen asinkronom inačicom.

Asinkrona komunikacija je način komunikacije u kojoj podaci mogu biti poslani u bilo kojem trenutku, bez obzira je li primatelj spreman primiti ih ili ne. Primatelj se pretplaćuje na neku određenu operaciju, primjerice, na operaciju čitanja i kada su podaci spremni za čitanje, poziva se funkcija (`callback`) u kojoj se izvršavaju željene akcije (obrada/spremanje pristignutih podataka,...). Za uvođenje asinkrone komunikacije postoji nekoliko razloga. Glavni razlog je to što uz sinkronu komunikaciju ne postoji mogućnost da poslužitelj prima više ulaznih akcija od jednog klijenta, nego svaki puta uzima po jednu akciju od svakog klijenta, primjeni je na igrača i ide na sljedećeg. No, što ako jedan igrač ima lošu vezu i nekoliko sekundi ne šalje nikakve ulazne podatke? To znači da će cijela igra stati na strani poslužitelja sve dok najsporiji igrač ne uspije poslati ono što bi trebao. To, dakako, nije prihvatljivo i takvu igru nije moguće igrati.

Dodavanjem asinkronih operacija čitanja i pisanja omogućeno je da se svi klijenti i poslužitelj registriraju na svoju operaciju (čitanja, pisanja ili isteka brojila (engl. *Timer*)). U trenutku kada su podaci pristigli (ili su poslani), poziva se odgovarajuća metoda kao što je prikazano u isječku koda 4.8. Nakon obrade, registrira se iduće čitanje korištenjem funkcije `async_read`. Za sve asinkrone metode korišten je prefiks `async` kako bi se razlikovale od sinkronih inačica istih.

```
void Server::on_read_movement_command(MovementCommand mc,
                                      std::string name)
{
    movement_commands[name].push_back(mc);
    // Register new read operation.
    connections[name]->async_read(
        [this, name = name](std::shared_ptr<Connection> /*connection*/,
                            std::string message) {
            on_read_movement_command(
                Serialization::deserialize<MovementCommand>(message), name);
        });
}
```

Isječak 4.4: Funkcija koja se poziva nakon operacije čitanja.

Asinkroni poslužitelj

Prelaskom u asinkroni način rada, gubi se eksplicitna petlja koja je prikazivala rad poslužitelja, no, njegova struktura se i dalje može iščitati iz funkcije `game_step` koja prikazuje jedan korak igre. Početno se provjerava je li igra završena (prozor igre je zatvoren) i u potvrdnom slučaju se sve zaustavlja i završava s radom. Nakon toga se prvo ažuriraju pozicije svih igrača, a zatim i cijelo stanje igre. Pozicija se ažurira na

temelju svih ulaznih akcija koje je klijent poslao do trenutka ažuriranja (interno se spremaju u listu ulaznih akcija koja se prazni nakon što se izvedu sva ažuriranja).

```
void Server::game_step()
{
    if (!running) { stop_game(); }
    update_player_positions();

    // Check if some player has been eaten, update player size,
    // create new food instances etc.
    game->on_update_state();

    send_game_state(); // Send updated game state to all clients.

    game_renderer->draw_scene(camera_controller, window);
    window.on_update();
}
```

Isječak 4.5: Korak igre na poslužitelju

U funkciji `send_game_state` se svim klijentima šalje ažurirano stanje igre, a na kraju se iscrtava scena i izvršava zamjena međuspremnika. Nakon što završi jedan korak igre, ponovno se aktivira brojilo (isječak koda 4.9 čijim se istekom ponovno poziva `game_step`). Upravo na taj način se ponovno postiže učinak petlje koja se izvršava u pravilnim vremenskim razmacima.

Asinkroni klijent

Kao i kod asinkronog poslužitelja, i kod klijenta se izvršava korak igre, nakon čega se registrira operacija `post` koja ponovno poziva korak petlje. Pozivom te operacije, poziva se idući korak igre bez ikakve petlje. Konceptualno, `post` ustvari predstavlja brojilo koje ističe odmah.

```
void Client::game_step() {
    auto movement_command = get_movement_command();
    send_movement_command(movement_command); // Async operation.
    if (game_renderer->prediction_enabled()) {
        predict_new_state(movement_command);
    }
    window.poll_events();
    render_scene();
    // Register next game step operation.
    boost::asio::post(io_context, [this]() { game_step(); });
```

```
}
```

Isječak 4.6: Glavna petlja za singleplayer igru.

Osim koraka igre, važno je za istaknuti i rad funkcije `send_movement_command` u kojoj je simulirano kašnjenje slanja između klijenta i poslužitelja (isječak koda 4.7). Kašnjenje se također simulira korištenjem brojila. Tek istekom `delay` ms (*ping*/2) vremena, poziva se funkcija koja poslužitelju zapisuje novu ulaznu akciju. Kod asinkronih operacija, potrebno je osigurati da se ne registrira više operacija pisanja (ili čitanja) prije nego je jedan `handler` obrađen, pa se dodatno uvodi red podataka u koji se spremaju sve ulazne akcije koje se slijedno šalju prema poslužitelju.

```
void Client::send_movement_command(MovementCommand movement_command)
{
   // Send mouse position to the server with delay (simulates lag).
   auto timer = std::make_unique<Timer>(io_context.get_executor());
   timer->expires_after(delay_ms);
   timer->async_wait([this, movement_command, timer = std::move(timer)](
                    const ErrorCode& error) {
      if (error) { throw std::runtime_error(error.message()); }
      response_queue.register_write(
         Serialization::serialize(movement_command));
   });
}
```

Isječak 4.7: Slanje ulazne akcije uz simuliranje kašnjenja.

Kao što je prikazano u isječku koda 4.6 u liniji 5, sljedeća implementirana funkcionalnost je predikcija na strani klijenta.

4.1.4. Predikcija na strani klijenta

Nakon uvođenja predikcije na strani klijenta, nije bilo dovoljno da klijent čuva samo stanje igre nego mu je na raspolaganje bilo potrebno staviti i metode za ažuriranje tog stanja. U ovom slučaju tu se radi samo o ažuriranju pozicije i veličine nakon što je hrana pojedena jer detekciju kolizije treba potvrditi poslužitelj.

```
void Client::predict_new_state(MovementCommand movement_command)
{
   game->update_player_position(
      configuration.player_name,
      Timestep(std::chrono::milliseconds(1000 / polling_rate)),
      movement_command);
}
```

Isječak 4.8: Glavna petlja za singleplayer igru.

4.1.5. Vremenski korak poslužitelja

U igrama se stanje na poslužitelju najčešće ažurira fiksni broj puta u sekundi (`tickrate`). Na taj način je ostvareni i asinkroni poslužitelj gdje se korak petlje poziva nakon što istekne brojilo.

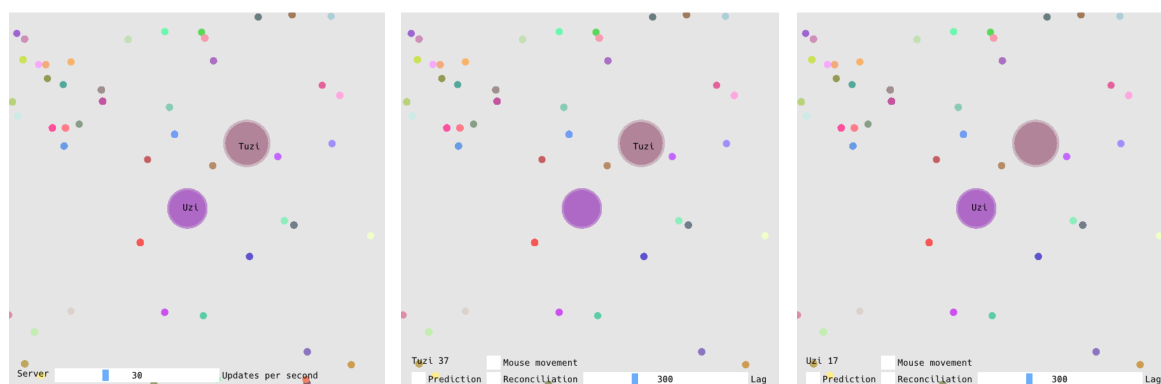
```
void Server::on_timer_expired()
{
    game_step();

    update_timer.expires_after(1 / tickrate);
    update_timer.async_wait([this](const ErrorCode& error) {
        if (error) { throw std::runtime_error(error.message()); }
        on_timer_expired();
    });
}
```

Isječak 4.9: Glavna petlja za singleplayer igru.

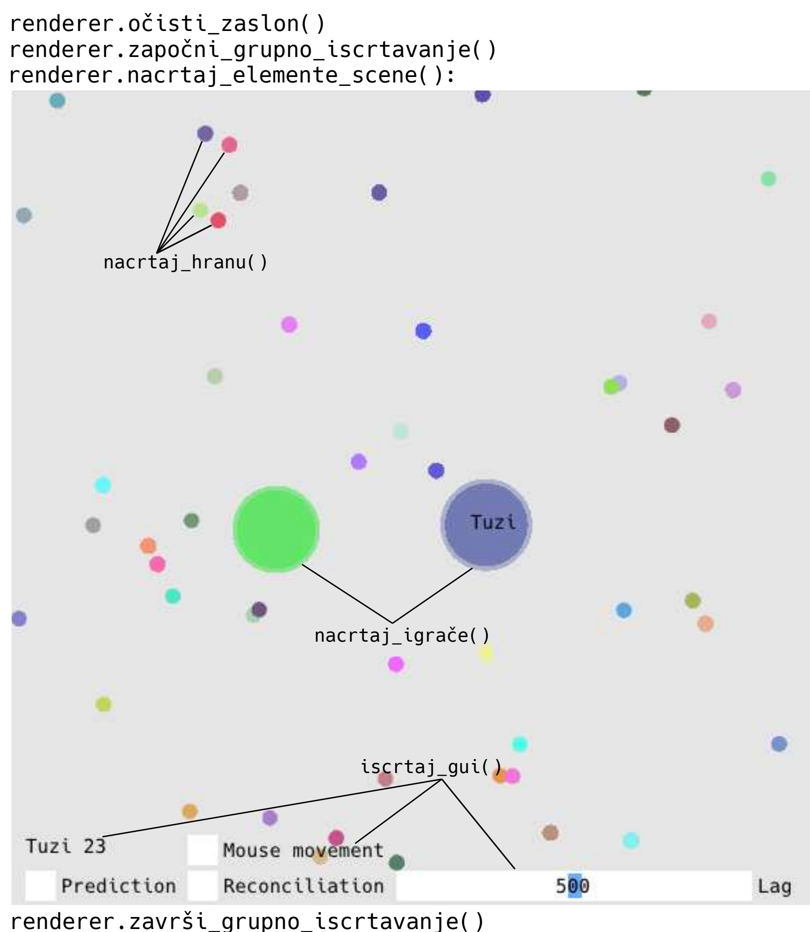
4.2. Implementacija grafičkog dijela

4.2.1. Grafički prikaz scene



Slika 4.2: Isječak iz igre Agar

Slika 4.2 prikazuje zaslon koji se pojavljuje pri pokretanju igre s dva klijenta i poslužiteljem. Vidljivo je da se otvara tri različita prozora (i različita programa) koji nude jednostavan prikaz igre Agar. Poslužitelj nudi grafički prikaz svojeg stanja iskjučivo kako bi bilo moguće demonstrirati prethodno navedene koncepte. Scena se sastoji od niza krugova koji predstavljaju igrače i hranu, te grafičkog sučelja za upravljanje postavkama igre.



Slika 4.3: Isječak iz igre Agar

Ilustracija 4.3 prikazuje rad `renderer`-a. U svakoj iteraciji prikaza, prvo se zaslon čisti od elemenata iz prošle scene, zatim se pripremaju postavke za grupno iscrtavanje i na kraju se iscrtavaju svi elementi scene.

4.2.2. Grupno iscrtavanje

Grupno iscrtavanje je način iscrtavanja u kojem se grupira više poziva za crtanje u jedan veći poziv jer je ponovni poziv za svaki objekt scene izrazito neučinkovit. Na taj način se efikasnije može iskoristi grafička kartica, ali je za jedan objekt sada potrebno više memorije (sprema se više podataka o tom objektu).

U početnoj inačici igre je za crtanje svakog elementa bio pozivan zaseban OpenGl poziv. Takav pristup se pokazao neefikasnim. Ilustrativno, na prikazu 6400 kvadrata je s grupom veličine 10000 u odnosu na grupu veličine 1 postignuto ubrzanje od 20 puta, a to ubrzanje se samo povećava kako se povećava i broj poziva za crtanje. Grupno prikazivanje je implementirano premještanjem dodatnih informacija iz uniform-a u spremnik vrhova (engl. *vertex buffer*). Na ovaj način se spremaju informacije o objektima u međuspremnik koji se tada prazni pri pozivu koji dolazi kada se međuspremnik napuni ili kada su pohranjeni svi objekti scene. Pri tome je potrebno

grupirati samo one objekte koji koriste iste sjenčare.

4.2.3. Kamera

Kao što je ostvareno pokretanje igrača kroz scenu, tako je i implementirana mogućnost kretanja, rotacije i približavanja/udaljavanja kamere od neke točke. Isječak koda 4.10 prikazuje potrebne transformacije MVP matrice kako bi se postigao tak efekt. U ovom slučaju se na jediničnu matricu dodaje trenutna pozicija kamere, zatim se odvija operacija rotacije za zadani broj radijana, to se invertira i na kraju množi s projekcijskom matricom.

```
void OrthographicCamera::set_rotation(float rotation) noexcept
{
   this->rotation = rotation;

   glm::mat4 transform = glm::translate(glm::mat4(1.0f), position) *
                    glm::rotate(glm::mat4(1.0f),
                             glm::radians(rotation),
                             glm::vec3(0.0f, 0.0f, 1.0f));

   view_matrix = glm::inverse(transform);
   view_projection_matrix = projection_matrix * view_matrix;
}
```

Isječak 4.10: Rotacija kamere.

Kretanje kamere kroz scenu je ostvareno na sličan način kao i kretanje igrača, korištenjem strelica i proteklog vremena od prošle kretnje.

4.3. Logika igre

4.3.1. Stanje igre

Stanje igre se sastoji od proizvoljnog broja igrača gdje je svaki opisan značajkama iz tablice, fiksne količine instanci hrane i granica mape. Za dohvaćanje pojedinog igrača prema imenu, poslužitelj može koristiti metodu `get_player(name)`. I klijent i poslužitelj čuvaju svoje stanje igre koje se razlikuje zbog kašnjenja na mreži. Ažuriranje stanja se dijeli na dva dijela. Prvo se ažuriraju sve pozicije, nakon čega se detektira kolizija, ažurira veličina i stvara nova količina hrane (kako bi uvijek bila konstantna količina u sustavu).

Tablica 4.1: Značajke igrača

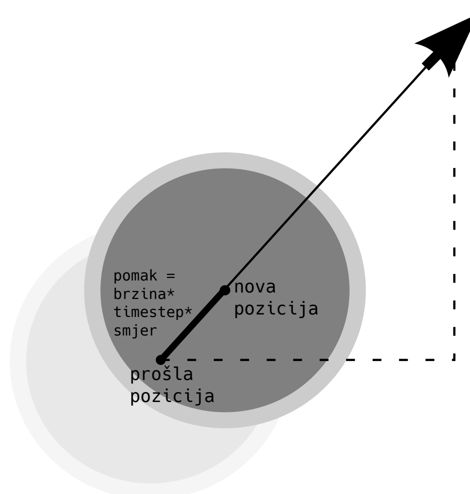
name	Ime igrača.
center	Središte igrača.
radius	Radijus igrača.
color	Boja igrača.
score	Trenutni rezultat.
speed	Trenutna brzina.
nEaten	Broj pokupljene hrane u prethodnoj iteraciji.

Ažuriranje pozicije igrača

Prema isječku koda 4.11 i slici 4.4 vidljiva je formula po kojoj se ažurira pozicija svakog igrača. Dodatno se vrši provjera prelazi li igrač nakon pomaka izvan mape, i ako je to slučaj, stavlja ga se na granicu mape.

```
void PlayerController::on_update(Timestep timestep,
                                  glm::vec2 movement_command) noexcept
{
    glm::vec2 delta = player.speed * timestep * movement_command;
    move_player(delta);
}
```

Isječak 4.11: Ažuriranje pozicije igrača.



Slika 4.4: Pomak igrača.

Pozicija igrača ovisi o tri komponente, a to su: brzina igrača, proteklo vrijeme i smjer kretanja igrača. Smjer je predstavljen dvodimenzionalnim vektorom čije komponente su u rasponu od -1 do 1. Kao što je ranije opisano u poglavlju 3.4,

poslužitelj prima od klijenta ulazne akcije sve dok ne prođe $1/tickrate$ vremena i tek nakon toga primjenjuje niz primljenih akcija na stanje igre. U ostvarenom programskom rješenja u sklopu rada se za svakog igrača taj niz akcija izvršava jedan za drugim i kada su sve akcije izvršene, ide se na idućeg igrača. Tu valja opaziti da onaj igrač čije se akcije izvršavaju prve ima prednost jer se pozicija drugih igrača još nije ažurirala iako su se u stvarnosti oni možda ranije pomaknuli. Ovaj nedostatak bi se mogao smanjiti tako da se naizmjence primjenjuju akcije igrača, no to nije implementirano u sklopu rada.

Ažuriranje veličine igrača

Nakon što je određena nova pozicija svih igrača, određuje se i njihova veličina. Veličina ovisi o broju sakupljene hrane i broju sakupljenih igrača. Isječak koda 4.12 prikazuje ažuriranje veličine za iznos `increase_step` na temelju pokupljene hrane, a ostatak je opisan u poglavlju 4.3.2.

```
// Increase player size.
for (auto& player : players) {
   player.radius += player.n_eaten * increase_step;
}
```

Isječak 4.12: Inicijalizacija modernog OpenGL-a.

Ažuriranje brzine igrača

U ovisnosti o veličini igrača, mijenja se i brzina kako bi se manji igrači mogli lakše skloniti od većih. Smanjenje brzine odvija se tako da se trenutna brzina podijeli s faktorom `decrease_step`. Taj faktor se ipak smanjuje kako veličina igrača raste kako se nebi dogodilo da veliki igrači postanu toliko spori da se skoro uopće i ne miču. Takvo dodavanje inercije je opisano u isječku koda 4.13.

```
// Decrease player size.
for (auto& player : players) {
   if (player.score >= max_limit) {
      max_limit *= 2;
      decrease_step /= 2;
   } else {
      player.speed *= decrease_step;
   }
}
```

Isječak 4.13: Dodavanje inercije.

4.3.2. Detekcija kolizije

Kako bi se uspješno detektiralo koji je igrač u koliziji s kojim, provjerava se dva uvjeta: je li igrač veći od protivnika i nalazi li se unutar njega s barem 50% svoje površine. Ako je to slučaj, igrač se povećava za određeni iznos, a protivnik se ponovno stvara na nekom nasumičnom mjestu unutar granica mape s inicijalnim rezultatom (isječak koda 4.14).

```
// Check if some player has been eaten.
for (auto& player : state.players) {
    for (auto& opponent : state.players) {
        if (player.bigger_than(opponent) && opponent.is_inside(player)) {
            player.radius += opponent.score * increase_step;
            // Spawn opponent on random position inside borders.
            opponent = createPlayer(opponent.name, opponent.color);
        }
    }
}
```

Isječak 4.14: Inicijalizacija modernog OpenGL-a.

4.3.3. Generiranje hrane

U igri se nalazi konstantna količina hrane početno definirana parametrom `n_instances`. U svakoj iteraciji poslužitelj generira sakupljeno hranu kao što je prikazano u liniji 3 isječka 4.15.

```
void Food::create_new_instances() noexcept
{
    std::size_t n_new = n_instances - food_instances.size();
    for (std::size_t i = 0; i < n_new; ++i) {
        food_instances.push_back(
            {get_position(playable_range, radius), get_color()});
    }
}
```

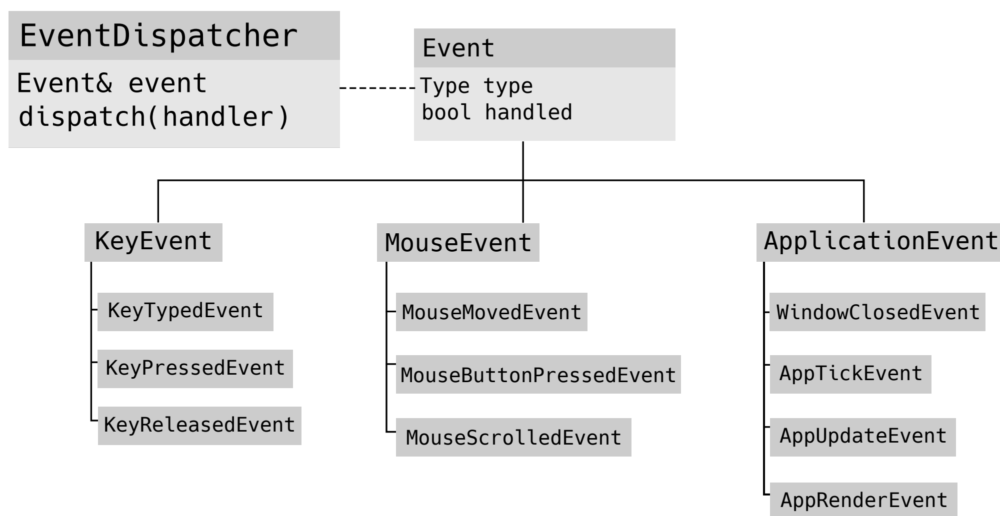
Isječak 4.15: Generiranje nove hrane.

4.3.4. Sustav događaja

Osnovna ideja sustava događaja je da, umjesto da svaka komponenta komunicira međusobno, svi šalju informacije prema sustavu događaja koji zatim to dostavlja primateljima. Komponente (npr. poslužitelj ili klijent) se registriraju na sustav događaja (s povratnom funkcijom (engl. *callback*)) [13]. Komponente dodaju obavijesti u sustav događaja. U implementaciji je ostvareno da je svaki događaj blokirajući (čeka

se da se obradi prije nego može doći novi), no to bi bilo bolje ostvariti strukturom podataka poput reda (engl. *queue*).

Slika 4.5 prikazuje pojednostavljenu verziju dijagrama razreda u sustavu događaja. Svaki događaj ima svoj tip i informaciju o tome je li obrađen, a ovisno o kojoj vrsti događaja se radi, događaji poprimaju tip (tipkovnica, miš, prozor). Također, u sustavu postoji jedan objekt koji je zadužen za raspodjelu događaja komponentama koje su se pretplatile na njega. Glavna metoda u kojoj se to radi je prikazana u isječku koda 4.16.



Slika 4.5: Prikaz strukture sustava događaja.

```
template<typename EventType>
bool EventDispatcher::dispatch(std::function<bool(EventType&)> handler)
{
   // If event is of requested type, safely static cast the event to the
   // requested type (without dynamic cast) and call the handler.
   if (event.get_type() == EventType::get_static_type()) {
      ASSERT(dynamic_cast<EventType*>(&event),
           "Event types match, but the dynamic cast has failed.");
      EventType& e = *(static_cast<EventType*>(&event));
      event.handled = handler(e);
      return true;
   }
   return false;
}
```

Isječak 4.16: Poziv funkcije za obradu kod slanja događaja.

5. Korištene tehnologije i alati

Program je implementiran u programskom jeziku C++ korištenjem programskog sučelja OpenGL inačice 4.6 i knjižnice za mrežnu komunikaciju Bost.Asio inačice 1.76. Za sjenčanje na grafičkoj kartici korišten je jezik visoke razine apstrakcije GLSL (OpenGL Shading Language). GLSL je jezik sličan C-u koji pruža više izravne kontrole nad grafičkim cjevovodom bez potrebe za korištenjem asemblera.

Kako bi bilo omogućeno jednostavno korištenje modernih OpenGL funkcija koje su implementirane u grafičkim upravljačkim programima (engl. *driver*) u projekt je uključena i knjižnica samo s datotekama zaglavlja GLEW (OpenGL Extension Wrangler Library) bez koje bi bilo potrebno ručno provjeravati koja je verzija OpenGL-a podržana na pojedinom hardveru, koje su funkcije dostupne, te za svaku funkciju koju će biti korištena ručno inicijalizirati pokazivač. GLEW ovaj cijeli postupak pojednostavljuje na način da se uključi jedno zaglavlje te pozove jedna funkcija tijekom inicijalizacije prikazana u isječku koda 5.1.

Za jednostavnije stvaranje i manipulaciju prozorom, stvaranje konteksta, te rukovanje ulaznim uređajima (poput tipkovnice i miša) korištena je knjižnica GLFW 3.3. (Graphics Library Framework) koja za razliku od starije alternative GLUT omogućava veću kontrolu rada. GLFW ostavlja programeru da implementira vlastitu glavnu petlju (engl. *event loop*) što omogućava mnogo precizniji rad s vremenom i samim time i manja kašnjenja.

Prednost navedenih knjižnica je to što su višeplatformske knjižnice otvorenog koda.

Rad s vektorima olakšan je uporabom matematičke knjižnice GLM za C++ čija je sintaksa vrlo slična sintaksi GLSL-a. GLM je posebno prikladan za uporabu s OpenGL-om zato što koriste isti način pohranjivanja matrica u memoriju, odnosno pohranjuju matrice kao vektor stupce. To je pogodno iz dva razloga, glavni razlog su performanse, a drugi je praktičnost jer programer ne treba razmišljati je li unio matricu u formatu koji odgovara OpenGL-u kao što bi to bio slučaj kada bi matrice bile pohranjene u različitim formatima.

```
if (glewInit() != GLEW_OK) {
    throw std::runtime_error("Failed to initialize GLEW.");
}
```

Isječak 5.1: Inicijalizacija modernog OpenGL-a.

5.1. Korištenje programa

Parametri programa podešavaju se korištenjem grafičkog sučelja i konfiguracijske datoteke. Program je trenutno isproban samo u Linux okruženju, a pokreće se uz sljedeće naredbe: meson build, cd build, ninja run.

5.2. OpenGL

OpenGL (Open Graphics Library) je višeplatformska specifikacija koja je podržana u velikom broju programskih jezika, a služi za prikaz 2D i 3D scena [14]. Iako bi bilo moguće implementirati sve funkcionalnosti programski, danas su gotovo sve implementirane u hardveru kako bi se rasteretio procesor i kako bi se iskoristila paralelna moć grafičkih kartica. U nastavku ovog poglavlja će biti opisani neki osnovni koncepti s kojima je potrebno biti upoznat u svrhu izrade scena u modernom OpenGL-u inačice 3.3 na više.

5.2.1. Vertex Buffer Object (VBO)

`Vertex Buffer Object` je memorijski međuspremnik u kojem su spremljeni atributi poput koordinata točaka, vektora normala, boje i koordinata tekstura. Atributi su spremljeni u radnoj video memoriji i za razliku od OpenGL-a inačice 1.x koji iscrtava točke između poziva funkcija `glBegin()` i `glEnd()` neposredno nakon svakog slanja točke na grafičku karticu, `VBO` omogućava iscrtavanje cijelog spremnika nakon poziva odgovarajuće OpenGL funkcije za crtanje. Prednost `VBO`-a u odnosu na stariju alternativu je to što se podaci u memoriji čuvaju na način koji je optimalan za grafičku karticu.

5.2.2. Index Buffer Object (IBO)

`Index Buffer Object` omogućava ponovno korištenje postojećih definicija vrhova. Primjerice, ako neki program treba iscrtati kvadrat koji se sastoji od dva trokuta, bez `IBO`-a je potrebno `VBO`-u proslijediti šest vrhova iako je dovoljno samo četiri kako bi kvadrat bio u potpunosti definiran. Indeks zapravo predstavlja redni broj pozicije vrha iz `VBO`-a koju je potrebno nacrtati.

5.2.3. Vertex Array Object (VAO)

`Vertex Array Object` je poseban tip objekta koji enkapsulira sve podatke koji su povezani s procesiranjem vrhova. Umjesto da sadrži stvarne podatke, `VAO` sadrži reference na jedan ili više `VBO`-a i točno jedan `IBO` i raspored svakog vrha (u smislu redoslijeda pojavljivanja atributa, primjerice: u ovom `VBO`-u se nalazi tri broja s pomičnim zarezom koji definiraju poziciju i tri broja koja definiraju boju). Odnosno,

VAO omogućava da se VBO-u jednom dodijeli njegov raspored atributa i zatim korisnik pri korištenju VBO-a više ne treba svaki put ponovno dodjeljivati taj isti raspored (isječak koda 5.2).

```
glVertexAttribPointer(i,
                     element.count,
                     element.type,
                     element.normalized ? GL_TRUE : GL_FALSE,
                     layout.get_stride(),
                     reinterpret_cast<const void*>(offset));
```

Isječak 5.2: Dodjela rasporeda atributa VAO-u.

5.2.4. Sjenčar vrhova

Sjenčar vrhova je program koji se izvršava na grafičkoj kartici za svaki vrh definiran VBO-om i pripadnim IBO-om. Njegova svrha je transformirati svaku 3D poziciju vrha u 2D prostor koji se prikazuje na ekranu i ovisno o načinu transformacije uzeti u obzir Z-spremnik. To se najčešće radi uz pomoć *model-pogled-projekcija* (MVP) matrice koja je definirana preko *uniforma*. *Uniform* je globalna varijabla koju dijele svi vrhovi.

Sjenčar vrhova može manipulirati i bojom, koordinatama teksture i normalama i zbog toga pruža veliku kontrolu nad detaljima osvjetljenja, kretanja i boje u sceni.

```
#version 460 core

layout(location = 0) in vec4 position;
uniform mat4 MVP;

void main()
{
    gl_Position = MVP * position;
}
```

Isječak 5.3: Primjer jednostavnog sjenčara vrhova.

5.2.5. Sjenčar fragmenata

Sjenčar fragmenata je program koji se izvršava na grafičkoj kartici za svaki slikovni element u sceni. Kao ulazni podatak prima jedan slikovni element i promijenjeni element vraća na izlaz. Takav program računa boju i ostale atribute. Sjenčar fragmenata nije obavezan dio grafičkog cjevovoda i ako nije korišten boja elementa poprima nedefiniranu vrijednost.

```
#version 460 core
```

```
layout(location = 0) out vec3 out_color;
uniform vec3 color;

void main()
{
   out_color = color;
}
```

Isječak 5.4: Primjer jednostavnog sjenčara fragmenata.

5.3. ImGui

ImGui je prenosiva knjižnica za ostvarenje grafičkog sučelja napisana u programskom jeziku C++ [15]. ImGui je dizajniran tako da omogućuje brze izmjene i postizanje željenog cilja. Posebno je prikladan za integraciju s raznim okruženjima za razvoj igara (engl. *Game Engine*) (prikaz alata), 3D aplikacije u stvarnom vremenu, ugradbene aplikacije i slično.

5.4. Serijalizacija i deserijalizacija

Serijalizacija poruka napravljena je korištenjem knjižnice Boost.Serialization. Ta knjižnica omogućuje pretvorbu C++ objekata u niz okteta koji mogu biti spremljeni. Isječak koda 5.5 i A.2 prikazuju primjer funkcije koju treba implementirati kako bi deserijalizacija bila moguća i serijalizacije konkretnog objekta (tipa `Player`).

```
template<typename T>
T deserialize(const std::string& serialized_data)
{
   static_assert(std::is_default_constructible_v<T>,
              "Deserialization type must be default constructible.");
   T data;
   {
      std::istringstream archive_stream(serialized_data);
      boost::archive::binary_iarchive archive(archive_stream);
      archive >> data;
   }
   return data;
}
```

Isječak 5.5: Deserijalizacija.

5.5. Boost.Asio

Boost.Asio je višeplatformska knjižnica napisana u programskom jeziku C++ koja služi za mrežnu komunikaciju i I/O programiranje niske razine koji programerima pruža konzistentni asinkroni model [16]. Većina programa je na neki način u interakciji s vanjskim svijetom (preko datoteka, mreže, serijskog kabla, konzole,...). Ponekad, kao kod mrežne komunikacije, pojedina I/O operacija može trajati dugo prije nego završi. To stvara posebne probleme pri razvoju. Boost.Asio pruža alate koji upravljaju operacijama koje se dugo izvršavaju bez da programi trebaju koristiti višedretveni model uz eksplicitni mehanizam zaključavanja.

Boost.Asio ima nekoliko glavnih ciljeva [16]:

– Portabilnost
– Skalabilnost
– Efikasnost
– Lako korištenje
– Baza za daljnju apstrakciju

Boost.Asio može izvršavati sinkrone i asinkrone operacije na I/O objektima poput socket-a. U sklopu ovog rada se za operaciju povezivanja na poslužitelj, prihvaćanja zahtjeva za povezivanjem od klijenta i inicijalno razmjenjivanje početnog stanja igre koriste sinkrone operacije, dok se za razmjenjivanje poruka (čitanje i pisanje na mrežu i brojila koriste asinkrone operacije.

5.5.1. Povezivanje i prihvaćanje veze

Klijent se povezuje na poslužitelja korištenjem metode `connect` kojoj predaje IP adresu i port na koje se želi spojiti.

```
socket.connect(
   Endpoint(boost::asio::ip::make_address(ip_address), port));
```

Isječak 5.6: Povezivanje klijenta s poslužiteljem

A poslužitelj prihvaća klijentu vezu korištenjem metode `accept` kao što je prikazano u isječku koda 5.7.

```
Acceptor acceptor(io_context.get_executor(),
             Endpoint(boost::asio::ip::tcp::v4(), port));
auto connection =
   std::make_shared<Connection>(io_context.get_executor());
acceptor.accept(connection->get_socket());
```

Isječak 5.7: Prihvaćanje zahtjeva za vezom od klijenta

5.5.2. Pisanje i čitanje s mreže

Boost.Asio funkcija za pisanje podataka na mrežu prikazana je u isječku koda 5.8. Ta funkcija prima socket, spremnik koje treba zapisati i funkciju koja će se pozvati nakon zapisivanja (`handler`).

```
boost::asio::async_write(
   socket,
   buffers,
   [self, handler = std::move(handler), bytes_to_write](
      const ErrorCode& e, std::size_t written) {
      if (written != bytes_to_write) {
         throw std::runtime_error(e.message());
      }
      if (e) {
         throw std::runtime_error(e.message());
      }
      handler(self);
   });
```

Isječak 5.8: Prihvaćanje zahtjeva za vezom od klijenta

5.5.3. Timeri

I/O operacije koje dugo traju često imaju rok do kojeg moraju biti završene. Ti rokovi mogu biti izraženi preko apsolutnih vremena, ali se često računaju i relativno u odnosu na trenutno vrijeme. Primjer korištenja timera u asinkronoj operaciji čekanja prikazan je u isječku koda 5.9. Nakon što prođe vrijeme definirano u funkciji `expires_from_now`, aktivirat će se brojilo i pozvat će se funkcija `handler`.

```
void handler(boost::system::error_code ec) { ... }
...
io_context i;
...
deadline_timer t(i);
t.expires_from_now(std::chrono::time::milliseconds(400));
t.async_wait(handler);
...
i.run();
```

Isječak 5.9: Timer

Osim brojila, kroz rad je korištena i primitiva koja se naziva `post`, a koja je po svojem ponašanju jednaka pozivanju brojila koji ističe odmah pri aktivaciji. Ta primitiva se koristi kod iscrtavanja klijentovog prikaza na zaslon.

Računalna igra Agar u okruženju više igrača

Sažetak

U sklopu rada su demonstrirani koncepti u računalnim igrama u okruženju više igrača na pojednostavljenom primjeru popularne igre Agar. Igra je ostvarena uz klijent-poslužitelj arhitekturu i predikciju na strani klijenta. Dodatno su opisani i sljedeći koncepti: usuglašavanje, interpolacija, mrtvi obračun i kompenzacija kašnjenja. Za grafički prikaz je korišten OpenGL, a komunikacija između klijenta i poslužitelja je omogućena programskom knjižnicom Boost Asio.

Ključne riječi: igre u okruženju više igrača, klijent-poslužitelj arhitektura, predikcija na strani klijenta, usuglašavanje, kompenzacija kašnjenja, interpolacija, mrtvi obračun, OpenGL, Boost Asio, sustav događaja

Multiplayer computer game Agar

Abstract

Multiplayer game concepts have been demonstrated on a simplified version of the popular browser game Agar. This version of the game is implemented using the client-server architecture and the client-side prediction. Additionally, the techniques of reconciliation, entity interpolation, dead reckoning and lag compensation have been described. Presented implementation leverages OpenGL for displaying graphics and Boost Asio library to facilitate communication between game clients and the server.

Keywords: multiplayer games, client-server architecture, client-side prediction, reconciliation, lag compensation, entity interpolation, dead reckoning, OpenGL, Boost Asio, event system

Literatura

[1] **Povijest online igara u okruženju više igrača**, URL: https://gamebin.tv/the-gaming-industry/history-of-online-multiplayer-games/.

[2] **Maze War**, URL: https://www.dungeoncrawlers.org/game/maze-war/.

[3] Gaffer on games, **Što bi svaki programer trebao znati o komunikaciji u igrama**, URL: https://www.gafferongames.com/post/what%5C_every%5C_programmer%5C_needs%5C_to%5C_know%5C_about%5C_game%5C_networking/.

[4] **Agar**, URL: http://arrow-io.com/io-games/agar-io.html.

[5] **Transportni sloj**, URL: https://www.guru99.com/difference-tcp-ip-vs-osi-model.html.

[6] Valve, **UDP vs TCP**, URL: https://www.gafferongames.com/post/udp_vs_tcp/.

[7] **Mrežna komunikacija u Unityju**, URL: https://docs.unity3d.com/Manual/UNetGettingStarted.html.

[8] Valve, **Peer to peer mreža**, URL: https://techround.co.uk/tech/online-peer-to-peer-gaming/.

[9] **Mrežna komunikacija u igrama**, URL: https://ruoyusun.com/2019/09/21/game-networking-5.html.

[10] Vokal.io, **Usuglašavanje klijenta i poslužitelja**, URL: https://bytes.vokal.io/20160822-multiplayer-client-server-reconciliation/.

[11] Gabriel Gambetta, **Kompenzacija kašnjenja**, URL: https://gabrielgambetta.com/lag-compensation.html.

[12] Valve, **Mrežna komunikacija u igrama u okruženju više igara**, URL: https://developer.valvesoftware.com/wiki/Source%5C_Multiplayer%5C_Networking.

[13] Yan Chernikov, **Sustav događaja**, URL: https://www.youtube.com/watch?v=yuhNj8yGDJQ&list=PLlrATfBNZ98dC-V-N3m0Go4deliWHPFwT&index=19.

[14] Marko Čupić i Željka Mihajlović, **Interaktivna računalna grafika kroz primjere u OpenGL-u**, 2018.

[15] **ImGui**, URL: https://blog.conan.io/2019/06/26/An-introduction-to-the-Dear-ImGui-library.html.

[16] Boost, **Boost.Asio**, URL: https://www.boost.org/doc/libs/1_76_0/doc/html/boost_asio/overview/rationale.html.

Dodatak A
Dodatni tekstovi programa

```
void Game::move_player(glm::vec2 delta, Player& player) noexcept
{
   auto [min_x, max_x, min_y, max_y] = state.get_playable_range();
   auto new_pos = player.center + delta;
   player.center = {
      clamp_to_range(
         new_pos.x, min_x + player.radius, max_x - player.radius),
      clamp_to_range(
         new_pos.y, min_y + player.radius, max_y - player.radius)};
}
```

Isječak A.1: Timer

```
template<class Archive>
void serialize(Archive& ar, Player& p, const unsigned int /*version*/)
{
   ar & p.name;
   ar & p.center;
   ar & p.radius;
   ar & p.color;
   ar & p.score;
   ar & p.speed;
}
```

Isječak A.2: Timer

```
void Server::accept(IOContext& io_context, std::size_t n_players)
{
   Acceptor acceptor(io_context.get_executor(),
                Endpoint(boost::asio::ip::tcp::v4(), port));
   for (std::size_t i = 0; i < n_players; ++i) {
      auto connection =
         std::make_shared<Connection>(io_context.get_executor());
      acceptor.accept(connection->get_socket());

      auto name = connection->read(); // Read player name.
      auto [it, is_inserted] =
         connections.insert({std::move(name), connection});
```

```
        if (!is_inserted) {
            throw std::runtime_error(fmt::format(
                "Player with name {} already connected.", it->first));
        }
    }
}
```

Isječak A.3: Glavna petlja poslužitelja.

```
void Server::update_player_positions()
{
    for (auto& [name, movement_command_cl] : movement_commands) {
        std::size_t i = 0;
        while (i < movement_command_cl.size()) {
            game->update_player_position(
                name,
                Timestep(std::chrono::milliseconds(1000 / tickrate)),
                movement_command_cl[i++]);
        }
        movement_command.clear();
    }
}
```

Isječak A.4: Ažuriranje pozicija klijenata.