

# ČAS ZA OBRAČUN

Strokovno poročilo za 4. predmet poklicne mature

Mentorica: Nataša Makarovič, prof.

Avtor: Klemen Skok, R 4. A

Moravče, april 2025

## **Zahvala**

Zahvaljujem se mentorici Nataši Makarovič, prof. za podporo in nasvete pri izdelavi projekta. Zahvala gre tudi sošolcem za pomoč in ideje, ki so pripomogle k izboljšanju igre. Posebna hvala pa družini za vso spodbudo in razumevanje skozi celoten proces razvoja.

## **Povzetek**

V strokovnem poročilu je predstavljen razvoj večigralske 2D videoigre, ki deluje po principu odjemalec-strežnik. Najprej so predstavljena orodja in tehnologije, ki so bile uporabljene pri razvoju. Sledi opis načrtovanja projekta, nato pa še podrobnejši opis načina komunikacije med napravami, skupnih funkcionalnosti, opis strani odjemalca in strani strežnika. Sama igra je napisana v programskem jeziku C++ z uporabo knjižnice SDL2 za prikaz igre in omrežno komunikacijo ter drugih knjižnic. Komunikacija je izvedena s pomočjo protokola UDP.

**Ključne besede:** C++, SDL2, večigralska videoigra, odjemalec, strežnik, UDP

## **Abstract**

This technical report represents the development of a multiplayer 2D video game that operates on a client-server model. The tools and technologies used in the development are introduced first, followed by a description of the project planning. Then, a more detailed explanation of the communication between devices, shared functionalities, and descriptions of both the client-side and server-side are provided. The game itself is written in C++ using the SDL2 library for rendering and network communication, along with other libraries. Communication is implemented using the UDP protocol.

**Keywords:** C++, SDL2, multiplayer videogame, client, server, UDP

## Kazalo vsebine

1. Uvod .....	6
2. Orodja in knjižnice .....	6
2.1 Programski jezik C++ .....	6
2.2 XML.....	6
2.3 SDL2 .....	7
2.4 Spdlog .....	7
2.5 TinyXML-2 .....	7
2.6 Program Tiled.....	8
2.7 CMake.....	8
3. Načrtovanje sistema.....	8
3.1 Cilji projekta .....	8
3.2 Potek razvoja.....	9
4. Struktura in implementacija igre .....	9
4.1 Osnovni koncepti .....	9
4.1.1 Tipi objektov v igri .....	9
4.1.2 Potek igre .....	12
4.1.3 Mrežni sistem.....	12
4.1.4 XML struktura zemljevida .....	13
4.1.5 Komunikacijski protokol .....	15
4.1.6 Zagon programa .....	17
4.2 Implementacija strežnika .....	18
4.2.1 Glavna zanka.....	19
4.2.2 Čakalna vrsta paketov .....	20
4.2.3 Obdelava podatkov .....	21
4.2.4 Vodenje seje .....	22
4.3 Implementacija odjemalca .....	23
4.3.1 Glavna zanka.....	23
4.3.2 Čakalna vrsta paketov .....	24
4.3.3 Prikazovanje stanja .....	25
4.3.4 Uporabniški vmesnik .....	26
5. Zaključek .....	27
6. Viri in gradiva .....	29

## Kazalo slik

Slika 1: Igralca modre in rdeče ekipe (lastni vir, 2025).....	10
Slika 2: Izstrelka modre (zgoraj) in rdeče (spodaj) ekipe (lastni vir, 2025) .....	10
Slika 3: Zastava (lastni vir, 2025) .....	11
Slika 4: Primera ovir - zaboj in kavč (lastni vir, 2025).....	11
Slika 5: Spolzka (levo) in lepljiva past (desno) (lastni vir, 2025).....	12
Slika 6: Zemljevid igre v programu Tiled (lastni vir, 2025) .....	13
Slika 7: Zagon strežnika (lastni vir, 2025).....	17
Slika 8: Zagon odjemalca (lastni vir, 2025).....	17
Slika 9: Zagon odjemalca z argumenti (lastni vir, 2025).....	18
Slika 10: Prizor med igro (lastni vir, 2025) .....	26
Slika 11: Prikaz trenutnega rezultata (lastni vir, 2025).....	26
Slika 12: Vrstica življenjskih točk (lastni vir, 2025).....	27

# 1. UVOD

---

Moj maturitetni izdelek se je razvil iz ideje o igri, ki jo lahko več igralcev igra skupaj v realnem času. Od mnogih idej se je obdržala igra, ki deluje po konceptu “ujemi zastavo” (angl. “Capture the Flag”). Igralci, razdeljeni v ekipe, se gibljejo po igralnem polju in poskušajo pobrati zastavo ter jo prinesiti nazaj na svoje izhodišče. Pri tem lahko z metanjem izstrelkov upočasnijo in onemogočijo nasprotnike. Igra zahteva sodelovanje med igralci v ekipi in strategijo.

Sistem je implementiran po principu odjemalec-strežnik (angl. “client-server”). Strežnik je glavna entiteta v sistemu. Njegova vloga je vodenje igre, procesiranje vnosov igralcev in pošiljanje stanja igre. Odjemalci prebirajo vnose uporabnika in jih pošiljajo strežniku ter prikazujejo novo stanje.

Cilj tega projekta je bil ustvariti preprosto, a funkcionalno omrežno igro. Pozornost je bila posvečena tako igri kot tudi stabilnosti omrežne povezave, hitremu odzivu uporabniškega vmesnika in gladke uporabniške izkušnje.

## 2. ORODJA IN KNJIŽNICE

---

### 2.1 Programski jezik C++

Programski jezik C++ je visokonivojski programski jezik, ki je bil razvit v 80. letih kot razširitev jezika C. Podpira tako objektno kot tudi proceduralno programiranje. Znan je predvsem po hitrosti izvajanja, nadzoru nad programsko opremo in učinkovitosti.

Jezik podpira koncepte, kot so razredi, dedovanje, večpojavnost (angl. “polymorphism”), šablone (ang. “templates”) in delo s pomnilnikom z uporabo kazalcev. Poleg tega ima standardno knjižnico (“STL”), ki ponuja širok nabor podatkovnih struktur in algoritmov. Za vsem tem pa stoji podrobna dokumentacija, ki vsebuje vse potrebne podatke za razvijalce.

### 2.2 XML

XML (“eXtensible Markup Language”) je označevalni jezik in datotečni format za shranjevanje in prenos podatkov v obliki, ki je razumljiva tako ljudem, kot tudi računalnikom.

Podatki so organizirani v drevesni strukturi z elementi (oz. značkami). Značke lahko vsebujejo attribute za dodatne informacije o posamezni znački, znotraj njih pa lahko tudi gnezdimo ostale značke. Imena značk niso vnaprej določena, pač pa jih avtor dokumenta smiselno poimenuje sam.

## 2.3 SDL2

SDL2 je odprtokodna knjižnica, ki je namenjena razvoju iger in odprtokodnih aplikacij v jeziku C oz. C++. Razvijalcem omogoča, da na enostaven način dostopajo do različnih vrst strojne opreme, kot so miška, tipkovnica, grafična kartica in podobno. Omogoča naprednejše funkcionalnosti, kot so rokovanje z dogodki, upravljanje z okni, predvajanje zvoka in uporaba širšega nabora vhodnih naprav (igralni ploščki ipd.).

Polega osnovne knjižnice obstajajo še uradne podknjižnice, ki razširjajo njeno funkcionalnost. Nekatere od teh so `SDL2_image`, ki se uporablja za prikazovanje slik v različnih formatih, `SDL2_net` za omrežno komunikacijo in `SDL2_ttf` za uporabo poljubnih TrueType pisav v oknu.

## 2.4 Spdlog

Spdlog je odprtokodna knjižnica za C++, ki se uporablja za beleženje izvajanja programa (ang. “logging”). Omogoča izpisovanje ličnih sporočil o delovanju programa v konzolo, datoteko ali druge izhode. Knjižnica je zelo zmogljiva in enostavna za uporabo, saj lahko beleženje nastavimo že z nekaj vrsticami. Podpira več nivojev beleženja, na primer *info*, *warn*, *error*, *debug* in *trace*. To razvijalcem olajša razvijanje programa in iskanje napak v izvajanju.

## 2.5 TinyXML-2

TinyXML-2 je preprosta in hitra knjižnica za delo z XML datotekami. Omogoča učinkovito razčlenjevanje XML datotek in manipulacijo podatkov, ki jih vsebujejo.

Knjižnica uporablja objektni model dokumenta (ang. “Document Object Model” – DOM), kar pomeni, da XML datoteko razčleni v C++ objekte, katere lahko razvijalec spreminja, dodaja ali briše, in jih potem po potrebi shrani nazaj v datoteko.

## 2.6 Program Tiled

Tiled je brezplačen in odprtokoden program za oblikovanje nivojev 2D iger, ki temeljijo na zgradbi iz plošč (angl. “tiles”). Omogoča vizualno oblikovanje nivojev z uporabo različnih orodij in razdelitev na plasti. Projekt lahko izvozimo v različnih datotečnih formatih, med drugimi tudi v XML, kar omogoča enostavno prebiranje podatkov o ustvarjenem nivoju. Program je priljubljen med samostojnimi razvijalci videoiger, saj omogoča enostavno integracijo v igralne pogone in ostale oblike projektov.

## 2.7 CMake

CMake je zmogljiv sistem za generiranje gradbenih (angl. “build”) sistemov, namenjenih prevajanju programske opreme, napisane v jezikih C in C++. Omogoča enostavno vključitev zunanjih knjižnic, upravljanje nastavitev gradnje. Podpira kompleksne projekte, hkrati pa je neodvisen od operacijskega sistema in prevajalnika. Namesto neposrednega prevajanja kode CMake ustvari ustrezne datoteke za orodja, kot sta Make ali Ninja, ki nato poskrbita za dejansko prevajanje. Pri ponovnem prevajanju CMake prevede le tiste datoteke, ki so bile spremenjene, kar lahko bistveno skrajša čas prevajanja.

# 3. NAČRTOVANJE SISTEMA

---

## 3.1 Cilji projekta

Glavni cilj tega projekta je bil ustvariti zabavno 2D igro za več igralcev, ki temelji na arhitekturi odjemalec-strežnik (angl. “client-server”). Za komunikacijo s strežnikom poskrbi protokol UDP, ki omogoča hiter, a nezanesljiv prenos podatkov po omrežju.

Sama igra deluje po principu “ujemi zastavo” (angl. “Capture the Flag”). Štirje igralci so postavljeni v areno in razdeljeni v dve ekipi. Cilj igre je, da ena izmed ekip zastavico prinese na svojo stran igralnega polja. Igro popestrijo ovire, ki so lahko slabše opazne. Polega tega pa se lahko igralci med seboj obmetavajo s predmeti in tako preprečijo nasprotni ekipi hiter napredek. Igra se odvija hitro, potrebno pa je veliko ekipnega dela in strategije.



## 3.2 Potek razvoja

Igre sem se najprej lotil brez računalnika. Kot prve so nastale skice strežniške arhitekture, poteka povezovanja na strežnik in komunikacijskega protokola. Sledilo je testiranje orodij ter knjižnic, priprava razvojnega okolja, konfiguracije CMake in ostalih orodij, ki sem jih uporabil za razhroščevanje in prevajanje programa. Program odjemalca je bil napisan na operacijskem sistemu Windows 11, program strežnika na operacijskem sistemu Linux.

Pisanje kode sem začel na strani strežnika. Definiral sem glavno zanko in implementiral komunikacijo prek omrežja. Povezani igralci so bili razvrščeni v seje.

Ko je bila narejena podlaga na strežniku, sem lahko začel z razvijanjem aplikacije za odjemalce. Pri zgradbi programa sem se zgledoval po shemi strežnika. Ko so se igralci lahko povezali na strežnik, se premikali ter lahko videli ostale igralce, sem lahko na tej podlagi zgradil celotno igro.

## 4. STRUKTURA IN IMPLEMENTACIJA IGRE

---

### 4.1 Osnovni koncepti

V tem podpoglavju so predstavljeni koncepti, ki se jih držijo tako odjemalci kot strežnik, in so ključni za delovanje celotnega sistema.

#### 4.1.1 Tipi objektov v igri

Igra vsebuje več različnih objektov, od katerih vsak po svoje vpliva na igralno izkušnjo. Razdeljeni so v dve skupini, in sicer objekte, ki se lahko premikajo in objekte, ki so zgolj del mape in ves čas stojijo na mestu.

#### Premični objekti

**Igralec** je najpomembnejši premični objekt. Je osnovna enota v igri, ki jo upravlja uporabnik. Ima naslednje lastnosti:

- Položaj (x, y) in smer pogleda.
- Hitrost premikanja, ki se lahko spreminja glede na površino
- Možnost pobiranja zastave. Nošenje zastave ga upočasni in tako naredi bolj ranljivega.
- Življenjske točke oziroma “stopnja telesne drže” (angl. “posture”). Zmanjša se, če je igralec zadet z izstrelkom. Ko pade na 0, se igralec težje premika odkler se vrednost čez nekaj sekund ne poveča. Če nosi zastavo, jo izpusti.

Vse igralce strežnik označi z identifikacijsko (ID) številko, številko seje in ekipo. Igralec lahko pripada rdeči ali modri ekipi, kar je tudi vidno v igri.



Slika 1: Igralca modre in rdeče ekipe (lastni vir, 2025)

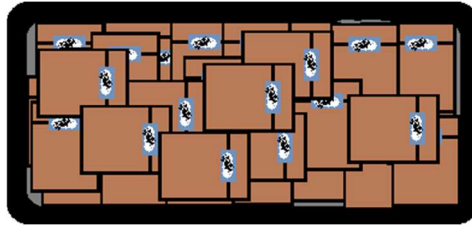
**Izstrelki** so objekti, ki jih igralci lahko mečejo v igralce nasprotne ekipe z namenom, da jih upočasni oziroma jim onemogočijo nošenje zastave. Izstrelki imajo naslednje lastnosti:

- Začetni položaj in smer gibanja
- Hitrost in največjo razdaljo, ki jo lahko prepotujejo. Ko je ta presežena, izstrellek izgine.
- Učinek – če zadanejo igralce nasprotne ekipe, mu odvzamejo življenjske točke.
- Ob trku z oviro se uničijo



Slika 2: Izstrelka modre (zgoraj) in rdeče (spodaj) ekipe (lastni vir, 2025)

**Zastava** se na začetku igre nahaja na sredini igralnega polja, med obema ekipama. Igralci jo lahko poberejo ali izpustijo. Sama po sebi ne vsebuje nobene logike, pač pa njen položaj sledi položaju igralca, ki jo nosi.



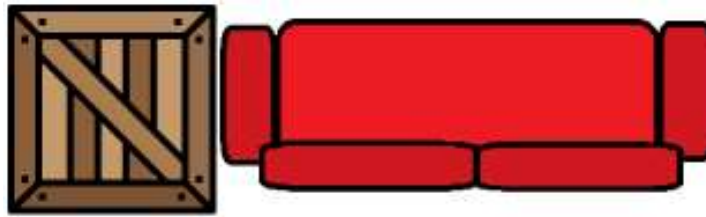
Slika 3: Zastava (lastni vir, 2025)

## Statični objekti

**Ovire** (angl. “barriers”) so objekti, ki služijo kot fizične prepreke za igralce in izstrelke.

Uporabljene so v več namenov:

- Kot stene, ki določajo igralno površino,
- samostojni objekti, ki oblikujejo notranjost igralne površine,
- ali pa kot dekoracija.

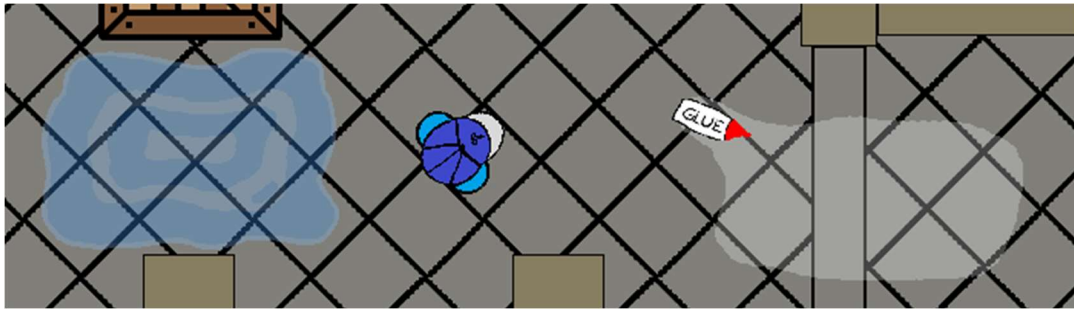


Slika 4: Primera ovir - zaboj in kavč (lastni vir, 2025)

**Pasti** so posebna vrsta površin, ki igralcu ne preprečujejo premikanja, pač pa nanj vplivajo na različne načine:

- *Lepljive pasti* igralca upočasnijo.
- *Spolzke pasti* igralcu odvzamejo nadzor nad premikanjem tako, da zmanjšajo trenje in tako povzročijo učinek drsenja.

Igralcu so manj vidne, zato lahko delujejo kot element presenečenja.



Slika 5: Spolzka (levo) in lepljiva past (desno) (lastni vir, 2025)

### 4.1.2 Potek igre

Ko je v sejo dodan prvi igralec, se začne faza čakanja na igralce. V tej fazi se igralci lahko prosto premikajo in raziskujejo igralno površino. Ko se poveže četrti igralec, se čakanje zaključi in začne se štirisekundno odštevanje do začetka runde. Igralci se teleportirajo na začetne položaje. Premikanje je onemogočeno. Ko se runda začne, se igralci lahko premikajo, pobirajo razstavo in streljajo izstrelke. Runda se konča, ko je zastava s svojo celotno površino na eni izmed baz. Rundo dobi ekipa, ki je zastav prinesla do svoje baze. Sledi trisekundni odmor, med katerim se na zaslonu izpiše zmagovalec runde. Zatem se začne ponovno odštevanje do naslednje runde.

Zmaga ekipa, ki v petih rundah doseže največ zmag.

### 4.1.3 Mrežni sistem

Namesto, da bi bili deli mape shranjeni shranjeni zgolj v navadnem seznamu, sem se odločil za pristop z mrežnim sistemom. Gre za mrežo plošč, ki si jo lahko predstavljamo kot poenostavljen koordinatni sistem, v katerem vsaka plošča zavzema eno enoto. Vsaka izmed plošč vsebuje seznam objektov, ki se je dotikajo.

Pri dodajanju objekta v mrežo je kazalec nanj shranjen v vseh ploščah, ki se jih objekt dotika oziroma si z njim deli koordinate. Večji objekti se lahko razpostirajo čez več plošč.

Metoda, ki oviro doda v mrežni sistem:

```
void MapData::AddBarrier(Barrier& b) {
    auto pos = b.getPosition();

    int start_x = getGridKey(pos.x);
```

```

int start_y = getGridKey(pos.y);
int end_x = getGridKey(pos.x + b.getWidth());
int end_y = getGridKey(pos.y + b.getHeight());

if(int(pos.x + b.getWidth()) % GRID_CELL_SIZE == 0) end_x++;
if(int(pos.y + b.getHeight()) % GRID_CELL_SIZE == 0) end_y++;

b.setTexture(AssetManager::GetTexture(b.getTextureId()));

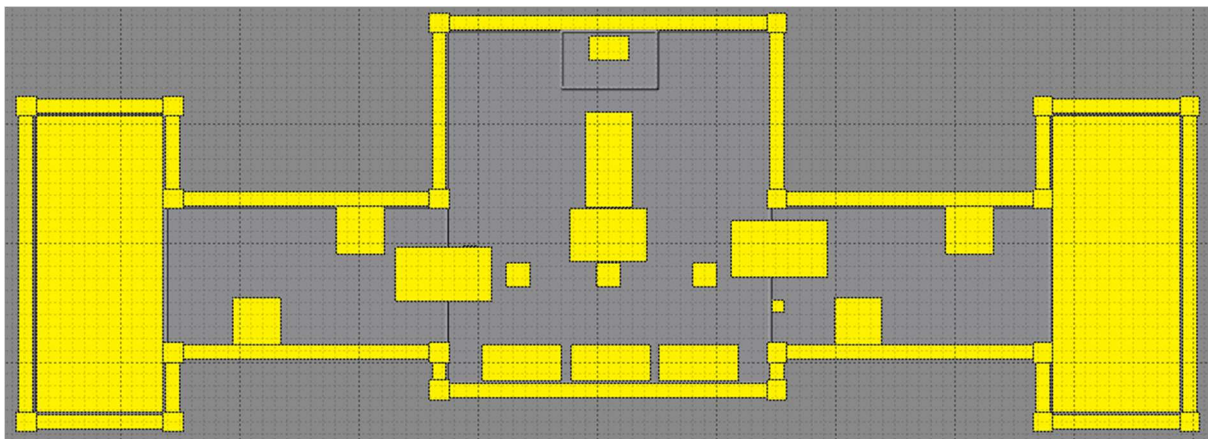
for(int x = start_x; x <= end_x; x++) {
    for(int y = start_y; y <= end_y; y++) {
        grid[x][y].push_back(std::make_shared<Barrier>(b));
    }
}
}

```

Ta pristop sem izbral predvsem zaradi potrebe po pogostem preverjanju trkov med premikajočimi se objekti in deli mape (stene, ovire ipd.). Namesto, da bi za vsak objekt preverjal trke z vsemi elementi na mapi, program izračuna približno lokacijo objekta mrežnem sistemu in preveri le trke s tistimi elementi, s ki se nahajajo v istih ploščah. Ta način lahko znatno zmanjša število nepotrebnih preverjanj in prispeva k optimizaciji programa. Slednja je posebej pomembna na strežniku, saj ta upravlja več iger hkrati. Za omrežni sistem skrbi razred *MapData*.

#### 4.1.4 XML struktura zemljevida

Zemljevid igralne površine sem izdelal v programu Tiled. Najprej sem določil obliko tlorisa s stenami, potem pa sem prazen prostor popestril z raznimi ovirami, kot so stebri, škatle in kavči. Kasneje sem dodal še pasti.



Slika 6: Zemljevid igre v programu Tiled (lastni vir, 2025)

Do podatkov o položajih objektov sem lahko enostavno dostopal prek projektne datoteke programa Tiled. Podatke sem nato pretvoril v svoj XML format in jih uporabil v programu.

Primer zapisa ovire v XML formatu:

```
<barrier>
  <position x="408" y="515" />
  <size w="50" h="50" />
  <texture id="02" />
</barrier>
```

Podatki o celotni mapi so shranjeni v XML datoteki *map\_data.xml*. Podatki so razdeljeni v več skupin glede na funkcijo objekta:

- *<barriers>*: podatki o vseh ovirah. Vsaka ovira vsebuje podatke o položaju in dimenzijah. Na strani odjemalca je dodana še številka strukture, ki določa, kako je dana ovira prikazana na zaslonu.
- *<floor>*: program na strani odjemalca mora igralnemu polju narisati tudi tla. Ta so prikazana s pomočjo ene majhne slike, ki se ponavlja, dokler ni zapolnjena celotna površina tal. Delo sem si olajšal tako, da sem celotno igralno površino razdelil na nekaj večjih prostorov, ki jih potem program zapolni z vzorcem. Tukaj so shranjeni podatki o teh prostorih.
- *<sites>*: podatki o bazah obeh skupin. Vsaka baza vsebuje položaj, dimenzije in številko ekipe (1 ali 2)
- *<traps>*: podatki o pasteh. Vsaka past vsebuje položaj, dimenzije, številko teksture (samo za odjemalce) in vrsto pasti.

Prednost shranjevanja strukture zemljevida v zunanji datoteki je, da ob spremembah ni potrebno ponovno prevesti programa. To omogoča hitre popravke in dodajanje novih elementov zemljevida po potrebi.

Okvirna struktura datoteke *map\_data.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<map>
  <barriers>
    <!-- ... -->
  </barriers>

  <floor>
    <!-- ... -->
  </floor>

  <sites>
    <!-- ... -->
```

```

        </sites>

        <traps>
            <!-- ... -->
        </traps>
    </map>

```

Ob zagonu program prebere to datoteko in s pomočjo knjižnice TinyXML-2 iz nje pridobi vse potrebne podatke. Najprej so prebrane vse ovire, ki jih funkcija *parseBarrierNode* napolni s podatki. Nato so dodane v mrežni sistem. Na enak način so dodane tudi pasti. Sledita bazi obeh ekip, ki sta zaradi hitrega dostopa shranjeni posebej. Na koncu odjemalci pridobijo še podatke, potrebne za risanje tal, ki so prav tako shranjeni ločeno od mrežnega sistema.

Del funkcije, ki iz datoteke prebere ovire:

```

XMLNode* barriers =
rootHandle.FirstChildElement("barriers").FirstChildElement("barrier").ToNode();
if(barriers == nullptr) {
    throw std::runtime_error("Failed to find barriers in the map data.");
}
else {
    // extract map barriers
    for(XMLNode* node = barriers; node != nullptr; node = node->NextSiblingElement("barrier")) {
        Barrier b;
        if(parseBarrierNode(node, b) == 0) {
            AddBarrier(b);
        }
    }
}

```

### 4.1.5 Komunikacijski protokol

Za komunikacijo med odjemalci in strežnikom uporabljen omrežni protokol UDP, ker omogoča hiter prenos podatkov, kar je pomembno pri akcijskih igrah, kot je ta. S tem prinaša tudi slabosti, saj paketi lahko pridejo podvojeni, v napačnem vrstnem redu, ali pa sploh ne pridejo. Med načrtovanjem projekta je bilo potrebno paziti prav na to.

### Zgradba omrežnega paketa

Vsak paket, ki je poslan v omrežje, ima natanko določeno zgradbo. Pri načrtovanju sem se zgledoval po delovanju zanesljivejšega protokola TCP, ki pakete označuje z zastavicami (angl. "flags").

Pri komunikaciji so uporabljene sledeče zastavice:

- ACK: potrditev nečesa. V nekaterih primerih odsotnost te zastavice pomeni zavrnitev.
- FIN: zahteva za prekinitev povezave. V kombinaciji z ACK pomeni odobritev te zahteve.
- SYN: zahteva za vzpostavitev povezave.
- KEEPALIVE: paket služi za vzdrževanje povezave.
- DATA: paket s to zastavico vsebuje podatke o igri.

Vsak paket je sestavljen iz zastavic. Sledi številka seje in številka igralca, na katerega se nanaša paket. Tu je še zaporedna številka paketa, tip paketa in podatkovni del.

## Vrste paketov

Vrsta paketa prejemniku paketa pove, kako mora prebrati paket, da lahko iz njega pravilno izloči podatke. Vrste paketov so definirane s pomočjo oštevilčenja (angl. “enumeration”).

*enum PacketType:*

```
enum class PacketType: uint8_t {  
    // odjemalec -> strežnik  
    PLAYER_UPDATES,  
  
    // strežnik -> odjemalec  
    PLAYERS_IN_RANGE,  
    PROJECTILES_IN_RANGE,  
    GAME_STATE,  
    FLAG_STATE  
};
```

## Povezovanje in seja

Povezavo med odjemalcem in strežnikom mora vedno začeti odjemalec. To stori tako, da strežniku pošlje zahtevo za vzpostavitev povezave. Strežnik zahtevo predela in poskusi dodati novega igralca. Če je to mogoče, odjemalcu pošlje potrditev. V nasprotnem primeru povezavo zavrne.

Ko želi odjemalec prekiniti povezavo, strežniku pošlje zahtevo za prekinitev povezave, skupaj s svojo ID številko in števiko seje. Strežnik odgovori s potrditvijo ter igralca odstrani iz igre.



## Izmenjava podatkov

Strežnik nadzoruje potek igre in določa veljavne položaje vseh objektov in igralcev na polju. Odjemalci strežniku pošiljajo svoja dejanja (stanja tipk, usmerjenost pogleda), strežnik pa te podatke preveri, izvede logiko igre (premikanje objektov, trke, interakcije z zastavo ipd.) in nato vsem igralcem pošlje posodobljeno stanje.

Nezanesljivost protokola UDP je med izvajanjem seje rešena tako, da strežnik podatke o stanju igre pošilja periodično, in sicer približno vsakih 32 milisekund oziroma s frekvenco 30 Hz. To je dovolj za gladko igralsko izkušnjo, hkrati pa z relativno nizko frekvenco pošiljanja razbremeni strežnik.

### 4.1.6 Zagon programa

Program je torej sestavljen iz dveh delov: odjemalca in strežnika. Oba se zaganjata prek ukazne vrstice. Najprej je potrebno zagnati strežnik, nato pa se lahko nanj povezujejo odjemalci.

#### Zagon strežnika

Za zagon strežnika se moramo premakniti v mapo z izvršljivo datoteko in uporabiti naslednji ukaz:



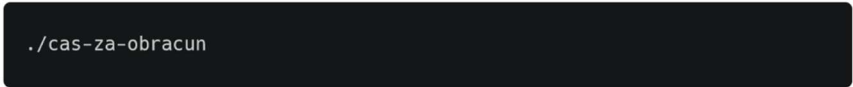
```
./game_server
```

Slika 7: Zagon strežnika (lastni vir, 2025)

Strežnik za sprejemanje paketov uporablja UDP omrežna vrata 55000, za pošiljanje pa vrata 55001. Promet skozi ta vrata moramo omogočiti tudi v nastavitvah požarnega zidu.

#### Zagon odjemalca

Za zagon odjemalca se moramo prav tako premakniti v mapo z izvršljivo datoteko in ga zagnati:



```
./cas-za-obracun
```

Slika 8: Zagon odjemalca (lastni vir, 2025)

Mogoče je tudi ročno nastaviti naslov strežnika in vrata, na katerih sprejema omrežni promet. Podatka navedemo kot argumenta programa:

```
./cas-za-obracun <naslov strežnika> <vrata strežnika>
```

Slika 9: Zagon odjemalca z argumenti (lastni vir, 2025)

Pri odjemalcu lahko požarni zid blokira UDP promet, ki prihaja s strežnika, zato je včasih potrebno programu dodeliti ustrezne pravice v nastavitvah požarnega zidu.

## 4.2 Implementacija strežnika

Strežniška aplikacija deluje na treh nitih (angl. “threads”). Na glavni niti se izvaja osrednja funkcionalnost strežnika, na stranskih nitih pa poteka komunikacija – pošiljanje paketov na eni niti in sprejemanje na drugi niti.

Osrednji del je sestavljen iz več razredov s statičnimi metodami in lastnostmi, kar omogoča logično razdelitev funkcionalnosti programa. Glavni razred je razred *Server*, ki med drugimi vsebuje tri glavne metode za zagon strežnika. Te metode so *Server::Setup*, ki poskrbi za zagon komunikacijskih nitih ter nalaganje zemljevida iz XML datoteke, *Server::Run*, kjer se izvaja glavna zanka programa in *Server::Cleanup*, ki se izvede ob zaključku programa in poskrbi za pravilno zaustavitev. Razred vsebuje še metode, ki se izvajajo v glavni zanki in metode, ki so namenjene upravljanju z odjemalci in seji.

Lastnosti, ki jih vsebuje *Server*, so seznam aktivnih sej ter seznama prostih ID številke sej in odjemalcev, ki niso več v uporabi in bodo imeli prioriteto, ko bo potrebno dodeliti ID številko novemu odjemalcu ali seji. Razlog za to je omejenost s prostorom v omrežnem paketu. Številka seje zavzema 1 bajt (vrednosti od 0 do 255), številka igralca pa 2 bajta (vrednosti od 0 do 65535), kar pomeni, da bi številke hitro zmanjkale in prišlo bi do napake. Zato se številke sej in igralcev ponovno uporabljajo.

Definicija razreda *Server*:

```
class Server {
    static std::unordered_map<uint8_t, std::unique_ptr<GameSession>>
    _sessions;
    static std::set<uint16_t> _free_client_ids;
    static std::set<uint8_t> _free_session_ids;

public:
    // glavne funkcije
```

```

static void Setup(uint16_t i, uint16_t o);
static void Run();
static void Cleanup();

// funkcije v glavni zanki
static void processNewPackets();
static void manageGameSessions();
static void sendPendingPackets();
static void checkClientInactivity();

// delo z odjemalci
static int addClient(IPAddress ip);
static void removeClient(uint16_t c_id);
static void removeClient(uint16_t c_id, uint8_t s_id);
static int queryClient(uint16_t c_id);
static int queryAddress(IPAddress ip);
static IPAddress getClientAddr(uint16_t c_id);

// delo s sejami
static int addSession();
static void removeSession(uint8_t id, UDPsocket socket);
};

```

### 4.2.1 Glavna zanka

Glavna zanka strežika izgleda dokaj preprosto. Najprej funkcija *Server::processNewPackets* poskrbi za procesiranje novih paketov, ki so bili prejeti od zadnje iteracije. Nato se kliče metoda *Server::manageGameSessions*. Ta metoda posodobi stanja vseh sej, ki so trenutno aktivne. Hkrati tudi zaključi prazne in končane seje. Metoda *Server::sendPendingPackets* pridobi vse pakete, ki so jih v prejšnjem koraku ustvarile seje in jih doda v kolono za pošiljanje. Metoda *Server::checkClientInactivity* odstrani tiste odjemalce, ki že dolgo niso poslale nobenega paketa (zaradi izgube povezave ali drugih dejavnikov). Tako se sprosti prostor za nove igralce.

Na koncu še preverimo, ali je uporabnik zahteval zaustavitev strežnika. Sledi še relativno kratek zamik, s katerim omejimo hitrost delovanja strežnika, hkrati pa poskrbimo, da vseeno ostane dovolj odziven.

Metoda *Server::Run*:

```

void Server::Run() {
    bool quit = false;

    while(!quit) {

        Server::processNewPackets();

        Server::manageGameSessions();

        Server::sendPendingPackets();

        Server::checkClientInactivity();
    }
}

```

```

    SDL_Event e;
    while(SDL_PollEvent(&e)) {
        if(e.type == SDL_QUIT) {
            quit = true;
        }
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(1));
}
}

```

### 4.2.2 Čakalna vrsta paketov

Vse tri niti med seboj povezuje dva vsebnika (angl. “containers”) iz C++ standardne knjižnice *std::queue*, ki vsebujeta pakete, ki morajo biti poslani oziroma predelani. To so objekti razreda *UDPmessage*, ki vsebujejo vse potrebne podatke (kanal, IP naslov, dolžino in vsebino paketa). Pri uporabi teh vsebnikov je potrebno paziti na pravilno dostopanje do njih. Lahko se namreč zgodi, da do istega vsebnika hkrati dostopata dve niti, kar lahko privede do nedefiniranega stanja in napak. V ta namen se uporablja zaklepanje z uporabo objekta *std::mutex* – “ključavnica”. Ko poskušamo zakleniti želeni vsebnik, bo *std::mutex* čakal, dokler ne prost za uporabo. Nato lahko dostopamo do vsebnika.

Primer dodajanja paketov v čakalno vrsto in zaklepanja z *std::mutex*:

```

void addMessagesToQueue(std::vector<std::unique_ptr<UDPmessage>>& data) {
    {
        std::lock_guard<std::mutex> lock(sendq_mutex);
        for(auto& msg : data) {
            sendQueue.push(std::move(msg));
        }
    }
    data.resize(0);
    data.shrink_to_fit();
}

```

### Pošiljanje paketov

Pakete, ki morajo biti poslani odjemalcem, glavna nit doda v čakalno vrsto – vsebnik *sendQueue*. Za pošiljanje teh paketov skrbi razred *SocketSpeaker*. Ob zagonu se ustvari nova nit, v kateri se izvaja metoda *SocketSpeaker::Speak*.

Razred *SocketSpeaker*:

```

class SocketSpeaker {

```

```

static std::unique_ptr<std::thread> worker;
static UDPsocket socket;

public:
    static std::atomic<bool> _shutdown;
    static std::atomic<bool> _running;

    static void Start(uint16_t port);
    static void Speak(UDPsocket) noexcept;
    static void Stop() noexcept;
    static UDPsocket getSocket() noexcept;
};

```

## Sprejemanje paketov

Paketi, ki so bili prejeti, so shranjeni v vsebniku *RecievedQueue* in čakajo na procesiranje. Za sprejemanje paketov skrbi razred *SocketListener*. Deluje na podoben princip kot razred za pošiljanje – ob zagonu se ustvari nova nit, v kateri se izvaja metoda *SocketListener::Listen*. Metoda *Listen* vse prejete pakete doda v čakalno vrsto, kjer čakajo na obdelavo v glavni niti.

Razred *SocketListener*:

```

class SocketListener {

    static std::unique_ptr<std::thread> worker;
    static UDPsocket socket;

public:
    static std::atomic<bool> _shutdown;
    static std::atomic<bool> _running;

    static void Start(uint16_t port);
    static void Listen(UDPsocket) noexcept;
    static void Stop() noexcept;
    static UDPsocket getSocket() noexcept;
};

```

### 4.2.3 Obdelava podatkov

Obdelava podatkov poteka v metodi *Server::processNewPackets*, ki se kliče v glavni zanki strežnika. Metoda preveri, ali je bil v čakalno vrsto *recievedQueue* dodan nov paket in ga vzame iz vrste.

Najprej ugotovi, ali je pošiljatelj paketa že poznan. Pri tem strežnik izkorišča funkcionalnost knjižnice *SDL2\_net*, ki omogoča shranjevanje IP naslova kot številko t.i. “kanala” (angl. “channel”). Ta številka je pri povezavi določena glede na ID številko igralca.

Če je torej strežnik ugotovi, da pošiljatelj podatkov še niti poznan in če vsebina paketa ustreza zahtevani obliki za vzpostavitev povezave, strežnik poskusi ustvariti novega igralca. Če vsebina paketa ni veljavna, se paket ignorira.

V primeru, da je pošiljatelj že znan, program s pomočjo zanke preveri stanje zastavic in ustrezno obdela paket:

- FIN – z odjemalcem prekine povezavo.
- KEEPALIVE – posodobi igralčev časovnik na trenutni čas, kar strežniku pove, da je povezava še vedno aktivna, tudi če se igralec ne premika.
- DATA – podatke posreduje ustrezni seji (angl. “session”) za nadaljnjo obdelavo.

#### 4.2.4 Vodenje seje

Vse seje so objekti razreda *GameSession*, ki vsebuje vse potrebne lastnosti in metode za vodenje seje. To vključuje vsebnike za vse premične objekte na igralnem polju, ki so odvisni od seje – igralce (objekti razreda *Player*), izstrelke (objekti razreda *Projectile*) in zastavo (ne vključuje pa elementov zemljevida – ti so skupni vsem sejam). Ločeno so shranjeni še podatki o odjemalcih – številka in čas zadnjega prejetega paketa, številka zadnjega poslanega paketa ter IP naslov. Prisotne so še vrednosti za vodenje stanja igre – trenutna faza in števeci trajanja. Seje omogočajo še dodajanje in odstranjevanje igralcev.

Glavna metoda za vodenje seje je *manageSession*, ki predstavlja glavno zanko seje. Metoda preveri, koliko časa je minilo od zadnje posodobitve. Ko je čas za posodobitev, se kličejo naslednje metode:

1. *updateEverything*: metoda posodobi vse premične objekte. Kot argument sprejme čas, ki je minil od zadnje posodobitve (angl. “delta time”). Položaji so posodobljeni s formulo:  $noviPoložaj = stariPoložaj + hitrost * deltaTime$ . Posodobljen je tudi položaj zastavice, če jo kdo nosi. Za razliko od ostalih objektov so trki igralcev z elementi zemljevida preverjeni že pri posodobitvi igralca.
2. *checkCollisions*: metoda preveri še vse ostale trke. Najprej so preverjeni trki izstrelkov z igralci, sledijo še trki izstrelkov s stenami. Na koncu se preveri še, če se zastava nahaja na kateri izmed baz, saj bi to pomenilo zaključek runde.
3. *checkGameState*: po potrebi spremeni fazo igre.
4. *broadcastUpdates*: odjemalcem pošlje nove položaje igre in ostale posodobitve.

## 4.3 Implementacija odjemalca

Aplikacija odjemalca se zgleduje po strežniški arhitekturi. Na glavni niti poteka osrednji del programa, za razliko od strežnika pa pri odjemalcu komunikacija poteka le na eni stranski niti. To poenostavi komunikacijo s strežnikom, saj za vso komunikacijo odjemalec uporablja ista UDP vrata (angl. “port”).

Program je logično razdeljen na več razredov s statičnimi lastnostmi in metodami, od katerih ima vsak svojo vlogo. Osrednji razred je razred *Game*, ki med drugimi vsebuje glavno zanko in metode za zagon in zaustavitev programa.

- *Setup*: metoda je klicana ob zagonu programa. Zažene komunikacijsko nit, odpre okno, naloži texture za uporabniški vmesnik ter pripravi zemljevid igre.
- *setServerIP*: nastavi naslov strežnika. Če je uporabnik navedel svoj naslov strežnika, je nastavljen ta. Sicer je uporabljen privzeti naslov (*skok.cc*).
- *Run*: v tej metodi teče glavna zanka programa.
- *Cleanup*: ta metoda je klicana ob izteku programa in poskrbi za pravilno zaustavitev.

Razred vsebuje še druge lastnosti in metode, ki služijo delu s paketi ter upravljanju s povezavo in podatki o igri.

Zaradi preglednosti obstajajo še drugi razredi s statično vsebino, ki skrbijo za različna opravila:

- *SocketHandler*: skrbi za pošiljanje in sprejemanje paketov
- *PacketHandler*: obdela prejete pakete in sestavlja pakete, ki bodo poslani
- *EventHandler*: skrbi za zaznavanje uporabnikovih vnosov
- *RenderWindow*: skrbi za prikaz igre in uporabniškega vmesnika na zaslon

### 4.3.1 Glavna zanka

Glavna zanka odjemalca najprej pripravi vse potrebne spremenljivke (klic metode *Initialize*). Med izvajanjem glavne zanke si sledi več korakov.

Najprej program s klicem metode *processNewPackets* pridobi in predela prejete pakete. Nato preveri uporabnikove vnose in ob morebitnih spremembah ustvari paket posodobitev za strežnik in ga doda v čakalno vrsto. Zatem klic metode *manageConnection* poskrbi za redno

pošiljanje ustreznih paketov (zahteve za povezavo, pakete za ohranitev povezave ali zahteve za prekinitev povezave. To je odvisno od stanja povezave). Iteracija se zaključi z lokalnim posodabljanjem igre (*Update*) in prikazom stanja (*Render*).

Glavna zanka se izvaja s frekvenco 60 hercov (med dvema iteracijama je približno 16,6 milisekund premora). To je dovolj za gladko prikazovanje stanja.

Metoda *Game::Run*:

```
void Game::Run() {  
  
    Game::Initialize();  
    Uint32 lastUpdate = SDL_GetTicks();  
  
    while(Game::_running) {  
        Uint32 now = SDL_GetTicks();  
        if(now - lastUpdate < GAME_LOOP_DELAY) {  
            continue;  
        }  
        int deltaTime = now - lastUpdate;  
        lastUpdate = now;  
  
        Game::processNewPackets();  
        EventHandler::HandleEvents();  
        Game::manageConnection();  
        PacketHandler::sendPendingPackets();  
  
        if(Game::current_state != GameState::NONE) {  
            Game::Update(deltaTime);  
        }  
  
        Game::Render();  
    }  
    Window::Close();  
  
    std::this_thread::sleep_for(std::chrono::milliseconds(100));  
}
```

### 4.3.2 Čakalna vrsta paketov

Glavno in komunikacijsko nit povezujeta dva vsebnika *std::queue* – *recievedQueue* za sprejete pakete, ki morajo biti predelani in *sendQueue* za pakete, ki morajo biti poslani strežniku. Tudi tukaj je potrebno paziti na dostopanje do vsebnikov, zato ima vsak od njiju svojo ključavnico – *std::mutex*, ki poskrbi za posamično uporabo vsebnikov.

Za komunikacijo na stranski niti skrbi razred *SocketHandler*. Na niti se ob zagonu začne izvajati metoda *Work*, ki skrbi tako za pošiljanje kot za sprejemanje paketov.



Razred *SocketHandler*:

```
class SocketHandler {  
  
    static std::unique_ptr<std::thread> worker;  
    static UDPsocket socket;  
  
public:  
    static std::atomic<bool> _shutdown;  
    static std::atomic<bool> _running;  
  
    static void Start();  
    static void Work(UDPsocket) noexcept;  
    static void Stop() noexcept;  
    static UDPsocket getSocket() noexcept;  
};
```

### 4.3.3 Prikazovanje stanja

Paketi, ki jih prejme odjemalec, so takoj sprocesirani, vendar nova stanja objektov, ki jih vsebujejo, niso nujno takoj prikazana. Razlog za to je, da odjemalska aplikacija igro prikazuje z višjo frekvenco (60 Hz), kot jo strežnik pošilja (30 Hz). Če bi nova stanja prikazali takoj po prejemu, bi bilo premikanje igralcev in drugih objektov videti zelo grobo.

To je rešeno tako, da novo stanja ni prikazano takoj. Namesto tega najprej prikažemo vmesno stanje med prejšnjim in novim stanjem. Vmesno stanje izračunamo s pomočjo *linearne interpolacije* (angl. “linear interpolation”) med znanima stanjema, pri čemer uporabimo določen faktor glajenja (vrednost med 0 in 1). Faktor glajenja določa, kako gladek bo prehod med stanji – prehod bo manj gladek (in hitrejši), ko se bližamo 1. Enačba za linearno interpolacijo, ki jo uporablja program, je sledeča:

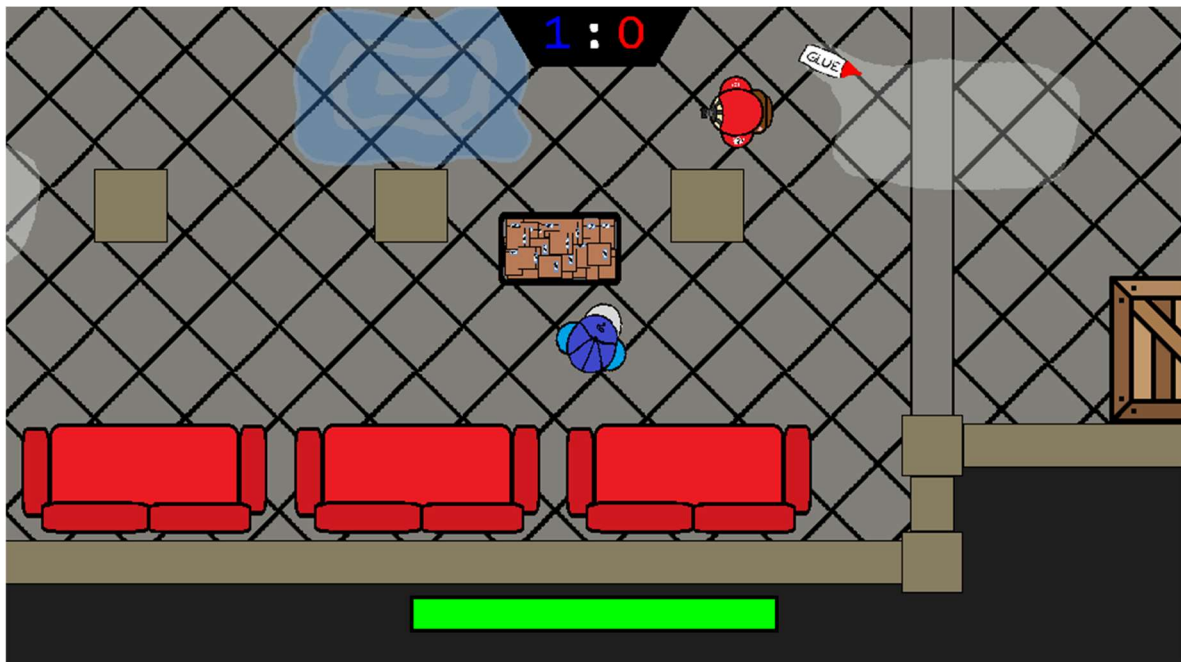
$$vmesniPoložaj = stariPoložaj + faktorGlajenja * (noviPoložaj - stariPoložaj)$$

V kolikor paket z novim položajem še ni prispel, interpolacija s prejšnjim stanjem pa je že zaključena (npr. ob izgubi paketa ali krajši prekinitvi povezave), program poskusi napovedati naslednji položaj na podlagi zadnjega znanega stanja tipk in hitrosti objekta.

Za lokalnega igralca se uporablja drugačen pristop. Če bi namreč za lokalnega igralca prikazovali le stanje, ki ga pošlje strežnik, bi se igra zdela počasna in neodzivna. Zato odjemalec takoj prikaže novi položaj glede na svoje podatke, s podatki s strežnika pa vrednosti le popravi. Za boljšo uporabniško izkušnjo je tudi tu uporabljena interpolacija, vendar se strežniški podatki tu uporabijo takoj, brez zakasnitve.

#### 4.3.4 Uporabniški vmesnik

Uporabniški vmesnik je preprost in intuitiven. Ob zagonu aplikacije se prikaže glavni meni, na katerem je izpisan naziv igre in navodilo za začetek. Ob pritisku na tipko se program poskuša povezati s strežnikom. Po koncu igre se glede na rezultat prikaže zaslon z obvestilom o zmagi ali porazu. S pritiskom na tipko 1 se program zapre.



Slika 10: Prizor med igro (lastni vir, 2025)

Med igranjem so oknu prisotni trije glavni deli uporabniškega vmesnika:

##### Prikaz trenutnega rezultata

Na sredini zgornjega roba okna je prikazan trapez, v katerem je prikazan trenutni rezultat igre. Točke modre ekipe so prikazane v modri barvi, točke rdeče ekipe pa v rdeči.



Slika 11: Prikaz trenutnega rezultata (lastni vir, 2025)

## Statusna vrstica življenjskih točk

Pogosteje znana kot “health bar”. Nad spodnjim robom okna je prikazana vrstica, ki ponazarja stanje igralčeve telesne drže.

Ko je ta vrednost višja od 50%, je vrstica obarvana zeleno, med 25 % in 50 % oranžno, pod 25 % rdeče. Dolžina obarvanega dela je sorazmerna z odstotkom zdravja.



Slika 12: Vrstica življenjskih točk (lastni vir, 2025)

## Smer zastave

Vsakič, ko je zastava predaleč, da bi bila vidna na zaslonu, se prikaže puščica, ki kaže v njeno smer. Puščica se lahko prikaže v treh različnih barvah:

- **Siva**, ko je zastava na tleh,
- **modra**, ko zastavo nosi član modre ekipe,
- **rdeča**, ko zastavo nosi član rdeče skupine.

To igralcem pomaga pri prepoznavanju trenutnega stanja igre – hitro lahko ugotovijo, v katero smer morajo in koga lahko tam pričakujejo.

## 5. ZAKLJUČEK

Igra *Čas za obračun* je bila moj najzahtevnejši projekt doslej. Med razvojem sem se veliko naučil – utrdil sem znanje programskega jezika C++ in se seznanil tudi z naprednejšimi funkcionalnostmi jezika. Čeprav sem knjižnico SDL2 poznal že prej, sem jo tokrat spoznal še bolj poglobljeno. Razvoj projekta, ki uporablja omrežno povezavo za prenos podatkov se je izkazal za dokaj zahtevnega, saj sem moral zasnovati lastni komunikacijski protokol, četudi preprost.

Pogosto sem naletel na težave, kar me je prisililo učinkovitejše iskanje in odpravljanje napak. Nepravilnosti so pogosto izvirale iz pomanjkljivega načrtovanja, kar mi služi kot pomembna

lekcija za prihodnje projekte. Prav zaradi pomanjkanja začetnega načrta so nekatere funkcionalnosti, kot je uporabniški vmesnik, ostale manj dodelane.

Čeprav v igri opažam številne pomanjkljivosti in možnosti za izboljšavo, sem s končnim rezultatom zadovoljen. Dosegel sem svoj glavni cilj – ustvariti zabavno in stabilno igro, ki omogoča sodelovanje med več igralci.

## 6. VIRI IN GRADIVA

---

- C++ reference. Dostopno prek: <https://en.cppreference.com/w/> (16. 4. 2025)
- SDL2 Wiki. 2025. Dostopno prek: <https://wiki.libsdl.org/SDL2/APIByCategory> (16. 4. 2025)
- SDL2 Wiki – SDL2\_Net. Dostopno prek: [https://wiki.libsdl.org/SDL2\\_net/CategoryAPI](https://wiki.libsdl.org/SDL2_net/CategoryAPI) (16. 4. 2025)
- SDL2 Wiki – SDL2\_Image. Dostopno prek: [https://wiki.libsdl.org/SDL2\\_image/CategoryAPI](https://wiki.libsdl.org/SDL2_image/CategoryAPI) (16. 4. 2025)
- SDL2 Wiki – SDL2\_Ttf. Dostopno prek: [https://wiki.libsdl.org/SDL2\\_ttf/CategoryAPI](https://wiki.libsdl.org/SDL2_ttf/CategoryAPI) (16. 4. 2025)
- Spdlog. Dostopno prek: <https://github.com/gabime/spdlog> (16. 4. 2025)
- Cmake Reference Documentation. Dostopno prek: <https://cmake.org/cmake/help/latest/index.html> (16. 4. 2025)

### Prevodi strokovnih besed

- Računalniški slovarček. Dostopno prek: [https://dis-slovarcek.ijs.si/list\\_searched](https://dis-slovarcek.ijs.si/list_searched) (16. 4. 2025)
- PONS prevajalnik. Dostopno prek: <https://sl.pons.com/prevod> (16. 4. 2025)

### Viri slik

- Tekstura zaboja. Dostopno prek: [https://images.tcdn.com.br/img/img\\_prod/343949/categoria\\_img\\_755\\_20200306113809.png](https://images.tcdn.com.br/img/img_prod/343949/categoria_img_755_20200306113809.png) (16. 4. 2025)

## **Izjava o avtorstvu**

Izjavljam, da je strokovno poročilo Čas za obračun v celoti moje avtorsko delo, ki sem ga izdelal samostojno s pomočjo navedene literature in pod vodstvom mentorice.

Moravče, 16. 4. 2025

Klemen Skok

