УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Алекса Вукомановић

RustyArena - мултиплејер игра на програмском језику Rust

ЗАВРШНИ РАД

Основне академске студије

Нови Сад, 2025

	УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА** 21000 НОВИ САД, Трг Доситеја Обрадовића 6	Број:
	ЗАДАТАК ЗА ЗАВРШНИ РАД	Датум:

(Податке уноси предметни наставник - ментор)

Студијски програм:	Софтверско инжењерство и информационе технологије		
Студент:	Алекса Вукомановић	Број индекса:	SV66/2021
Степен и врста студија:	Основне академске студије		
Област:	Електротехничко и рачунарско инжењерство		
Ментор:	Игор Дејановић		

НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ЗАВРШНИ РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:

- проблем – тема рада;
- начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;

НАСЛОВ ЗАВРШНОГ РАДА:

RustyArena - мултиплејер игра на програмском језику Rust

ТЕКСТ ЗАДАТКА:

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

Руководилац студијског програма:	Ментор рада:

Примерак за: □ - Студента; □ - Ментора

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографска документација
Тип записа, **ТЗ**:		Текстуални штампани материјал
Врста рада, **ВР**:		Дипломски - бечелор рад
Аутор, **АУ**:		Алекса Вукомановић
Ментор, **МН**:		Др Игор Дејановић, редовни професор
Наслов рада, **НР**:		RustyArena - мултиплејер игра на програмском језику Rust
Језик публикације, **ЈП**:		српски/ћирилица
Језик извода, **ЈИ**:		српски/енглески
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		2025
Издавач, **ИЗ**:		Ауторски репринт
Место и адреса, **МА**:		Нови Сад, трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		6/46/22/0/26/0/2
Научна област, **НО**:		Електротехничко и рачунарско инжењерство
Научна дисциплина, **НД**:		Примењене рачунарске науке и информатика
Предметна одредница/Кључне речи, **ПО**:		Шаблон, завршни рад, упутство
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		Овај документ представља упутство за писање завршних радова на Факултету техничких наука Универзитета у Новом Саду. У исто време је и шаблон за Typst.
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		01.01.2025
Чланови комисије, **КО**:	Председник:	Др Петар Петровић, ванредни професор
	Члан:	Др Марко Марковић, доцент
	Члан:	Потпис ментора
	Члан, ментор:	Др Игор Дејановић, редовни професор

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	Monographic publication
Type of record, **TR**:	Textual printed material
Contents code, **CC**:	
Author, **AU**:	Aleksa Vukomanović
Mentor, **MN**:	Igor Dejanović, Phd., full professor
Title, **TI**:	RustyArena - a multiplayer game written in Rust
Language of text, **LT**:	Serbian
Language of abstract, **LA**:	Serbian/English
Country of publication, **CP**:	Republic of Serbia
Locality of publication, **LP**:	Vojvodina
Publication year, **PY**:	2025
Publisher, **PB**:	Author's reprint
Publication place, **PP**:	Novi Sad, Dositeja Obradovica sq. 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	6/46/22/0/26/0/2
Scientific field, **SF**:	Electrical and Computer Engineering
Scientific discipline, **SD**:	Applied computer science and informatics
Subject/Key words, **S/KW**:	Template, thesis, tutorial
UC	
Holding data, **HD**:	The Library of Faculty of Technical Sciences, Novi Sad
Note, **N**:	
Abstract, **AB**:	This document provides guidelines for writing final theses at the Faculty of Technical Sciences, University of Novi Sad. At the same time, it serves as a Typst template.
Accepted by the Scientific Board on, **ASB**:	
Defended on, **DE**:	01.01.2025

Defended Board, **DB**:		
	President:	Petar Petrović, Phd., assoc. professor
	Member:	Marko Marković, Phd., asist. professor
	Member:	
	Member, Mentor:	Igor Dejanović, Phd., full professor

Menthor's sign

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

ИЗЈАВА О НЕПОСТОЈАЊУ СУКОБА ИНТЕРЕСА

Изјављујем да нисам у сукобу интереса у односу ментор – кандидат и да нисам члан породице (супружник или ванбрачни партнер, родитељ или усвојитељ, дете или усвојеник), повезано лице (крвни сродник ментора/кандидата у правој линији, односно у побочној линији закључно са другим степеном сродства, као ни физичко лице које се према другим основама и околностима може оправдано сматрати интересно повезаним са ментором или кандидатом), односно да нисам зависан/на од ментора/кандидата, да не постоје околности које би могле да утичу на моју непристрасност, нити да стичем било какве користи или погодности за себе или друго лице било позитивним или негативним исходом, као и да немам приватни интерес који утиче, може да утиче или изгледа као да утиче на однос ментор-кандидат.

У Новом Саду, дана _____

Ментор

_____

Кандидат

_____

Садржај

Глава 1

Увод

1.1 Мотивација

Развој мултиплејер видео-игара представља једну од интересантнијих области софтверског инжењерства. Почевши од једноставних текстуалних игара и LAN пинг-понг игре, до великих мултиплејер система са масовним бројем играча. Мултиплејер игре стално померају границе у погледу архитектуре, перформанси и интеракције између играча. Архитектуре мултиплејер система, комуникација између сервера и клијената, синхронизација стања света и дизајн протокола представљају комбинацију теорије и праксе у софтверском инжењерству.

Историја мултиплејер видео игара може се посматрати кроз неколико етапа [1]:

- 1970-1980: Прве текстуалне и једноставне LAN игре, као што су *Colossal Cave Adventure* [2] и *Pong* [3], које су омогућавале ограничену интеракцију и радиле на локалним системима.
- 1990-те: Прве онлајн игре и локалне мреже, игре попут *Doom* и *Quake* омогућавале су играчима да се такмиче преко Интернета или локалне мреже.
- 2000-те: Масовне мултиплејер игре и MMORPG-ови, као што су *World of Warcraft*[1] и *Counter-Strike*[2], захтевале су сложеније серверске архитектуре и бољу синхронизацију играча.
- Данас: Мултиплејер игре користе напредне архитектуре које омогућавају хиљадама играча да истовремено учествују, често користе *cloud* и мобилне платформе, као и сложене мреже за приказ у реалном времену.

Архитектуре оваквих система су интересантне због сложености комуникације, захтева за синхронизацијом и потребе за оптимизацијом и перформансама у реалном времену. Важно је напоменути да компаније које развијају комерцијалне мултиплејер игре често не откривају детаље имплементације, што чини ову област мање транспарентном и повећава значај истраживања и експеримената у оквиру академских пројеката.

1.2 Rust и пројекат

Пројекат RustyArena има за циљ истраживање могућности развоја мултиплејер игре користећи програмски језик *Rust*. Познат по безбедносном механизму за руковањем

[1] https://worldofwarcraft.blizzard.com/en-us/
[2] https://www.counter-strike.net/

меморијом, конкурентности и перформансама, *Rust* представља погодан избор за креирање стабилних и ефикасних мрежних система у видео-играма.

Пројекат кроз реализацију једноставне мултиплејер игре испитује колико Rust може да олакша развој, обезбеди сигурност и одржава висок ниво перформанси у контексту мултиплејер видео-игара.

Додатна инспирација потиче са сајта “*Are we game Yet?*”[3], који прати развој алата и екосистема за развој игара у *Rust*-у, пружајући увид у доступне библиотеке, фрејмворке (енг. *framework*) и ресурсе, и потврђује да Rust нуди све потребне копмоненте за развој игара.

1.3 Методологија

Рада приказује процес развоја игре RustyArena, укључујући архитектуру система, имплементацију мрежне логике, као и евалуацију применљивости и предности могућности *Rust*-а у контексту мултиплејер игара.

1.4 Структура рада

Рад је организован у неколико поглавља која представљају различите аспекте развоја видео-игре. Прво се обрађују теоријске основе неопходне за разумевање решења, након чега следи детаљнији увид у имплементацију система, са посебним осврtом на комуникацију и синхронизацију између њих. Након тога се представља визуелизација игре и анализа имплементираног решења, како би се оцениле ефикасност и стабилност решења. На самом крају, закључно поглавље сумира постигнуте резултате, анализира предности и ограничења пројекта и даје смернице за будућа унапређења.

[3] https://arewegameyet.rs/

Глава 2

Спецификација система и теоријске основе

2.1 ОСИ модел и мрежни протоколи

OSI модел представља концептуални модел који описује како различити делови мрежних система комуницирају један са другим. Модел је подељен на седам слојева: физички слој, слој везе, мрежни слој, транспортни слој, слој сесије, слој презентације и слој апликације. У контексту мултиплејер игара, посебан акценат ставља се на транспортном слоју, јер они одређују како подаци путују од сервера до клијента и обрнуто. Два најчешћа протокола су TCP (*Transmission Control Protocol*) и UDP (*User Datagram Protocol*) [4].

IP је протокол мрежног слоја који омогућава пренос и рутирање података преко интернета. Подаци се деле на мање јединице – пакете, при чему сваки пакет садржи информације о адресама извора и одредишта, што омогућава његово правилно усмеравање кроз мрежу. Сваки повезан на интернет има своју јединствену IP адресу, а пакети се шаљу и примају управо на основу тих адреса. Када пакет стигне на одредиште, његова обрада зависи од транспортног протокола који се користио. Најчешћи мрежни протоколи транспортног слоја су TCP и UDP [5].

TCP је поуздан протокол који обезбеђује да сваки пакет података стигне до примаоца у исправном редоследу, контролише грешке и поново шаље изгубљене пакете. TCP се користи за податке који морају бити потпуно тачни и где је битно да порука буде испоручена. Добар пример коришћења овог протокола су информације о пријављивању на систем, системи за дописивање или други критични подаци који не смеју бити изгубљени [6].

UDP је протокол без гаранције редоследа или поузданости испоруке пакета. Углавном је много бржи од TCP-а јер не врши потврду примања и нема контролу грешака. Због тога је UDP погодан за податке у реалном времену у мултиплејер видео-играма, као што су позиције играча, акције или брзи догађаји, где је важнија брзина од поузданости. Још једно место где се UDP користи је у системима за ћаскања уживо, пример оваквог система је *Discord* [7]. Често се за UDP може пронаћи израз “*пошаљи и заборави*” (енг. “*fire and forget*”) [8].

У мултиплејер играма, иако UDP преовладава, могућа је и комбинација оба протокола: TCP за критичне податке, а UDP за ажурирање стања света у реалном времену. Ова комбинација омогућава и поузданост и висок ниво перформанси.

2.2 Rust

Rust је модеран програмски језик који се истиче по безбедном руковању меморијом, конкурентности и високом нивоу перформанси, где се често пореди са перформансама које достижу C и C++. Ове особине га чине изузетно погодним за развој мултиплејер игара, где је потребно управљати великим бројем истовремених веза и обрађивати податке у реалном времену.

Један од кључних предности *Rust*-а је безбедност на нивоу компајлера, који спречава грешке попут двоструког ослобађања меморије или међусобног блокирања (енг. *deadlock*). Концепти као што су власништво, позајмљивање и животни век омогућавају креирање стаблиних и поузданих система без додатних провера у време извршавања. Све ове функционалност обезбеђене су од стране *borrow checker*-а.

У развоју RustyArena видео-игре коришћени су различити библиотечки сандуци:
- *bincode* - ефикасна серијализација података, корисна за слање стања игра преко мреже [9].
- *serde* - за флексибилну серијализацију и десеријализацију сложених структура података [10].
- *tokio* - асинхроно извршно окружење, олакшава конкурентно програмирање и комуникацију коришћењем асинхроних задатака [11].
- *gdext/godot* - интеграција Rust кода у *Godot* сцене уместо *GDScript*-а [12].
- *SQLx* - сандуку који олакшава комуникацију са базом података [13].
- *axum* - једноставан фрејмворк за развој веб апликација [14].
- *reqwest* - HTTP клијент високог нивоа [15].

У контексту мултиплејер игара, *Rust* пружа стабилан и ефикасан начин за имплементацију серверске стране, асинхроне обраде порука и управљање ресурсима. Поменута библиотека *gdext* нам омогућава коришћење *Rust* кода и на клијентској страни.

2.3 *Godot* и *Godot extensions*

Godot engine је софтвер отвореног кода (енг. *open-source*) који се користи за развој 2Д и 3Д игара бесплатан за коришћење. Познат је по својој једноставаности и архитектури заснованој на сценама. Једна од његових највећих предности је сценски систем (енг. *scene tree*), који представља хијерархију објеката (чворова) који заједно граде свет игре. Сваки чвор има одређену улогу – може представљати визуелни елемент, део корисничког интерфејса (UI, енг. *user interface*), физички објекат или извор звука. Оваква структура омогућава модуларност и поновну употребу компоненти, што знатно олакшава развој и проширивост игре [16].

У контексту приказа света, *Godot* комбинује сцене у једно стабло чворова које се извршава током игре. Свака сцена може да садржи друге сцене, што омогућава да сложене сцене разложимо да подскуп сцена.

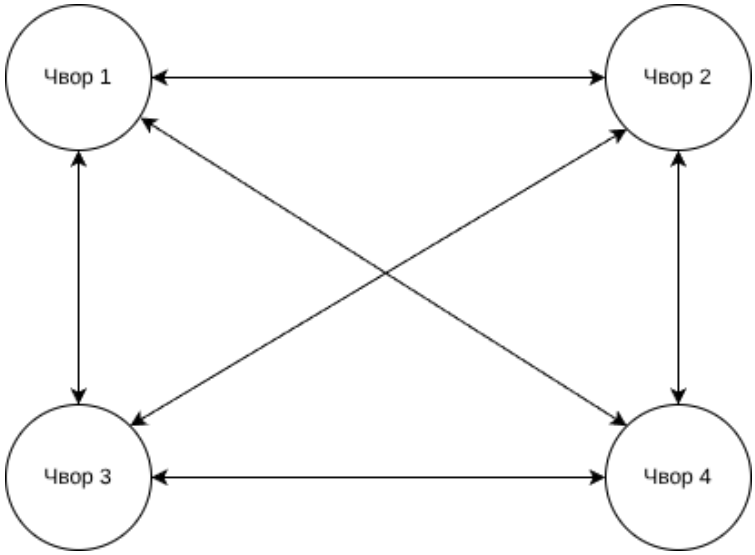
Подразумевани скриптни језик у *Godot*-у је *GDScript*, који је синтаксно сличан Пајтон-у и намењен је брзом развоју логике. Ипак за пројекте који захтевју високе перформансе или ближи контакт са системским ресурсима, постоји могућност коришћења других програмских језика преко *bindings* система. *Godot* је претежно писан у C++ језику, па *bindings* за овај програмски су релативно документовани и подржани, док за језик по пут *Rust*-а и подршка и документација су доста слабији.

Овај приступ комбинује најбоље од оба света:
- *Rust* обезбеђује ефикасност, стабилност и функционалност *borrow checker*-а [17].
- *Godot* пружа интерактивно окружење, визуелну презентацију и брз развој пројекта

Захваљујући томе, представља пример како се алати отвореног кода могу комбиновати ради постизања професионалних резултата уз потпуну контролу.

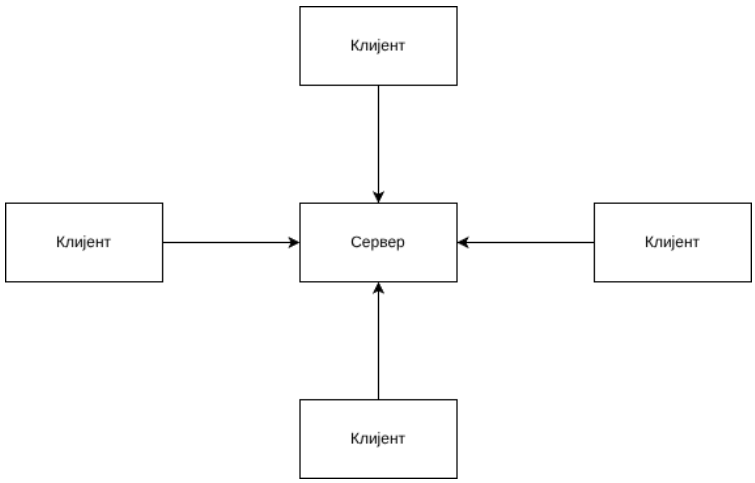
2.4 Мрежна архитектура видео-игара

У почетку развоја мултиплејер видео-игара преовладавао је *peer-to-peer* комуникациони модел. *Peer-to-peer* модел представља децентрализован комуникациони модел, где сваки чвор у мрежи има једнака права, не постоји специјални чвор. Ослања се на потпуно повезану мрежасту топлогију (енг. *fully connected mesh topology*). Овај модел за комуникацију се још увек користи у појединим играма, пример су игре стратегије у реалном времену (RTS, енг. *real-time strategy*) [18]. На слици 1 је приказан изглед овакве архитектуре.



Слика 1: *Peer-to-peer* комуникациони модел

Модел клијент-сервер представља дистрибуирану структуру апликације који раздваја задатке или количину посла на пружаоца ресурса и потрошача [19]. Слика 2 илуструје овакву архитектуру.



Слика 2: Клијент-сервер архитектура

У овом моделу постоји јасна подела улога:

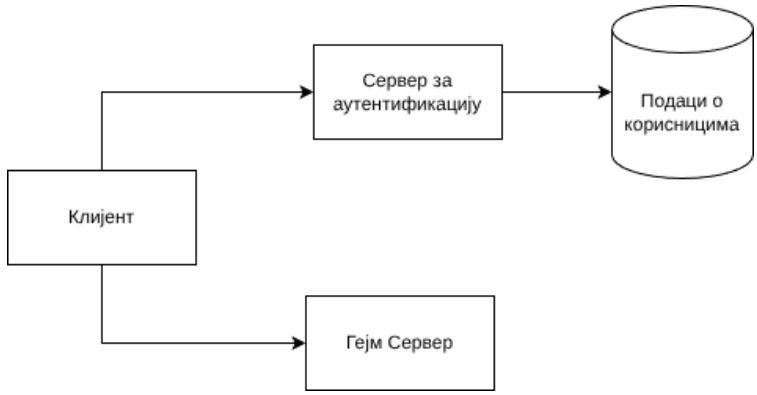
- Сервер - управља стањем света игре, валидира улазе и синхронизује клијенте
- Клијенти - шаљу своје уносе, као што су покрет и акције, серверу и приказују стање света које сервер шаље

Ауторитативан сервер (енг. *authoritative server*) представља централну компоненту која има потпуну контролу над светом игре и представља једини извор истине (енг. *single source of truth*). Свака акција клијента пролази кроз сервер који проверава њену испрвност, након чега сервер ажурира стање [20].

Овакав приступ омогућава:
- сигурност – клијент не може изменити стање самостално
- доследност између свих играча
- једноставнију синхронизацију јер сервер диктира стање игре

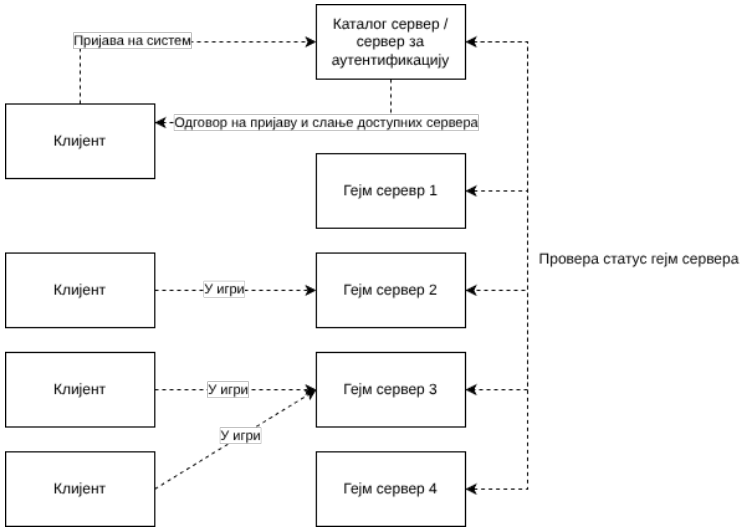
Слика 3 приказује архитектуру RustyArena пројекта на високом нивоу апстракције која обухвата неколико кључних компоненти:

- Сервер за игру (енг. *game server*) - имплементиран у *Rust*-у, задужен за управљање стањем света, обрадом логике и слање UDP порука клијентима.
- Клијент за игру (енг. *game client*) - реализован у *Godot engine*-у, шаље команде серверу и визуелизује добијене податке, често се може пронаћи назив “*dumb clients*”.
- Сервер за аутентификацију (енг. *authentication server*) - независан модул који верификује играча пре приступа серверу за игру, чиме се осигурава контролисани приступ као и распоређивању играч у сесије.



Слика 3: Архитектура пројекта RustyArena на високом нивоу апстракције

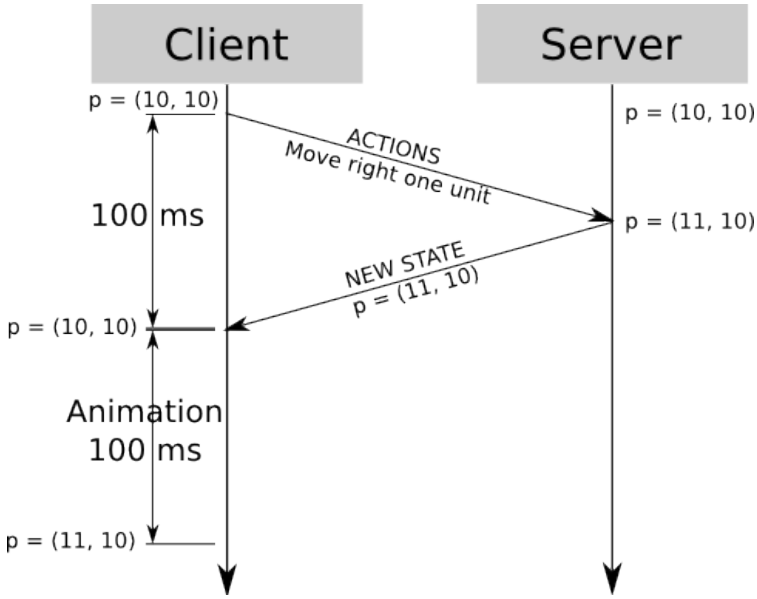
Ово даје добру основу за наставак пројекта, где би следећи корак био додавање више инстанци сервера. Сервер за аутентификацију се може користити као сервер који садржи информације о осталим гејм серверима. Када се корисник пријави на систем, аутентификациони сервер му може понудити листу тренутно активних сервера. Пример овакве архитектуре је приказан сликом 4.



Слика 4: Архитектура која укључује више гејм сервера

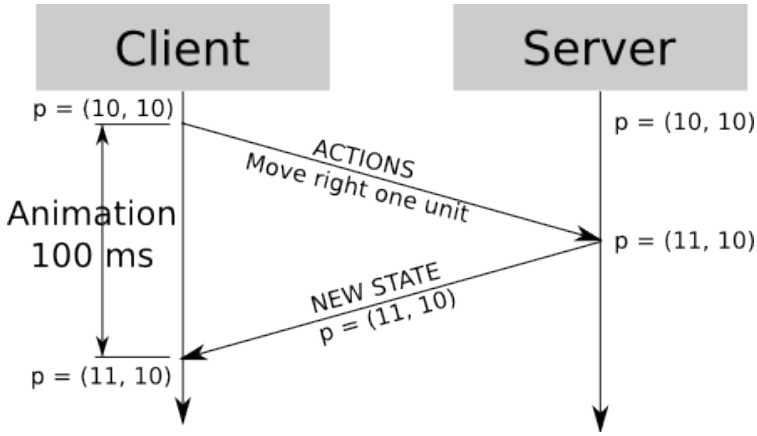
2.5 Технике за синхронизацију у мултиплејер играма

У мултиплејер видео-играма, корисничко искуство (енг. *user experience* - UX) је поред самог корисничког интерфејса повезано са начином на који систем обрађује комуникацију између клијента и сервера, и како синхронизује стање света између њих. Кашњење (енг. *latency*) и губитак пакета (енг. *packet loss*) могу значајно утицати на игру, неопходно је применити различите технике које минимизују њихов утицај на перцепцију играча. Слика 5 приказује овај проблем.



Слика 5: Приказ проблеме које уводи кашњење одговора [21]

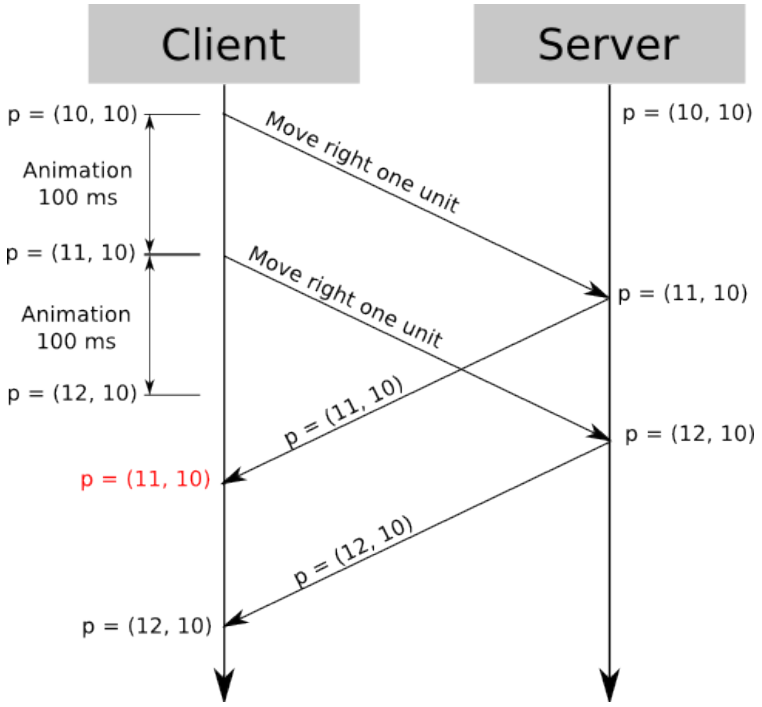
Предикција на клијентској страни (енг. *client-side prediction*) - ова техника подразумева да уколико је игра довољно детерминистичка, клијент може да предвиди резултат одређене радње пре него што сервер потврди њену исправност и ажурира стање света. На пример када играч помери свог лика у одређеном правцу, клијент одмах прикзује ту промену, без чекања на одговор сервера [21]. Слика 6 приказује како ово изгледа.



Слика 6: Основне идеје о предикцији на клијентској страни [21]

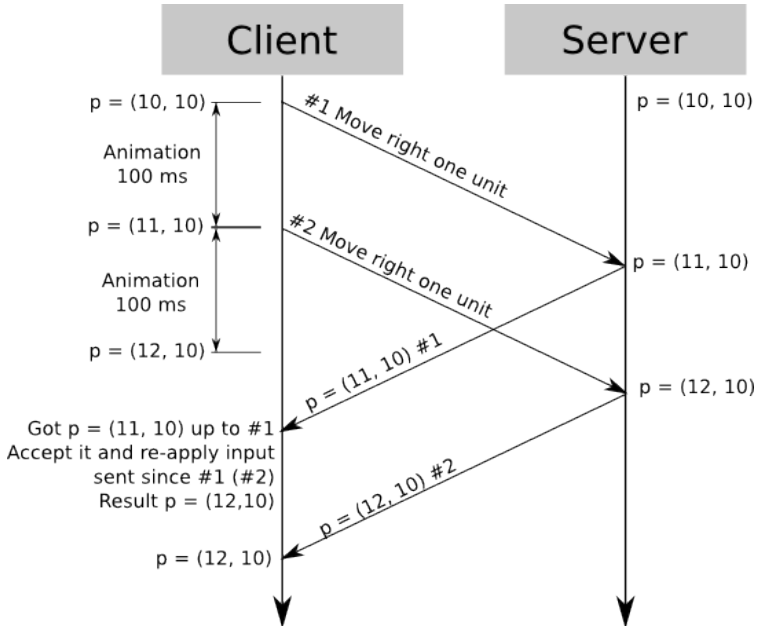
Када сервер касније пошаље званично стање, клијент га упоређује с својом предикцијом, и уколико постоји разлика врши интерполацију и корекцију. Наведена техника важи под претпоставком да ће команда послата серверу бити упсешно извршена. Ово омогућва играчима да имају осећај тренутног одговора, чак и при већим кашњењима,

али уводи додатан проблем. Ако корисник пошаље две команде једну за другом, пре него што је добио потврду од сервера за прву команда. Ово може произвести ефекат да играч “скаче” током игре [21]. Слика 7 описује овај проблем.



Слика 7: Проблема који настаје наивном имплементацијом [21]

Усаглашавање са сервером (енг. *server reconciliation*) - је техника која омогућава клијенту да након исправке локалног стања, коришћењем података добијених од сервера, поново примени све радње које су се догодиле у међувремену. То спречава нагле скокове и промене положаја играча (енг. *jittering*). У пракси, клијент чува историју својих улаза и када добије податке о стању на серверу, он враћа стање на серверско, а затим поново репродукује све улазе који су се догодили након тог тренутка [21]. На слици 8 је приказано шта ова техника решава.



Слика 8: Проблем који се решава техником усаглашавања са сервером [21]

2.6 Функционални захтеви

RustyArena је мултиплејер видео-игра са аспектом борбе у реалном времену. Изгледом подсећа на чувену игру Астероиди (енг. *Asteroids*). Сваки играч управља свемирским бродом, где је циљ игре уништити бродове осталих играча и истовремено избегавати астероиде који се крећу по мапи.

Систем испуњава основне функционалне захтеве:

- Омгућити кориснику регистрацију на систем коришћењем корисничког имена и лозинке
- Омогућити кориснику да се пријави на систем како би могао покренути игру
- Основни гејмплеј игре који обухвата:
  - кретање играча на мапи
  - могућност да корисник пуца на друге играче
  - животне поене
  - астероиде које играчи избегавају или уништавају
  - ресетовање играча након што је његов брод уништен

2.7 Нефункционални захтеви

Нефункционални захетви које систем обухвата:

- Систем подржава рад са више истовремених клијената
- Синхронизација између клијената и сервера
- Видео-игра треба да буде стабилна

Глава 3

Имплементација

У овом поглављу описана је детаљнија имплементација игре. Архитектура и имплементација апликације зависи од самог домена, односно конктретне видео-игре која се имплементира и њеног гејмплеја (енг. *gameplay*). Ако посматрамо пример неке игре са картама попут игре UNO[4], за овкав тип игре није нам критично време одзива сервера. Односно ако одговор закасни и пар секунди, играч се неће осетити ошећено, као да је игра намештена. Код других игара које захтевају брзо ажурирање стања света, попут игре *Valorant*[5], мало кашњење може произвести лоша корисничка искуства. Због тога је важан добар одабир протокола за комуникацију и планирање унапред, јер различити жанрови видео-игара захтевју другачију архитектуру система.

3.1 Дељени код

Bincode и *serde* су *Rust* библиотечки сандуци који олакшавају серијализацију и десеријализацију *Rust* типова, као и самостално дефинисаних структура и енумерација. Све заједничке структуре које се користе у комуникацији смештене су у пројекат *common* како би смањили дуплирање кода. Све што се серијализује мора да имплементира одговарјуће *Rust* особине (енг. *traits*) *Serialize* и *Deserialize*. Пример дефинисаног модела је дат у листингу 1.

```rust
use bincode::{Decode, Encode};

#[derive(Encode, Decode, Debug, Clone)]
pub struct Player {
    pub id: u32,
    pub x: f32,
    pub y: f32,
    pub rotation: f32,
    pub vx: f32,
    pub vy: f32,
    pub hp: u16,
    pub last_shot_ms: u64,
    pub fire_rate_ms: u64,
    pub last_processed_input_seq: u32,
}
```

Листинг 1: Пример пакета који се користи за комуникацију

[4] https://www.unorules.com/

[5] https://playvalorant.com/en-gb/

3.2 Серверска страна

Game server је ауторитативан сервер на коме се врши симулација света игре. При покретању, сервер отвара два сокета (енг. *socket*), један од њих је намењен за комуникацију преко TCP-а док је други намењен за комуникацију преко UDP-а. Комуникација преко, односно операције читања и писања порука могу бити дуге и блокирајуће операције, ово се може решити ако сервер имплементирамо као вишенитни сервер (енг. *multi-threaded*). Коришћењем *Tokio* сандука можемо раздвојити обраду порука у одвојене асинхроне задатке. Конкретни асинхрони задаци подразумевају:

- Обраду TCP захтева и слање одговара
- Обраду долазних UDP захтева и прослеђивање истих симулацији света
- Емитовање (енг. *broadcasting*) стања света свим повезаним играчима
- Симулацију света игре, такозвани “*game loop*”

3.2.1 Обрада захтева

Обрада TCP захтева обухвата пријем порука, обради и по потреби прослеђивању ка нити која симулира игру коришћењем канала (енг. *channel*). У тренутној имплементацији се овај механизам користи како би играч обавестио сервер да жели да приступи игри, али је могуће додати нове функционалност које захтевају поузданост.

Обрада долазних UDP захтева је слична обради код TCP захтева. Долазне поруке су дефинисане у пројекту *common*, које се путем канала за асинхрону комуникацију прослеђују даље задатку која симулира игру. Ове поруке су заправо команде које играчи шаљу како би мењали стање света. Пре него што дође до промене стања, команда се валидира како бисмо спречили злонамерне кориснике да варају у играма и заобилазе правила игре. На листингу 2 прикзан је модел података команде која се шаље када корисник притисне одговарајући тастер.

```rust
#[derive(Encode, Decode, Clone, Debug, Copy)]
pub enum InputAction {
    RotateLeft,
    RotateRight,
    Shoot,
    Thrust,
    Hello,
}

#[derive(Encode, Decode, Clone, Debug)]
pub struct PlayerInput {
    pub id: u32,
    pub seq: u32,
    pub action: InputAction,
}
```

Листинг 2: Модел података команде

Иако корисник приликом првог приступа серверу шаље TCP поруке, играч мора послати још једну поруку преко UDP конекције како би сервер био обавештен на коју адресу (IP и порт) треба да шаље стање света како би играчи имали ажуран приказ игре. Приликом прве команде, клијент јавља серверу да је спреман да приказује игру кориснику и на коју адресу треба да га обавештава о стању игре. За поступак обавештавања на серверу задужен је *broadcasting* асинхрони задатак. Он периодично емитује стање света играчима који су забележени као активни када нит која симулира свет то затражи.

3.2.2 *Game loop*

Game loop је бесконачна петља која симулира свет и правила игре. Она ради фиксном фреквенцијом, која одређује број извршених итерација у једној секунди. Често се узима вредност од 60Hz, што значи да једна итерација траје око 16ms. Да бисмо осигурали да симулација не тече брже, проверавамо време између итерација петљи и, по потреби, успаваљујемо нит на преостали део периода. Псеудо код који представља овакву архитектуру дат је испод.

```
loop {
    // Провери да ли постоји нови играч
    // Провери да ли постоје нове команде
    // Симулирај свет за једну итерацију
    // Емитуј ново стање света свим активним играчима
    // Сачекај преостали период ако за тим има потребе
}
```

Листинг 3: Апстрактан преглед *game loop*-а

Проверавамо прво да ли постоје нови играчи који желе да се прикључе игри, након чега проверавамо да ли постоје команде од стране играча које треба обрадити. На крају ажурирамо стање игре и шаљемо тренутно стање свим активним играчима. Како је тренутна сепцификација игре једноставна, можемо приуштити да у свакој итерацији емитујемо целокупно стање света, ово се касније може изменити и оптимизовати у зависности од будућих захтева.

Унутар сваке итерације петље имамо ажурирање света, симулирамо физику, кретања, покушавако да детектујемо колизије (енг. *collision detection*) и након тога проверавмо правила игре, као што је правило о смањивању животних поена (енг. *health points*).

3.2.3 Ентитети и њихова ажурирања

Конкретно код RustyArena игре прво ажурирамо позиције ентитета, односно играча, астероида и метака. Играч има могућност ротације, као кретања лево и десно, и има могућност да се лансира и креће у смеру и правцу одређеним његовом положајем. Положај играча је представљен *x* и *y* координатама и углом ротације - угао који заклапа са позитивним делом *x*-осе. Када играч притисне тастер за кретање напред сервер

заправо том играчу мења брзину кретања по осама у односну на његов положај, његов глобални положај у игри ажурира се тек када се ажурира свет. Када играч отпусти тастер за кретање напред, брод се не зауставља одмах већ наставља кретање, али успорава постпено до заустављања. Свет је ограничен са 4 стране, тако да играч не може прећи изван тих граница већ ће привидно деловати као да постоји невидљиви зид. Имплементација кретања дата је у листингу 4.

```
pub fn update_player_position(&mut self) {
    // Померање енетитета по x и y оси
    self.x += self.vx;
    self.y += self.vy;

    // У случају да прелазе границе терана постави их на ивицу
    self.x = self.x.clamp(-800.0, 800.0);
    self.y = self.y.clamp(-600.0, 600.0);

    // Симулирај успоравање
    self.vx *= 0.9;
    self.vy *= 0.9;
}
```

Листинг 4: Кретање играча

Осим кретања, играч има могућност да пуца како би уништио бродове других играча. Када играч пртисне тастер за пуцање, сервер на основу координата и ротације брода рачуна правац и брзина кретања метка по *x* и *y* оси. Поред ових параметара бележи се и пређени пут метка како бисмо га након неког времена да уништили, као и време последње испаљеног метка, како бисмо ограничили број пуцања.

Поред метака и бродова као игача, имамо и астероиде. Астероиди уводе ефекат насумичности који повећава изазов игре како игра не би брзо досадила. Иницијализују се на следећи начин:

- На случајан начин се бира једна ивица света на којој ће се астероид иницијализовати.
- Након што је ивица одабрана, на случајан начин бира се положај на тој ивици и додајемо одређену маргину у случају да се неки играч нађе на ивици.
- За правац кретања астероида се прво на случајан начин бира неко од играча, након чега се астероиду додељује брзина кретања која тако што се прво пронађе јединични вектор између играча и почетне позиције астероида након чега се множи са брзином кретања.

Као и код испаљених метака постоји пређени пут који се рачуна при свакој итерацији чија вредност се користи за уклањање астероида који прелазе неку границу. Само кретање испаљених метака и астероида је нешто једноставније, односно не постоји скалирање брзине током времена како бисмо симолулирали успоравање.

3.2.4 Детекција колизија

Након ажурирања положаја свих ентитета, сервер проверава да ли је дошло до колизија између одређених ентитета. У имплементацији коришћен је једноставан алгоритам, посматра се удаљеност центара између ентиета, и уколико је раздаљина мања од неког прага сматра се да су се ентиетити сударили. Конкретно сервер проверава:

- Да ли је неки метак погодио неки астероид
- Да ли је неки метак погодио неког играча, под условом да то није играч који је испалио метак
- Да ли је неки астероид погодио неког играча

Свака колизија се бележи након чега се примењују правила игре. Конкретна правила игре су следећа:

- Ако је метак погодио астероид, и астероид и метак се тада уништавају.
- Ако је метак погодио неког играча, метак се уништава док се погђеном играчу одузимају животни поени. Конкретна се одузима 20 поена, али је ову вредност могуће конфигурисати.
- Ако је астероид погодио неког играча, астероид се уништава, а играчу се смањује број животних поена за одређену суму.
- Ако неко од активних играча има 0 животних бодова, сматрати да је он уништен и иницијализовати га на нову позицију.
- Иницијализују астероиде ако постоје услови за то. Конкретно астероиде инцијализујемо након сваких 1000ms како не бисмо преплавили игру астероидима. Ово је такође нешто што се може прилагодити.

3.3 Клијентска страна

Сама имплементација у *Godot*-у заснива се на томе што алат допушта проширивање путем екстензија за *Rust*. Уместо да се код компајлира у апликацију или библиотечки сандук, заправо креирамо динамичку библиотеку која излаже интерфејс у програмском језику C. Динамичка библиотека се учитава у *Godot* у време извршавања, преко *GDExtension* интерфејса.

Олакшавајућа околност је то што можемо користити сандуке попут *tokio* и *reqwest* на клијентској страни. Ово је посебно битно јер омогућава неблокирајуће слушање порука од главног сервера. Из перспективе корисника, генерално није пожељно да блокирамо игру док чекамо или обрађујемо одговор од сервера. Постоје изузеци ако је одговор од сервера потребан за даљи наставак, али и у таквим ситуацијама пожељно је не блокирати програм у потпуности.

3.3.1 Основне функционалности

Елементи у *Rust*-у који се користе у *Godot* морају да имплементирају одређене особине или буду одређеног типа.

```
#[derive(GodotClass)]
#[class(base=CharacterBody2D)]
pub struct PlayerWrapper {
    base: Base<CharacterBody2D>,
    ...
}
```

Листинг 5: Пример наслеђивања *Godot* класе у *Rust* коду

Особине које су наведене у исечку кода означавају да дата структура наслеђује одређену *Godot* класу. Иако *Rust* није објектно оријентисани језик, на овај начин се симулира наслеђивање. Поред дефинисања структура, може се имплементирати интерфејс који наслеђена класа поседује. Ово је некада обавезно, посебно `init` метода која се користи за иницијализацију чвор. Пример је дат у листингу 6.

```
#[godot_api]
impl ICharacterBody2D for PlayerWrapper {
    fn init(base: Base<CharacterBody2D>) -> Self {
        Self {
            base,
            ...
        }
    }

    fn ready(&mut self) {
      ...
    }

    fn process(&mut self, delta: f64) {
      ...
    }

    fn physics_process(&mut self, delta: f64) {
      ...
    }
}

#[godot_api]
impl PlayerWrapper {
    #[func]
    pub fn do_something(&mut self) {
      ...
    }
}
```

Листинг 6: Пример функција које *Godot* структура имплементира

Важно је разумети ове структуре јер већина алата за развој видео игара се базира на њима:

- *ready* - функција се позива само једном када сцена први пут постане видљива унутар *scene tree*-а.
- *process* - функција се позива сваки пут када се фрејм (енг. *frame*) исцртава, варира од брзине процесора.
- *physics_process* - функција се позива на фиксан период, погодан за обраду и симулацију физике.

Поред имплементације интерфејса за неку *Godot* класу, могу се дефинсати и додатне методе унутар имплементационог блока дефинисане структуре.

3.3.2 Мрежна комуникација

RustyArena поседује две структуре чији фокус је комуникација са серверима. Прва структура је *NetworkClient* који се користи за комуникацију са сервером на коме се игра симулира. При покретању игре учитава се адресе на којима се налазе сервери. Код у листингу 7 представља имплементацију отварања сокета за UDP комуникацију. За адресу наведена је локална адреса (0.0.0.0), док за порт је постављена нула што означава коришћење било ког слободног порта.

```
...
pub fn connect_to_server(&mut self) {
    let addr = &self.game_server_address_udp; // адреса је учитана у
објекту пре позива методе
    let sock_future = async move {
        match UdpSocket::bind("0.0.0.0:0").await { // отварамо сокет
            Ok(sock) => {
              // враћање резултата
            }
            Err(_) => {
              // руковање грешком
            }
        }
    };
    let socket = AsyncRuntime::block_on(sock_future); // чектање на одговор
да ли је сокет успешно отворен
}
...
```

Листинг 7: Код који отвара конекцију са сервером

Такође при слању команди на притисак тастера иницијализујемо посебан задатак за слање поруке како би игра остала респонзивна. Сервер је имплементиран тако да не враћа одговор кориснику на његову команду, већ само мења стање игре, а корисник ће видети промене када сервер пошаље стање игре свим играчима.

```
    pub fn send_input(&self, id: u32, seq: u32, action_code: u32) {
        let socket = self.socket.clone();
        let input = PlayerInput { // спремање команде
            ...
        };

        let input_bytes = ... // серијализација команде у низ бајтова
        AsyncRuntime::spawn(async move {
            let _ = socket.unwrap().send(&input_bytes).await; // слање
преко мреже у посебном асинхроном задатку
        });
    }
```

Листинг 8: Код који обаља слање команде

Имплементација функције која шаље иницијалну поруку преко TCP конекције је дата следећем исечку кода. Сервер као одговор шаље исти идентификатор који је клијент послао као потврду да је сервер постао свестан новог играча.

```
    pub async fn send_handshake(&mut self) -> Result<u32, std::io::Error> {
        let auth_address = ... // добављање адресе и порта на коме сервер
ослушкује TCP саобраћај
        let mut stream = ... // отварање конекције за пријем и слање

        let id = self.controller_id;

        let request = ... // серијализација идентификатора
        stream.write_all(&request).await?; // слање идентификатора

        let mut buffer = ... // бафер у коме смештамо одговор
        stream.read_exact(&mut buffer).await?; // читање одговора
        let player_id = ... // конверзија у одговарајући тип (u32)
        Ok(player_id) // повратна вредност
    }
```

Листинг 9: Иницијална комуникација са сервером - *Handshake*

Након што је све иницијализовано, клијент је спреман да слуша поруке које сервер доставља како би визуализово стање игре. Користи се канал за асинхрону комуникацију. Оно што може бити проблематично је што типове које *Godot* пружа нису безбедни да се користе у конкурентним програмима, тако да ако желимо да емитујемо сигнал или приступимо неком *Godot* типу, то није могуће постићи на једноставан начин. Једно решење које је имплементирано је слање резултата путем канала у функцију `process` која проверава да ли постоји порука на коју треба одреаговати.

```
pub fn start_listening(&mut self) -> Option<UnboundedReceiver<GameWorld>> {
    let listen_sock = socket.clone();
    let (tx, rx) = unbounded_channel(); // канал који се користи за
асинхрону комуникацију

    AsyncRuntime::spawn(async move {
        ...
        loop {
            match listen_sock.recv(&mut buf).await {
                Ok(len) => {
                    if let Ok((world, _)) =
                        bincode::decode_from_slice::<GameWorld,
_>(&buf[..len], config) // конверзија из низа бајтова у конкретну структуру
                    {
                        if tx.send(world).is_err() { // слање поруке кроз
канал за асинхрону комуникацију
                            break;
                        }
                    }
                }
                Err(e) => {
                    godot_error!("Failed to receive snapshot: {}", e);
                    break;
                }
            }
        }
    });

    Some(rx) // канал који се користи у функцији 'process'
}
```

Листинг 10: Функција која отпочиње асинхрони задатак који се бави читањем пристиглог UDP саобраћаја

Друга структура која је намењена за комуникацију је *NetworkAPI* која се фокусира на комуникацију са сервером за аутентификацију. Методе која ова стркутура поседује су `login` и `register` које се баве пријавом и регистрацијом на систем. За комуникцију између чворова, *Godot* поседује посебан механизам сигнала. Овај мехнизам заправо омогућава имплементацију *observer* обрасца. У наведеним методама коришћењем *reqwest* сандука, шаље се захтев аутентификационом сервер и чека се одговор у посебној нити. Када одговор пристигне шаље се `process` методи *NetworkAPI* чвора коришћењем асинхроних канала за комуникацију. Затим се проверава тип одговора који је пристигао, и емитује се одговарајући сигнал. Сви елементи, функције које су претплаћене и слушају задати сигнал, ће се извршити. На овај начин можемо обезбедити ефикасну асинхрону комуникацију без блокирања апликације.

Разлог за раздвајање на две структуре је само због лакше одрживости, ако је потребно проширити једну страну.

3.3.3 Обрасци у развоју клијентске стране

Singleton образац се користи када је потребно да постоји једна инстанца класе у систему. Данас се често сматра анти-обрасцем (енг. *anti-pattern*) јер је тешко да се *mock*-ује приликом тестирања, а поред тога глобални објекти су често лош дизајн. Код развоја игара овај образац се може применити, добар пример је када играч треба прећи са једног нивоа на други, а желимо да сачувамо контекст и податке тренутног играча. Помоћу овог обрасца можемо чувати податке који су дељени, односно различите сцене захтевају приступ подацима или неким функционалностима. RustyArena имплементира овај образац на два места:

- *AsyncRuntime* - омогућава да креирамо само један *tokio Runtime* и једноставније креирање нових задатака.
- *NetworkClient* - омогућава да код одржавамо чистим јер различите сцене захтевају позиве метода ове структуре.

Проблеми који су настали приликом израде овог дела задатка су спорије решавани због тежине да се дебагује код који се конкуретно извршава, посебно на клијентској страни. Технике попут исписа порука на одређеним местима у коду овде неће радити, односно порука неће бити исписана у *Godot* конзоли. Дешава се да програм настви са извршавањем, иако је наишао на грешку.

3.4 Пробелем синхронизације

У мултиплејер играма неопходно је обезбедити да клијен и сервер остану синхронизовани, чак и када се команде извршавају асинхроно. Сервер је имплементиран тако да не шаље одговор на извршену команду, односно клијент не добија одговор од сервера за послату команду и да ли се она извршила. Разлог зашто је на овај начин одрађена имплементација је то што је игра борба у реалном времену и број пакета који се размењује огроман. Због тога је као протокол за комуникацију између клијената и сервера изабран UDP, уместо TCP, како би се смањило кашњење и избегли проблеми са редоследом испоруке.

Клијенту је доступно стање света на основу кога он може да закључи да ли се његова команда извршила или тако што ће приметити да је та команда донела промене. У овој игри, свака команда коју клијент шаље серверу садржи редни број команде, број који се инкрементује за сваку наредну команду, који означава редослед њеног слања. Иако не враћа одговор на команду, сервер за сваког играча чува информацију о последњој обрађеној команди.

Када клијент добије ново стање света од сервера, он проверав у својој локалној историји команди која је послења команда коју је сервер обрадио. Све команде које су

послте пре тог тренутка се одбацују, док се све накнадне команде поново примењују. На овај начин клијент коригује своју предикцију и остаје усклађен са серверским стањем, без потребе за експлицитним потврдама након сваке акције.

Исечак кода који имплементир овај механизам дат је у листингу 11.

```
pub fn reconcile_with_server(&mut self, server_player: Player, delta: f64)
{
    // ажурирање података корисника пристиглих са сервера
    ...

    // остављамо само команде које нису још увек обрађене од стране сервера
    let last_ack = server_player.last_processed_input_seq;
    self.pending_inputs.retain(|input| input.seq > last_ack);

    // команде које још увек нису обрађене извршавамо локално
    let unack = self.pending_inputs.clone();
    for input in unack.iter() {
        self.apply_local_input(input.action, delta);
    }
}

pub fn apply_local_input(&mut self, action: InputAction, delta: f64) {
    match action {
        // за акције корисника дефинишемо локалко понашање, због
једноставности користићемо имплементацију сличну серверској
    }
    // по потреби одрадити додатна ажурирања
    self.data.update_player_position();
}
```

Листинг 11: Предикција са клијентске стране и синхронизација са сервером

3.5 Сервер за аутентификацију

Сервер за аутентификацију имплементиран је коришћењем *Tokio Axum* и *SQLx* сандука. Његова главна сврха је да омогући корисницима приступ систему путем регистрације и пријаве, пре него што започну игру.

Кориснички подаци се чувају у *PostgreSQL* бази података, при чему се лозинке хешују (енг. *hashing*) како би се повећала безбедност. При свакој успешној пријви, сервер враћа индетификатор, који клијент касније користи за идентификацију током комуникације са сервером игре. Овај приступ је поједностављен и погодан за развојну фазу система, јер омогућава једноставну интеграцију са клијентом.

Комуникација са сервером обавља се путем HTTP протокола, а подаси се размењују у JSON формату. У будућности, овај систем може бити проширен коришћењем сесијских токена ради боље безбедности и контроле приступа.

За потребе развоја и тестирања обезбеђена је скрипта init.sql која креира потребне табеле. У *Rust*-у, ред у табели корисника представљен је структуром приказаном у листингу 12, док је пример DTO објекта дат у листингу 13.

```rust
#[derive(Debug, Serialize, Deserialize, sqlx::FromRow)]
pub struct User {
    pub id: i32,
    pub username: String,
    pub password_hash: String
}
```

Листинг 12: Модел података приказан у Rust-у

```rust
#[derive(Deserialize, Serialize, Clone)]
pub struct LoginRequest {
    pub username: String,
    pub password: String
}

#[derive(Deserialize, Serialize, Clone)]
pub struct AuthResponse {
    pub id: i32,
}
```

Листинг 13: DTO објекти у Rust-у

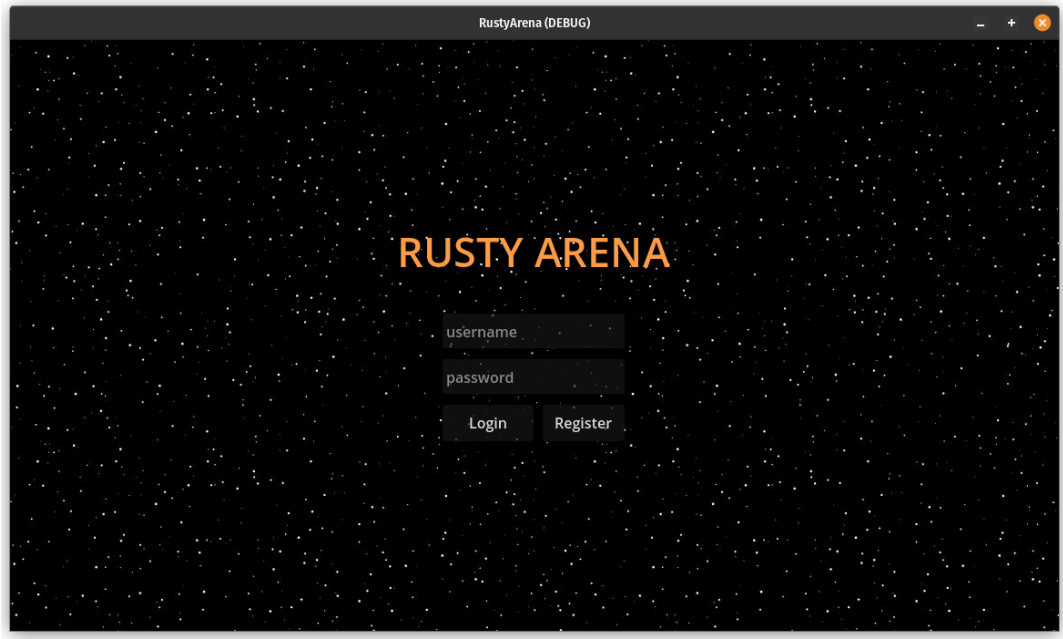
Глава 4

Визуелизација

Ово поглавље за циљ има да прикаже имплементиран систем, опише могућности унапређења корисничког интерфејса и побољшања корисничког искуства, као и проблеме који су постојали током имплементације решења.

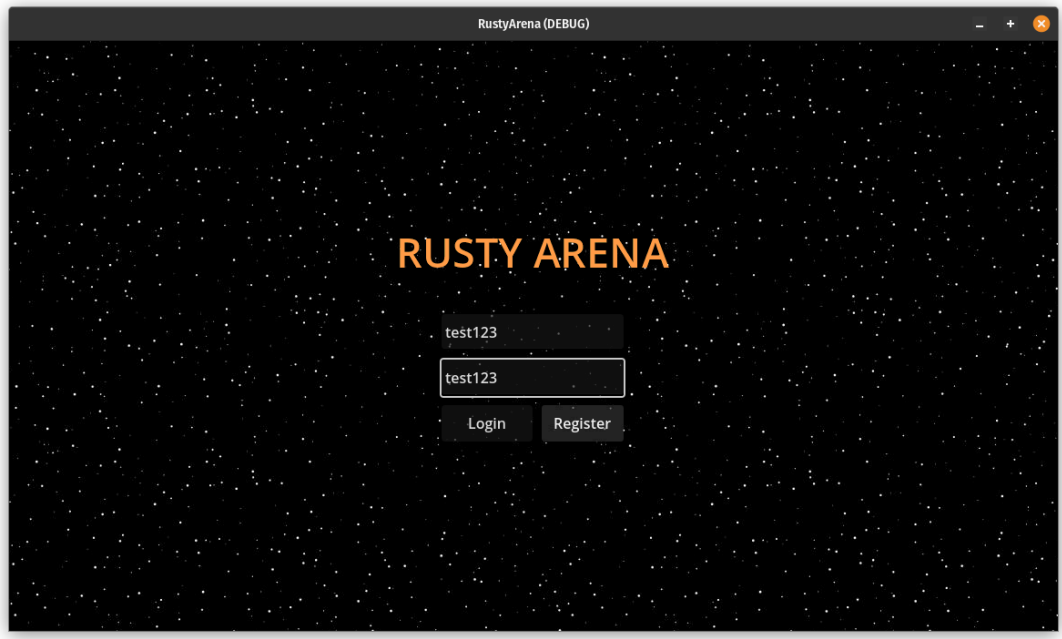
4.1 Главни мени

При покретању апликације, кориснику се прво приказује главни мени, који представља улазну тачку игре (енг. *entry point*). Мени омогућава избор опција као што су регистрација, пријава и покретање игре. На слици 9 је приказан изглед главног менија.



Слика 9: Главни мени

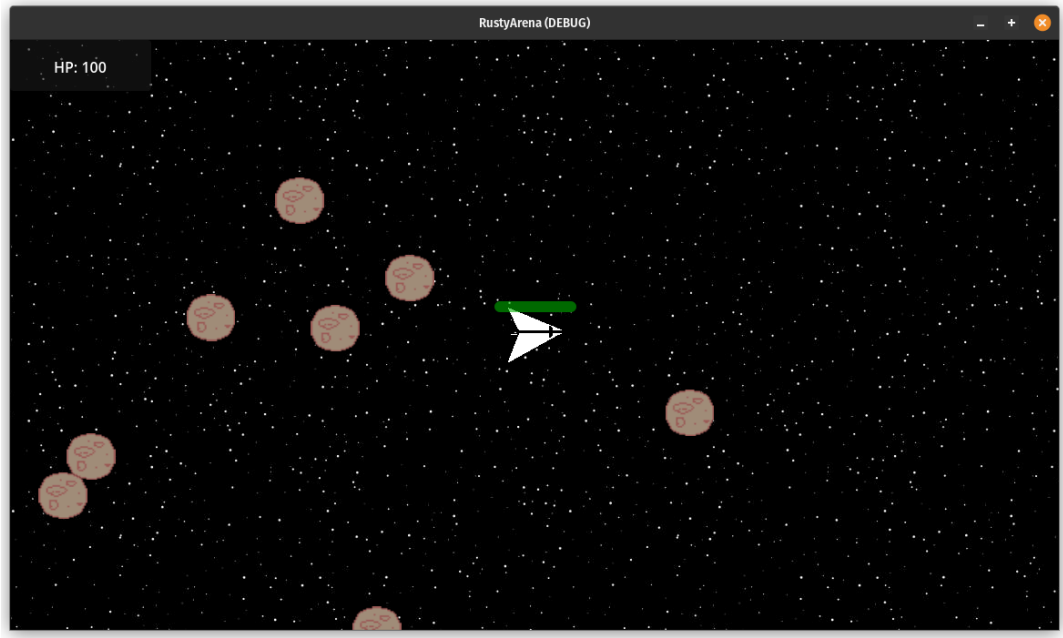
Корисник који први пут приступа игри има могућност регистрације, док се већ постојећи корисници могу пријавити уносом корисничког имена и лозинке. Форма за регистрацију и пријаву је реализована унутар *Godot* окружења, а комуникација са сервером је остварена помоћу мрежног модула који је описан у претходном поглављу. Слика 10 приказује овај екран.



Слика 10: Пријава и регистрација на систем

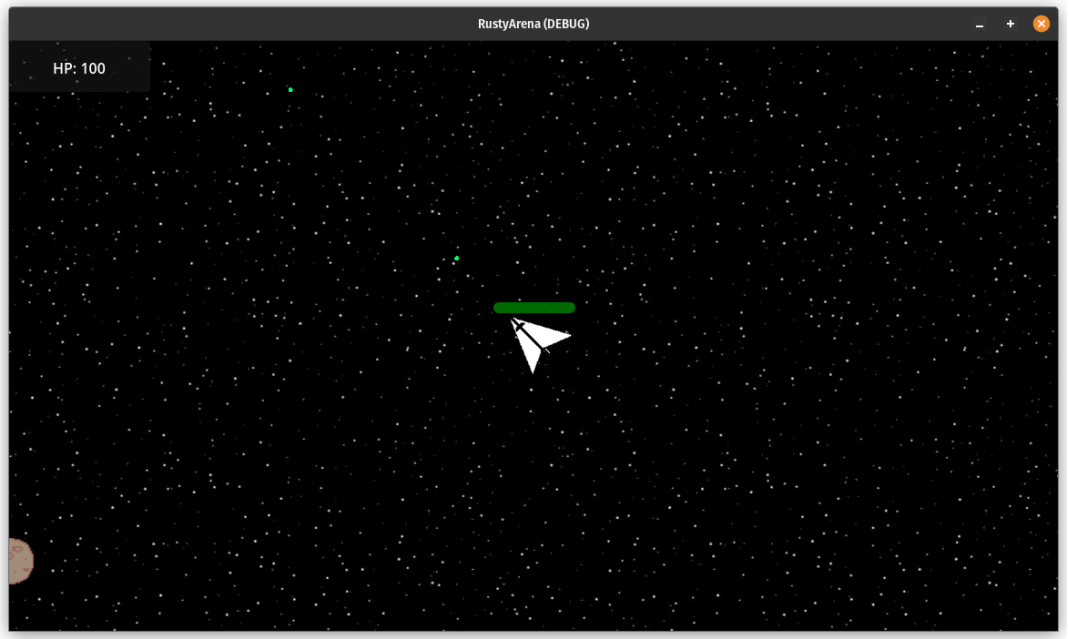
4.2 Игра

Након успешне пријаве корисник се преусмерава на главни приказ игре, где се налази мапа са више ентиета, укључујући играче, астероиде и испаљене метке. Слика 11 приказује екран након што се корисник улоговао.



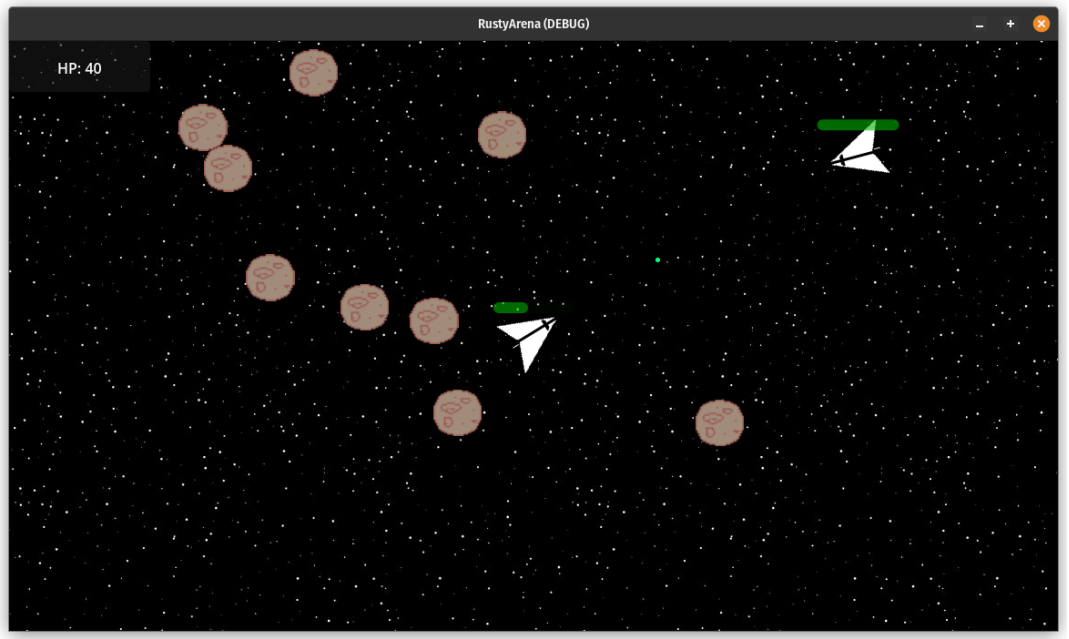
Слика 11: Приказ видео игре

Приликом инстанцирања локалног корисника (играча који започиње игру) иницијализује се камера. Камера се помера заједно са корисником и прати га, тако да се корисник који игра игру увек налази у самом центру. Приказ видео игре приказан на слици 12.



Слика 12: Приказ видео игре

Изнад сваког игача приказано је његово тренутно стање животних поена. Слика 13 приказује како изгледа када је више играча истовремено у игри.



Слика 13: Приказ када је у игри приказано више играча

Сви ентиети су дефинисани кроз *Godot* сценских систем, док је њихово понашање имплементирано у *Rust*-у. Текстуре за ентиете креирани су коришћењем бесплатаног алата отвореног кода *Krita*[6].

4.3 Проблеми и могућа побољшања

Визуелизација приказује основне елементе интерфејса и механику рада игре, чиме се потврђује функционалност система имплементираног у овом раду. Иако систем испуњава основне захтеве, постоји простор за побољшање корисничког искуства, нарочито у домену аудио-визуелних ефеката и дизајна интерфејса.

[6] https://krita.org/en/

Глава 5

Анализа

У овом поглављу извршена је анализа имплементираног решења и његовог понашања у пракси, као и поређење са примером у индрустрији. Анализа обухвата преглед стабилности система, мрежне комуникације, перформанси и преносивости, као и примене употребљених технологија у контексту развоја видео-игара.

Развијени систем је тестирањем потврђено да исправно функционише у локалном окружењу. Пројекат је развијен на оперативном систему *Pop!_OS (Linux)*, користећи *Godot engine* са *Rust* екстензијом. Примарна комуникација између клијената и сервера остварена је путем UDP протокола, што је омогућило брзу и ефикасну размену података уз минимално кашњење. Репликација стања света показала се као поуздана током целог тестирања.

Иако се у индустрији развоја видео-игара традиционално користи *Windows* као примарна платформа за развој, реализација овог пројекта на *Linux* окружењу представља значајан корак ка већој преносивости и независности од оперативног система. Рад на *Linux* окружењу показао је да се савремени алати могу успешно интегрисати и ван традиционлног *Windows* екосистема. Овакав избор представља корак ка већој флексибилности и техничкој независности у развоју видео-игара.

Поред тога, игра је *cross-plaform*, што значи да се може извршавати на различитим оперативним системима из истог изворног кода. Иста верзија игре се може једноставно компајлирати и експортовати без измене кода и за *Windows* системе, што потврђује њену преносивост и техничку флексибилност.

Тестирање је спроведено у локалној мрежи и резултати су показали стабилну комуникацију и правилну репликацију стања света. Систем се може прилагодити за рад на удаљеном серверу, а уз минималне измене може се поставити и на инфраструктуре у облаку (енг. *cloud*). Као пример овакве инфраструктуре може се навести AWS (Amazon Web Services), који користе и велике компаније попут Riot Games, где је целокупна мрежна архитектура игре *League of Legends* изграђена управо на AWS инфраструктури. Оваква архитектура омогућава лако скалирање и потенцијално повезивање већег броја клијената [22].

Перформансе су оцењене као задовољавајуће, са минималним оптерећењем процесора током игре и стабилним протоком података. Највећи технички изазови јављали су се

приликом синхронизације објеката у сценама и одржавања конзистентности података између више клијената.

Глава 6

Закључак

Један од главних циљева овог рада био је да покаже да се *Rust* може успешно користити за развој видео-игара и реализацију мрежне комуникације, са високим перформансама и безбедним руковањем меморијом. Поред тога, рад показује снагу бесплатних алат и алата отвореног кода, као што су *Godot engine* и *Krita*, који омогућавају развој комплетног система без потребе за великим финансијским улагањима куповине лиценци са софтвер.

6.1 Решење

Развијени систем омогућава стабилну интеракцију више играча у реалном времену и пружа основну инфраструктуру за даљи развој. Клијентска апликација се може експортовати и на *Linux* и на *Windows* оперативним системима, а могуће је експортовати и за остале системе као што су *iOS*, *MacOS* и *Android*. Тестирања су показла стабилност и поузданост мрежне комуникације, а употреба бесплатних алата и алата отвореног кода омогућава развој комплетног система без великих финансијских трошкова. Употреба бесплатних алата и алата отвореног кода није само техничка карактеристика, већ омогућава и већем броју инди гејм девелопера (енг. *indie game developers*) да креирају и реализују своје идеје, пружајући простор за изражавање креативности и експериментисање без потребе за финансијским улагањима. Овај рад потврђује да је могуће имплементирати функционалну мултиплејер игру са стабилном мрежном конекцијом.

6.2 Даљи развој пројекта

Техничке и нефункционалне тачке:

- Рефакторисање ситема, повећање модуларност компоненти и смањити спрегнутост како би олакшали даљи развој система
- Имплементација напреднијих техника синхронизације - имплементција техника као што су *entity interpolation* и *lag compensation* ради глатког приказа других играча и прецизнијег руковња кашњењем у мрежној комуникацији.
- Фокус рада је био на мрежној комуникацији, али не треба запоставити безбедносни апсект. Потребно је порадити на безбедности аутентификације, као и системе за спречавање варања у игри. Треба бити обазрив око доношења одлуке о шифровању комуникације јер то уводи додатно кашњење пакета.

Функционалности и корисничко искуство:

- Побољшање визуелног приказа, корисничког интерфејса и UX - додавање звука, позадинске музике, визуелних ефеката и побољшање приказа стања игре ради бољег

корисничког искуства. Побољшање корисничког искуства треба да омогући корисницима праведно и угодно искуство и спречи потенцијалне злоупотребе.

- Скалирање система за инфраструктуру у облаку - подршка за већи број играча, рад на удаљеним серверима и инфраструктурама у облаку као што је AWS, омогућавајући лако проширење система и примену у већим пројектима.
- Систем за проналажење партије (енг. *match-making system*) - пожељно је имплементирати и подржати већ број инстанци симулације света, било то више инстанци унутар једног сервера или једноставно покренути више инстанци сервера.
- Додавање нових функционалности - тренута гејмплеј игре је једноставан, потребно је проширити игру са новим функционалностима. Неки примери оваквих проширења би били додавањем различитог оружја или појачања (енг. *power-ups*) које корисник може да искористи. Поред ових аспеката могуће је убацити и козметичке аспекте као што су могућност да корисник модификује свој брод. У индустрији често је случај да се козметички аспект користи за монетизацију игре (енг. *game monetization*), односно представља види како игра профитира.

Друштвени апсект:

- Подстицање креативност и доступности алата - употреба бесплатних алата и алата отвореног кода омогућавају већем броју девелопера да експериментишу, креирају и реализују своје идеје. Подстаћи друге људе да учествују у развоју поменутих алата, као и развоју примера у којима се ти алати користе.

Списак слика

Списак листинга

Списак коришћених скраћеница

Скраћеница	Опис
API	Application Programming Interface (апликациони програмски интерфејс)
AWS	Amazon Web Services (Амазон веб сервиси)
DTO	Data Transfer Object (објекат за пренос података)
HTTP	HyperText Transfer Protocol (протокол за пренос хипертекста)
IP	Internet Protocol (интернет протокол)
JSON	JavaScript Object Notation (формат за размену података)
LAN	Local Area Network (локална мрежа)
MMORPG	Massive Multiplayer Online Role-Playing Game (Масовна мултиплејер онлајн игра улога)
OSI	Open Systems Interconnection (модел отворених система)
P2P	Peer-to-Peer (равноправна мрежна архитектура)
SQL	Structured Query Language (структурирани упитни језик)
TCP	Transmission Control Protocol (протокол за контролу преноса)
TLS	Transport Layer Security (безбедност транспортног слоја)
UML	Unified Modeling Language (језик за моделовање дијаграма)
UI	User Interface (кориснички интерфејс)
URL	Uniform Resource Locator (јединствени идентификатор и локатор ресурса)
UX	User Experience (корисничко искуство)

Списак коришћених појмова

Појам	Објашњење
Асихнрони рад	Рад који се изводи независно од главног тока извршавања, омогућавајући наставак других операција без чекања на његов завршетак.
Клијент-сервер архитектура	Модел комуникације у коме клијент шаље захтеве, а сервер их обрађује и шаље одговоре.
Cross-platform	Способност софтвера да се извршава на више различитих оперативних система из истог кода.
Game loop	Главна петља у видео игри која управља симулацијом света, обрадом улаза и приказом слике.
Предикција клијента	Техника којом клијент процењује резултате сопствених акција пре него што сервер пошаље потврду , како би се избегло кашњење у приказу.
Репликација	Процес синхронизације стања објеката између сервера и клијената у мултиплејер игри.
Сервер за аутентификацију	Део система који потврђује идентитет корисника приликом пријаве.
Синхронизација са сервером	Поступак усаглашавања стања игре клијента са оним које се налази на серверу.
Godot engine	Бесплатан, отвореног кода погон за развој видео игара који подржава 2Д и 3Д игре, овде проширен Rust библиотеком.
Rust	Системски програмски језик оријентисан на безбедност и перформансе, погодан за конкурентне и мрежне апликације.
UDP комуникација	Мрежна комуникација која не гарантује редослед или поузданост пакета, али обезбеђује високу брзину преноса.
TCP комуникација	Поуздана комуникација у којој се сваки пакет потврђује и испоручује у исправном редоследу.

Биографија

Алекса Вукомановић рођен је 22. априла 2002. године у Крушевцу, Република Србија. Завршио је природно-математички смер Гимназије “Вук Караџић” у Трстенику 2021. године. Исте године уписао је основне академске студије на Факултету техничких наука у Новом Саду, смер Софтверско инжењерство и информационе технологије. Положио је све испите предвиђене планом и програмом са просечном оценом 9.70.

Литература

[1] Future Publishing, „ACE Magazine, Issue 06 (March 1988)“, *ACE (Advanced Computer Entertainment)*, Март 1988, Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://archive.org/details/ACE_Issue_06_1988-03_Future_Publishing_GB/page/n34/mode/1up

[2] R. Adams, „A history of 'Adventure'“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://rickadams.org/adventure/

[3] „Atari PONG“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://www.computinghistory.org.uk/det/4007/Atari-PONG

[4] International Organization for Standardization и International Electrotechnical Commission, „Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model“. [На Интернету]. Доступно на https://www.ecma-international.org/wp-content/uploads/s020269e.pdf

[5] „What is the Internet Protocol?“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://www.cloudflare.com/learning/network-layer/internet-protocol/

[6] I. Grigorik, „Building Blocks of TCP“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://hpbn.co/building-blocks-of-tcp/

[7] „Discord documentation - Topic - Voice“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://discord.com/developers/docs/topics/voice-connections

[8] B. Gorman, „TCP vs UDP: Differences between the protocols“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://www.avast.com/c-tcp-vs-udp-difference

[9] „Bincode crate documentation“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://docs.rs/bincode/latest/bincode/

[10] „Serde“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://serde.rs/

[11] „Tokio crate documentation“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://docs.rs/tokio/latest/tokio/

[12] „Godot crate documentation“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://godot-rust.github.io/docs/gdext/master/godot/

[13] „SQLx crate documentation“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://docs.rs/sqlx_wasi/latest/sqlx/

[14] „Axum crate documentation“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://docs.rs/axum/latest/axum/

[15] „Reqwest crate documentation“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://docs.rs/reqwest/latest/reqwest/

[16] „Godot engine - Your free, open-source game engine.“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://godotengine.org/

[17] „References and Borrowing“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html

[18] M. Boroń, J. Brzeziński, и A. Kobusińska, „P2P matchmaking solution for online games“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://link.springer.com/article/10.1007/s12083-019-00725-3

[19] G. Fiedler, „What Every Programmer Needs To Know About Game Networking“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/

[20] G. Gambetta, „Fast-Paced Multiplayer (Part I): Client-Server Game Architecture“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://www.gabrielgambetta.com/client-server-game-architecture.html

[21] G. Gambetta, „Fast-Paced Multiplayer (Part II): Client-Side Prediction and Server Reconciliation“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://www.gabrielgambetta.com/client-side-prediction-server-reconciliation.html

[22] A. Events, „AWS re:Invent 2024 - Effortless game launches: How League of Legends runs at scale on AWS (GAM307)“. Приступљено: 08. Новембар 2025. [На Интернету]. Доступно на https://www.youtube.com/watch?v=iNYmyuFVMCo