Modeliranje objektno-orijentirane baze podataka za potrebe videoigre žanra roguelike

Brleković, Luka

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Luka Brleković

MODELIRANJE OBJEKTNO-ORIJENTIRANE BAZE PODATAKA ZA POTREBE VIDEOIGRE ŽANRA ROGUELIKE

ZAVRŠNI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Luka Brleković

Matični broj: 0016148697

Studij: Informacijski i poslovni sustavi

MODELIRANJE OBJEKTNO-ORIJENTIRANE BAZE PODATAKA ZA POTREBE VIDEOIGRE ŽANRA ROGUELIKE

ZAVRŠNI RAD

Mentor:

doc. dr. sc. Bogdan Okreša Đurić

Varaždin, rujan 2024.

Luka Brleković

Izjava o izvornosti

Izjavljujem da je ovaj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI Radovi

Sažetak

Roguelike igre, definirane kao podžanr igre uloga (engl. role playing game, RPG) s proceduralno generiranim sadržajem, perma-smrću likova i poteznom mehanikom, privlače igrače zbog svoje nepredvidljivosti i dubokih strategijskih elemenata. Ovaj rad istražuje primjenu objektno-orijentiranih baza podataka (OOBP) u razvoju roguelike igara, s posebnim naglaskom na prednosti i izazove koje ove baze podataka donose. Analizirana je fleksibilnost modeliranja i brzina manipulacije podatcima koje OOBP-i omogućuju, te je demonstrirana primjena kroz konkretne primjere, poput upravljanja entitetima unutar igre. Poseban fokus stavljen je na ugrađivanje dragulja u mač, gdje je jasno vidljiva prednost OOBP-a u odnosu na relacijske baze podataka. Metodološki, rad se oslanja na eksperimentalni pristup, koristeći razne programske alate i platforme za razvoj videoigara, uključujući Unity, Flask, i Zope Object Database (ZODB). Također, istraženi su načini kako koristiti OOBP na udaljenom računalu, što dodatno unapređuje upravljanje podatcima i resursima. Zaključci rada ukazuju na to da OOBP-i predstavljaju ključno rješenje za optimizaciju razvoja roguelike igara, nudeći poboljšane performanse, prirodno mapiranje objekata i veću fleksibilnost. Unatoč izazovima implementacije, prednosti koje pružaju nadmašuju te prepreke, otvarajući nove mogućnosti za inovacije u gaming industriji.

Ključne riječi: roguelike igre; objektno-orijentirane baze podataka; razvoj videoigara; fleksibilnost modeliranja; performanse; inovacije u gaming industriji

Sadržaj

1. Uvod

Roguelike igre, često definirane kao podžanr igre uloga (engl. role playing game, RPG) koji karakterizira proceduralno generirani sadržaj, perma-smrt likova i potezna mehanika, privlače pažnju igrača već desetljećima. Njihova jedinstvena kombinacija izazova, nepredvidljivosti i dubokih strategijskih elemenata čini ih izazovnim za razvoj i intrigantnim za istraživanje. U svijetu sve sofisticiranijih tehnologija, objektno-orijentirane baze podataka (OOBP) postaju ključan alat za razvoj ovih igara, omogućujući razvojnim timovima da učinkovito upravljaju složenim entitetima i njihovim odnosima.

Ovaj rad istražuje primjenu OOBP-a u kontekstu razvoja roguelike igara. Fokus je na prednostima i izazovima koje OOBP donose u ovom specifičnom žanru igara, analizirajući kako njihova fleksibilnost modeliranja i brza manipulacija podatcima mogu unaprijediti razvojni proces. Također, su istraženi specifični primjeri modeliranja entiteta u roguelike igrama koristeći OOBP, pružajući konkretne primjere implementacije i njihovog utjecaja na performanse i funkcionalnosti igre.

Kroz ovaj rad, cilj je istaknuti potencijal OOBP-a za optimizaciju razvoja roguelike igara, kao i razumijevanje specifičnih izazova s kojima se razvojni timovi susreću pri integraciji ovih tehnologija. Konačno, istaknute su važnosti odabira odgovarajuće tehnološke infrastrukture u kontekstu razvoja modernih igara, nudeći uvid u buduće smjerove istraživanja i primjene u ovom dinamičnom području industrije razvoja videoigara.

Odlučio sam istražiti ovu temu iz nekoliko ključnih razloga. Prvenstveno, želja mi je sudjelovati u razvoju videoigara, a razumijevanje OOBP-a ima ključnu ulogu u optimizaciji ovog procesa. Umjesto tradicionalnog pristupa instaliranja OOBP-a izravno na lokalno računalo korisnika, istražujem mogućnost kako se te baze podataka mogu efikasno koristiti na udaljenom računalu, što otvara vrata za bolje upravljanje podatcima i resursima u razvoju igara. Ova inovativna perspektiva primjene OOBP-a može pridonijeti ne samo efikasnosti razvoja, već i dubljem razumijevanju njihovog utjecaja na performanse i funkcionalnosti samih igara.

2. Metode i tehnike rada

U ovom poglavlju su opisane metode i tehnike koje su korištene pri razradi teme, istraživačke aktivnosti provedene tijekom rada te programski alati ili aplikacije koje su korištene. Primijenjen je eksperimentalni pristup iz razloga što prilikom istraživanja nije pronađena niti jedna videoigra koja koristi objektno-orijentiranu bazu podataka koja je poslužena na udaljenom računalu.

2.1. Istraživačke aktivnosti

Istraživanje je uglavnom obuhvaćalo korištenje online platformi poput specijaliziranih foruma za razvoj videoigara (Unity Forum, Reddit - r/gamedev, Stack Overflow, GitHub), kao i interaktivnih alata kao što su Copilot i ChatGPT. Također je bilo potrebno istražiti i testirati razne objektno-orijentirane baze podataka za što su pomogli forumi, dokumentacije i web-stranice razvojnih timova tih baza podataka. Ovi resursi su omogućili prikupljanje relevantnih informacija, savjeta i primjera koji su doprinijeli razumijevanju najboljih praksi u razvoju OOBP.

2.2. Programski alati

U razvoju ove videoigre ključno je bilo koristiti niz specifičnih alata i programskih jezika kako bi se omogućila integracija različitih komponenti, poput baze podataka, poslužitelja i korisničkog sučelja. Kombinacija Python skripti za pozadinsku (engl. backend) logiku, Unity za razvoj igre i MySQL za upravljanje korisničkim podacima osigurala je stabilnu i funkcionalnu igru, dok su alati poput Flask-a i NodeJS-a omogućili učinkovitu komunikaciju između različitih sustava.

1. **Programski jezici:**
    - SQL: Korišten za modeliranje i upravljanje podatcima u MySQL bazi podataka.
    - Python: Korišten za implementaciju pozadinske (engl. backend) logike, posebno za komunikaciju sa Zope Object Database (ZODB).
    - C#: Korišten za razvoj komponenti aplikacije unutar Unity platforme.
    - JavaScript: Primijenjen za razvoj NodeJS poslužitelja koji omogućuje komunikaciju sa MySQL bazom podataka.
2. **Korišteni programi:**
    - ChatGPT i Copilot: Korišteni za generiranje kôda, verifikaciju i dobivanje rješenja za specifične razvojne izazove.
    - Unity: Primarno okruženje za razvoj videoigre.

- ZODB (Zope Object Database): Objektno-orijentirana baza podataka koja sprema python objekte, no unutar ovog rada je pohranjivala i objekte za Unity koji koristi C# jezik.
- ZEO (Zope Enterprise Object poslužitelj): Server korišten radi mogućnosti korištenja ZODB na računalima izvan mreže poslužitelja to jest da igrači izvan mreže poslužitelja mogu učitati svoje podatke unutar igre.
- MySQL: Korišten za razvoj sustava za prijavu i registraciju korisnika/igrača te za tablicu s najboljim rezultatima unutar videoigre.
- NodeJS: Služi kao posrednik između korisnika/igrača i MySQL-a tako da se korisnici/igrači mogu ulogirati u igru iako su izvan mreže poslužitelja te na tablicu najboljih rezultata objaviti svoje rezultate igre.
- Visual Studio 2022 i Visual Studio Code: Korišteni za razvoj Unity skripti, Python skripti za ZODB poslužitelj te MySQL poslužitelja u NodeJS-u.
- Flask: Korišten za implementaciju poslužitelja koji omogućuje interakciju između videoigre i ZODB-a.
- Discord: Korišten za distribuciju videoigre i aktualne IP adrese poslužitelja prilikom testiranja igre pomoću više igrača.

3. **Specifični alati unutar Unity-a:**
    - TextMeshPro (TMPro): Korišten za dizajn korisničkog sučelja i interaktivnost unutar kreirane videoigre.
    - MiniJSON: JSON parser za efektivnu komunikaciju s MySQL i ZODB bazama podataka.
    - Scriptable Object: Objekti kreirani unutar Unity igre koji prenose svoje podatke iz jedne u drugu scenu.

3. Razrada teme

U sljedećim poglavljima detaljno je prikazan cijelokupan proces razvoja videoigre i dvije baze podataka potrebne za nju. Prvo je predstavljen osnovni koncept projekta kako bi se pojasnio cilj ovog rada. Nakon toga, fokus je na razvoju baza podataka, uključujući dizajn i pohranu podataka. Slijedi razrada razvoja same videoigre, što uključuje pisanje opsežnog kôda. Nakon toga slijedi povezivanje videoigre s bazama podataka, pri čemu su istaknuti izazovi i problemi vezani uz korištenje ZODB-a. Na kraju, opisano je testiranje videoigre i baza podataka kako bi se osigurala njihova funkcionalnost i stabilnost.

3.1. Koncipiranje videoigre i baza podataka

Ideja ove videoigre je pružiti igračima okruženje u kojem mogu eksperimentirati na različite načine unutar roguelike sustava. Igra omogućuje odabir različitih likova s posebnim karakteristikama. Tijekom igre igrači mogu nadograditi bilo koje karakteristike svojega lika pri prelasku na višu razinu (engl. level up). Kako bi se iskoristila OOBP, kreiran je sustav za stvaranje snažnih predmeta koji poboljšavaju lik igrača. Ti predmeti uključuju dragulje kao zasebne objekte, koji se potom mogu ugraditi u mač, također zaseban objekt, čime se koristi objektna orijentacija baze podataka.

Igrač može zaraditi navedene predmete i nadograđivati ih kako bi imao što bolji početak svake runde. Za videoigru su kreirane dvije baze podataka. Prva je relacijska baza podataka MySQL, koja služi za login i registracijski sustav te za tablicu najboljih rezultata igrača. Druga baza podataka je ZODB, koja bilježi inventar igrača, uključujući razne objekte poput mačeva i dragulja, te količinu valute koju igrač posjeduje. Valuta se koristi za nadogradnju dragulja, a dragulji se također mogu nadograditi prikupljanjem tri dragulja iste vrste.

U igri se igrač suočava s jednim protivnikom nakon kojeg slijedi jači protivnik, i tako beskonačno dok igrač ne doživi poraz. Ako igrač obori rekord na tablici - nagrađen je mačem, a ako skupi barem polovicu bodova prvoplasiranog - nagrađen je draguljem. Cilj igre je potaknuti igrače na pronalaženje što jačeg lika, kako bi se otkrile greške u balansu igre te prilagodila igra tako da je sve teže postići visok broj bodova.

Kako bi igra bila zanimljivija i zabavnija, implementiran je dodatni sustav tempiranja. Igrač, osim što može kliknuti "Attack" (napad) tijekom bitke, sada mora tempirati svoj napad kako bi postigao veći učinak. Nakon što igrač klikne "Attack", vertikalna linija počinje putovati s lijeva na desno duž dugačke horizontalne linije. Igračev zadatak je tempirati svoj sljedeći klik na "Attack" kako bi zaustavio vertikalnu liniju što bliže sredini horizontalne linije. Što je igrač precizniji, to će više štete nanijeti protivniku. Ovaj sustav postaje značajniji s većom težinom igre. Na "Easy" (laganoj) razini, sustav tempiranja je zanemaren i igrač dobiva manje bodova. Međutim, na "Hard" razini, preciznost u tempiranju postaje ključna. Ukoliko igrač potpuno promaši prilikom tempiranja, neće nanijeti nikakvu štetu protivniku, ali će za ovaj rizik biti nagrađen većim brojem bodova. Ovaj sustav dodaje dodatni element vještine i strategije, čineći igru izazovnijom i zanimljivijom.

Unutar videoigre implementiran je sustav za direktno povezivanje na udaljeni poslužitelj (engl. direct connect), što omogućuje igračima da se povežu na različite udaljene poslužitelje. Ovaj sustav osmišljen je kako bi omogućilo testiranje igre s velikim brojem igrača koji nisu povezani na istu mrežu kao poslužitelj. Na taj način, igra može biti testirana u uvjetima sličnima stvarnom korištenju, osiguravajući da sve funkcionalnosti rade ispravno i da je iskustvo igranja konzistentno za sve korisnike, bez obzira na njihovu fizičku lokaciju.

S obzirom na to da je ovo eksperimentalni rad, nije posvećeno vrijeme razvoju sigurnosnih mjera za korištene sustave. Svi podatci se prenose preko HTTP protokola te igrači mogu poslati GET i POST requestove na ZODB i MySQL poslužitelje ukoliko poznaju njihove API-je. Ova implementacija služi isključivo za demonstraciju i testiranje, stoga sigurnosni aspekti nisu detaljno obrađeni.

3.2. Objektno-orijentirane baze podataka

3.2.1. Pregled tipova baza podataka i sustava za upravljanje bazama podataka

Razumijevanje različitih tipova baza podataka i sustava za upravljanje bazama podataka ključno je za odabir najprikladnijeg rješenja za specifične potrebe razvoja softvera. Postoji nekoliko tipova baza podataka, a svaki od njih ima svoje specifične prednosti i nedostatke, ovisno o prirodi podataka i načinu njihova korištenja.

- **Relacijske baze podataka (engl. relational database management system, RDBMS)**: Ove baze podataka koriste tablice za pohranu podataka i odnose između njih definirane pomoću ključnih atributa. Primjeri uključuju MySQL, PostgreSQL i Oracle. Relacijske baze podataka su vrlo pogodne za aplikacije koje zahtijevaju složene upite i transakcije te imaju jasno definiranu shemu.
- **Objektno-orijentirane baze podataka (OOBP)**: OOBP pohranjuju podatke u obliku objekata, sličnih onima korištenim u objektno-orijentiranim programskim jezicima. Ovo omogućuje prirodnije mapiranje složenih objekata i njihovih odnosa bez potrebe za dodatnim slojem mapiranja između objekata i tablica. Primjeri uključuju ZODB i ObjectDB.
- **NoSQL baze podataka**: Ovaj tip uključuje razne vrste baza podataka koje nisu bazirane na relacijskom modelu, kao što su dokumentne baze (npr. MongoDB), grafovske baze (npr. Neo4j) i baze podataka ključ-vrijednost (npr. Redis). NoSQL baze podataka često nude veću fleksibilnost i skalabilnost za specifične primjene kao što su analiza velikih podataka i aplikacije u stvarnom vremenu (engl. real-time).
- **Hibridni sustavi**: Neki sustavi za upravljanje bazama podataka nude mogućnosti koje kombiniraju značajke različitih tipova baza podataka. Na primjer, SQL/NoSQL

hibridi omogućuju pohranu strukturiranih i nestrukturiranih podataka u istom sustavu.

3.2.2. Teorija objektno-orijentiranih baza podataka

Objektno-orijentirane baze podataka (OOBP) razvijene su kako bi se prevladali nedostaci tradicionalnih relacijskih baza podataka u kontekstu složenih aplikacija koje upravljaju velikim brojem povezanih objekata. U OOBP-u, podaci su predstavljeni kao objekti, s atributima i metodama koji omogućuju prirodnije modeliranje stvarnog svijeta. Ovo omogućuje:

- **Prirodno mapiranje objekata**: OOBP omogućuju izravno mapiranje složenih objekata i njihovih odnosa, što smanjuje potrebu za dodatnim slojem apstrakcije između objekata i tablica. Ovo je posebno korisno za aplikacije u kojima su podaci složeni i usko povezani, kao što su videoigre.
- **Poboljšane performanse**: Sposobnost OOBP-a da optimizira pristup velikim količinama složenih objekata može značajno poboljšati performanse aplikacija koje zahtijevaju brzu manipulaciju podacima.
- **Fleksibilnost i proširivost**: OOBP omogućuju lakšu prilagodbu i proširivanje modela podataka, što je korisno u dinamičnim razvojnim okruženjima gdje se zahtjevi i dizajn često mijenjaju.

3.2.3. Primjena objektno-orijentiranih baza podataka u videoigrama

U kontekstu razvoja videoigara, OOBP nudi značajne prednosti. Videoigre često uključuju složene objekte kao što su likovi, predmeti i okruženja, koji imaju brojne međusobne odnose. OOBP omogućuju izravno pohranjivanje i upravljanje tim objektima, što može pojednostaviti razvoj i poboljšati performanse igre.

- **Izravno modeliranje objekata igre**: OOBP omogućuju izravno modeliranje i pohranu objekata igre kao što su likovi, predmeti i okruženja, čime se eliminira potreba za kompleksnim mapiranjem između objekata i tablica. Ovo može značajno ubrzati razvoj i optimizirati performanse igre.
- **Upravljanje složenim odnosima**: Igre često uključuju složene odnose između različitih objekata, kao što su interakcije između likova i predmeta. OOBP pružaju prirodan način za upravljanje tim odnosima, što može poboljšati iskustvo igre i olakšati razvoj.
- **Fleksibilnost u razvoju**: S obzirom na to da se zahtjevi u razvoju videoigara često mijenjaju, OOBP omogućuju lakšu prilagodbu i proširivanje modela podataka bez potrebe za složenim migracijama baza podataka.

3.3. Žanr videoigara: Roguelike

3.3.1. Opis žanra roguelike

Roguelike je specifičan žanr videoigara koji se karakterizira proceduralno generiranim sadržajem, trajnim gubitkom napretka (engl. permadeath) i često složenim, taktičkim igranjem. Glavne karakteristike roguelike igara uključuju:

- **Proceduralna generacija**: Roguelike igre koriste proceduralnu generaciju za stvaranje razina, predmeta i izazova. Ovaj pristup omogućuje visoku razinu ponovljivosti i raznolikosti u igranju, jer svaki put kada igrač započne novu igru, sadržaj može biti drugačiji.
- **Permadeath**: U roguelike igrama, smrt igrača obično rezultira gubitkom svih napredaka i početkom nove igre od nule. Ovaj element povećava težinu igre i potiče strateško razmišljanje jer igrači moraju pažljivo planirati svoje poteze.
- **Taktičko igranje**: Roguelike igre često zahtijevaju duboko strateško razmišljanje i planiranje zbog svojih kompleksnih sustava i izazovnih mehanika. Igrači moraju koristiti svoje resurse mudro i prilagođavati se nepredviđenim situacijama.

3.3.2. Primjena objektno-orijentiranih baza podataka u roguelike igrama

Primjena objektno-orijentiranih baza podataka (OOBP) u razvoju roguelike igara može biti osobito korisna zbog složenosti i dinamičnosti sadržaja. OOBP omogućuju:

- **Upravljanje proceduralno generiranim sadržajem**: OOBP omogućuju učinkovito upravljanje kompleksnim objektima i njihovim međusobnim odnosima, što može olakšati rad s proceduralno generiranim sadržajem u roguelike igrama. Ovo uključuje stvaranje i pohranu razina, predmeta i izazova koje igra koristi.
- **Reprezentacija entiteta i odnosa**: OOBP omogućuju prirodnu reprezentaciju likova, predmeta i okruženja, čime poboljšavaju organizaciju i manipulaciju tih elemenata u igri. Ključna prednost OODB-a je mogućnost modeliranja složenih odnosa između objekata tako što se objekti mogu pohranjivati unutar drugih objekata. Na primjer, u OODB-u, jedan objekt može sadržavati druge objekte kao svoje atribute, što omogućuje precizno modeliranje hijerarhija i odnosa. Objekti također mogu međusobno komunicirati putem svojih metoda (funkcija), što doprinosi dinamičnom i fleksibilnom upravljanju stanjem i interakcijama u igri.
- **Adaptivni modeli podataka**: Fleksibilnost OOBP-a u prilagodbi modela podataka može pomoći u bržem razvoju i iteraciji kompleksnih sustava koji su ključni za roguelike igre. Ovo omogućuje brze promjene i nadogradnje modela podataka kako bi se zadovoljili zahtjevi dinamičnog razvoja igre.

Razumijevanje teoretskog okvira objektno-orijentiranih baza podataka i njihovih primjena, posebno u kontekstu videoigara, omogućava bolje usklađivanje tehnologija s potrebama specifičnih žanrova poput roguelike. OOBP nude značajne prednosti u upravljanju složenim podacima i odnose se izravno na izazove s kojima se susreću razvojni timovi u ovom dinamičnom žanru

3.4. Modeliranje baze podataka

Modeliranje baze podataka ključno je za dizajn učinkovitih i održivih sustava za upravljanje podacima. Ovaj proces omogućuje strukturirano pristupanje složenim informacijama i njihovim međusobnim odnosima, što je ključno za uspješno upravljanje i korištenje podataka u različitim aplikacijama.

3.4.1. Proces modeliranja baze podataka

Proces modeliranja baze podataka obuhvaća nekoliko ključnih faza, uključujući konceptualno, logičko i fizičko modeliranje. Svaka faza ima specifične ciljeve i koristi različite tehnike za osiguranje da baza podataka ispravno i učinkovito zadovoljava zahtjeve aplikacije.

3.4.1.1. Konceptualno modeliranje baze podataka

Konceptualno modeliranje je prva faza u procesu modeliranja baze podataka i fokusira se na definiranje osnovne strukture i odnosa između podataka bez ulaska u tehničke detalje implementacije. Glavni ciljevi ove faze su:

- **Identifikacija entiteta**: Razlikovanje glavnih objekata (entiteta) koji će biti pohranjeni u bazi podataka. Na primjer, u bazi podataka za videoigru, entiteti mogu uključivati likove, predmete i razine.
- **Definiranje atributa**: Određivanje važnih informacija koje svaki entitet treba sadržavati. Na primjer, likovi mogu imati atribute kao što su ime, zdravlje i iskustvo.
- **Utvrđivanje odnosa**: Definiranje kako se entiteti međusobno povezuju. Na primjer, lik može posjedovati predmete, a predmeti mogu biti smješteni u razinama igre.

Za vizualizaciju konceptualnog modela često se koristi dijagram entiteta i odnosa (ER dijagram).

3.4.1.2. Logičko modeliranje baze podataka

Logičko modeliranje slijedi nakon konceptualnog modeliranja i usmjerava se na prevođenje konceptualnog modela u strukturirani oblik koji se može lakše implementirati u stvarnom sustavu za upravljanje bazom podataka. Ova faza uključuje:

- **Normalizacija podataka**: Proces organiziranja podataka kako bi se minimizirali redundantni podaci i osigurao njihov integritet. Ovo uključuje definiranje normalnih formi i primjenu pravila normalizacije.
- **Definiranje sheme baze podataka**: Određivanje tablica, njihovih atributa i odnosa između njih. Na primjer, u bazi podataka za videoigru, tablice mogu uključivati 'Likovi', 'Predmeti' i 'Razine', s definiranim ključevima i stranim ključevima.
- **Stvaranje odnosa među tablicama**: Uspostavljanje odnosa između tablica pomoću stranih ključeva koji osiguravaju referencijski integritet.

Za vizualizaciju logičkog modela koristi se dijagram tablica (kao što su UML dijagram klasa ili relacijski dijagram).

3.4.1.3. Fizičko modeliranje baze podataka

Fizičko modeliranje se bavi implementacijom logičkog modela u konkretnom sustavu za upravljanje bazom podataka. Ova faza uključuje:

- **Odabir tehnologije**: Odabir specifičnog sustava za upravljanje bazom podataka (engl. database managment system, DBMS) kao što su MySQL, PostgreSQL ili MongoDB.
- **Definiranje fizičkih struktura**: Određivanje konkretnih struktura u DBMS-u, uključujući tablice, indekse, ključeve, te postavke za optimizaciju performansi i sigurnost.
- **Implementacija i testiranje**: Stvaranje baza podataka prema fizičkom modelu, učitavanje podataka i testiranje funkcionalnosti baze podataka.

Za fizičko modeliranje mogu se koristiti SQL skripte za kreiranje tablica i definiranje odnosa, kao i alati za upravljanje bazama podataka za testiranje i optimizaciju.

3.4.2. Odabrana metodologija za modeliranje baze podataka

Za modeliranje baze podataka u ovom radu, korišten je grafički jezik UML (Unified Modeling Language), koji omogućava vizualizaciju, specifikaciju i dokumentaciju strukture i odnosa podataka.

Grafički jezik *UML* uključuje korištenje dijagrama klasa za konceptualno i logičko modeliranje, dok fizičko modeliranje uključuje kreiranje Python skripti i korištenje DBMS-a za implementaciju baza podataka.

3.4.2.1. Proces modeliranja prema odabranoj metodologiji

- **Konceptualno modeliranje**: Kroz UML dijagram klasa (prikazan na slici 1) identificirni su ključni entiteti, njihovi atributi i odnosi. U kontekstu modela za ZODB,

entiteti poput Igrač, Mač i Dragulj predstavljeni su kao zasebni objekti. Ovaj model ističe jedinstvene odnose specifične za ZODB: Mač može sadržavati do tri Dragulja, dok Igrač može posjedovati više Mačeva i Dragulja. Ovi odnosi omogućuju složenu i dinamičnu interakciju između objekata, koja je ključna za implementaciju u ZODB-u.

- **Logičko modeliranje**: Za logičko modeliranje korišten je UML dijagram klasa te su korišteni i dijagrami sekvenci (kao što je prikazan na slici 7) kako bi se prikazala interakcija između videoigre i ZODB-a, kao i između SQL baze podataka i ZODB-a. Ovi dijagrami pomažu u organiziranju i razumijevanju kako različiti sustavi komuniciraju i kako podaci teku kroz aplikaciju.
- **Fizičko modeliranje**: Fizičko modeliranje obuhvaća implementaciju modela u odabranom DBMS-u. Koristeći MySQL za SQL bazu podataka i Python za ZODB, kreirane su tablice, definirani su indeksi, i postavljeni su odnosi između tablica. Također su provedena testiranja kako bi se osigurala funkcionalnost baze podataka i provjerila ispravnost svih operacija i upita.

Na ovaj način, modeliranje baze podataka provedeno je kroz sve ključne faze, koristeći odabranu metodologiju kako bi se osiguralo da baza podataka zadovoljava sve tehničke i funkcionalne zahtjeve projekta.

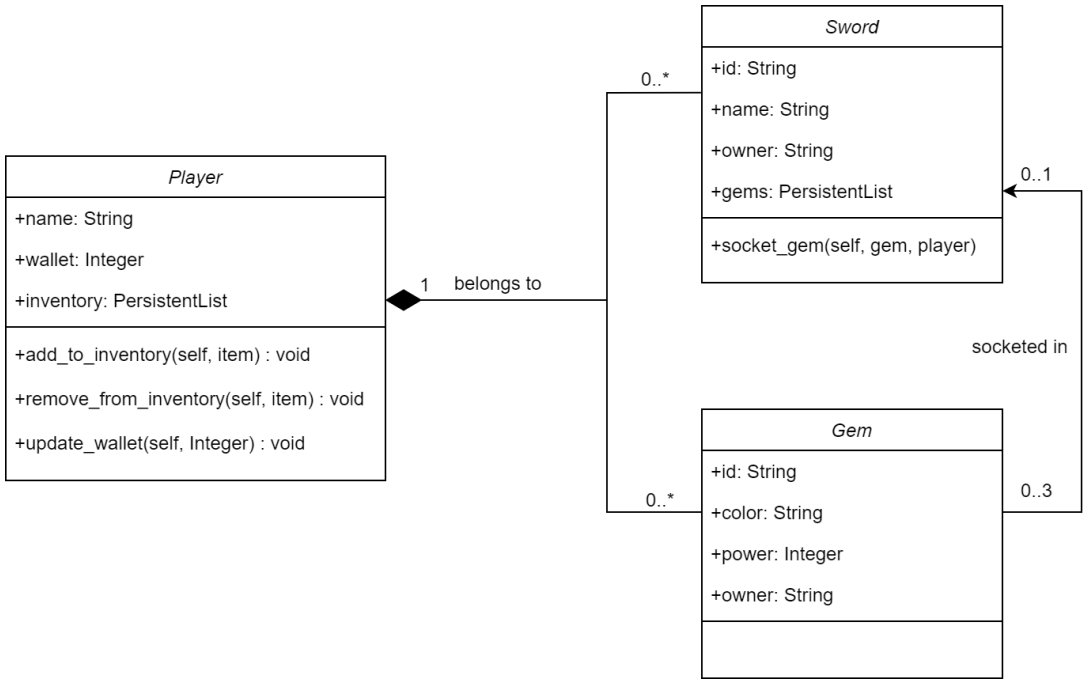
3.5. Razvoj baza podataka

Za ovu igru su razvijene dvije baze podataka. Prva baza podataka je ZODB, koja je objektno-orijentirana baza podataka. ZODB se koristi za spremanje objekata kao što su inventar igrača, uključujući predmete poput mačeva i dragulja, te novčanu vrijednost koju igrač posjeduje. Druga baza podataka je relacijska baza podataka MySQL. MySQL baza služi za login sustav, registraciju korisnika/igrača te bilježenje najboljih rezultata koje igrač postigne tijekom igre.

3.5.1. Zope Object Database

Ova baza podataka je glavna tema ovog rada te su unutar njega iskorištene njene specifične karakteristike. Primarno vrijedi istaknuti da ova baza podataka, osim objekata, sadrži i njihove funkcije. Prethodno je navedeno da ova baza podataka sadrži inventar igrača, koji je spremljen kao objekt. Taj objekt uključuje funkciju dodavanja (append) drugih objekata u sebe, uklanjanja objekata (remove) iz sebe te ažuriranja količine novčane vrijednosti koju igrač posjeduje.

Osim inventara, posebnu funkciju ima i mač koji se nalazi unutar inventara igrača. Mač ima samo jednu funkciju koja ugrađuje dragulj iz inventara igrača u mač. U nastavku je moguće pregledati UML dijagram klasa koji je služio za dizajn ove baze podataka te je moguće pročitati kôd Python skripte za modeliranje tih objekata baze podataka, čiji je naziv `my_model.py`.



Slika 1: UML dijagram klasa ZODB

```python
from persistent import Persistent
from persistent.list import PersistentList
import uuid
```

Isječak koda 1: Isječak kôda Python skripte `my_model.py`: potrebne biblioteke

Isječak kôda 1 uvozi potrebne biblioteke koje su ključne za rad s trajnim objektima i listama te za generiranje jedinstvenih identifikatora (UUID) u Pythonu.

```python
class Player(Persistent):
    def __init__(self, name):
        self.name = name
        self.wallet = 0  # Initialize wallet with 0 money
        self.inventory = PersistentList()  # The player's inventory of gems and
        ↪  swords

    def add_to_inventory(self, item):
        item.owner = self.name  # Set the owner of the item
        self.inventory.append(item)

    def remove_from_inventory(self, item):
        if item in self.inventory:
            self.inventory.remove(item)

    def update_wallet(self, amount):
        self.wallet += amount
```

Isječak koda 2: Isječak kôda Python skripte `my_model.py`: Player klasa

Isječak kôda 2 definira klasu `Player` koja nasljeđuje klasu `Persistent`. Svaki igrač ima svoje ime, novčanik koji se inicijalizira na 0, te inventar koji je `PersistentList`. Metode `add_to_inventory`, `remove_from_inventory` i `update_wallet` omogućuju upravljanje igračevim inventarom i novčanikom.

```
class Gem(Persistent):
    def __init__(self, color, power, owner):
        self.id = str(uuid.uuid4())  # Assign a unique UUID
        self.color = color
        self.power = power
        self.owner = owner

class Sword(Persistent):
    def __init__(self, name, owner):
        self.id = str(uuid.uuid4())  # Assign a unique UUID
        self.name = name
        self.owner = owner
        self.gems = PersistentList()

    def socket_gem(self, gem, player, x, y):
        if gem.owner == self.owner and gem in player.inventory:
            self.gems.append({'gem': gem, 'x': x, 'y': y})
            player.remove_from_inventory(gem)
```

Isječak koda 3: Isječak kôda Python skripte `my_model.py`: Gem klasa, Sword klasa i metoda `socket_gem`

Isječak kôda 3 definira dvije klase i jednu metodu. Klasa `Gem` predstavlja dragulj koji ima svoja svojstva boje, snage i vlasnika te je svaki dragulj jedinstveno identificiran pomoću UUID-a. Klasa `Sword` predstavlja mač koji također ima svoje ime, vlasnika i listu gemova. Metoda `socket_gem` omogućuje umetanje dragulja u mač ako dragulj pripada istom vlasniku kao i mač te je u igračevom inventaru.

Kako bi navedene funkcije ispravno radile unutar videoigre razvijene u Unity-u, koja je napisana u C#, potreban je poslužitelj s nizom specifičnih funkcija koje omogućuju interakciju između igre i ZODB-a. U nastavku je detaljan opis svih potrebnih i razvijenih funkcija. Sve funkcije koje au prikazane mogu se pronaći u pyhton skripti za poslužitelj čiji je naziv `server.py`.

Za pokretanje samog poslužitelja i aktiviranje njegovih funkcija ključno je pravilno konfigurirati pokretanje poslužitelja te definirati način na koji će on obrađivati različite GET i POST zahtjeve. U tu svrhu su u skriptu uključeni sljedeće biblioteke i modeli objekata.

```python
from flask import Flask, request, jsonify
from my_model import GameObject, Gem, Sword, Player
import ZODB, ZODB.FileStorage
import transaction
from ZEO.ClientStorage import ClientStorage
from ZODB import DB
import threading
import time
```

Isječak koda 4: Isječak kôda Python skripte `server.py` koji prikazuje korištene biblioteke

Biblioteke koje su uključene u isječak kôda `server.py` imaju ključnu ulogu u implementaciji poslužitelja za interakciju između Unity videoigre i Zope Object Database (ZODB).

1. **Flask**: Flask je mikro razvojni okvir (engl. framework) za Python koji omogućuje brzo i jednostavno stvaranje web-aplikacija. U ovom slučaju, Flask se koristi za kreiranje HTTP poslužitelja koji omogućuje komunikaciju između videoigre i ZODB-a. `Flask`, `request` i `jsonify` su uključeni radi obrade HTTP zahtjeva i odgovora, omogućujući videoigri da šalje podatke i prima odgovore preko HTTP-a.

2. **my_model**: `my_model` je lokalni Python modul koji sadrži definicije objekata koji se koriste unutar aplikacije. To uključuje `GameObject`, `Gem`, `Sword` i `Player`, koji su klase koje predstavljaju različite objekte u igri te definiraju kako se ti objekti povezuju s podatcima u ZODB-u.

3. **ZODB i ZEO**: Zope Object Database je objektno-orijentirana baza podataka za Python aplikacije. `ZODB` i `ZODB.FileStorage` se koriste za rad s lokalnim ZODB datotekama i njihovim spremanjem. `ZEO.ClientStorage` i `DB` se koriste za povezivanje s udaljenim ZODB poslužiteljem putem ZEO (Zope Enterprise Objects).

4. **transaction**: `transaction` je biblioteka koja omogućuje upravljanje transakcijama u ZODB-u, osiguravajući konzistentnost i atomičnost operacija nad podatcima.

5. **threading i time**: `threading` i `time` su Python standardne biblioteke koje se koriste za upravljanje višedretvenošću (engl. threading) i vremenskim operacijama. U ovom slučaju, koriste se za upravljanje višekorisničkim pristupima i osiguravanje da poslužitelj reagira na promjene u ZODB-u u realnom vremenu.

Ove biblioteke su odabrane i uključene u skriptu kako bi se omogućila komunikacija, upravljanje podatcima i osiguravanje stabilnosti i performansi poslužitelja koji podržava interakciju između videoigre razvijene u Unity-u i ZODB baze podataka.

```python
app = Flask(__name__)
```

Isječak koda 5: Inicijalizacija Flask aplikacije u skripti `server.py`

Inicijalizacija Flask aplikacije ‘app‘ koja predstavlja webaplikaciju temeljenu na Flask frameworku za Python.

```
# Initialize ZODB connection
def get_db_connection():
    try:
        storage = ClientStorage(('localhost', 8100),
                                wait_timeout=30,
                                min_disconnect_poll=10)  # Connect to ZEO server
                                ↪  with timeout and connection pool size
        db = DB(storage)
        connection = db.open()
        root = connection.root()
        print("DB connection initialized successfully")  # Debug print
        return db, connection, root
    except Exception as e:
        print(f"Error initializing DB connection: {e}")  # Debug print
        return None, None, None
```

Isječak koda 6: Inicijalizacija ZODB veze u skripti `server.py`

Funkcija ‘get_db_connection()‘ inicijalizira vezu sa ZEO poslužiteljem koristeći ZODB za pohranu podataka. Koristi se za uspostavu početne veze s objektno-orijentiranom bazom podataka.

```
# Maintain a connection pool to handle reconnections
class DBConnectionPool:
    def __init__(self):
        self.db, self.connection, self.root = get_db_connection()
        self.lock = threading.Lock()

    def get_connection(self):
        with self.lock:
            if self.connection is None or self.connection.transaction_manager is
            ↪  None or self.connection.transaction_manager.get().status ==
            ↪  'Inactive':
                print("Reinitializing DB connection")  # Debug print
                self.db, self.connection, self.root = get_db_connection()
            else:
                print("DB connection is active")  # Debug print
            return self.db, self.connection, self.root

connection_pool = DBConnectionPool()
```

Isječak koda 7: Upravljanje bazom podataka putem Connection Pooling-a u skripti `server.py`

Klasa ‘DBConnectionPool‘ implementira Connection Pooling za upravljanje vezama s bazom podataka. Koristi se za učinkovito rukovanje s ponovnim uspostavama veze i optimizaciju performansi aplikacije.

```python
@app.before_request
def before_request():
    global db, connection, root
    db, connection, root = connection_pool.get_connection()

@app.teardown_appcontext
def close_connection(exception):
    _, connection, db = connection_pool.get_connection()
    if connection is not None:
        if connection.transaction_manager is not None and
        ↪  connection.transaction_manager.get().status == 'Active':
            connection.transaction_manager.abort()
        connection.close()
```

Isječak koda 8: Flask middleware i teardown funkcije u skripti `server.py`

'before_request()' middleware funkcija osigurava da svaki zahtjev ima pristup aktualnoj bazi podataka putem Connection Pooling-a. Funkcija 'teardown_appcontext()' se koristi za zatvaranje veze s bazom podataka nakon svakog zahtjeva.

```python
if __name__ == '__main__':
    _, connection, _ = connection_pool.get_connection()
    while connection is None or connection.transaction_manager is None or
    ↪  connection.transaction_manager.get().status == 'Inactive':
        print("Waiting for DB connection to be ready...")
        time.sleep(1)
        _, connection, _ = connection_pool.get_connection()
    print("Starting Flask server...")
    app.run(host='0.0.0.0', port=5000, debug=True)
```

Isječak koda 9: Pokretanje Flask poslužitelja u skripti `server.py`

Ovaj dio kôda pokreće Flask poslužitelj nakon uspješne inicijalizacije veze s bazom podataka i čeka da veza bude spremna prije pokretanja poslužitelja.

Prilikom kreiranja ove baze podataka, suočavamo se s brojnim izazovima i problemima. Prvi problem koji se pojavio bio je pokretanje ZODB poslužitelja bez ZEO-a. Ako se poslužitelj pokuša pokrenuti bez ZEO-a, kreira dokument s nastavkom ".fs.lock". Svrha ovog dokumenta je spriječiti više procesa da istovremeno pokušaju izmijeniti podatke u bazi, čime bi se izbjegle razne greške. Međutim, nakon što poslužitelj kreira ovu datoteku, sprječava i sam sebe u pristupu bazi podataka, uzrokujući grešku i prekid rada. Zbog toga je bilo potrebno koristiti ZEO, koji omogućuje raznim procesima istovremeni pristup bazi podataka putem ZEO poslužitelja.

Za pokretanje ZEO poslužitelja, potrebno je napisati konfiguraciju koja specificira port na kojem se poslužuje ZEO te put do baze podataka:

```
<zeo>
  address 8100
</zeo>

<filestorage>
  path game_data.fs
</filestorage>
```

Isječak koda 10: Konfiguracija ZEO poslužitelja zapisana u datoteci `zeo.conf`

Nakon toga se pojavio drugi problem - bez ikakvog specifičnog razloga, Flask poslužitelj ignorira prvi POST zahtjev. Stoga je potrebno prilikom pokretanja poslužitelja poslati barem jedan GET ili POST zahtjev.

Treći problem se pojavio prilikom modeliranja podataka. Bilo je potrebno da svaki mač i dragulj imaju poseban ID kako bi se izbjegli nesporazumi o kojem objektu je riječ prilikom izvođenja pojedinih funkcija. Prvi pokušaj implementacije koristio je integer varijablu koja bi se inkrementalno povećavala svaki put kada se objekt kreira. Međutim, problem je bio u tome što bi ZODB zaboravio na kojem broju je stao svaki put kada bi se ZEO poslužitelj ponovno pokrenuo, te bi krenuo brojati ispočetka, što nije prihvatljivo. Zbog toga je bilo potrebno koristiti biblioteku "uuid" koja samostalno postavlja jedinstveni ID svakom objektu za kojeg je to potrebno.

3.5.2. MySQL i relacijska baza podataka

Ova baza podataka, kao što je već spomenuto, namijenjena je isključivo za sustav prijave i registracije korisnika/igrača te za praćenje njihovih rezultata tijekom igre. Prilikom pokretanja igre, igrač mora proći kroz proces registracije za novi račun ili prijavu na postojeći račun. Svi podatci se prenose preko HTTP protokola, stoga je važno da korisnici ne koriste svoju uobičajenu lozinku (jer zlonamjerne osobe mogu pristupiti njihovim podatcima preko "sniffinga" paketa). Tijekom registracije, korisnici imaju opciju unosa svoje e-mail adrese koja je korisna u slučaju zaboravljene lozinke ili korisničkog imena.

Podatci se pohranjuju u MySQL bazu podataka bez enkripcije, što znači da su korisnički podatci vidljivi u bazi, što dodatno potiče važnost korištenja jedinstvenih lozinki.

Osim toga, MySQL baza podataka se koristi za prikaz i pohranu rezultata koje igrači postižu tijekom igre. Igrači imaju uvid u sve bitne informacije o performansama drugih igrača, uključujući nadogradnje karaktera, odabrane likove, vrste nanesene i primljene štete, osvojene bodove, težinu igre na kojoj su igrali, i još mnogo toga.

Za ovu bazu podataka, međutim, se ne koriste nikakvi dijagrami zbog njene jednostavnosti. Sastoji se od samo dvije tablice. Prva tablica sadrži podatke o korisničkim računima, uključujući `ID` (`integer`), `username` (`string`), `email` (`string`) i `password` (`string`). Druga tablica prikuplja sve podatke o završetku pojedine igre, uključujući `ID` (`integer`), `name` (`string`), `score` (`integer`), `maxhp` (`integer`), `AD` (`integer`), `AP` (`integer`), `Armour` (`integer`), `Pen` (`integer`), `Cleave` (`integer`), `CritP` (`integer`), `Level` (`integer`), `password` (`string`), `DMGdone` (`integer`), `DMGmitigated` (`integer`), `DMGtaken` (`integer`), `PhysDMG`

(integer), APDMG (integer) i Class (string).

Slijede objašnjenja pojedinih podataka:

- ID je jedinstveni broj koji označava pojedini rezultat igrača.
- name je naziv rezultata kojeg dodjeljuje igrač završetkom igre.
- score su ostvareni bodovi igrača.
- maxhp je maksimalna vrijednost zdravlja koju je karakter igrača imao.
- AD predstavlja snagu udarca karaktera (*Attack Damage*).
- AP predstavlja snagu čarolija koje karakter koristi (*Ability Power*).
- Armour predstavlja snagu zaštite karaktera.
- Pen predstavlja koliko bi zaštite protivnika karakter ignorirao (*Penetration*).
- Cleave predstavlja koliko posto zaštite protivnika bi karakter ignorirao.
- CritP predstavlja postotak šanse za kritični udarac (*Critical hit percentage*).
- Level je razina snage koju je karakter postigao.
- password je zapisan i u ovoj tablici kako bi se spriječilo brisanje tuđih rezultata unosom istog naziva rezultata.
- DMGdone je ukupna šteta nanesena protivnicima (*damage done*).
- DMGmitigated je ukupna šteta umanjena pomoću zaštite karaktera (*damage mitigated*).
- DMGtaken je ukupna šteta koju je karakter primio (*damage taken*).
- PhysDMG je ukupna fizička šteta koju je karakter naneo (*physical damage*).
- APDMG je ukupna šteta koju je karakter nanio čarolijom (*ability power damage*).
- Class uključuje koja je vrsta karaktera korištena, kao i koja je težina igre bila odabrana.

Zanimljivost koja se ne prikazuje dijagramima je integracija podataka između MySQL baze podataka i ZODB-a. MySQL se koristi za autentifikaciju korisnika i bilježenje rezultata igre, ali je ključan i za upravljanje korisničkim inventarom u ZODB-u. Prilikom registracije novog igrača u MySQL bazu podataka, također se registrira i novi igrač u ZODB kako bi se omogućilo pristupanje njegovom inventaru tijekom igre. Prijava igrača na svoj račun provjerava se u MySQL bazi podataka, a zatim se bilježi i uspješna prijava u ZODB kako bi se osiguralo dosljedno korisničko iskustvo u igri.

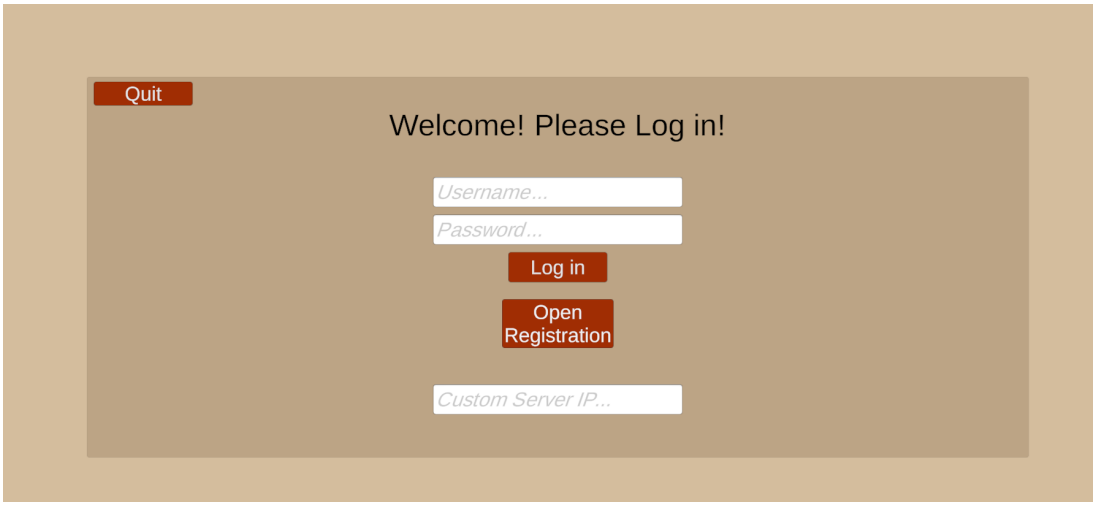
3.6. Razvoj videoigre

Ova igra je razvijena na temelju svoje prethodne iteracije. Prethodno, ova igra je bila kreirana kao web-stranica te je koristila HTML, CSS i JS. To nije bilo idealno jer je dizajn korisničkog sučelja bio prilično naporan i nespretan te kôd videoigre nije bio modularan. Pomoću Unity-a razvoj videoigre je poprilično efikasniji. Stoga je na početku rada već bio dostupan sveukupan dizajn koji se pratio prilikom razvoja videoigre.

3.6.1. Prolazak kroz videoigru

Videoigra počinje s zaslonom za prijavu ili registraciju korisnika. Na tom zaslonu igrač unosi svoje korisničko ime i lozinku. Ako igrač nema račun, može kliknuti na "Open Registration" i stvoriti novi korisnički račun. Prilikom registracije, korisnik treba unijeti korisničko ime, lozinku i e-mail. Iako e-mail nije obavezan za igru, potreban je za oporavak računa ako korisnik zaboravi korisničko ime ili lozinku. Nakon unosa svih potrebnih podataka, korisnik treba upisati IP adresu poslužitelja. Budući da se poslužitelj pokreće na računalu s dinamičkom IP adresom, važno je provjeriti trenutnu IP adresu. Za testiranje i komunikaciju s igračima, koristi se Discord aplikacija za distribuciju igre i ažuriranja IP adrese poslužitelja.

U prvoj verziji ekrana za prijavu umjesto "Open Registration" je pisalo "Register" što je motiviralo igrače da prvo upišu podatke u polja za login a tek onda kliknu "Register" što bi rezultiralo frustracijom igrača iz razloga što bi se uneseni podatci obrisali te bi se pojavilo novo polje u koje se zapisuje e-mail adresa. Kako bi se zaobišao problem, tekst gumba "Register" se promijenio u "Open Registration". Slijedi slika 2 koja prikazuje konačnu verziju ekrana za prijavu.



Slika 2: Slika zaslona prijave u videoigru

Nakon prijave u videoigru igrač se susreće s ekranom za odabir lika/karaktera kojeg koristi tijekom svoje igre. Osim što bira lika, također bira i jedan predmet kojeg će ponijeti sa sobom. Može ponijeti ili mač ili dragulj.

Dragulji ojačavaju karakteristiku odabranog lika ukoliko ikona karakteristike i dragulj imaju istu ili barem sličnu boju. Moguće boje su zelena za povećanje ukupnog zdravlja (max health), crvena za povećanje štete uzrokovane kritičnim udarcima (critical hit damage), žuta za snažnije udarce (attack damage), plava za jaču snagu čarolije (ability power) i ljubičasta za jače probijanje zaštite protivnika (armour penetration). Dragulji se mogu nadograditi time što se za njih plati novčanom valutom igre ili time što se skupi tri dragulja iste vrste i snage.

U mač je moguće ugraditi do tri dragulja te se na taj način može u igru ponijeti više predmeta. Ali zaraditi mač je problem. Kako bi igrač zaradio mač, mora oboriti rekord prvoga igrača na tablici rezultata. Ako igrač ostvari barem pola bodova prvoga igrača s tablice, bit će nagrađen samo jednim draguljem nasumične boje i snage prve razine.

Osim karaktera i predmeta, igrač mora odabrati i razinu težine igre. Prethodno je već navedena mehanika tajmiranja napada na koju utječe ovaj odabir. Igrač može vidjeti utjecaje svojeg odabira na popisu karakteristika odabranog lika koja se nalazi na gornjem desnom dijelu zaslona. Podatci tog popisa se ažuriraju prilikom promjene odabranog lika i odabranog predmeta. Svaki lik ima i posebnu karakteristiku čiji se opis nalazi na doljnjem lijevom kutu zaslona.

U doljnjem desnom kutu igrač ima opciju prodati odabrani predmet, nadograditi predmet ukoliko je odabran dragulj i ako igrač ima dovoljno novčane valute te ima opciju zamijeniti tri jednaka dragulja za jedan dragulj iste boje ali veće snage.

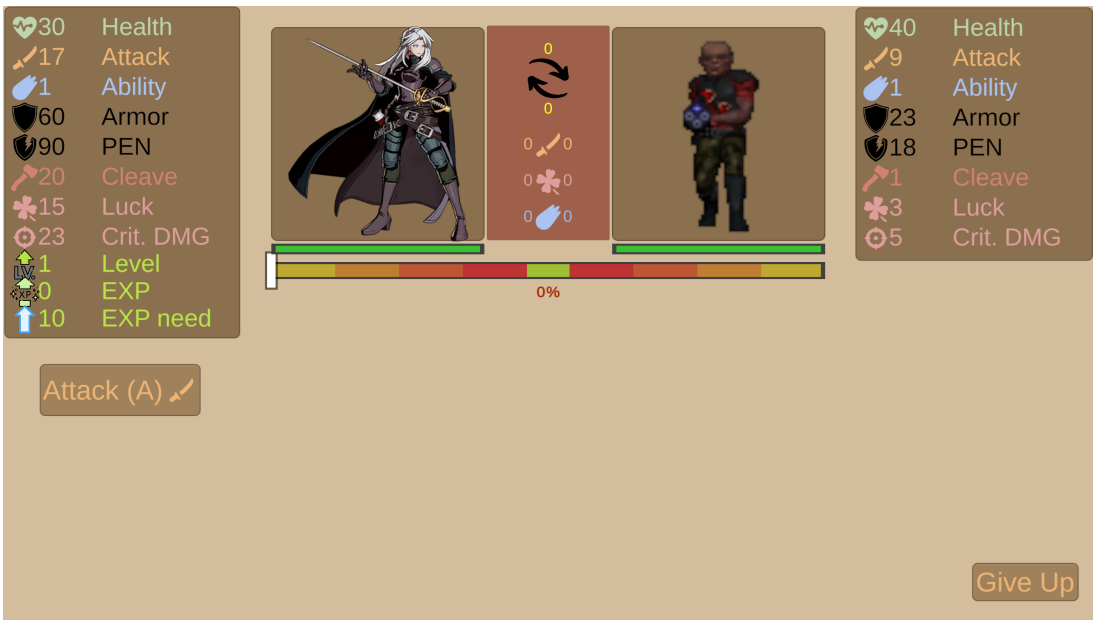
Kako bi ugradio dragulj u mač igrač mora prvo kliknuti na mač, zatim kliknuti na gumb "Socket" te na kraju kliknuti na dragulj kojega želi ugraditi. Nakon toga se ažurira ekran sa željenim promjenama. Dragulji na sebi imaju napisan broj koji predstavlja njihovu snagu, no ako su ugrađeni u mač taj broj se ne prikazuje. Bez obzira na to, njihov efekt je očit na popisu karakteristika ako je mač odabran. U nastavku slijedi slika 3 koja prikazuje opisani zaslon.

Slika 3: Slika zaslona pripreme za igru

Nakon što igrač napravi bilo koju od prethodno navedenih promjena, videoigra šalje POST zahtjeve, ponekad i s JSON sadržajem (engl. payload), Flask serveru koji zatim provodi zatražene promjene u ZODB. Nakon što primi potvrdu da je funkcija dobro završila, igra šalje GET zahtjev Flask serveru kako bi ažurirala podatke koje igraču prikazuje. U ovom procesu je nastao problem. Igra bi poslala Flask serveru dva GET zahtjeva istovremeno. Jedan GET zahtjev je tražio inventar igrača, a drugi stanje novčane valute koju igrač posjeduje. Flask poslužitelj bi ponekad odbio odgovoriti na prvi GET zahtjev te se inventar igrača ne bi učitao. Zbog toga videoigra prvo šalje jedan GET zahtjev. Tek nakon što primi odgovor prvog GET zahtjeva - šalje drugi.

Zaslon također ima nekoliko gumba na samom vrhu. Jedan gumb je za izlazak iz igre, drugi za odjavu igrača, treći i četvrti gumbi nisu funkcionalni jer njihove funkcije još nisu implementirane te peti gumb otvara tablicu rezultata koje su igrači postigli.

Kad je igrač zadovoljan sa svojim odabirom pokreće igru klikom na gumb "Start" te mu se prikazuje sljedeći zaslon.



Slika 4: Slika zaslona pokrenute igre

Na ovom zaslonu igrač ima pregled statusa svojeg karaktera/lika na lijevoj strani zaslona kao i pregled statusa svojeg protivnika na desnoj strani zaslona. Na sredini zaslona su prikazane vrijednosti zadane štete, konkretno je napisano koliko su u jednom udarcu igrač i njegov protivnik nanijeli štete jedan drugome. Prva dva broja predstavljaju ukupnu nanesenu štetu, druga dva fizičku, treća dva predstavljaju štetu kritičnog udarca te zadnja dva prestavljaju štetu postignutu čarolijom. Ispod crteža odabranog lika igrača i crteža protivnika nalazi se njihov prikaz zdravlja (engl. health bar). Ispod toga se nalazi horizontalna linija koja služi mehanici tajmiranog udarca koji je prethodno opisan. Ispod te horizontalne linije se nalazi postotak koji predstavlja koliko precizno je igrač zaustavio vertikalnu liniju na polovici horizontalne linije.

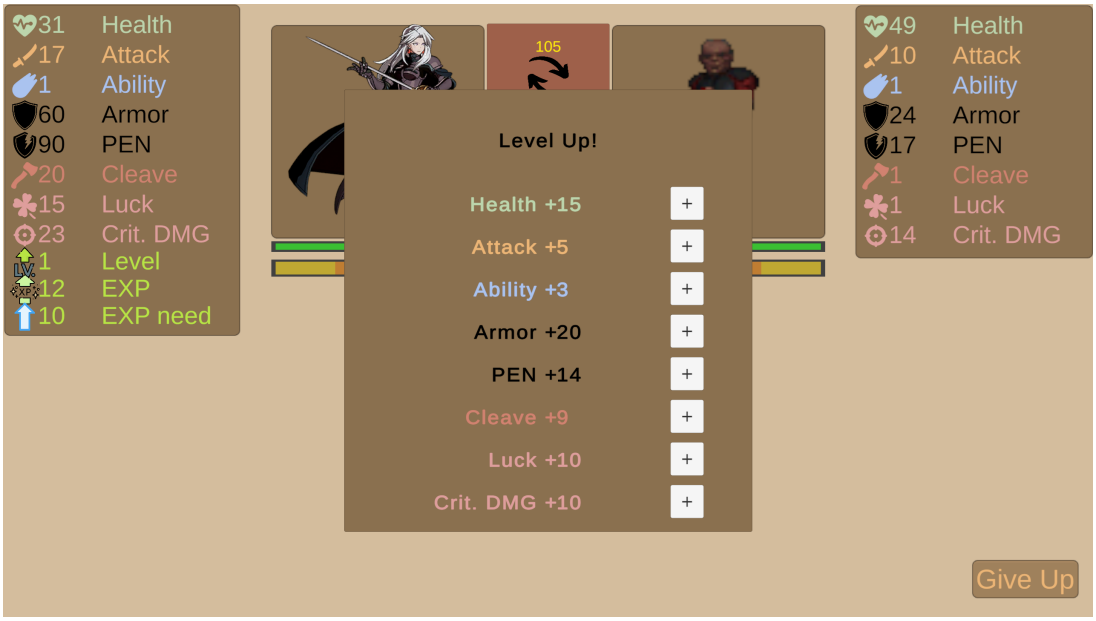
Igrač ima samo dvije opcije. Prva opcija je napasti protivnika pomoću gumba "Attack"

ili korištenjem tipke "A" na tipkovnici. Druga opcija je odustati klikom na gumb "Give Up" što je pogotovo korisno ako igrač nadmaši cjelokupnu igru te nije u mogućnosti izgubiti i privesti igru kraju.

Kao što se može vidjeti na slici 4, sprite-ovi likova igrača su nasumični crteži preuzeti od raznih autora. A sprite-ovi protivnika su preuzeti iz serijala videoigara naziva "DOOM". Radi osjećaja responzivnosti igre na akcije igrača protivnik je također animiran. Ukoliko primi udarac, pokrene se animacija u kojoj je protivnik zanesen te se pokrene zvuk udarca i reakcije protivnika koje su također preuzete iz serijala videoigra "DOOM". Ako igrač postigne prilično visoke bodove, sprite-ovi i zvukovi protivnika su zamjenjeni čudovištem "Imp" iz prvog izdanja serijala "DOOM". Ako je postignut kritičan udarac također se pusti drugačiji zvuk udarca što bi trebalo oraspoložiti igrača iz razloga što je imao sreće.

Videoigra također ima razne tajne mehanike, jedna od njih se zove "Overkill". Ostvaruje se tako da igrač toliko smanji zdravlje (engl. health) protivniku da protivnik nakon udarca ima zdravlje u minusu većem nego što je u početku imao maksimalno zdravlja. Na primjer, ako je protivnik na početku imao zdravlje na broju deset (engl. 10 Health) a nakon udarca igrača ostane mu najviše minus deset zdravlja (engl. -10 Health) protivniku se pokrene posebna animacija i zvuk.

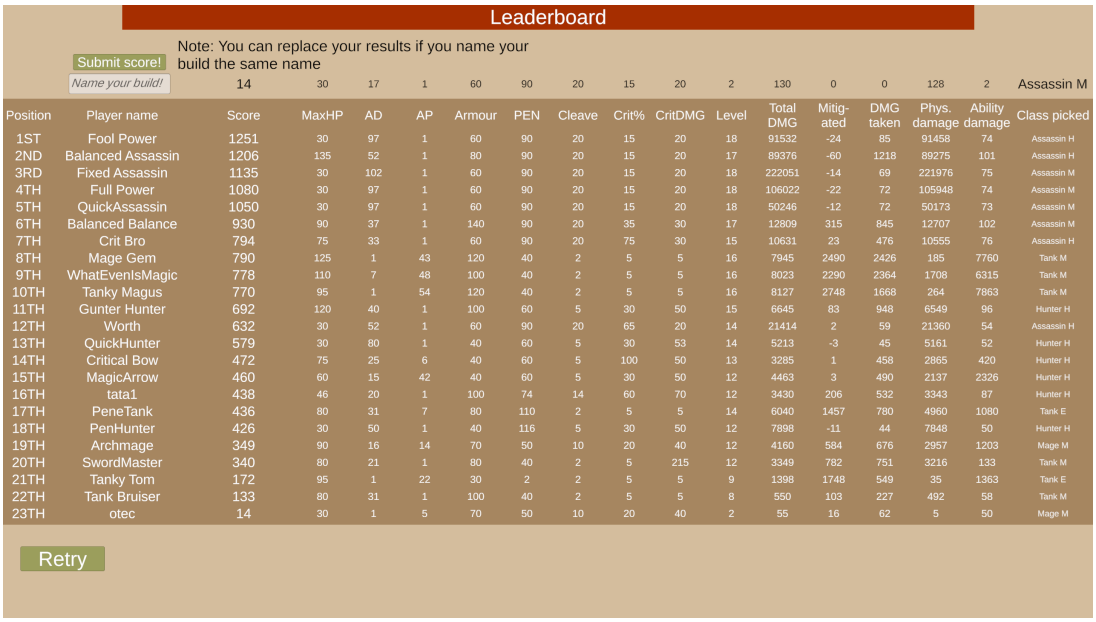
Nakon što igrač porazi protivnika time što mu smanji zdravlje na nulu ili manje, nagrađen je iskustvom (engl. EXP - experience) i povratkom zdravlja na trenutni maksimum te se susreće sa sljedećim protivnikom. Nakon što skupi dovoljno iskustva za podizanje razine snage karaktera (engl. level up) igraču iskače sljedeći prozor unutar kojeg može nadograditi bilo koju karakteristiku svojeg karaktera.



Slika 5: Slika iskočnog prozora za nadogradnju karaktera

Klikom na gumb "+" pored željene nadogradnje skočni prozor se zatvara, odabrana karakteristika se nadograđuje te se igra nastavlja dok igrač ne doživi poraz time što mu zdravlje

padne na nulu ili manje. Nakon što je igrač poražen ili nakon što odustane otvara se sljedeći zaslon.



Slika 6: Slika tablice rezultata nakon igre

Ovaj zaslon prvo provjerava je li igrač došao ovdje klikom na gumb "Leaderboard" u zaslonu za odabir karaktera. U tom slučaju su prikazani samo rezultati drugih igrača. Ako je došao s bilo kakvim bodovima (što znači da je prošao kroz igru), igrač ima opciju zabilježiti svoju igru na tablici time što napiše naziv svoje igre u polje "Name your build!". Ako napiše isto ime svoje druge igre, ta je igra onda prebrisana novim rezultatima. Ukoliko igrač pokuša nazvati svoju igru istim nazivom igre nekog drugog igrača - igra ispisuje grešku i odbija prihvatiti slanje rezultata. Ukoliko je igrač oborio rekord igrača na vrhu tablice također se pojavi i zeleno polje unutar kojega igrač može dati ime maču kojeg je osvojio.

Na tablici su prikazani svi relevantni podatci za pregled ideja i rezultata koje su drugi igrači ostvarili. Nakon što igrač objavi svoj rezultat tablica se ažurira. Kada je igrač pregledao svoje i/ili tuđe rezultate može se vratiti na ekran za odabir karaktera klikom na gumb "Retry". Igrač zatim vidi promjene u svojem inventaru ako je zaradio mač ili dragulj te može izaći iz igre klikom na gumb "Quit" ili pokušati ponovno igrati.

3.6.2. Prolazak kroz kôd videoigre

Ova videoigra sadrži četiri Unity scene kroz koje igrač prolazi u sljedećoj sekvenci: Login, CharacterSelect, GameplayScene, LeaderBoard. Kroz sve scene koristi se Scriptable Object pod nazivom "PlayerData" stoga je za sve scene bitna njegova istoimena skripta `PlayerData.cs`. Scriptable Object se koristi kao u memoriji (engl. in-memory) objektno-orijentirana baza podataka. Odnosno, koristi se za čuvanje objekata u memoriji računala korisnika/igrača kako bi se spremljeni objekti mogli koristiti na svim Unity scenama kroz koje igrač prolazi. U nastavku se može pregledati kôd te skripte.

```
using UnityEngine;

[CreateAssetMenu(fileName = "PlayerData", menuName = "Custom/PlayerData", order =
↪  2)]
public class PlayerData : ScriptableObject
{
    public int score;
    public string username; //it's not "name" because of syntax
    public string password;
    public string serverIP;
    public string baseURL;
    public string ZODBURL;
    public int maxhp;
    public int AD;
    public int AP;
    public int Armour;
    public int Pen;
    public int Cleave;
    public int CritP;
    public int CritDMG;
    public int Level;
    public int DMGdone;
    public int DMGmitigated;
    public int DMGtaken;
    public int PhysDMG;
    public int APDMG;
    public string className;

    public void ResetScore()
    {
        score = 0;
        maxhp= 0;
        AD= 0;
        AP= 0;
        Armour= 0;
        Pen= 0;
        Cleave= 0;
        CritP= 0;
        CritDMG= 0;
        Level= 0;
        DMGdone= 0;
        DMGmitigated= 0;
        DMGtaken= 0;
        PhysDMG= 0;
        APDMG= 0;
        className = "";
    }
}
```

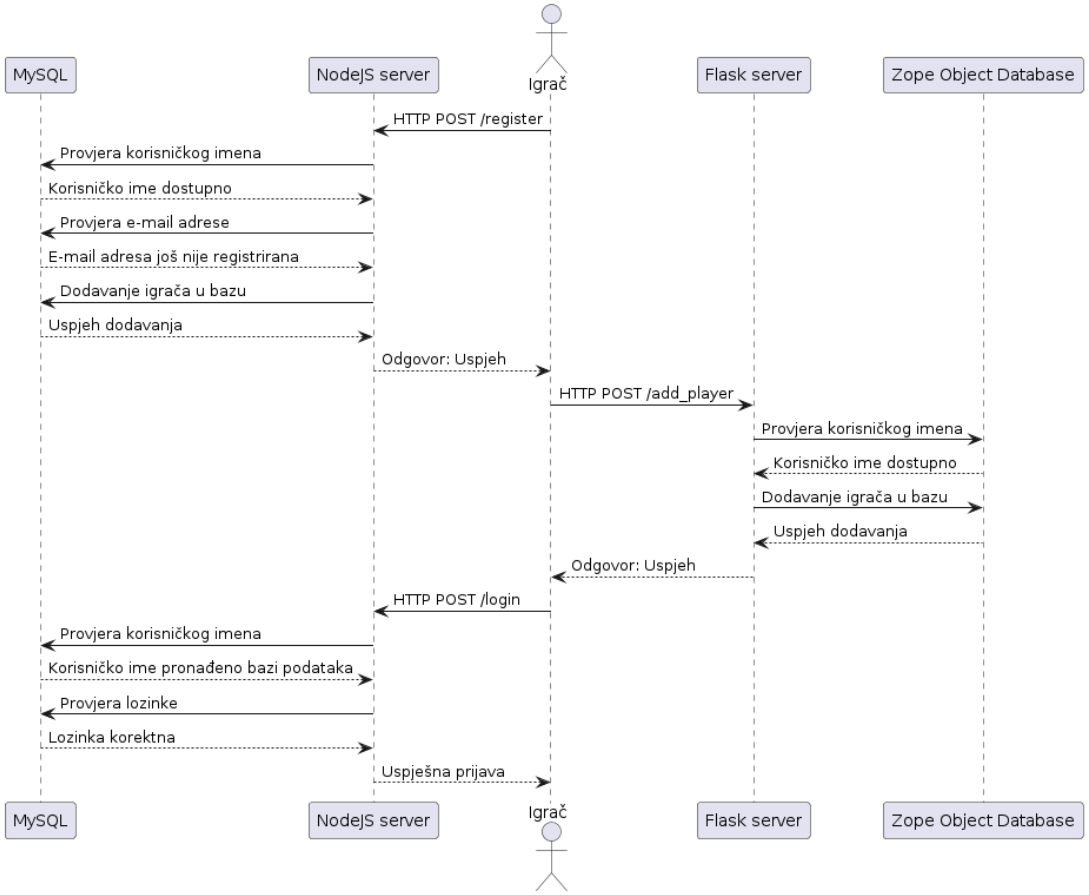
Isječak koda 11: Kôd skripte `PlayerData.cs`

PlayerData Scriptable Object bilježi koji korisnik/igrač je prijavljen u videoigru te bilježi

razne statističke podatke koji su potrebni za pregled i objavljivanje rezultata kojeg igrač postigne tijekom igre. Ovaj objekt ima samo jednu metodu naziva "ResetScore" koja samo resetira sve statističke podatke kada igrač otvori zaslon odabira karaktera. U sljedećim potpoglavljima prikazan je sav bitan kôd za pokretanje ove videoigre te njezinu komunikaciju s bazama podataka. Nije uključen kôd koji nije relevantan za ovaj rad kao što su, na primjer, tajne mehanike videoigre.

3.6.2.1. Login scena

Za "Login" scenu relevantne su sljedeće skripte: `LogIn.cs` i `PlayerData.cs`. Stoga je u nastavku analiza kôda skripte `LogIn.cs` koja je glavni pokretač scene. Ta skripta upravlja podatcima koje igrač unese prilikom procesa prijave ili registracije računa. Također prilikom klika na gumb "Open Registration" skripta otvara ekran za registraciju. Ukoliko se dogodi greška, skripta ispisuje crveni tekst koji objašnjava korisniku/igraču u čemu je problem. Ako korisnik/igrač klikne na gumb "Log in" pokreće se funkcija "Login()" koja zatim pokreće funkcije "UpdateServerIP(serverIP.text)" i "LoginRequest(username, password)". Slijedi prikaz navedenih funkcija nakon dijagrama sekvenci.



Slika 7: Slika dijagrama sekvenci Login scene

```
public void Login()
{
    string username = usernameLogin.text;
    string password = passwordLogin.text;

    UpdateServerIP(serverIP.text);
    StartCoroutine(LoginRequest(username, password));
}
```

Isječak koda 12: Login funkcija skripte `LogIn.cs`

Funkcija "Login()" zabilježi što je korisnik/igrač unio u polje "usernameLogin" i "passwordLogin" te proslijedi te podatke funkciji "LoginRequest(username, password)" koja se pokreće kao korutina. Osim toga, prosljeđuje se upisana IP adresa funkciji "UpdateServerIP(serverIP.text)" kako bi se mogao na korektnu IP adresu poslati POST zahtjev. U nastavku slijedi prikaz funkcije "UpdateServerIP(serverIP.text)".

```
public void UpdateServerIP(string IP)
    {
        playerData.serverIP = IP;
        baseURL = "http://" + playerData.serverIP + ":3074";
        ZODBURL = "http://" + playerData.serverIP + ":5000";
        playerData.baseURL = baseURL;
        playerData.ZODBURL= ZODBURL;
        Debug.Log("IP:" + IP + " baseURL:" + baseURL + " ZODBURL:" + ZODBURL);
    }
```

Isječak koda 13: Funkcija za ažuriranje IP adrese poslužitelja, skripta `LogIn.cs`

Funkcija "UpdateServerIP(string IP)" prihvaća niz znakova (engl. string) IP koji se sprema u Scriptable Object Player Data kako bi se koristio u drugim Unity scenama. Zatim se sastave URL-ovi na koje se šalju POST i GET zahtjevi bazama podataka te se ti URL-ovi također spreme u Player Data za daljnje korištenje. NodeJS poslužitelj MySQL-a se poslužuje na portu 3074, a Flask poslužitelj za ZEO se poslužuje na portu 5000. Slijedi prikaz funkcije "LoginRequest(username, password)".

```
IEnumerator LoginRequest(string username, string password)
    {
        string loginURL = baseURL + "/login";
        string jsonData = "{\"username\": \"" + username + "\", \"password\": \"" +
        ↪  password + "\"}";

        // Convert JSON data to bytes
        byte[] jsonDataBytes = System.Text.Encoding.UTF8.GetBytes(jsonData);

        // Create a UnityWebRequest and set its method to POST
        UnityWebRequest www = new UnityWebRequest(loginURL, "POST");

        // Set the request body as the JSON data bytes
        www.uploadHandler = new UploadHandlerRaw(jsonDataBytes);

        // Set the content type
        www.SetRequestHeader("Content-Type", "application/json");

        // Send the request and wait for the response
        yield return www.SendWebRequest();

        // Check for errors
        if (www.result != UnityWebRequest.Result.Success)
        {
            ErrorMessage.text = "Cannot connect to server";
            Debug.LogError("Error logging in: " + www.error);
        }
        else
        {
            playerData.username = username;
            playerData.password = password;
            SceneManager.LoadScene(1);
        }

        if (www.responseCode == 401)
        {
            ErrorMessage.text = "Wrong username or password";
            Debug.LogError("Unauthorized access: " + www.error);
        }

        // Dispose the UnityWebRequest to free resources
        www.Dispose();
    }
```

Isječak koda 14: Funkcija za slanje POST zahtjeva NodeJS serveru u svrhu prijave korisnika, skripta `LogIn.cs`

Funkcija "LoginRequest(string username, string password)" na "login" endpoint NodeJS poslužitelja šalje POST zahtjev unutar kojeg se nalazi korisničko ime i lozinka koje je korisnik/igrač upisao. Ako je došlo do greške, skripta na zaslon ispisuje poruku koja obavještava korisnika o vrsti greške. Ako je primljena poruka kôda 200 (u skripti UnityWebRequ-

est.Result.Success), korisnika se uputi na scenu s indeksom jedan (1), točnije na scenu odabira karaktera. Kôd endpoint-a poslužitelja NodeJS i Flask moguće je pregledati u poglavlju "Povezivanje videoigre s bazama podataka".

Proces registracije je sličan prijavi ali uz male, no značajne razlike. Prvo korisnik/igrač mora kliknuti na gumb "Open Registration" što pokrene jednostavnu funkciju koja sakrije ekran za prijavu te prikaže ekran za registraciju. Na novom ekranu potrebno je unijeti korisničko ime, lozinku te e-mail adresu. Nakon što su uneseni potrebni podatci kao i IP adresa poslužitelja klikom na gumb "Register" pokreće se funkcija "Register()" čiji se prikaz nalazi u nastavku.

```
1 public void Register()
2 {
3     string username = usernameRegister.text;
4     string password = passwordRegister.text;
5     string email = emailInput.text;

6     UpdateServerIP(serverIPRegister.text);
7     StartCoroutine(RegisterRequest(username, password, email));
8     StartCoroutine(AddPlayerToZODB(username));
9 }
```

Isječak koda 15: Funkcija za registraciju novog korisnika, skripta `LogIn.cs`

Funkcija "Register()" prosljeđuje korisničko ime, lozinku i e-mail adresu funkciji "RegisterRequest(username, password, email)". Također prosljeđuje IP adresu poslužitelja funkciji "UpdateServerIP(serverIPRegister.text)". Na kraju prosljeđuje uneseno korisničko ime funkciji "AddPlayerToZODB(username)". Najprije je prikazana funkcija "RegisterRequest(username, password, email)".

```
IEnumerator RegisterRequest(string username, string password, string email)
{
    string registerURL = baseURL + "/register";
    string jsonData = "{\"username\": \"" + username + "\", \"password\": \"" +
    ↪  password + "\", \"email\": \"" + email + "\"}";

    // Convert JSON data to bytes
    byte[] jsonDataBytes = System.Text.Encoding.UTF8.GetBytes(jsonData);

    // Create a UnityWebRequest and set its method to POST
    UnityWebRequest www = new UnityWebRequest(registerURL, "POST");

    // Set the request body as the JSON data bytes
    www.uploadHandler = new UploadHandlerRaw(jsonDataBytes);

    // Set the content type
    www.SetRequestHeader("Content-Type", "application/json");

    // Send the request and wait for the response
    yield return www.SendWebRequest();

    // Check for errors
    if (www.result != UnityWebRequest.Result.Success)
    {
        ErrorMessageRegister.text = "Cannot connect to server";
        Debug.LogError("Error registering: " + www.error);
    }
    else
    {
        playerData.username = username;
        playerData.password = password;
        SceneManager.LoadScene(1);
    }

    if (www.responseCode == 400)
    {
        ErrorMessageRegister.text = "E-mail already registered";
    } else if (www.responseCode == 401)
    {
        ErrorMessageRegister.text = "Username already in use";
    }
    // Dispose the UnityWebRequest to free resources
    www.Dispose();
}
```

Isječak koda 16: Funkcija za slanje POST zahtjeva NodeJS serveru u svrhu registracije novog korisnika, skripta `LogIn.cs`

Funkcija "RegisterRequest(string username, string password, string email)" šalje POST zahtjev na "register" endpoint NodeJS poslužitelja. U POST zahtjevu šalje se korisničko ime, lozinka i e-mail adresa. Ukoliko dođe do greške skripta obavještava korisnika o vrsti greške.

Tek kada je primljen odgovor da je registracija uspjela u Player Data se sprema korisničko ime i lozinka korisnika te se učitava scena odabira karaktera. Na kraju slijedi funkcija za registraciju novog korisnika u ZODB.

```
IEnumerator AddPlayerToZODB(string username)
{
    string addPlayerURL = ZODBURL + "/add_player/" + username;

    // Create a UnityWebRequest and set its method to POST
    UnityWebRequest www = new UnityWebRequest(addPlayerURL, "POST");

    // Set the content type
    www.SetRequestHeader("Content-Type", "application/json");

    // Send the request and wait for the response
    yield return www.SendWebRequest();

    // Check for errors
    if (www.result != UnityWebRequest.Result.Success)
    {
        Debug.LogError("Error adding player to ZODB: " + www.error);
    }
    else
    {
        Debug.Log("Player added to ZODB successfully");
    }

    // Dispose the UnityWebRequest to free resources
    www.Dispose();
}
```

Isječak koda 17: Funkcija za slanje POST zahtjeva Flask serveru u svrhu registracije novog korisnika, skripta `LogIn.cs`

Funkcija "AddPlayerToZODB(string username)" šalje POST zahtjev na "add_player" endpoint Flask poslužitelja. POST zahtjev sadrži samo korisničko ime. U navedenom kôdu nije kreirano rukovanje s različitim greškama koje se mogu pojaviti jer ih uglavnom nije ni bilo prilikom testiranja. Jedine pojave problema su u slučaju da je poslužitelj ugašen ili ako je korisnik već registriran onda će biti odbijen pokušaj ponovne registracije i na Flask i NodeJS serverima. Kôd endpoint-a je vidljiv u poglavlju "Povezivanje videoigre s bazama podataka" te je tamo vidljiva logika koja se koristi.

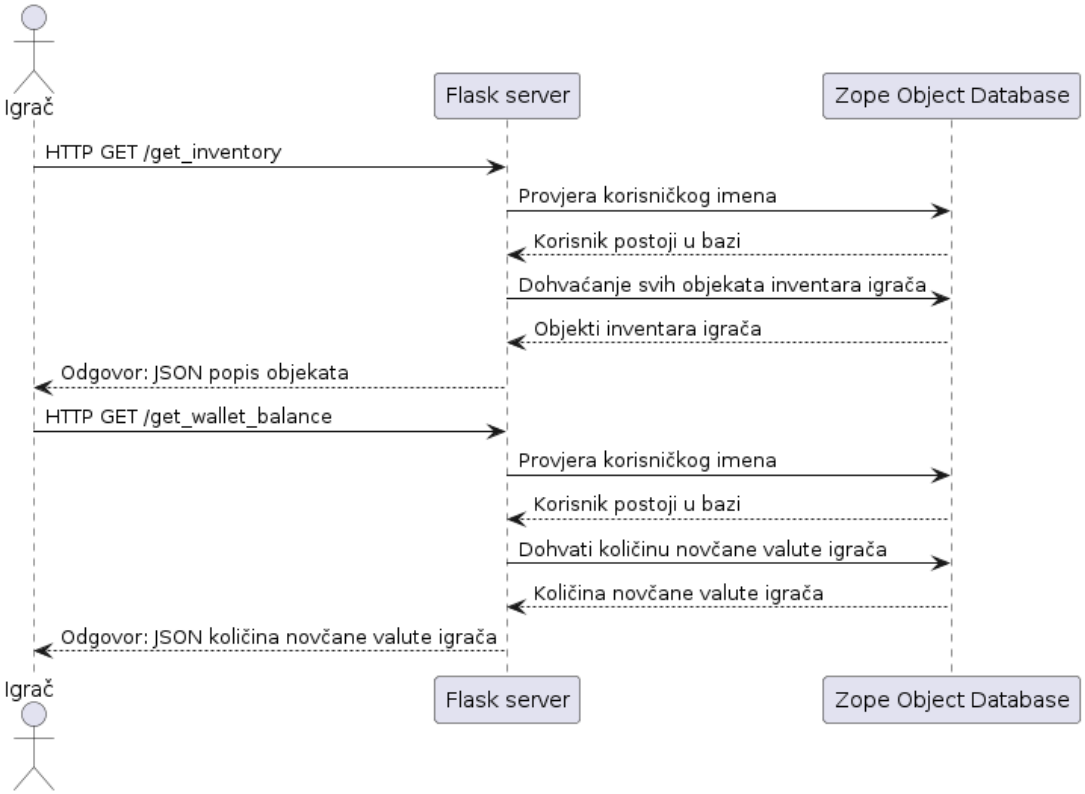
Tijekom testiranja korisnici su često pokušavali koristiti tipku "TAB" kako bi prešli u drugo polje za unos podataka stoga je implementirana ta mogućnost unutar "Update()" funkcije koja slijedi u nastavku.

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Tab))
    {
        Selectable next =
        ↪ system.currentSelectedGameObject.GetComponent<Selectable>().FindSelectableOnDown();
        if (next != null)
        {
            InputField inputfield = next.GetComponent<InputField>();
            if (inputfield != null)
                inputfield.OnPointerClick(new PointerEventData(system));  //if it's
                ↪ an input field, also set the text caret
            system.SetSelectedGameObject(next.gameObject, new
            ↪ BaseEventData(system));
        }
    }
}
```

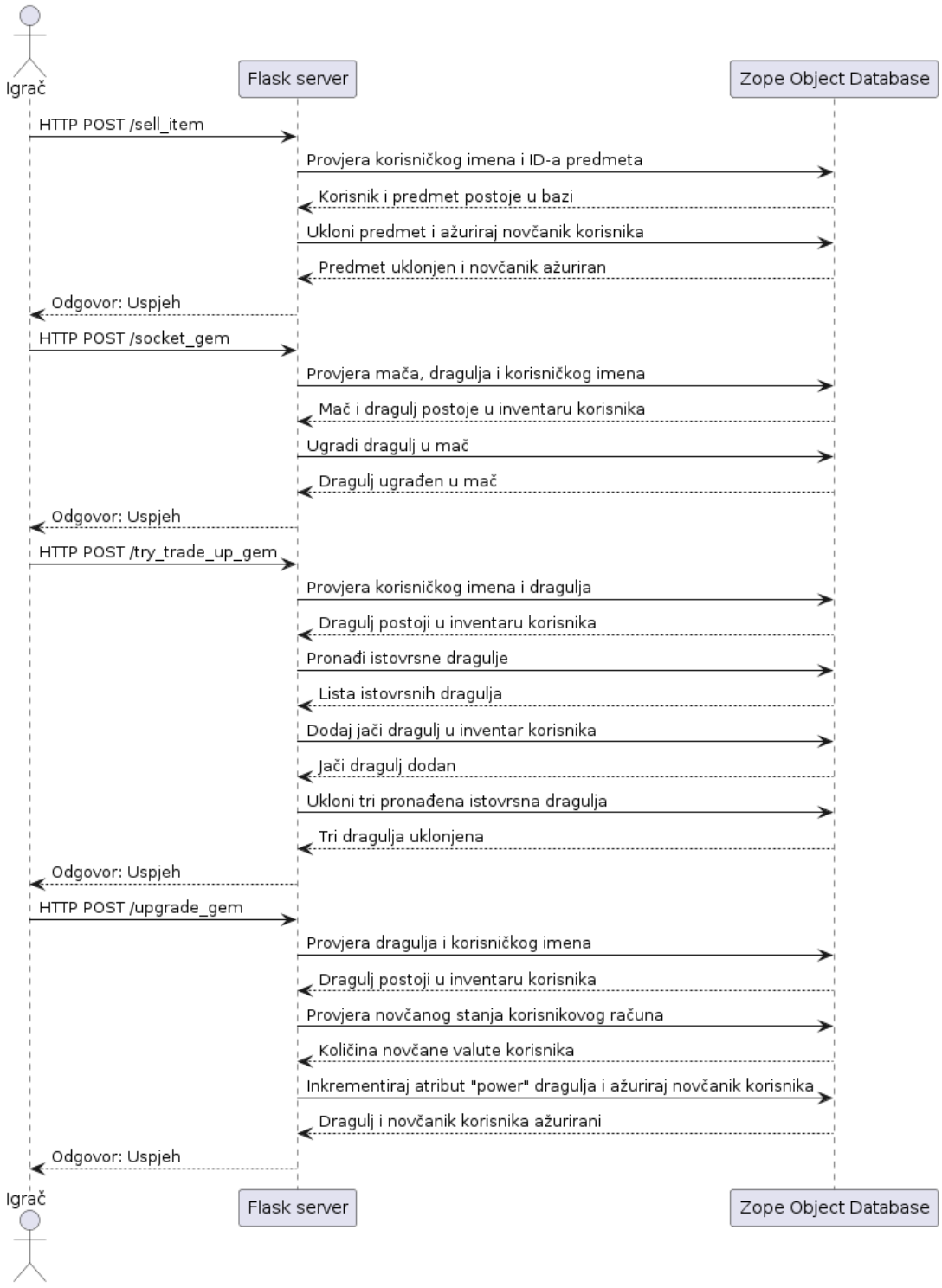
Isječak koda 18: Update funkcija skripte `LogIn.cs`

3.6.2.2. Scena CharacterSelect

Za Unity scenu CharacterSelect relevantne su sljedeće skripte: `CharacterSelect.cs`, `InventoryDisplayManager.cs`, `GameData.cs`, `SelectedItem.cs` i `PlayerData.cs`. Prvo bi bilo korisno pogledati skripte `GameData.cs` i `SelectedItem.cs` jer se pomoću tih manjih skripti kreiraju dva Scriptable Object-a koji služe za prijenos podataka na "GameplayScene" scenu. Prvo slijede dva dijagrama sekvenci a zatim prikaz skripte `GameData.cs`. Prvi dijagram sekvenci prikazuje proces učitavanja scene. Drugi dijagram sekvenci prikazuje sve dostupne interakcije igrača i ZODB-a.



Slika 8: Slika dijagrama sekvenci učitavanja CharacterSelect scene



Slika 9: Slika dijagrama sekvenci CharacterSelect scene

```
using UnityEngine;

[CreateAssetMenu(fileName = "GameData", menuName = "Custom/GameData", order = 1)]
public class GameData : ScriptableObject
{
    public int selectedClass;
    public int selectedDifficulty;
}
```

Isječak koda 19: Skripta za kreiranje Scriptable Object-a "GameData"

Vidljivo je da ovaj Scriptable Object samo bilježi kojeg karaktera je igrač odabrao pomoću varijable "selectedClass" te bilježi odabranu razinu težine igre pomoću varijable "selectedDifficulty". Zabilježeni podatci će biti iskorišteni u sceni "GameplayScene". Podatke ovog Scriptable Object-a ažurira skripta `CharacterSelect.cs` U nastavku je prikazana skripta `SelectedItem.cs`

```
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(fileName = "SelectedItem", menuName = "Custom/SelectedItem")]
public class SelectedItem : ScriptableObject
{
    public string id;
    public string itemName;
    public string itemType;
    public string itemColor;
    public int itemPower;
    public List<GemInfo> itemGems; // Only relevant for swords
}
```

Isječak koda 20: Skripta za kreiranje Scriptable Object-a "SelectedItem"

Ovaj Scriptable Object bilježi podatke odabranog objekta na kojeg je igrač kliknuo prilikom odabira predmeta iz prikaza inventara. Ovaj Scriptable Object očekuje podatke vezane uz mač ili dragulj. Ako igrač klikne na mač, spremit će se ID mača, ime mača, vrsta odabranog objekta i lista dragulja koji su ugrađeni u odabrani mač. Ako je igrač kliknuo na dragulj, učitat će se ID dragulja, boja dragulja, vrsta odabranog objekta te snaga dragulja. Ove podatke ažurira skripta `InventoryDisplayManager.cs`. Podatci ovog Scriptable Object-a će biti korišteni u sceni "GameplayScene". U nastavku se nalazi analiza glavnih procesa skripte `InventoryDisplayManager.cs`. Skripta je prilično velika te ima nekoliko sekvenci funkcija koje se provode kako bi se učitao inventar igrača te omogućilo upravljanje objektima unutar njega. Stoga prvo slijedi analiza spomenutih kritičnih sekvenci funkcija. Zatim slijedi analiza funkcija čija je svrha upravljanje inventarom i komuniciranjem promjena Flask serveru.

```
1  void Start()
2  {
3      nextButton.onClick.AddListener(NextPage);
4      previousButton.onClick.AddListener(PreviousPage);
5      socketButton.onClick.AddListener(OnSocketButtonClicked);
6      sellButton.onClick.AddListener(OnSellButtonClicked); // Add listener for the
       ↪  sell button
7      tradeUpButton.onClick.AddListener(OnTradeUpButtonClicked);  // Add listener for
       ↪  the tradeup button
8      upgradeButton.onClick.AddListener(OnUpgradeButtonClicked);  // Add listener for
       ↪  the upgrade button
9      socketButton.interactable = false; // Initially disable the socket button

10     StartCoroutine(LoadData());
11 }
```

Isječak koda 21: Start funkcija skripte `InventoryDisplayManager.cs`

Start funkcija skripte `InvenotryDisplayManager.cs` je početak sekvence funkcija koja se grana provedbom funkcije "LoadData()". Ova funkcija uglavnom dodaje "onClick Listener" na UI elemente koji uključuju gumbe za listanje stranica inventara (NextPage i PreviousPage), gumb za ugrađivanje dragulja u mač (OnSocketButtonClicked), gumb za prodaju odabranog predmeta (OnSellButtonClicked), gumb koji pokreće zamjenu tri dragulja za jedan dragulj veće snage te gumb za nadogradnju dragulja pomoću novčane valute igrača. Prije pozivanja funkcije "LoadData()" također se i zabrani korištenje gumba za ugrađivanje dragulja u mač jer ni jedan predmet u početku nije odabran. Funkciju "LoadData()" nije potrebno pregledati, ona jednostavno pokreće dvije funkcije kao korutine. Jedna funkcija učitava inventar igrača iz ZODB-a pomoću Flask poslužitelja. Druga funkcija šalje GET zahtjev za količinu novčane valute koju igrač posjeduje tek nakon što je prihvaćen odgovor Flask poslužitelja na prvu funkciju i inventar prikazan. Razlog čekanju jest problem koji se pojavio s Flask serverom. Ako se pošalju oba GET zahtjeva istovremeno Flask poslužitelj ponekad ne može odgovoriti te vraća poruku greške, najčešće ako je u pitanju učitavanje inventara jer je to prilično zahtjevna funkcija. Stoga funkcija "LoadData()" čeka kraj izvedbe prve funkcije pomoću "yield return StartCoroutine(...)" a zatim pokreće drugu funkciju na isti način. U nastavku slijedi analiza prve funkcije čiji je naziv "FetchInventory".

```
IEnumerator FetchInventory(string playerName)
{
    string url = $"http://{playerData.serverIP}:5000/get_inventory/{playerName}";
    UnityWebRequest request = UnityWebRequest.Get(url);
    yield return request.SendWebRequest();

    if (request.result == UnityWebRequest.Result.ConnectionError || request.result
    ↪  == UnityWebRequest.Result.ProtocolError)
    {
        Debug.LogError(request.error);
    }
    else
    {
        string jsonResponse = request.downloadHandler.text;
        Debug.Log("JSON Response: " + jsonResponse); // Log the JSON response

        try
        {
            JArray jsonArray = JArray.Parse(jsonResponse);
            foreach (JObject jsonObject in jsonArray)
            {
                string itemType = jsonObject["type"].ToString();
                if (itemType == "sword")
                {
                    Sword sword = jsonObject.ToObject<Sword>();
                    playerInventory.Add(sword);
                }
                else if (itemType == "gem")
                {
                    Gem gem = jsonObject.ToObject<Gem>();
                    playerInventory.Add(gem);
                }
            }
            InstantiateAllItems();
            DisplayPage(currentPage);
        }
        catch (JsonException ex)
        {
            Debug.LogError("JSON Deserialization error: " + ex.Message);
        }
    }
}
```

Isječak koda 22: FetchInventory funkcija skripte `InventoryDisplayManager.cs`

Funkcija "FetchInventory(string playerName)" prihvaća ime igrača u obliku znakovnog niza (engl. string) kao argument. Taj podatak funkcija "LoadData()" dohvaća i prosljeđuje iz Scriptable Object-a PlayerData. Pomoću istog Scriptable Object-a "FetchInventory()" funkcija slaže URL na kojeg će poslati GET zahtjev. GET zahtjev će biti poslan na IP adresu odabranog poslužitelja i port 5000, odnosno na Flask poslužiteljevu krajnju točku (engl. endpoint) "get_inventory". Ta krajnja točka uzima ime igrača, zatim pregledava ZODB te ako taj igrač

postoji - kao odgovor pošalje JSON sadržaj unutar kojeg su podatci inventara traženog igrača. Cijela logika endpointa je vidljiva u poglavlju "Povezivanje videoigre s bazama podataka". Ako funkcija uspješno primi JSON sadržaj slijedi spremanje primljenih podataka u listu predmeta naziva "playerInventory". Ta lista je instancirana kao `List<Item>` što znači da uključuje listu objekata koji su definirani pomoću klase "Item". Nakon što spremi sve objekte JSON sadržaja, pokreće se funkcija "DisplayPage()" koja prikazuje stranicu inventara s obzirom na integer koji je proslijeđen. U nastavku se nalazi kratka analiza "DisplayPage()" funkcije a zatim slijedi pregled klase "Item" radi razumijevanja oblika podataka koje Flask poslužitelj šalje u JSON sadržaju.

```
1  void DisplayPage(int page)
2  {
3      // Calculate the range of items to display
4      int startIndex = page * itemsPerPage;
5      int endIndex = Mathf.Min(startIndex + itemsPerPage, playerInventory.Count);

6      // Toggle visibility of items
7      for (int i = 0; i < inventoryItems.Count; i++)
8      {
9          if (i >= startIndex && i < endIndex)
10         {
11             inventoryItems[i].SetActive(true);
12         }
13         else
14         {
15             inventoryItems[i].SetActive(false);
16         }
17     }

18     // Update button states
19     previousButton.interactable = page > 0;
20     nextButton.interactable = endIndex < playerInventory.Count;
21 }
```

Isječak koda 23: DisplayPage funkcija skripte `InventoryDisplayManager.cs`

Funkcija "DisplayPage(int page)" prihvati cjelobrojni tip podatka (engl. integer) koji određuje koja stranica inventara bi trebala biti otvorena. Taj integer pomnoži s maksimalnim brojem objekata na jednoj stranici (taj broj je konkretno postavljen na petnaest (15)). Na taj način se određuje raspon objekata iz inventara igrača koji će se prikazati. Nakon što je određeni raspon objekata prikazan te ostatak objekata sakriven funkcija određuje jesu li gumbi za listanje stranica inventara dostupni.

```
public class Item
{
    public string id;
    public string type;
    public string name;
    public string color;
    public int power;
    public string owner;
}
```

Isječak koda 24: Item klasa skripte `InventoryDisplayManager.cs`

Klasa "Item" služi za spremanje primljenih objekata iz JSON sadržaja u listu objekata. Spremljena lista objekata se zatim koristi za prikaz igračevog inventara unutar videoigre. Ova klasa je spremna primiti podatke vezane uz objekte mač ili dragulj. Za oba objekta potreban je string "id", string "type" i string "owner". Za objekt mač potreban je string "name". Za objekt dragulj potreban je string "color" koji određuje vrstu bonusa koje dragulj pruža te integer "power" koji određuje snagu tih bonusa. U nastavku slijedi funkcija "FetchWalletBalance()" koja je dio druge sekvence funkcija.

```
public IEnumerator FetchWalletBalance(string username)
{
    string url = $"http://{playerData.serverIP}:5000/get_wallet_balance/{username}";

    UnityWebRequest www = UnityWebRequest.Get(url);
    yield return www.SendWebRequest();

    if (www.result != UnityWebRequest.Result.Success)
    {
        Debug.LogError("Error fetching wallet balance: " + www.error);
    }
    else
    {
        string jsonResponse = www.downloadHandler.text;
        WalletResponse response =
        ↪  JsonUtility.FromJson<WalletResponse>(jsonResponse);
        goldText.text = response.balance.ToString();
    }
}
```

Isječak koda 25: FetchWalletBalance funkcija skripte `InventoryDisplayManager.cs`

Ova funkcija šalje GET zahtjev Flask serveru koji zatim odgovara na zahtjev JSON sadržajem to jest brojem koji predstavlja novčanu valutu koju igrač posjeduje. Funkcija primitkom JSON sadržaja ažurira zaslon igrača time što promijeni tekst UI elementa "goldText" u trenutnu količinu novčane valute koju igrač posjeduje.

Kao što je prethodno prikazano, unutar Start funckije skripte `InventoryDisplayManager.cs` dodaju se mnogi Listener-i na razne gumbe. Prva dva navedena gumba listaju stranice inven-

tara time što inkrementalno povećaju ili smanje varijablu "currentPage" te pozovu funkciju "DisplayPage()". Ostali gumbi pokreću funkcije koje također komuniciraju s Flask serverom. Prvi bitan gumb je naziva "socketButton" koji služi za ugrađivanje dragulja u mač. Njegova logika nije kompliciranа. Ako igrač klikne na objekt mač u svojem inventaru te zatim klikne na gumb "socketButton" - funkcija "OnSocketButtonClicked()" samo postavi boolean varijablu "isSocketing" na TRUE. Ako funkcija utvrdi da mač već sadrži tri (3) dragulja ništa se ne dogodi jer je maksimalno moguće ugraditi tri dragulja u jedan mač. No, kao što je navedeno, prvo je potrebno kliknuti na mač, zatim na gumb "socketButton" te na dragulj. To je zato što ugrađivanje dragulja u mač ostvaruje funkcija "StoreItemData" koja se poziva klikom na bilo koji objekt unutar inventara igrača. U nastavku je pregled navedene funkcije.

```
1  void StoreItemData(Item item)
2  {
3      if (isSocketing && item.type == "gem")
4      {
5          Gem gem = item as Gem;
6          if (gem != null && selectedSword != null)
7          {
8              Debug.Log($"Socketing gem with ID: {gem.id} into sword with ID:
                   {selectedSword.id}");
9              StartCoroutine(SocketGem(selectedSword, gem));
10         }
11         isSocketing = false;
12         socketButton.interactable = false; // Disable the socket button after
               socketing
13     }
14     ...
```

Isječak koda 26: StoreItemData funkcija skripte `InventoryDisplayManager.cs`, dio 1

Funkcija "StoreItemData(Item item)" najprije provjeri je li boolean varijabla "isSocketing" postavljena na TRUE. Ako jest i ako je igrač kliknuo na dragulj - pokreće se funkcija "SocketGem()" koja obavlja ugrađivanje odabranog dragulja u mač komunikacijom s Flask serverom.

```
...
    else
    {
        selectedItem.itemName = item.name;
        selectedItem.itemType = item.type;
        selectedItem.itemColor = item.color;
        selectedItem.itemPower = item.power;

        // Ensure the correct ID is set for gems
        if (item.type == "gem")
        {
            Gem gem = item as Gem;
            if (gem != null)
            {
                selectedItem.id = gem.id;
                int upgradeCost = CalculateUpgradeCost(gem.power);
                upgradeButtonText.text = $"Upgrade ({upgradeCost} gold)";
            }
        }
        else
        {
            selectedItem.id = item.id;
        }

        int sellPrice = CalculateSellPrice(item);
        sellButtonText.text = $"Sell ({sellPrice} gold)";

        if (item.type == "sword")
        {
            Sword sword = item as Sword;
            if (sword != null)
            {
                selectedItem.itemGems = sword.gems;
                selectedSword = sword;
                socketButton.interactable = true; // Enable the socket button
            }
        }
        else
        {
            selectedItem.itemGems = null;
            selectedSword = null;
            socketButton.interactable = false; // Disable the socket button
        }
        characterSelect.UpdateUI();
    }
}
```

Isječak koda 27: StoreItemData funkcija skripte `InventoryDisplayManager.cs`, dio 2

Ako je boolean varijabla "isSocketing" postavljena na FALSE, ova funkcija sprema podatke odabranog objekta te ažurira određene UI elemente. Ako je odabran dragulj, ažurira se cijena za koju bi se dragulj prodao, cijena za nadogradnju dragulja te popis karakteristika oda-

branog lika tako da se u popis doda bonus kojeg bi dragulj pružio. Ako je odabran mač, ažurira se cijena za koju bi se mač prodao te se ažurira popis karakteristika odabranog lika tako da se u popis dodaju bonusi koje bi mač pružio. Ako je mač odabran, također se omogući klik na gumb "socketButton". Funkcija "UpdateUI()" se nalazi u skripti `CharacterSelect.cs` te je prikazana u analizi te skripte. U nastavku slijedi analiza funkcije "SocketGem()".

```
IEnumerator SocketGem(Sword sword, Gem gem)
{
    string url = $"http://{playerData.serverIP}:5000/socket_gem";
    JObject requestBody = new JObject
    {
        { "sword_id", sword.id },
        { "gem_id", gem.id }
    };

    UnityWebRequest request = new UnityWebRequest(url, "POST");
    byte[] bodyRaw = System.Text.Encoding.UTF8.GetBytes(requestBody.ToString());
    request.uploadHandler = new UploadHandlerRaw(bodyRaw);
    request.downloadHandler = new DownloadHandlerBuffer();
    request.SetRequestHeader("Content-Type", "application/json");

    yield return request.SendWebRequest();

    if (request.result == UnityWebRequest.Result.ConnectionError || request.result
    ↪  == UnityWebRequest.Result.ProtocolError)
    {
        Debug.LogError(request.error);
    }
    else
    {
        string jsonResponse = request.downloadHandler.text;
        Debug.Log("Socket Gem Response: " + jsonResponse);
    }
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
}
```

Isječak koda 28: SocketGem funkcija skripte `InventoryDisplayManager.cs`

Funkcija "SocketGem(Sword sword, Gem gem)" šalje ID odabranog mača i dragulja na endpoint Flask poslužitelja čiji je naziv "socket_gem". Flask poslužitelj zatim odradi proces ugrađivanja dragulja u mač. Nakon što je taj proces uspješno ili neuspješno završio ponovno se učitava scena odabira karaktera kako bi se opet učitalo stanje inventara igrača. Moguće je implementirati elegantniju funkciju koja ne bi cijelu scenu učitavala ispočetka, no na ovaj način se prikazuje performansa udaljene objektno-orijentirane baze podataka.

Ako je igrač kliknuo na sljedeći bitan gumb, to jest na gumb "sellButton", pokreće se funkcija "SellItem()" čija je analiza u nastavku.

```
public IEnumerator SellItem(string playerName, string itemId)
{
    string url = $"http://{playerData.serverIP}:5000/sell_item";
    string jsonData = "{\"player_name\": \"" + playerName + "\", \"item_id\": \"" +
    ↪  itemId + "\"}";

    byte[] jsonDataBytes = System.Text.Encoding.UTF8.GetBytes(jsonData);
    using (UnityWebRequest www = new UnityWebRequest(url, "POST"))
    {
        www.uploadHandler = new UploadHandlerRaw(jsonDataBytes);
        www.downloadHandler = new DownloadHandlerBuffer();
        www.SetRequestHeader("Content-Type", "application/json");

        yield return www.SendWebRequest();

        if (www.result != UnityWebRequest.Result.Success)
        {
            Debug.LogError("Error selling item: " + www.error);
        }
        else
        {
            string jsonResponse = www.downloadHandler.text;
            SellItemResponse response =
            ↪  JsonUtility.FromJson<SellItemResponse>(jsonResponse);
            Debug.Log("Item sold successfully for " + response.sell_price + "G. New
            ↪  wallet balance: " + response.new_wallet_balance);
            // Update the UI to reflect the sold item and new player money
            SceneManager.LoadScene(SceneManager.GetActiveScene().name);
        }
    }
}
```

Isječak koda 29: SellItem funkcija skripte `InventoryDisplayManager.cs`

Funkcija "SellItem(string playerName, string itemId)" šalje POST zahtjev krajnjoj točci Flask poslužitelja naziva "sell_item". U POST zahtjevu je kao sadržaj uključeno korisničko ime igrača te ID objekta kojeg želi prodati. Na kraju, ako je proces uspješno završio, funkcija pokrene ponovno učitavanje scene odabira karaktera kako bi se ažurirao prikaz inventara igrača te prikaz količine njegove novčane valute.

Sljedeći bitan gumb je UI element naziva "tradeUpButton" koji pokreće funkciju "TryTradeUpGem()" čija je analiza u nastavku.

```
public IEnumerator TryTradeUpGem(SelectedItem selectedItem, string playerName)
{
    string url = $"http://{playerData.serverIP}:5000/try_trade_up_gem";
    TradeUpRequest requestPayload = new TradeUpRequest(selectedItem.id, playerName);
    string jsonData = JsonConvert.SerializeObject(requestPayload);

    using (UnityWebRequest www = new UnityWebRequest(url, "POST"))
    {
        www.uploadHandler = new
        ↪  UploadHandlerRaw(System.Text.Encoding.UTF8.GetBytes(jsonData));
        www.downloadHandler = new DownloadHandlerBuffer();
        www.SetRequestHeader("Content-Type", "application/json");

        yield return www.SendWebRequest();

        if (www.result != UnityWebRequest.Result.Success)
        {
            Debug.LogError("Error during trade-up: " + www.error);
        }
        else
        {
            Debug.Log("Response: " + www.downloadHandler.text);
            // Handle the response, update UI accordingly
            SceneManager.LoadScene(SceneManager.GetActiveScene().name);
        }
    }
}
```

Isječak koda 30: TryTradeUpGem funkcija skripte `InventoryDisplayManager.cs`

Funkcija "TryTradeUpGem(SelectedItem selectedItem, string playerName)" na endpoint Flask poslužitelja naziva "try_trade_up_gem" šalje POST zahtjev čiji sadržaj uključuje korisničko ime igrača i ID odabranog objekta. Ako je funkcija uspješno izvršena - scena odabira karaktera se ponovno učita.

Zadnji bitan gumb je UI element naziva "upgradeButton" koji pokreće funkciju "UpgradeGem()" čija je analiza u nastavku.

```
public IEnumerator UpgradeGem(SelectedItem selectedItem, string playerName)
{
    string url = $"http://{playerData.serverIP}:5000/upgrade_gem";
    UpgradeGemRequest requestPayload = new UpgradeGemRequest(selectedItem.id,
    ↪  playerName);
    string jsonData = JsonConvert.SerializeObject(requestPayload);

    using (UnityWebRequest www = new UnityWebRequest(url, "POST"))
    {
        www.uploadHandler = new
        ↪  UploadHandlerRaw(System.Text.Encoding.UTF8.GetBytes(jsonData));
        www.downloadHandler = new DownloadHandlerBuffer();
        www.SetRequestHeader("Content-Type", "application/json");

        yield return www.SendWebRequest();

        if (www.result != UnityWebRequest.Result.Success)
        {
            Debug.LogError("Error during gem upgrade: " + www.error);
        }
        else
        {
            Debug.Log("Response: " + www.downloadHandler.text);
            // Handle the response, update UI accordingly
            SceneManager.LoadScene(SceneManager.GetActiveScene().name);
        }
    }
}
```

Isječak koda 31: UpgradeGem funkcija skripte `InventoryDisplayManager.cs`

Funkcija "UpgradeGem(SelectedItem selectedItem, string playerName)" šalje POST zahtjev na endpoint "upgrade_gem" Flask poslužitelja. POST zahtjev uključuje ID odabranog dragulja te korisničko ime igrača. U analizi endpoint-a vidljivo je da se provede provjera ima li igrač dovoljno novčane valute za nadogradnju dragulja te se dragulj nadogradi a novčana valuta oduzme. Pri uspješno provedenom procesu funkcija poziva ponovno učitavanje trenutne scene kako bi se prikazani podatci ažurirali.

Ovdje završava pregled svih bitnih dijelova skripte `InventoryDisplayManager.cs`. U nastavku slijedi analiza bitnih dijelova skripte `CharacterSelect.cs`. Svrha ove analize je prikazati utjecaj ZODB-a i Scriptable Object-a kao objektno-orijentirane baze podataka na provođenje ostatka videoigre. Prva bitna funkcija te skripte je naziva "UpdateUI()" koja se pojavila i u skripti `InvenotryDisplayManager.cs`. U nastavku slijedi analiza te funkcije.

```
public void UpdateUI()
{
    if (selectedDifficulty == 0) DifficultyNameText.color = new Color(0, 0.6f, 0);
    else if (selectedDifficulty == 1) DifficultyNameText.color = new Color(0.6f,
    ↪  0.4f, 0);
    else if (selectedDifficulty == 2) DifficultyNameText.color = new Color(0.6f, 0,
    ↪  0);

    // Update stat numbers based on the selected class
    UpdateStatNumbers();
    // Update character image
    UpdateCharacterImage();
}
```

Isječak koda 32: UpdateUI funkcija skripte `CharacterSelect.cs`

Funkcija "UpdateUI()" ažurira boju teksta UI elementa "DifficultyNameText" koji predstavlja odabranu težinu igre. Zatim se pokreću funkcije "UpdateStatNumbers()" koja ažurira popis karakteristika odabranog lika i "UpdateCharacterImage()" koja ažurira sliku odabranog karaktera. Za ovaj rad je samo bitna funkcija "UpdateStatNumbers()" čija analiza slijedi u nastavku.

```
void UpdateStatNumbers()
{
    ClassDescription.text = ClassAbilityDescriptions[selectedClass];
    // Retrieve default stats for the selected class
    int[] stats = defaultStats[selectedClass];
    // Update text for each stat number
    for (int i = 0; i < StatNumber.Length; i++)
    {
        StatNumber[i].text = stats[i].ToString();
    }
    if (selectedItem.itemType == "gem")
    {
        switch (selectedItem.itemColor)
        {
            case "green":
                StatNumber[0].text += "+" + selectedItem.itemPower;
                break;
            ...
            default:
                Debug.LogError($"Unknown gem color: {selectedItem.itemColor}");
                break;
        }
    }
    else if (selectedItem.itemType == "sword")
    {
        int b0 = 0, b7 = 0, b1 = 0, b2 = 0, b4 = 0; //bonus ammount
        foreach (var gem in selectedItem.itemGems)
        {
            switch (gem.color)
            {
                case "green":
                    b0 += gem.power;
                    break;
                ...
                default:
                    Debug.LogError($"Unknown gem color: {gem.color}");
                    break;
            }
        }
        if (b0 > 0)
        {
            StatNumber[0].text += "+" + b0;
        }
        ...
        if (b7 > 0)
        {
            StatNumber[7].text += "+" + b7;
        }
    }
}
```

Isječak koda 33: UpdateStatNumbers funkcija skripte `CharacterSelect.cs`

Funkcija "UpdateStatNumbers()" ažurira podatke na popisu karakteristika odabranog lika i opis posebne karakteristike odabranog lika. Prvo se postave svi osnovni podatci lika a zatim se dodaju bonusi koje pruža odabrani objekt.

Nakon što je igrač zadovoljan sa svojim odabirom klikne gumb s tekstom "Start" koji pokreće funkciju "StartGame" čija analiza slijedi u nastavku.

```
1  public void StartGame()
2  {
3      // Set selected character and difficulty in GameData
4      gameData.selectedClass = selectedClass;
5      gameData.selectedDifficulty = selectedDifficulty;

6      if (selectedClass == 1) gameData.selectedDifficulty = 2;

7      // Load the gameplay scene
8      SceneManager.LoadScene(2);
9  }
```

Isječak koda 34: StartGame funkcija skripte `CharacterSelect.cs`

Funkcija "StartGame()" u Scriptable Object naziva "gameData" sprema odabranu vrstu/klasu karaktera, odabranu težinu igre te učitava Unity scenu "GameplayScene". Ostatak ove skripte je nebitan za ovaj rad stoga je u nastavku prikazana analiza bitnih dijelova skripte `GameManager.cs`.

3.6.2.3. Scena Gameplay

Skripta `GameManager.cs` koristi podatke svih Scriptable Object-a: "GameData", "PlayerData" i "SelectedItem". U ovoj sceni podatci i funkcije iz objektno-orijentiranih baza podataka kreiraju razne bitne efekte. Za skriptu `GameManager.cs` je također bitna skripta `Character.cs` koju nije potrebno detaljno analizirati iz razloga što je velik dio skripte nevezan uz ZODB. Stoga je prvo prikaz jedine bitne funkcije skripte `Character.cs` čiji je naziv "TakeDamage()".

```
public bool TakeDamage(int attackDamage, int abilityPower, int penetration, int cleave, int luck, int critDMG)
{
    int critDamageDone = 0;
    int luckRoll = Random.Range(0, 100);
    bool hitCrit = (luckRoll < luck);
    if (hitCrit) critDamageDone = (int)((float)attackDamage * ((float)critDMG / 100f));
    CritDMGtaken = critDamageDone;

    // Calculate penetration factor
    float penetrationFactor = 100 / Mathf.Clamp((100 + (((float)this.Armor * (1 - (CleaveCalculation(cleave) / 100f))) - (float)penetration)), 0.1f, 1000000f);

    // Calculate damage done
    int physDamageDone = (int)((1f + (float)ChainAttack / 10f) * (float)attackDamage * penetrationFactor);
    int abilityDamageDone = abilityPower;
    int totalDamageDone = physDamageDone + abilityDamageDone;

    // Update statistics
    if (isEnemy)
    {
        playerData.DMGdone += totalDamageDone;
        playerData.PhysDMG += physDamageDone;
        playerData.APDMG += abilityPower;
    }
    else
    {
        playerData.DMGtaken += totalDamageDone;
        playerData.DMGmitigated += (int)((1 + ChainAttack / 10f) * attackDamage * (1 - penetrationFactor));
    }

    // Apply damage to health
    this.CurrentHealth -= totalDamageDone;
    this.CurrentHealth -= critDamageDone; // crit damage is true damage

    // Update specific damage taken
    PhysDMGtaken = physDamageDone;
    APDMGtaken = abilityDamageDone;
    TotalDMGtaken = totalDamageDone;

    // Check if the enemy is killed
    if (this.CurrentHealth <= 0) return true;
    else return false;
}
```

Isječak koda 35: TakeDamage funkcija skripte `Character.cs`

Funkcija "TakeDamage(int attackDamage, int abilityPower, int penetration, int cleave, int

luck, int critDMG)" najprije izračuna količinu postignute štete pomoću karakteristika igračevog lika ili karakteristika neprijatelja te pomoću raznih mehanika igre koje su uklonjene u ovom prikazu kôda jer su nebitne. Ostvarena šteta na neprijatelju ili na igračevom liku se bilježi u Scriptable Object "PlayerData" kako bi se kasnije ti podatci mogli koristiti u statističke svrhe.

Nakon ove analize kôda moguće je bolje razumijevanje provođenje skripti `GameManager.cs` i `LeaderBoard.cs`. Stoga je u nastavku prikaz analize bitnih dijelova skripte `GameManager.cs`. Prva koja će se analizirati bitna funkcija te skripte je "Start()".

```
private void Start()
{
    selectedClass = gameData.selectedClass;
    selectedDifficulty = gameData.selectedDifficulty;
    playerData.ResetScore();
    switch (selectedClass)
    {
        case 0: playerData.className += "Tank "; break;
        case 1: playerData.className += "Hunter "; break;
        ...
    }
    switch (selectedDifficulty)
    {
        case 0: playerData.className += "E"; break;
        case 1: playerData.className += "M"; break;
        case 2: playerData.className += "H"; break;
    }
    // Instantiate the character based on the selected class
    character = new Character(defaultStats[selectedClass], playerData, false);
    enemyCharacter = RandomEnemyGenerate();

    // Update player stats based on the selected item
    if (selectedItem.itemType == "gem")
    {
        switch (selectedItem.itemColor)
        {
            case "green":
                character.MaxHealth += selectedItem.itemPower;
                break;
            ...
        }
    }
    else if (selectedItem.itemType == "sword")
    {
        foreach (var gem in selectedItem.itemGems)
        {
            switch (gem.color)
            {
                case "green":
                    character.MaxHealth += gem.power;
                    break;
                ...
            }
        }
    }
    // Update UI
    UpdateStats(false);
    UpdateStats(true);
}
```

Isječak koda 36: Start funkcija skripte `GameManager.cs`

Funkcija "Start()" iz Scriptable Object-a "GameData" pročita koju klasu karaktera je igrač odabrao te koju težinu igre je odabrao. Nakon što ažurira svoje varijable pomoću pročitanih podataka funkcija instancira igračevog karaktera. Pri instanciranju prosljeđuje Scriptable Object "PlayerData" samo kako bi konstruktor te klase zabilježio koja klasa karaktera je odabrana. Funkcija zatim generira protivnika čije karakteristike ovise o varijabli "enemyPowerLevel" koja se sve više skalira s obzirom na broj poraženih protivnika. Funkcija zatim pomoću Scriptable Object-a "SelectedItem" određuje koje bonuse će dobiti igračev karakter. Nakon što su bonusi ostvareni funkcija poziva ažuriranje popisa karakteristika igračevog karaktera i protivnika. Kada igrač doživi poraz ili odustane od igre klikom na gumb s tekstom "Give Up" poziva se funkcija "EndGame()" čija je analiza u nastavku.

```
1  public void EndGame()
2  {
3      Debug.Log("score before modifier = " + playerData.score);
4      switch (selectedDifficulty)
5      {
6          case 0:
7              playerData.score = (int)(playerData.score * 0.8);
8              break;
9          case 1:
10             playerData.score = (int)(playerData.score * 1);
11             break;
12         case 2:
13             playerData.score = (int)(playerData.score * 1.2);
14             break;
15     }
16     Debug.Log("score after modifier = " + playerData.score);
17     character.UpdatePlayerData();
18     SceneManager.LoadScene(3);
19 }
```

Isječak koda 37: EndGame funkcija skripte `GameManager.cs`

Funkcija "EndGame()" izračuna konačne bodove koje je igrač postigao s obzirom na odabranu razinu težine igre. Ostvareni bodovi se zabilježe u Scriptable Object "PlayerData". Zatim se poziva funkcija "UpdatePlayerData()" skripte `Character.cs`. Na kraju funkcije se učitava Unity scena "LeaderBoard". Prije prelaska na tu scenu slijedi analiza funkcije "UpdatePlayerData()" skripte `Character.cs`.

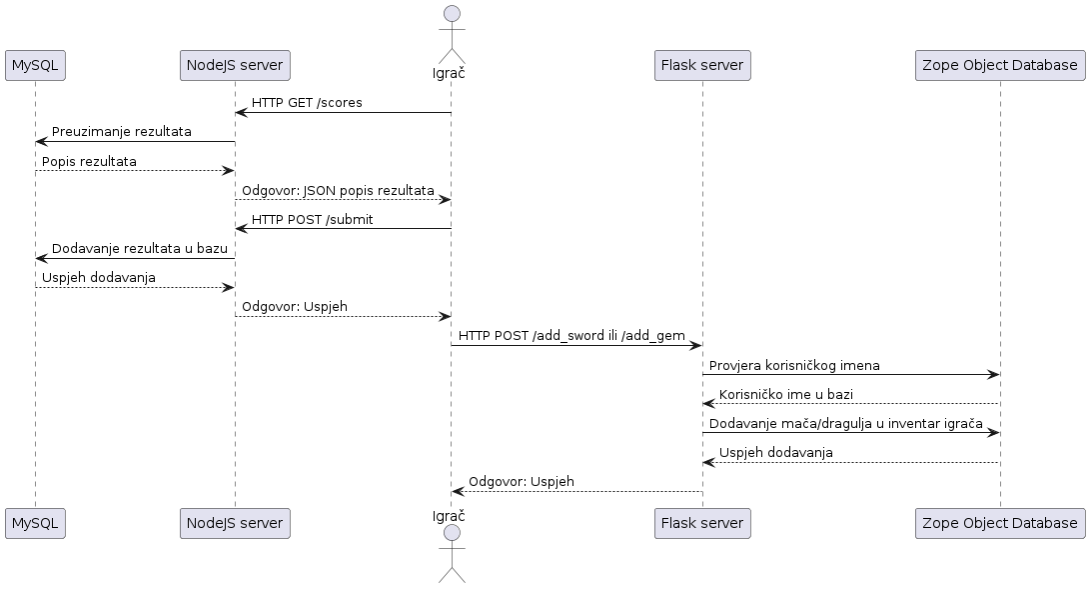
```
public void UpdatePlayerData()
{
    playerData.maxhp = this.MaxHealth;
    playerData.AD = this.AttackDamage;
    playerData.AP = this.AbilityPower;
    playerData.Armour = this.Armor;
    playerData.Pen = this.Penetration;
    playerData.Cleave = this.Cleave;
    playerData.CritP = this.Luck;
    playerData.CritDMG = this.CritDamage;
    playerData.Level = this.Level;
    return;
}
```

Isječak koda 38: UpdatePlayerData funkcija skripte `Character.cs`

Funkcija "UpdatePlayerData()" sprema sve konačne karakteristike koje su postignute na kraju igre. Ovi podatci se spremaju u Scriptable Object "PlayerData" kako bi se mogli objaviti u Unity sceni "LeaderBoard" čija je analiza u nastavku.

3.6.2.4. Scena LeaderBoard

Scena "LeaderBoard" koristi skriptu `LeaderBoard.cs` i podatke iz Scriptable Object-a "PlayerData". Prva bitna funkcija skripte `LeaderBoard.cs` je funkcija "Start()" koja provjerava ako je igrač postigao više od nula bodova te poziva funkciju "UpdatePlayerScore()". Osim toga se poziva i funkcija "DownloadLeaderboard()" kao korutina. Slijedi dijagram sekvenci koji prikazuje cijelokupan proces a zatim slijedi analiza funkcije "UpdatePlayerScore()".



Slika 10: Slika dijagrama sekvenci LeaderBoard scene

```
private void UpdatePlayerScore()
{
    playerStats[0].text = playerData.score.ToString();
    playerStats[1].text = playerData.maxhp.ToString();
    playerStats[2].text = playerData.AD.ToString();
    playerStats[3].text = playerData.AP.ToString();
    playerStats[4].text = playerData.Armour.ToString();
    playerStats[5].text = playerData.Pen.ToString();
    playerStats[6].text = playerData.Cleave.ToString();
    playerStats[7].text = playerData.CritP.ToString();
    playerStats[8].text = playerData.CritDMG.ToString();
    playerStats[9].text = playerData.Level.ToString();
    playerStats[10].text = playerData.DMGdone.ToString();
    playerStats[11].text = playerData.DMGmitigated.ToString();
    playerStats[12].text = playerData.DMGtaken.ToString();
    playerStats[13].text = playerData.PhysDMG.ToString();
    playerStats[14].text = playerData.APDMG.ToString();
    playerStats[15].text = playerData.className;
    return;
}
```

Isječak koda 39: UpdatePlayerScore funkcija skripte `LeaderBoard.cs`

Funkcija "UpdatePlayerScore()" čita statističke podatke zapisane u Scriptable Object-u "PlayerData" te ih prikazuje igraču pomoću UI elemenata koji su svrstani u polje "playerStats". U nastavku slijedi analiza funkcije "DownloadLeaderboard()".

```
IEnumerator DownloadLeaderboard()
{
    using (UnityWebRequest www = UnityWebRequest.Get(baseURL + "/scores"))
    {
        yield return www.SendWebRequest();
        if (www.result != UnityWebRequest.Result.Success)
        {
            Debug.LogError("Error downloading leaderboard: " + www.error);
        }
        else
        {
            string leaderboardData = www.downloadHandler.text;
            Debug.Log("Leaderboard Data: " + leaderboardData);
            LeaderboardDataWrapper dataWrapper =
            ↪  JsonUtility.FromJson<LeaderboardDataWrapper>("{\"entries\":" +
            ↪  leaderboardData + "}");
            if (dataWrapper != null && dataWrapper.entries != null &&
            ↪  dataWrapper.entries.Count > 0)
            {
                int rank = 1;
                foreach (HighScoreEntry highScoreEntry in dataWrapper.entries)
                {
                    CreateHighScoreEntryTransform(highScoreEntry, entryContainer,
                    ↪  highScoreEntryTransformList, rank);
                    rank++;
                }
                highScore = dataWrapper.entries[0].score;

                // Check if the player has beaten the high score after it has been
                ↪  set
                if (playerData.score > highScore)
                {
                    playerEarnedSword.SetActive(true);
                }
            }
        }
    }
}
```

Isječak koda 40: DownloadLeaderboard funkcija skripte `LeaderBoard.cs`

Funkcija "DownloadLeaderboard()" šalje GET zahtjev NodeJS poslužitelju na endpoint "scores" koji vraća rezultate ostalih igrača. Funkcija zatim sprema primljene podatke u varijablu "leaderboardData" kako bi funkcija "CreateHighScoreEntryTransform()" sastavila tablicu i prikazala igraču primljene rezultate. Funkcija provjerava koji je rekord ostvarenih bodova te sprema taj rekord u varijablu "highScore". Ukoliko je igrač oborio rekordni broj bodova funkcija ga nagrađuje s mačem.

Nakon što igrač imenuje svoj rezultat i opcionalno mač - potvrđuje svoj rezultat klikom na gumb s tekstom "Submit score!". Klik na taj gumb pokreće funkciju "ConfirmScore()" koja poziva funkciju "SubmitScore()" kao korutinu. U nastavku slijedi analiza funkcije "SubmitScore()".

```
IEnumerator SubmitScore()
{
    // Check if the playerData score is greater than 0 and submit it to the server
    if (playerData.score > 0)
    {
        string submitURL = baseURL + "/submit";
        // Construct JSON data with all variables
        string jsonData = "{\"name\": \"" + buildNameInputField.text + "\",
        ↪  \"score\": " + playerData.score +
            ", \"password\": \"" + playerData.password + "\", \"maxhp\": " +
            ↪  playerData.maxhp +
            ", \"AD\": " + playerData.AD + ", \"AP\": " + playerData.AP + ",
            ↪  \"Armour\": " +
            playerData.Armour + ", \"Pen\": " + playerData.Pen + ", \"Cleave\": "
            ↪  + playerData.Cleave +
            ", \"CritP\": " + playerData.CritP + ", \"CritDMG\": " +
            ↪  playerData.CritDMG +
            ", \"Level\": " + playerData.Level + ", \"DMGdone\": " +
            ↪  playerData.DMGdone +
            ", \"DMGmitigated\": " + playerData.DMGmitigated + ", \"DMGtaken\": "
            ↪  + playerData.DMGtaken +
            ", \"PhysDMG\": " + playerData.PhysDMG + ", \"APDMG\": " +
            ↪  playerData.APDMG + ", \"ClassName\": \"" + playerData.className +
            ↪  "\"}";
        byte[] jsonDataBytes = System.Text.Encoding.UTF8.GetBytes(jsonData);
        using (UnityWebRequest www = UnityWebRequest.PostWwwForm(submitURL, "POST"))
        {
            www.uploadHandler = new UploadHandlerRaw(jsonDataBytes);
            www.SetRequestHeader("Content-Type", "application/json");
            yield return www.SendWebRequest();
            if (www.result != UnityWebRequest.Result.Success)
            {
                Debug.LogError("Error submitting score: " + www.error);
            }
            else
            {
                Debug.Log("Score submitted successfully");
                yield return StartCoroutine(RewardPlayer());
                playerData.ResetScore();
                SceneManager.LoadScene(3);
            }
        }
    }
}
```

Isječak koda 41: SubmitScore funkcija skripte `LeaderBoard.cs`

Funkcija "SubmitScore()" još jednom provjerava ima li igrač više od nula (0) bodova. Funkcija zatim na endpoint "submit" NodeJS poslužitelja šalje POST zahtjev. POST zahtjev uključuje podatke potrebne za održavanje tablice rezultata u obliku JSON formata. Ako su rezultati uspješno poslani i spremljeni u MySQL bazu podataka pokreće se funckija "Reward-

Player()". Na kraju funkcije se obrišu podatci vezani uz rezultate igre pomoću metode "ResetScore()" Scriptable Object-a "PlayerData" te se Unity scena "LeaderBoard" ponovno učita kako bi igrač mogao vidjeti svoj rezultat zapisan u tablicu s ostalim rezultatima. U nastavku slijedi analiza funkcije "RewardPlayer()".

```
IEnumerator RewardPlayer()
{
    using (UnityWebRequest www = UnityWebRequest.Get(baseURL + "/scores"))
    {
        yield return www.SendWebRequest();

        if (www.result != UnityWebRequest.Result.Success)
        {
            Debug.LogError("Error downloading leaderboard: " + www.error);
        }
        else
        {
            string leaderboardData = www.downloadHandler.text;
            LeaderboardDataWrapper dataWrapper =
            ↪ JsonUtility.FromJson<LeaderboardDataWrapper>("{\"entries\":" +
            ↪ leaderboardData + "}");

            if (dataWrapper != null && dataWrapper.entries != null &&
            ↪ dataWrapper.entries.Count > 0)
            {
                if (playerData.score > highScore)
                {
                    // Reward the player with a sword
                    Debug.Log("Player has beaten the high score. Adding sword to
                    ↪ inventory.");
                    StartCoroutine(AddSwordToInventory(playerData.username,
                    ↪ swordNameInputField.text));
                }
                else if (playerData.score > highScore * 0.5f)
                {
                    // Reward the player with a gem
                    string gemColor;
                    switch (UnityEngine.Random.Range(0, 4))
                    {
                        case 0:
                            gemColor = "green";
                            break;
                        ...
                    };
                    StartCoroutine(AddGemToInventory(playerData.username, gemColor,
                    ↪ 1));
                }
            }
        }
    }
}
```

Isječak koda 42: RewardPlayer funkcija skripte `LeaderBoard.cs`

U slučaju da je igrač oborio rekord igre, funkcija "SubmitScore()" poziva funkciju "AddSwordToInventory()" kao korutinu. Kada igrač ostvari barem pola bodova trenutnog rekorda generira se broj od nula (0) do četiri (4) kako bi se nasumično odredila boja dragulja kojega je

igrač zaradio. Zatim se pokreće funkcija "AddGemToInventory()" kao korutina i prosljeđuje joj se nasumično određena boja dragulja. U nastavku slijedi analiza funkcije "AddSwordToInventory()".

```
IEnumerator AddSwordToInventory(string playerName, string swordName)
{
    string url = baseURLZODB + "/add_sword";
    string jsonData = "{\"name\": \"" + swordName + "\", \"owner\": \"" + playerName
    ↪  + "\"}";

    byte[] jsonDataBytes = System.Text.Encoding.UTF8.GetBytes(jsonData);
    using (UnityWebRequest www = UnityWebRequest.PostWwwForm(url, "POST"))
    {
        www.uploadHandler = new UploadHandlerRaw(jsonDataBytes);
        www.SetRequestHeader("Content-Type", "application/json");

        yield return www.SendWebRequest();

        if (www.result != UnityWebRequest.Result.Success)
        {
            Debug.LogError("Error adding sword to inventory: " + www.error);
            Debug.LogError("Response: " + www.downloadHandler.text);
        }
        else
        {
            Debug.Log("Sword added to inventory successfully");
            Debug.Log("Response: " + www.downloadHandler.text);
        }
    }
}
```

Isječak koda 43: AddSwordToInventory funkcija skripte `LeaderBoard.cs`

Funkcija "AddSwordToInventory(string playerName, string swordName)" šalje POST zahtjev na "add_sword" krajnju točku Flask poslužitelja. POST zahtjev uključuje korisničko ime igrača i ime mača kojeg mu je igrač dodijelio. U nastavku slijedi analiza funkcije "AddGemToInventory()".

```
IEnumerator AddGemToInventory(string playerName, string color, int power)
{
    string url = baseURLZODB + "/add_gem";
    string jsonData = "{\"owner\": \"" + playerName + "\", \"color\": \"" + color +
    ↪  "\", \"power\": " + power + "}";

    byte[] jsonDataBytes = System.Text.Encoding.UTF8.GetBytes(jsonData);
    using (UnityWebRequest www = UnityWebRequest.PostWwwForm(url, "POST"))
    {
        www.uploadHandler = new UploadHandlerRaw(jsonDataBytes);
        www.SetRequestHeader("Content-Type", "application/json");

        yield return www.SendWebRequest();

        if (www.result != UnityWebRequest.Result.Success)
        {
            Debug.LogError("Error adding gem to inventory: " + www.error);
            Debug.LogError("Response: " + www.downloadHandler.text);
        }
        else
        {
            Debug.Log("Gem added to inventory successfully");
            Debug.Log("Response: " + www.downloadHandler.text);
        }
    }
}
```

Isječak koda 44: FetchWalletBalance funkcija skripte `LeaderBoard.cs`

Funkcija "AddGemToInventory(string playerName, string color, int power)" šalje POST zahtjev na "add_gem" krajnju točku Flask poslužitelja. U POST zahtjevu je uključeno korisničko ime igrača, boja dragulja koja je određena nasumično generiranim brojem te snaga dragulja.

3.7. Povezivanje videoigre s bazama podataka

Odabrana objektno-orijentirana baza podataka za ovaj rad je Zope Object Database (ZODB). ZODB je programiran u programskom jeziku Python dok je videoigra programirana u programskom jeziku C#. Kako bi videoigra mogla koristiti navedenu bazu podataka kreirani su ZEO i Flask serveri.

ZEO pruža većem broju procesa pristup u ZODB. Flask poslužitelj pristupa ZODB-u pomoću ZEO poslužitelja te obavlja brojne funkcije koje su dostupne videoigri pomoću programiranih endpoint-a. Endpoint-i Flask poslužitelja prihvaćaju GET i POST zahtjeve čiji sadržaj dolazi u obliku JSON formata. Flask poslužitelj videoigri također šalje sadržaje u JSON formatu. Stoga je u Flask serveru potrebno uvesti "jsonify" iz "Flask" biblioteke dok je u videoigri korišten Unity-ev "JsonUtility". To je potrebno kako bi se mogle sastavljati i čitati poruke JSON formata. U prethodnom poglavlju "Prolazak kroz kôd videoigre" vidljiv je proces sastavljanja i čitanja poruka JSON formata unutar videoigre te se u ovom poglavlju nalaze analize tog pro-

cesa unutar Flask poslužitelja. U nastavku je prikazana analiza svih dostupnih endpoint-a Flask poslužitelja koji su korišteni u videoigri.

```
@app.route('/add_player/<username>', methods=['POST'])
def add_player(username):
    _, connection, root = connection_pool.get_connection()
    if username in root:
        return jsonify({"error": "Username already exists"}), 400
    player = Player(username)
    root[username] = player
    try:
        transaction.commit()
    except Exception as e:
        return jsonify({"error": "Failed to add player"}), 500
    return jsonify({"message": "Player added successfully"}), 200
```

Isječak koda 45: `add_player` funkcija Flask poslužitelja

Funkcija "add_player(username)" prima POST zahtjev sa zadanim korisničkim imenom. Provjerava postoji li već korisničko ime u bazi podataka. Ako postoji, vraća poruku o grešci. Ako ne postoji, dodaje novog igrača i potvrđuje transakciju. Uspješni rezultat vraća poruku broja 200.

```
@app.route('/add_gem', methods=['POST'])
def add_gem():
    data = request.get_json()
    _, connection, root = connection_pool.get_connection()
    if 'color' in data and 'power' in data and 'owner' in data:
        gem = Gem(data['color'], data['power'], data['owner'])
        player = root.get(data['owner'])
        if player:
            player.add_to_inventory(gem)
            root[gem.id] = gem
            try:
                transaction.commit()
                return jsonify(success=True, id=gem.id)
            except Exception as e:
                return jsonify(error='Failed to add gem'), 500
        else:
            return jsonify(error='Player not found'), 404
    else:
        return jsonify(error='Invalid request'), 400
```

Isječak koda 46: `add_gem` funkcija Flask poslužitelja

Funkcija "add_gem()" prima POST zahtjev s JSON podatcima koji sadrže boju, snagu i vlasnika dragulja. Provjerava valjanost podataka i dodaje dragulj u inventar igrača te bazu podataka. Ako transakcija uspije, vraća se status uspjeha. Ako igrač nije pronađen ili podatci nisu valjani, vraća se odgovarajuća poruka o grešci.

```python
@app.route('/add_sword', methods=['POST'])
def add_sword():
    data = request.get_json()
    _, connection, root = connection_pool.get_connection()
    if 'name' in data and 'owner' in data:
        sword = Sword(data['name'], data['owner'])
        player = root.get(data['owner'])
        if player:
            player.add_to_inventory(sword)
            root[sword.id] = sword
            try:
                transaction.commit()
                return jsonify(success=True, id=sword.id)
            except Exception as e:
                return jsonify(error='Failed to add sword'), 500
        else:
            return jsonify(error='Player not found'), 404
    else:
        return jsonify(error='Invalid request'), 400
```

Isječak koda 47: `add_sword` funkcija Flask poslužitelja

Funkcija "add_sword()" prima POST zahtjev s JSON podatcima koji sadrže ime i vlasnika mača. Provjerava valjanost podataka i dodaje mač u inventar igrača te bazu podataka. Ako transakcija uspije, vraća se status uspjeha. Ako igrač nije pronađen ili podatci nisu valjani, vraća se odgovarajuća poruka o grešci.

```python
@app.route('/get_inventory/<username>', methods=['GET'])
def get_inventory(username):
    _, connection, root = connection_pool.get_connection()
    player = root.get(username)
    if player:
        inventory = []
        for item in player.inventory:
            if isinstance(item, Sword):
                gems_info = [{'id': gem['gem'].id, 'color': gem['gem'].color,
                ↪  'power': gem['gem'].power, 'x': gem['x'], 'y': gem['y']} for gem
                ↪  in item.gems]
                inventory.append({'type': 'sword', 'id': item.id, 'name': item.name,
                ↪  'owner': item.owner, 'gems': gems_info})
            elif isinstance(item, Gem):
                inventory.append({'type': 'gem', 'id': item.id, 'color': item.color,
                ↪  'power': item.power, 'owner': item.owner})
        return jsonify(inventory)
    else:
        return jsonify({'error': 'Player not found'}), 404
```

Isječak koda 48: `get_inventory` funkcija Flask poslužitelja

Funkcija "get_inventory(username)" prima GET zahtjev sa zadanim korisničkim ime-

nom. Dohvaća inventar igrača iz baze podataka i vraća popis svih predmeta u JSON formatu. Ako igrač nije pronađen, vraća se odgovarajuća poruka o grešci.

```python
@app.route('/socket_gem', methods=['POST'])
def socket_gem():
    data = request.get_json()
    _, connection, root = connection_pool.get_connection()
    sword_id = data.get('sword_id')
    gem_id = data.get('gem_id')
    x = data.get('x', 0)
    y = data.get('y', 0)

    sword = root.get(sword_id)
    gem = root.get(gem_id)

    if sword and gem and isinstance(sword, Sword) and isinstance(gem, Gem):
        player = root.get(sword.owner)
        if player and gem in player.inventory:
            try:
                sword.socket_gem(gem, player, x, y)
                transaction.commit()
                return jsonify({'success': True})
            except Exception as e:
                return jsonify({'error': 'Failed to socket gem'}), 500
        else:
            return jsonify({'error': 'Player or gem not found in inventory'}), 404
    else:
        return jsonify({'error': 'Invalid sword or gem ID'}), 400
```

Isječak koda 49: `socket_gem` funkcija Flask poslužitelja

Funkcija "socket_gem()" prima POST zahtjev s JSON podatcima koji sadrže ID mača, ID dragulja te koordinate umetanja. Provjerava valjanost ID-ova i postojanje predmeta u inventaru igrača te umeće dragulj u mač. Ako transakcija uspije, vraća se status uspjeha. Ako predmeti nisu pronađeni ili su podatci neispravni, vraća se odgovarajuća poruka o grešci. Kao što je vidljivo u prethodnom poglavlju, pozicija dragulja nije iskorištena u implementaciji.

3.8. Testiranje videoigre i baza podataka

Videoigru su testirale razne osobe. Testiranja su bila opservirana kako bi se pronašle mane videoigre i objektno-orijentirane baze podataka ZODB. U ovom poglavlju se nalaze problemi otkriveni testiranjem. Uključeni su samo problemi vezani uz dizajn, mrežu i ZODB.

- U Unity sceni "Login" igrači su htjeli pomoću tipke "TAB" prelaziti u druga polja unosa.
- U slučaju da su igrači htjeli registirati novi račun, prvo bi upisali podatke a zatim kliknuli gumb s tekstom "Register" što bi rezultiralo brisanjem unesenih podataka

te pojavom polja za unošenje e-mail adrese. Taj problem bi frustrirao igrače zbog čega je tekst gumba promijenjen u "Open Registration". Nakon toga bi pojedini igrači napravili istu grešku ali ne bi bili frustrirani jer nisu smatrali da je kriv dizajn korisničkog sučelja.

- Nakon što je razvijena funkcija unosa IP adrese poslužitelja videoigre, igrači nisu mogli poslati POST i GET zahtjeve jer se ti zahtjevi šalju preko HTTP protokola. Problem je u tome što je HTTP protokol po defaultu zabranjen unutar Unity projekta. Stoga je potrebno prije izgradnje/kompilacije videoigre otvoriti "Player settings...", tab "Player", zatim karticu "WebGL settings" te postaviti "Allow downloads over HTTP" na "Always allowed". Ako je sigurnost videoigre korektno implementirana - ovaj problem se ne bi trebao pojaviti. Naročito ako se koristi HTTPS umjesto HTTP protokola.

- U Unity sceni "CharacterSelect" pojavio se problem performanse ZODB poslužitelja ZEO. Videoigra bi prilikom učitavanja navedene scene poslala dva GET zahtjeva na Flask poslužitelj istovremeno. Flask poslužitelj ponekad ne bi uspio ostvariti dobar rezultat te bi vratio poruku greške koja tvrdi da igrač nije pronađen u bazi podataka. Ta greška bi rezultirala time da igrač ne vidi objekte unutar svojeg inventara. Rješenje je pričekati da prvi GET zahtjev bude do kraja procesiran te zatim poslati drugi GET zahtjev.

3.9. Provedena istraživanja za razvoj objektno-orijentirane baze podataka

Razvoj videoigara danas zahtijeva složene i učinkovite sustave za upravljanje podatcima. Tradicionalne relacijske baze podataka često se koriste za te svrhe, no s porastom kompleksnosti igara, sve se više nameće potreba za fleksibilnijim rješenjima. Objektno orijentirane baze podataka nude niz prednosti koje mogu značajno unaprijediti razvoj i performanse videoigara. Ovo poglavlje istražuje razvoj i implementaciju OOBP-a u kontekstu videoigara, te analizira prednosti i izazove koji prate ovu tehnologiju.

Objektno orijentirane baze podataka pohranjuju podatke u obliku objekata, slično kao i objektno-orijentirani programski jezici. Za razliku od relacijskih baza koje koriste tablice za pohranu podataka, OOBP koristi objekte koji mogu direktno predstavljati entitete i njihove međusobne odnose. Prema Kim [1], "objektno-orijentirane baze podataka pružaju značajnu fleksibilnost u upravljanju složenim strukturama podataka".

Jedna od ključnih prednosti OOBP-a je njihova sposobnost da prirodno mapiraju složene objekte i njihove odnose. U kontekstu videoigre, ovo omogućuje direktno pohranjivanje i upravljanje objektima igre kao što su likovi, predmeti i okruženja bez potrebe za dodatnim slojem mapiranja između objekata i relacijskih tablica. Kako navode Atkinson et al. [2], "objektno-orijentirani pristup omogućava direktnu reprezentaciju stvarnog svijeta kroz objekte".

OOBP-i su optimizirani za rad s velikim količinama složenih objekata i njihovih međusob-

nih odnosa. To može rezultirati značajnim poboljšanjem performansi u igrama koje zahtijevaju brze i učestale pristupe podatcima. Cattell i Barry [3] ističu da "OOBP-i često pružaju superiorne performanse za aplikacije koje zahtijevaju manipulaciju složenim podatcima". U ovom radu su se doduše pojavili problemi performanse ZODB. No uz optimizaciju kôda videoigre je ostvarena konzistentnija performanca.

OOBP-i omogućuju lakšu prilagodbu i proširivanje modela podataka. Promjene u strukturi objekata mogu se jednostavno implementirati bez potrebe za složenim migracijama baza podataka. Ovo je posebno važno u iterativnom razvoju igara gdje se zahtjevi i dizajn često mijenjaju. Prema Bertino i Martino [4], "fleksibilnost OOBP-a omogućava brzu prilagodbu na promjene u zahtjevima aplikacije".

Iako OOBP-i nude brojne prednosti, njihova implementacija dolazi s određenim izazovima. Modeliranje podataka u OOBP-u može biti složenije u usporedbi s relacijskim bazama. Potrebno je dodatno obrazovanje i prilagodba kako bi se maksimalno iskoristile prednosti objektno-orijentiranog pristupa. Kako navodi Stonebraker [5], "prijelaz na objektno-orijentirani model zahtijeva značajnu promjenu u razmišljanju i pristupu dizajnu baze podataka".

Iako je podrška za OOBP-e u porastu, još uvijek postoji manje alata i biblioteka u usporedbi s relacijskim bazama podataka. To može ograničiti izbor razvojnih alata i povećati vrijeme potrebno za razvoj specifičnih funkcionalnosti. Prema Dittrich i Dayal [6], "nedostatak standardizacije i alata za OOBP-e predstavlja značajnu prepreku širokoj adopciji".

Integracija OOBP-a s postojećim sustavima koji koriste relacijske baze može biti izazovna. Potrebni su pažljivo planiranje i implementacija kako bi se osigurala kompatibilnost i održivost sustava. Stonebraker i Kemnitz [7] navode da "integracija OOBP-a s relacijskim sustavima može zahtijevati kompleksna rješenja i kompromise". U ovom radu je ostvarena uspješna kolaboracija MySQL-a i ZODB-a bez posebnih problema isključujući sigurnosne probleme.

Implementacija ZODB-a rezultirala je bržim vremenom odziva za složene upite i operacije nad objektima u igri. Također je omogućila jednostavniju prilagodbu modela podataka tijekom razvoja igre, smanjujući vrijeme potrebno za implementaciju novih značajki. Najznačajniji utjecaj je ostvaren prilikom implementacije učitavanja inventara igrača. Pomoću ZODB-a bilo je jednostavnije napraviti algoritam koji učitava ugrađene dragulje mača. Također je bilo jednostavnije kreirati funkciju ugrađivanja dragulja u mač.

4. Zaključak

Objektno-orijentirane baze podataka predstavljaju moćno rješenje za upravljanje složenim podatcima u videoigrama. Njihova sposobnost prirodnog mapiranja objekata, poboljšane performanse i fleksibilnost čine ih privlačnim izborom za razvojne timove. U kontekstu roguelike igara, ove prednosti postaju još izraženije zbog prirode žanra koji zahtijeva dinamičko upravljanje velikim brojem složenih entiteta i njihovih međusobnih odnosa.

Jedna od ključnih prednosti OOBP-a je njihova mogućnost da se prilagode specifičnim zahtjevima igara bez potrebe za dodatnim slojevima mapiranja podataka. To omogućuje bržu i jednostavniju manipulaciju objektima unutar igre, što može rezultirati značajnim poboljšanjem performansi. Primjerice, u ovom radu, prednost OOBP-a je očita u implementaciji ugrađivanja dragulja u mač. Dragulj se jednostavno dodaje u listu objekata (dragulja) koja je dio objekta "mač", što značajno pojednostavljuje proces i poboljšava efikasnost.

Nadalje, OOBP-i su često bolje prilagođeni horizontalnoj skalabilnosti, što je ključno za online igre s velikim brojem korisnika. Horizontalna skalabilnost omogućuje dodavanje novih poslužitelja kako bi se nosili s povećanim opterećenjem, što rezultira boljom raspodjelom resursa i povećanom dostupnošću igre za korisnike. Ova sposobnost je posebno važna u kontekstu roguelike igara koje često zahtijevaju real-time interakciju i brze odgovore na akcije igrača.

Unatoč prednostima, postoje i određeni izazovi u implementaciji OOBP-a. Jedan od glavnih izazova je složenost modeliranja podataka i potreba za specifičnim znanjem o objektno-orijentiranom pristupu. Također, postoje pitanja vezana uz integraciju s postojećim sustavima i alatima koji su često optimizirani za relacijske baze podataka. Međutim, ove prepreke se mogu prevladati kroz odgovarajuću edukaciju i prilagodbu razvojnih procesa.

Kroz ovaj rad, istražena je primjena OOBP-a u razvoju roguelike igre, s fokusom na specifične primjere modeliranja entiteta i njihove manipulacije. Rezultati pokazuju da korištenje OOBP-a može značajno unaprijediti razvojni proces i poboljšati performanse igre. Osim toga, istraživanje je pokazalo da OOBP-i pružaju veću fleksibilnost u razvoju igara, omogućujući lakšu prilagodbu promjenama u dizajnu i logici igre.

Zaključno, objektno-orijentirane baze podataka predstavljaju ključan alat za optimizaciju razvoja modernih roguelike igara. Njihova sposobnost efikasnog upravljanja složenim entitetima i podrška za horizontalnu skalabilnost čine ih idealnim izborom za razvojne timove koji žele iskoristiti prednosti naprednih tehnologija. Buduće istraživanje u ovom području može dodatno unaprijediti razumijevanje i primjenu OOBP-a, otvarajući nove mogućnosti za inovacije u gaming industriji.

Popis literature

[1] W. Kim, *Introduction to Object-Oriented Databases*. The MIT Press, 1990.

[2] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier i S. Zdonik, „The Object-Oriented Database System Manifesto," *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, 1989.

[3] R. G. G. Cattell i D. K. Barry, *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Addison-Wesley, 1997.

[4] E. Bertino i L. Martino, *Object-Oriented Database Management Systems: Concepts and Issues*. Springer-Verlag, 1993.

[5] M. Stonebraker, *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann, 1990.

[6] K. R. Dittrich i U. Dayal, „Object-Oriented Database Systems: Promises, Reality, and Future," *ACM Computing Surveys*, 1986.

[7] M. Stonebraker i G. Kemnitz, „The POSTGRES Next-Generation Database Management System," *Communications of the ACM*, 1991.

Popis slika

Popis isječaka koda