

# Projekat iz Verifikacije Softvera

Vuk Stefanović  
Indeks: 1036/2024

18. februar 2025.

## Sadržaj

<b>1</b>	<b>Uvod</b>	<b>3</b>
<b>2</b>	<b>Cppcheck</b>	<b>3</b>
2.1	Analiza . . . . .	3
2.1.1	Problemi sa konstruktorima i destruktorima . . . . .	4
2.1.2	Problemi sa memorijom i sigurnost pokazivača . . . . .	4
2.1.3	Problemi sa tipovima i kastovanjem podataka . . . . .	4
2.1.4	Optimizacija performansi i efikasnost koda . . . . .	4
2.2	Zaključak . . . . .	5
<b>3</b>	<b>Clang-Tidy</b>	<b>5</b>
<b>4</b>	<b>Analiza rezultata</b>	<b>5</b>
4.1	Pokretanje Clang-Tidy alata . . . . .	5
4.2	Otkriveni problemi i preporučene optimizacije . . . . .	6
4.2.1	Korišćenje <code>nullptr</code> umesto <code>NULL</code> . . . . .	6
4.2.2	Automatsko određivanje tipa pomoću <code>auto</code> . . . . .	6
4.2.3	Korišćenje <code>range-based for</code> petlji . . . . .	7
4.2.4	Korišćenje <code>emplace_back</code> umesto <code>push_back</code> . . . . .	7
4.2.5	Korišćenje <code>using</code> umesto <code>typedef</code> . . . . .	7
<b>5</b>	<b>Doxygen</b>	<b>8</b>
<b>6</b>	<b>Analiza rezultata</b>	<b>8</b>
6.1	Pokretanje Doxygen-a . . . . .	8
6.2	Opis parametara u Doxyfile-u . . . . .	9
6.3	Dodavanje komentara u kod . . . . .	11

6.3.1	Primer za klase u .hpp fajlovima . . . . .	11
6.3.2	Primer za .cpp implementaciju klasa . . . . .	11
<b>7</b>	<b>Valgrind</b>	<b>11</b>
7.1	Memcheck . . . . .	12
7.1.1	Analiza rezultata . . . . .	12
7.2	Callgrind . . . . .	13
7.2.1	Analiza rezultata . . . . .	14
7.2.2	Funkcije sa najvećom potrošnjom instrukcija . . . . .	14
7.2.3	Najčešće pozivane funkcije . . . . .	14
7.2.4	Moguće optimizacije . . . . .	15
7.2.5	Vizuelna analiza sa KCacheGrind . . . . .	15
<b>8</b>	<b>Zaključak</b>	<b>15</b>

# 1 Uvod

Verifikacija softvera predstavlja ključni deo razvoja aplikacija, osiguravajući njihovu ispravnost, efikasnost i sigurnost. U okviru ovog projekta analiziran je kod igre *The Legend of Zelda*, koja je implementirana u programskom jeziku C++. Ova igra je jedna od najpoznatijih akciono-avanturističkih franšiza, gde igrač preuzima ulogu Linka, istražuje svet, rešava zagonetke i bori se protiv neprijatelja kako bi spasio princezu Zeldu.

U cilju provere ispravnosti i optimizacije koda, korišćeni su alati kao što su Cppcheck, Clang-Tidy, Doxygen i Valgrind. Pomoću ovih alata analizirana je statička i dinamička struktura koda, otkrivene su potencijalne greške, analizirana memorijska potrošnja i generisana automatska dokumentacija. Korišćenjem statičke analize omogućeno je rano otkrivanje sintaksičkih i logičkih grešaka, dok je dinamičkom analizom procenjena efikasnost izvršavanja programa i potencijalni problemi sa memorijom.

Cilj ovog projekta je da se kroz sistematsku verifikaciju obezbedi stabilnost i optimizacija koda igre, što omogućava bolje performanse i sigurnije iskustvo za korisnike.

## 2 Cppcheck

Cppcheck je moćan alat za analizu koda namenjen isključivo C i C++ programima, sa ciljem otkrivanja različitih vrsta grešaka, problema sa memorijom i potencijalnih bezbednosnih propusta. Njegova ključna prednost je mogućnost analize bez potrebe za kompajliranjem koda, čime omogućava rano prepoznavanje problema pre izvršavanja programa.

Alat identifikuje sintaksičke i logičke greške, neoptimalne konstrukcije i nedostatke u upravljanju resursima, što olakšava poboljšanje efikasnosti i stabilnosti aplikacija. Može se koristiti kroz komandnu liniju, ali je takođe kompatibilan sa različitim razvojnim okruženjima. Cppcheck pruža fleksibilne opcije izveštavanja, omogućavajući programerima dublju analizu koda i otkrivanje skrivenih problema. S obzirom na to da je open-source, često se koristi kako u industriji, tako i u akademskim krugovima, gde doprinosi unapređenju kvaliteta softvera.

### 2.1 Analiza

Alat Cppcheck pokrenut je direktno iz komandne linije nakon pozicioniranja u direktorijum `cmake-build-debug`. Pokretanje je izvršeno sledećom komandom:

```
cppcheck --project=compile_commands.json --enable=all --std=c++20 -
inconclusive --report-progress &> cpp_check_res.txt
```

Rezultati analize su sačuvani u fajlu `cpp_check_res.txt`.

### 2.1.1 Problemi sa konstruktorima i destruktorkama

Primećeno je da neki konstruktori nisu eksplicitno definisani, što može dovesti do neželjenih implicitnih konverzija. Na primer, u fajlu `animation.hpp`, linija 7, konstruktor nije označen kao `explicit`, što može izazvati nepredvidene probleme tokom inicijalizacije objekata. Preporučuje se dodavanje `explicit` ključne reči kako bi se sprečile implicitne konverzije.

Dodatno, u fajlu `drawable.hpp`, linija 8, klasa koja sadrži virtuelne metode nema deklarisan virtuelni destruktorka, što može izazvati curenje memorije pri oslobađanju objekata bazne klase. Preporučuje se dodavanje virtuelnog destruktorka.

### 2.1.2 Problemi sa memorijom i sigurnost pokazivača

Detektovana su potencijalna curenja memorije. Konkretno, u fajlu `tmx_map.cpp`, linija 26, postoji dinamički alocirana memorija koja nije adekvatno oslobođena. Preporučuje se korišćenje `std::unique_ptr` ili osiguravanje ručnog oslobađanja memorije pomoću `delete`.

U fajlu `stb_image_aug.c`, detektovan je problem *use-after-free*, gde se memorija koristi nakon što je već oslobođena. Kao rešenje, preporučuje se postavljanje pokazivača na `nullptr` nakon što je memorija oslobođena, kako bi se sprečio nenamerni pristup nevalidnim memorijskim adresama.

### 2.1.3 Problemi sa tipovima i kastovanjem podataka

U fajlu `animation_hitbox.cpp`, linija 22, uočena je nebezbedna konverzija pokazivača pomoću C-style kastovanja. Preporučuje se korišćenje `dynamic_cast` za polimorfizam, jer omogućava dodatne provere i minimizuje rizik od neispravnog kastovanja.

### 2.1.4 Optimizacija performansi i efikasnost koda

Jedan od uočenih problema u analizi je neefikasno upravljanje memorijom pri korišćenju `std::vector`. Konkretno, u fajlu `sprite_set.cpp`, linija 11, vektori se dinamički proširuju pomoću `push_back()`, što može dovesti do čestih realokacija memorije i usporiti izvršavanje koda. Preporučuje se

korišćenje `reserve()` kako bi se unapred alocirao potreban prostor i smanjio broj realokacija.

## 2.2 Zaključak

Cppcheck analiza je omogućila detaljan uvid u potencijalne probleme koda i predložila načine za njihovo rešavanje. Otkrivene su kritične greške kao što su curenje memorije, *use-after-free* i nebezbedna manipulacija pokazivačima, koje bi trebalo ispraviti kako bi se osigurala stabilnost programa. Pored toga, prepoznati su problemi sa kodnim stilom i optimizacijom koji mogu poboljšati efikasnost rada programa.

## 3 Clang-Tidy

**Clang-Tidy** je alat za statičku analizu koda koji je sastavni deo Clang ekosistema i namenjen je unapređenju kvaliteta C i C++ koda. Njegova osnovna svrha je otkrivanje grešaka, analiza strukture koda i primena preporučenih praksi za poboljšanje čitljivosti i optimizacije.

Alat omogućava prepoznavanje sintaksičkih i logičkih grešaka pre faze kompajliranja, pomažući u ranom otkrivanju potencijalnih problema. Takođe, Clang-Tidy nudi podršku za automatizovano formatiranje koda prema definisanim stilskim pravilima, osiguravajući doslednost u kodnoj bazi.

Još jedna značajna funkcionalnost je mogućnost automatskog ispravljanja određenih tipova problema, kao što su optimizacija izraza, dodavanje nedostajućih zaglavlja i poboljšanje kompatibilnosti sa modernim standardima jezika. Clang-Tidy se može koristiti iz komandne linije ili integrisati u razvojna okruženja, što ga čini praktičnim alatom za kontinuiranu analizu koda i održavanje visokih standarda programiranja.

## 4 Analiza rezultata

### 4.1 Pokretanje Clang-Tidy alata

Alat Clang-Tidy pokrenut je iz komandne linije nakon pozicioniranja u direktorijum `cmake-build-debug`. Analiza je izvršena sledećom komandom:

```
run-clang-tidy -checks='clang-diagnostic-*, clang-analyzer-*,
modernize-use-auto, modernize-use-nullptr, modernize-use-noexcept,
modernize-use-emplace, modernize-use-emplace-back, modernize-loop-convert,
modernize-use-using' .. &> clang-tidy-report.txt
```

Rezultati analize su sačuvani u fajlu `clang-tidy-report.txt`, omogućavajući dalju obradu i analizu detektovanih problema.

## 4.2 Otkriveni problemi i preporučene optimizacije

U nastavku su predstavljeni najznačajniji nalazi Clang-Tidy analize, zajedno sa preporučenim rešenjima.

### 4.2.1 Korišćenje `nullptr` umesto `NULL`

Moderni C++ standardi preporučuju upotrebu `nullptr` umesto `NULL` kako bi se izbegli problemi sa implicitnim konverzijama i poboljšala sigurnost koda.

**Detektovani problem** Prilikom analize koda u fajlu `AudioDevice.cpp` (linija **36**), primećeno je korišćenje zastarelog `NULL` pokazivača:

```
ALCdevice* audioDevice = NULL;
```

**Preporučeno rešenje** Zamenom `NULL` sa `nullptr`, poboljšava se jasnoća koda i osigurava bolja kompatibilnost sa modernim C++ standardima.

```
ALCdevice* audioDevice = nullptr;
```

### 4.2.2 Automatsko određivanje tipa pomoću `auto`

Uočeno je ponavljanje tipa prilikom inicijalizacije promenljive pomoću `new` u fajlu `guard.cpp` na liniji **38**. Korišćenje ključne reči `auto` omogućava kompaktniji i lakše održiv kod.

**Detektovani problem** Eksplicitno navođenje tipa može otežati kasnije izmene:

```
AnimationHitbox* hitbox = new AnimationHitbox(position_,  
current_action- CurrentAnimation());
```

**Preporučeno rešenje** Korišćenjem `auto`, kod postaje kompaktniji i fleksibilniji:

```
auto hitbox = new AnimationHitbox(position_,  
current_action- CurrentAnimation());
```

### 4.2.3 Korišćenje range-based for petlji

U fajlu **quadtree.cpp** (linija 27) primećeno je eksplicitno indeksiranje u for petlji, koje može biti zamagljeno i manje čitljivo.

**Detektovani problem** Tradicionalna iteracija kroz niz:

```
for(int i = 0; i < 4; ++i) {  
    // Obrada children_ elemenata  
}
```

**Preporučeno rešenje** Korišćenjem range-based for petlje, kod postaje jednostavniji i sigurniji:

```
for (auto& child : children_) {  
    // Obrada children_ elemenata  
}
```

### 4.2.4 Korišćenje `emplace_back` umesto `push_back`

U fajlu **SoundRecorder.cpp** (linija 145) Clang-Tidy analiza je detektovala da se koristi metoda `push_back`, iako je poželjno koristiti `emplace_back` zbog bolje efikasnosti pri dodavanju elemenata u STL kontejnere.

**Detektovani problem** Postojeći kod koristi `push_back`, što može dovesti do dodatnog kopiranja objekta prilikom dodavanja u vektor:

```
deviceNameList.push_back(deviceList);
```

**Preporučeno rešenje** Korišćenjem `emplace_back`, objekat se direktno kreira u vektoru bez nepotrebnog kopiranja, čime se poboljšava efikasnost:

```
deviceNameList.emplace_back(deviceList);
```

Ova promena omogućava optimizaciju performansi i preporučuje se primena u svim mestima gde se novi objekti dodaju u STL kontejnere.

### 4.2.5 Korišćenje `using` umesto `typedef`

U fajlu **title\_screen.cpp** (linija 20) Clang-Tidy analiza je detektovala korišćenje `typedef`, iako moderni C++ standardi preporučuju korišćenje `using` zbog bolje čitljivosti i fleksibilnosti.

**Detektovani problem** Postojeći kod koristi `typedef` za definisanje pokazivača na funkciju:

```
typedef void (*Drawer)();
```

**Preporučeno rešenje** Umesto `typedef`, preporučuje se korišćenje `using`, jer je sintaksički jednostavnije i pogodnije za složenije tipove:

```
using Drawer = void (*)();
```

Ova promena poboljšava čitljivost koda i usklađuje ga sa modernim C++ standardima.

## 5 Doxygen

Doxygen je alat za **automatsku generaciju dokumentacije** iz komentara u izvornom kodu, najčešće korišćen za C, C++, ali i za druge jezike poput Python-a, Java-e i Fortran-a. Omogućava **izradu strukturisane i lako pretražive dokumentacije** direktno iz koda, čime poboljšava održavanje i razumevanje projekata.

Glavna prednost Doxygen-a je što omogućava **automatsko izdvajanje opisa funkcija, klasa, struktura i interfejsa**, olakšavajući programerima da prate arhitekturu i funkcionalnost softvera. Takođe, podržava generisanje **HTML, LaTeX, PDF i XML dokumentacije**, čime omogućava prilagođavanje različitim potrebama.

Još jedna važna funkcionalnost je **podrška za grafički prikaz odnosa između klasa i zavisnosti između fajlova**, čime se vizuelno olakšava analiza kompleksnih kodnih baza. Doxygen se može koristiti iz komandne linije ili integrisati sa razvojnim okruženjima, što ga čini **korisnim alatom za kontinuirano održavanje i unapređenje dokumentacije u softverskim projektima**.

## 6 Analiza rezultata

### 6.1 Pokretanje Doxygen-a

Doxygen se pokreće izvršavanjem skripte `generate_doxygen.sh`, koja unutar sebe generiše konfiguracioni fajl `doxyfile`. Nakon izmene konfiguracije, dokumentacija se generiše komandom:



doxygen doxyfile

Generisana dokumentacija može se pregledati u pretraživaču otvaranjem:

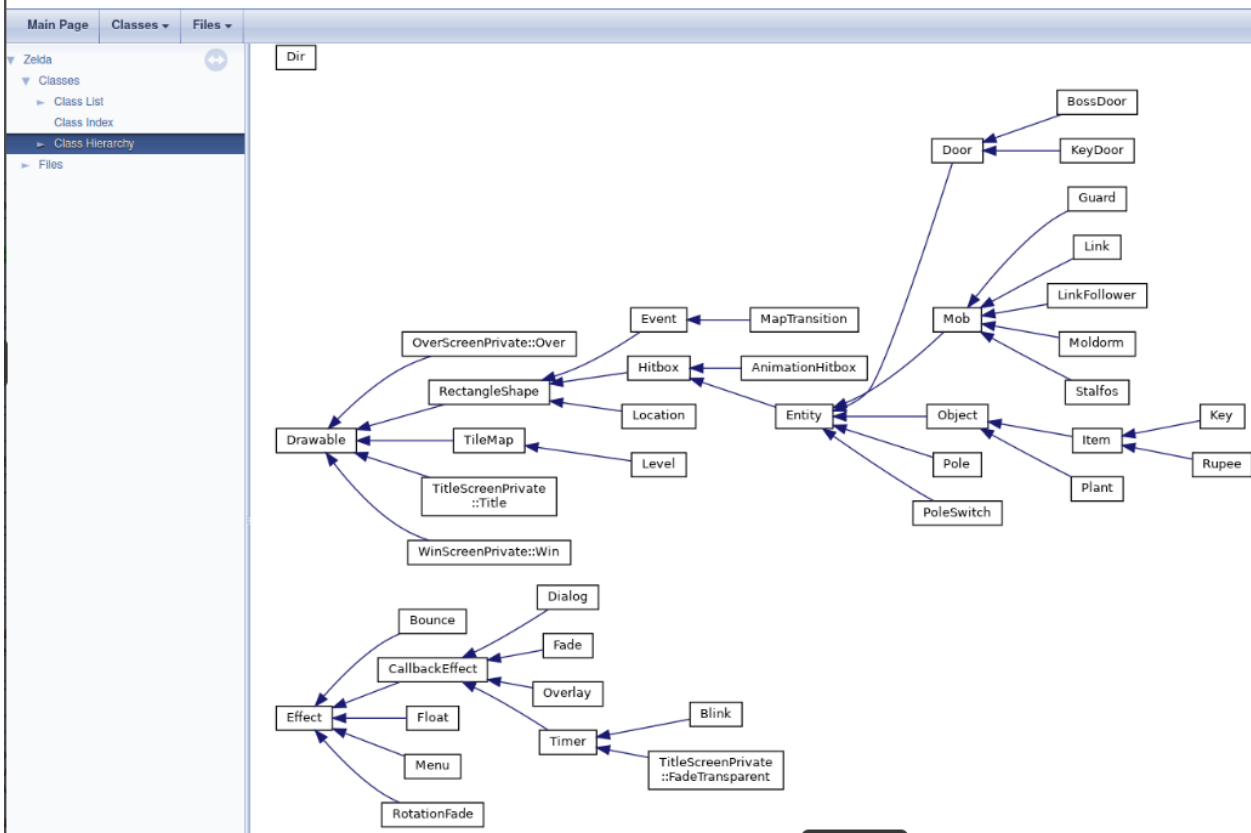
firefox docs/html/index.html

## 6.2 Opis parametara u Doxyfile-u

- **INPUT = src**  
Ovaj parametar određuje ulazni direktorijum iz kojeg će Doxygen čitati izvorne datoteke.
- **RECURSIVE = YES**  
Omogućava rekurzivno pretraživanje unutar svih poddirektorijuma definisanih u INPUT.
- **OUTPUT\_DIRECTORY = docs**  
Definiše direktorijum u kojem će biti smešteni generisani dokumentacioni fajlovi.
- **EXTRACT\_ALL = YES**  
Omogućava dokumentovanje svih entiteta u kodu, čak i ako nisu eksplicitno dokumentovani.
- **HAVE\_DOT = YES**  
Omogućava upotrebu Graphviz dot alata za generisanje dijagrama.
- **CALL\_GRAPH = YES**  
Generiše graf poziva (*call graph*) koji prikazuje odnose među funkcijama.
- **CALLER\_GRAPH = YES**  
Generiše graf pozivaoca (*caller graph*) koji prikazuje koje funkcije pozivaju određenu funkciju.
- **CLASS\_DIAGRAMS = YES**  
Omogućava generisanje dijagrama klasa.
- **DOT\_IMAGE\_FORMAT = png**  
Definiše format slike za grafove generisane putem dot alata.

Ovim podešavanjima omogućavamo generisanje hijerarhije klasa, grafova funkcionalnosti i drugih vizuelnih prikaza.

# Zelda



Slika 1: Deo hijerarhije klasa generisan Doxygen-om

## 6.3 Dodavanje komentara u kod



Da bi Doxygen pravilno generisao dokumentaciju, potrebno je koristiti Doxygen kompatibilne komentare u kodu.

### 6.3.1 Primer za klase u .hpp fajlovima

```
/** * @class Screen * @brief Opis. */ class Screen { ... };
```

### 6.3.2 Primer za .cpp implementaciju klasa

```
/** * @brief Opis. */ Screen::Screen() { ... }
```

 <b>TileMap</b>	Upravljanje slojevima i kolizijama pločica u igri
 <b>Tileset</b>	Upravljanje teksturama i prikazivanje pločica u igri

Slika 2: Vizualizacija opisa klasa

## 7 Valgrind

Valgrind je moćan alat za analizu i profilisanje softvera, prvenstveno korišćen u C i C++ programiranju, ali može biti primenjen i na druge jezike. Njegova glavna svrha je detekcija grešaka u memoriji, optimizacija performansi i analiza konkurentnog izvršavanja. Funkcioniše tako što pokreće program unutar sopstvenog virtuelnog okruženja, omogućavajući detaljno praćenje svih memorijskih operacija i izvršavanja funkcija.

Ovaj alat se često koristi u razvoju softvera gde je potrebno osigurati stabilnost, sprečiti curenje memorije i otkriti potencijalne probleme u upravljanju resursima. Valgrind omogućava programerima da testiraju svoje aplikacije bez potrebe za dodatnim modifikacijama koda, pružajući detaljne izveštaje o eventualnim greškama koje mogu dovesti do nepredvidivog ponašanja programa.

Glavni alati u okviru Valgrind paketa su: **Memcheck**, **Cachegrind**, **Callgrind**, **Massif**, **Helgrind** i **DRD**.

U ovom projektu biće primenjeni alati **Memcheck** i još jedan alat, koji će biti detaljnije opisani u nastavku.

## 7.1 Memcheck

Kako bi se proverila ispravnost upravljanja memorijom u aplikaciji, korišćen je Valgrind Memcheck alat. Ovaj alat detektuje curenje memorije, nevalidne pristupe i druge potencijalne probleme u dinamičkoj alokaciji memorije.

### 7.1.1 Analiza rezultata

Memcheck se pokreće izvršavanjem skripte `run_memcheck.sh`, koja omogućava analizu upravljanja memorijom tokom izvršavanja aplikacije.

Tokom testiranja, primetili smo da pokretanje Memcheck-a sa snažnijim opcijama poput `-leak-check=full` i `-show-leak-kinds=all` uzrokuje značajno usporavanje i zamrzavanje aplikacije. Kako bi analiza mogla da se izvede bez prekida rada aplikacije, korišćena je manje precizna konfiguracija Memcheck-a, koja i dalje pruža korisne informacije o upravljanju memorijom, ali sa smanjenim opterećenjem.

Rezime potrošnje memorije

Memcheck analiza je generisala sledeće podatke o memorijskim alokacijama:

- **Ukupno zauzeto na kraju izvršavanja:** 2,454,443 bajta u 21,964 blokova.
- **Ukupna potrošnja heap memorije:** 901,471 alokacija i 879,507 oslobađanja, sa ukupno 330,229,732 bajta dodeljene memorije.

Iako je većina memorije oslobođena tokom izvršavanja, postoji blago neslaganje između broja alokacija i oslobađanja, što može ukazivati na curenje memorije.

Otkrivena curenja memorije

Memcheck analiza je identifikovala sledeće oblike curenja memorije:

- **Definitivno izgubljeno:** 9,056 bajtova u 177 blokova – memorija koja nije oslobođena i do koje više ne postoji pristup.
- **Indirektno izgubljeno:** 1,875,545 bajtova u 18,507 blokova – memorija zauzeta kroz pokazivače koji su izgubljeni pre nego što su mogli biti oslobođeni.
- **Moguće izgubljeno:** 1,296 bajtova u 28 blokova – memorijski blokovi do kojih postoji ograničen pristup, što može ukazivati na potencijalno curenje.

- **Još uvek dostupno:** 568,546 bajtova u 3,252 blokova – memorija koja nije oslobođena, ali ostaje dostupna do završetka programa.

Indirektno izgubljena memorija predstavlja najveći problem, jer ukazuje na objekte koji su dinamički alocirani, ali nisu pravilno oslobođeni pre kraja izvršavanja programa.

Preporučene optimizacije za smanjenje curenja memorije su:

- Korišćenje pametnih pokazivača (`std::unique_ptr` i `std::shared_ptr`) u C++ okruženju kako bi se automatski oslobađala memorija.
- Dodavanje odgovarajućih `delete` poziva gde god se koristi dinamička alokacija memorije.
- Ponovno izvršavanje Valgrind Memcheck analize sa opcijom `-leak-check=full` u manje zahtevnim delovima aplikacije kako bi se preciznije locirali problemi.
- Korišćenje dodatnih alata kao što su Address Sanitizer ili GDB debugger za dublju analizu problema sa memorijom.

## 7.2 Callgrind

Callgrind je alat iz Valgrind paketa koji se koristi za **profilisanje performansi programa**. On analizira način na koji program koristi procesorske resurse i generiše detaljan izveštaj o pozivima funkcija i njihovom vremenu izvršavanja.

Tokom rada, Callgrind meri:

- Broj poziva svake funkcije (*call count*).
- Vreme provedeno u izvršavanju funkcije.
- Odnose između funkcija i njihove međuzavisnosti (*call graph*).
- Keš promašaje i optimizaciju pristupa memoriji.

Callgrind se često koristi zajedno sa alatom **KCachegrind**, koja omogućava vizuelizaciju podataka i olakšava analizu uskih grla u performansama programa.

Glavne prednosti Callgrind-a su:

- Omogućava analizu efikasnosti koda i identifikaciju funkcija koje troše najviše vremena.

- Pomaže u optimizaciji poziva funkcija i smanjenju prekomernog korišćenja CPU-a.
- Koristan je za refaktorisanje koda i poboljšanje ukupne efikasnosti aplikacije.

### 7.2.1 Analiza rezultata

Callgrind se pokreće izvršavanjem skripte `run_callgrind.sh`, koja omogućava analizu performansi i profilisanje potrošnje procesorskih instrukcija tokom izvršavanja aplikacije.

### 7.2.2 Funkcije sa najvećom potrošnjom instrukcija

U tabeli 1 prikazane su funkcije koje su najviše opteretile procesor u pogledu izvršenih instrukcija.

Funkcija	Broj instrukcija
0x000000000000302a0 (libopenal)	1,420,719,272 (7.86%)
RectangleShape::CollidesWith	734,788,391 (4.07%)
Mob::_UpdatePosition	628,620,872 (3.48%)
std::operator!=	519,141,766 (2.87%)
__expf_fma (math)	504,945,280 (2.80%)
std::_List_const_iterator::operator++	427,599,924 (2.37%)
RectangleShape** std::__copy_move	332,590,476 (1.84%)
vec2<float>::vec2	302,327,296 (1.67%)

Tabela 1: Funkcije sa najvećom potrošnjom instrukcija

### 7.2.3 Najčešće pozivane funkcije

Najčešće pozivane funkcije, koje se nalaze u osnovi interakcije unutar igre, uključuju:

- RectangleShape::CollidesWith
- Mob::\_UpdatePosition
- std::operator!=
- vec2<float>::vec2
- RectangleShape::position()

- `Level::Update(double)`
- `Mob::MeleeAttack(Hitbox*)`

#### 7.2.4 Moguće optimizacije

Funkcije sa najvećom potrošnjom instrukcija uključuju operacije detekcije sudara, ažuriranje pozicija neprijatelja i rad sa STL kontejnerima. Optimizacije se mogu postići uvođenjem prostorne deobe za sudare, smanjenjem učestalosti ažuriranja neprijatelja i zamenom `std::list` sa `std::vector` radi boljeg iskorišćenja CPU keša.

Najčešće pozivane funkcije pripadaju osnovnim mehanikama igre, uključujući rad sa objektima i fizičkim svojstvima. Optimizacija može uključivati unapređenje algoritama kretanja i izbegavanje suvišnih poziva na iteraciju kroz STL strukture.

#### 7.2.5 Vizuelna analiza sa KCacheGrind

Za dublju analizu podataka generisanih Callgrind alatom, korišćen je KCacheGrind, koji omogućava vizuelizaciju toka izvršenja funkcija i njihove međusobne zavisnosti. Slike grafova bice dostupne u okviru projekta.

## 8 Zaključak

Primena različitih alata za analizu koda omogućila je detaljnu proveru stabilnosti i efikasnosti aplikacije. Detektirani su potencijalni problemi u upravljanju memorijom, optimizaciji izvršavanja i strukturi koda, što pruža smernice za dalja poboljšanja.

Analiza pomoću Valgrind Memcheck-a ukazala je na postojanje curenja memorije, dok su druga testiranja pomogla u prepoznavanju mogućih optimizacija i poboljšanja performansi. Iako je testiranje uz snažne opcije Memcheck-a bilo ograničeno zbog opterećenja sistema, blaži pristup omogućio je uvid u ključne probleme bez narušavanja stabilnosti aplikacije.

Dalje unapređenje sistema može se postići optimizacijom memorijskih operacija, poboljšanjem upravljanja resursima i dodatnim verifikacionim testovima. Implementacija predloženih poboljšanja povećala bi efikasnost i pouzdanost aplikacije, smanjujući rizike povezane sa neoptimalnim rukovanjem memorijom i performansama.