Analiza projekta dagger-relic

Rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Jovan Vukićević
jormundur00@gmail.com

October 21, 2025

Contents

1 O analiziranom projektu

dagger-relic je mali gejming endžin otvorenog koda, prvobitno razvijen za potrebe stručnog kursa **Razvoj video igara u C++-u**, koji je kompanija Ubisoft držala tokom akademske 2022/2023. godine na Matematičkom fakultetu u Beogradu. Projekat je napisan u programskom jeziku C++ i sastoji se od fajla **Main.cpp**, koji definiše igru implementiranu u dagger-u , i implementacije samog endžina. Kako se na main grani projekta nalazi minimalistička

implementacija igre **Pong**, mi u ovoj analizi analiziramo granu **team/straw-hat-crew**, na kojoj je pravljena igra koja koristi više elemenata endžina.

2 Primenjeni alati i rezultati analize

Za potrebe poboljšavanja kvaliteta koda, kao i pronalaska grešaka u istom, nad ovim projektom su primenjeni sledeći alati: **clang-tidy**, **Valgrind Memcheck**, **Valgrind Massif** i **cppcheck**. U nastavku ovog rada ćemo predstaviti rezultate analize korišćenjem ovih alata zajedno sa zaključcima i potencijalnim poboljšanjima u samom kodu projekta.

2.1 clang-tidy

clang-tidy je alat za statičku analizu C++ koda koji pomaže u otkrivanju grešaka, potencijalnih problema i nepridržavanja stilskih smernica još pre izvršavanja programa. Osim detekcije sintaksnih i logičkih grešaka, **clang-tidy** pruža automatske sugestije i refaktorisanja, čime doprinosi boljoj čitljivosti i održivosti koda. Zbog integracije sa Clang kompajlerom, alat je precizan i može analizirati i kompleksne C++ konstrukcije, uključujući moderne standarde poput C++11, C++14 i C++17.

Za potrebe ove analize odabrano je praćenje C++17 standarda. Alat je pokrenut sa dodatnim proverama: **clang-analyzer-*** - odnosi se na pronalazak grešaka u kodu, **performance-*** - odnosi se na poboljšavanje performansi, kao i **cppcoreguidelines-***, **modernize-*** i **readability** provere, koje se odnose na poboljšavanje čitljivosti koda. Rezultati pokretanja alata su generisani korišćenjem skripte **run_clang_tidy.sh** u **clang-tidy** direktorijumu ovog projekta.

U narednim podsekcijama su navedeni problemi pronadjeni korišćenjem alata **clang-tidy**, kao i preporučena rešenja istih.

2.1.1 Inicijalizovanje promenljivih koje se nikada ne koriste

Pronadjene su inicijalizovane promenljive koje čuvaju rezultat poziva funkcije, ali se nikada kasnije ne koriste u kodu.

```
/home/jormundur/Jovan/VS/2024_Analysis_dagger-relic/dagger-relic/builddir/../src/Main.cpp:115:8:
    warning: Value stored to 'score' during its initialization is never
    read [clang-analyzer-deadcode.DeadStores]
              auto score = spawn()
                   ^~~~~   ~~~~~~~
```

Rešenje ovog problema je jednostavno: dovoljno je da samo uklonimo promenljive i pozovemo date funkcije bez čuvanja njihovih povratnih vrednosti.

2.1.2 Bespotrebno kopiranje objekata

U odredjenom broju funkcija prosledjujemo argumente po vrednosti, i u telu istih opet prosledjujemo date argumente po vrednosti, stvaranjem bespotrebnih

kopija (gde se ti argumenti više ne koriste u originalnoj funkciji).

```
/home/jormundur/Jovan/VS/2024_Analysis_dagger-relic/dagger-relic/builddir/../src/Animations.cpp:8:48:
    warning: parameter 'animation_name' is passed by value and only
    copied once; consider moving it to avoid unnecessary copies
    [performance-unnecessary-value-param]
       const auto current = Spritesheet::get_by_name(animation_name);
                                                     ^
                                                    std::move(   )
```

Umesto ponovnog kopiranja argumenata funkcija, mi možemo njihove resurse da prosledimo unutrašnjoj funkciji korišćenjem poziva **std::move**. Osim toga, treba uzeti u obzir i prenošenje konstantnih argumenata po referenci kao argumente funkcija, kada je to moguće, umesto dodatnih bespotrebnih kopiranja.

2.1.3 Nedostatak virtuelnog destruktora za klase koje imaju virtuelne metode

Nekoliko klasa nasledjuju klasu **SignalProcessor** koja ima definisane virtuelne metode, ali nema definisan virtuelni destruktor. Prilikom brisanja bilo kog izvedenog objekta preko pokazivača na baznu klasu neće biti pozvan destruktor izvedene klase, što dovodi do neodredjenog ponašanja programa.

```
../src/Main.cpp:27:8: warning: 'Brawl' has virtual functions but
    non-virtual destructor [clang-diagnostic-non-virtual-dtor]
struct Brawl : public Game, public SignalEmitter<GameStartSignal>,
    public MutAccessStorage<Player>, public MutAccessStorage<Enemy>,
    public MutAccessStorage<Healthbar>, public
    MutAccessStorage<ScoreRender>, public MutAccessStorage<TimeRender>
      ^
```

Rešenje je da u samu natklasu u kojoj su definisani ovi virtuelni pozivi definišemo virtuelni destruktor:

```
virtual ~SignalProcessor() = default;
```

2.1.4 Korišćenje magičnih brojeva

Magični brojevi su korišćeni u veoma velikoj količini pri definisanju konfiguracije same igre. Ovo može dovesti do manje čitljivosti koda, kao i poteškoća pri menjanju željenih vrednosti, jer one moraju biti ručno promenjene na svakom mestu gde se nalaze.

```
/home/jormundur/Jovan/VS/2024_Analysis_dagger-relic/dagger-relic/builddir/../src/Main.cpp:46:18:
    warning: 100 is a magic number; consider replacing it with a named
    constant
    [cppcoreguidelines-avoid-magic-numbers,readability-magic-numbers]
                     .with<Health>(100)
```

```
                                                                    ^
```

Kako klase **Main.cpp** i **TextRenderer.cpp** imaju veliki presek izmedju magičnih brojeva koje koriste, najbolje rešenje je da se definiše novi fajl **GameConfiguration.h** koji će sadržati definicije promeljivih koje čuvaju vrednosti magičnih brojeva.

2.1.5 Korišćenje makroa za definisanje konstanti

Makro **#define** ne bi trebao da se koristi pri definisanju konstanti, jer je on preprocesorska direktiva koja nema tip, ne poštuje opsege i ne učestvuje u proveri tipova.

```
../src/Main.cpp:8:9: warning: macro 'SCREEN_WIDTH' used to declare a
    constant; consider using a 'constexpr' constant
    [cppcoreguidelines-macro-usage]
#define SCREEN_WIDTH 800
        ^
```

Bolja praksa je da se koristi **constexpr** pri definisanju konstanti.

```
-#define SPEED_MOD 200.0f
+constexpr float SPEED_MOD = 200.0f;
```

2.1.6 Izbegavanje korišćenja vitičastih zagrada

U delovima koda gde je telo funkcije ili petlje jednolinijsko je izbegnuto korišćenje vitičastih zagrada (**{}**).

```
/home/jormundur/Jovan/VS/2024_Analysis_dagger-relic/dagger-relic/builddir/../src/SpriteRendering.cpp:3
    warning: statement should be inside braces
    [readability-braces-around-statements]
              if (sprite.texture == (ecs::Entity)0 || sprite.texture ==
                  ecs::no_entity)
                                                                      ^
                                                                       {
```

Ovo može dovesti manje čitljivog koda, pa je poželjno dodati zagrade tamo gde se trenutno ne koriste.

2.2 Valgrind Memcheck

Valgrind Memcheck je alat za dinamičko testiranje C i C++ programa koji detektuje greške u upravljanju memorijom. Prati sve alokacije i dealokacije memorije tokom izvršavanja programa i prijavljuje:

1. Curenja memorije (memory leaks) – memorija koja je alocirana, ali nije oslobodjena,

2. Korišćenje neinicijalizovane memorije,

3. Prekoračenja granica alociranih blokova (buffer overflow/underflow),

4. Neispravne dealokacije (npr. dvostruko oslobadjanje memorije).

Rezultati pokretanja alata su generisani korišćenjem skripte **run_memcheck.sh** u **valgrind/memcheck** direktorijumu ovog projekta. Kako **Memcheck** vrši svoju analizu tokom izvršavanja aplikacije, aplikacija je puštena da radi 10 sekundi pod analizom alata.

Inicijalnim pokretanjem alata nad projektom, pronadjen je odredjen broj definitivnih curenja memorije, kao i broj *mogućih* curenja.

```
LEAK SUMMARY:
   definitely lost: 109,392 bytes in 26 blocks
   indirectly lost: 8,906 bytes in 22 blocks
   possibly lost: 102,081 bytes in 173 blocks
   still reachable: 3,344,451 bytes in 10,254 blocks
                      of which reachable via heuristic:
                         multipleinheritance: 14,400 bytes in 21 blocks
    suppressed: 0 bytes in 0 blocks
```

Od ključnog značaja za aplikaciju je popravljanje curenja memorije definisanih kao *definitivna* curenja.

Kao jedan od uzroka definitivno izgubljenih blokova memorije je otkriven problem alociranja memorije unutar funkcije **TextureLoader::load_texture**.

```
==66273== 176 bytes in 22 blocks are definitely lost in loss record
    3,494 of 4,367
==66273==   at 0x4849013: operator new(unsigned long) (in
    /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==66273==   by 0x1CA91D:
    TextureLoader::load_texture(std::__cxx11::basic_string<char,
    std::char_traits<char>, std::allocator<char> > const&)
    (Loaders.cpp:29)
==66273==   by 0x1CAA56:
    TextureLoader::load_asset(std::__cxx11::basic_string<char,
    std::char_traits<char>, std::allocator<char> >,
    std::__cxx11::basic_string<char, std::char_traits<char>,
    std::allocator<char> >) (Loaders.cpp:49)
==66273==   by 0x1CB127: TextureLoader::load_assets() (Loaders.cpp:125)
```

Lako rešenje ovog problema se može dobiti prelaskom na korišćenje *pametnih* pokazivača u odnosu na prethodno korišćene.

```
-  memory::RawPtr<Texture> load_texture(const String&);
+  std::unique_ptr<Texture> load_texture(const String&);
```

Problem koji se odnosi na sve ostale izgubljene blokove memorije (a i većinu svih izgubljenih blokova) je problem curenja memorije u klasama **TimeRender** i **SpriteRender**, jer njihovi objekti nisu oslobadjali resurse koje je alocirala biblioteka **SDL_ttf**.

```
==66273== 59,053 (54,608 direct, 4,445 indirect) bytes in 2 blocks are
    definitely lost in loss record 4,358 of 4,367
==66273==   at 0x4848899: malloc (in
    /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==66273==   by 0x48FACD9: ??? (in
    /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.18.2)
==66273==   by 0x341737: TTF_OpenFontIndexDPIRW (SDL_ttf.c:1784)
==66273==   by 0x3420A7: TTF_OpenFontIndexDPI (SDL_ttf.c:2040)
```

Rešenje ovog problema je dodavanje destruktora koji poziva **TTF_CloseFont(font)** i onemogućavanje kopiranja objekta ovih klasa. Time se sprečava gubitak memorije i potencijalni problemi sa višestrukim oslobadjanjem istog resursa.

```
TimeRender::~TimeRender() {
    if (font) {
        TTF_CloseFont(font);
        font = nullptr;
    }
}

TimeRender(const TimeRender&) = delete;
TimeRender& operator=(const TimeRender&) = delete;
```

Sa ove dve popravke prethodno navedenih problema, parsiranjem rezultata analize ponovnim pokretanjem alata nad izmenjenim projektom, definitivnih, kao i indirektnih (nastalih kao produkt definitivnih) curenja više nema.

```
LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
   possibly lost: 102,081 bytes in 173 blocks
   still reachable: 3,707,871 bytes in 11,015 blocks
                      of which reachable via heuristic:
                         multipleinheritance: 19,200 bytes in 28 blocks
    suppressed: 0 bytes in 0 blocks
```

2.3 Valgrind Massif

Valgrind Massif je alat za profajliranje memorije koji analizira upotrebu hip memorije tokom izvršavanja programa. Prati koliko memorije program zauzima u različitim trenucima i generiše *snepšotove* memorijskog stanja. Na osnovu tih podataka moguće je otkriti curenja memorije, prekomernu alokaciju ili

neefikasnu upotrebu resursa. Rezultati se obično vizualizuju pomoću **ms_print**, što omogućava pregled najvećih potrošača memorije u programu.

Rezultati pokretanja alata su generisani korišćenjem skripte **run_massif.sh** u **valgrind/massif** direktorijumu ovog projekta. Kako **Massif** vrši svoju analizu tokom izvršavanja aplikacije, aplikacija je puštena da radi 10 sekundi pod analizom alata. Inicijalnim pokretanjem alata nad projektom, i parsiranjem njegovih rezultata korišćenjem alata **ms_print**, dobijen je sledeći grafik:

```
    MB
22.62^                             #
     |                             #
     |                             #
     |                             #
     |                             #
     |                             #
     |                             #
     |                             #
     |
         @@@#:::::@@:::::::::::::::::::@::::@:::::@::
     |                              @ @#: :::@ :: : : : ::: ::::@::::@:::
         :@::
     |                            @@:@ @#: :::@ :: : : : ::: ::::@::::@:::
         :@::
     |                          :@ :@ @#: :::@ :: : : : ::: ::::@::::@:::
         :@::
     |                        :::@ :@ @#: :::@ :: : : : ::: ::::@::::@:::
         :@::
     |                        : :@ :@ @#: :::@ :: : : : ::: ::::@::::@:::
         :@::
     |                        : :@ :@ @#: :::@ :: : : : ::: ::::@::::@:::
         :@::
     |                        : :@ :@ @#: :::@ :: : : : ::: ::::@::::@:::
         :@::
     |                      @: :@ :@ @#: :::@ :: : : : ::: ::::@::::@:::
         :@::
     |                      @: :@ :@ @#: :::@ :: : : : ::: ::::@::::@:::
         :@::
     |                  @   @: :@ :@ @#: :::@ :: : : : ::: ::::@::::@:::
         :@::
     |                @@@:::@: :@ :@ @#: :::@ :: : : : ::: ::::@::::@:::
         :@::
     |          ::::::@@ @: :@: :@ :@ @#: :::@ :: : : : ::: ::::@::::@:::
         :@::
   0
       +-------------------------------------------------------------------------->s
     0                                                                    15.99

Number of snapshots: 54
 Detailed snapshots: [10, 11, 12, 15, 18, 20, 21, 22 (peak), 27, 40, 45,
     50]
```

Prvi porast u korišćenoj hip memoriji se može naći u 10. snepšotu:

```
95.79% (2,184,778B) (heap allocation functions) malloc/new/new[],
    --alloc-fns, etc.
->20.71% (472,368B) 0x522D976: ??? (in
    /usr/lib/x86_64-linux-gnu/libX11.so.6.4.0)
| ->20.71% (472,368B) 0x522E05D: _XimLocalOpenIM (in
    /usr/lib/x86_64-linux-gnu/libX11.so.6.4.0)
|   ->20.71% (472,368B) 0x5225A8F: _XimOpenIM (in
    /usr/lib/x86_64-linux-gnu/libX11.so.6.4.0)
|     ->20.71% (472,368B) 0x495F345: ??? (in
    /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.18.2)
|       ->20.71% (472,368B) 0x4966AFC: ??? (in
    /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.18.2)
|         ->20.71% (472,368B) 0x493888D: ??? (in
    /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.18.2)
|           ->20.71% (472,368B) 0x4898426: ??? (in
    /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.18.2)
|             ->20.71% (472,368B) 0x21CF3D:
    core::WindowingModule::on_start() (Windowing.cpp:15)
|               ->20.71% (472,368B) 0x221072: core::Engine::run()
    (Engine.h:143)
|                 ->20.71% (472,368B) 0x2205A1: main (Main.cpp:197)
```

Detaljnom analizom ovog snepšota možemo primetiti da se ovaj porast u korišćenju memorije može pripisati inicijalizaciji grafičkog sistema i prozora. Ovde možemo napraviti malo poboljšanje u korišćenju memorije, jer se može primetit da opcijom **SDL_INIT_EVERYTHING** inicijalizujemo više komponenti sistema koje nam zapravo nisu potrebne, kao što su podrška za zvuk ili džojstik kontrolere, pa možemo ručno inicijalizovati samo komponente koje zapravo koristimo.

```
-  if (SDL_Init(SDL_INIT_EVERYTHING) < 0)
+  if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_EVENTS) < 0)
```

Posmatrajući grafik, vidimo da se najveći utrošak memorije dogadja u 22. snepšotu.

```
97.33% (23,087,732B) (heap allocation functions) malloc/new/new[],
    --alloc-fns, etc.
->33.70% (7,994,920B) 0x48AA1F8: ??? (in
    /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.18.2)
| ->33.70% (7,994,920B) 0x4931DCF: ??? (in
    /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.18.2)
|   ->33.70% (7,994,920B) 0x2CFBEB: IMG_LoadPNG_RW (IMG_png.c:382)
|   | ->33.70% (7,994,920B) 0x2CD117: IMG_LoadTyped_RW (IMG.c:289)
|   |   ->33.70% (7,994,920B) 0x2CCF11: IMG_Load (IMG.c:205)
|   |     ->33.70% (7,994,920B) 0x1CA953:
    TextureLoader::load_texture(std::__cxx11::basic_string<char,
```

```
    std::char_traits<char>, std::allocator<char> > const&)
    (Loaders.cpp:31)
```

Ovaj snepšot predstavlja trenutak u izvršavanju kada se učitavaju teksture u igri. Kako se sve teksture učitavaju odjednom u igri (i koriste kontinualno), ovde nema mogućnosti za poboljšanje performansi korišćenjem tehnika pojedinačnog učitavanja i de-učitavanja tekstura u trenutku kada su potrebne. Jedno potencijalno poboljšanje može biti čuvanje više tekstura unutar jednog fajla (kao *sprajtmape*), ali to u ovom projektu nije radjeno.

2.4 cppcheck

Cppcheck je statički analizator koda za programski jezik C/C++. Njegova svrha je da otkrije potencijalne greške, neinicijalizovane promenljive, curenja memorije, neefikasnosti i probleme sa stilom, bez izvršavanja programa. Za razliku od kompajlera (pa time i alata **clang-tidy**), **cppcheck** fokusira se na logičke i runtime-prone greške, a kao i prethodni i na preporuke za poboljšanje kvaliteta i sigurnosti koda.

Rezultati pokretanja alata su generisani korišćenjem skripte **run_cppcheck.sh** u **cppcheck** direktorijumu ovog projekta. Radi lakše vizuelizacije rezultata, oni se mogu parsirati alatom **cppcheck-gui**.

U narednim podsekcijama su navedeni problemi pronadjeni korišćenjem alata **cppcheck**, kao i preporučena rešenja istih. Prva dva problema se odnose na probleme nivoa **warning**, koji predstavljaju probleme koje mogu dovesti do ozbiljnih grešaka u kodu. Posle njih naveden je problem nivoa **performance**, koji predstavlja problem koji može loše uticati na performanse aplikacije. Stilski propusti su većinski pokriveni prethodnim izvršavanjem alata **clang-tidy**, pa se u ovoj sekciji nećemo obazirati na njih.

2.4.1 Pozivanje čiste virtuelne funkcije u konstruktoru

U konstruktoru klase `SignalProcessor` registrovana je čista virtuelna funkcija `process_signal` za rad sa signalima:

```
template<typename T>
struct SignalProcessor : public AccessTrait
{
    virtual void process_signal(T& signal) = 0;

    SignalProcessor()
    {
        core::Engine::get_instance().dispatcher.sink<T>()
            .template connect<&SignalProcessor<T>::process_signal>(this);
    }
};
```

Problem nastaje zato što C++ standard nalaže da virtuelne funkcije ne pozivaju izvedene implementacije kada se objekat još uvek konstruiše. Poziv čiste virtuelne funkcije u konstruktoru može dovesti do **runtime grešaka** ili segfault-a, jer funkcija ne postoji u bazičnom konstruktoru.

Rešenje je da se registracija funkcije odloži i izvrši **nakon konstrukcije objekta**, kroz posebnu inicijalizacionu metodu:

```
struct MyProcessor : public SignalProcessor<MySignal> {
    MyProcessor() { ... }
    void init_signals() { SignalProcessor::init_signals(); }
    void process_signal(MySignal& s) override { ... }
};
```

Na ovaj način je sigurno da je virtuelna funkcija već definisana u izvedenoj klasi kada se registruje, čime se izbegavaju runtime problemi.

2.4.2 Neinicijalizovana polja klasa

U C++-u, polja klasa koje nisu eksplicitno inicijalizovane u konstruktoru mogu imati nepredvidive vrednosti, što može dovesti do neispravnog ponašanja programa. Alati za statičku analizu koda, kao što je **Cppcheck**, mogu detektovati kada član promenljiva nije inicijalizovana u konstruktoru. Detektovani problem nalaže da je polje **Brawl::num** jedno takvo polje. Pogledom na kod, možemo videti da je ovo polje greškom deklarisano izvan tela metode u kojoj se koristi, pa je i time registrovano kao polje. Lako rešenje ovog problema je refaktorisanje polja u promenljivu metode.

```
-  int num;
   void on_start() override
   {
-          num = rand() % RANGE_X;
+          int num = rand() % RANGE_X;
```

2.4.3 Funkcija koja može biti statička

U klasi `SignalEmitter` funkcija `emit` ne koristi nijedan član klase (`this`), što znači da njeno ponašanje nije vezano za konkretnu instancu. Cppcheck u ovom slučaju sugeriše da se funkcija može označiti kao `static`.

Označavanje funkcije kao `static` ima nekoliko prednosti:

- Ne prosledjuje se implicitni `this` pokazivač prilikom poziva, što može doneti minimalno poboljšanje performansi.
- Jasno pokazuje da funkcija ne menja niti ne koristi stanje instance klase.
- Može se, u nekim slučajevima, premestiti u *unnamed namespace* ako je lokalna za fajl i ne zahteva globalnu vidljivost.

Ipak, iako je tehnički moguće označiti funkciju kao `static`, to ne mora biti konceptualno ispravno u svim slučajevima. Funkcija može biti logički vezana za klasu i njenu namenu, čak i ako trenutno ne koristi članove, pa je potrebno razmotriti dizajn pre primene ove promene.

3 Zaključci

Pokretanjem alata **clang-tidy** otkriven je veliki broj stilskih propusta, koji uglavnom potiču iz autorovog korišćenja starijeg C-olikog stila pri pisanju C++ projekta, kao i korišćenja velikog broja magičnih brojeva. Pored toga, alat je identifikovao i propuste koji mogu uticati na performanse aplikacije, poput bespotrebnog kopiranja i korišćenja kopija promenljivih.

Pokretanjem alata **Valgrind Memcheck** je otkriven problem korišćenja polja objekta pre nego što je inicijalizovano, što dovodi do nedefinisanog ponašanja programa. Osim toga, pronadjeno je nekoliko definitivnih curenja memorije, gde su veća curenja produkti nedostatka destruktora TimeRender i ScoreRender objekata (pa i oslobadjanja njihovih dinamički alociranih polja), dok se manje curenje zasnivalno na nedostatku oslobadjanja pokazivača tekstura.

Pokretanjem alata **Valgrind Massif** je analizirana upotreba hipa tokom izvršavanja programa. Zaključeno je za najveći deo upotrebe hipa zasluženo učitavanje tekstura, koje se radi odjednom. Kako aplikacija odmah po učitavanju tekstura koristi sve teksture, nema potrebe za ručnim učitavanjem tekstura onda kada su potrebne (što bi predstavljalo optimizaciju u slučajevima gde se ne koriste sve teksture u isto vreme). Zauzeće hip memorije je takodje povećano više nego potrebno zbog odabira opcije SDL_INIT_EVERYTHING pri inicijalizovanju SDL biblioteke/sistema, jer time automatski inicijalizujemo i podsisteme koje ne koristimo u našoj aplikaciji (kao podrška za zvuk ili džojstik kontrolere).

Pokretanjem alata **cppcheck** je pronadjeno par grešaka koje mogu dovesti do nedefinisanog ponašanja pri pokretanju aplikacije, kao što su ne-inicijalizovano polje objekta klase, kao i pozivanje čiste virtuelne metode u konstruktoru klase. Detektovano je i par grešaka koje bi mogle da utiču na performanse aplikacije, koje se odnose na funkcionalno statičke metode koje su definisane kao instancne metode klasa, kao i inicijalizovanje objektnih promenljivih u delovima koda gde one možda neće biti korišćene.