Izrada igre obrane tornjeva u programskom jeziku C++ i SDL-u

Vugrinec, Tomislav

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Vugrinec

Izrada igre obrane tornjeva u programskom jeziku C++ i SDL-u

ZAVRŠNI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Vugrinec

Matični broj: 44915/16–R

Studij: Informacijski sustavi

Izrada igre obrane tornjeva u programskom jeziku C++ i SDL-u

ZAVRŠNI RAD

Mentor:

Prof. dr. sc. Radošević Danijel

Varaždin, rujan 2019.

Tomislav Vugrinec

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski 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

Tema ovog rada jest izrada igre obrane tornjeva (eng. Tower Defense game) korištenjem dinamičke SDL 2.0 biblioteke čije funkcionalnosti koristimo za različita manipuliranja događajima od korisnikova pritiska tipke na mišu ili tipkovnici pa sve do spremanja slika (*eng. sprite*) objekata igre u VRAM memoriju (*eng. Video Random Access Memory*) unutar grafičke kartice. Sljedeći cilj projekta je ispitati objektno orijentirani instrumentarij i sintaksu koji C++ jezik nudi. Različiti objekti odvojeni su u zasebne korisničke klase definirane u C++ jeziku koje imaju jasno definirano javno sučelje namjenjeno upotrebi klase u drugim izvornim datotekama (*eng. source code*) te privatno sučelje koje je sakriveno korisniku klase, no potrebno je budući da privatno sučelje omogućuje sve funkcionalnosti koje klasa pruža preko svojeg javnog sučelja. Ovakva upotreba klasa i odjeljivanje kodova u zasebne, manje, jedinice glavna je karakteristika objektno orijentiranog programiranja – načina programiranja koji se danas sve češće primjenjuje u velikim korporacijama zbog svoje fleksibilnosti, modularnosti i ponovne iskoristivosti koda. U uvodu rada opisan je sam žanr igre te nekoliko predstavnika tog žanra. U metodama i tehnikama ukratko su opisani alati korišteni za izradu ove igre, a u razradi teme opisan je osnovni kostur igre te najbitniji elementi igre obrane tornjeva, poput: implementacije neprijatelja i sustava kojim neprijatelji prate unaprijed zadani put.

Ključne riječi: C++, C++ STL, SDL api, OOP (Object Oriented Programming), računalna igra, igra obrana tornjeva, programiranje

Sadržaj

1. Uvod

Igra obrane tornjeva zapravo je pod žanr strateških igara koje se izvode u stvarnom vremenu (*eng. real-time strategy game*). [4] Cilj igre jest obraniti se od valova neprijatelja koji neumorno marširaju ka resursu koji je za igrača od životne važnosti (ili ka rubu ekrana), stoga igrač gradi napadačke tornjeve koji će mu poslužiti kao primarna zaštita od navale neprijateljskih vojnika. Igra *Bloons Tower Defense*[1] jest tipičan predstavnik igre obrane tornjeva. U toj igri neprijatelji slijede unaprijed određeni put na kojem nailaze na razne tornjeve koji ih pokušavaju uništiti. Postojeći tornjevi se s prolaskom vremena mogu i unaprijediti te time dobiti na brzini ispaljivanja projektila ili na samoj snazi svakog zasebnog projektila. Iako su osnovna pravila igre vrlo jednostavna „ zaštititi najvrijednije resurse od navale neprijatelja“ u ovom žanru postoji mnogo varijacija na osnovnu mehaniku igre. *Onslaught*[2] upravo je varijacija igre obrane tornjeva gdje tornjevi dobivaju značajne moći (*eng. combo*) ukoliko su određeni tipovi tornjeva izgrađeni u njihovoj neposrednoj blizini. Mala promjena ili dodana funkcionalnost igre može uvelike promijeniti percepciju igrača prilikom igranja same igre, ponekad su igre toliko različite od samog žanra kojim se opisuju da niti sami igrači ne mogu odlučiti u koji žanr bi smjestili tu igru. Dobar primjer radikalne promjene dizajna igre obrane tornjeva jest igra *Plants vs Zombies*[3] sa istim ciljem – odagnanjem valova neprijatelja kako nebi dospjeli do kraja ekrana ili resursa koji igrač čuva, no sada igrač istovremeno brani 5 staza te tornjeve (u ovoj igri biljke) postavlja direktno na stazu koju neprijatelji napadaju. Osim biljaka (tornjeva) koje napadaju, igrač može stvoriti i biljke koje ne mogu gađati, no proizvode resurs potreban za izgradnju ostalih obrambenih biljaka. Taj segment novih tornjeva uvodi lagani sistem ekonomije u igru te time značajno utječe na stil igre igrača te njegovu percepciju prema samoj igri. Navedene igre redom su 2D igre ili simulacija 3D perspektive , no postoje i 3D igre obrane tornjeva poput popularne igre „ Orcs Must Die 2!“ ili „Dungeon Defenders II“ u kojoj osim standardnog iskustva za jednog igrača (*eng. single player game*) igra podržava istovremenu igru više igrača (*eng. multiplayer game*) na istom nivou. Igra koja je izrađena u okviru ovoga rada primjer je 2D igre obrane tornjeva, a po dizajnu same igre dijeli sličnosti sa igrom *Bloons Tower Defense*[3].

[1] Bloons Tower Defense 5, https://ninjakiwi.com/Games/Tower-Defense/Play/Bloons-Tower-Defense-5.html
[2] Onslaught 2, http://onslaught.playr.co.uk/
[3] Plants vs Zombies GOTY, https://store.steampowered.com/app/3590/Plants_vs_Zombies_GOTY_Edition/

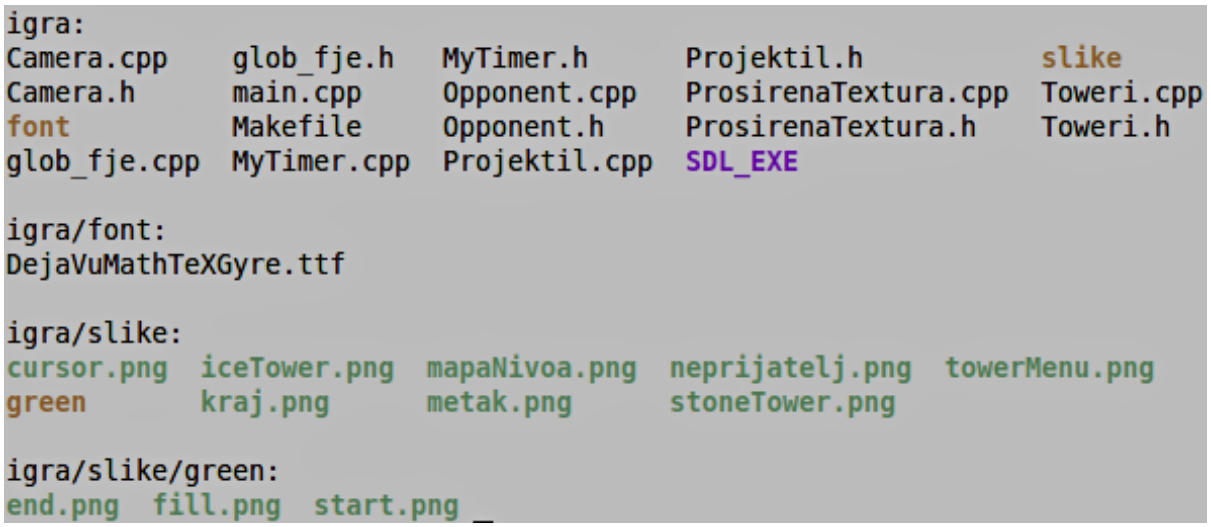
2. Metode i tehnike rada

Kod igre implementiran je u Emacs 25.2.2 proširivom uređivaču teksta (*eng. text editor*) korištenjem objektno orijentiranog pristupa tj. kod je podijeljen na više izvornih (.cpp) datoteka koje datoteke zaglavlja (.h) povezuju u jednu zajedničku cjelinu. Prilikom kodiranja igrice korišten je besplatan Git sustav za verzioniranje koda (*eng. version control system*). Projekt je kompajliran (*eng. compiled*) korištenjem Clang++ 6.0.0 kompajlera i Clang-tidy alata LLVM projekta koji prilikom pisanja koda igre simultano provjerava postoje li greške ili potencijalne greške unutar napisanog koda vodeći se smjernicama C++14 standarda. [1] Način samog kompajliranja izvornih datoteka definiran je u „Makefile“ datoteci korištenjem make sistema za izgradnju izvršnih datoteka (*eng. make build system*). Nivo igre rađen je u besplatnom programu „Tiled“ koji je također otvorenog koda, a za interakciju igrača sa igrom – događaji tipkovnice, miša, prikazivanje i spremanje slika igrice korištena je SDL 2.0 dinamička biblioteka (*eng. dynamic library*). U nastavku detaljnije će biti opisana SDL 2.0 dinamička biblioteka.

2.1. Izrađene programske datoteke

Kako bi se realizirao ovaj projekt izrađene su izvorne datoteke: „Camera.cpp“, „glob_fje.cpp“, „main.cpp“, „MyTimer.cpp“, „Opponent.cpp“, „Projektil.cpp“, „ProsirenaTextura.cpp“ i „Toweri.cpp“. Izvorne datoteke međusobno su povezane unutar „main.cpp“ izvorne datoteke uključivanjem izrađenih datoteka zaglavlja: „Camera.h“, „glob_fje.h“, „MyTimer.h“, „Opponent.h“, „Projektil.h“, „ProsirenaTextura.h“ te „Toweri.h“. Sve te datoteke u finalnu aplikaciju naziva „SDL_EXE“ objedinila je „Makefile“ datoteka koja sadrži recept izgrade izrađenih programskih kodova aplikacije. Osim tih datoteka za realizaciju igre korišteni su i direktoriji „font“ i „slike“. Unutar direktorija font nalazi se besplatan preuzeti font „DejaVuMathTexGyre.ttf“ koji se koristi prilikom ispisa završne poruke na površinu prozora aplikacije. Direktorij slike sadržava svu multimediju korištenu unutar izrađene aplikacije. Slika „cursor.png“ slika je pokazivača miša unutar aplikacije. IceTower.png i stoneTower.png jesu slike tornjeva unutar igre. MapaNivoa.png jest slika nivoa igre visoke rezolucije. Unutar direktorija slike nalazi se i direktorij naziva green koji sadržava slike korištene za realizaciju sistema životnih bodova igrača. Slika metak.png moja je osobna izrada, dok su ostale slike ili preuzete besplatno ili su dio kupljenog paketa slika sa webstranice craftpix.net[4] . Slika broj 1. prikazuje spomenute datoteke direktorija.

[4] Puna putanja web adrese kupljenog paketa slika: https://craftpix.net/product/tower-defense-2d-game-kit/



Slika 1: Izgled mape igre obrane tornjeva

2.2. SDL 2.0

Simple DirectMedia Layer (u nastavku: SDL) razvojna je biblioteka pisana u C programskom jeziku koja podržava sve poznatije operacijske sustave i platforme. Trenutno podržani sustavi i platforme su: Windows, GNU/Linux, Mac OS X, iOS i Android. [2] Neovisnost programskog koda o platformi koju SDL biblioteka pruža u C++ jeziku realizira se provjerom predefiniranih makroa operacijskih sustava prikazanu sljedećim programskim kodom:

```
#ifdef __linux__
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>
#include <SDL2/SDL_mixer.h>
#include <SDL2/SDL_ttf.h>
#elif
#include <SDL.h>
#include <SDL_image.h>
#include <SDL_mixer.h>
#include <SDL_ttf.h>
#endif
```

U ovom primjeru #ifdef naredba provjerava postoji li definiran makro imena __linux__ koji će biti definiran ako je operacijski sustav na kojem se izvodi C++ kompajler GNU/Linux, a ukoliko __linux__ makro nije definiran tada će se izvoditi blok naredbi nakon #elif naredbe. Predefiniran makro za provjeru Windows platforme je _WIN32, no u ovom primjeru kompajler će uključiti SDL2 datoteke zaglavlja ukoliko se kod kompajlira na linuxu, a na svim ostalim platformama uključiti će SDL zaglavlja nakon #elif naredbe.

SDL je trenutno u razvoju 20ak godina, a prvotno ga je stvorio Sam Latinga za tvrtku „Loki software“ čija je glavna djelatnost bila prenašanje (*eng. port*) postojećih titula igara na

Linux operacijski sustav. SDL je tokom godina razvoja konstantno unaprjeđivan te je danas on „de facto“ standard za razvoj multimedijskih aplikacija izvan Windows operacijskog sustava, a dizajniran je kako bi omogućio niski pristup (*eng. low level access*) prema zvuku, tipkovnici, mišu, komandnoj palici (*eng. joystick*) i grafičkoj kartici preko Direct3D i OpenGLa.

SDL 2.0 licenciran je zlib licencom koja omogučava korištenje same biblioteke besplatno, čak i u komercijalne svrhe. Osim toga zlib licenca dopušta i modifikaciju biblioteke uz uvjet da se naznači da je to modificirana verzija i uz uvjet da osoba koja je izmjenila biblioteku prizna zasluge kreatorima originalne biblioteke. SDL biblioteka posjeduje primitivan sustav prikaza slika i primitivnih oblika na ekran operacijskog sustava po izboru, no razvojni inžinjer nije ograničen striktno na SDL biblioteku, drugim riječima, SDL biblioteka zapravo je sučelje optimizirano za Direct3D i OpenGL API (*eng. application programming interface*). [3] Iako mnogi programski alati za stvaranje igra (*eng. game engine*) omogučavaju iste pa čak i dodatne funkcionalnosti, oni to često rade s više abstrakcije dosta pojednostavljujući kod same igre, no zbog te visoke abstrakcije koda često korisnik nije skroz upučen u procese koji se izvršavaju unutar tih programskih alata. Drugi razlog zašto neki proizvođači igra koriste SDL jest mogučnost veće optimizacije koda ili možda lakša implementacija ključnih mehanika igre budući da su razvojnom inžinjeru aplikacije dostupne SDL naredbe niže abstrakcije. O samoj funkcionalnosti koju SDL biblioteka pruža govori i činjenica da su mnoge popularne komercijalne igre danas, nastale korištenjem SDL biblioteke, poput: „Team Fortress 2“, „Prison Architect“ i „World of Goo“. Osim za razvoj igra, tvrtka „Valve“ koristi SDL u implementaciji brze klijentske aplikacije vrlo niske potrošnje resursa „Steam overlay“ koja korisniku Steam platforme omogućava brz pristup platformi ukolko se korisnik nalazi unutar igre kako bi primjerice prijatelja pozvao da zajedno odigraju neku video igricu [2].

Slika 2: Logo SDL biblioteke [2]

3. Razrada teme

Tema ovog rada jest ispitati instrumentarij C++ jezika zajedno sa funkcionalnostima SDL 2.0 biblioteke uključene unutar jezika, stoga je u nastavku struktura nadolazećih podpoglavlja sljedeća. Pri samom početku analiziramo osnovni kostur igre te ga grupiramo u veće logičke i funkcionalne dijelove. Nakon toga analiziraju se funkcije i metode korištene unutar igre koje su nam od velikog značaja za samo razumijevanje kostura igre. Kraj razrade teme analizira dizajn izdvojenih glavnih komponenata igre, poput: sustava kamere, životnih bodova, implementacije prohodnog puta te sustava stvaranja neprijatelja koji slijede unaprijed određeni put. Paralelno uz analizu izrađenih komponenata igre ispitivati će se postojeći instrumentarij i sintaksa C++ jezika i opisivati korištene naredbe uključene SDL biblioteke.

3.1. Osnovni kostur igre

Ovako izgleda kostur samog programa. Najprije bool init() globalna funkcija stvara prozor „gWindow“ u kojem ćemo prikazivati našu igru, a zatim i globalni prikazivač „gRenderer“ koji možemo zamisliti kao skriveni i još ne prikazani okvir (*eng. frame*) prozora. Nakon toga dolazi globalna funkcija bool loadResources() koja prije pokretaja same igre najprije učitava font i slike koje se prikazuju unutar igre. Ukoliko su obje funkcije prethodno rezultirale uspjehom glavni program učitava lokalne variable, strukture i liste.

```
int main(){

    if ( !init() ) {
        printf("Greska prilikom pokretanja programa!\n");
    }
    else {
        if ( !loadResources() ) {
            printf("Greska prilikom ucitavanja resursa!\n");
        }
        else {  // sve se ispravno ucitalo

            SDL_Event event;
            bool isRunning = true;

            Camera cam;
            std::list< Enemy> creeps;
            std::list< RealTower> towerList;
            std::list< Projektil> bullets;

            path walkPath[3];
            loadWalkablePaths( walkPath );

            MyTimer timerSpawn;
            timerSpawn.Start();
```

```
            while ( isRunning ) {
                while ( SDL_PollEvent( &event )) {
                    if ( event.type == SDL_QUIT ) {
                        isRunning = false;
                    }

                    cam.handleCameraMouse( event, misX, misY );

                } // kraj event loopa
                if ( cam.cursorAtBorder ) {
                    cam.adjustCameraBorder( cam.getForceX(),
cam.getForceY(), timerCursorMove.GetTicks() );
                }

                if ( timerSpawn.GetTicks() >= 1000 ) {
                    Enemy ufo(walkPath[0]);
                    ufo.sprite = &neprijateljPT;
                    creeps.push_back( ufo );

                    timerSpawn.Start();
                }

                SDL_SetRenderDrawColor( gRenderer, 0xff, 0xff, 0xff, 0xff);
                SDL_RenderClear( gRenderer );

                level.render(0, 0, &kamera );
                renderGameHealth();
                gKursorMisaPT.render( misX, misY );

                SDL_RenderPresent( gRenderer );
            }     // kraj game loopa
        }
    }

    close();
    return 0;
}
```

Posebno treba istaknuti SDL_Event event strukturu. To je struktura koja pamti događaje od pritiska tipke na tipkovnici pa sve do događaja operacijskog sustava koji gasi aplikaciju. Različiti događaji zahtjevaju pamćenje različitih podataka stoga je ova struktura realizirana kao unija svih tipova događaja. [5] Za samo spremanje objekata neprijatelja, projektila i tornjeva odabrao sam spremnik std::list<> iz C++ standardne biblioteke nacrta (*eng. C++ Standard Template Library, kraće STL*). To je C++ implementacija dvostruko vezane liste, a budući da je u obliku nacrta (*eng. template*) to znači da povezuje objekte proizvoljna tipa koje navodimo u < i > zagradama. Prva while petlja koja provjerava zastavicu isRunning označava ulazak u glavnu petlju programa. Unutar glavne petlje programa ugnježdena je petlja događaja (*eng. event loop*) koja se uvijek izvodi ispred svih ostalih naredbi kako bi glavna petlja bila uvijek ažurna ovisno o neposredno nastalim događajima. Nakon provjere novo nastalih događaja slijedi logika igre koja obuhvaća sve kalkulacije udaljenosti objekata, kreaciju novih objekata ovisno o proteklom vremenu, pomicanje objekata i prilagođavanja kamere nivoa. Zadnje naredbe glavne petlje programa uglavnom

su spremanje precizno poredanih sličica (*eng. sprite*) prikazivač (*eng. renderer*) „gRenderer“ koji možemo zamisliti kao sakriveni okvir prozora. Naredbom SDL_RenderPresent(gRenderer) prikazujemo sadržaj stvorenog prikazivača u dodjeljenom mu prozoru.

Potrebno je napomenuti kako je redoslijed spremanja sličica u sakriveni okvir izrazito bitan – primjerice ukoliko bismo u okvir najprije spremili sliku pokazivača miša (u našem primjeru gKursorMisaPT), te zatim sliku nivoa, prilikom prikaza prvo bi se prikazala slika kursora pa zatim slika nivoa, no igrač bi primjetio kako je kursor miša nestao. Ovaj problem manifestacija je pogrešno poredanih slika prikaza unutar istog okvira. Sustav prikaza okvira je takav da ukoliko dođe do kolizije piksela unutar sakrivenog okvira posljednja slika prebrisati će piksele prethodne slike (osim ukoliko posljednja slika ima alfa, transparentni kanal, manji od brojčane vrijednosti 256). Glavna petlja programa prestaje se izvoditi ukoliko korisnik izazove događaj tipa SDL_QUIT. To je događaj operacijskog sustava koji programu šalje signal završetka rada, a korisnik ga može inducirati pritiskom na gumb x u gornjem desnom kutu prozora ili dobro poznatom kraticom ALT + F4. Aplikacija neposredno prije samog zatvaranja dealocira preostale dinamički alocirane resurse globalnom funkcijom void close().

3.2. Funkcije korištene unutar glavnog kostura programa

Kroz kostur glavnog programa koriste se razne funkcije od kojih su neke naredbe SDL biblioteke, kao što su naredbe koje počinju sa „SDL_“ prefiksom. No koriste se i korisnički definirane metode klase ProsirenaTextura: „bool loadScaledFromFile( )“ „void render( )“ te globalne funkcije „bool init( )“, „bool loadResources( )“, „void close( )“koje je potrebno malo detaljnije pojasniti.

3.2.1. Globalna funkcija bool init()

Void init( ) globalna je funkcija deklarirana unutar zaglavlja „glob_fje.h“ i definirana unutar „glob_fje.cpp“ izvorne datoteke izrađena po uzoru na [7]. Ova funkcija se prva izvršava unutar „main( )“ glavne funkcije naše aplikacije. Zadaća init funkcije jest pokrenuti SDL podsisteme te stvoriti globalni prozor i globalni prikazivač (*eng. renderer*).

```
extern const int DUZINA_EKRANA;  // variable definirane u drugim
extern const int VISINA_EKRANA;  // izvornim datotekama

extern SDL_Window* gWindow;
extern SDL_Renderer* gRenderer;

bool init() {
    bool uspjelo = true;
```

```
    if( SDL_Init( SDL_INIT_VIDEO | SDL_INIT_TIMER ) < 0 ){
        printf("Greska prilikom inicializacije SDL podsistema.\n");
        uspjelo = false;
    }
    else {
        gWindow = SDL_CreateWindow("Tower Defense Game",
SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, DUZINA_EKRANA,
VISINA_EKRANA, 0);
        if ( gWindow == NULL ) {
                printf("Greska prilikom stvaranja prozora!\n");
                uspjelo = false;
        }
        else {
            gRenderer = SDL_CreateRenderer( gWindow, -1,
SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC );
            if ( gRenderer == NULL ) {
                printf("Greska prilikom stvaranja renderera!\n");
                uspjelo = false;
            }
            else {
                SDL_SetRenderDrawColor( gRenderer, 0xff, 0xff, 0xff, 0xff);
                int imgFlags = IMG_INIT_PNG | IMG_INIT_JPG;
                if ( (IMG_Init( imgFlags ) & imgFlags) != imgFlags ) {
                    printf("Greska prilikom ucitavanja podsistema za PNG i
JPG slike: %s\n", IMG_GetError());
                    uspjelo = false;
                }
                if ( TTF_Init() == -1 ) {  // pokretanje podsistema za font
                    printf("Greska prilikom inicializacije SDL_ttf: %s\n",
TTF_GetError());
                    uspjelo = false;
    }   }   }   }

    return uspjelo;
}
```

Budući da je prozor „gWindow“ globalni pokazivač na strukturu prozora inicializiran unutar „main.cpp“ izvorne datoteke, kako bi ga mogli definirat unutar „glob_fje.cpp“ izvorne datoteke koristimo ključnu riječ extern ispred pokazivača „SDL_Window* gWindow“ te izostavljamo inicializaciju. Ključna riječ extern daje kompajleru do znanja da pokazivači na strukture gWindow i gRenderer postoje u drugim jedinicama prevođenja (*eng. translate unit*) stoga neka tretira ove pokazivače kao da se već sa njima susreo jer kasnije će program povezivač (*eng. linker*) spojiti sve jedinice prevođenja u izvršnu datoteku. Isti proces događa se sa globalnim variablama „DUZINA_EKRANA“ i „VISINA_EKRANA“. Kako bi aplikacija mogla koristiti naredbe SDL biblioteke najprije je potrebno uključiti potrebne SDL podsisteme pomoću naredbe SDL_Init ( ). Argumenti funkcije su predefinirane SDL zastavice (*eng. flags*) poput zastavice „SDL_INIT_VIDEO“ za podsistem prikazivanja slika te SDL_INIT_TIMER za podsistem računanja proteklog vremena unutar aplikacije. Ukoliko bismo željeli implementirati zvuk unutar naše aplikacije kao argument bi još naveli i zastavicu SDL_INIT_AUDIO. Zastavice međusobno povezujemo bitovnim OR operatorom zato što su ove zastavice dio istog pobrojenja (*eng. enumeration*). Ukoliko su svi željeni podsistemi inicializirani program SDL_CreateWindow( ) naredbom stvara strukturu s podatcima prozora

te ukoliko je naredba uspjela stvoriti prozor, u pokazivač „gWindow“ prosljeđuje se ispravan pokazivač na strukturu novo kreiranog prozora. Ukoliko se prilikom kreacije prozora dogodila greška, naredba SDL_CreateWindow( ) vraća NULL pokazivač. Nakon prozora naredba SDL_CreateRenderer(„vezani prozor“ , -1, „zastavice) stvara prikazivač. Prvi argument SDL_CreateRenderer naredbe jest prozor gdje će se prikazivač prikazivati kada nad tim prikazivačem pozovemo SDL_RenderPresent( ) naredbu. Sljedeći argument predstavlja indeks upravljačkog programa (*eng. driver*) koji će naredbe nad prikazivačem prosljeđivati grafičkoj kartici, a vrijednost -1 označuje da nam indeks upravljačkog programa nije bitan već da tim poslom zaduži prvi koji je slobodan. Naposljetku zadnji argument su zastavice SDL_RENDERER_ACCELERATED i SDL_RENDERER_PRESENTVSYNC. Prva zastavica označava stvaranje prikazivača sa hardverskim ubrzanjem (*eng. hardware acceleration*), drugim rječima, instrukcije novo stvorenog prikazivača, umjesto procesora, odrađivati će grafička kartica. Druga zastavica SDL_RENDERER_PRESENTVSYNC označuje frekvenciju osvježivanja prikazivača, kratica VSYNC (*eng. vertical sync*) označuje da će se prikazivač osvježiti istovremeno sa osvježivanjem ekrana – na većini današnjih ekrana to je 60Hz tj. 60 puta u sekundi. Zatim naredbom SDL_SetRenderDrawColor( ) određujemo boju ispune praznog prostora prikazivača, u našem slučaju to je bijela boja zadana prvim trima heksadekatskim vrijednostima „0xff“, koje su ekvivalentne broju 256 u dekatskom sustavu, a označuju maksimalnu vrijednost boje u 8bitnom prikazu. Posljednja „0xff“ vrijednost označuje vrijednost transparentnog alfa kanala (*eng. alpha channel*) koji ukoliko je bliže vrijednosti 0 slika postaje prozirnija. Zadnjom naredbom IMG_Init( ) vraća broj koji simbolizira zastavice koje je naredba uspjela učitati, dakle nama je potrebna JPG i PNG podrška stoga provjeravamo povratnu vrijednost naredbe te provjeravamo njezinu jednakost sa zadanim zastavicama. Naredbom TTF_Init( ) inicializiramo truetype font API [2].

3.2.2. bool loadScaledFromFile() metoda klase ProsirenaTextura

Bool loadScaledFromFile( ) jedna je od metoda klase ProsirenaTextura. Najčešće se poziva unutar loadResources( ) globalne funkcije prije ulaska u glavnu petlju igre kako bi sprječili eventualni zastoj igre uzrokovan učitavanjem multimedije igre.

```
bool ProsirenaTextura::loadScaledFromFile( std::string putanja, int duzina,
int visina){
    free();  // dealociranje postojeće strukture piksela
    SDL_Surface* originalnaSlika = IMG_Load( putanja.c_str() );
    if ( originalnaSlika == NULL ) {
        printf("PT Load Greska: %s\n", IMG_GetError() );
    }
    else { // izvorna slika je uspješno učitana u RAM memoriju
        SDL_Surface* resizeSlika = SDL_CreateRGBSurfaceWithFormat( 0,
duzina, visina, 32, SDL_PIXELFORMAT_RGBA32 );
        if ( resizeSlika == NULL ) {
```

```
printf("SDL_CreateRGBSurfaceWithFormat nije uspio alocirati SDL_Surface*
resizeSlika!\n");
        }
        else {  // preslikavanje izvorne slike u sliku zadanih dimenzija
            if ( SDL_BlitScaled( originalnaSlika, NULL, resizeSlika, NULL )
!= 0 ) {
                printf("PT BlitScaled Greska: %s\n", SDL_GetError() );
            }
            else {
                pTexture = SDL_CreateTextureFromSurface( gRenderer,
resizeSlika );
                if ( pTexture == NULL ) {
                    printf("PT createTexture greska prilkom pretvorbe
surface-a u teksturu!\n");
                }
                else {

                    pDuzina = resizeSlika->w;
                    pVisina = resizeSlika->h;
                }
                SDL_FreeSurface( resizeSlika );
        }   }
        SDL_FreeSurface( originalnaSlika );
    }
    return ( pTexture != NULL);
}
```

Metoda prima argumente relativne putanje koja ovisi o poziciji izvršne datoteke koja definira loadScaledFromFile metodu, a sljedeća dva argumenta su visina i dužina slike na koju izvornu sliku želimo smanjiti ili povećati. Prva naredba metode jest free( ) metoda iste klase koja provjerava pokazuje li privatni atribut klase tipa SDL_Texture* na neki alocirani resurs, te ukoliko pokazivač nije NULL , dealocira stari alocirani resurs, jer će kasnije pokazivač pokazivati na novi resurs alociran loadScaledFromFile metodom. IMG_Load( ) je naredba SDL2/SDL_image.h biblioteke koja alocira strukturu tipa SDL_Surface sa potrebnim podacima piksela tražene slike izvorne veličine u RAM memoriji te vraća pokazivač na novo alociranu strukturu. Nakon toga naredbom SDL_CreateRGBSurfaceWithFormat( ) stvara praznu površinu (*eng. surface*) sa 32bitnim prikazom piksela sa transparentnim kanalom sa dimenzijama koje je korisnik odredio. Ukoliko je uspješno stvorena prazna površina zadanih dimenzija, tada naredbom SDL_BlitScaled ( ) podatke originalne slike smanjuje na dimenzije novo kreirane prazne površine. Prvi NULL argument označava da želimo kopirati cijelu izvornu sliku, a drugi NULL argument da je želimo kopirati tako da zauzme čitav prazan prostor površine, ukoliko bismo željeli kopirati samo dio izvorne slike ili kopirati sliku u određeni dio novo kreirane površine tada umjesto NULL argumenata prosljeđujemo pokazivač na strukturu tipa SDL_Rect koja enkapsulira 4 atributa { x, y, w, h } koji označavaju x i y koordinate gornje lijeve točke pravokutnika i w – dužinu te h – širinu tog pravokutnika. Za sada u RAM memoriji imamo alociranu izvornu sliku i sliku izvornu sliku korsnički specificiranih dimenzija, no budući da naš prikazivač „gRenderer“ koristi hardversko ubrzanje tj. instrukcije nad prikazivačem obavlja grafička kartica bilo bi bolje kada bismo

našu sliku koja se trenutno nalazi u RAM memoriji, premjestili u VRAM memoriju specializiranu za izvršavanje operacija nad grafičkim elementima. Upravo u tu svrhu u SDL biblioteci postoji struktura SDL_Texture koja je identična strukturi SDL_Surface, no sa bitnom razlikom da tražene resurse umjesto u RAM alocira unutar VRAMa grafičke kartice. U našem slučaju privatan atribut klase ProsirenaTextura „pTexture“ je pokazivač na strukturu tipa SDL_Texture te mu naredbom SDL_CreateTextureFromSurface( ) prosljeđujemo prikazivač koji će ga prikazivati te sliku traženih dimenzija. Ukoliko je naša slika uspješno alocirana u VRAM memoriju postavljamo privatne atribute pDuzina i pVisina na dimenzije tražene slike. Budući da sada imamo sliku u VRAM memoriji izvorna slika i slika korisnički specificiranih dimenzija u RAM memoriji više ne služe nikakvoj svrsi te ih naredbom SDL_FreeSurface( ) dealociramo iz RAM memorije. Metoda je tipa bool te ukoliko alociranje slike u VRAM memoriju nije uspjelo naredba SDL_CreateTextureFromSurface( ) vraća NULL pokazivač, time ukoliko pTexture pokazuje na alociranu strukturu tada tvrdnja (pTexture != NULL) postaje istina te metoda vraća istinu kao naznaku uspješnog provođenja unutarnjih instrukcija.

3.2.3. Globalna funkcija bool loadResources()

Ukratko ovako izgleda globalna bool loadResources( ) funkcija, koja vraća false ukoliko nije uspjela učitati zadane resurse / multimediju aplikacije.

```
bool loadResources(){
    bool uspjelo = true;
    gFont = TTF_OpenFont("font/DejaVuMathTeXGyre.ttf", 24);
    if ( gFont == NULL ) {
        ...
        uspjelo = false;
    }
    if ( !neprijateljPT.loadScaledFromFile("slike/neprijatelj.png", 40, 40)) {
        ...
        uspjelo = false;
    }
    if ( !healthPT[START].loadScaledFromFile("slike/green/start.png", 20, 50) ) {
        ...
        uspjelo = false;
    }
    ...
return uspjelo;
}
```

Najprije otvara globalni font te ukoliko ga nije uspjela učitati u globalni pokazivač „gFont“ tipa TTF_Font* vraća neuspjeh. Isti postupak učitavanja izvršava i za dohvaćanje slika aplikacije uz pomoć ranije objašnjene metode: bool rosirenaTextura::loadScaledFromFile( ).

3.2.4.Globalna funkcija void close()

Nakon izlaza iz glavne petlje programa poziva se void close( ) globalna funkcija. Ona dealocira sve preostale alocirane resurse kako s prestankom aplikacije nebi nastavili zauzimati dane računalne resurse. Zatim oslobađa resurse koje je zauzimao globalni font te oslobađa resurse prozora i prikazivača. Posljednje naredbe koje funkcija izvodi su „_Quit( )“ naredbe kojima se redom gase glavni SDL podsistemi pokrenuti globalnom init( ) funkcijom pri samom pokretanju aplikacije.

```
void close(){
    // Tu dealociraj globalne objekte scene
    level.free();
    neprijateljPT.free();
    gKursorMisaPT.free();

    ...
    for (int i = 0; i < HEALTH_SPRITES_TOTAL; ++i) {
        healthPT[i].free();
    }
    TTF_CloseFont( gFont );
    gFont = NULL;
    SDL_DestroyWindow( gWindow ); // oslobodi resurse prozora
    gWindow = NULL;
    SDL_DestroyRenderer( gRenderer );  // oslobodi resurse prikazivača
    gRenderer = NULL;
    TTF_Quit();
    IMG_Quit();
    SDL_Quit();
}
```

3.2.5.void Render() metoda klase ProsirenaTextura

Void ProsirenaTextura::render( ) je metoda kojom prikazujemo teksture (slike u VRAM memoriji) na površinu prozora. Izrađena je po uzoru na [7]

```
void ProsirenaTextura::render(int x, int y, SDL_Rect* Clip, double kut,
SDL_Point* centar, SDL_RendererFlip zrcaljenje){

    SDL_Rect odrediste { x, y, pDuzina, pVisina};

    if ( Clip != NULL ) {
        odrediste.w = Clip->w;
        odrediste.h = Clip->h;
    }
    SDL_RenderCopyEx( gRenderer, pTexture, Clip, &odrediste, kut, centar,
zrcaljenje);
}
```

Metoda prima 6 argumenata. Prva 2, x i y označuju koordinate unutar prikazivača početnog gornjeg lijevog vrh slike koja će se na njemu prikazati. Clip je pokazivač na pravokutni okvir teksture koji će naredba SDL_RenderCopyEx „zalijepiti“ na prikazivač „gRenderer“. Ostala 3 argumenta definiraju rotacijski kut, centar rotacije i tip zrcaljenja. Naime nakon x, i y obaveznih argumenata, ostali argumenti se mogu ali i ne moraju navesti.

Ukoliko se ne navedu čitava tekstura će se zalijepiti unutar prikazivača bez rotacije i bez zrcaljenja. Treba napomenuti kako će sadržaj prikazivača biti vidljiv u prozoru aplikacije tek nakon SDL_RenderPresent( ) naredbe koja će sadržaj prikazivača prikazati unutar prozora.

3.3. Dizajn izdvojenih glavnih komponenta igre

U ovome dijelu detaljnije će biti opisane izdvojene glavne komponente koje zajedno čine izrađenu igru. Najprije ću opisati način implementacije kamere unutar igre koja se pomiče ovisno kako se kazaljka miša približava rubovima ekrana. Zatim pojasniti pokretne te mirujuće vizualne elemente igre. Nakon toga ukratko ću opisati implementaciju mjerača vremena i njegove metode koje se kasnije koriste za stvaranje neprijatelja u pravilnim vremenskim razmacima. Također ću objasniti i razloge odabira pojedinih spremnika C++ STL-a za implementaciju prohodnog puta i implementaciju neprijatelja unutar igre. Na kraju ću opisati i vizualno prikazati sustav prikaza životnih bodova igrača untar igre.

3.3.1. Implementacija kamere

Kamera nivoa pamti podatke pravokutnika { **koordinate** , **dužinu** , **visinu** } koji definira pravokutni prostor nivoa koji je trenutno vidljiv korisniku aplikacije. Kako su dimenzije nivoa veće od dimenzije prozora, a time i kamere, igrač u svakom trenutku može vidjeti samo djelić nivoa igre. Kako bi mogao odabrati dio koji želi gledati u klasu kamera dodane su metode za pomicanje kamere: handleCameraMouse( ) i adjustCameraBorder( ). Void handleCameraMouse( ) očitava nastale događaje te ukoliko je korisnik pomaknuo miš, u prosljeđene reference misX i misY sprema novu poziciju miša, dok metoda void adjustCameraBorder( ) pomiče pravokutni prostor pogleda nivoa (kameru) ovisno o koordinatama miša. U nastavku te metode biti će detaljnije analizirane.

```
#ifndef CAMERA_H
#define CAMERA_H

#include "MyWindow.h"
#include "ProsirenaTextura.h"

extern MyWindow gWindow;
extern ProsirenaTextura level;

class Camera{
  public:
    int x;  // koordinata pravokutne kamere
    int y;  // koordinata pravokutne kamere
    int w;  // dužina kamere
    int h;  // visina kamere
    bool cursorAtBorder;  // zastavica ruba kamere

    Camera();
```

```
    void handleCameraMouse( SDL_Event& e, int& misX, int&misY );
    void adjustCameraBorder( double forceX, double forceY, Uint32
timerTicks );
    double getForceX();
    double getForceY();


private:
    int moveSpace;   // odmak od ruba kamere (izražen u pikselima)
    double forceX, forceY;  // snaga pomicanja kamere
    double brzinaKamere;  // korak pomaka kamere (izražen u pikselima)

};

#endif
```

Nadalje u svakoj datoteci zaglavlja (*eng. header file*) , pa tako i Camera.h koriste se čuvari uključivanja zaglavlja (*eng. include guards*). Čuvari uključivanja zaglavlja razlikuju se od ostalih naredbi po tome što počinju sa # simbolom. U C++ jeziku naredbe koje počinju # simbolom naredba su predprocesoru (alatu zamjene koji neposredno prije kompajlera substituira pred procesorske naredbe i odlučuje koje dijelove koda će kompajler kasnije kompilirat (*eng. compile*)). [6] Naredba #ifndef prvotno provjeri postoji li defniran makro (*eng. macro*) naziva „CAMERA_H“ , ukoliko postoji kompajler će preskočiti blok koda obrubljen #endif naredbom. Ukoliko makro „CAMERA_H“ ne postoji taj blok koda kompajler prvi puta susreće te izvršava #define naredbu koja stvara makro „CAMERA_H“. Kada druga datoteka ponovno uključi ovo zaglavlje sada budući da je „CAMERA_H“ makro definiran , kompajler ga neće ponovo čitati te će time izbjeći dvostruku deklaraciju i definiciju klasa i funkcija unutar tog zaglavlja.

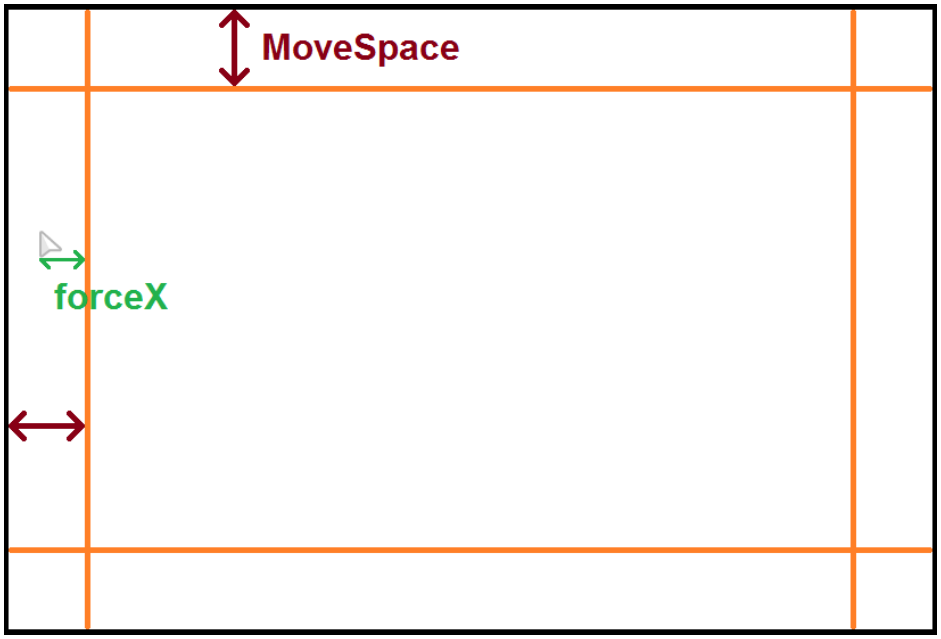
3.3.1.1. Metoda handleCameraMouse

Naredbe unutar ove metode izvršavati će se samo ukoliko je u metodu prosljeđen događaj pomaka miša. Nakon pomaka miša sa vrijednosti variable moveSpace koja dobiva svoju početnu vrijednost u pozivu konstruktora klase određujemo krajnje rubove prozora koji će pomicati kameru ukoliko korisnik pokazivač miša pomakne do ruba prozora. Vizualizacija ruba kamere prikazana je slikom broj 3. Dakle ukoliko je miš udaljen od ruba prozora za manje od moveSpace piksela, tada se, ovisno o količini prodora piksela u moveSpace prostor izračunava snaga pomaka kamere u smjeru prodora kursora miša. U ovom primjeru prikazani su izračuni za snagu horizontalnog pomaka kamere, no vertikalni pomak kamere slijedi istu prikazanu if, else if šablonu.

```
void Camera::handleCameraMouse(SDL_Event& e, int& misX, int& misY){
    if ( e.type == SDL_MOUSEMOTION ) {
        int cameraRightTolerance = DUZINA_EKRANA - moveSpace;
        int cameraLeftTolerance = moveSpace;
        int cameraTopTolerance = moveSpace;
        int cameraBottomTolerance = VISINA_EKRANA - moveSpace;
```

```
        SDL_GetMouseState( &misX, &misY ); // dohvaća koordinate kursora

        if ( misX > cameraRightTolerance ) {
            forceX = (misX - cameraRightTolerance +1) / (double)moveSpace;
            cursorAtBorder = true;
        }
        else if ( misX < cameraLeftTolerance ) {
            forceX = (misX - cameraLeftTolerance) / (double)moveSpace;
            cursorAtBorder = true;
        }
        ...

        if ( !((misX > cameraRightTolerance) || (misX <
cameraLeftTolerance) || (misY > cameraBottomTolerance) || (misY <
cameraTopTolerance)) ) {  // ako pokazivac misa nije na rubu prozora
            cursorAtBorder = false;
            forceX = 0;
            forceY = 0;
        }
    }
}
```

Ovisno o prodoru kamere lijevo forceX biti će u intervalu [-1, 0], odnosno prodorom s desna [0, 1]. U izračunu snage pomaka kamere forceX, moveSpace explicitno pretvaramo u decimalni double tip podataka jer bi u suprotnom C++ kompajler ovo dijeljenje interpretirao kao cijelobrojno, i dobili bi isti efekt kao da smo zadali da je moveSpace 1 piksel odmaka od ruba ekrana.



Slika 3: Vizualizacija granica kamere

3.3.1.2. Metoda adjustCameraBorder

BrzinaKamere i timerTicks uvijek su pozitivnih vrijednosti, dok je snaga pomaka kamere „forceX“ i „forceY“ pozitivna ili negativna, ovisno na kojim je trenutno koordinatama pokazivač miša. Ukoliko je miš lijevo i/ili gore tada su snage pomaka kamere negativne , a ukoliko se miš nalazi desno ili dolje tada su snage pomaka kamere pozitivne. Rezultat

umnažanja tih vrijednosti dodajemo na koordinate kamere x i y. Ukoliko je rezultat pozitivan x i/ili y koordinata će se povećati te ukoliko je negativan one će se smanjiti za broj piksela dobivenih umnoškom brzineKamere , forceX / Y i timerTicks variabli.

```
void Camera::adjustCameraBorder( double forceX, double forceY, Uint32
timerTicks ){
    x += brzinaKamere * forceX * timerTicks;  //nove koordinate kamere
    y += brzinaKamere * forceY * timerTicks;
// granice kamere
    if ( x < 0) {
        x = 0;
    }
    else if ( x + w > level.getDuzina() ) {
        x = level.getDuzina() - w;
    }
    if ( y < 0) {
        y = 0;
    }
    else if ( y + h > level.getVisina() ) {
        y = level.getVisina() - h;
    }
}
```

Izračun pomaka piksela mogli bismo računati i bez timerTicks variable ukoliko bismo uvijek garantirali, a time i ograničili korisnika na konstantnu frekvenciju osvježivanja ekrana (*eng. frames per second, kraće FPS*), primjerice na 60 FPSa, no u današnje doba opcija promjene frekvencije osvježivanja ekrana dio je osnovnog paketa opcija koje igra nudi i koje bi svaka igra trebala imati. Zbog promjenjive vrijednosti osvježivanja ekrana naša glavna petlja programa izvršiti će se više ili manje puta , ovisno o jačini računalne konfiguracije korisničkog računala na kojem se izvodi naša aplikacija. Brzina kamere jest fiksna vrijednost koja određuje osnovni broj pomaka piksela kamere, no aplikacija koja se osvježuje primjerice 100 puta u sekundi, znatno bi brže pomakivala kameru od aplikacije koja se osvježuje 60 puta u sekundi zato što će se programski kod metode kamere na računalu jačih performansa izvoditi dodatnih 40 puta. Kako bi izbjegli ovisnost kamere o broju osvježivanja ekrana koji ovisi o hardverskim komponentama korisnikova računala, koristimo timerTicks vrijednost koja označava broj proteklih milisekundi unutar jednog izvršavanja svih naredbi glavne petlje,a nakon očitanog mjerenja taj tajmer (*eng. timer*) pokrenemo iznova.

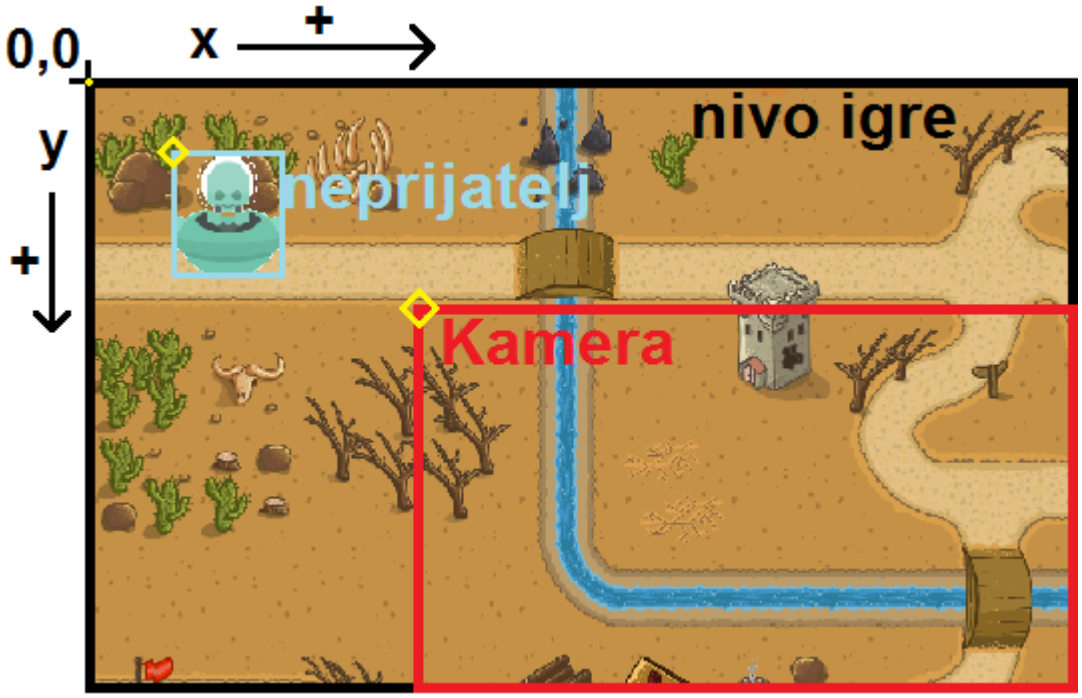
Problem pomicanja kamere neovisno o FPSu aplikacije je rješen, no sljedeći problem kamere jest trenutak kada kamera prođe koordinate nivoa. Korisniku aplikacije ovaj će se problem manifestirati kao nadolazeća bijela pozadina budući da smo praznu ispunu našeg prikazivača naredbom SDL_SetRenderDrawColor( ) zadali na bijelu boju. Kako bi izbjegli bijele piksele i nestanak slike nivoa, nakon izračuna koordinata kamere provjeravamo ukoliko su koordinate kamere unutar našega nivoa, ako nisu trenutne koordinate kamere korigiramo da se nalaze na granicama nivoa.

3.3.2. Prikaz pokretnih elemenata relativno prema poziciji kamere

Ovu igru izgrađuju pokretni i mirujući vizualni elementi – slike. Mirujući elementi – najčešće elementi korisničkog sučelja igre (*eng. user-interface, kraće UI*) uvijek su na istoj koordinatnoj poziciji prozora aplikacije, stoga njih prikazujemo na fiksno definiranim koordinatama. U izrađenoj igri to je element izbornika tornjeva i životnih bodova igrača. No dijelovi igre koji nisu dio korisničkog sučelja već su objekti unutar nivoa igre, budući da nivo igre nikad nije vidljiv u cijelosti, moraju se prikazivati relativno prema poziciji kamere. Zbog tog razloga metoda koja prikazuje neprijatelja koristi trenutne koordinate neprijatelja, ali i trenutne prihvaća argumente trenutnih koordinata kamere kako bi objekt neprijatelja prikazala relativno prema trenutnoj poziciji kamere.

```
void Enemy::prikazi( int camX, int camY ){
    sprite->render( (_position.x) - camX, (_position.y - sprite-
>getVisina()) - camY );
}
```

Relativan pomak prikaza objekta ostvaruje se substitucijom trenutne pozicije kamere od trenutne pozicije neprijatelja. Ukolko su koordinate lijevog gornjeg kuta kamere veće od koordinata neprijatelja, vrijednosti argumenata render metode biti će negativne te se taj neprijatelj neće prikazati unutar prozora aplikacije budući da su koordinate prvog piksela prikazanog u prozoru aplikacije, počevši od gornjeg lijevog kuta (0, 0) tj. nenegativne vrijednosti.



Slika 4: Vizualizacija koordinata i prikaz pogleda kamere

Žutom bojom označene su vršne točke elemenata čije vrijednosti spremamo u x i y koordinate odgovarajućeg objekta. Neprijatelj neće biti prikazan na ekranu budući da nije unutar okvira kamere.

```
Camera cam;
...
    SDL_Rect kamera = {cam.x, cam.y, cam.w, cam.h};

    level.render(0, 0, &kamera );  // prikaz nivoa igre
    for ( auto it = creeps.begin(); it != creeps.end(); ++it) {
        ...
        (*it).prikazi( cam.x, cam.y );
    }
}
```

Nivo igre također se prikazuje ovisno o poziciji kamere iako intuitivno znamo da bi nivo igre trebao biti statičan. Teren ćemo pomicati zato što iako koordinate kamere nisu konstantne, sadržaj kamere uvijek će se prikazivati iz lijevog gornjeg kuta prozora aplikacije, a budući da ne možemo pomicati prozor aplikacije ovisno o kameri, pomicati će se nivo igre i objekti unutar nivoa. Rezultat konačnog pogleda koji će se prikazati na prozor aplikacije prikazan je slikom broj 5.

Slika 5: Igračev pogled na igru

3.3.3. Kontrola vremenskog toka

Igra jest primarno vizualna aplikacija koja u preciznim vremenskim trenutcima mijenja svoj sadržaj – prikazane slike. Vremenska komponenta (*eng. timer*) omogućava nam kontrolu nad vremenskim tokom prikazivanja redoslijeda slika. Kontrola vremenskog toka izrađena je po uzoru na [7].

```
class MyTimer{

  public:
    MyTimer();
    void Start();       // pokreni vremensku komponentu
    void Pause();       // privremeno zaustavi vremensku komponentu
    void Unpause();
    Uint32 GetTicks();  // dohvati proteklo vrijeme
    ...

  private:
    Uint32 _mainTicks;   // izmjerene milisekunde
    Uint32 _storedTicks; // spremljene pauzirane milisekunde (ticks)
    ...
};
```

Vremenska komponenta počinje mjeriti vrijeme nakon poziva Start( ) metode,a svaki slijedeći poziv te metode resetira proteklo izmjereno vrijeme. Tajmer u svakom trenutku možemo zaustaviti Pause( ) metodom i ponovo pokrenuti Unpause( ) metodom. Metodom Uint32 GetTicks( ) saznajemo proteklo izmjereno vrijeme u mjernoj jedinici milisekunda. U samom projektu najčešće se koriste Start( ) i GetTicks( ) metoda koje će biti detaljnije opisane.

3.3.3.1. Metoda void Start( )

Ova metoda pokreće vremensku komponentu. Atributi koji spremaju vremenske vrijednosti _mainTicks i _storedTicks mijenjaju se striktno pozivom javnih metoda i unutar aplikacije ne postoje pozadinski procesi koji bi protekom vremena mijenjali njihov sadržaj, no MyTimer instanca ažurno prikazuje proteklo vrijeme igre precizno u svaku milisekundu.

```
void MyTimer::Start(){

    _mainTicks = SDL_GetTicks();
    _storedTicks = 0;

    _started = true;  // zastavica
    _paused = false;  // zastavica

}
```

To postiže SDL_GetTicks( ) naredbom iz SDL biblioteke koja unutar privatne variable _mainTicks sprema trenutno proteklo vrijeme igre koje je do tada zabilježila vremenska komponenta SDL-a koja se protekom vremena postepeno uvećava.

3.3.3.2. Metoda Uint32 GetTicks( )

Ova metoda vraća trenutno izmjereno proteklo vrijeme od zadnjeg pokretanja vremenske komponente. Budući da pozadinski tajmer SDL-a u pozadini mjeri proteklo vrijeme od pokretanja aplikacije, naša vremenska komponenta prilikom svojeg pokretanja spremila je tada trenutno izmjerene milisekunde SDL tajmera. Sada prilikom poziva

GetTicks( ) metode kako bi vratila vrijeme u milisekundama, vraća razliku novih milisekunda SDL tajmera sa _mainTicks prethodno spremljenim milisekundama.

```
Uint32 MyTimer::GetTicks(){
    if ( _started ) {
        if ( _paused ) {  // ukoliko je vremenska komponenta pauzirana
            return _storedTicks;
        }
        else { // ukoliko je vremenska komponenta pokrenuta
            return SDL_GetTicks() - _mainTicks;
        }
    }
    return 0;
}
```

3.3.4.Implementacija prohodnog puta

Neprijatelji slijede prohodan put koji im je dodijeljen putem konstruktora prilikom njihova nastanka. Put je zapravo vector spremnik SDL_Point struktura koje sadrže x i y koordinate.

```
struct path{
    vector< SDL_Point> node;
};

path walkPath[3];
loadWalkablePaths( walkPath );  // stvaranje čvorova puta
...
if ( timerSpawn.GetTicks() >= (500 - spawnCount)) {

    Enemy ufo(walkPath[ rand() % 3 ]);  // nasumično dodjeljivanje puta
    ufo.sprite = &neprijateljPT;
    creeps.push_back( ufo );      // dodavanje neprijatelja u creeps listu

spawnCount++;         // broj stvorenih neprijatelja
timerSpawn.Start();  // resetiranje tajmera
}
```

Za spremnik struktura odabrao sam std::vector<> koji je dio C++ STL-a zato što je nad vector spremnikom definiran [ ] operator (*eng. subscript operator*). Unutar igre odredio sam 3 prohodna puta stoga variabla path sadržava 3 vektora SDL_Point struktura. Kreacija novih neprijatelja izvršava se unutar if bloka koji se izvršava svakih 500milisekundu tj. 0.5 sekundi ,a svakim dodatno kreiranim neprijateljem igra ubrzava kreiranje neprijatelja za 1milisekundu. Neprijatelju se kao argument prosljeđuje prohodan put koji slijedi nasumično između 3 kreirana prohodna puta. Na samom kraju if bloka povećava se broj stvorenih neprijatelja i vrijednost timerSpawn tajmera vraća se na nulu.

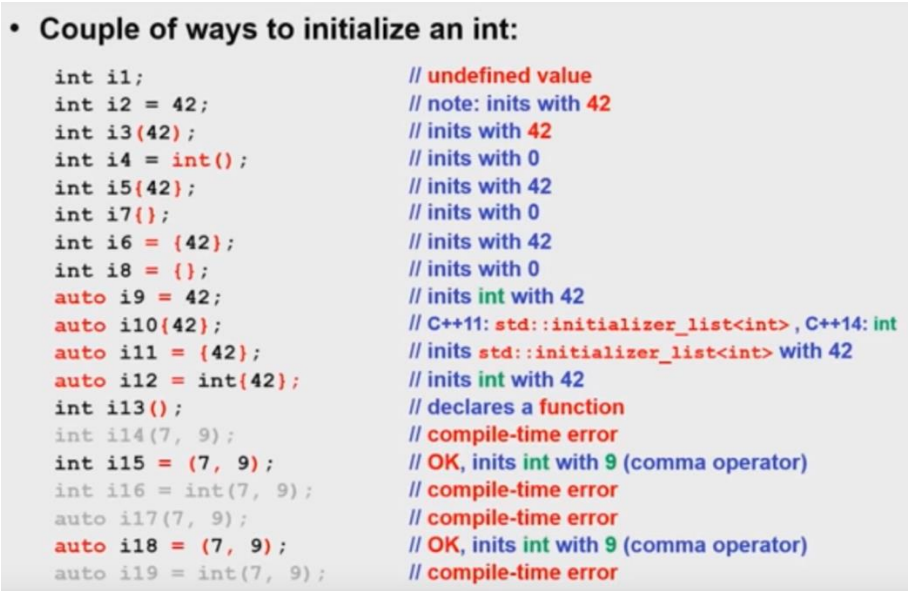
3.3.4.1. Globalna funkcija loadWalkablePaths

Put se definira unutar loadWalkablePaths( ) globalne funkcije. Budući da je proslijeđen pokazivač zapravo pokazivač na niz vektora puta pri početku definiramo reference imena prvi, drugi i treći radi lakšeg snalaženja prilikom dodavanja novih čvorova

puta. Čvor puta dodajemo preko .push_back() metode definirane unutar STL vectora kojoj prosljeđujemo listu inicializacije (*eng. initializer-list*) sa parom { x, y } koordinata. Točnije riječ je o kopirajućoj inicializaciji liste (*eng. copy-list-initialization*).

```
void loadWalkablePaths( path* put ){
    path& prvi = put[0];
    path& drugi = put[1];
    path& treci = put[2];
    prvi.node.push_back({ 0 , 320});  // kopirajuća lista inicializacije
    prvi.node.push_back({ 1075 , 320});
    prvi.node.push_back({ 1075 , 395});
    ...
    prvi.node.push_back({ -30 , 840});
    ...

}
```

Iako naoko jednostavna funkcionalnost C++ jezika, uz liste inicializacije vežu se procesi direktne inicializacije, kopirajuće inicializacije, inicializacije po vrijednosti (*eng. value initialization*) , inicializacije gomile (*eng. aggregate initialization*) , sužavanje tipa variabli (*eng. narrowing conversions*) te ovisno o C++ standardu korištenjem ključne riječi **auto** koja kompajleru naznačuje da otkrije (*eng. deduction*) tip variable naknadno iz danog konteksta, kompajler će za isti djelić koda kopajliranog standardom C++11 i C++14 deducirati različite tipove podataka stoga se liste inicializacije pretežito izostavljaju pri korištenju te ključne riječi. Iz priloženog je evidentna sama kompleksnost korištenja vitičastih zagrada kao sintakse c++ jezika za inicializaciju variabla, objekata i nizova, no ova funkcionalnost C++ jezika svakim novim C++ standardom uglađuje svoje neintuitivne slučajeve korištenja kako bi jednog dana objedinila sintaksu inicializacije C++ jezika koja zbog kompatibilnosti sa C jezikom još uvijek nije ujedinjena. [8] O samoj kompleksnosti inicializacije C++ jezika koja je ponekad nejasna i iskusnim C++ veteranima svjedoči slika 6.



Slika 6: Devetnaest načina inicializacije variable tipa int [8]

3.3.5. Implementacija neprijatelja

Klasa Enemy posjeduje nekoliko javnih metoda neke od kojih su konstruktor klase , walk() i getID() metoda. Osim metoda, javno sučelje Enemy klase posjeduje i pokazivač na objekt tipa ProširenaTextura naziva „sprite“ koji sadrži sličicu tog objekta.

```
class Enemy {
  public:
    Enemy( path& put);
    void walk();
    long getID() { return _id; }
    ...
    ProsirenaTextura* sprite;
  private:
    ...
}
```

Kada želimo pristupiti slici objekta i prikazati je na ekran jednostavno dereferenciramo pokazivač kako bi saznali podatke piksela potrebne za prikaz slike. Na ovaj način prilikom kreiranja novog objekta neprijatelja program ne duplicira sliku u memoriji, već se samo stvara pokazivač na postojeću sliku neprijatelja. Ovakav način realizacije neprijatelja ima svoje prednosti i mane. Naravno prednost ovakvog pristupa jest da je instanca Enemy klase kompaktnije veličine i brzo se stvara, dok primjerice kad bismo za svaku instancu stvarali novu zasebnu sliku , sam objekt klase zauzimao bi više prostora hrpe (*eng. heap space*) budući da objekte neprijatelja dinamički alociramo u programu te bi kreiranje objekta neprijatelja bilo bi dosta sporije od kreiranja objekta neprijatelja preko pokazivača. Negativna strana pristupa slici preko pokazivača je ta da sam objekt nije ujedno i vlasnik (*eng. owner*) te slike, prema tome ukoliko želimo promijeniti boju neprijatelja kojeg smo jednoznačno odredili putem long getID( ) metode u crvenu boju kako bi označili da je neprijatelj ranjen, dereferenciramo pokazivač na sliku kako bi dohvatili i promijenili piksele slike u memoriji, nakon promjene piksela slike, odjednom svi neprijatelji sada su crvene boje. Naime sam kod aplikacije je ispravan, no vidljivo daje krive rezultate, a problem predstavlja sam dizajn Enemy klase. Budući da svi objekti klase pokazuju na istu sliku u memoriji , kada se njezini pikseli promjene – bez obzira koji objekt ih je promijenio, svi objekti sada pokazuju na sliku s promjenjenim pikselima.

3.3.5.1. Konstruktor Enemy klase

Svaka klasa stvorena u C++ jeziku posjeduje svoj **konstruktor**. To je javna metoda istoimena nazivu klase, bez povratnog tipa koja se automatski poziva prilikom kreiranja novog objekta klase koja se može preopteretiti (*eng. overloading*) na više inačica od kojih svaka prima različite tipove i/ili broj argumenata.

```
class Enemy {
  public:
    Enemy( path& put);  // konstruktor Enemy klase
    void walk();  // metoda kretanja neprijatelja
    long getID() { return _id; }
    ...

  private:
    static int newID; // sakriveni djeljeni id neprijatelja
    long _id;  // id stvorenog neprijatelja
    SDL_Point _position;

    int _WalkNodeIndex;  // index čvora cilja
    path _path;  // nasumično proslijeđen put
}

Enemy::Enemy( path& put ){  // konstruktor klase
    _id = newID++;

    _path = put;
    _position.x = _path.node.front().x +1;
    _position.y = _path.node.front().y +1;

    _WalkNodeIndex = 1;
    _moveSpeed = 9;
    _health = 100;
    _done = false;
    ...
}
```

Namjena konstruktora jest inicializacija privatnih atributa klase na njihove početne vrijednosti. Konstruktor klase je bitan jer nam pruža izravan način definiranja privatnih varijabli klase koje su objektu klase tek neposredno dostupne ukoliko postoje metode javnog sučelja koje ih dohvaćaju ili mijenjaju, a nužne su za njegov ispravan rad. Primjerice, svaki neprijatelj (instanca Enemy klase) posjeduje svoj jedinstven i jednoznačan **_id** koji se koristi za navođenje projektila stvorenih unutar tornjeva ka neprijatelju. Jednoznačnost **_id**-a neprijatelja ostvarena je pomoću privatne statične variable **newID** koja je početno 0 i uvečava se za 1 svaki nakon kreacije novog objekta klase. Osim inicializacije **_id** variable, konstruktor klase svakom objektu dodjeljuje početne koordinate dohvaćene iz prvog čvora prohodnog puta te postavlja **_WalkNodeIndex** na indeks drugog čvora puta (indeksa 1) budući da se stvoreni neprijatelj već nalazi prvom čvoru puta.

3.3.5.2. Kretanje neprijatelja po prohodnom putu

Neprijatelji se po prohodnom putu kreću pomoću očitavanja koordinata sljedeće točne sadržane u vektoru puta. Ukoliko je udaljenost neprijatelja od čvor po X i Y koordinati 0, drugim riječima ukoliko se neprijatelj nalazi na koordinatama čvora koji mu je zadan kao cilj, tada mu se zadaje novi cilj uvećavanjem **_WalkNodeIndex** variable. Ukoliko je neprijatelj dospio do posljednje točke puta, igrač gubi jedan životni bod.

```
void Enemy::walk(){
...
    if ( udaljenostX == 0 && udaljenostY == 0 ) {
        _WalkNodeIndex++;

        if (_WalkNodeIndex >= _path.node.size() ) {
            _done = true;
            GameHealth -= 1; // životni bodovi igrača
...
    }
}
```

Kako je prije napomenuto da bi se neprijatelju dodjelila nova točka puta, neprijatelj mora dospjeti na točne koordinate prethodne točke, no neprijateljev korak nije zrnatosti jednog piksela, već jedan neprijateljev korak jest udaljenosti devet piksela određenih unutar Enemy javnog konstruktora. Kako bi neprijatelja pozicionirali na točne koordinate cilja, prije svakog koraka neprijatelja provjeravamo udaljenost do konačnog cilja po X i Y koordinati. Ako je udaljenost koordinata cilja manja od dužine neprijateljeva koraka, tada će se neprijatelj umjesto za udaljenost koraka, po toj koordinati pokrenuti samo za preostalu izračunatu udaljenost.

```
void Enemy::walk(){


    if ( _moveSpeed > udaljenostX ) {
        if (dst.x > _position.x ) {
            _position.x += udaljenostX;
        }
        else if ( dst.x < _position.x ) {
            _position.x -= udaljenostX;
        }
    }
    else {
        if (dst.x > _position.x ) {
            _position.x += _moveSpeed;
        }
        else if ( dst.x < _position.x ) {
            _position.x -= _moveSpeed;
        }
    }
}
```

U ovom primjeru prikazana je logika programskog koda za kretnju neprijatelja po osi apscisa, no kod za kretnju po osi ordinata slijedi identičnu if shemu prikazanu ovim programskim kodom.

3.3.5.3. Brisanje neprijatelja

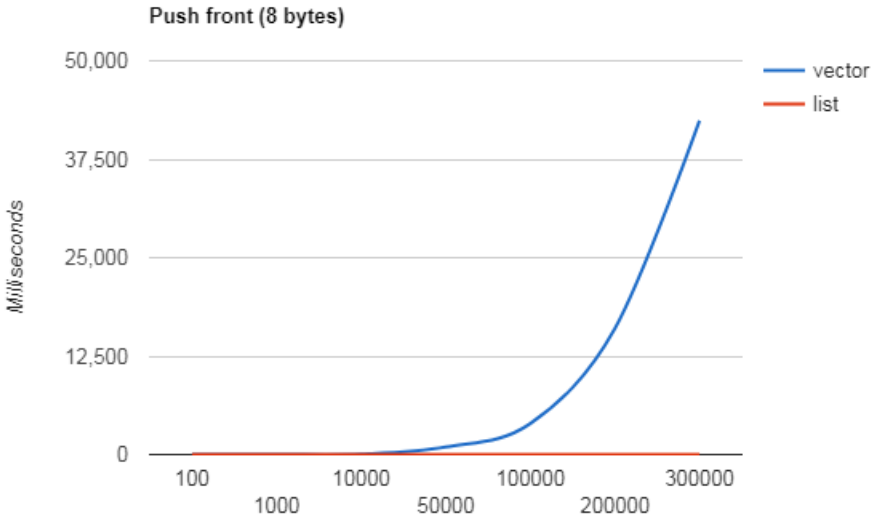
Brisanje neprijatelja posljedica je dvaju događaja. Neprijatelj se briše ukoliko je potrošio svoje životne bodove ili ukoliko je dospio do zadnje točke puta, odnosno kraja nivoa. Ispred same for petlje stvaramo iterator liste koji će pamtiti zadnji dobar pokazivač na element nakon što se obriše jedan element iz liste. For petlja inicializira it variablu sa ključnom riječju „auto“ što je uputa kompajleru da tip it variable zaključi iz danog mu

konteksta. Dani kontekst je creeps.begin( ) tj. adresa prvog elementa liste te tom informacijom kompajler it variabli pridružuje tip std::list<Enemy>::iterator koji smo mogli i sami napisati, no ovako je kod čitljiviji. For petlja će uvećavati iterator liste sve dok ne naiđe na posljednji element liste tj. creeps.end( ). Unutar for petlje it iterator koji u ovom slučaju možemo interpretirati kao pokazivač koji pokazuje na adresu elementa liste, moramo najprije dereferencirati kako bismo pristupili javnim atributima i metodama elementa. Dereferenciranje u C++ jeziku možemo obaviti sa sintaksom C jezika: (*it).isDone( ) ili novijom C++ sintaksom: it ->isDone( ). IsDone( ) metoda vraća stanje privatne „_done“ zastavice neprijatelja, te ukoliko je neprijatelj zbog nekog razloga označen za brisanje, tada će mu zastavica biti postavljena na true vrijednost. Ukoliko je to slučaj before iterator postavljamo na prethodni element kako bi trenutni mogli obrisati. Nakon brisanja it iteratoru prosljeđujemo adresu na koju pokazuje before iterator jer operacija brisanja invalidira adresu na koju je it iterator pokazivao prije brisanja. Ovdje postoji rubni slučaj koji nije u potpunosti razrješen ovim algoritmom. Ukoliko je neprijatelj kojeg želimo obrisati prvi element liste, before naredba trebala bi dobiti adresu njemu prethodnog elementa, no budući da je to prvi element liste ovaj programski kod nebi trebao raditi, međutim program radi bez greške. Razlog tome je taj što prvi element liste zapravo ima svoga prethodnika, a to je glava liste jer bez glave liste na prostoru stoga (*eng. stack space*) ne možemo ni dinamički alocirati ostatak elemenata liste. Srećom glava liste ne posjeduje _done zastavicu, već sadrži podatke poput veličine liste te će naša provjera it->isDone( ) izvršena nad prosljeđenom glavom liste rezultirati lažnom tvrdnjom (*eng. false statement*) i algoritam će nastaviti sa radom bez smetnji. Sljedeći programski kod prikazuje brisanje elemenata C++ STL list <> spremnika unutar for petlje.

```
std::list<Enemy>::iterator before;
for ( auto it = creeps.begin(); it != creeps.end(); ++it) {
    if ( it->isDone() ) {
        before = prev(it, 1);  // before pokazuje na prethodni element
        creeps.erase( it );    // brisanje trenutnog elementa
        it = before;
    }
    else {  // zato jer lista kad se unisti na pocetku, sljedeci element na
koji pokazuje je zapravo glava liste sa vrijednosti "size"
        (*it).prikazi( cam.x, cam.y );
    }
}
```

Lista kao tip spremnika neprijatelja odabrana je s namjerom maksimiziranja performansa igre. Liste za razliku od vector spremnika, zbog načina spremanja i povezivanja elemenata, pogodne su za brisanje elemenata neovisno o poziciji u listi koju je taj element zauzimao neposredno prije njegova brisanja. Toranj koji brani teritorij od valova neprijatelja u svakom trenutku može ubiti neprijatelja u njegovoj neposrednoj blizini te ovisno o poziciji na koju je toranj postavljen, taj neprijatelj može biti na bilokojoj poziciji unutar liste. Budući da

program svejedno mora proći kroz cijelu listu kako bi prikazao svakog neprijatelja zasebno, najprije provjeri zastavicu pa zatim odluči hoće li obrisati ili prikazati tog neprijatelja. Posljednji razlog odabira STL liste za spremnik neprijatelja je taj da performanse liste ne ovise o veličini pojedinog elementa, dok vektor porastom veličine pojedinog elementa gubi na svojim performansama. Slika broj 7. prikazuje razliku performansa prilikom nasumičnog brisanja 1000 vrijednosti unutar oba spremnika.



Slika 7: Performanse vektora i liste prilikom nasumičnog brisanja elemenata [9]

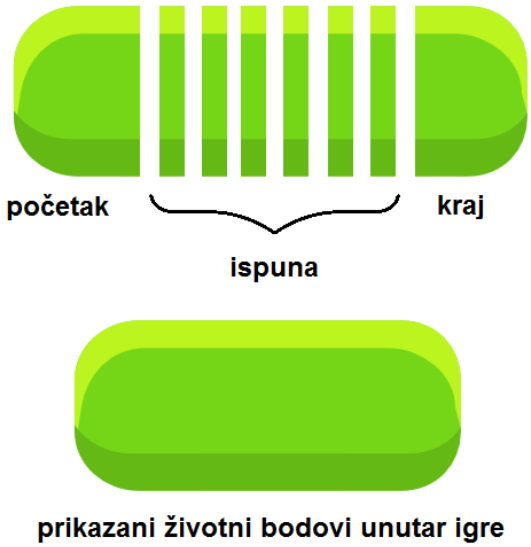
3.3.6. Implementacija životnih bodova igrača

Posljednja komponenta koja objedinjuje čitavu igru jest prikaz životnih bodova igrača. Ova komponenta ostvarena je globalnom funkcijom void renderGameHealth( ) i globalnom variablom GameHealth koja sadrži igračeve trenutne životne bodove.

```
enum Health{
    START,
    FILL,
    END
};

void renderGameHealth(){
    int startW = healthPT[START].getDuzina(), fillW =
healthPT[FILL].getDuzina();

    if ( GameHealth > 0) {
        healthPT[START].render(0, 0);       // početak
        int i = 0;
        for ( ; i < GameHealth-1; ++i) {

            healthPT[FILL].render( startW + i * fillW , 0); // ispuna
        }
        if ( GameHealth > 0 ) {
            healthPT[END].render( startW+i * fillW, 0);  // kraj
        }
```

```
    }
}
```

Void renderGameHealth poziva netom prije poziva zadnje naredbe render dijela glavne petlje – naredbe SDL_RenderPresent( ) zato jer ne želimo da se neka novo stvorena slika prikazuje ispred životnih bodova igre. Kako ne bi morali pamtiti indeks healthPT niza za svaku sliku posebno, uvodimo Health pobrojenje. If blokovi prikazuju slike unutar prozora aplikacije ukoliko igrač ima pozitivne životne bodove.



Slika 8: Vizualizacija prikaza životnih bodova igrača

Primjerice ukoliko je igraču preostao zadnji životni bod, tada prikazujemo samo početak i kraj healthPT teksture, a za svaki dodatni život unutar igre prikazujemo sliku ispune. Princip rada ove globalne funkcije jasnije se vidi na slici broj 8.

4. Zaključak

U okviru ovog rada izrađena je računalna igra obrane tornjeva u C++ programskom jeziku proširenim SDL 2.0 dinamičkom bibliotekom. Za razvoj aplikacije odabran je C++ jezik zato što podržava objektno orijentirani stil programiranja i kompatibilnost s C programskim jezikom zbog koje u C++ programski kod možemo integrirati naredbe SDL biblioteke koje koristimo za čitanje i upravljanje događajima budući da je SDL biblioteka u potpunosti pisana u C jeziku. Nadalje C++ jezik pruža i STL spremnike podataka različitih namjena od kojih su neki korišteni za spremanje objekata unutar igre.

Prilikom razvoja računalne igre spoznao sam da su funkcionalnosti koje SDL biblioteka nudi osmišljene s minimalnim otiskom (*eng. footprint*) na resurse računala, što i nije na iznenađenje budući da je čitava biblioteka pisana u C programskom jeziku koji je namjenjen razvoju aplikacija visokih performansa. Korištenjem STL vector i list spremnika primjetio sam njihove sličnosti i razlike, a i namjenu zbog kojih se koriste. Primjerice kako je vector spremnik sporiji prilikom brisanja podataka sa početka ili sredine spremnika te kako list spremnik ne podržava [] operator pristupa. Zbog navedenih spoznaja zaključujem kako projekt realiziran C++ jezikom uz korištenje SDL 2.0 biblioteke posjeduje kvalitete za izradu stabilnih aplikacija brzih performansa, a sam programski jezik omogućava visoku fleksibilnost u izradi aplikacija različitih namjena, u ovom projektu ispitanih kroz izradu računalne igre.

Popis literature

[1] Clang frontend for LLVM (2019.), dostupno na: https://clang.llvm.org , dana 7.9.2019.

[2] Simple Directmedia Layer (2019.), dostupno na: https://www.libsdl.org , dana: 6.9.2019.

[3] Ryan Gordon (2014.), „Game Development with SDL 2.0 (Steam Dev Days 2014)“ dostupno na: https://www.youtube.com/watch?v=MeMPCSqQ-34 , dana: 6.9.2019.

[4] Wikipedija (2019.), Tower Defense, dostupno na: https://en.wikipedia.org/wiki/Tower_defense, dana: 6.9.2019.

[5] SDL Wiki (2019.), dostupno na: https://wiki.libsdl.org, dana: 6.9.2019.

[6] Tutorials Point (2019.), „C Preprocessors“, dostupno na: https://www.tutorialspoint.com/cprogramming/c_preprocessors.htm , dana: 6.9.2019.

[7] LazyFoo (2019.), „Getting an Image on the Screen“, dostupno na: http://lazyfoo.net/tutorials/SDL/02_getting_an_image_on_the_screen/index.php , dana: 6.9.2019.

[8] Nicolai Josuttis (2018), „The Nightmare of Initialization in C++“, CppCon 2018 conference, Aurora, Colorado, USA , dostupno na: https://www.youtube.com/watch?v=7DTlWPgX6zs , dana 7.9.2019.

[9] Baptiste Wicht (2012.), „C++ benchmark – std::vector vs std::list“, dostupno na: https://baptiste-wicht.com/posts/2012/11/cpp-benchmark-vector-vs-list.html , dana 10.9.2019.

Popis slika