# Izrada 2D pokretača igre za računalnu igru izrađenu u C++-u i SDL-u

**Gazdek, Leonardo**

**Undergraduate thesis / Završni rad**

**2021**

*Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj:* **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

*Permanent link / Trajna poveznica:* https://urn.nsk.hr/urn:nbn:hr:211:819096

**SVEUČILIŠTE U ZAGREBU**
**FAKULTET ORGANIZACIJE I INFORMATIKE**
**V A R A Ž D I N**

Leonardo Gazdek

# IZRADA 2D POKRETAČA IGRE ZA RAČUNALNU IGRU IZRAĐENU U C++-U I SDL-U

**ZAVRŠNI RAD**

**Varaždin, 2021.**

**SVEUČILIŠTE U ZAGREBU**
**FAKULTET ORGANIZACIJE I INFORMATIKE**
**V A R A Ž D I N**

**Leonardo Gazdek**

**Matični broj: 0016135816**

**Studij: Informacijski sustavi**

# IZRADA 2D POKRETAČA IGRE ZA RAČUNALNU IGRU IZRAĐENU U C++-U I SDL-U

**ZAVRŠNI RAD**

**Mentor:**

Doc. dr. sc. Mario Konecki

**Varaždin, rujan 2021.**

*Leonardo Gazdek*

**Izjava o izvornosti**

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

*Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi*

**Sažetak**

Ovaj rad bavi se izradom 2D pokretača za izradu video igara koristeći C++ i Simple DirectMedia Library (SDL). SDL je višeplatformska biblioteka koja omogućava pristup niske razine računalnom hardveru za zvuk i grafiku te ulazno-izlaznim uređajima. Prilikom izrade pokretača za igre potrebno je riješiti neke bitne probleme poput delta vremena, limitiranja broja sličica po sekundi, izrade formata datoteka koje igra može učitavati te koji se alatima mogu uređivati, fizike, sučelja… Također je bitno rukovanje memorijom jer se igra vrti u glavnoj petlji a to znači da ako slučajno nešto alociramo u svakom koraku petlje i zaboravimo to osloboditi, memorija računala jako će se brzo popuniti. Ovaj pokretač sam po sebi nije video igra već alat za lakšu izradu video igre. Cilj pokretača je da poštedi programera trivijalnih zadataka i izračuna. Treba izraditi određenu razinu apstrakcije koristeći funkcije ugrađene u SDL biblioteku. Ova apstrakcija pomaže programerima koji se bave razvojem video igara da svoj posao rade brže i efikasnije.

Ovaj pokretač lako bi se mogao prilagoditi za više platforma jer je SDL sam po sebi višeplatformska biblioteka te se u kodu ovog pokretača koriste samo mogućnosti definirane u C++17 standardu. Projekt nije vezan niti za jednu platformu.

**Ključne riječi:** 2D, pokretač, igra, računalo, razvoj, sdl, c++, alat

# Sadržaj

# 1. Uvod

Tema ovog rada je izrada 2D pokretača za igre u C++u i SDL-u. Svaka videoigra razvijena je u nekoj vrsti pokretača za igre. Neke razvojne tvrtke razvijaju svoje pokretače dok se neke tvrtke okreću već postojećim rješenjima. Trenutno su najpopularniji pokretači za igre Unreal Engine i Unity (Perforce, 2020.).

Neke od komponenata koje pokretač igara mora imati su:

- Ulazni sustav – korisnik mora imati način da komunicira s igrom te se na računalu to uglavnom radi pomoću miša i tipkovnice. Pokretač igre mora imati način da uhvati događaje poput pritiska tipke na tipkovnici, mišu, itd. (Study tonight, 2021.).
- Grafika – pokretač mora imati način da crta (renderira) ono što se događa u igri na korisnikov ekran. Većina modernih pokretača podržava napredne grafičke mogućnosti poput praćenja zraka (ray tracing), svjetlosnih efekata, sjena… U ovom radu riječ je o jednostavnom pokretaču koji podržava samo 2D grafiku (Study tonight, 2021.).
- Fizika – omogućuju relativno realističnu predodžbu svijeta u igri. Gravitacija, kolizije, rotacije, itd. dio su sustava fizike u pokretaču igre (Study tonight, 2021.).
- Zvuk – pokretač mora znati komunicirati sa zvučnom karticom i reproducirati zvučne efekte (Study tonight, 2021.).

Motivacija iza ove ideje je ta što me oduvijek zanima razvoj računalnih igara te mi je C++ jezik koji mi oduvijek najbolje leži. Odlučio sam se na 2D pokretač, a ne 3D jer je razvijanje 3D pokretača mnogo kompleksnije te smatram da je to malo preteška tema za završni rad.

# 2. Metode i tehnike rada

SDL (Simple DirectMedia Layer) višeplatformska je biblioteka koja omogućava pristup niske razine računalnom hardveru za zvuk i grafiku te ulazno-izlaznim uređajima (libsdl.org, 2021.). U ovom radu, ta će se biblioteka koristiti za praktički sve – kreiranje prozora, renderiranje 2D grafike, upravljanje kontrolama (miš i tipkovnica)… Američka tvrtka za video igre Valve koristi SDL u svrhu kompatibilnosti s većim brojem platformi. S obzirom na to da velika tvrtka ovisi o ovoj biblioteci, ona je jako dobro podržana te je zadnja stabilna verzija izdana u prosincu 2020. godine (libsdl.org, 2021.).

U ovom radu koristit će se zadnja stabilna verzija za vrijeme izrade rada – 2.0.14. SDL biblioteka pisana je u programskom jeziku C, ali radi u C++-u bez potrebe za ikakvim prilagodbama. Ovaj rad biti će pisan u programskom jeziku C++ te će glavni fokus biti na Windows platformi, iako se u teoriji ovaj rad može lako prilagoditi da radi na ostalim platformama kao što su Linux i macOS. Za renderiranje na Windowsu koristit će se Direct3D jer ga SDL podržava. Iako ime Direct3D sugerira da se radi o 3D grafici, taj API također je odgovoran i za 2D grafiku. Na Windows platformi podržan je i OpenGL, ali u ovom radu odabrat ćemo Direct3D renderer iz razloga što je obično on bolje podržan na grafičkim driverima za Windows, iako to nužno ne mora biti tako. S obzirom na to da SDL nudi dosta visoku razinu apstrakcije za grafički API, nije pretjerano bitno koji renderer odaberemo.

Kao IDE u ovom radu koristit će se Visual Studio 2019 te njegov standardni kompajler (Microsoft C++ Compiler – MSVC). Razlog tome su dobra podrška i alati od strane Microsofta kao što je npr. debugger.

# 3. Opseg rada

## 3.1. Pokretač igre

Pokretač igre zadužen je za najosnovnije stvari od kojih se sastoji video igra. On se mora brinuti za glavnu petlju (game loop). Glavna petlja je jedan od najvažnijih aspekta programiranja video igara. Ona je zadužena za svaku sličicu (frame) koju igra prikazuje te je broj iteracija te petlje u sekundi jednak FPS-u (sličice po sekundi) kojim se igra pokreće. Što je bolje optimizirana glavna petlja neke igre, to će više sličica po sekundi igra moći ostvariti. Glavna petlja također je zadužena za snimanje korisničkih unosa u svakom trenutnu te je zadužena za reguliranje brzine igre. Zapravo se skoro cijeli programski kod igre nalazi unutar glavne petlje, iako će kod biti odvojen u različite datoteke, točnije klase.

### 3.1.1. Inicijalizacija Simple DirectMedia Layer-a

Da bismo koristili SDL, prvo ga moramo inicijalizirati pomoću ugrađene funkcije SDL_Init koja kao argument prima zastavice. U ovom pokretaču koristi se SDL_INIT_EVERYTHING.

```
if (SDL_Init(SDL_INIT_EVERYTHING) < 0) {
        std::cout << "SDL init failed: " << SDL_GetError() <<
std::endl;
        return 1;
}
```

SDL_Init vraća 0 ako je inicijalizacija uspješna ili vraća negativni kod greške prilikom neuspješne inicijalizacije (SDL Wiki, 2021.).

```
#define SDL_INIT_EVERYTHING ( \
                SDL_INIT_TIMER | SDL_INIT_AUDIO | SDL_INIT_VIDEO |
SDL_INIT_EVENTS | \
                SDL_INIT_JOYSTICK | SDL_INIT_HAPTIC |
SDL_INIT_GAMECONTROLLER | SDL_INIT_SENSOR \
            )
```

Iz ove definicije vidimo da SDL_INIT_EVERYTHING inicijalizira podsustave za timer (korisno za računanje delta vremena i za limitiranje FPS-a), zvuk, grafiku, događaje (poput pritiska tipke na tipkovnici ili zatvaranja programa), joystick, haptiku (korisno za kontroliranje npr. vibracije na joysticku), igrače kontrolere (ekstenzija joysticka) te senzoriku (npr. akcelerometar u mobilnom uređaju).

### 3.1.2. Prozor i renderer

Nakon inicijalizacije SDL-a, potrebno je kreirati prozor (SDL_Window) i renderer (SDL_Renderer). Za te radnje koriste se funkcije SDL_CreateWindow i SDL_CreateRenderer. Razlog zbog kojeg SDL ima funkciju za kreiranje prozora je taj što je SDL višeplatformska biblioteka, a kreacija prozora se izvodi drugačije na svakoj platformi.

```
SDL_Window* window = NULL;
window = SDL_CreateWindow("Engine", SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED, resX, resY, SDL_WINDOW_SHOWN);


if (!window) {
   std::cout << "Window creation failed: " << SDL_GetError() <<
std::endl;
   return 1;
}
```

Kao što je vidljivo u kodu za kreiranje prozora, funkcija SDL_CreateWindow prima 6 argumenata: naziv prozora, X pozicija, Y pozicija, X rezolucija, Y rezolucija te zastavice za razne postavke u prozoru (SDL Wiki, 2021.). Funkcija vraća pokazivač tipa SDL_Window* što znači da se u funkciji odvija dinamička alokacija memorije. Ako kreacija prozora iz nekog razloga ne uspije, SDL_CreateWindow vratit će nul-pokazivač. S obzirom da ovo nije statička alokacija, na kraju je potrebno osloboditi memoriju.

```
SDL_DestroyWindow(window);
```

U ovom pokretaču funkcija SDL_DestroyWindow poziva se prilikom izlaska iz igre. S obzirom na to da prozor mora biti aktivan kroz cijelo vrijeme pokretanja igre uništavanje prozora moglo bi se izostaviti. Većina operacijskih sustava automatski oslobađa memoriju programa prilikom izlaska. Ipak, uvijek je dobro eksplicitno očistiti memoriju.

Renderer u SDL-u je, najlakše objašnjeno, površina unutar prozora nad kojom se vrše radnje crtanja (renderiranja). Sve što se crta na renderer ima hardversku akceleraciju – to znači da se radnje crtanja odvijaju na grafičkoj procesorskoj jedinici (GPU).

```
SDL_Renderer* renderer = NULL;
renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED |
SDL_RENDERER_TARGETTEXTURE);
```

Funkcija SDL_CreateRenderer prima 3 argumenta – prozor na koji se spaja, indeks pokretača (drivera) te zastavice za opcije. Ako se za indeks stavi -1, renderer će koristiti prvi pokretač koji podržava navedene zastavice (SDL Wiki, 2021.). U ovom pokretaču koristimo dvije zastavice – SDL_RENDERER_ACCELERATED koji eksplicitno navodi da se koristi hardverska akceleracija te SDL_RENDERER_TARGETTEXTURE koji eksplicitno navodi da ovaj renderer podržava crtanje tekstura.

Isto kao što vrijedi i za prozor, nakon što smo gotovi s rendererom dobro bi bilo osloboditi memoriju.

```
SDL_DestroyRenderer(renderer);
```

### 3.1.3. Površina i tekstura

SDL-u za crtanje postoje dvije bitne strukture, a to su površina (SDL_Surface) i tekstura (SDL_Texture). Glavna razlika između te dvije strukture je ta što se površina crta softverski na centralnoj procesorskoj jedinici (CPU), dok se tekstura crta hardverski na grafičkoj procesorskoj jedinici (GPU). S obzirom na to da je ranije spomenuto da renderer koristi hardversku akceleraciju to znači da na renderer ne možemo nacrtat površinu, već samo teksturu. Rješenje za taj problem je jednostavno – površina se može pretvoriti u teksturu pomoću funkcije SDL_CreateTextureFromSurface.

```
SDL_Surface *surface = IMG_Load("textures/character.png");
SDL_Texture *texture = SDL_CreateTextureFromSurface(renderer, surface);
```

Gornji kod je pravilan primjer korištenja funkcije SDL_CreateTextureFromSurface. Ona prima samo dva argumenta – renderer na koji crta te površinu iz koje generira teksturu. Ovaj primjer također koristi funkciju IMG_Load koja je dio službene SDL biblioteke SDL_Image, više o tome kasnije.

Da bi se na renderer nacrtala tekstura, potrebno je pozvati funkciju SDL_RenderCopy koja prima 4 argumenta – renderer na koji se crta, tekstura koju treba nacrtati, SDL_Rect koji označava koji dio teksture treba nacrtati (NULL ako želimo nacrtati cijelu teksturu) i SDL_Rect koji označava na koji dio renderera treba nacrtati teksturu (NULL ako želimo crtati preko cijele površine renderera). Struktura SDL_Rect sadrži četiri atributa – x pozicija, y pozicija, širina i visina. SDL_Rect zapravo predstavlja pravokutnik.

```
SDL_Rect destRect = SDL_Rect {100, 100, 100, 100};
SDL_RenderCopy(renderer, texture, NULL, &destRect);
```

Nakon korištenja teksture i površine pametno bi bilo osloboditi memoriju.

```
SDL_FreeSurface(surface);
SDL_DestroyTexture(texture);
```

### 3.1.4. SDL_Image i SDL_TTF

Uz sam SDL, u ovom projektu koriste se i dvije službene SDL biblioteke, a to su SDL_Image i SDL_TTF.

SDL_Image služi da bi se mogli crtati razni formati slika, npr.: png, jpg, gif… SDL sam po sebi podržava samo bitmape i iz tog razloga postoji potreba za ovom bibliotekom. Biblioteku SDL_Image prvo treba inicijalizirati prije korištenja.

```
int imgFlags = IMG_INIT_PNG | IMG_INIT_JPG;
if (!(IMG_Init(imgFlags) & imgFlags)) {
   std::cout << "SDL_image init failed: " << IMG_GetError() <<
std::endl;
   return 1;
}
```

Za inicijalizaciju koristi se funkcija IMG_Init koja kao parametar prima zastavice koje označavaju podržane formate slike. Moguće zastavice su IMG_INIT_PNG, IMG_INIT_JPG, IMG_INIT_TIFF i IMG_INIT_WEBP. Za naš pokretač koristit ćemo samo png i jpg. Funkcija IMG_Init vraća zastavice koje su se inicijalizirale. Iz tog razloga koristimo operator I (AND - & u C++). Ako vraćene zastavice i naše zadane zastavice spojimo I operatorom, prilikom uspješne inicijalizacije taj cijeli izraz trebao bi biti istinit.

Nakon inicijalizacije korištenje biblioteke SDL_Image je jako jednostavno – koristi se funkcija IMG_Load koja prima putanju do slike te vraća pokazivač na SDL_Surface.

```
SDL_Surface *surface = IMG_Load("textures/character.png");
```

Biblioteka SDL_TTF koristi se da bismo mogli crtati tekst koristeći TrueType fontove. Biblioteku SDL_TTF treba inicijalizirati prije korištenja.

```
if (TTF_Init() == -1) {
   std::cout << "SDL_TTF init failed: " << TTF_GetError() << std::endl;
   return 1;
}
```

Funkcija TTF_Init ne prima argumente te vraća -1 prilikom neuspješne inicijalizacije. Prilikom uspješne inicijalizacije vraća 0 (libsdl.org, 2009.).

Nakon inicijalizacije treba učitati font u strukturu TTF_Font.

```
TTF_Font *font = TTF_OpenFont("fonts/arial.ttf", 32);
```

Funkcija TTF_OpenFont prima dva argumenta: putanju do .ttf fonta te veličinu fonta za crtanje. Ona vraća pokazivač tipa TTF_Font* te može vratiti nul-pokazivač ako je učitavanje neuspješno.

Nakon učitavanja fonta preostaje nam napraviti površinu koristeći taj font.

```
SDL_Surface *textSurface = TTF_RenderText_Solid(font, "Tekst za ispis",
SDL_Color {0, 0, 0, SDL_ALPHA_OPAQUE});
```

Funkcija TTF_RenderText_Solid prima tri argumenta: pokazivač tipa TTF_Font*, tekst za crtanje te boju teksta (struktura SDL_Color koja sadrži količinu crvene, zelene i plave boje te alpha parametar koji može biti od 0 do 255 za prozirnost). Funkcija vraća površinu koju kasnije možemo direktno nacrtati ili pretvoriti u teksturu i nacrtati na renderer.

### 3.1.5. Portabilni binarni datotečni format

Za pokretač potrebno je izraditi portabilni datotečni format. Taj format sprema mape (razine) koje korisnik može odabrati u glavnom izborniku. Taj datotečni format sadrži sve bitne informacije o mapi koja se igra, a to su: pozadinska slika, početna X pozicija igrača, početna Y pozicija igrača, broj prepreka (n) u mapi te se onda n puta ponavlja sljedeća struktura: x pozicija prepreke, y pozicija prepreke, širina prepreke, visina prepreke i tekstura prepreke. Sve pozicije spremaju se kao std::uint16_t umjesto kao običan int jer je cilj formata da bude portabilan, a C++ standard ne definira širinu podatkovnog tipa int te svaka arhitektura/prevoditelj mogu taj dio implementirati proizvoljno. Za spremanje znakovnih nizova koristi se obično char polje od 50 znakova jer je na svim relevantnim arhitekturama širina char-a ista (C standard, 2007.).

Ovaj datotečni format vidljiv je u klasi MapData u datoteci zaglavlja MapData.h.

```
class MapData{
public:
   bool mapLoaded = false;
   char background[50];
   std::uint16_t startingPosX, startingPosY, obstacles;
   Obstacles obs = Obstacles(GameData::GetInstance()->renderer);
   void LoadMap(std::string mapName);
   std::vector<std::string> GetAllMaps();
};
```

Razlog zbog kojeg se inicijalizira obs prilikom kreiranja klase je taj da klasa Obstacles nema zadani (default) konstruktor.

### 3.1.6. Delta vrijeme

Delta vrijeme služi da bi se prilagodila brzina kretanja igre s brzinom računala. Na primjer, ako želimo da se objekt u videoigri pomakne za 50 jedinica u jednoj sekundi, moramo koristiti delta vrijeme jer objekt u igri mičemo prilikom renderiranja svake sličice. Bez delta vremena, računalo koje pokreče igru na 60 FPS pomaknulo bi objekt brže nego računalo koje pokreće igru na 30 FPS. U ovom radu koristit ćemo varijabilno delta vrijeme koje računamo na način da izmjerimo vrijeme proteklo od prošle sličice (trenutno vrijeme – vrijeme nastanka prethodne sličice). Ako svaku kretnju pomnožimo s delta vremenom, dobit ćemo igru čija se fizika izvršava jednako brzo neovisno o broju sličica o sekundi (Gaffer On Games, 2004.).

Za računanje proteklog vremena između sličica koristi se funkcija SDL_GetTicks() koja vraća broj proteklih milisekundi od početka inicijalizacije programa. Funkcija vraća Uint32 (definiran u SDL-u), a mogao bi se koristiti i std::uint32_t definiran u C++ standardu te u zaglavlju cstdint.

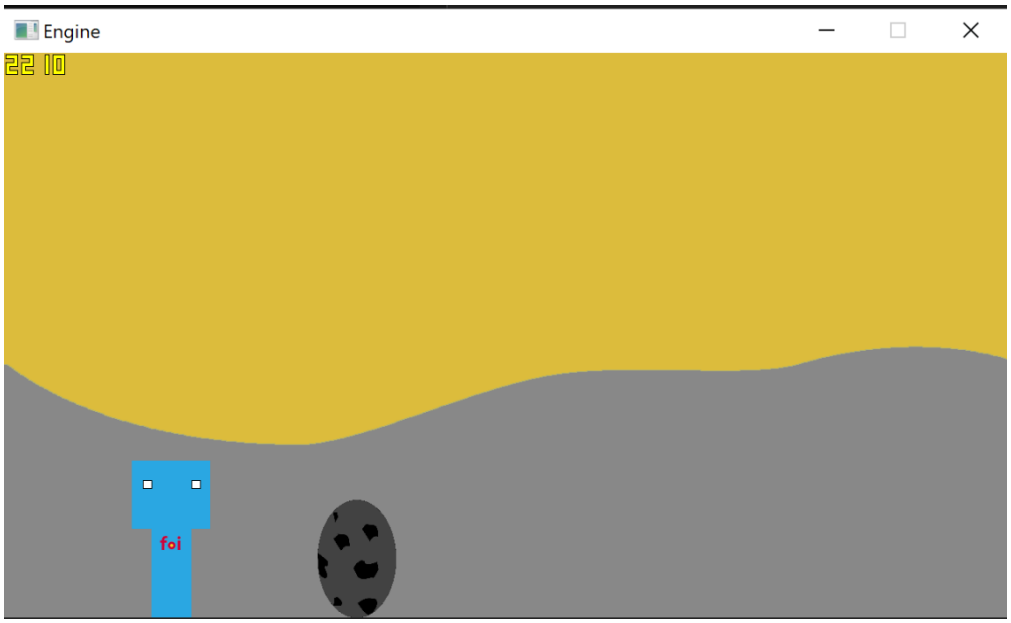
```
Uint32 lastUpdate = SDL_GetTicks();
while (!stop) {
    Uint32 current = SDL_GetTicks();
    float dT = (current - lastUpdate) / 10.0f;
    lastUpdate = current;
}
```

Ako oduzmemo trenutni broj proteklih milisekundi s brojem proteklih milisekundi za vrijeme nastanka prošle sličice dobit ćemo delta vrijeme. U ovom pokretaču to se još dodatno dijeli s 10 jer je takav broj prikladniji za potrebe fizike u ovom pokretaču.

Sada je u varijabli dT spremljeno delta vrijeme te tu vrijednost možemo množiti sa svim pokretima vezanim uz fiziku igre koji trebaju biti konzistentni kroz vrijeme (poput pokreta igrača).

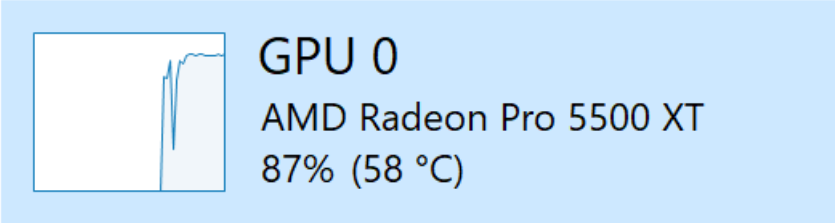
### 3.1.7. Ograničavanje broja sličica po sekundi (FPS)

Na modernim računalima ovako jednostavan pokretač radit će na jako visokom broju sličica po sekundi. To nije nikakav problem što se tiče korisničkog iskustva unutar igre, ali problem je što jedna 2D igra jednostavno ne treba prikazivati vrtoglavi broj sličica po sekundi. To je trošak resursa računala (pogotovo GPU).




Slika 1. Broj sličica u sekundi bez ograničenja (vlastita izrada)

Na gornjoj slici vidimo da se na mom računalu (i7 10700k i Radeon Pro 5500XT) igra pokreće na 2210 sličica po sekundi. To je daleko više nego što bilo koji monitor danas može prikazati. Čak i da neki monitor može to prikazati, za 2D igru to jednostavno nije potrebno. Visok broj FPS-a poželjan je u kompetitivnim igrama, a čak i tad se priča o brojkama od nekoliko stotina, a ne tisuća.



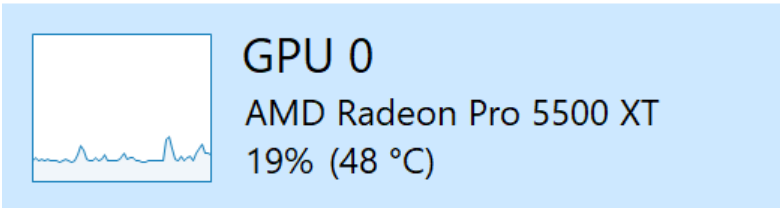

Slika 2. Iskorištenost GPU bez FPS ograničenja (vlastita izrada)

Potrebno je implementirati ograničenje sličica po sekundi. Funkcija SDL_GetPerformanceCounter vraća brojač koji je koristan samo u usporebi sa samim sobom. Razlog zašto se koristi ta funkcija, a ne SDL_GetTicks je taj što je SDL_GetPerformanceCounter puno precizniji. Vrijednost brojača može se pretvoriti u milisekunde korištenjem funkcije SDL_GetPerformanceFrequency – podijelimo brojač s vraćenom vrijednosti frekvencije brojača i podijelimo sve sa 1000.

Način na koji ćemo limitirati FPS je taj da ćemo izmjeriti vremensku razliku između trenutka početka crtanja sličice i trenutka kraja crtanja sličice te ćemo od željenog vremena crtanja sličice oduzeti razliku. To se može kombinirati s funkcijom SDL_Delay koja blokira glavnu dretvu na određeni broj milisekundi.

```
unsigned int maxFps = 250;
while (!stop) {
    std::uint64_t start = SDL_GetPerformanceCounter();
    // kod za procesiranje i crtanje sličice
    std::uint64_t end = SDL_GetPerformanceCounter();
    float elapsedMS = (end - start) /
static_cast<float>(SDL_GetPerformanceFrequency()) * 1000.0f;
    int delay = static_cast<int>(abs(1000.0f / maxFps - elapsedMS));
    SDL_Delay(delay);
}
```


Slika 3. FPS nakon ograničavanja na 250 (vlastita izrada)



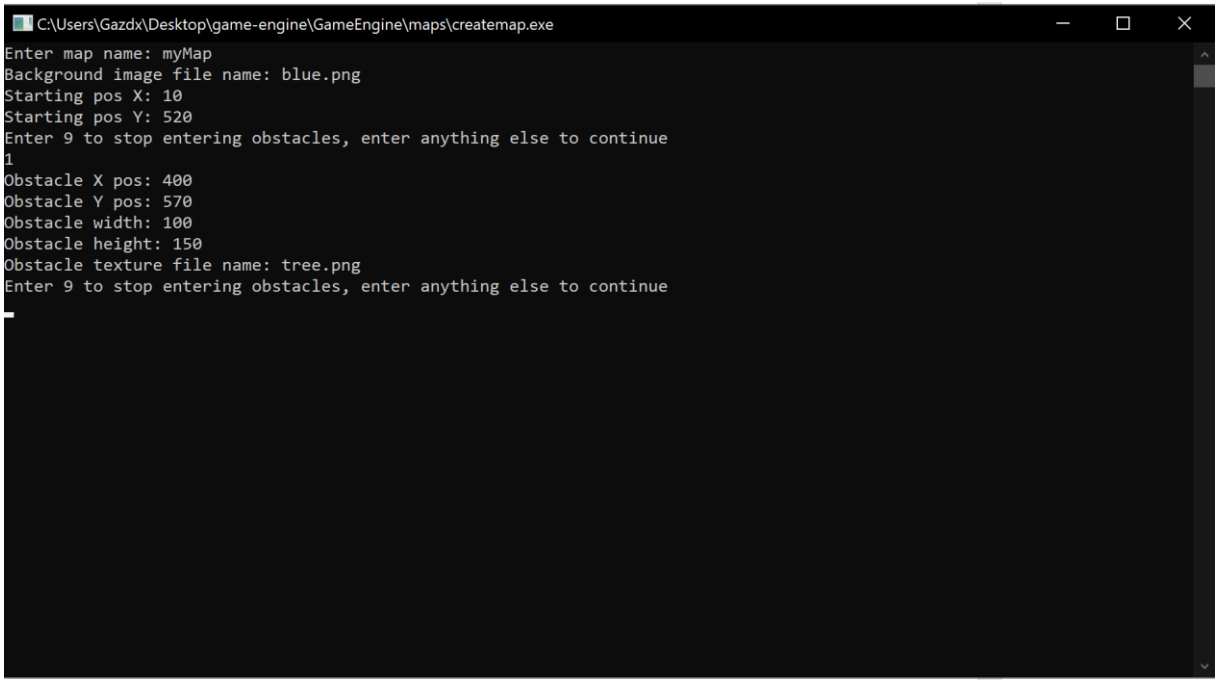

Slika 4. Iskorištenost GPU s FPS ograničenjem (vlastita izrada)

Kao što je vidljivo nakon ograničavanja broja sličica po sekundi smanji se i opterećenje na računalo, točnije GPU.

## 3.2. Uređivač mapa

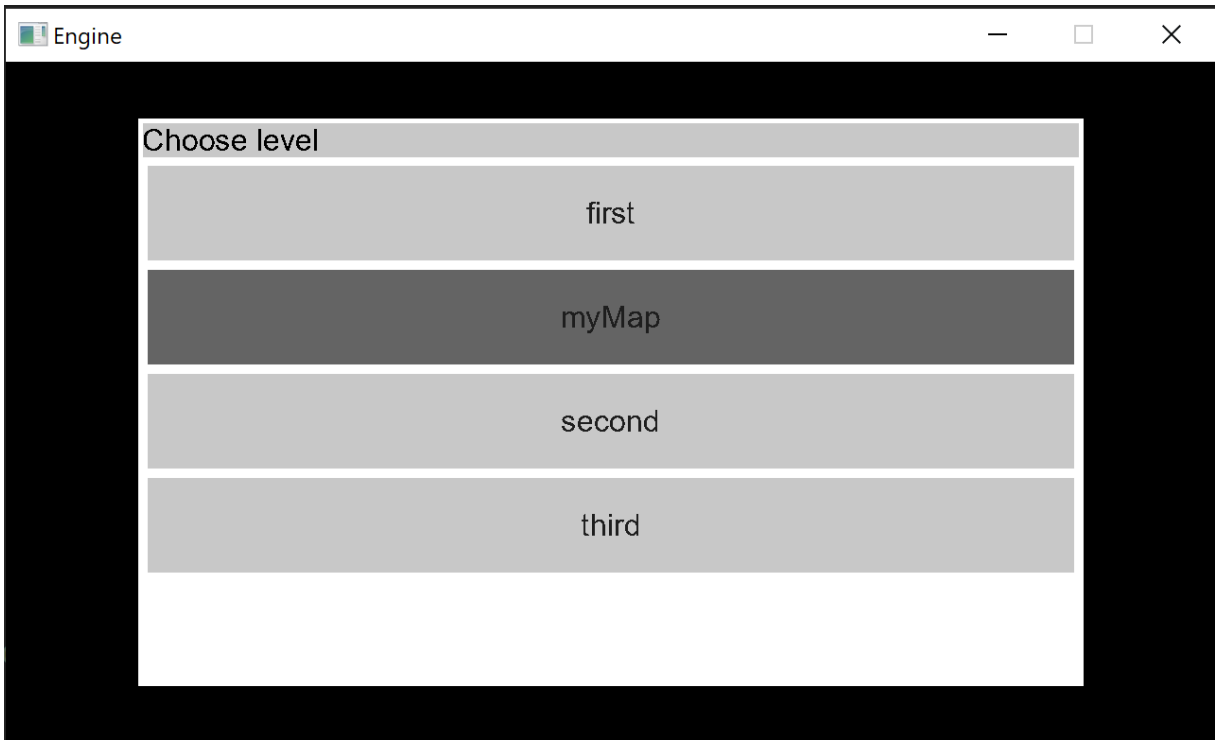
Prilikom pokretanja igre, tj. pokretača, igra pita korisnika da odabere koju mapu želi učitati. Mapa sadrži podatke koji su opisani ranije, a to su pozicija igrača, pozadina, prepreke, itd. S obzirom na to da su mape spremljene u binarnom formatu u folderu maps/ kao .gmap datoteke, mora postojati neki način da se te mape kreiraju. Iz tog razloga se u maps/ folderu nalazi map editor te se njegov kod nalazi u datoteci createmap.cpp.

Uređivač mapa je poprilično jednostavan i napravljen je kao konzolni program koji pita korisnika detalje o tome kako želi da mapa bude raspoređena.

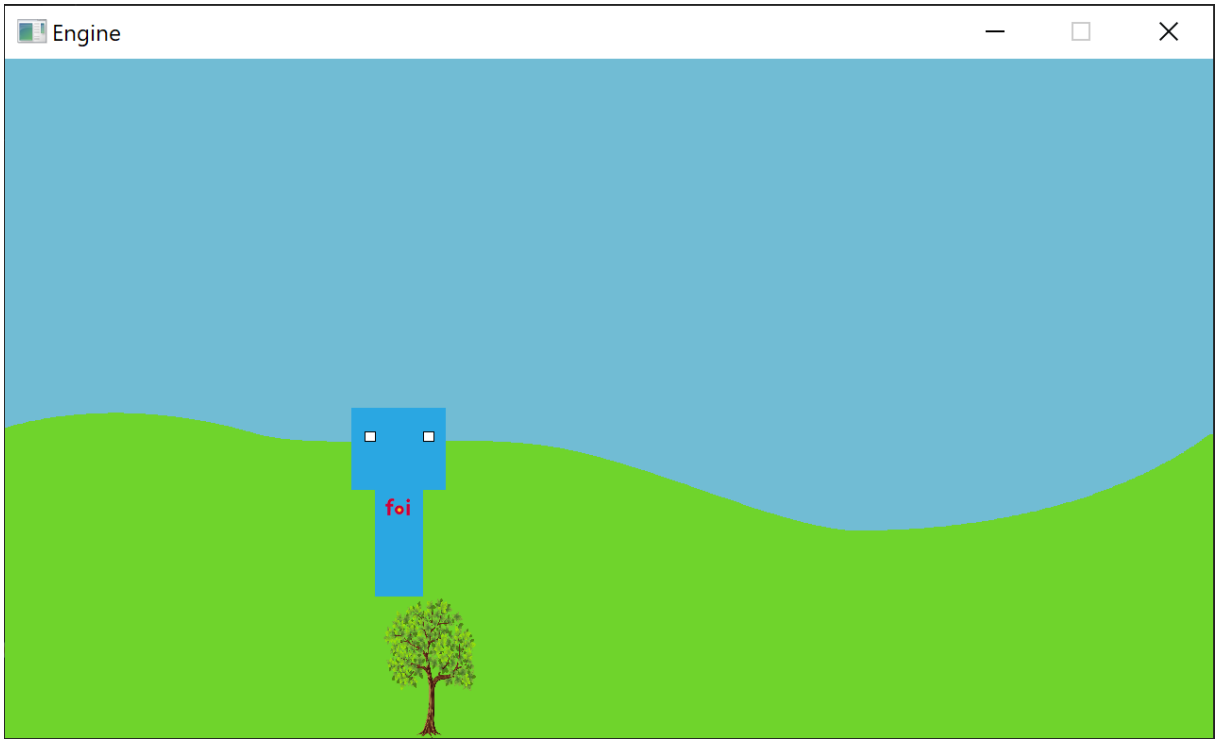



Slika 5. Konzolni uređivač mapa (vlastita izrada)

Korisnik prvo unosi ime mape koje će se ispisati u glavnom izborniku igre. Mapa će se spremiti u folder maps/ pod nazivom uneseno_ime.gmap. Zatim, korisnik upisuje ime slikovne datoteke za pozadinu. Slikovne datoteke nalaze se u folderu textures/. Nakon toga, upisuju se početna x i y pozicija igrača te se unose prepreke – njihova x i y pozicija, širina, visina te slikovna datoteka teksture.




Slika 6. Vidljivost nove mape u glavnom izborniku (vlastita izrada)




Slika 7. Nova mapa učitana (vlastita izrada)

# 4. Mogućnosti pokretača igre

## 4.1. Sustav za korisničko sučelje (UI)

Da bi programer lako mogao tražiti korisnički unos dobro bi bilo napraviti neko korisničko sučelje. Korisničko sučelje u ovom pokretaču sastoji se od prozora, gumba, teksta… Za ovu svrhu napravljene su zasebne klase i funkcije koje se lako pozivaju.

### 4.1.1. Klasa UIElement

Klasa UIElement služi kao bazna klasa koju nasljeđuje svaki element korisničkog sučelja poput prozora i gumba. Elementi sučelja imaju neke zajedničke karakteristike te su one definirane u ovoj klasi.

```
class UIElement
{
protected:
   bool autoLayout = false;
   bool show = true;
public:
   virtual void Draw() = 0;
   virtual bool IsAutoLayout() = 0;
   virtual SDL_Rect* GetRect() = 0;
   virtual void Show();
   virtual void Hide();
};
```

Ova klasa sadrži dva atributa – booleane autoLayout i show. Atribut autoLayout označava mogućnost elementa sučelja da se sam prilagođava kontekstu u kojem se nalazi. Da bi se dobro objasnilo autoLayout prvo treba objasniti klasu UIHandler (koja je kontekst elementa), tako da više o tome biti će objašnjeno kasnije u radu. Atribut show označava vidljivost elementa sučelja. Naime, svaki element sučelja može se skrivati i prikazivati po potrebi. Metodu Draw mora implementirati svaki element sučelja jer ta metoda definira kako se svaki element crta na renderer. IsAutoLayout vraća vrijednost atributa autoLayout za potrebe klase UIHandler. GetRect vraća pokazivač na SDL_Rect (pravokutnik) tako da se može saznati pozicija elementa i njegova veličina. Metode Show i Hide modificiraju vrijednost atributa show te po potrebi skrivaju i prikazuju element.

### 4.1.2. Klasa UIHandler

Klasa UIHandler služi kao kontekst za sve elemente sučelja te su svi elementi sučelja sadržani unutar te klase u polju (vektoru). Ova klasa brine se za razne bitne radnje s elementima sučelja kao što su automatsko raspoređivanje elemenata (na temelju atributa autoLayout) ili navigiranje izbornika (korištenje strelica na tipkovnici da bismo označili sljedeći element unutar izbornika). UIHandler je napravljen na način da obavezno mora sadržavati prozor (UIWindow), a ostali elementi su proizvoljni. Deklaracija klase UIHandler nalazi se u datoteci UIHandler.h.

```
class UIHandler
{
private:
    std::vector<std::shared_ptr<UIElement>> elements;
    int highlightedButton;
    void HighlightPrevButton();
    void HighlightNextButton();
    void HighlightSelectedButton();
public:
    UIHandler();
    void Draw();
    void Add(const std::shared_ptr<UIElement> &element);
    int CountBtns();
    void HandleEvents(SDL_Event* ev);
    const std::shared_ptr<UIWindow> GetWindow();
    SDL_Rect* GetBottomElementRect();
    void Clear();
    void HideAll();
    void ShowAll();
};
```

Vektor elements sadrži sve pokazivače na elemente koji se nalaze u kontekstu. Elementi se dodaju u kontekst koristeći metodu Add. Sustav sučelja u ovom pokretaču jako se oslanja na pametne pokazivače (tzv. smart pointers) i moderni C++ jer su na taj način šanse za curenje memorije minimalne. Na pametne pokazivače može se gledati kao na mehanizam sličan sakupljaču smeća (garbage collector), iako oni zapravo rade na potpuno različit način te se ne mogu smatrati sakupljačima smeća. Naime, kada se uništi zadnja referenca na objekt, on će se automatski dealocirati tog trenutka, dok se sakupljači smeća pozivaju po potrebi te programer generalno nema kontrolu nad tim. C++ nikad neće imati sakupljač smeća, ali pametni pokazivači mogu se koristiti kao zamjena za tu funkcionalnost u nekim slučajevima.

Atribut highlightedButton označava indeks gumba (UIButton) koji je trenutno označen. Metode HighlightPrevButton, HighlightNextButton i HighlightSelectedButton su pomoćne metode za kontrolu označavanja gumba.

Metoda Add koristi se za dodavanje elementa unutar konteksta UIHandler te ona samo dodaje element u vektor elements.

Metoda HandleEvents poziva se unutar glavne petlje igre te upravlja korisničkim unosima.

```
   void UIHandler::HandleEvents(SDL_Event* const ev) {
      switch (ev->type) {
            case SDL_KEYDOWN: {
                  switch (ev->key.keysym.sym) {
                        case SDLK_DOWN: {
                              HighlightNextButton();
                              break;
                        }
                        case SDLK_UP: {
                              HighlightPrevButton();
                              break;
                        }
                        case SDLK_RETURN: {
                              int btnIndex = 0;
                              for (auto const& el : elements) {
                                    auto btnPtr =
   std::dynamic_pointer_cast<UIButton>(el);
                                    if (btnPtr != nullptr) {
                                          if (btnIndex == highlightedButton)
   {
                                                btnPtr->CallAction();
                                                break;
                                          }
                                          btnIndex++;
                                    }
                              }
                              break;
                        }
                  }
                  break;
            }
      }
}
```

Kao što je vidljivo u kodu, ako korisnik pritisne strelicu dolje označit će se sljedeći gumb. Ako pritisne strelicu gore označit će se prošli gumb. Ako pritisne enter (return), pronalazi se točan gumb koji je označen po indeksu (highlightedButton) te kad se pronađe, pozove se njegova radnja. Svaki gumb implementira CallAction() koja zapravo poziva callback funkciju koja se gumbu šalje kao argument tipa std::function.

Ranije je spomenuto da svaki UIHandler mora sadržavati prozor i upravo iz tog razloga postoji metoda GetWindow koja vraća pokazivač na pronađeni prozor. Prozor se može nalaziti bilo gdje u vektoru elements.

Metoda GetBottomElementRect vraća pokazivač na SDL_Rect elementa koji u isto vrijeme mora imati autoLayout atribut istinit te mora imati najveću Y poziciju (tako da je na dnu). Ova metoda korisna je za autoLayout funkcionalnost gdje se elementi automatski raspoređuju jedan ispod drugog. Automatski se uzme najniža Y pozicija te se ispod nje doda novi element.

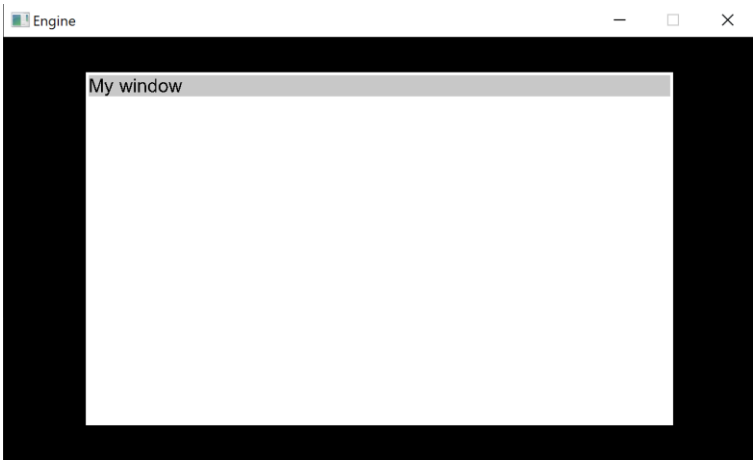
```
SDL_Rect* UIHandler::GetBottomElementRect() {
    int maxY = 0;
    SDL_Rect* result = nullptr;
    for (auto const& el : elements) {
        if (el->IsAutoLayout()) {
            SDL_Rect* r = el->GetRect();
            if (r->y > maxY) {
                maxY = r->y;
                result = r;
            }
        }
    }
    return result;
}
```

### 4.1.3. Klasa UIWindow

Klasa UIWindow koristi se za crtanje prozora te nasljeđuje UIElement. Ta klasa poprilično je jednostavna jer ne vrši nikakve radnje već samo nacrta prozor, naslovnu traku i naziv prozora unutar nje.

```
class UIWindow : public UIElement {
private:
    std::string windowTitle;
    int x, y, w, h;
    SDL_Rect windowRect, titleBarRect;
    TTF_Font* titleFont;
    SDL_Color titleColor{ 0,0,0 };
    SDL_Surface* titleSurface;
    SDL_Texture* titleTexture;
    SDL_Rect titleRect;
public:
    UIWindow(std::string windowTitle, int w, int h);
    void Draw();
    SDL_Rect* GetRect();
    bool IsAutoLayout();
};
```

U deklaraciji klase nalaze se atributi: windowTitle za naziv prozora koji će pisati u naslovnoj traci, x, y, w i h definiraju veličinu i poziciju prozora, windowRect definira pravokutnik prozora, titleBarRect definira pravokutnik naslovne trake, titleFont je pokazivač na TTF_Font* kojeg vraća funkcija TTF_OpenFont, titleColor predstavlja boju teksta u naslovnoj traci, titleSurface i titleTexture su pokazivači na površinu i teksturu na koje će se crtati zadani tekst te titleRect služi za određivanje pozicije teksta na ekranu.




Slika 8. Prazan UIWindow (vlastita izrada)

### 4.1.4. Klasa UIButton

Klasa UIButton nasljeđuje klasu UIElement te definira gumb u sučelju. Gumb može ili ne mora biti dio UIHandler-a. Ako nije dio UIHandler-a onda mu treba zadati poziciju i veličinu. Ako je dio UIHandler-a onda samo u konstruktoru treba poslati referencu na UIHandler.

```
class UIButton : public UIElement {
private:
    std::function<void()> action;
    int x, y, w, h;
    std::string btnText;
    SDL_Rect btnRect, textRect;
    TTF_Font *btnFont;
    SDL_Surface* btnSurface;
    SDL_Texture* btnTexture;
    bool highlighted;
public:
    void InitCommon(std::string& btnText);
    UIButton(int x, int y, int w, int h, std::string btnText, const
std::function<void()>& action);
    UIButton(UIHandler &hnd, std::string btnText, const
std::function<void()>& action);
    void Draw();
    void Highlight();
    void RemoveHighlight();
    void CallAction();
    bool IsAutoLayout();
    SDL_Rect* GetRect();
};
```

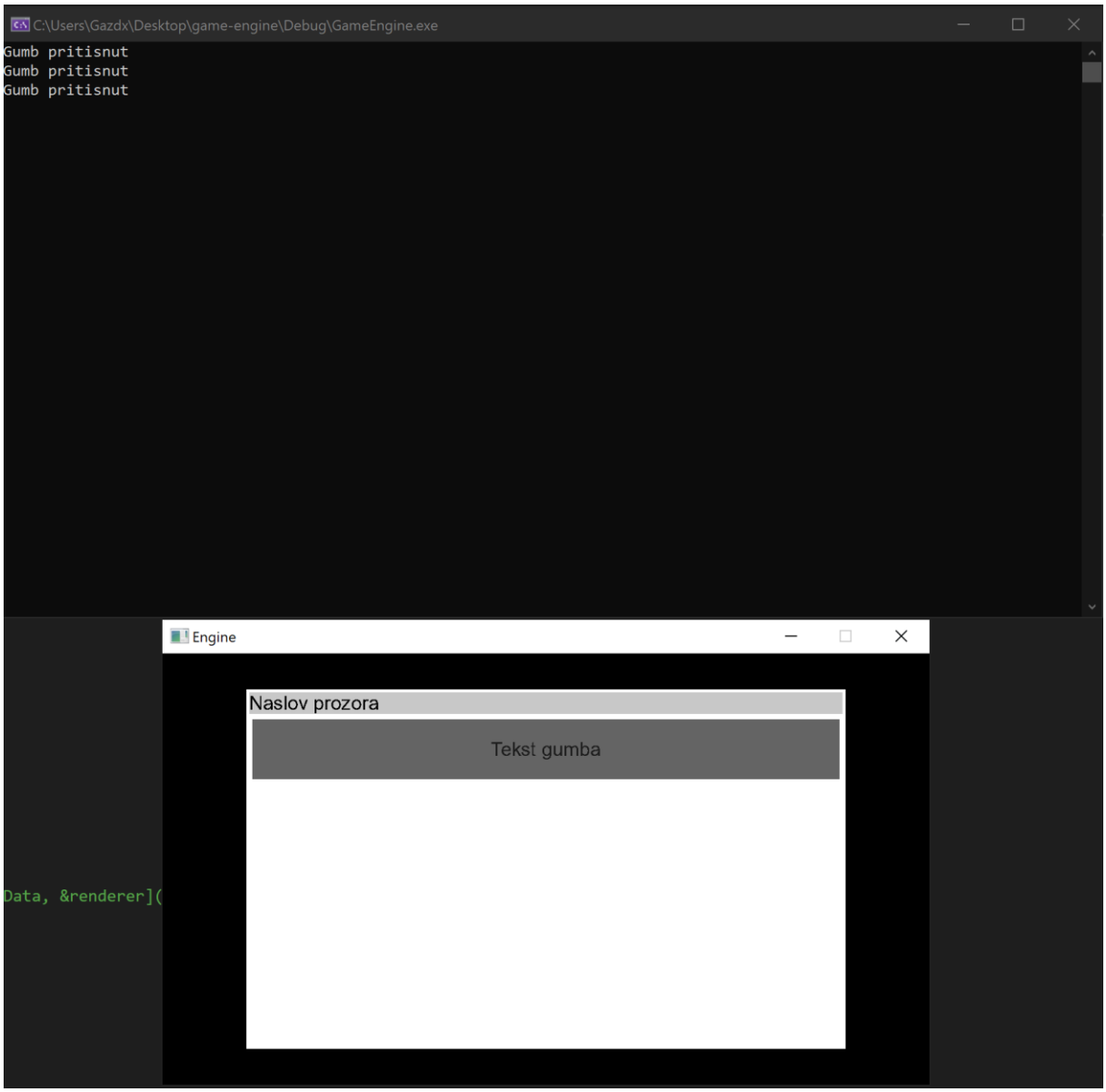
Upravo iz razloga što gumb ne mora biti dio UIHandler-a postoje dva konstruktora. UIButton kao argumente u konstruktoru prima ili UIHandler ili x, y, širinu i visinu, tekst gumba te funkciju koja se pozove prilikom pritiska na gumb. Ako se gumbu pošalje UIHandler on se automatski pozicionira jer će mu tad autoLayout biti istinit. Pozicija se računa u konstruktoru na temelju pozicije najnižeg elementa te na temelju pozicije i veličine prozora (upravo iz ovog razloga UIHandler mora imati prozor).

```
UIButton::UIButton(UIHandler &hnd, std::string btnText, const
std::function<void()>& action) {
   this->action = action;
   SDL_Rect* wndRect = hnd.GetWindow()->GetRect();
   this->x = wndRect->x + 10;
   SDL_Rect* bottomElem = hnd.GetBottomElementRect();
   if (bottomElem == nullptr) {
         this->y = wndRect->y + 50;
   }
   else {
         this->y = bottomElem->y + bottomElem->h + 10;
   }
   this->w = wndRect->w - 20;
   this->h = 100;
   InitCommon(btnText);
   this->autoLayout = true;
}
```

U kodu konstruktora vidi se da se iz handler-a dohvati pravokutnik koji predstavlja prozor te pravokutnik koji predstavlja element najniže na ekranu. Na temelju toga računaju se širina, visina i pozicija gumba. Konstruktor za verziju bez UIHandler-a nešto je jednostavniji i nema ga potrebe objašnjavati.

#### 4.1.4.1. Primjer kreiranja UIHandler-a, UIWindow-a i UIButton-a te zadavanje radnje gumbu

```
UIHandler uiHandler;
auto levelSelectWindow = std::make_shared<UIWindow>("Naslov prozora",
1000, 600);
uiHandler.Add(levelSelectWindow);
auto testBtn = std::make_shared<UIButton>(uiHandler, "Tekst gumba", []()
{
   std::cout << "Gumb pritisnut" << std::endl;
});
uiHandler.Add(testBtn);
```




Slika 9. Demonstracija gumba i radnje (vlastita izrada)

## 4.2. Fizika i ponašanje igre

### 4.2.1. Klasa Player

Glavni dio fizike definiran je u klasi Player. Ono što je očekivano od igrača je da se može normalno kretati (korištenjem strelica na tipkovnici), skakati, padati i zaletjeti se u prepreku bez da prođe kroz nju (kolizije). Sve ove mogućnosti definirane su u klasi Player.

```
class Player
{
private:
    float playerX, playerY;
    float deltaX, deltaY;
    float jumpDelta;
    SDL_Rect playerRect;
    SDL_Surface* imageTexture;
    SDL_Texture* playerTexture;
    SDL_Renderer* renderer;
    bool canJump;
public:
    Player(SDL_Renderer* renderer, int x, int y);
    ~Player();
    void HandleEvents(SDL_Event* event);
    void HandlePhysics(float dT, Obstacles* obstacles);
    void HandleDrawing();
};
```

#### 4.2.1.1. Metoda HandleEvents

```
void Player::HandleEvents(SDL_Event* const event) {
    switch (event->type) {
            case SDL_KEYDOWN: {
                    switch (event->key.keysym.sym) {
                            case SDLK_UP: {
                                    if (canJump) {
                                            jumpDelta = JUMP_MAX_VELOCITY;
                                    }
                                    break;
                            }
                            case SDLK_LEFT: {
                                    deltaX = -3.0f;
                                    break;
                            }
                            case SDLK_RIGHT: {
                                    deltaX = 3.0f;
                                    break;
                            }
                            break;
                    }
                    break;
            }
            case SDL_KEYUP: {
                    SDL_KeyCode verticalMovementKeys[] = { SDLK_DOWN, SDLK_UP };
                    SDL_KeyCode horizontalMovementKeys[] = { SDLK_LEFT, SDLK_RIGHT };
                    if (std::find(std::begin(verticalMovementKeys),
std::end(verticalMovementKeys), event->key.keysym.sym) !=
std::end(verticalMovementKeys)) {
                            deltaY = 0.0f;
                    }
                    if (std::find(std::begin(horizontalMovementKeys),
std::end(horizontalMovementKeys), event->key.keysym.sym) !=
std::end(horizontalMovementKeys)) {
                            deltaX = 0.0f;
                    }
                    break;
            }
    }
}
```

Atributi playerX i playerY čuvaju trenutnu poziciju igrača. DeltaX te deltaY definiraju kojom se brzinom igrač kreće - ako ide lijevo deltaX mora biti negativan broj, a ako ide desno mora biti pozitivan te isto tako ako ide prema gore deltaY mora biti negativan broj, a ako pada dolje mora biti pozitivan broj. JumpDelta specifičan je parametar kojim se određuje fizika skakanja – pozitivan broj predstavlja uspon u skoku, dok negativan broj predstavlja pad. Kada se inicira skok, jumpDelta postavlja se na konstantu JUMP_MAX_VELOCITY te se svaku sličicu smanjuje za JUMP_DECCEL. Također je definira konstanta GRAVITY koja služi za slučajeve dok igrač sklizne s ruba i treba pasti na zemlju.

```
#define JUMP_MAX_VELOCITY 10.0f
#define JUMP_DECCEL 0.15f
#define GRAVITY 3.0f
```

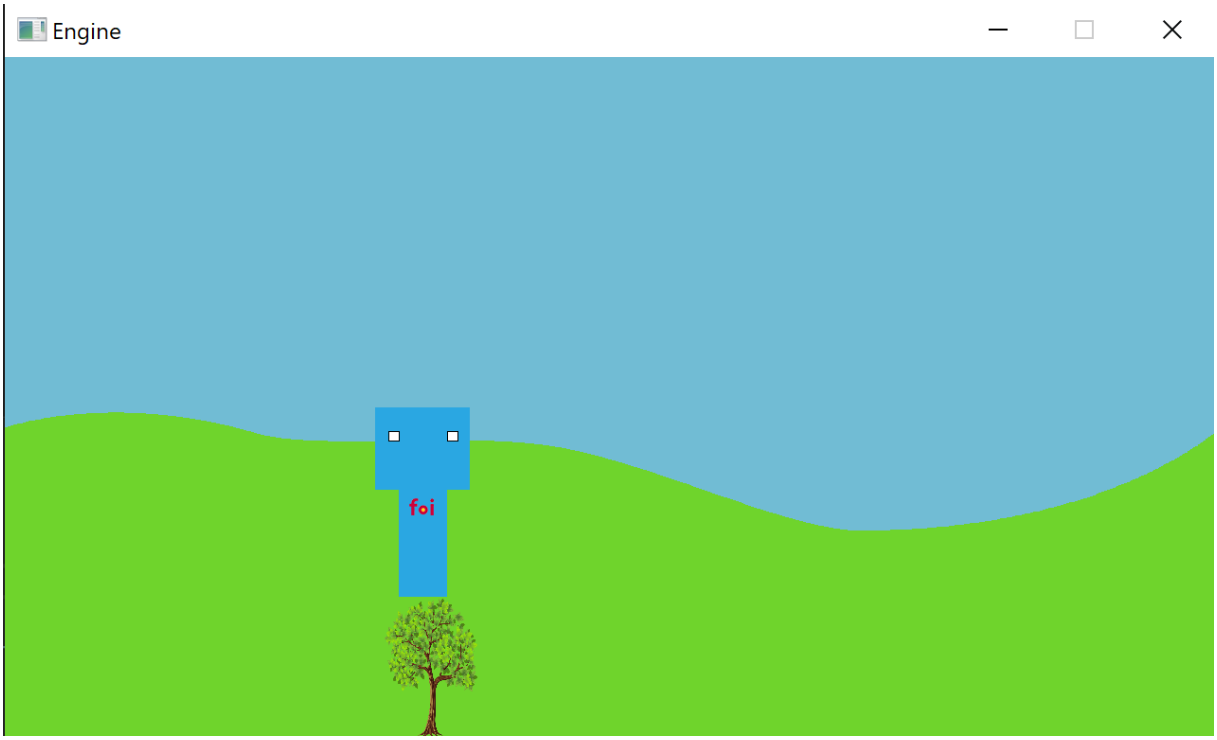
Na događaju (eventu) SDL_KEYUP moramo zaustaviti igrača. To činimo tako da odredimo veže li se podignuta tipka za X ili Y poziciju igrača te postavimo deltu potrebne pozicije na 0.

#### 4.2.1.2. Metoda HandlePhysics

```
void Player::HandlePhysics(float dT, Obstacles* const obstacles) {
   float tempX = std::max(playerX + deltaX * dT, 0.0f);
   float tempY = std::max(playerY + (GRAVITY - jumpDelta) * dT, 0.0f);
   jumpDelta = std::max(jumpDelta - JUMP_DECCEL * dT, 0.0f);
   SDL_Rect tempRect{ static_cast<int>(tempX), static_cast<int>(tempY),
100, 200 };
   bool collisionFound = false;
   SDL_Rect intersectRect;
   for (Obstacle &o : *(obstacles->GetObstacles())) {
       if (SDL_IntersectRect(&tempRect, o.GetRect(), &intersectRect)) {
               collisionFound = true;
               cout << intersectRect.w <<"," << intersectRect.h << endl;
               break;
       }
   }
   if (!collisionFound || (collisionFound && (intersectRect.w >
intersectRect.h))) {
         playerX = tempX;
         playerRect.x = static_cast<int>(playerX);
   }
   if (tempY < (720 - playerRect.h) && (!collisionFound ||
(collisionFound && (intersectRect.w < intersectRect.h)))) {
         playerY = tempY;
         playerRect.y = static_cast<int>(playerY);
         canJump = false;
   }
   else canJump = true;
}
```

Ako izuzmemo glavnu petlju, ovo je možda čak i najvažniji dio koda. Brine se za sve dijelove fizike igrača – kretnje, skakanje, kolizije te gravitacija. Prvo odredimo X poziciju tako da izračunamo playerX + deltaX * dT i koristimo std::max da se osiguramo da to neće biti manje od nule (da igrač ne može izaći iz ekrana). Također treba odrediti Y poziciju tako da izračunamo playerY + (GRAVITY – jumpDelta) * dT te se također funkcijom std::max osiguramo da igrač ne može biti na negativnoj poziciji. Y pozicija igrača uvijek bi se trebala povećavati za GRAVITY jer igrača gravitacija vuče prema dolje kao i stvarne fizičke objekte. Vrijednost jumpDelta bori se protiv gravitacije te ako je jumpDelta veća od gravitacije, igrač će se kretati prema gore (uspon skoka). Naravno, to ne traje dugo jer skok usporava i iz tog razloga se jumpDelta smanjuje svaku sličicu za vrijednost konstante JUMP_DECCEL.

SDL sadrži korisnu funkciju SDL_IntersectRect koja računa sjecište dva pravokutnika te kao rezultat vraća true ili false te ako sjecišta ima sprema pravokutnik koji predstavlja sjecište u pokazivač tipa SDL_Rect*. Ako pravokutnik igrača ima sjecište s bilo kojim pravokutnikom to znači da se dogodila kolizija te SDL_IntersectRect spremi pravokutnik koji predstavlja sjecište. Ako je širina tog pravokutnika veća nego njegova visina radi se o koliziji na X osi, a ako je manja onda se dogodila kolizija na Y osi. Ako su širina i visina jednake dogodila se kolizija točno na rubu pravokutnika. Za Y os još moramo uzeti u obzir da računamo koliziju ako je igrač na podu jer on stoji na podu. Također, ako se dogodi bilo kakva kolizija na Y osi treba spriječiti proces skakanja igrača. Ako se u navedenom kodu dogodi kolizija, pozicija igrača jednostavno se ne ažurira nego ostane ista. Na taj način igrač ne može proći kroz prepreku.




Slika 10. Demonstracija kolizije (vlastita izrada)

U slici 10 vidi se da igrač stoji na prepreci (drvu) i ne propada kroz prepreku.

### 4.2.2. Klase Obstacle i Obstacles

Klasa Obstacle čuva podatke o jednoj prepreci. Ti podaci su: tekstura, površina, renderer i pravokutnik koji predstavlja položaj i veličinu prepreke. U konstruktoru prima renderer na koji se crta, pravokutnik koji predstavlja položaj i veličinu te znakovni niz koji predstavlja putanju do slikovne datoteke za teksturu.

```
class Obstacle {
private:
   SDL_Surface* textureImg;
   SDL_Texture* texture;
   SDL_Rect rect;
   SDL_Renderer* renderer;
public:
   Obstacle(SDL_Renderer* renderer, SDL_Rect rect, const char*
textureImg);
   void Draw();
   SDL_Rect* GetRect();
};
```

Ova klasa također sadrži metodu za crtanje Draw te metodu koja se koristi da bi se očitali položaj i veličina za svrhe prepoznavanja kolizija – GetRect. Ona vraća pokazivač tipa SDL_Rect koji predstavlja pravokutnik prepreke.

Klasa Obstacles u sebi čuva objekte tipa Obstacle. U sebi sadrži vektor koji drži zapise tipa Obstacle po vrijednosti (jer se učitava samo jednom pa implikacije na performanse nisu velike) te metode Draw, Add i GetObstacles.

```
class Obstacles {
private:
   SDL_Renderer* renderer;
   std::vector<Obstacle> obstacles;
public:
   Obstacles(SDL_Renderer* renderer);
   void Add(Obstacle obstacle);
   void Draw();
   std::vector<Obstacle>* GetObstacles();
};
```

Metoda Draw u klasi Obstacles poziva metodu Draw od svih elemenata u vektoru obstacles.

```
void Obstacles::Draw() {
   for (Obstacle &o : this->obstacles) {
         o.Draw();
   }
}
```

Na taj način se odjednom crtaju sve prepreke. Glavna petlja igre će pozvati Obstacles::Draw te će ta metoda pozvati Obstacle::Draw.

Metoda Add prima jedan argument tipa Obstacle te služi samo da bi se u vektor obstacles mogle dodati prepreke. Metoda GetObstacles vraća pokazivač tipa std::vector<Obstacle>* što zapravo znači da vraća pokazivač na vektor obstacles zadan u klasi. Na taj način i vanjske klase mogu doći do podataka o preprekama.

# 5. Zaključak

Na kraju svega može se zaključiti da je SDL i više nego prikladan za izradu 2D pokretača za video igre. Ovaj pokretač koji je razvijen ima osnovne mogućnosti za crtanje, mjerenje vremena, fiziku, kolizije, generiranje korisničkog sučelja, hvatanje događaja te čitanje i generiranje binarnih datoteka. Za crtanje slikovnih datoteka u radu se koristi SDL-ova službena biblioteka SDL_Image, a za crtanje teksta korištenjem TrueType fontova koristi se SDL-ova službena biblioteka SDL_TTF. U radu je također objašnjeno kako otprilike funkcionira SDL i potrebne biblioteke za ovaj pokretač.

Ovaj pokretač je također portabilan jer se u kodu koriste samo SDL-ove višeplatformske biblioteke te standardne funkcije C++17 standarda. Razlog zbog kojeg se u ovom radu koristi C++17 standard, a ne C++14 koji je zadan kao početni u Visual Studiu je taj što C++17 ima standardizirani način za iteriranje kroz datoteke u direktoriju, a to se u ovom radu koristi za učitavanje liste mapa u glavnom izborniku.

Valjalo bi napomenuti i da SDL ima svoju ulogu i u razvoju 3D igara. SDL se često koristi za kreiranje prozora (jer u jeziku C++ ne postoji standardan način za to) te za snimanje događaja. SDL sadrži pomoćne funkcije za rad s OpenGL-om te za kreiranje konteksta na raznim platformama kao što su: Linux/X11, Win32, BeOS, MacOS Classic/Toolbox, Mac OS X, FreeBSD/X11 i Solaris/X11 (libsdl.org, 2013.).

# 6. Popis literature

Simple DirectMedia Layer (2021.), preuzeto s https://www.libsdl.org/ (1.7.2021.)
Glenn Fielder (2004.), Fix Your Timestep!, Gaffer On Games, preuzeto s https://gafferongames.com/post/fix_your_timestep/ (1.7.2021.)
SDL Wiki, SDL_Surface, preuzeto s https://wiki.libsdl.org/SDL_Surface (1.9.2021.)
SDL Wiki, SDL_Texture, preuzeto s https://wiki.libsdl.org/SDL_Texture (1.9.2021.)
SDL Wiki, SDL_CreateWindow, preuzeto s https://wiki.libsdl.org/SDL_CreateWindow (1.9.2021.)
SDL Wiki, SDL_CreateRenderer, preuzeto s https://wiki.libsdl.org/SDL_CreateRenderer (1.9.2021.)
ISO/IEC 9899:TC3 (2007.), C standard, preuzeto s http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf (1.9.2021.)
Perforce, What Are the Most Popular Game Engines (2020.), preuzeto s https://www.perforce.com/blog/vcs/most-popular-game-engines (1.9.2021.)
Study tonight, Game Engine and History of Game Development (2021.), preuzeto s https://www.studytonight.com/3d-game-engineering-with-unity/game-engine (1.9.2021.)
Simple DirectMedia Layer, Using OpenGL With SDL (2013.), preuzeto s https://www.libsdl.org/release/SDL-1.2.15/docs/html/guidevideoopengl.html (1.9.2021.)

# 7. Popis slika

# 8. Prilozi

https://github.com/leonardogazdek/game-engine - sav izvorni kod i Visual Studio rješenje nalazi se u ovom GitHub repozitoriju