



PROG2i

## **desktopová hra – Walking Shooter**

**DANIEL KREJSKA**

**V4C**

**Profilová část maturitní zkoušky  
MATURITNÍ PRÁCE**

**BRNO 2022**

# Prohlášení

Prohlašuji, že jsem maturitní práci na téma „desktopová hra WWS“ vypracoval samostatně a použil jen zdroje uvedené v seznamu literatury.

Prohlašuji, že:

- Beru na vědomí, že zpráva o řešení maturitní práce a základní dokumentace k aplikaci bude uložena v elektronické podobě na intranetu Střední školy informačních technologií a sociální péče, Brno, Purkyňova 97.
- Beru na vědomí, že bude má maturitní práce včetně zdrojových kódů uložena v knihovně SŠITSP Brno, Purkyňova 97, dostupná k prezenčnímu nahlédnutí. Škola zajistí, že nebude pro nikoho možné pořizovat kopie jakékoliv části práce.
- Beru na vědomí, že SŠITSP Brno, Purkyňova 97 má právo celou moji práci použít k výukovým účelům a po mém souhlasu nevýdělečně moji práci užít ke své vnitřní potřebě.
- Beru na vědomí, že pokud je součástí mojí práce jakýkoliv softwarový produkt, považují se za součást práce i zdrojové kódy, které jsou předmětem maturitní práce, případně soubory, ze kterých se práce skládá. Součástí práce není cizí ani vlastní software, který je pouze využíván za přesně definovaných podmínek, a není podstatou maturitní práce.

Daniel Krejska

Bartošova 3,

602 00 Brno

V Brně dne 22.4.2022

.....

# Poděkování

Děkuji vedoucímu maturitní práce Ing. Petru Pernesovi za metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé maturitní práce. Dále děkuji Jakubu Procházkovi, Jiřímu Moravcovi a Janu Lozrtovi za pomoc při testování funkčnosti programu.

# Anotace

Cílem této maturitní práce je tvorba počítačové videohry s vlastním enginem. Hráčovým cílem je zničení všech terčů v co nejkratším časovém úseku na mapách, které může uživatel sám jednoduše tvořit za pomoci textových souborů. Tudíž se program musí starat nejen o input hráče a samotná herní pravidla, ale i o technickou funkčnost, jako je správa vstupu, korektní řízení toku programu a vykreslování grafiky.

# Obsah

## Obsah

Prohlášení.....	2
Poděkování.....	3
Anotace .....	4
Obsah.....	5
Seznam použitých zkratk .....	7
Úvod.....	8
UML Diagramy .....	9
Rozbor řešení .....	10
Použité technologie / nástroje.....	10
programovací jazyk C++ .....	10
knihovna SFML.....	10
Asprite.....	10
Gimp.....	10
Rozbor programu .....	11
Engine.....	11
Reakce na změnu velikosti okna .....	12
Nahrání map do programu a načtení vybrané mapy ke hře .....	14
GameObject .....	19
hráč – třída Soldier .....	19
terče – třída Target .....	23
TextureHolder – načítání obrázků ze souborů .....	25
Animace .....	26
Střelba – třída Bullet.....	28
Systémové požadavky .....	31
Závěr .....	32

Seznam ilustrací.....	33
Zdroje .....	35
Přílohy .....	36
Návod k použití.....	38
Uživatelská tvorba mapy .....	38
Pohyb v menu .....	41
Ovládání hry .....	42

# Seznam použitých zkratek

FPS – Frames Per Second

SW – Software

SFML – Simple and Fast Multimedia Library

UI – User Interface

OS – operační systém

UML – Unified Modeling Language

API – Application Programming Interface

# Úvod

Videohry jsou zábavní software. Jsou tvořeny audiovizuálním výstupem, který reaguje na vstup uživatele. Jsou interaktivní a musejí mají nějaký cíl, kterého se hráč snaží dosáhnout.

Já jsem se pro tuto práci rozhodl, protože jsem si chtěl zkusit vývoj videohry a zajímá mě technická funkčnost takového programu. Proto nevytvářím pouze herní logiku samotnou, ale velkou část mé práce tvoří herní *engine* (SW, který se stará o základní funkčnost, jako je správa toku programu, vykreslování grafiky a další základní funkčnost společnou pro všechny videohry).

Program po spuštění nahraje levely, které jsou uloženy pomocí textových souborů, aby byl uživatel schopný jednoduše tvořit vlastní herní prostředí. Uživatel úspěšně dokončí level tím, že pomocí zbraní (pistole, samopalu) zničí všechny terče v co nejkratším čase. Během jednoho spuštění programu lze libovolně přepínat mezi levely.

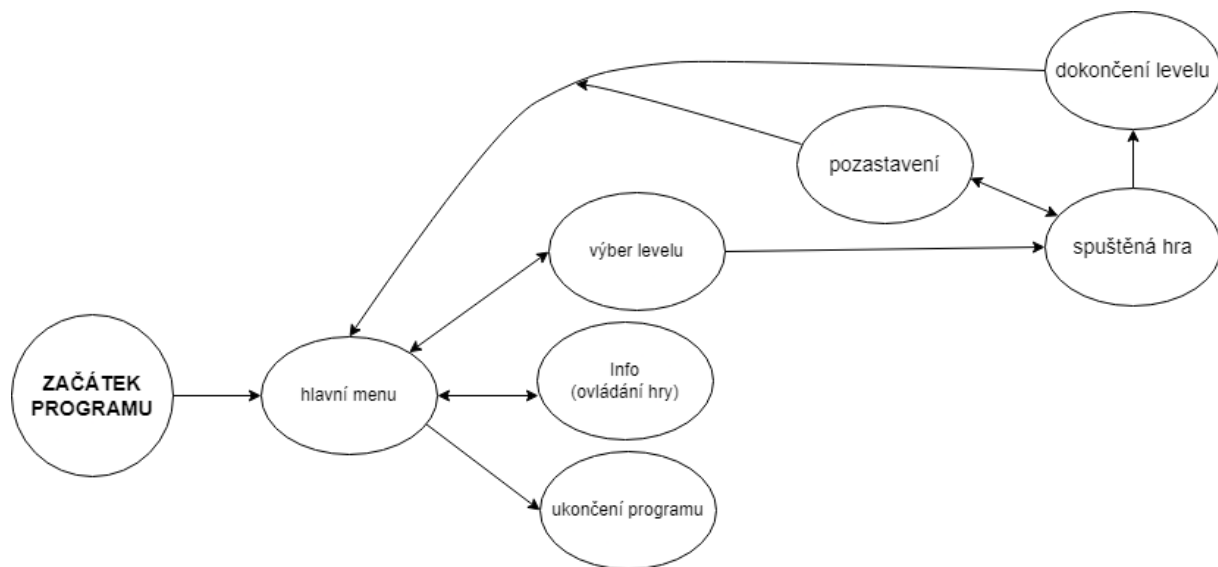


# UML Diagramy

UML (Unified Modeling Language) se používá při vývoji SW. Pomáhají rozvrhnout, jak má systém fungovat. Tj. jaké možnosti má uživatel, jak program postupuje při svém běhu, ...

UML lze použít jako jednoduchý náčrt (sketch), plán (blueprint) nebo jako programovací jazyk. Sketch je jednoduchý náčrtek, který nám pomůže abstraktně si představit, jak systém funguje. Myšlenka blueprintu je podobná, ale diagram je už mnohem detailnější a pomáhá programátorům při implementaci systému. Programovací jazyk UML se používá pro vygenerování detailní šablony implementace.

Tento sketchový UML diagram zobrazuje jednotlivé stavy programu a jak může uživatel postupovat při jeho užívání.



1. sketch UML diagram

# Rozbor řešení

## Použité technologie / nástroje

### programovací jazyk C++

Multiparadigmatický (nezaměřuje se čistě na jedno programovací paradigma, jako třeba Java - oběktově) programovací jazyk. Jedná se o “následníka” jazyka C. S trochou nadsázky by se dalo říct, že se jedná o jazyk C obohacený o objekty. Z toho vychází, že každý kód jazyka C, je validním kódem jazyka C++. Předností jazyka je vysoká rychlost. Proto se používá zejména k programování aplikací, jako jsou systémové knihovny, AAA hry a jiné programy, kde na rychlosti záleží.

Já jsem si tento jazyk na tento projekt vybral, protože má skvělou podporu ze strany knihovny SFML a tudíž není problémem najít řešení pro vzniklé problémy. S jazykem mám taky zkušenost z předešlých podobných projektů a jeho logika a syntaxe mi vyhovuje.

### knihovna SFML

Je to multiplatformní knihovna pro vývoj SW, která má API, které je velmi srozumitelné a jednoduché k použití. Já ve svém projektu využil zejména interface pro grafiku a audio.

Vybral jsem tuto knihovnu právě kvůli velmi srozumitelné dokumentaci, velkému množství tutoriálů a spoustě podpory ze strany uživatelů, kteří nabízejí pomoc při problémech. To vše dělá z této knihovny velmi “pohodlně” použitelnou knihovnu.

### Asprite

SW na editaci obrázkových souborů, který se primárně používá pro pixel art. Tento SW jsem využil pro jeho funkci, kdy z jednotlivých obrázků dokáže vytvořit sprite sheet (obraz skládající se z více malých obrázků, které obsahují jednotlivé snímky animace).

### Gimp

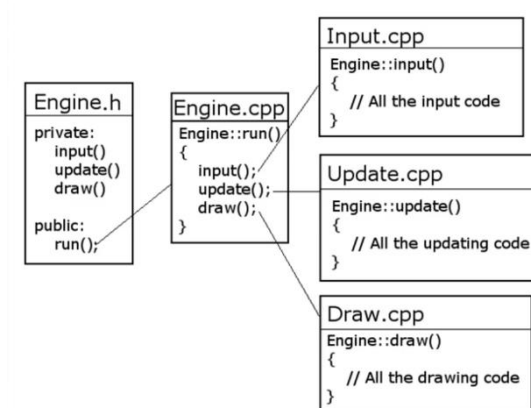
Open-source obrázkový editor. Obsahuje užitečné nástroje, ale zachovává si svoji jednoduchost k pochopení a použití.

Program jsem použil pro úpravu textur a sprite sheetů.

# Rozbor programu

## Engine

Hlavní myšlenkou třídy Engine je, že se bude starat o tok programu a jeho základní funkce. Základní funkce, které má engine na práci jsou zpracování vstupu uživatel, aktualizace herních objektů, aktualizace a vykreslování grafiky.



2. rozvržení třídy Engine

Engine je objekt zvaný singleton. To znamená, že program obsahuje pouze jednu instanci této třídy. Toho docílíme tak, že konstruktor umístíme do privátní části třídy a ve veřejné části vytvoříme statickou metodu `getEngine()`, která vrátí referenci na jedinou instanci třídy Engine, která je uložena v statické proměnné v této funkci.

```
// třída Engine má pouze jednu instanci
Engine();

public:

    void run();

    static Engine& getEngine();
    ~Engine();
};
```

```
Engine& Engine::getEngine()
{
    static Engine only_engine;
    return only_engine;
}
```

3 implementace getteru pro singleton

4 singleton, privátní konstruktor

Ve veřejné části třídy Engine je dále pouze destruktorka a metoda `run()`, kterou engine přebírá celý běh programu. Ve funkci `main()` proběhne první vytvoření engine a předání běhu programu engine.

```
1  #include "Engine.h"
2
3  int main()
4  {
5      Engine& engine = Engine::getEngine();
6      engine.run();
7      return 0;
8  }
```

5 funkce main - hlavní funkce programu

Všechny aplikace založené na eventech, jako jsou i videohry potřebují hlavní smyčku. Tou je metoda `run()`. Ta spočítá kolik času uběhlo od posledního framu a postupně zavolá všechny tři základní funkce.

```
void Engine::run()
{
    Clock frameClock;
    Time deltaTime;

    while (window.isOpen())
    {
        deltaTime = frameClock.restart();
        Vector2i mousePosition = Mouse::getPosition(window);
        mouseWorldPosition = window.mapPixelToCoords(mousePosition);

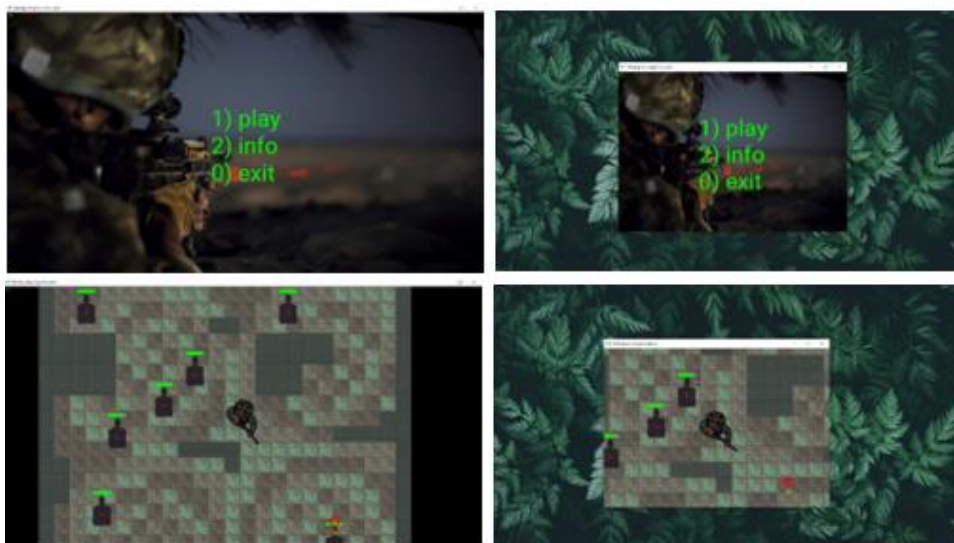
        input();
        update(deltaTime);
        draw();
    }
}
```

*6 main loop programu*

Ostatní komponenty programu, které budou obsluhovat konkrétní úkony budou spadat pod třídu `Engine`, která je bude ovládat (budou pro ni zprostředkovávat interface).

### Reakce na změnu velikosti okna

Defaultně je tato akce řešena škálováním okna, což má za následek, že při změně velikosti okna se textury roztáhnou nebo smrští, což není funkčnost, kterou chci. Jediný objekt, u které toto chování chci, je pozadí menu. U ostatních objektů chci, aby se jejich proporce nezměnili, ale změnila se část plochy která je viditelná a ty, které jsou součástí menu se přemístí vždy do středu okna.



*7 resize okna*

Každý frame se prochází fronta eventů, které se od posledního testování vytvořily. K tomu použijeme while loop a metodu `RenderWindow::pollEvent()`, která bere jako argument proměnnou typu `Event` a přebírá si ji jako referenci. Pokud je ve frontě nějaký event, tak jej do proměnné uloží a vrátí „true“. Tímto způsobem můžeme procházet eventy a dle testování jejich vlastností zjistíme, k jakému eventu vlastně došlo. Pro náš problém se změnou velikosti okna budeme testovat, zda je typ eventu „Resize“.

```
Event event;
while (window.pollEvent(event))
{
    if (event.type == Event::Closed)
    {
        currentState = GameState::EXIT;
    }
    else if (event.type == Event::Resized)
    {
        hudResizeUpdate();
    }
}
```

*8 průchod eventů za frame*

Pokud na takový event narazíme, zavoláme metodu `Engine::hudResize()`.

```
void Engine::hudResizeUpdate()
{
    view.setSize(window.getSize().x, window.getSize().y);
    // pozadí je ve fullhd a já to celý programuju na 1080p monitoru
    scaleToFullHD.x = view.getSize().x / 1920.f;
    scaleToFullHD.y = view.getSize().y / 1080.f;
    bgSprite.setScale(scaleToFullHD);

    menuText.setPosition(window.getSize().x / 2 - menuText.getGlobalBounds().width / 2,
        window.getSize().y / 2 - menuText.getGlobalBounds().height / 2);

    levelPickText.setPosition(window.getSize().x / 2 - levelPickText.getGlobalBounds().width / 2,
        window.getSize().y / 2 - levelPickText.getGlobalBounds().height / 2);

    // info stránka a její pozadí (ten obdélník)
    auto vs = view.getSize();

    infoBG.setPosition(vs.x / 2 - infoBG.getSize().x / 2,
        vs.y / 2 - infoBG.getSize().y / 2);
    infoText.setPosition(vs.x / 2 - infoText.getGlobalBounds().width / 2,
        vs.y / 2 - infoText.getGlobalBounds().height / 2 - 5);
}
```

*9 funkce pro přensatvení škálování a pozic textů*

Nejdřív změníme plochu, která se do okna renderuje, pak změníme škálování pozadí menu, aby jeho velikost odpovídala velikosti okna a poté už jenom všechny menu texty přemístíme do středu okna.

## Nahrání map do programu a načtení vybrané mapy ke hře

O správu herních map se stará třída LevelManager. Při spuštění programu se provede sken složky, kde jsou uloženy textové soubory, pomocí kterých jsou mapy uloženy. Uloží si jejich názvy a cestu k nim. Uživateli dá pak možnost spustit některou z těchto map.



10 ilustrace nahrání map

```
LevelManager::LevelManager()
{
    mapsNum = 0;
    string path = "maps/";
    for (const auto& file : directory_iterator(path))
    {
        auto filePath = file.path();
        string stringFilePath = filePath.string();
        mapNames.push_back(stringFilePath);
        int strlen = stringFilePath.size() - 9;
        menuString += to_string(++mapsNum) + ") " +
            stringFilePath.substr(5, strlen) + '\n';
        if (mapsNum == MAX_MAP_NUM) break;
    }
    for (int i = mapsNum + 1; i < MAX_MAP_NUM + 1; i++)
        menuString += to_string(i) + ") [SLOT EMPTY]\n";
    menuString += "0) exit";
}
```

11 funkce nahrání názvů map

Načtení mapy do hry pak probíhá tak, že se nejprve zkontroluje, zda nějaká mapa již není načtená. Pokud je, tak ji tento kód vymaže.

Vymaže se informace o tom, kde je zeď a kde podlaha.

Potom se smaže umístění textur pro podlahu a pro zeď.

```
void LevelManager::deleteCurrentMap()
{
    if (arrayMap != nullptr)
    {
        for (int i = 0; i < mapSize.y; i++)
        {
            delete[] arrayMap[i];
        }
        delete[] arrayMap;
        delete pVertexMap;
        arrayMap = nullptr;
        wallPositions.clear();
        mapSize.x = mapSize.y = 0;
    }
}
```

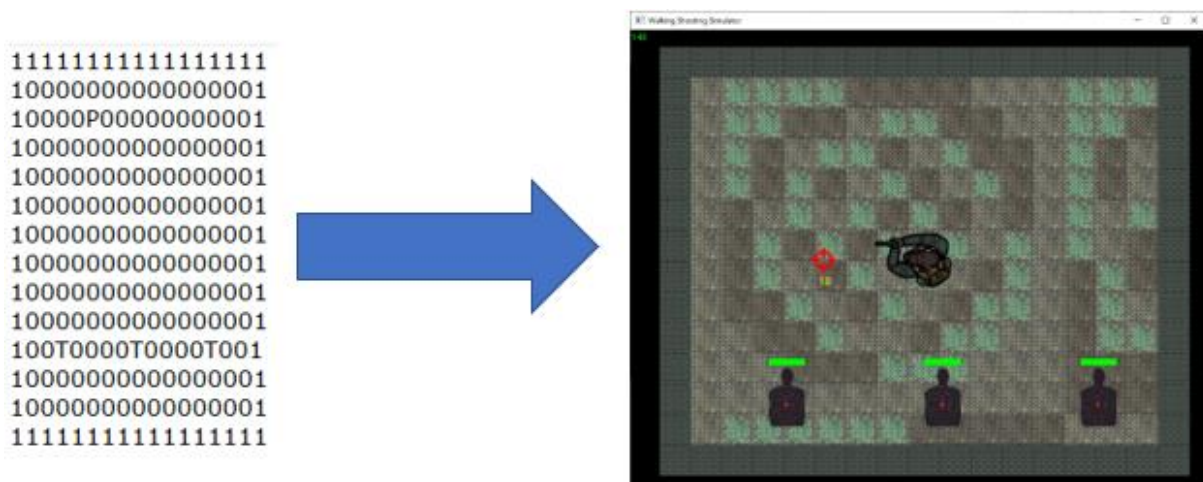
12 vymazání nahané mapy

Samotné načtení ze souboru má za úkol tato funkce:

<pre> void LevelManager::technicalInit(int mapIndex) {     ifstream inputFile(mapNames[mapIndex]);     string rowString;      // získání velikosti mapy     while (getline(inputFile, rowString))     {         ++mapSize.y;     }     mapSize.x = rowString.length();      // FileInputStream zpět na začátek     inputFile.clear();     inputFile.seekg(0, ios::beg);      deleteCurrentMap();     // vytvoření 2d pole     arrayMap = new char* [mapSize.y];     for (int i = 0; i &lt; mapSize.y; i++)     {         arrayMap[i] = new char[mapSize.x];     }      rowString.clear();      // zapisování hodnot do 2d pole     int y = 0;     while (inputFile &gt;&gt; rowString)     {         for (int x = 0; x &lt; rowString.length(); x++)         {             const char val = rowString[x];             arrayMap[y][x] = val;         }         y++;     }     inputFile.close(); } </pre>	<p>Vytvoření objektu pro načtení souboru</p> <p>Získání velikosti mapy</p> <p>Za pomoci dynamické alokace paměti vytvoříme 2D pole o velikosti mapy. (jedna dlaždice = jeden znak)</p> <p>Samotné čtení souboru a ukládání hodnot do 2D pole.</p>
--	---

13 metoda pro načtení dat mapy do 2D mapy

Ted' máme v paměti uložené informace o tom, kde je zeď, kde podlaha a také pozice terčů a výchozí pozice hráče. Ty jsou uloženy pomocí znaků „P” – hráč, „T” – terč, zeď – 1 a zbylé místo (podlaha) je vyplněno znakem „0”. Podlaha bude také na místech, kde se nachází terče a výchozí pozice hráče



14 ilustrace převodu dat na grafiku



Po tom, co je rozložení nahráno do paměti, tak se metoda `LevelManager::graphicalInit()` postará o grafické vytvoření a mapy a objektům nastaví výchozí pozice.

Pro grafiku mapy se používá třída `VertexArray`, která na každém indexu uchovává jednak souřadnice v okně programu a druhak souřadnice textury. Pomocí bodů vytvoříme čtverce v textuře, které budou odpovídat čtvercům umístěným v okně a čtverec textury se umístí do okna. Pro grafiku mapy je použit sprite sheet obsahující čtyři textury. Horní tři textury jsou kachličky a dolní je zeď.



15 sprite sheet pro podlahu a stěny

Nejdříve inicializujeme proměnnou, kde uchováváme `VertexArray`. Protože třída neobsahuje žádnou metodu pro smazání obsahu, tak použijeme dynamickou alokaci paměti a při každém nahrání nastavování nové mapy objekt znovu vytvoříme. Potom objektu řekneme, že budeme ukládat souřadnice čtverců a upravíme velikost pole na „počet dlaždic (zdi i podlaha)“ krát počet bodů ve čtverci (4). Vytvoříme proměnnou, kde budeme ukládat index prvního bodu čtverce ze čtyř.

```
void LevelManager::graphicalInit(Soldier& player, vector<Target>& targets)
{
    pVertexMap = new VertexArray();
    pVertexMap->setPrimitiveType(Quads);
    pVertexMap->resize(mapSize.x * mapSize.y * 4);
    VertexArray& va = *pVertexMap;

    static const int FLOOR_TYPES = 3;
    int currentVertex = 0;
```

16 grafika mapy - příprava



Ted' budeme postupně procházet všechny pozice mapy. Jako první přiřadíme čtverci souřadnice na mapě

```
for (int w = 0; w < mapSize.x; w++)
{
    for (int h = 0; h < mapSize.y; h++)
    {
        // přiřazení pozice bodů v mapě
        va[currentVertex + 0].position = Vector2f(w * TILE_SIZE,
            h * TILE_SIZE);
        va[currentVertex + 1].position = Vector2f((w * TILE_SIZE) + TILE_SIZE,
            h * TILE_SIZE);
        va[currentVertex + 2].position = Vector2f((w * TILE_SIZE) + TILE_SIZE,
            (h * TILE_SIZE) + TILE_SIZE);
        va[currentVertex + 3].position = Vector2f((w * TILE_SIZE),
            (h * TILE_SIZE) + TILE_SIZE);
    }
}
```

*17 grafika mapy - pozice ve VertexArray*

Potom čtverci v okně přiřadíme čtverec v textuře. Pro podlahu si pomocí generátoru pseudonáhodného čísla vygenerujeme číslo 1 až 3 a tím zvolíme texturu podlahy.

```
if (arrayMap[h][w] == '1')
{
    wallPositions.push_back(Vector2f(w * TILE_SIZE, h * TILE_SIZE));
    va[currentVertex + 0].texCoords = Vector2f(0,
        0 + FLOOR_TYPES * TILE_SIZE);
    va[currentVertex + 1].texCoords = Vector2f(TILE_SIZE,
        0 + FLOOR_TYPES * TILE_SIZE);
    va[currentVertex + 2].texCoords = Vector2f(TILE_SIZE,
        TILE_SIZE + FLOOR_TYPES * TILE_SIZE);
    va[currentVertex + 3].texCoords = Vector2f(0,
        TILE_SIZE + FLOOR_TYPES * TILE_SIZE);
}
else
{
    // použije se jedna náhodně vybraná textura ze tří textur podlahy
    srand((int)time(0) + h * w - h);
    int choosen = (rand() % FLOOR_TYPES);
    int verticalOffset = choosen * TILE_SIZE;

    va[currentVertex + 0].texCoords = Vector2f(0,
        0 + verticalOffset);
    va[currentVertex + 1].texCoords = Vector2f(TILE_SIZE,
        0 + verticalOffset);
    va[currentVertex + 2].texCoords = Vector2f(TILE_SIZE,
        TILE_SIZE + verticalOffset);
    va[currentVertex + 3].texCoords = Vector2f(0,
        TILE_SIZE + verticalOffset);
}
currentVertex += 4;
```

*18 grafika mapy - nastavení textury*

Při procházení dat mapy nastavíme pozice herních objektů. Na pozici znaku „P“ nastavíme výchozí pozici hráče a pro každé „T“ vložíme do mapy jeden terč.

```
/*
 * POZICE OBJEKTŮ
 */
Vector2f objectPosition(w * TILE_SIZE + (TILE_SIZE / 2),
                        h * TILE_SIZE + (TILE_SIZE / 2));
if (arrayMap[h][w] == 'P')
{
    player.setPosition(objectPosition);
}
else if (arrayMap[h][w] == 'T')
{
    targets.push_back(Target(objectPosition));
}
```

*19 nastavení pozice objektů*

Vykreslení mapy pak probíhá v metodě Engine::draw(), kde metodě RenderWindow::draw() předáme jako argument VertexArray. A adresu textury.

```
window.draw(*levelManager.getVertexMap(), &floorTexture);
```

*20 vykreslení mapy*

Co znamenají ty speciální znaky před argumenty? První argument má před sebou hvězdičku proto, protože LevelManager::getVertexMap() vrací ukazatel na na VertexArray, kde jsou informace o vykreslení textur. Pro uložení VertexArray používáme ukazatel právě kvůli dynamické alokaci paměti. Znak „&“ u druhého argumentu nám dá adresu, kde je proměnná uložena (ukazatel). Toto jsou předpisy přetížené metody RenderWindow::draw(), které nám SFML nabízí:

void	draw (const Drawable &drawable, const RenderStates &states=RenderStates::Default)	Draw a drawable object to the render target. <a href="#">More...</a>
void	draw (const Vertex *vertices, std::size_t vertexCount, PrimitiveType type, const RenderStates &states=RenderStates::Default)	Draw primitives defined by an array of vertices. <a href="#">More...</a>
void	draw (const VertexBuffer &vertexBuffer, const RenderStates &states=RenderStates::Default)	Draw primitives defined by a vertex buffer. <a href="#">More...</a>
void	draw (const VertexBuffer &vertexBuffer, std::size_t firstVertex, std::size_t vertexCount, const RenderStates &states=RenderStates::Default)	Draw primitives defined by a vertex buffer. <a href="#">More...</a>

*21 přetížení metody RenderWindow::draw()*

## GameObject

GameObject je základní třídou, ze které dědí objekty, které jsou herními objekty s texturou a graficky se vykreslují. V této hře jsou to třídy Soldier a Target.

GameObject dědí ze třídy Drawable, která je součástí SFML a zprostředkovává metodu draw(), která se stará o vykreslení objektu. Tuhle metodu pak můžeme přepsat a tím ji upravit pro potřeby našeho objektu.

```
class GameObject : public Drawable
{
private:
    virtual void draw(RenderTarget& target, RenderStates states) const;

protected:
    Vector2f position;
    Sprite sprite;

public:
    GameObject(float x = 0, float y = 0);

    void setPosition(float x, float y);
    void setPosition(Vector2f newPosition);

    const Sprite& getSprite();

    Vector2f getPosition();
};
```

22 předpis třídy GameObject

## hráč – třída Soldier

Herní objekt, který má interface pro ovládání pohybu, střelby a přebití. Objekt této třídy se využije jako postava, kterou hráč ovládá svým vstupem.

Pohyb je proveden tak, že v metodě Engine::input() je testován vstup pro tlačítka W (nahoru), A (doleva), S(dolů), D (doprava) a tyto informace pak předám objektu, který si data o svém pohybu uloží jako násobek Xové a Ypsilónové souřadnice vůči své rychlosti pohybu.

Tyto informace se potom využijí v metodě, která vypočítá pozici objektu po provedení pohybu

```
void Soldier::moveUp()
{
    verticalMove -= 1;
}

void Soldier::moveDown()
{
    verticalMove += 1;
}

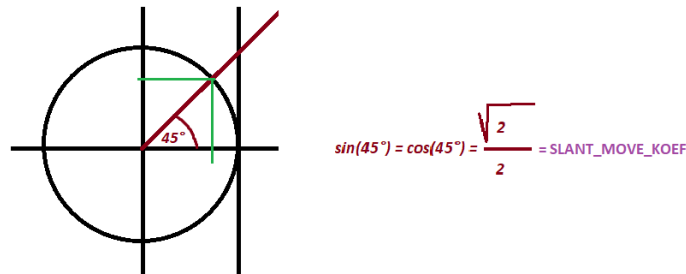
void Soldier::moveLeft()
{
    horizontalMove -= 1;
}

void Soldier::moveRight()
{
    horizontalMove += 1;
}

Vector2f Soldier::calculateNextPosition(Time dtTime)
{
    Vector2f nextPos;
    if (horizontalMove != 0 && verticalMove != 0)
    {
        float actualSpeed = MOVE_SPEED * SLANT_MOVE_KOEF * dtTime.asSeconds();
        nextPos.x = position.x + (actualSpeed * horizontalMove);
        nextPos.y = position.y + (actualSpeed * verticalMove);
    }
    else
    {
        nextPos.x = position.x + (MOVE_SPEED * dtTime.asSeconds() * horizontalMove);
        nextPos.y = position.y + (MOVE_SPEED * dtTime.asSeconds() * verticalMove);
    }
    return nextPos;
}
```

23 výpočet nadcházející pozice postavy

Pozor musíme dát na to, aby jsme při šikmém pohybu jen nedbale nepřipočítali celý krok vertikálně i horizontálně, neboť to by vyústilo v to, že při šikmém pohybu by postava dělala větší kroky (pohybovala se rychleji) než při pohybu čistě vodorovném či svislém. Proto se v takovém případě musí posun v obou směrech vynásobit koeficientem pro šikmý pohyb. Ten vypočítáme takto:



24 výpočet koef. pro šikmý pohyb

Před tím, než novou pozici nastavíme, jako pozici objektu, tak je otestováno, zda je tato nová pozice validní vůči mapě, tj. že na nové pozici nebude postava kolidovat se zdí. Pokud ano, tak se otestuje, zda probíhá kolize na ose x, y, či na obou a tomu se nová pozice přizpůsobí.

```
const vector<Vector2f>& wallPositions = levelManager.getWalls();
FloatRect twr; // temp wall rect
FloatRect pr = player.getRect().getGlobalBounds();
pr.left = playerNextPosition.x;
pr.top = playerNextPosition.y;
for (Vector2f position : wallPositions)
{
    twr.left = position.x;
    twr.top = position.y;
    twr.height = twr.width = LevelManager::TILE_SIZE;

    // pokud by hráčův následující krok vedl do zdi, tak to ho zruš
    if (pr.intersects(twr))
    {
        FloatRect withNewX(pr.left, player.getPosition().y, pr.width, pr.height);
        FloatRect withNewY(player.getPosition().x, pr.top, pr.width, pr.height);
        if (withNewX.intersects(twr))
        {
            playerNextPosition.x = player.getOldPosition().x;
        }
        if (withNewY.intersects(twr))
        {
            playerNextPosition.y = player.getOldPosition().y;
        }
    }
}
```

25 testování kolize na nadcházející pozici

Jelikož se jedná o 2D hru s pohledem „top down“ (ze shora dolů), tak je potřeba vyřešit, aby se postava otáčela za kurzorem a tím bylo možné míření. Toho docílíme tímto postupem:

Každý frame se aktualizuje poloha myši a engine si ji uloží, ale před tím je nutné si její souřadnice převést ze souřadnic na monitoru na souřadnice v rámci okna.

```
Vector2i mousePosition = Mouse::getPosition(window);
mouseWorldPosition = window.mapPixelToCoords(mousePosition);
```

Pak vypočítáme úhel ze souřadnic kurzoru a postavy.

```
void Engine::rotatePlayer()
{
    float dtX = mouseWorldPosition.x - player.getCenter().x;
    float dtY = mouseWorldPosition.y - player.getCenter().y;
    float rotation = atan2(dtY, dtX);
    rotation = rotation * (180.f / PI);
    player.rotate(rotation);
}
```

26 výpočet natočení postavy

Soldier se vždy nachází v jednom ze 4 stavů (Soldier::PlayerState). Ty slouží k tomu, aby se dalo určit, co objekt v danou chvíli vykonává za úkon. Tato informace je důležité k určení, kterou animaci pro postavu je třeba použít a druhým úkolem je kontrola, zda může postava provést nějakou akci. Například by nemělo být možné vystřelit, pokud se zrovna přebíjí. Využívám zde enumeráty.

```
class Soldier : public GameObject
{
public:
    enum PlayerState {
        IDLE = 0,
        MOVE,
        SHOOT,
        RELOAD
    };
};
```

27 stavy třídy Soldier

Ke změně stavu dochází vstupem od hráče, třeba když vystřelí, zmáčkne tlačítko pro přebíjení, nebo začne s postavou pohybovat. Po dokončení výstřelu či přebíjení pak program musí vrátit postavu do stavu IDLE. Se stavem MOVE je to jednodušší, protože ten závisí čistě na tom, zda se postava pohla či nikoli. Pro zmíněný výstřel nebo přebíjení je třeba mít časovač, který sleduje, zda již byl úkon dokončen. Také se může stát, že během přebíjení jedné zbraně bude chtít hráč vybavit zbraň jinou. Na to musí program zareagovat a zrušit přebíjení.

O sledování stavu je zodpovědná tato metoda.

```
void Soldier::stateUpdate(Time dtTime)
{
    // ukončení výstřelu a přebíjení
    if (reloadingTime.asSeconds() >= RELOAD_DURATION &&
        currentState == PlayerState::RELOAD)
    {
        currentState = PlayerState::IDLE;
        canShoot = true;
        this->animationsReset();
        this->reloadMag();
    }
    else if (shootTime.asSeconds() >= SHOOT_DURATION &&
        currentState == PlayerState::SHOOT)
    {
        currentState = PlayerState::IDLE;
        canShoot = true;
        this->animationsReset();
        if (equipedWeapon == WeaponTypes::HANDGUN)
            currHandMag--;
        else
            currRifleMag--;
    }

    // nastavení statusu
    if (currentState != RELOAD && currentState != SHOOT)
    {
        if (oldPosition != position)
        {
            currentState = PlayerState::MOVE;
        }
        else
        {
            currentState = PlayerState::IDLE;
        }
    }

    // reakce na stav
    if (currentState == PlayerState::RELOAD)
    {
        reloadingTime += dtTime;
    }
    else if (currentState == PlayerState::SHOOT)
    {
        shootTime += dtTime;
    }
}
```

Po úspěšném přebití můžeme zase střílet, doplní se nám zásobník a přejdeme do stavu IDLE

Po skončení krátké časové suspense po výstřelu zase lze vystřelit a ze zásobníku se odečte náboj. Objekt přejde do IDLE.

Pokud se zrovna nepřebíjí nebo nestřílí, tak se jako stav objektu vybere MOVE nebo IDLE a to podle toho, zda se postava od posledního framu pohla.

Probíhá přebíjení či střelba

*28 metoda pro správu stavů postavy*



## terče – třída Target

Objekty této třídy představují herního „nepřítele“, kterého má hráč za úkol zničit. To znamená, že je možné do nich střílet.

```
class Target : public GameObject
{
private:
    int hp;
    RectangleShape collideRect;
    RectangleShape currentHealthbar;
    RectangleShape healthbarBg;
    virtual void draw(RenderTarget& target, RenderStates states) const;

public:
    Target(Vector2f position);
    bool isDestroyed() const;
    void takeDamage(int damagePoints);
    const FloatRect getColliderRect() const;
    ~Target();
};
```

29 předpis třídy Target

Objekty typu Target, jak jsem si ukázali v části o načítání mapy do hry, jsou tvořeny při grafickém tvoření mapy. V konstruktoru se nastaví hodnota „hp“, která představuje počet „životů“, které terč zůstává před jeho zničením, pak je nastavena pozice, nahrána textura a nakonec je vytvořen healthbar, který nám bude ve hře graficky znázorňovat počet zbývajících životů.

```
#include "Target.h"
#include "TextureHolder.h"
#define TARGET_SCALE Vector2f(0.15f, 0.15f)
#define START_HP 5

Target::Target(Vector2f position)
{
    hp = START_HP;
    this->position = position;
    sprite.setPosition(this->position);
    sprite.setTexture(TextureHolder::getTexture("graphics/target.png"));
    sprite.setScale(TARGET_SCALE);
    auto tempRect = sprite.getGlobalBounds();
    const float weirdConst = tempRect.height / 9;
    collideRect.setPosition(tempRect.left, tempRect.top + weirdConst);
    collideRect.setSize(Vector2f(tempRect.width, tempRect.height - (weirdConst)));
    collideRect.setFillColor(Color::Transparent);
    collideRect.setOutlineThickness(2);
    healthbarBg.setSize(Vector2f(tempRect.width, weirdConst));
    healthbarBg.setPosition(this->position.x, this->position.y - 5 - weirdConst);
    healthbarBg.setFillColor(Color(142, 142, 142));
    currentHealthbar.setPosition(healthbarBg.getPosition());
    currentHealthbar.setSize(healthbarBg.getSize());
    currentHealthbar.setFillColor(Color::Green);
}
```

30 konstruktor - nastavení terče

Když terč dostane zásah tak se sníží jeho počet životů a upraví se grafické znázornění zdraví pomocí health baru. Ten sníží svoji šířku o úsek, který odpovídá jeho celkové šířce dělené počtem životů.

```
void Target::takeDamage(int damagePoints)
{
    Vector2f newSize(currentHealthbar.getSize().x - (healthbarBg.getSize().x / START_HP),
        currentHealthbar.getSize().y);
    currentHealthbar.setSize(newSize);
    hp -= damagePoints;
}
```

31 udělení poškození po zásahu

U vykreslování terče využijeme vlastnosti OOP a přepíšeme zděděnou metodu draw(). To znamená, že v části kódu, kde vykreslujeme objekty typu Target, nemusíme řešit, vykreslování všech komponent a prostě zavoláme metodu na celý objekt.

```
for (Target& t : targets)
{
    window.draw(t);
}
```

32 vykreslení projektilů

To, jak se bude objekt vykreslovat můžeme tedy upravit pomocí zmíněného přepsání metody draw. V naší metodě Target::draw() pošleme na vyrendrování sprite, který obsahuje texturu samotného terče a krom toho také kousek nad terčem vykreslíme dva obdélníky. Šedý je pozadí healthbaru a ukazuje jeho původní (celkovou) šířku a zelený na něm ukazuje současný stav životů.

```
void Target::draw(RenderTarget& target, RenderStates states) const
{
    target.draw(sprite, states);
    //target.draw(collideRect, states);
    target.draw(healthbarBg, states);
    target.draw(currentHealthbar, states);
}
```

33 přepsání zděděné metody draw()



36 terč - nezasažený



35 terč po 1 zásahu



34 terč po 4 zásazích



## TextureHolder – načítání obrázků ze souborů

TextureHolder je třída, která netvoří objekty a je využívána pouze jedna její statická metoda, která má za úkol vrátit odkaz na texturu. Cílem je, aby se každá textura do programu nahrála pouze jednou a uloží se do paměti, kdybychom ji chtěli třeba znovu zpřístupnit. Tenhle problém šel vyřešit i běžnou funkcí a nebylo nutno tvořit třídu, ale bylo to pohodlnější z hlediska importování do ostatních souborů.

```
#pragma once
#include <SFML/Graphics.hpp>
using sf::Texture;
using std::string;

class TextureHolder
{
public:
    static Texture& getTexture(const string& filename);
};
```

*37 předpis TextureHolder*

Objekt typu map, je slovník. Proměnná je statická, tudíž její hodnota se neztrácí. Jako klíč je hodnota cesta k textuře na disku a hodnota je nahraná textura. Nejdřív se funkce podívá, zda texturu již dříve nenahrála a pokud ano, tak na ni vrátí odkaz. V opačném případě ji nahraje ze souboru a uloží na pozdější použití.

```
using std::map;

Texture& TextureHolder::getTexture(const string& filename)
{
    static map<string, Texture> textureStorage;

    auto pair = textureStorage.find(filename);

    if (pair != textureStorage.end())
    {
        return pair->second;
    }
    else
    {
        auto& newTexture = textureStorage[filename];
        newTexture.loadFromFile(filename);
        return newTexture;
    }
}
```

*38 načítání a ukládání textur na později*

## Animace

Knihovna SFML nabízí možnost zobrazení jen určité části z textury. Obdélník, kterým se určí část textury k zobrazení se předává, jako druhý parametr pro metodu `draw()`, kterou může naše třída `Soldier` přepsat, jelikož dědí z třídy `GameObject`, jenž je potomkem `Drawable`. Třída `Animation` drží informaci o tomto obdélníku a náležitě ho posouvá v rámci `sprite sheetu`.

```
class Animation
{
private:
    Time duration;
    Time timeForFrame;
    Time currTime;
    int numOfframes;
    int currFrameIndex;
    IntRect currFrame;
public:
    Animation();
    void setup(int frameNum, int textureWidth, int textureHeight, Time dur);
    const IntRect& update(Time dtTime);
    void reset();
    ~Animation();
};
```

39 předpis Animation

Metodou `Animation::setup()` provedeme inicializaci objektu. Tento úkon je dán do metody jiné, než je konstruktor. Důvodem je, že objekty typu `Animation` jsou uloženy v enginu a ten je vytvoří již při svém vytvoření, takže by bylo třeba využít poněkud velmi zmateného zápisu, kde by parametry pro konstruktor `Animation` musely být psány mezi název konstruktoru `Engine` a tělo toho konstruktoru. Proto je přehlednější použít to takhle a reálně to na program nemá žádný vliv.

```
void Animation::setup(int frameNum, int textureWidth, int textureHeight, Time dur)
{
    this->duration = dur;
    this->reset();
    currFrame.height = textureHeight;
    currFrame.width = textureWidth / frameNum;
    numOfframes = frameNum;
    timeForFrame = seconds(dur.asSeconds() / frameNum);
}

void Animation::reset()
{
    currTime = Time::Zero;
    currFrame.top = currFrame.left = 0;
    currFrameIndex = 0;
}
```

40 příprava objekt Animation

Po tom, co předáme třídě informace, kolik textur se ve sprite sheetu nachází, jaké mají rozměry a jaká je doba trvání jednoho přehrání animace, můžeme použít metodu `update`, která bere jako argument čas od posledního framu a vrací čtverec s parametry, které odpovídají textuře ve sprite sheet.

Čas od posledního framu přičteme k času, po který se vykresluje současně vybraná textura a pokud dosáhla limitu, tak časovač vynulujeme a posuneme texturu na další (případně vrátíme na začátek)

```
const IntRect& Animation::update(Time dtTime)
{
    currTime += dtTime;
    if (currTime >= timeForFrame)
    {
        currTime = Time::Zero;
        currFrameIndex++;
        if (currFrameIndex >= numOfFrames)
        {
            currFrameIndex = 0;
        }
        currFrame.left = currFrameIndex * currFrame.width;
    }
    return currFrame;
}
```

41 výběr textury v animaci



42 sprite sheet přebití pušky



43 přebíjení pušky - animace

## Střelba – třída Bullet

Klíčovým prvkem hry je možnost střelby. Potřebujeme ji ke zničení terčů a tudíž ke splnění cíle hry. V tom nám pomůže třída Bullet, která představuje vystřelenou kulku, jenž je testována na kolizi s terčem. Bullet není potomkem GameObject. To protože GameObject nabízí možnosti nastavení pozice a komponentu Sprite, která se používá při užívání textur. To mi pro naši kulku nepotřebujeme. Kulka prezentována pohybujícím se obdélníkem, který nemá žádnou texturu a má pouze rovný pohyb jedním směrem, než narazí do nějaké překážky a zanikne.

Když hráč vydá příkaz ke střelbě, tak se zkontroluje, zda postava zrovna nepřebíjí, má dostatek munice a zda byla dokončena předchozí střelba. Pokud je vybavenou zbraní útočná puška, tak nemusí hráč kliknout pro každý výstřel zvlášť, ale může podržet levé tlačítko myši pro automatickou střelbu.

Pokud tedy hráč úspěšně vystřelí, vytvoří se nový objekt typu Bullet, kterému se do konstrukturu předá pozice hráče a pozice kurzoru.

```
if (Mouse::isButtonPressed(Mouse::Button::Left)
    && player.getState() != Soldier::PlayerState::RELOAD && player.shootAvailable())
{
    // RILFE je automatická zbraň, ostatní jsou poloautomaty
    if (!leftMouseOnHold || player.getWeapon() == Soldier::WeaponTypes::RIFLE)
    {
        if (player.shoot())
        {
            bullets.push_back(Bullet(player.getCenter(), mouseWorldPosition));
            soundManager.playShoot();
        }
    }
}
// jestli je myš držena
leftMouseOnHold = Mouse::isButtonPressed(Mouse::Button::Left);
```

44 střelba - vytvoření objektu Bullet

Kulka se bude pohybovat po polopřímce, která začíná na pozici postavy a prochází pozicí kurzoru v momentě výstřelu. V konstruktoru Bullet::Bulet() se nastaví pozice na první předaný argument.

```
□ Bullet::Bullet(Vector2f start, Vector2f target)
{
    bulletBody.width = bulletBody.height = 4;
    position = start;
    targetPosition = target;

    shape.setSize(Vector2f(bulletBody.width, bulletBody.height));
    shape.setOutlineThickness(2);
    shape.setOutlineColor(Color::Red);

    bulletBody.left = start.x;
    bulletBody.top = start.y;
```

45konstruktor bullet - příprava

Pak se vypočítá poměr mezi Xovou a Yovou vzdáleností mezi těmito dvěma souřadnicemi. Uděláme absolutní hodnotu poměru. Pomocí tohoto poměru potom vypočítáme rychlost, kterou se bude pohybovat kulka svisle a vodorovně.

```

float gradient = (start.x - target.x) / (start.y - target.y);
gradient = gradient < 0 ? -gradient : gradient;

float ratio = Bullet::SPEED / (1 + gradient);
frameTravel.x = ratio * gradient;
frameTravel.y = ratio ;

if (target.x < start.x)
{
    frameTravel.x *= -1;
}
if (target.y < start.y)
{
    frameTravel.y *= -1;
}

```

46 konstruktor Bullet - výpočet zklonu

Potom už stačí na kulku každý frame zavolat metodu update, která vykonává její pohyb.

```

void Bullet::update(Time dtTime)
{
    position.x += frameTravel.x * dtTime.asSeconds();
    position.y += frameTravel.y * dtTime.asSeconds();

    shape.setPosition(position);
    bulletBody.left = position.x;
    bulletBody.top = position.y;
}

```

47 pohyb náboje

Testování kolize probíhá v metodě Engine::wallCollisions(). Z objektu LevelManager nahrajeme pozice všech stěn. Ty potom procházíme ve „for“ smyčce a tuto pozici nastavujeme čtverci.

```

const vector<Vector2f>& wallPositions = levelManager.getWalls();
FloatRect twr; // temp wall rect
FloatRect pr = player.getRect().getGlobalBounds();
pr.left = playerNextPosition.x;
pr.top = playerNextPosition.y;
for (Vector2f position : wallPositions)
{
    twr.left = position.x;
    twr.top = position.y;
    twr.height = twr.width = LevelManager::TILE_SIZE;
}

```

48 loopování zkrze všechny stěny na mapě

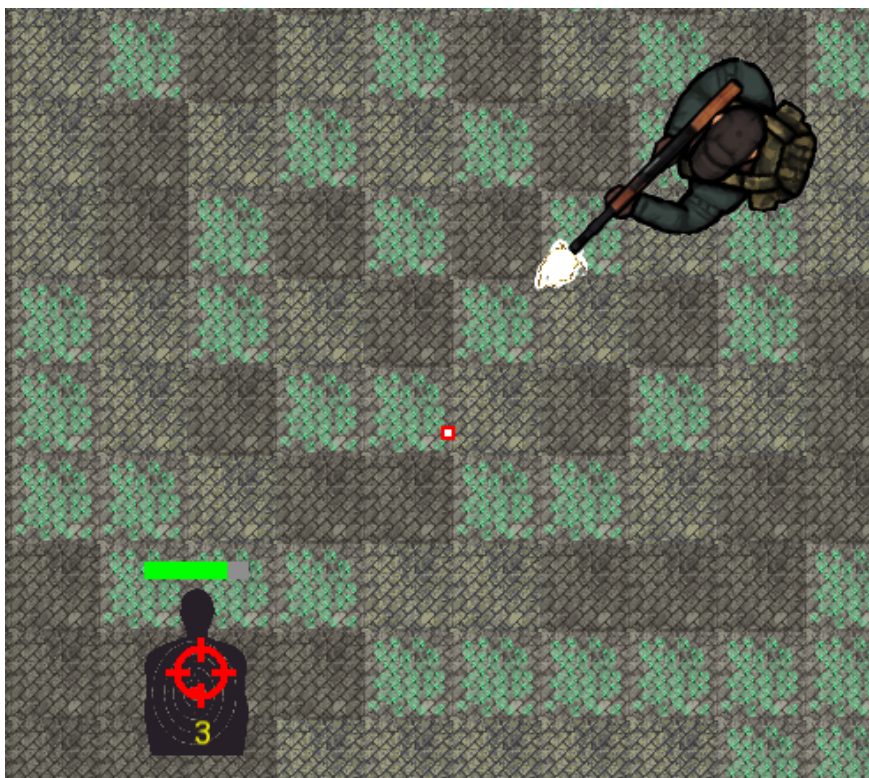


V této smyčce pak vytvoříme druhou smyčku, kde budeme procházet právě aktivní náboje a testovat jejich kolizi vůči stěnám. Pokud narazíme na kolizi, tak kulku odstraníme.

```
// náboje
auto it = bullets.begin();
while (it != bullets.end())
{
    // zeď
    if (it->getRect().intersects(twr))
    {
        it = bullets.erase(it);
        soundManager.playWallHit();
    }
    else
    {
        ++it;
        break;
    }
}
```

49 kolize kulky se stěnou

Jak již bylo zmíněno, pro vykreslení kulky se nepoužívá sprite s texturou, ale pouze obdélník s červeným obrysem. Ten je reprezentován třídou RectangleShape, která je také potomkem Drawable, což znamená, že ji lze vykreslit.

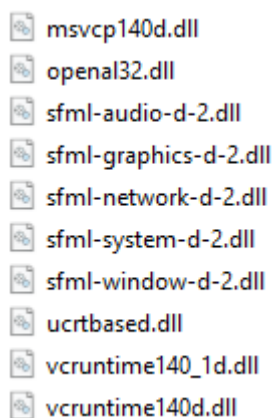


50 ukázka střelby

# Systemové požadavky

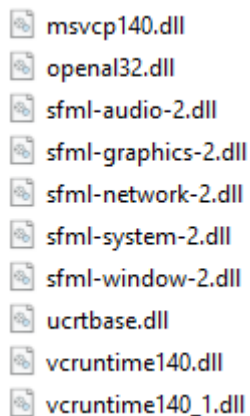
Program je tvořen a kompilován pro platformu Windows 10 (64 bit) a novější.

Pro spuštění programu jsou potřeba tyto .dll knihovny:



msvcp140d.dll  
openal32.dll  
sfml-audio-d-2.dll  
sfml-graphics-d-2.dll  
sfml-network-d-2.dll  
sfml-system-d-2.dll  
sfml-window-d-2.dll  
ucrtbased.dll  
vcruntime140\_1d.dll  
vcruntime140d.dll

*52 knihovny verze Debug*



msvcp140.dll  
openal32.dll  
sfml-audio-2.dll  
sfml-graphics-2.dll  
sfml-network-2.dll  
sfml-system-2.dll  
sfml-window-2.dll  
ucrtbase.dll  
vcruntime140.dll  
vcruntime140\_1.dll

*51 knihovny pro Release*

Knihovny jsou přiloženy v adresáři projektu u spustitelných souborů.

# Závěr

Výsledkem práce je funkční videohra. Hra obsahuje možnost hrát uživatelsky vytvořené mapy. Jsou implementovány dvě zbraně s odlišnými vlastnostmi, které může hráč použít. Nepřátelé jsou implementováni v podobě terčů. Program umí nahrát textové soubory s informacemi o mapě. Programové ukládání do souborů není implementováno, protože program neobsahuje žádný generátor levelů přímo v sobě a jiná data program nepotřebuje sdílet mezi jednotlivými spuštěními. Hra také obsahuje funkční zvukové efekty pro chůzi, střelbu a kolize kulky zdí a terčem.

Program se nachází v plně hratelném stavu. Lze tvořit vlastní mapy, které lze úspěšně dohrát. Hra neobsahuje známé chyby, které by způsobovali její nehratelnost.

Celý projekt je open source a je dostupný na stránce [github.com](https://github.com) a kdokoli si jej může stáhnout, upravovat a libovolně s ním nakládat. Díky tomu může kdokoli program libovolně rozšířit a stavět na něm. Program by se dal rozšířit například o generátor map, který by uměl by tvořit a ukládat uživatelem vytvořené mapy. Hra by se také mohla zlepšit v ohledu testování kolizí. Ty v současné době závisí na rychlosti počítače. Na moderních počítačích to nečiní žádný problém, ale toto řešení má jisté nedostatky.

Stěžejní funkčností programu je fungující engine, který je hlavní částí projektu. Ten se povedlo úspěšně implementovat a nejsou v něm žádné známé problémy. Umí se zpracovat uživatelský vstup, ovládat komponenty, které jsou do něj zavedeny a vykreslování grafiky je také funkční.

Práce na tomto projektu mi dala spoustu zkušeností ve vývoji SW. Projekt považuji za úspěšný, dosáhlo se hlavních cílů a splnil moje očekávání. To ale nemusí nutně znamenat absolutní konec projektu. Stále jsou funkce, které bych chtěl implementovat nebo na něj může navázat někdo jiný, díky veřejně dostupnému zdrojovému kódu.



# Seznam ilustrací

1. sketch UML diagram .....	9
2. rozvržení třídy Engine .....	11
3 implementace getteru pro singleton .....	11
4 singleton, privátní konstruktor .....	11
5 funkce main - hlavní funkce programu .....	11
6 main loop programu .....	12
7 resize okna .....	12
8 průchod eventů za frame .....	13
9 funkce pro přensatvení škálování a pozic textů .....	13
10 ilustrace nahrání map .....	14
11 funkce nahrání názvů map .....	14
12 vymazání nahrané mapy .....	14
13 metoda pro načtení dat mapy do 2D mapy .....	15
14 ilustrace převodu dat na grafiku .....	15
15 sprite sheet pro podlahu a stěny .....	16
16 grafika mapy - příprava .....	16
17 grafika mapy - pozice ve VertexArray .....	17
18 grafika mapy - nastavení textury .....	17
19 nastavení pozice objektů .....	18
20 vykreslení mapy .....	18
21 přetížení metody RednerWindow::draw() .....	18
22 předpis třídy GameObject .....	19
23 výpočet nadcházející pozice postavy .....	19
24 výpočet koef. pro šikmý pohyb .....	20
25 testování kolize na nadcházející pozici .....	20
26 výpočet natočení postavy .....	21
27 stavy třídy Soldier .....	21
28 metoda pro správu stavů postavy .....	22
29 předpis třídy Target .....	23
30 konstruktor - nastavení terče .....	23
31 udělení poškození po zásahu .....	24
32 vykreslení projektilů .....	24
33 přepsání zděděné metody draw() .....	24
34 terč po 4 zásazích .....	24
35 terč po 1 zásahu .....	24
36 terč - nezasažený .....	24

37 předpis TextureHolder .....	25
38 načítání a ukládání textur na později .....	25
39 předpis Animation .....	26
40 příprava objekt Animation.....	26
41 výběr textury v animaci.....	27
42 sprite sheet přebití pušky .....	27
43 přebíjení pušky - animace .....	27
44 střelba - vytvoření objektu Bullet .....	28
45konstruktor bullet - příprava .....	28
46 konstruktor Bullet - výpočet zklonu.....	29
47 pohyb náboje .....	29
48 loopování zkrze všechny stěny na mapě .....	29
49 kolize kulky se stěnou.....	30
50 ukázka střelby .....	30
51 knihovny pro Release .....	31
52 knihovny verze Debug .....	31
53 adresář se spustitelným programem .....	38
54 přidání textového souboru do složky maps .....	38
55 "nakreslená" mapa .....	39
56 vytvořená mapa je na výběr ke hře.....	39
57 ukázka vytvořené mapy ve hře .....	40
58 hlavní menu .....	41
59 stránka s popisem ovládání .....	41
60 výběr hratelné mapy .....	42
61 šikmý pohyb s nárazem do stěny.....	42
62 postava s puškou .....	43
63 postava s pistolí .....	43
64stav zásobníku .....	43
65 průběh přebití .....	43
66 obrazovka s časem dokončení levelu .....	43

# Zdroje

- [1.] *Mistrovství v C++ 4. aktualizované vydání*, Computer Press, 2013 [kniha]  
[2022-04-19]
- [2.] *GeeksforGeeks* [online] [2022-04-19] Dostupné z:  
<https://www.geeksforgeeks.org/c-plus-plus/>
- [3.] *Cplusplus.com* [online] [2022-04-19] Dostupné z:  
<https://www.cplusplus.com/doc/tutorial/>
- [4.] *Fórum knihovny SFML* [online] [2022-04-19] Dostupné z:  
<https://en.sfml-dev.org/forums/>
- [5.] *Dokumentace knihovny SFML* [online]. [2022-04-19] Dostupné z:  
<https://www.sfml-dev.org/documentation/2.5.1/>
- [6.] *Fórum stackoverflow* [online] [2022-04-19] Dostupné z: <https://stackoverflow.com/>
- [7.] *C++ SFML Simple Apps and Games*, YouTube Playlist, Suraj Sharma [online]  
[2022-04-19] Dostupné z:  
[https://www.youtube.com/playlist?list=PL6xSOsbVA1eb\\_QqMTTcql\\_3PdOiE928up](https://www.youtube.com/playlist?list=PL6xSOsbVA1eb_QqMTTcql_3PdOiE928up)
- [8.] *SFML 2.4 For Beginners*, YouTube Playlist, Hilze Vonck [onlin] [2022-04-19]  
Dostupné z:  
[https://www.youtube.com/playlist?list=PL21OsoBLPpMOO6zyVlxZ4S4hwkY\\_SLRW9](https://www.youtube.com/playlist?list=PL21OsoBLPpMOO6zyVlxZ4S4hwkY_SLRW9)
- [9.] *OpenGameArt.org* [online] [2022-04-19] Dostupné z:  
<https://opengameart.org/content/animated-top-down-survivor-player>
- [10.] *Subpng* [online] [2022-04-19] Dostupné z: <https://www.subpng.com/png-pglfg3/>

# Přílohy

- Návod k použití (je součástí dokumentace)
- samotný projekt pro Visual Studio 2022 (obsahuje i spustitelný program):
  - <https://github.com/DanielKrejska/WalkingShootingSimulator>
- Samostatný spustitelný program
  - [https://github.com/DanielKrejska/WalkingShootingSimulator/releases/tag/public\\_version\\_1](https://github.com/DanielKrejska/WalkingShootingSimulator/releases/tag/public_version_1)



PROG2i

## **Návod k použití**

***Desktopová hra - Walking shooter***

**DANIEL KREJSKA**

**V4C**

**Profilová část maturitní zkoušky  
MATURITNÍ PRÁCE**

**BRNO 2022**

# Návod k použití

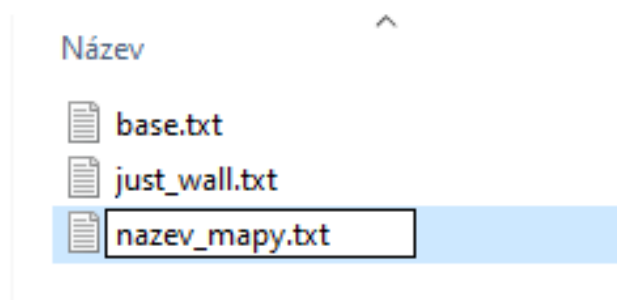
## Uživatelská tvorba mapy

Postup přidání vlastní hratelné mapy je velmi jednoduchý. Stačí přidat textový soubor (\*.txt) do složky „maps“, která se nachází v adresáři s programem.

fonts	23.11.2021 21:04	Složka souborů	
graphics	22.02.2022 16:17	Složka souborů	
maps	03.12.2021 16:45	Složka souborů	
sounds	10.03.2022 16:38	Složka souborů	
msvcp140d.dll	10.03.2022 16:54	Rozšíření aplikace	978 kB
openal32.dll	08.05.2018 22:40	Rozšíření aplikace	654 kB
sfml-audio-2.dll	15.10.2018 23:09	Rozšíření aplikace	989 kB
sfml-graphics-2.dll	15.10.2018 23:10	Rozšíření aplikace	793 kB
sfml-network-2.dll	15.10.2018 23:09	Rozšíření aplikace	129 kB
sfml-system-2.dll	15.10.2018 23:09	Rozšíření aplikace	49 kB
sfml-window-2.dll	15.10.2018 23:09	Rozšíření aplikace	121 kB
ucrtbased.dll	10.03.2022 16:54	Rozšíření aplikace	1 745 kB
vcruntime140_1d.dll	10.03.2022 16:54	Rozšíření aplikace	59 kB
vcruntime140d.dll	10.03.2022 16:54	Rozšíření aplikace	129 kB
WalkingShootingSimulator.exe	18.04.2022 11:27	Aplikace	97 kB

53 adresář se spustitelným programem

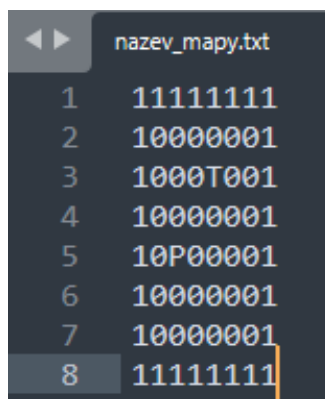
Zde přidáme nebo vytvoříme nový textový soubor, kde název souboru značí označení mapy. Program umí při jednom spuštění maximálně 9 map. Map může být i méně.



54 přidání textového souboru do složky maps

Mapu “nakreslíme” do souboru pomocí znaků

- 1 – stěna
- 0 – podlaha/výplň (podlaha je automaticky umístěna i na místa, kde je terč a výchozí pozice hráče)
- T – terč
- P – výchozí pozice hráče



```
nazev_mapy.txt
1  11111111
2  10000001
3  1000T001
4  10000001
5  10P00001
6  10000001
7  10000001
8  11111111
```

55 "nakreslená" mapa

Aby mapa byla bez problémů hratelná, musí se dodržet tato pravidla:

- je obdélníkového tvaru
- obsahuje výchozí pozici hráče
- obsahuje alespoň jeden terč
- na okrajích je ohraničena zdí

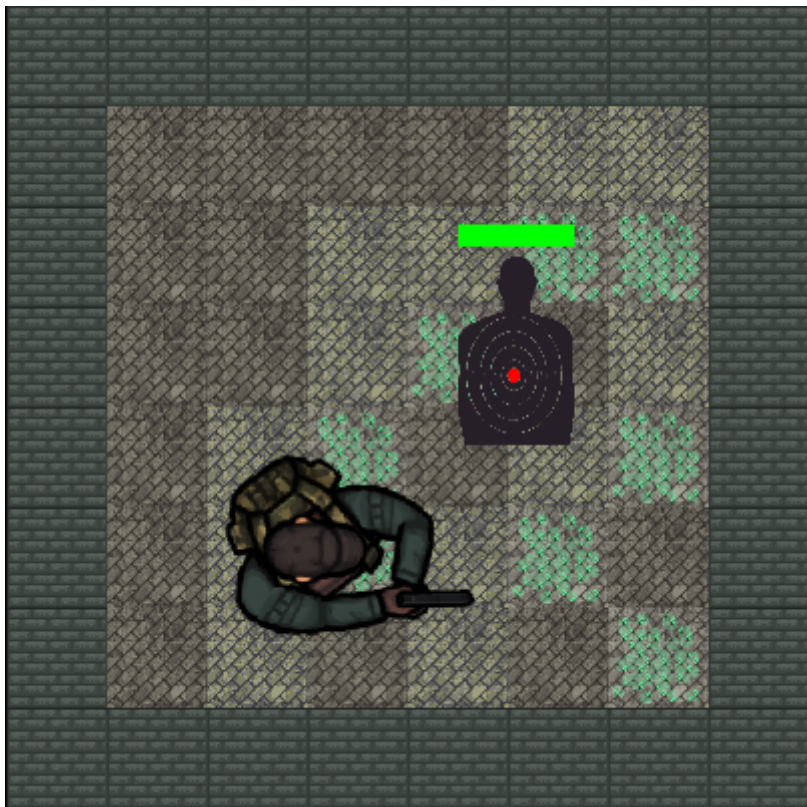
Mapa pak bude dostupná ke spuštění ve výběru herní mapy v programu.



```
1) base
2) just_wall
3) nazev_mapy
4) [SLOT EMPTY]
5) [SLOT EMPTY]
6) [SLOT EMPTY]
7) [SLOT EMPTY]
8) [SLOT EMPTY]
9) [SLOT EMPTY]
0) exit
```

56 vytvořená mapa je na výběr ke hře

Při správném postupu bude mapa hratelná.



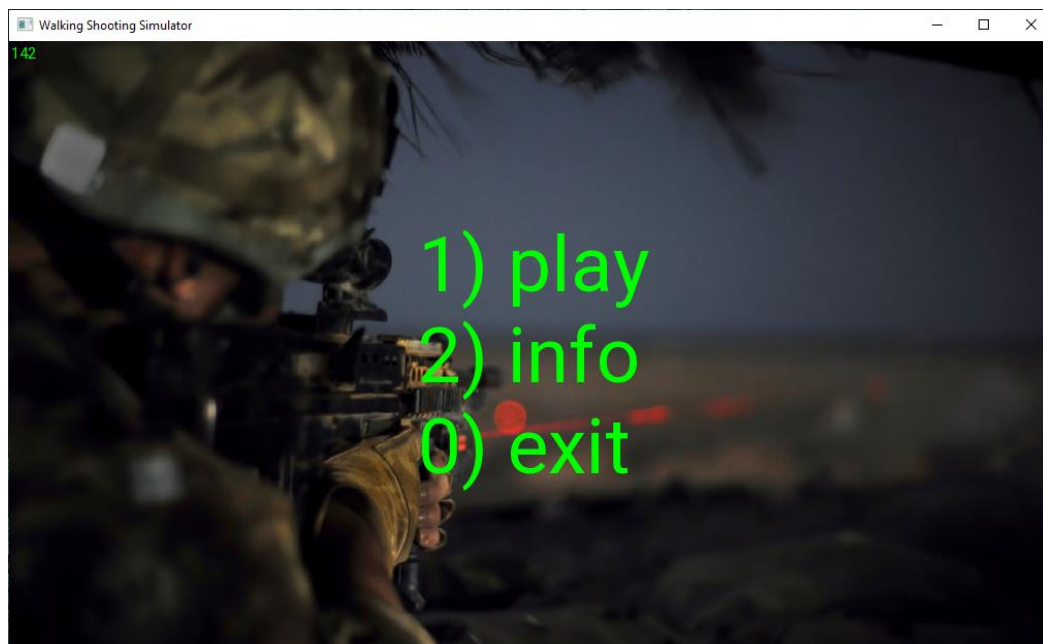
*57 ukázka vytvořené mapy ve hře*



## Pohyb v menu

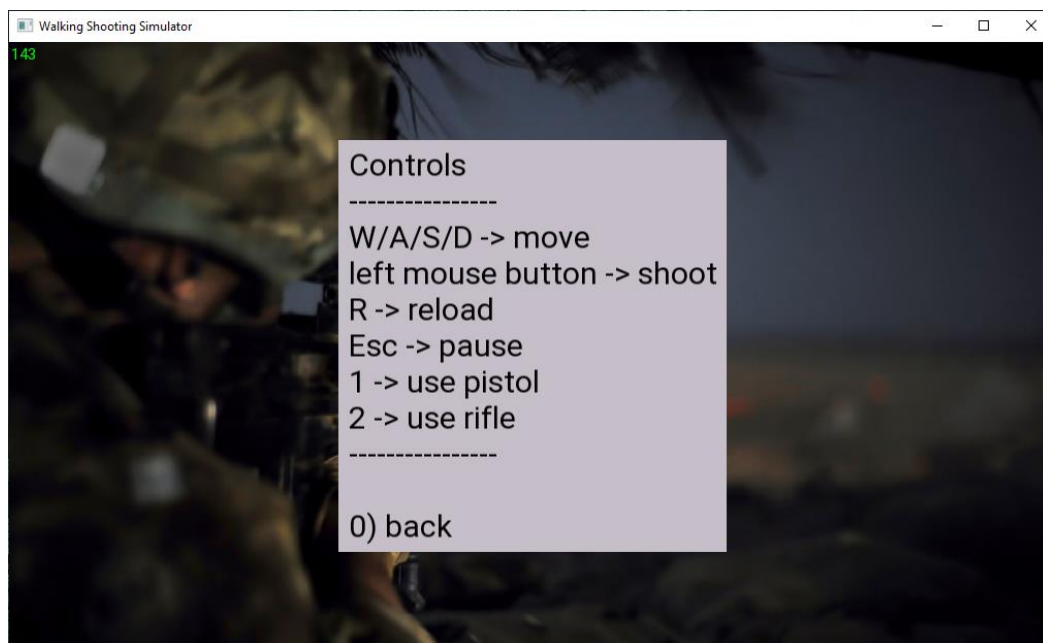
Skrze menu se v programu pohybujeme pomocí kláves 0-9 v horní řadě klávesnice nebo pomocí numpadu.

Při spuštění programu se dostaneme do Hlavního menu, zde máme na výběr pokračovat na výběr mapy, zobrazit informace o ovládání nebo program ukončit.



58 hlavní menu

Stránka „info“ nám nabízí přehled ovládání hry a možnost vrátit se zpět do hlavní nabídky.



59 stránka s popisem ovládání

Ve výběru mapy máme možnost vybrat mapu, která se má nahrát, nebo se můžeme vrátit do hlavního menu.



60 výběr hratelné mapy

## Ovládání hry

Postava se pohybuje pomocí kláves „W/A/S/D“. Pohybovat se může svisle, vodorovně i šikmě. Pokud se pohybuje šikmě a narazí do stěny, tak se bude pohybovat pouze směre, ve kterém nestojí stěna.



61 šikmý pohyb s nárazem do stěny

Po spuštění hry má postava vybavenou pistoli. Pistole má 10 nábojů a lze z ní střílet poloautomaticky. Takže pro každý jednotlivý výstřel je třeba kliknout samostatně. Puška má 30 nábojů v zásobníku a umí střílet i automaticky, takže u střelby stačí podržet levé tlačítko myši a puška bude opakovaně provádět střelbu.

Mezi zbraněmi lze přepínat pomocí kláves "1" pro pistoli a "2" pro pušku.



63 postava s pistolí



62 postava s puškou

Stav zásobníku sleduje číslo, které je umístěno pod crosshaimem.



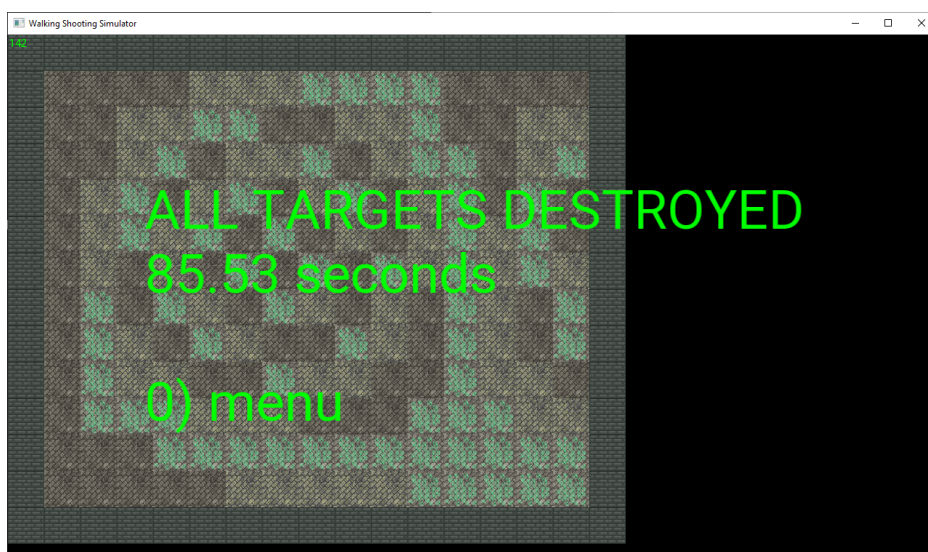
64stav zásobníku

Po vyprázdnění zásobníku nelze ze zbraně vystřelit, dokud ji nepřebijeme. Přebití zbraně trvá 2 sekundy a provedeme jej stisknutím klávesy "R".



65 průběh přebití

Cíle hry dosáhneme tak, že zničíme všechny terče na mapě. O splnění cíle nás informuje menu, které nám ukáže dosažený čas a dá možnost vrátit se do hl. menu.



66 obrazovka s časem dokončení levelu