Herní vývoj pomocí nástroje SFML

Václav Fencl

Bakalářská práce
2025

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2024/2025

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Václav Fencl**
Osobní číslo:	**A21051**
Studijní program:	**B0613A140020 Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Herní vývoj pomocí nástroje SFML**
Téma práce anglicky:	**Game Development Using the SFML Tool**

Zásady pro vypracování

1. Vypracujte literární rešerši na téma vývoj her.
2. Teoreticky popište open source nástroj Simple and Fast Multimedia Library (SFML).
3. Popište výhody oproti jiným konkurenčním nástrojům.
4. Na vhodném praktickém příkladu demonstrujte možnosti vývojového nástroje (tvorba interakcí, animací, zvuku...).
5. Výstupy, vizualizace výsledku a postupy vhodně popište.

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. MARTIN, Robert C., 2009. Clean Code: A Handbook of Agile Software Craftsmanship. 2nd ed. Prentice Hall. ISBN 9780132350884.
2. BARBIER, Maxime, 2015. SFML Blueprints: Sharpen Your Game Development Skills and Improve Your C++ and SFML Knowledge with Five Exciting Projects. Packt Publishing. ISBN 9781784395773. 1784395773.
3. PUPIUS, Raimondas, 2015. SFML Game Development By Example. Packt Publishing. ISBN 9781785283000. 1785283006.
4. CHANDLER, Heather a Rafael CHANDLER, 2011. Fundamentals of Game Development. 3rd ed. Jones & Bartlett Learning. ISBN 9780763778958. 0763778958.
5. SHEKAR, Siddharth, 2019. C++ Game Development By Example. Sonar Publishing. ISBN 9781789537345.

Vedoucí bakalářské práce: **Ing. Adam Ulrich**
Ústav matematiky

Datum zadání bakalářské práce: **25. července 2025**
Termín odevzdání bakalářské práce: **22. srpna 2025**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 5. srpna 2025

PROHLÁŠENÍ AUTORA BAKALÁŘSKÉ PRÁCE

Beru na vědomí, že

- odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení bakalářské práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – bakalářskou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji, že

- jsem na bakalářské práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.
- Prohlašuji, že při tvorbě této práce jsem použil nástroj generativního modelu AI Chat GPT; https://chatgpt.com za účelem kvalitnější formulace textu. Po použití tohoto nástroje jsem provedl kontrolu obsahu a přebírám za něj plnou zodpovědnost.

Ve Zlíně, dne ....

podpis autora

ABSTRAKT

Tato bakalářská práce řeší téma herního vývoje pomocí nástroje SFML, jeho moduly a metody. Nejprve popisuje jeho teoretické fungování. Následně vysvětluje praktické využití této knihovny a problémy se kterými se uživatel může potýkat pomocí několika menších demonstračních aplikací, které se věnují individuálním funkcionalitám SFML.

K řešení byla použita knihovna SFML verze 2.6.2 a 3.0.0 a vývojové prostředí Visual Studio 2022.

Výsledkem této práce je několik menších aplikací a hra Pong, která tyto funkcionality spojila do fungujícího finálního produktu. Přínosem práce je ukázat, že SFML představuje vhodný nástroj pro výukové účely a zároveň poskytuje začínajícím vývojářům vhled do základních principů vývoje her v C++.

Klíčová slova: SFML, herní vývoj, C++, pong, 2D hry, herní smyčka

ABSTRACT

This bachelor's thesis describes game development using the SFML library, its modules and functions. It outlines functionality of each individual module on a theoretical level. Thesis also shows practical use, along with common problems an average user may encounter via set of smaller applications that focuses on individual functionalities of SFML.

The practical solution uses SFML version 2.6.2 & 3.0.0 and Visual Studio 2022.

This thesis results in a set of smaller applications and fully functional game Pong that combined multiple functionalities demonstrated by those smaller applications. The thesis shows that SFML is a suitable tool for educational purposes while also providing beginner developers with a practical insight into the fundamental principles of game programming in C++.

Keywords: SFML, game development, C++, pong, 2D games, game loop

Rád bych vyjádřil své nejupřímnější díky všem, kteří mi věnovali čas, ochotu a pomohli s bakalářskou prací. Především děkuji Ing. Adamu Ulrichovi za ukázkové vedení práce, odborné rady a neuvěřitelnou trpělivost. Dále bych chtěl poděkovat své rodině za podporu po celou dobu mého studia.

Obsah

ÚVOD

Tato bakalářská práce má za úkol přiblížit čtenáři knihovnu SFML (Simple and Fast Multimedia Library) a její místo ve světě vývoje her. Pro tento úkol bude práce definovat vývoj her a jejich důležitost ve společnosti, také bude definovat co to vlastně SFML je, teoreticky popíše výhody a nevýhody oproti jiným nástrojům se stejnou funkcionalitou, na vhodném příkladu popíše jednotlivé možnosti tohoto nástroje a názorně ukáže výsledky praktického příkladu.

Toto téma jsem si zvolil z několika důvodů - zejména díky mému pozitivnímu vztahu k retro videohrám a zájmu o vývoj her. SFML knihovna mě zaujala především možností herního vývoje v jazyce C++, v němž vznikly legendární tituly jako Age of Empires, Witcher nebo Counter Strike.

Teoretická část práce bude popisovat historii vývoje her, SFML a moduly spadající pod tuto knihovnu, porovná ji s ostatními nástroji a popíše fungování základní prvků herního vývoje.

Praktická část se následně zaměří na samotné implementování daných modulů a jejich metod při vývoji jednoduché hry. Budu se zde věnovat budování aplikace od jejích počátků až po její dokončení, počínaje vytvořením jednoduchého okna, stavového manažeru, základní herní logiky a následné přidání textur, hudby a zvuků do této aplikace. Poslední kapitola praktické části se bude věnovat hře přes síť a problémům, které při jejím vývoji mohou vývojáře potkat.

Přínosem práce je ukázat, že SFML představuje vhodný nástroj zejména pro výukové a demonstrační účely. Není sice plnohodnotným herním enginem, avšak svou jednoduchostí a přímým přístupem k multimediálním funkcím poskytuje studentům i začínajícím vývojářům cenný vhled do základních principů herního vývoje a programování v jazyce C++.

I TEORETICKÁ ČÁST

1 HERNÍ VÝVOJ

Vývoj počítačových her je specifický obor softwarového inženýrství propojující programování, grafiku, zvuk, herní desing a v neposlední řadě podnikání a marketing. Samotné vytvoření technicky funkční aplikace je pouze prostředkem k výslednému prodkutu, tím je kvalitní interaktivní zážitek, který uživatele vtáhne do svého světa abstraktních pravidel a svobody od strastí reálného života.

Tato kapitola nejprve nastíní historický vývoj herního průmyslu, následně představí základní fáze vývoje her a klíčové programátorské principy, které se objevují napříč technologiemi a platformami.

1.1 Historický vývoj videoher

První videohra pochází z roku 1948 a byla velmi inspirovaná radarovým displejem. První hra kde člověk opravdu interagoval s počítačem je o pár let později. Když pan Douglas vyvinul OXO – piškvorky na počítači douglas. První hru více hráčů na svět přinesl W. Higinbotham a to v roce 1958 hru Tennis for two – neboli tennis pro dva. Hrála se na dvou obrovských počítačích. Velký rozvoj ve vývoji nastal v době domácích konzolí (např. Atari) [22]

Osmdesátá léta přinesla i krizi a to takzvaný videoherní crash způsobený zaplavením trhu nekvalitními tituly (např. nechvalně proslulá hra E.T.). Herní průmysl ale zachránilo Nintendo se značkou Super Mariem Bros. V 90. letech nastala éra 3D grafiky (např. The Legend of Zelda: Ocarina of Time).

S přelomem tisíciletí se hry přesunuly do online světa.

Největší hit představovala hra World of Warcraft (2004), kde dokonce došlo k incidentu, ve kterém debuff z boss fightu neskončil a náhodně infikoval celé virtuální město – vědci pak tuto událost využili k simulacím šíření epidemií. [20]

V dalších letech vystoupaly na vrchol mobilní hry (Angry Birds, Clash of Clans), které svým free-to-play modelem a dostupností získali obrovskou popularitu. Dnes se hry řadí mezi multimiliardové průmysly. Největšími novinkami a pravděpodobně budoucností v tomto oboru je využití VR, cloud gamingu (hra běží na vzdáleném serveru, tudíž není potřeba takřka žádný hardware) a umělé inteligence.

1.2 Fáze vývoje hry

Vývoj videoher, podobně jako normálních programů, lze rozdělit do několika základních fází:

- Pre-produkce – hledání nápadu, tvorba dokumentace (Game Design Document), sestavení týmu a plánování.
- Prototypování – rychlé ověřování herních mechanik, provizorní grafika.
- Produkce – hlavní fáze, kde vzniká kód, grafika, zvuk, herní úrovně a testují se herní mechaniky.
- Testing & QA – odhalování chyb, vyvažování obtížnosti, ladění výkonu.
- Release a post-launch – distribuce hry, opravy chyb, přidávání obsahu formou DLC či aktualizací.

I menší projekty, jako studentské ukázky v knihovnách typu SFML, tyto fáze následují, obvykle v jednodušší podobě a často se spojenými kroky.

Zatímco základní fáze vývoje hry lze aplikovat na jakýkoliv projekt, jejich realizace se znatelně liší na základě velikosti týmu, který hru vyvíjí. Jedná se o **AAA titul** nebo **indie projekt**?

AAA vývoj je charakteristický rozsáhlými týmy, rozděleným podle specializací do různých oddělení (programátoři, grafici, apod.). Díky vysoké investici a expertíze jednotlivých oddělení je možné vytvářet hry s komplexnějšími mechanikami, lepší grafikou a vysokou výnosností. Finanční a časová náročnost je zde ale největší nevýhodou. Vývoj se může odsouvat až do neurčita. [21]

Indie vývoj naopak vezme unikátní nápad a snaží se ho v malém teamu (či dokonce jednotlivci) realizovat. Role v teamu se často mísí (např. programátor je i grafikem). [21]

Oba přístupy se navzájem ovlivňují. AAA studia se inspirují kreativními nápady nezávislých vývojářů a indie tvůrci využívají osvědčených produkčních metod.

1.3 Programovací techniky ve hrách

Bez ohledu na velikost projektu se při vývoji her uplatňují určité osvědčené principy a vzory:

- Herní smyčka (game loop) – základní konstrukce, neustále zpracovává vstupy, aktualizuje logiku hry a vykresluje scénu.
- Stavové automaty – hry se skládají ze stavů (menu, samotná hra, pauza) pro jednodušší navigaci.

- Správa vstupů – kombinace událostního zpracování (event handling) a dotazování v reálném čase (polling).
- Renderování – v případě 2D se jedná o práci s texturami a sprity, u 3D existuje složitější grafická pipeline.
- Správa zdrojů – efektivní práce s pamětí, texturami, zvuky a dalšími assety.
- Kolize a fyzika – od jednoduchých bounding boxů po složitější algoritmy.

Knihovna SFML některé tyto koncepty ilustruje v jednodušší formě, pro studenty či začátečníky ideální způsob, jak se ponořit do světa vývoje her.

1.4 Enginy vs. knihovny

V herním vývoji rozlišujeme dva hlavní přístupy:

- Enginy (Unity, Unreal, Godot) – nabízejí komplexní sadu nástrojů (grafiku, fyziku, scripting, GUI). Umožňují rychlý vývoj, ale skrývají technické detaily.
- Knihovny a frameworky (SFML, SDL, Raylib) – poskytují nižší úroveň práce s grafikou, zvukem či vstupy. Vývojář má více kontroly, ale také více odpovědnosti.

Výhody enginů spočívají v rychlosti vývoje a dostupnosti hotových nástrojů. Naopak knihovny jsou vhodné pro ty, kteří chtějí proniknout do principů fungování her od základu.

2 SFML KNIHOVNA

SFML - Simple and Fast Multimedia Library – tedy jednoduchá a rychlá multimediální knihovna, je rozdělena do pěti základních modulů – o těch více v podkapitolách. Modularita umožňuje využití jen nezbytných modulů. Knihovna je jednoduchý způsob vytvoření okna či plnohodnotný způsob tvorby multimediálních aplikací. Jedná se o open source přenosnou API v jazyce c++, díky čemuž je velmi snadné ji využívat [1]

Byla založena za účelem zjednodušení SDL knihovny pro začínající vývojáře. Staví na OpenGL – podporuje metody knihovny OpenGL, ale zároveň vlastní metody SFML aktivně její využívají kód pro svou potřebu.

Podporuje škálu ruzných programovacích jazyků. Mezi nejznámnější patří Python, Java nebo Rust.

2.1 System

Tento modul komunikuje s operačním systémem. Díky tomu může být SFML multiplatformním jazykem, byť každý operační systém funguje jinak. [16]

Základní stavební kámen celé knihovny. Lze zde naleznout časovač, vlákna, matematické operace a další běžné funkce. Ostatní moduly závisí na systemu a není bez něj možné použít jiné moduly.

```
sf::Clock clock; // starts the clock
...
sf::Time elapsed1 = clock.getElapsedTime();
std::cout << elapsed1.asSeconds() << std::endl;
clock.restart();
...
sf::Time elapsed2 = clock.getElapsedTime();
std::cout << elapsed2.asSeconds() << std::endl;
...
clock.stop(); // stops the clock
std::cout << std::boolalpha << clock.isRunning() << std::endl;
clock.reset(); // resets elapsed time to zero
...
clock.start(); // starts the clock
sf::Time elapsed3 = clock.getElapsedTime();
std::cout << elapsed3.asSeconds() << std::endl;
```

Obrázek 1 Kód modulu systém - práce s časovačem

(zdroj: [2])

2.2 Window

Modul *Window* se zaměřuje na tvorbu oken, hlídá uživatelovy události a vstup ze zařízení zařízení.

```cpp
#include <SFML/Window.hpp>

int main()
{
    sf::Window window(sf::VideoMode({ 800, 600 }), "My window");

    // run the program as long as the window is open
    while (window.isOpen())
    {
        while (const std::optional event = window.pollEvent())
        {
            // "close requested" event: we close the window
            if (event->is<sf::Event::Closed>())
                window.close();
        }
    }
}
```

Obrázek 2 Definice základního okna v SFML

(zdroj: [3])

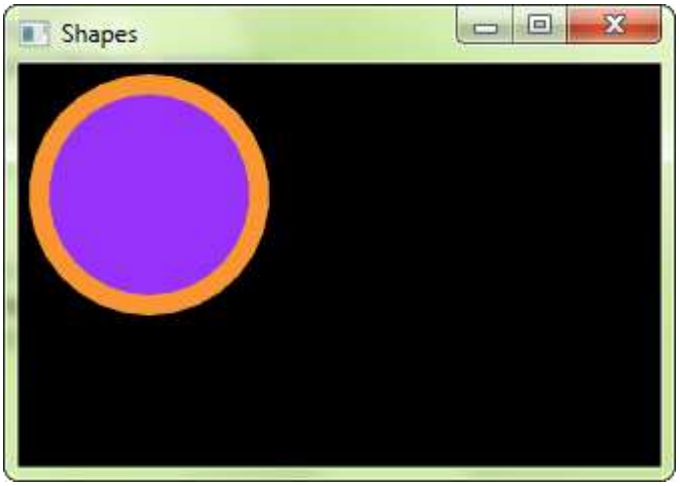
2.3 Graphics

Funkce tohoto modulu je čistě práce s 2D grafikou. Zahrnuje práci s texturami, tvary, textem, barvami a některými grafickými prvky.

```cpp
sf::CircleShape shape(50.f);
shape.setFillColor(sf::Color(150, 50, 250));

// set a 10-pixel wide orange outline
shape.setOutlineThickness(10.f);
shape.setOutlineColor(sf::Color(250, 150, 100));
```

Obrázek 3 Práce s tvary za pomoci grafického modulu

(zdroj: [4])



Obrázek 4 Výsledný obrázek kódu z obrázku 3

(zdroj: [4])

2.4 Audio

Modul *Audio* zajišťuje přehrávání zvukových souborů. Podporuje jak krátké zvukové efekty, tak i hudbu jako takovou, vhodné do pozadí.

```
#include <SFML/Audio.hpp>

int main()
{
    sf::SoundBuffer buffer("sound.wav"); // Throws sf::Exception if an error occurs

    // OR

    sf::SoundBuffer buffer;
    if (!buffer.loadFromFile("sound.wav"))
        return -1;

    sf::Sound sound(buffer);
    sound.play();
}
```

Obrázek 5 Přehrání jednoduchého zvuku sound.wav za pomoci modulu audio

(zdroj: [5])

2.5 Network

Síťový modul komunikuje mezi zařízeními pomocí socketů, podporuje protokoly TCP, UDP, ale i HTTP a FTP. Jeho využití je vhodné víceméně pouze pro síťové aplikace a multiplayerové hry.

```cpp
#include <SFML/Network.hpp>

sf::TcpSocket      socket;
sf::Socket::Status status = socket.connect({192, 168, 0, 5}, 53000);
if (status != sf::Socket::Status::Done)
{
    // error...
}
```

Obrázek 6 Spojení přes TCP pomocí modulu Network

(zdroj: [6])

2.6 Čestné zmínky

SFML jako takový nemá žádný GUI v základu. Patří to mezi jednu z nevýhod této knihovy, naštěstí lze tento problém obejít pomocí komunitní knihovny ***imgui-sfml*** založené ma ***dear imgui***. [2]

```cpp
#include "imgui.h"
#include "imgui-SFML.h"

#include <SFML/Graphics/CircleShape.hpp>
#include <SFML/Graphics/RenderWindow.hpp>
#include <SFML/System/Clock.hpp>
#include <SFML/Window/Event.hpp>

int main() {
    sf::RenderWindow window(sf::VideoMode({640, 480}), "ImGui + SFML = <3");
    window.setFramerateLimit(60);
    ImGui::SFML::Init(window);

    sf::CircleShape shape(100.f);
    shape.setFillColor(sf::Color::Green);

    sf::Clock deltaClock;
    while (window.isOpen()) {
        while (const auto event = window.pollEvent()) {
            ImGui::SFML::ProcessEvent(window, *event);

            if (event->is<sf::Event::Closed>()) {
                window.close();
            }
        }

        ImGui::SFML::Update(window, deltaClock.restart());

        ImGui::ShowDemoWindow();

        ImGui::Begin("Hello, world!");
        ImGui::Button("Look at this pretty button");
        ImGui::End();

        window.clear();
        window.draw(shape);
        ImGui::SFML::Render(window);
        window.display();
    }

    ImGui::SFML::Shutdown();
}
```

Obrázek 7 Ukázka imgui-sfml

(zdroj: [7])

3 SFML VE SVĚTĚ VÝVOJE HER

SFML je v herním vývoji relativně malý projekt i tak nabízí velké množství výhod oproti svým konkurentům a to i přes fakt, že jim v mnohém nestačí. Tato kapitola bude pojednávat o rozdílech SFML proti zbytku průmyslu herního vývoje. Konkrétně probere jednotlivé knihovny a enginy spolu s jejich výhodami, nevýhodami a případnými možnostmi použití.

Ačkoli SFML není tak rozšířené jako Unity či Unreal, komunita kolem něj vytvořila řadu open-source projektů, které slouží jako inspirace. Příkladem je 2D engine *Thor* rozšiřující funkce SFML, hra *Atom Zombie Smasher Clone* nebo různé retro-style platformery dostupné na GitHubu. Tyto projekty ukazují, že knihovna má své pevné místo v oblasti výuky, prototypování i menších herních titulů.

3.1 SFML

SFML, zadefinované v předchozí kapitole má několik výhod. Mezi ně patří například nízká náročnost na hardware a vysoká kontrola nad kódem, pamětí atd. Což je skvělé pro uživatele, kteří plánují se učit jak vyvíjet hry. Na druhou stranu tyto výhody spolu nesou nevýhody jako je chybějící GUI nebo nutnost spoustu věcí inventovat znovu (např. textureManager). Dalšími nevýhodami jsou velikost komunity SFML a chybějící podpora znaků mimo základní ASCII tabulku.

SFML se tedy skvěle hodí pro prototypování, menší osobní projekty či školní úlohy.

3.2 SDL

SDL (Simple DirectMedia Layer) je multiplaformní knihovna jejíž účelem je umožnit nízko úrovňový přístup k hudbě, klávesnici či myši. Podporuje všechny velké platformy. [8]

Velmi se podobá SFML ve spoustě ohledech. Avšak mezi největší rozdíly patří procedurální vývoj místo objektově zaměřeného jako je tomu u SFML

Tato knihovna byla vydána značnou dobu dříve než SFML a zvládla si vybudovat notnou komunitu, díky čemuž vzniklo nemalé množství podpůrných knihoven.

Práce s ní na druhou stranu znamená nutnost znalosti práce s pamětí apod. technické vědomosti. Není tedy vhodná pro začátečníky.

Využití je hlavně v menších projektech, ale SDL si našlo i místo u známých jmen v herním světě.

3.3 Godot

Godot je open-source engine pro vývoj 2D i 3D her. Nabízí vlastní skriptovací jazyk GDScript ale zvládá i C#. [9]

Oproti SFML má vlastní vizuální editor – je možné vytvořit hru bez nutnosti psát kód.

Mezi jeho největší výhodu patří právě fakt, že se jedná o open-source projekt. Komerční využití výsledných her je bez poplatků a tedy značně lákavé pro velké množství milovníku videoher.

Godot je výborný pro začínající vývojáře nebo tvůrce indie her.

3.4 Gamemaker

GameMaker je komerční engine zaměřený na 2D hry. Oproti SFML nevyžaduje žádné technické znalosti. Vlastní vlastní jazyk GML a vizuální editor podobný tomu v Godotu.

GameMaker využívá Direct3D na Windows, OpenGl na Linux a macOS a OpenGL ES na Andorid a IOS. Přestože je zaměřený na 2D hry, tak zvládá i limitované 3D projekty [10]

Vývoj v GameMakeru je velmi snadný, avšak jeho vlastní jazyk má určité limitace. Dalším mínusem je komerční stránka aplikace. Je nutné zakoupit licenci.

Využití je v komerčním vývoji her, několik známých děl bylo vytvořeno právě v tomto prostředí jako např. Undertale.

3.5 Unity

Unity je nejznámější herní engine současnosti. Umožňuje tvorbu 2D i 3D her a podporuje všechny možné platformy. Má neuvěřitelně rozsáhlou komunitu a nemalé množství výukových materiálů.

Tvůrci mohou vytvářet a prodávat 2D i 3D materiály dalším vývojářům na Unity Asset Store. [11]

Unity nepracuje takřka vůbec s nízkoúrovňovým kódem, ba naopak pracuje ve vyšších sférách abstrakce. Na jednu stranu pohodlné řešení, na druhou stranu to brání hlubšímu porozumění kódu.

Unity je skvělou volbou pro větší týmy, komerční projekty nebo multiplatformní vývoj.

Tabulka 1 Výhody a nevýhody SFML

Vlastnosti/Engine	SFML	SDL	GODOT	GAMEMAKER	UNITY
Nenáročnost	+	-	-	+	-
Kontrola	+	+	+	-	-
Výuka / jednoduchost	+	-	+	+	-
Rozšiřitelnost	-	+	+	-	+
Komunita	-	+	+	-	+
GUI / vizuální editor	-	-	+	+	+
Debugger	-	+	-	+	+
Multiplatformnost	+	+	+	-	+
Robustnost	-	+	-	-	+

(zdroj: vlastní)

4 SFML METODY POUŽITÉ V PRAKTICKÉ ČÁSTI

Tato kapitola se věnuje teoretickému základu jednotlivých funkcionalit knihovny SFML, které byly či mohly být použity v praktické části projektu a funkcionality. Cílem je stručně objasnit principy fungování a logiku jednotlivých modulů.

Produkční cyklus hry je rozdělen do několika fází, včetně konceptualizace, předprodukce, produkce, testování a vydání. [18]

Tato kapitola se zaměřuje na fázi produkce, konkrétně na implementaci klíčových technických prvků pomocí knihovny SFML.

4.1 Modul SFML Window

Modul ***sf::Window*** (respektive jeho rozšíření ***sf::RenderWindow***) definuje okno operačního systému, které je schopné přijímat vykreslovací funkce OpenGL. Dále modul poskytuje jednoduchý interface pro manipulaci s oknem, jeho velikostí či minimalizací. Umožňuje jednoduchou kontrolu pomocí metod ***pollEvent*** a ***waitEvent***.

4.1.1 Herní smyčka

Základem víceméně všech "real-time" počítačových her je tzv. herní smyčka. Jedná se o cyklus v programu, který se stará o aktualizaci stavu hry, a vykreslení jednoho tzv. snímku (frame) na obrazovku. Hry typicky fungují tak, že běží donekonečna v tomto cyklu (herní smyčce), a např. 60x za vteřinu aktualizují stav hry a poté jej vykreslí. [12]

Samotná smyčka je součástí vývoje her obecně spíše než SFML knihovny, avšak je nutné ji zmínit čistě kvůli nutnosti její existence. O vytvoření herní smyčky se stará metoda ***window.isOpen()***. Tato metoda ověřuje zda nebylo okno zavřeno uživatelem či jiným programem.

4.1.2 Zpracování událostí

SFML podporuje událostní model zpracování vstupů, podobně jako GUI knihovny. Pomocí metody ***pollEvent*** lze procházet frontu událostí a reagovat např. na zavření okna, stisk klávesy či pohyb myši. Každá událost je reprezentována strukturou ***sf::Event*** s informacemi o typu a souvisejících datech. Např***. sf::Event::Closed*** označuje požadavek na uzavení okna, ***sf::Event::KeyPressed*** nese informaci o stisku konkrétní klávesy

4.1.3 Zpracování přímého vstupu

Kromě událostí podporuje SFML i tzv. aktivní dotazování, tedy kontrolu aktuálního stavu vstupních zařízení. Třídy ***sf::Keyboard*** a ***sf::Mouse*** nabízejí metody typu ***isKeyPressed***, které se hodí pro plynulé ovládání postav, kde je nutné detekovat, zda je klávesa aktuálně držena. To je výhodné například pro ovládání pohybu nebo střelby v reálném čase.

4.1.4 Renderování

Všechny grafické objekty, které lze vykreslit, implementují rozhraní ***sf::Drawable***. Pomocí metody ***window.draw*** lze vykreslit textury, tvary, text nebo sprite. Mezi jednotlivými snímky je nutné volat ***window.clear*** a ***window.display*** – první metoda vymaže starý obsah okna, druhá aktualizuje obrazovku. Interně SFML využívá double-buffering, což zajišťuje plynulé překreslování bez blikání.

4.2 Práce s časem

Třída ***sf::Clock***i umožňuje sledování času od určitého okamžiku. Metoda restart vrací uplynulý čas a současně hodiny resetuje, což je ideální pro zjištění tzv. delta time mezi snímky. SFML využívá vnitřní třídu sf::Time pro reprezentaci časových úseků, čímž umožňuje přesné řízení animací, pohybu a dalších časově závislých prvků.

4.3 Textury a animace

sf::Texture slouží k načtení a uchovávání obrázkových dat (např. png, jpeg ad.). K načtení do paměti slouží obvykle metoda ***loadFromFile***. Vzhledem k její náročnosti je běžnou praxí ukládat načtené data do mapy (například ***std::map<std::string, sf::Texture>***) a jejich pozdější volání právě z této mapy. Velmi běžně programátor tvoří TextureManager pro práci s texturami.

4.4 Zvuk a hudba

Zvukové efekty a hudba jsou obsluhovány pomocí tříd sf::SoundBuffer, sf::Sound a sf::Music. První jmenovaná se používá pro kratší zvuky, druhá pro jejich přehrávání. sf::Music streamuje hudbu z disku a hodí se pro pozadí. SFML podporuje vícekanálové zvukové efekty, regulaci hlasitosti i prostorové efekty.

4.5 Práce s fonty a textem

Pro zobrazování textu nabízí SFML třídy sf::Font a sf::Text. Objekt sf::Font slouží k načtení fontu z externího souboru (např. .ttf formát) a jeho uchování v paměti, přičemž metoda loadFromFile zajišťuje načtení z disku.

Samotný text se reprezentuje pomocí sf::Text, který uchovává řetězec znaků, ukazatel na font a další grafické parametry jako velikost písma, barvu, řádkování nebo ohraničení.

Texty lze libovolně transformovat stejně jako ostatní grafické objekty – tedy posouvat (setPosition), otáčet (setRotation) a škálovat (setScale). SFML interně vykresluje text jako sadu trojúhelníků, což zajišťuje plynulé vykreslování i při změnách velikosti či při použití efektních fontů.

Použití těchto tříd je nezbytné např. při tvorbě herních skóre, stavových hlášení, menu nebo debugovacích informací.

4.6 Síťová komunikace

V době, kdy internet byl ve svých počátcích byla rychlsot spojení velmi pomalá a hry tak využívali ve hrách pouze spojení typu LAN. Postupem času se rychlost spojení zvyšovala a tento typ spojení postupně upadal ve prospěch opravdových síťových spojení. [15]

Modul sf::Network umožňuje práci s TCP a UDP protokolem. Pomocí tříd jako sf::TcpSocket a sf::TcpListener lze navázat spojení mezi dvěma klienty a vyměňovat si data. SFML zajišťuje i jednoduché serializační rozhraní pomocí sf::Packet, což usnadňuje přenos strukturovaných informací. Kromě běžného spojení podporuje knihovna i broadcast přes UDP, což je vhodné pro vyhledávání serverů v lokální síti. Multiplayerová část hry je navržena jako komunikace mezi dvěma hráči v rámci lokální sítě (LAN), přičemž jeden hráč vystupuje jako hostitel (server) a druhý jako klient. Celý systém praktické části je založen na kombinaci protokolů UDP a TCP a je implementován pomocí síťového modulu knihovny SFML.

4.6.1 Fáze navazování spojení

Proces připojení je rozdělen do dvou částí:

- Vyhledání hostitele (UDP broadcast)
- Navázání spojení (TCP)

UDP broadcast řeší problém klienta bez znalosti IP adresy a jeho spojení s hostitelem. Klient tedy odesílá UDP broadcast zprávy do lokální sítě. Hostitel tyto zprávy naslouchá pomocí ***sf::UdpSocket*** a při jejich přijetí odpoví potvrzením své dostupnosti.

Po identifikaci hostitele se klient přepne na přímou TCP komunikaci a pomocí ***sf::TcpSocket*** naváže spolehlivé spojení. Toto spojení slouží pro výměnu herních dat během samotné hry.

4.6.2 Rozdělení úkolů

Síťová architektura následuje princip autority serveru, což znamená, že hostitel spravuje hlavní herní logiku a klient zajišťuje pouze vstupy hráče, které jsou odesílány hostiteli. Tímto způsobem je zajištěna konzistence hry a minimalizace problémů způsobených rozdílnými výpočty na různých zařízeních. Klient se tak stává pasivním zobrazovačem stavu hry řízené hostitelem.

4.6.3 Výměna dat

Komunikace mezi klientem a hostitelem probíhá prostřednictvím objektů sf::Packet, které umožňují serializaci dat (např. pozice pálky, míčku, skóre). Data jsou odesílána a přijímána asynchronně ve vícevláknovém prostředí, přičemž každé vlákno má jasně definovanou odpovědnost.

4.6.4 Latence a synchronizace

Pro eliminaci negativního dopadu latence lze použít několik optimalizačních taktik. Např. odesílání paketů v pravidelném intervalu, zobrazování pouze posledního přijatého stavu či možnost interpolace. V praktické části byly využity pouze první dvě metody, interpolace zkreslovala pozici míčku což byla dostatečná nevýhoda, aby překryla výhody.

4.6.5 Výhody a omezení

Výhodou zvolené architektury je její jednoduchost, robustnost a nízké požadavky na konfiguraci. Použití TCP zajišťuje spolehlivost a korektní pořadí přenášených paketů, zatímco UDP broadcast zjednodušuje navazování spojení. Na druhé straně je řešení omezeno na LAN prostředí.

Interpolace je typ estimace, jeho úkolem je ze známých dat odhadnout budoucí pozici objektu. [14]

5 HERNÍ VÝVOJOVÉ METODY BĚŽNĚ VYUŽÍVANÉ SPOLU S SFML

5.1 Stavový systém (StateHandler nebo také StateManager)

V prvé řadě je nutné definovat co to vlastně ten stav (State) je – jedná se jakousi vrstvu aplikace ve které běží kód. Každá herní obrazovka je samostatný stav. Všechny takové obrazovky mají vlastní metodu ***handleInput***, ***update*** a ***render***. Vývojářovo prací je využít tento systém k rozmělnění daného úkolu či problému do menších, zvládnutelných States a přepínat mezi nimi.

StateManager má i další využití – lze jej použít k přepínání akcí jednoduchých AI (Stůj, Jdi nebo Útoč). [13]

```
enum class StateType{
    Intro = 1, MainMenu, Game, Paused, GameOver, Credits
};
```

Obrázek 8 Jednoduchá ukázka States

(zdroj: PUPIUS, Raimondas, 2015. SFML Game Development By Example. Packt Publishing. ISBN 9781785283000. 1785283006.)

```
void Game::Update(){
  switch(m_state){
    case(StateType::Intro):
      UpdateIntro();
      break;
    case(StateType::Game):
      UpdateGame();
      break;
    case(StateType::MainMenu):
      UpdateMenu();
      break;
    ...
  }
}
```

Obrázek 9 Jednoduchá ukázka States – gameUpdate

(zdroj: PUPIUS, Raimondas, 2015. SFML Game Development By Example. Packt Publishing. ISBN 9781785283000. 1785283006.)

5.1.1 Menu

MenuState je první stav, který se po spuštění aplikace načítá. Slouží jako hlavní rozcestník – zobrazuje základní uživatelské rozhraní s tlačítky jako „Hrát“, „Nastavení“ nebo „Ukončit hru“. Po kliknutí na příslušné tlačítko se StateHandler postará o přepnutí do dalšího stavu.

5.1.2 Herní logika

GameState reprezentuje jádro celé hry. Inicializuje se zde velké množství objektů, často mapa a v jednodušších projektech i multimediální soubory jako např. textury. Metoda update() bývá velmi robustní a složitá. V tomto stavu řídí průběh hry, výpočty herní fyziky, detekci kolizí a správu skóre. handleInput() zpracovává vstupy hráče (například pohyb pálky), zatímco render() vykresluje všechny objekty na obrazovku.

5.1.3 Pause State

Pauza se nejjednodušejí řeší právě přes samotný stav. Po aktivaci pauzy se GameState neaktualizuje, ale zůstává vykreslený na pozadí.. Na popředí se zobrazí PauseMenu, které nabízí možnosti jako pokračovat, uložit, nebo návrat do hlavního menu. Toho je dosaženo tak, že PauseState má vlastní render(), ale nevolá update() z GameState.

5.2 SoundManager

SoundManager má za úkol centralizovaně spravovat zvukové efekty potřebné k následnému přehrání všech zvukových efektů a veškeré hudby v aplikaci. Metody ***loadSound*** a ***playSound*** se využívají napříč aplikací pro přehrání specifických zvuků dle aktuální situace. Centralizovaná správa snižuje duplicitu kódu a udržuje tak kód přehledný.

Duplicitní kód zvyšuje šanci výskytu chyb a vynucuje provádět změny na vícero místech aplikace. [19]

5.3 TextureManager

TextureManager má obdobný účel jako SoundManager – jen místo zvuků se soustředí na textury a obrázky použité na pozadí různých States. Tímto způsobem se zabraňuje opakovanému načítání textur z disku a zlepšuje se výkon aplikace.

5.4 AnimationManager

AnimationManager rozšiřuje práci s texturami o možnost animace. Načtená textura je rozdělena na jednotlivé snímky animace (sprite sheet), které jsou přehrávány podle času (deltaTime) pomocí metody ***update***. Tímto způsobem lze vytvářet např. blikající efekty, pohybující se objekty nebo interaktivní tlačítka.

II PRAKTICKÁ ČÁST

6 UKÁZKA JEDNOTLIVÝCH MODULŮ

6.1 CMakeLists

Prvním krokem při tvorbě jakékoliv c++ aplikace je tvorba CMakeLists.txt. V mé aplikaci pro ukázku je tento list velmi prostý.

```
cmake_minimum_required(VERSION 3.20)
project(BP_SFML_Vaclav_Fencl LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(SFML_DIR "${CMAKE_SOURCE_DIR}/SFML-3.0.0/lib/cmake/SFML")
list(PREPEND CMAKE_PREFIX_PATH "${CMAKE_SOURCE_DIR}/SFML-3.0.0")

find_package(SFML 3.0.0 REQUIRED COMPONENTS Graphics Window System
 Audio)

set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/bin")

if(EXISTS "${CMAKE_SOURCE_DIR}/assets")
  add_custom_target(copy_assets
    COMMAND ${CMAKE_COMMAND} -E copy_directory
            "${CMAKE_SOURCE_DIR}/assets"
            "${CMAKE_BINARY_DIR}/bin/assets"
    BYPRODUCTS "${CMAKE_BINARY_DIR}/bin/assets"
    COMMENT "Copying assets -> build/bin/assets")
endif()

set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/bin")

add_subdirectory(APP1)
add_subdirectory(APP2)
add_subdirectory(APP3)
add_subdirectory(APP4)
add_subdirectory(APP5)
add_subdirectory(APP6)
add_subdirectory(APP7)
add_subdirectory(APP8)
add_subdirectory(APP9)
add_subdirectory(APP10)
```

6.2 Tvorba základního okna

Prvním krokem k úspěšnému programu či hře je vždy vytvoření základního okna se specifickou velikostí a názvem. Aby toto okno zůstalo dynamické potřebuje smyčku, která kontroluje, že uživatel okno nezavřel stisknutím křížku.

```
int main() {
       sf::RenderWindow window(sf::VideoMode({800u, 600u}), "APP1
 Treninkove okno");
    window.setFramerateLimit(60);
    while (window.isOpen()) {
        while (auto event = window.pollEvent()) {
            if (event->is<sf::Event::Closed>()) {
                window.close();
            }
        }
        window.clear(sf::Color::White);
        window.display();
    }
    return 0;
}
```

V kódu je krásně vidět tvorba okna pomocí objektu *sf::RenderWindow window(parametry)*, a následně jmenovaná smyčka *while (window.isOpen()){kód}*.

6.3 Časovač a snímky za vteřinu

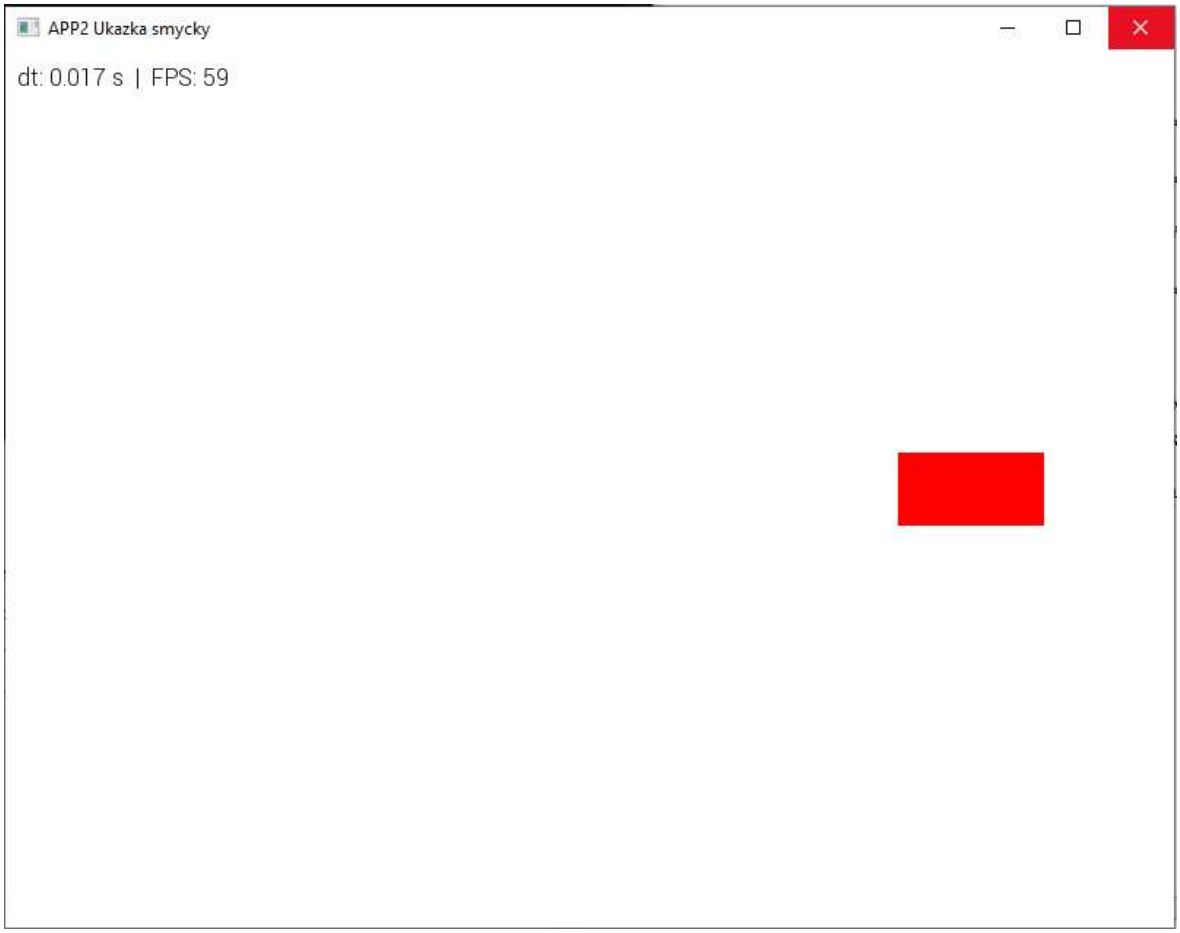
Časovač je nezbytný pro sjednocení doby mezi tzv. ticky programu. Tick je jeden průběh while cyklem. V moderní době jsou počítače takřka až moc rychlé pro hraní her a za vteřinu těch Ticků vznikne množství sahající k nižším tisícům.

Z tohoto důvodu první vývojáři vytvořili tzv. delta time, který stabilizuje množství akcí za vteřinu na rozumnou hodnotu. Toto množství akcí obsahuje i vykreslení všech objektů proto je nazýváme FPS (frames per second) neboli snímky za vteřinu. Díky těmto limitům je možné naprogramovat objekty které mají předvídatelné chování na obrazovce.

SFML umožňuje nastavit snímky za vteřinu na vývojářem předem definovanou hodnotu a z té pomocí časovače odvodit delta time.

```
window.setFramerateLimit(60);
sf::RectangleShape rect({100.f, 50.f});
    rect.setFillColor(sf::Color::Red);
    rect.setPosition(sf::Vector2f{
        window.getSize().x / 2.f - rect.getSize().x / 2.f,
        window.getSize().y / 2.f - rect.getSize().y / 2.f
    });
while (window.isOpen()) {
 while (auto event = window.pollEvent()) {
  if (event->is<sf::Event::Closed>()) window.close();
 }
 sf::Time dt = clock.restart();
 float fps = (dt.asSeconds() > 0.f) ? 1.f / dt.asSeconds() : 0.f;
 rect.move(sf::Vector2f{speed * dt.asSeconds(), 0.f});
}
 window.clear(sf::Color::White);
 window.draw(hud);
 window.draw(rect);
 window.display();
```

V kódu je krásně vidět funkce setFramerateLimit(60), jež nastavuje snímky za vteřinu na plynulých šedesát. Tím limitujeme kolikrát proběhne while cyklus a tím následně upravujeme hodnotu delta time. Rychlost prolétajícího obdélníku je ovlivněna právě touto hodnotou. Ve velkých herních titulech je delta time nezbytný k udržení jednotné rychlosti všech objektů a animací.



Obrázek 10 Výpis delta time a fps. Na obrazovce se posouvá obdélník.

(zdroj: vlastní)

6.4 Vstupy a události

Kód názorně předvádí rozdíl mezi vstupem a událostí.

Vstup je v živém čase a probíhá obvykle víckrát – držení tlačítka k pohybu čtvercem.

Událost se zpravidla děje jen jednou za čas. Zavření okna se například stane pouze jednou za život programu.

```
while (auto event = window.pollEvent()) {
    if (event->is<sf::Event::Closed>()) {
      window.close();
    }
    else if (const auto* mb = event ->
  getIf<sf::Event::MouseButtonPressed>()) {
   if (mb->button == sf::Mouse::Button::Left) {
     square.setFillColor(sf::Color::Blue);
   } else if (mb->button == sf::Mouse::Button::Right) {
```

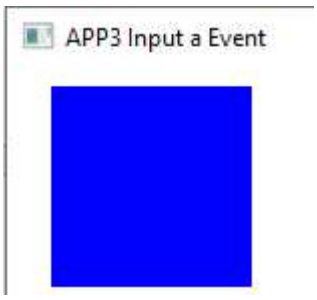
```
        square.setFillColor(sf::Color::Red);
      }
    }
  }
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Left))  delta.x -=
 speed * dt;
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Right)) delta.x +=
 speed * dt;
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Up))    delta.y -=
 speed * dt;
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Down))  delta.y +=
speed * dt;
```

Zásobník událostí hlídá zavření okna a stisknutí myši, pokud se nějaká z těchto akcí splní zařadí se do fronty a počká na řadu. Poté proběhne kód této události.

Kdežto živý vstup je hlídán několika if podmínkami, které okamžitě konají svůj kód a to i několikrát za sebou.

Obrázek 11 Červený čtverec po stisku pravého tlačítka myši

(zdroj: vlastní)



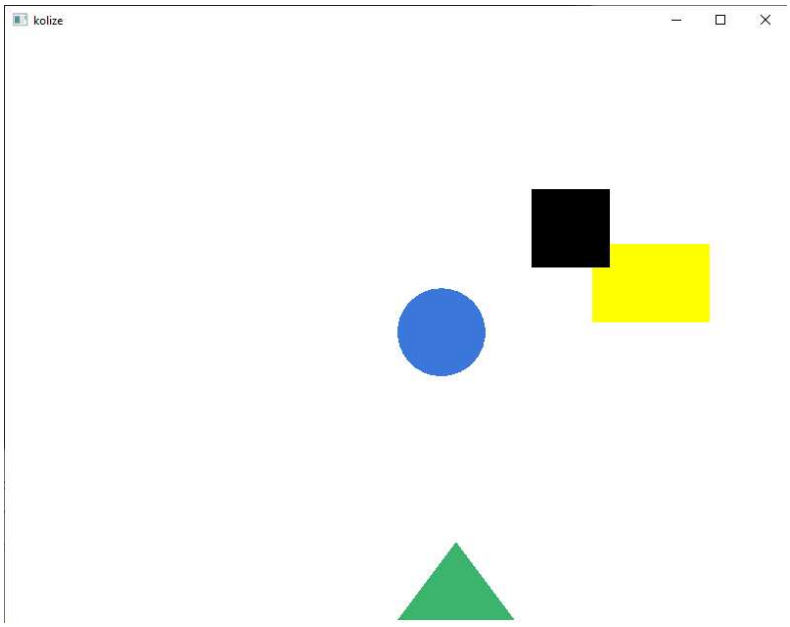
Obrázek 12 Modrý čtverec po stisku pravého tlačítka myši

(zdroj: vlastní)

6.5 Kolize

SFML řeší základní kolizi přes AABB (Axis-Aligned Bounding Box). Pro každý objekt vezmeme jeho globální opsaný obdélník getGlobalBounds() (zarovnaný na osy). Kolize nastane, když se AABB dvou objektů překrývají na obou osách. V SFML lze použít findIntersection(otherBounds), které vrátí obdélník průniku jako std::optional. Pokud průnik existuje, máme kolizi. Proto v kódu testuji *iBox/iCircle/iTri.has_value()* a při kolizi zvýrazním druhý objekt žlutě.

```
const auto p = player.getGlobalBounds();
const auto iBox    = p.findIntersection(box.getGlobalBounds());
const auto iCircle = p.findIntersection(circle.getGlobalBounds());
const auto iTri    = p.findIntersection(tri.getGlobalBounds());
const bool hitBox    = iBox.has_value();
const bool hitCircle = iCircle.has_value();
const bool hitTri    = iTri.has_value();
auto colorize = [](sf::Shape& s, bool hit, sf::Color base){
   s.setFillColor(hit ? sf::Color::Yellow : base);
};
colorize(box,    hitBox,    sf::Color(220, 60, 60));
colorize(circle, hitCircle, sf::Color(60, 120, 220));
colorize(tri,    hitTri,    sf::Color(60, 180, 110));
```



Obrázek 13 Kolize s obdélníkem vyúustila ve změnu jeho barvy

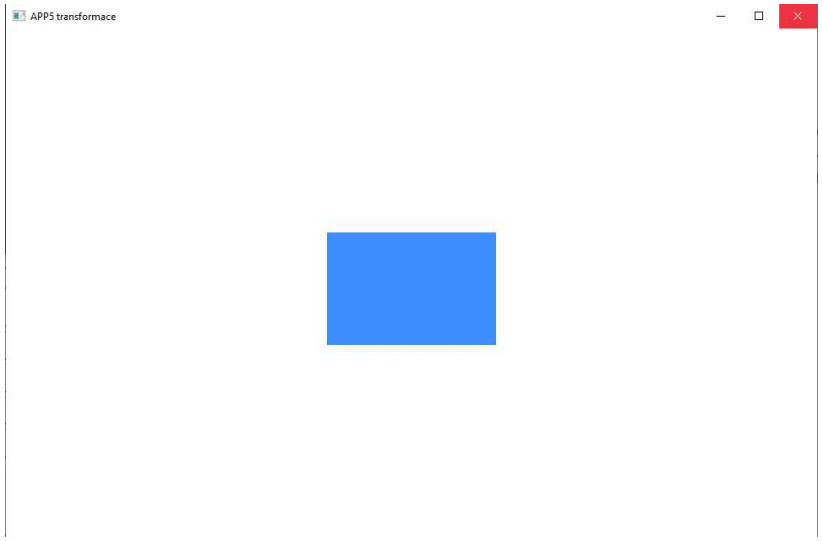
(zdroj: vlastní)

6.6 Transformace

Ukázkový kód demonstruje 4 základní druhy transformací. Translace (pohyb), rotace, škálování a zvětšování určitým směrem. Všechny tyto operace probíhají tak, že se nejprve vytvoří pomocná matice, a ta se následně vynásobí k celkové transformační matici **M**. Obdélník je nakonec vykreslen po kompletní transformaci. Jedná se o názornou ukázku skládání transformačních matic, pomocí kterých se realizuje takřka libovolná změna polohy či tvaru objektu.

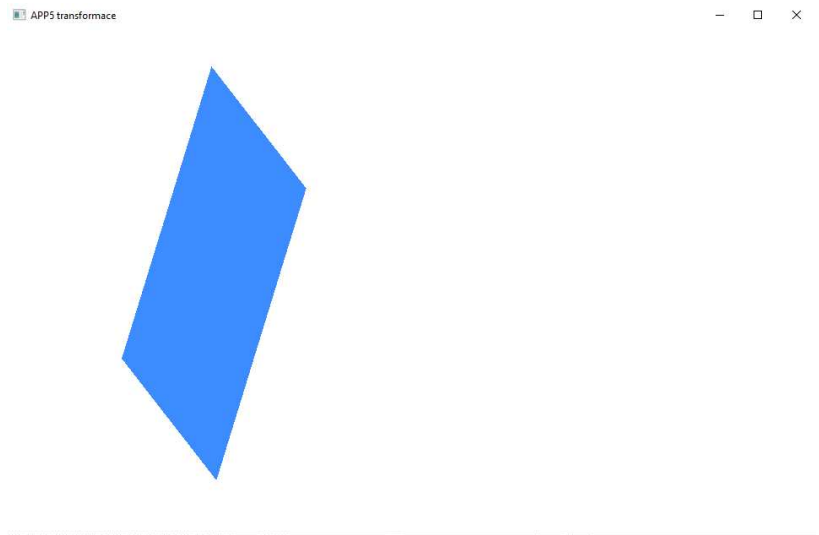
```
sf::Transform M;
auto reset = [&](){
   M = sf::Transform{};
   sf::Transform T; T.translate({480.f,300.f});
   M = T * M;
};
reset();
const sf::Vector2f half = rect.getSize()*0.5f;
const sf::Vector2f O  = M.transformPoint({0.f, 0.f});
const sf::Vector2f X1 = M.transformPoint({1.f, 0.f}) - O;
const sf::Vector2f Y1 = M.transformPoint({0.f,-1.f}) - O;
sf::Vector2f move{};
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Left))  move -= X1 *
 (moveSpeed*dt);
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Right)) move += X1 *
 (moveSpeed*dt);
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Up))    move += Y1 *
 (moveSpeed*dt);
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Down))  move -= Y1 *
 (moveSpeed*dt);
if (move.x!=0.f || move.y!=0.f) {
  sf::Transform T; T.translate(move);
  M = T * M;
}
if    (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Q))    drot    -=
 rotSpeed*dt;
if    (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::E))    drot    +=
 rotSpeed*dt;
if (drot != 0.f) {
    sf::Transform   R;   R.translate(O);   R.rotate(sf::degrees(drot));
    R.translate(-O);
    M = R * M;
}
```

```
bool plus  = sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Equal)  ||
sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Add);

bool minus = sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Hyphen) ||
 sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Subtract);

if (plus || minus) {

 float f = plus ? (1.f + uniRate*dt) : std::max(1.f - uniRate*dt,
 0.01f);

 sf::Transform S; S.translate(O); S.scale({f,f}); S.translate(-O);

 M = S * M;}
```



Obrázek 14 Modrý obdélník před transformací

(zdroj: vlastní)



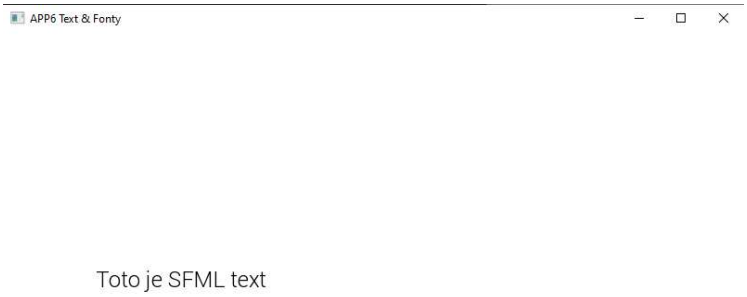
Obrázek 15 Modrý obdélník po pootočení a zvětšení s následným posunem doleva.

(zdroj: vlastní)

6.7 Práce s textem

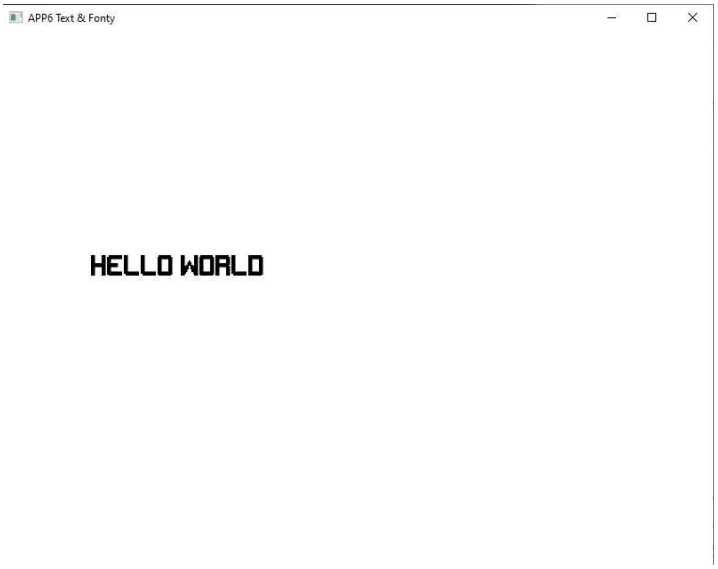
V následující ukázce bylo vytvořeno pomocí načtení sf::Font ze složky *assets/fonts/*. Objekt typu sf::Text následně načte tento font. Pomocí kláves lze přepínat mezi různými přednastavenými fonty (*text.setFont()*) a texty (*text.setString()*). Lze také upravovat velikost textu (*text.setCharacterSize(velikost)*).

```
 case Key::Num1:
  currentFont = 0;
  text.setFont(fonts[0]);
  break;
case Key::Num2:
 currentFont = 1;
 text.setFont(fonts[1]);
 break;
case Key::Num3:
 currentFont = 2;
 text.setFont(fonts[2]);
 break;
case Key::Q:
 text.setString(U"UTB se nachazi v Zline");
 break;
case Key::W:
 text.setString(U"Toto je SFML text");
 break;
case Key::E:
 text.setString(U"Hello world");
 break;
case Key::Add:
case Key::Equal:
 text.setCharacterSize(text.getCharacterSize() + 2);
 break;
case Key::Subtract:
case Key::Hyphen:
 if (text.getCharacterSize() > 2)
  text.setCharacterSize(text.getCharacterSize() - 2);
 break;
```



Obrázek 16 Změněný text.

(zdroj: vlastní)



Obrázek 17 Změněný text a font.

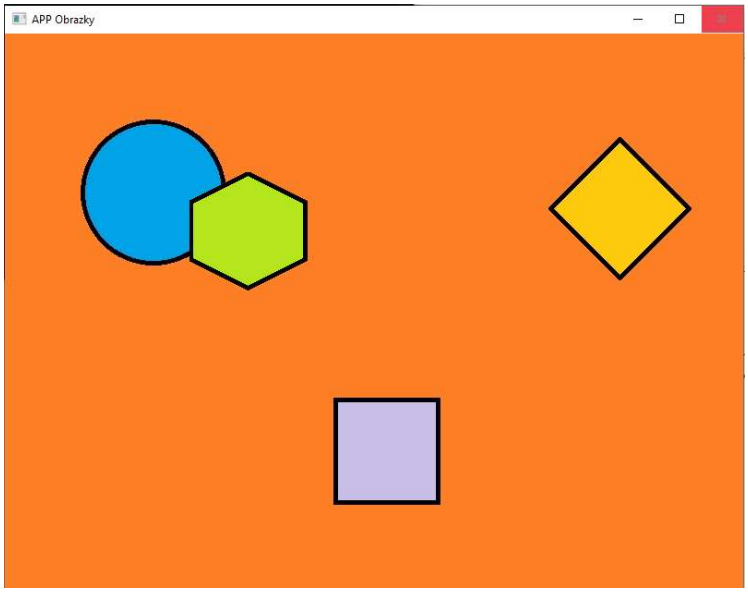
(zdroj: vlastní)

6.8 Práce s obrázky

Podobně jako text je SFML schopný načíst i obrázky různých typů a dokonce zachová jejich průhlednost. Následně se s obrázku dá upravovat nejenom průhlednost.

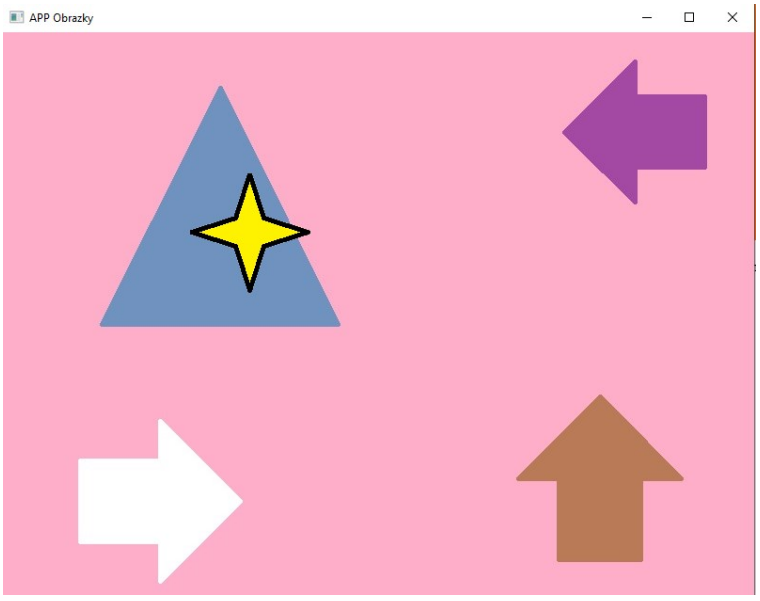
V ukázce kódu je znázorněno načítání obrázků, nastavování textury objektu a úprava průhlednosti.

```
 bgTextures[0].loadFromFile("../assets/textures/bg1.jpg");
 bgTextures[1].loadFromFile("../assets/textures/bg2.jpg");
 fgTextures[0].loadFromFile("../assets/textures/fg1.png");
 fgTextures[1].loadFromFile("../assets/textures/fg2.png");
switch (key->code) {
 case Key::Num1: background.setTexture(bgTextures[0]); break;
 case Key::Num2: background.setTexture(bgTextures[1]); break;
 case Key::Num3: background.setTexture(bgTextures[2]); break;

 case Key::Q: foreground.setTexture(fgTextures[0]); break;
 case Key::W: foreground.setTexture(fgTextures[1]); break;
 case Key::E: foreground.setTexture(fgTextures[2]); break;

 case Key::Add:
 case Key::Equal:
     if (alpha <= 240) alpha += 15; else alpha = 255;
     foreground.setColor({255, 255, 255, alpha});
     break;
case Key::Subtract:
case Key::Hyphen:
     if (alpha >= 15) alpha -= 15; else alpha = 0;
     foreground.setColor({255, 255, 255, alpha});
     break;
```



Obrázek 18 Obrázek č. 1 na pozadí č. 2.

(zdroj: vlastní)



Obrázek 19 Obrázek č. 3 na pozadí č. 3.

(zdroj: vlastní)

6.9 Animace

Animace fungují na velmi podobném principu jako obrázky. Rozdíl je v tom že obrázek se načte a zobrazí celý, kdežto pro animaci je potřeba velké množství obrázků uložené jako mozaika v jednom obrázku. Tímto postupem se šetří paměť, protože jinak by bylo potřeba načíst více obrázků a přepínat mezi nimi. Programátor následně musí zadefinovat jak velký obdélník z tohoto obrázku vyjme, jakožto takzvaný slide, a jak rychle animace projde skrz celý obrázek. V ukázce lze animaci pozastavit a následně ji procházet snímek za snímkem.

```
 const int frameWidth = 128;
    const int frameHeight = 128;
    const int frameCount = 6;
    float frameTime = 0.15f;
if (playing) {
   elapsed += dt;
   if (elapsed >= frameTime) {
       elapsed = 0.f;
       currentFrame = (currentFrame + 1) % frameCount;
       sprite.setTextureRect(frames[currentFrame]);
     }
   }
switch (key->code) {
  case Key::Space:
       playing = !playing;
     break;
  case Key::Period:
       playing = false;
       currentFrame = (currentFrame + 1) % frameCount;
      sprite.setTextureRect(frames[currentFrame]);
     break;
 case Key::Comma:
     playing = false;
     currentFrame = (currentFrame - 1 + frameCount) % frameCount;
     sprite.setTextureRect(frames[currentFrame]);
     break;
 default:
     break;
 }
```

6.10 Práce se zvukem

Zvuk v SFML nabírá dvou různých podob. Zvukové efekty a podkresová hudba. Zvukové efekty používají Sound Bufferu, který načte zvuk a přehraje ho na zavolání. Podkresová hudba naopak čte soubor přímo z uložiště, čímž se šetří paměť. V ukázkovém kódu lze naleznout obě podoby. Zvukový efekt je volán pouze při kliknutí myší, naopak podkresová hudba hraje do pozasavení či vypnutí již od spuštění aplikace.

```
sf::SoundBuffer clickBuf;
    clickBuf.loadFromFile("../assets/sounds/pong.wav");
    sf::Sound click(clickBuf);
    sf::Music music;
 bool                              musicOk                                  =
 music.openFromFile("../assets/sounds/background_music.flac");
float volume = 60.f;
    music.setVolume(volume);
    music.play();
    click.setVolume(volume);
    bool isMusicPlaying = true;
    bool isMusicPaused = false;
switch (kp->code) {
  case Key::Space:
      if (musicOk) {
         auto st = music.getStatus();
             if (isMusicPlaying) music.stop();
             else music.play();
       }
   break;
   case Key::P:
       if (musicOk) {
           auto st = music.getStatus();
           if (isMusicPaused) music.play();
           else music.pause();
       }
   break;
   case Key::Add:
       volume = std::clamp(volume + 5.f, 0.f, 100.f);
       music.setVolume(volume);
       click.setVolume(volume);
                       break;
```

```
case Key::Hyphen:
case Key::Subtract:
     volume = std::clamp(volume - 5.f, 0.f, 100.f);
     music.setVolume(volume);
     click.setVolume(volume);
break;
```

6.11 Práce s kamerou

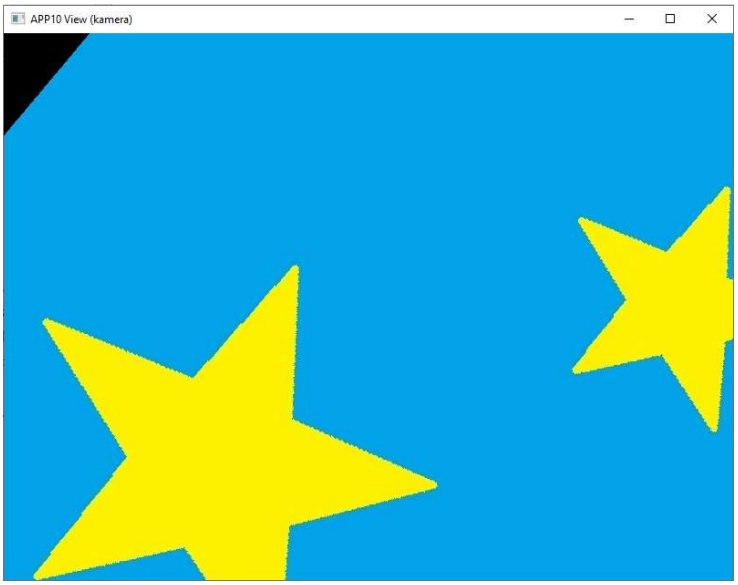
Tato podkapitola probírá práci s kamerou, což je jediný z modulů, které nebyly použity v následující sekci ***Tvorba hry pong***. Kamera vymezuje pouze tu část herního prostoru, která je v daný okamžik viditelná. Díky tomu lze vytvářet rozsáhlé herní mapy, jež může hráč postupně prozkoumávat, aniž by bylo nutné mít celou scénu neustále vykreslenou na obrazovce.

V kódu kamera zabírá celé pozadí, ale uživatel může pohled přiblížit či oddálit, posouvat a otáčet dle libosti. Jedná se jednoduchou demonstraci možností této funkcionality.

```
sf::View view(sf::FloatRect({0.f, 0.f}, {800.f, 600.f}));
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Up))
            view.move({0.f, -moveSpeed});
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Down))
            view.move({0.f,  moveSpeed});
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Left))
            view.move({-moveSpeed, 0.f});
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Right))
            view.move({ moveSpeed, 0.f});

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Q))
            view.rotate(sf::degrees(-1.f));
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::E))
            view.rotate(sf::degrees(1.f));

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Equal))
            view.zoom(0.98f);
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Hyphen))
            view.zoom(1.02f);
        window.setView(view);
```

Obrázek 20 Kamera s lehkým přiblížením.

(zdroj: vlastní)



Obrázek 21 Kamera po pootočení a oddálení.

(zdroj: vlastní)

7 TVORBA HRY PONG

Tato kapitola popisuje vývoj finálního produktu. Využívá jak teorie popsané v kapitole 4 tak její praktického zpracování z kapitoly 6.

7.1 Tvorba okna

V projektu pong je okno o velikosti standartních 800 na 600 s definovaným windowStyle, ten byl definován pro budoucí možnost Fullscreenu. Důležitý je i řádek který nastavuje kurzor myši na viditelný – bez tohoto řádku by kurzor sice byl plně viditelný v malém okně avšak ve Fullscreen režimu by vidět nebyl.

Okno je tvořeno následovně:

```
unsigned int screenWidth = 800;
unsigned int screenHeight = 600;


int main() {
    unsigned int windowStyle = sf::Style::Default;
    sf::RenderWindow window(sf::VideoMode(screenWidth, screenHeight),
 "Pong", windowStyle);
while (window.isOpen()) {
        if (Settings::needsWindowReload) {
            windowStyle = Settings::fullscreen ? sf::Style::Fullscreen
 : sf::Style::Default;
            window.create(sf::VideoMode(screenWidth,   screenHeight),
 "Pong", windowStyle);
            window.setMouseCursorVisible(true);
            Settings::needsWindowReload = false;
        }

        float dt = clock.restart().asSeconds();

        if (auto currentState = stateHandler.getCurrentState()) {
            currentState->handleInput(window);
            currentState->update(dt);

            if (!dynamic_cast<GameState*>(currentState.get()) &&
                !dynamic_cast<PauseState*>(currentState.get())) {
                window.clear();
            }
```

```
            currentState->render(window);
            window.display();
        }
    }
```

Jako styl okna je použit ***sf::Style::Default***, který vytváří titulkový pruh, umožňuje uživateli změnit velikost okna či jej uzavřít. Tento styl byl zvolen jako výchozí, protože poskytuje běžné ovládací prvky a přirozenou interakci s oknem aplikace.

Další část kódu využívá třídu ***StateHandler***, která slouží ke správě herních stavů. Jelikož se nejedná o součást knihovny SFML, nebude v této kapitole blíže rozebírána. Podobně je vynechán i ***SoundManager*** a třída ***sf::Clock***, jejichž detailnější popis bude uveden v samostatných částech.

Prvním krokem herní smyčky je kontrola zdali okno nebylo upraveno. Pokud ano, tak se znovu vytvoří s novými parametry. Čímž lze dynamicky upravovat velikost tohoto okna.

Dále se v rámci herního cyklu vykonává pět základních metod.

- handleInput – řeší vstupy uživatele
- clear – maže obsah obrazovky
- update – aktualizuje logiku v pozadí
- render – vykresluje výsledek metody update
- display – zobrazuje výsledky metody render

Určité herní stavy (např. GameState nebo PauseState) provádějí volání clear() ve vlastní metodě render() – typicky kvůli specifickému způsobu vykreslování, jako je průhlednost nebo zachování pozadí.

7.2 Zpracování událostí (Event Handling) a přímého vstupu (Input)

V různých částech programu lze nalézt různé příklady kontroly a zpracování eventů. Například MenuState kontroluje stisknutí tlačítka myši (viz Obrázek 22 Menu State v aplikaciObrázek 22)Dále lze kontrolovat vstup z klávesnice. V implementaci je například detekováno v ***GameState*** stisknutí klávesy Escape, které slouží k vyvolání pauzy během hry (viz Obrázek 26). Pokud je tato klávesa detekována v rámci události sf::Event::KeyPressed, dojde k přechodu do PauseState pomocí správce stavů:

```
void MenuState::handleInput(sf::RenderWindow& window) {
    sf::Event event;
    while (window.pollEvent(event)) {
        if (event.type == sf::Event::Closed) {
            window.close(); }
        if (event.type == sf::Event::MouseButtonPressed) {
            if (event.mouseButton.button == sf::Mouse::Left) {
                for (auto& button : buttons) {
                    if (button->isMouseOver(window)) {
                        button->playClickSound();
                        button->onClick();
         }}}}}}
```

```
void GameState::handleInput(sf::RenderWindow& window) {
    sf::Event event;
    while (window.pollEvent(event)) {
        if (event.type == sf::Event::Closed) {
            window.close(); }
        if (event.type == sf::Event::KeyPressed && event.key.code ==
 sf::Keyboard::Escape) {
stateHandler.pushState(std::make_shared<PauseState>(stateHandler,
 window)); }}}
```

V implementaci je přímý vstup vidět například v entitě ***Paddle***, kde ovládá pohyb samotné pálky.

```
void Paddle::updatePlayer(float dt) {
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::W)) moveUp(dt);
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::S)) moveDown(dt);
}
```

7.3 Stavový systém

Ke zpracování architektury aplikace byl využit StateHandler. Byl zvolen z důvodu jeho jednoduché implementace, přehlednosti, snadné rozšiřitelnosti a elegantnímu přepínání mezi režimy bez narušení herního toku. Taková architektura zajišťuje vysokou modularitu a oddělenou funkcionalitu. Každý stav používá rozhraní ***IGameState***, které definuje základní metody.

```
class IGameState {
public:
    virtual ~IGameState() = default;
    virtual void handleInput(sf::RenderWindow& window) = 0;
    virtual void update(float dt) = 0;
    virtual void render(sf::RenderWindow& window) = 0;
};
```

Třída StateHandler pak následně řídí přechody mezi různými stavy pomocí zásobníku. To umožňuje přerušení hry a zachování herního stavu vykresleného na pozadí. Po návratu je možné pokračovat ve hře přesně tam, kde skončila.

7.3.1 MenuState

Stav MenuState reprezentuje úvodní nabídku hry, která se načte po spuštění aplikace. Obsahuje pozadí a sadu tlačítek. Tato tlačítka jsou implementována jako objekty odvozené od obecné třídy Button a reagují na interakci myší. Každé tlačítko má metodu ***onClick***, která definuje, jaký stav se má při jeho aktivaci načíst. Tlačítka také načítají font, pro následné vypsání popisku, a zvuk, pro přehrání zvukového efektu kliknutí. Při vývoji Pongu bylo třeba zadefinovat třídu takového tlačítka:

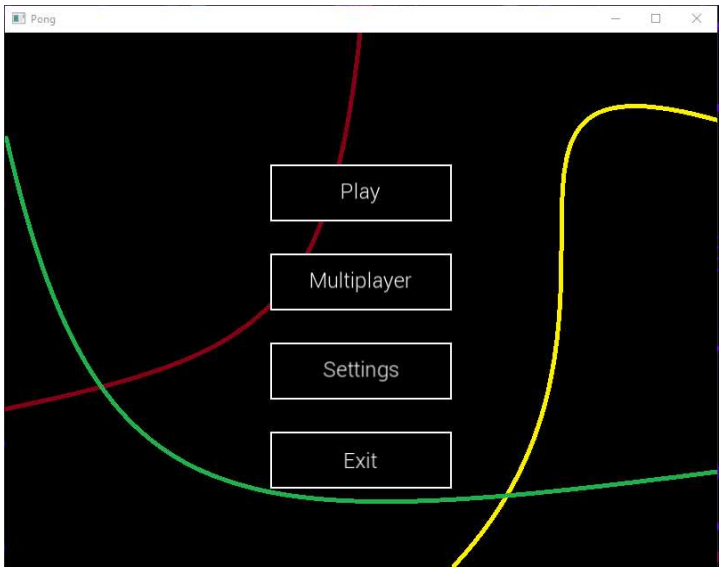
```
class Button {
public:
    Button(const sf::Vector2f& size, const sf::Vector2f& position,
 const std::string& text, sf::Font& font);
    virtual ~Button() = default;

    virtual void update(const sf::RenderWindow& window);
    virtual void render(sf::RenderWindow& window);
    virtual void onClick() = 0;

    bool isMouseOver(const sf::RenderWindow& window) const;
    static void playClickSound();
```

```
protected:
    static bool clickSoundLoaded;

    sf::RectangleShape body;
    sf::Text label;
};
```

A následně je registrovat v konstruktoru hlavního menu:

```
buttons.push_back(std::make_unique<PlayButton>(sf::Vector2f(200,
60), sf::Vector2f(300, 150), font, stateHandler, window));

buttons.push_back(std::make_unique<MultiplayerButton>(sf::Vector2f(2
00, 60), sf::Vector2f(300, 250), font, stateHandler, window));

buttons.push_back(std::make_unique<SettingsButton>(sf::Vector2f(200,
60), sf::Vector2f(300, 350), font, stateHandler, window));
buttons.push_back(std::make_unique<ExitButton>(sf::Vector2f(200,
60), sf::Vector2f(300, 450), font, window));
```

Metoda ***handleInput*** zpracovává kliknutí myší a rozhoduje, které tlačítko bylo aktivováno. ***update*** umožňuje např. detekci najetí kurzoru na tlačítko a ***render*** zajišťuje vykreslení pozadí a všech komponent menu.



Obrázek 22 Menu State v aplikaci

(zdroj: vlastní)

7.3.2 GameState

Tento stav reprezentuje jádro celé aplikace – aktivní herní obrazovku. V tomto stavu je vytvořena instance třídy Game. Metoda ***update*** přenáší řízení na tuto instanci a metoda ***render*** následně zobrazí aktuální stav hry na obrazovku.

```
GameState::GameState(StateHandler& handler, sf::RenderWindow& window)
    : stateHandler(handler), game(window) {}
void GameState::handleInput(sf::RenderWindow& window) {
    sf::Event event;
    while (window.pollEvent(event)) {
        if (event.type == sf::Event::Closed) {
            window.close(); }
        if (event.type == sf::Event::KeyPressed && event.key.code ==
 sf::Keyboard::Escape) {

 stateHandler.pushState(std::make_shared<PauseState>(stateHandler,
 window)); }}}
void GameState::update(float dt) {
    game.update(dt);
}
void GameState::render(sf::RenderWindow& window) {
    game.render();
}
```

7.3.3 Game.cpp

Tato třída má na starost celou logiku hry. Od inicializace všech proměných, nastavování objektů a načítání multimediálních souborů, přes výpočet herní logiky až po vykreslení hry do zpět do stavu GameState.

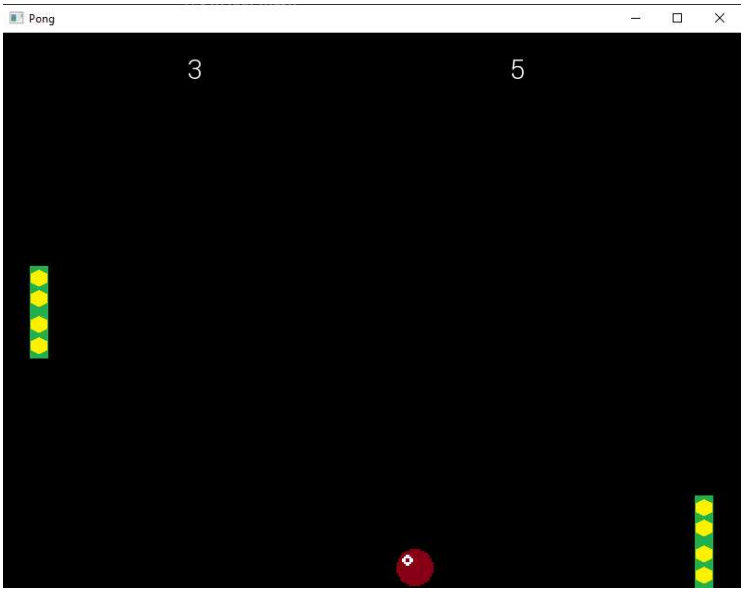
Během inicializace (názorná ukázka v kódu níže) dochází k načtení fontu, textur a zvukových souborů, nastavení obou pálek, míčku a textu vypisující skóre.

```
Game::Game(sf::RenderWindow& window)
    : window(window), ball(), paddle(30.f, 50.f, screenHeight),
 paddleAi(750.f, 50.f, screenHeight)
{
if (!font.loadFromFile("Thirdparty/fonts/Roboto-Light.ttf")) {}
  auto& paddleTexture =
 TextureManager::getTexture("Thirdparty/textures/paddle.png");
    auto& ballTexture =
 TextureManager::getTexture("Thirdparty/textures/ball.png");


    SoundManager::loadSound("pong", "Thirdparty/sounds/pong.wav");
    SoundManager::loadSound("score", "Thirdparty/sounds/score.wav");


    paddle.setTexture(paddleTexture);
    paddleAi.setTexture(paddleTexture);
    ball.setTexture(ballTexture);
```

```
        scoreText1.setFont(font);
        scoreText1.setCharacterSize(30);
        scoreText1.setFillColor(sf::Color::White);
        scoreText1.setPosition(200.f, 20.f);

        scoreText2.setFont(font);
        scoreText2.setCharacterSize(30);
        scoreText2.setFillColor(sf::Color::White);
        scoreText2.setPosition(550.f, 20.f);
    }
```

V metodě ***update*** se řeší logika samotné hry. Na začátku dochází k aktualizaci pozice pálek a míčku, následně probíhá kontrola zdali míček vrazí do vrchní či spodní neviditelné stěny.

```
if (ball.getPosition().y < 0.f || ball.getPosition().y + 40.f >
 screenHeight) {
  ball.bounceY();
  SoundManager::playSound("pong");
}
```



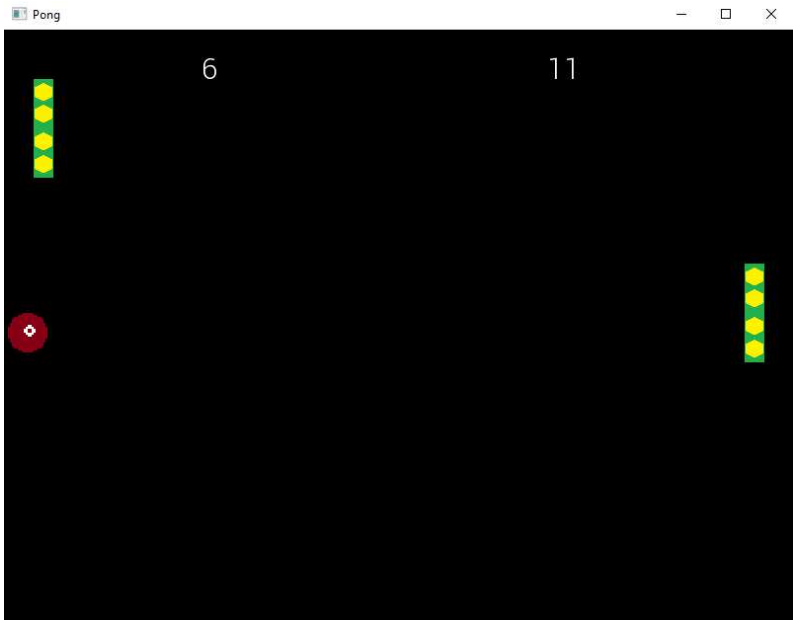
Obrázek 23 Kolize míčku s horizontálním okrajem obrazovky

(zdroj: vlastní)

Dalším krokem je kontrola kolize míčku s vertikálním okrajem nebo-li skórování.

```
if (ball.getPosition().x + ball.radius * 2 >= screenWidth) {
        score1 += 1;
        SoundManager::playSound("score");
        ball.reset();}
```

```
    else if (ball.getPosition().x <= 0.f) {
        score2 += 1;
        SoundManager::playSound("score");
        ball.reset();
    }
```

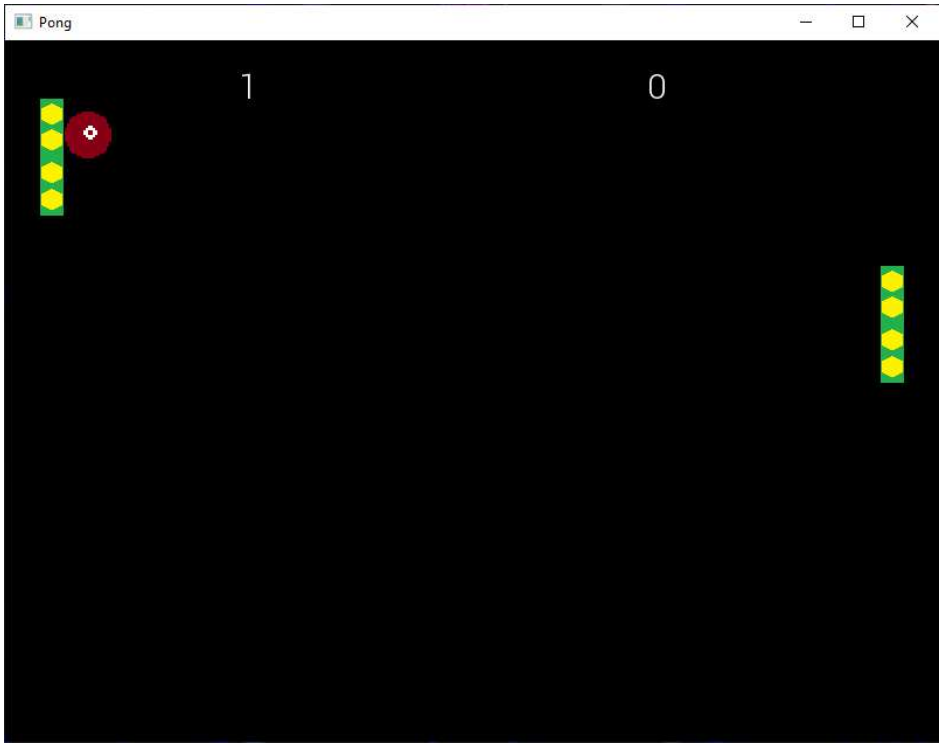


Obrázek 24 Snímek hry krátce před kolizí s vertikálním krajem obrazovky

(zdroj: vlastní)

Pokud však některý z hráčů chce míček odpálit pak musí proběhnout část kódu, kde logika řeší kolizi míčku a pálky. Kolize pro obě pálky se řeší takřka identickým způsobem, pouze boolean proměnná bounced má prohozené hodnoty.

```
if (ball.getBounds().findIntersection(paddle.getBounds())) {
        if (ball.getDirection().x < 0 && ball.bounced != 1) {
 ball.setPosition(paddle.getPosition().x + paddle.getSize().x + 0.1f,
 ball.getPosition().y);
 float paddleCenterY = paddle.getPosition().y + paddle.getSize().y /
 2.f;
 float ballCenterY = ball.getPosition().y + ball.radius;
            float diff = ballCenterY - paddleCenterY;
            float normalized = diff / (paddle.getSize().y / 2.f);
            ball.setDirection(normalized);
            ball.bounceX();
            ball.bounced = 1;
            SoundManager::playSound("pong");
        }
    }
```



Obrázek 25 Kolize pálky s míčkem v aplikaci

(zdroj: vlastní)

Posledním krokem metody ***update*** je aktualizace hodnoty skóre (viz **Error! Reference source not found.**)

Následně proběhne metoda ***render***, která pouze aktualizuje a vykreslí všechny objekty na obazovku.

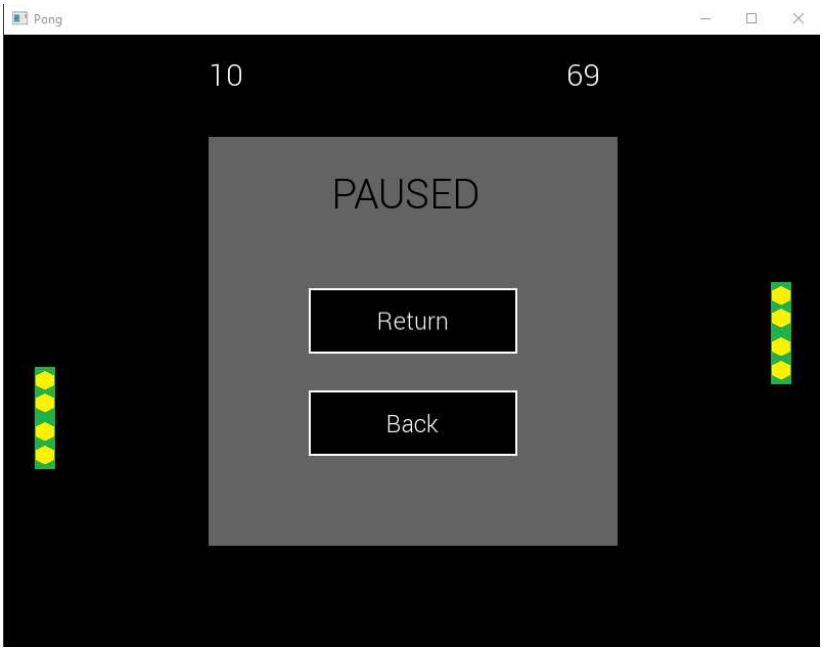
```
void Game::render() {
    window.clear();
    paddle.render(window);
    paddleAi.render(window);
    ball.render(window);
    window.draw(scoreText1);
    window.draw(scoreText2);
}
```

7.3.4 Pause State

PauseState slouží jako překryvná vrstva, která dočasně zastaví běh hry. Vyvolává se v GameState pomocí stisku klávesy Escape.

```
if (event.type == sf::Event::KeyPressed && event.key.code ==
 sf::Keyboard::Escape) {
 stateHandler.pushState(std::make_shared<PauseState>(stateHandler,
 window));
```

```
}
```



Obrázek 26 Hra pozastavena

(zdroj: vlastní)

Využívá efektu průhledného pozadí, díky čemuž zůstává herní obraz stále viditelný. Součástí stavu jsou dvě tlačítka: Pokračovat ve hře a Návrat do menu. Tlačítka jsou opět vytvořena jako instance tříd odvozených z Button.

```
PauseState::PauseState(StateHandler&     handler,     sf::RenderWindow&
 window)  : stateHandler(handler), window(window)
{
    font.loadFromFile("Thirdparty/fonts/Roboto-Light.ttf");
    background.setSize(sf::Vector2f(400, 400));
    background.setFillColor(sf::Color(100, 100, 100, 200));
    background.setPosition(200, 100);
    title.setFont(font);
    title.setString("PAUSED");
    title.setCharacterSize(40);
    title.setFillColor(sf::Color::Black);
    title.setPosition(320, 130);

 buttons.push_back(std::make_unique<ResumeButton>(sf::Vector2f(200,
 60), sf::Vector2f(300, 250), font, stateHandler));
    buttons.push_back(std::make_unique<BackButton>(sf::Vector2f(200,
 60), sf::Vector2f(300, 350), font, stateHandler, window));}
```

Metoda ***handleInput*** kromě kliknutí zpracovává také stisk klávesy Escape, který provede návrat zpět do hry.

```
if (event.type == sf::Event::KeyPressed) {
            if (event.key.code == sf::Keyboard::Escape) {
                stateHandler.popState();
            }
        }
```

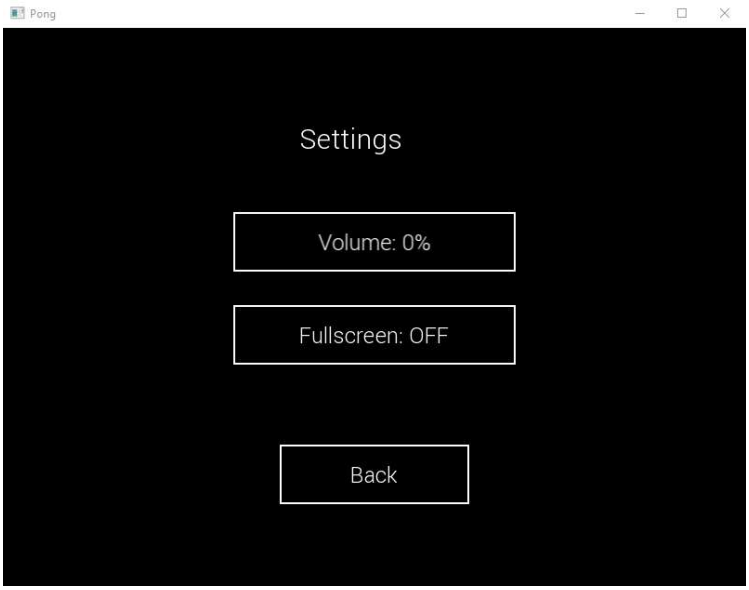
Díky tomu, že stav GameState zůstává na zásobníku, není třeba složitě ukládat herní stav – hra plynule pokračuje, jakmile se PauseState odstraní.

7.3.5 MpGameState

Multiplayerová verze hry je realizována ve stavu MpGameState. Ten zajišťuje nejen herní logiku podobnou jako v GameState, ale také inicializaci a běh síťové komunikace. Po spuštění stavu je podle zvolené role (hostitel/klient) spuštěno síťové vlákno, které obsluhuje příjem a odesílání dat přes TCP sockety. Více k tématu hry více hráčů v kapiole Networking.

7.3.6 SettingsState

Tento stav slouží k nastavení jednak hlasitosti hudby a zvukových efektů a druhak režimu obrazovky – okno či režim celé obrazovky.



Obrázek 27 SettingsState

(zdroj: vlastní)

7.4 Vykreslování objektů

7.4.1 Vykreslování základních tvarů

V implementaci byly využity základní tvary poskytnuté knihovnou SFML jako např ***sf::RectangleShape*** či ***sf::CircleShape***. Tyto tvary znázornili obě pálky a míček. Pro pálky bylo potřeba určit rozměry jednotlivých stran, barvu a základní pozici. Tvar míčku vyžaduje stejné údaje s jedinou změnou - radius místo délek stran. Entity použité v aplikaci mají konstruktor sestavený právě pro předání těchto nezbytných údajů.

```
Paddle::Paddle(float x, float y, float screenHeight) :
 screenHeight(screenHeight) {
    rectShape.setSize(sf::Vector2f(20.f, 100.f));
    rectShape.setFillColor(sf::Color::White);
    rectShape.setPosition(x, y);
    shape = &rectShape;
}
```

K samotnému vykreslení poté slouží metoda ***window.draw***, které byl předán tvar entity.

```
void Paddle::render(sf::RenderWindow& window) {
    window.draw(rectShape);
}
```

Tento způsob vykreslování se uplatnil v první verzi aplikace, kde postačovaly jednoduché geometrické tvary. Následující kapitola ukazuje přechod složitější strukturu - práci s texturami pro dosažení vyšší vizuální kvality.

7.4.2 Vykreslování textur

Vykrelování textur probíhá pomocí stejné metody jako vykreslování základního tvaru, pokud má přiřazenou texturu. K tomu slouží metoda ***setTexture***. Pro obecnost byla vytvořena třída TextureManager, jejíž jediný účel je načíst textury do bitmapy a následně předat entitám, které si o texturu zažádají.

```
std::unordered_map<std::string, sf::Texture> TextureManager::textures;

sf::Texture& TextureManager::getTexture(const std::string& filename) {
    auto it = textures.find(filename);
    if (it != textures.end()) {
        return it->second;
    }
```

```
        else {
            sf::Texture texture;
            if (!texture.loadFromFile(filename)) {
                throw std::runtime_error("Failed to load texture: " +
    filename);
            }
            textures[filename] = std::move(texture);
            return textures[filename];
        }
    }
```

V praxi je textura načtena a přiřazena entitě (např. pálce) během inicializace.

```
        auto& paddleTexture =
    TextureManager::getTexture("Thirdparty/textures/paddle.png");
        auto& ballTexture =
    TextureManager::getTexture("Thirdparty/textures/ball.png");
```

7.4.3 Vykreslování animací

Vykreslování animací navazuje na textury v tom smyslu slova, že entita vytvoří tvar, načte texturu a textura samotná je animací. Pro vytvoření pohyblivého obrázku bylo potřeba přidat třídu AnimationManager, jejíž jediný účel je rozkouskovat texturu do jednotlivých snímků animace a přehrávat je počas běhu aplikace. Konstruktor této třídy využije textury jako sprite, automaticky nastaví nultý snímek a rozlišení daného spritu v dané textuře pro následující animaci pomocí metody ***update***. Tato metoda postupně prochází texturu dokud se nedostane na její konec – pak se resetuje snímek na nultý index.

V realizaci vypadá konstruktor a hlavní funkce následovně:

```
AnimationManager::AnimationManager(sf::Sprite& sprite, int frameCount,
 float frameTime, sf::Vector2i frameSize)
    : sprite(&sprite), frameCount(frameCount), frameTime(frameTime),
    timer(0.f),   currentFrame(0),   isPlaying(true),   looping(true),
 frameSize(frameSize)
{
    sprite.setTextureRect(sf::IntRect(0,        0,        frameSize.x,
 frameSize.y));
}


void AnimationManager::update(float dt) {
    if (!isPlaying || frameCount <= 1) return;
    timer += dt;
    if (timer >= frameTime) {
        timer -= frameTime;
        currentFrame++;
        if (currentFrame >= frameCount) {
            if (looping) {
                currentFrame = 0;
            }
            else {
                currentFrame = frameCount - 1;
                isPlaying = false;
            }
        }
        sprite->setTextureRect(sf::IntRect(currentFrame * frameSize.x,
 0, frameSize.x, frameSize.y));
    }
}
```

7.5 Přehrávání hudby a zvuků

Prvním krokem je volání metod ***loadSound*** a ***loadMusic***, které načtou zvukový soubor do paměti. Následně lze tyto zvuky přehrát metodami ***playSound*** a ***playMusic***. Pro nastavení hlasitosti je nutno použít metodu ***setVolume***. V implementaci byla zvolena cesta vytvoření třídy MusicManager, která následně bude řešit veškerou práci se zvukovými soubory a metodami s nimi spojenými.

```
void SoundManager::loadSound(const std::string& name, const
 std::string& filepath) {
    sf::SoundBuffer buffer;
    if (!buffer.loadFromFile(filepath)) {
        std::cerr << "Failed to load sound: " << filepath << std::endl;
        return;
    }
    soundBuffers[name] = buffer;
    sf::Sound sound;
    sound.setBuffer(soundBuffers[name]);
    sound.setVolume(currentVolume);
    sounds[name] = sound;
}
void SoundManager::playSound(const std::string& name) {
    auto it = sounds.find(name);
    if (it != sounds.end()) {
        it->second.play();
    }
}
void SoundManager::loadMusic(const std::string& filepath) {
    if (!music.openFromFile(filepath)) {
        std::cerr << "Failed to load music: " << filepath << std::endl;
    }
    music.setVolume(currentVolume);
}
void SoundManager::playMusic(bool loop) {
    music.setLoop(loop);
    music.play();
}
```

7.6 Práce s textem

Pro práci s textem je nutné načíst font a uložit ho jako objekt ***sf::Font***. V implementaci k tomu slouží běžné řešení pomocí ***loadFromFile***. Pro práci s textem jsou využity běžné SFML metody setFont, setCharacterSize, setFillColor, setPosition a setString. Ve hře se tedy textu přednastaví vlastnosti.

```
scoreText1.setFont(font);
    scoreText1.setCharacterSize(30);
    scoreText1.setFillColor(sf::Color::White);
    scoreText1.setPosition(200.f, 20.f);

    scoreText2.setFont(font);
    scoreText2.setCharacterSize(30);
    scoreText2.setFillColor(sf::Color::White);
    scoreText2.setPosition(550.f, 20.f);
```

A následně se dynamicky aktualizuje text.

```
    scoreText1.setString(std::to_string(score1));
    scoreText2.setString(std::to_string(score2));
```

7.7 Networking – hra přes síť

K navázání spojení slouží kombinace protokolů UDP a TCP. Nejprve klient využije UDP broadcast, pomocí jehož detekuje přítomnost hostitele na lokální síti bez předchozí znalosti jeho IP adresy.

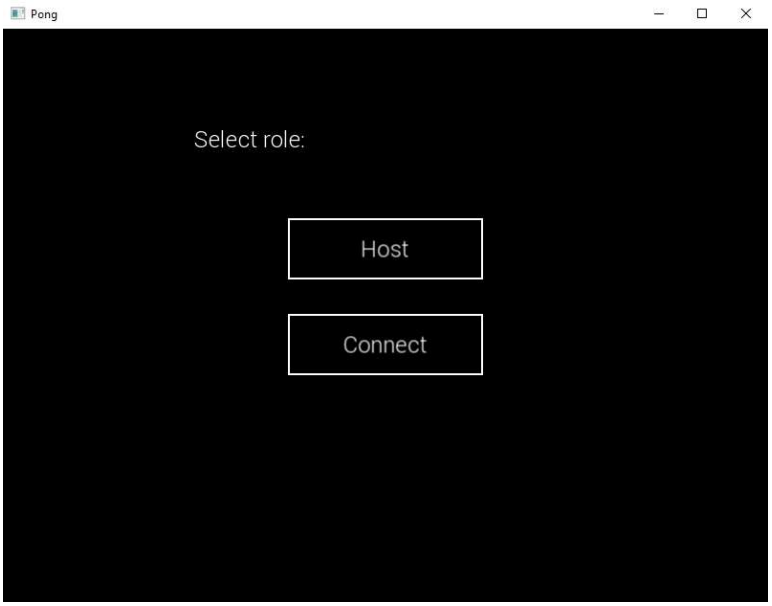
```
void MpGameState::hostListen() {
    if (listener.listen(4500) != sf::Socket::Done) {
        std::cerr << "[HOST] Failed to listen on port 4500\n";
        return;
    }

    std::thread([this]() {
        sf::UdpSocket udpSocket;
        udpSocket.setBlocking(false);
        sf::Packet packet;
        packet << std::string("PONG_HOST");

        while (running && !isConnected) {
```

```
            udpSocket.send(packet, sf::IpAddress::Broadcast, 4502);
            sf::sleep(sf::seconds(1));
        }
        }).detach();

        std::cout << "[HOST] Listening on LAN IP: " << sf::IpAddress::getLocalAddress() << ":4500\n";

        if (listener.accept(socket) == sf::Socket::Done) {
            socket.setBlocking(false);
            std::cout << "[HOST] Client connected\n";
            isConnected = true;
            std::thread(&MpGameState::networkLoop, this).detach();
        }
}
```

Následně pokud je host nalezen, naváže klient s hostem TCP spojení, které poté slouží pro obousměrnou komunikaci a výměnu herních dat.

```
void MpGameState::clientConnect() {
    sf::UdpSocket udpSocket;
    if (udpSocket.bind(4502) != sf::Socket::Done) {
        std::cerr << "[CLIENT] Failed to bind UDP socket\n";
        return;
    }
    udpSocket.setBlocking(false);
    sf::IpAddress hostIp;
    sf::Clock timeoutClock;
    bool foundHost = false;
    while (timeoutClock.getElapsedTime().asSeconds() < 5.0f) {
        sf::Packet packet;
        sf::IpAddress sender;
        unsigned short port;
        if (udpSocket.receive(packet, sender, port) == sf::Socket::Done) {
            std::string header;
            if (packet >> header && header == "PONG_HOST") {
                hostIp = sender;
                foundHost = true;
                break;
            }
```

```
            }
            sf::sleep(sf::milliseconds(100));
        }
        udpSocket.unbind();
        if (!foundHost) {
            std::cerr << "[CLIENT] Could not find host on LAN\n";
            return;
        }
        std::cout << "[CLIENT] Found host at " << hostIp.toString() << ",
 attempting to connect...\n";
        if     (socket.connect(hostIp,     4500,     sf::seconds(5))     ==
 sf::Socket::Done) {
            socket.setBlocking(false);
            std::cout << "[CLIENT] Connected to host\n";
            isConnected = true;
            std::thread(&MpGameState::networkLoop, this).detach();
        }
        else {
            std::cerr << "[CLIENT] Connection to host failed\n";
        }
    }
```



Obrázek 28 Výběr role aplikace v kontextu síťové hry

(zdroj: vlastní)

Z důvodu možné latence byl zvolen přístup komunikace přes více vláken. Hostitelova práce je nejen řešit pohyb pálky svého hráče ale také udržování logiky hry. Klientovou prací je

pouze posílat vstup hráče a následně přijmout herní informace od hostitele a zobrazit je v okně. Tímto způsobem se předchází možným nesrovnalostem způsobeným latencí.

Veškerá komunikace probíhá pomocí třídy sf::TcpSocket, přičemž data jsou serializována pomocí sf::Packet. Ten vezme všechna data a pošle je jako balíček. Je nutné dodržet stejnou strukturu dat na obou zařízeních. Pro zlepšení plynulosti hry pak klient zobrazí data jen z posledního přijatého paketu.

Komunikace Hosta s Klientem.

```
if (isHost) {
            sf::Packet inputPacket;
            if (socket.receive(inputPacket) == sf::Socket::Done) {
                float y;
                if (inputPacket >> y) {

 paddleClient.setPosition(paddleClient.getPosition().x, y);
                }
            }
            sf::Packet gamePacket;
            gamePacket.clear();
            gamePacket << ball.getPosition().x << ball.getPosition().y
                << ball.getDirection().x << ball.getDirection().y
                <<            paddleHost.getPosition().x            <<
 paddleHost.getPosition().y
                << score1 << score2;
            if (socket.send(gamePacket) != sf::Socket::Done) {
                std::cerr << "[HOST] Failed to send packet!\n";
            }
        }
```

A klientova komunikace nazpět.

```
else {   sf::Packet inputPacket;
            inputPacket << paddleClient.getPosition().y;
            if (socket.send(inputPacket) != sf::Socket::Done) {
                std::cerr << "[CLIENT] Failed to send input!\n";
            }
            sf::Packet gamePacket;
            if (socket.receive(gamePacket) == sf::Socket::Done) {
                std::lock_guard<std::mutex> lock(packetMutex);
                latestPacket = gamePacket;
```

```
                hasNewPacket = true;
                float bx, by, dx, dy, hostX, hostY;
                int s1, s2;
                if (gamePacket >> bx >> by >> dx >> dy >> hostX >>
hostY >> s1 >> s2) {
                    ball.setPosition(bx, by);
                    ball.setVelocity({ dx, dy });
                    paddleHost.setPosition(hostX, hostY);
                    score1 = s1;
                    score2 = s2;
                }
                else {
                    std::cerr << "[CLIENT] Failed to parse packet!\n";
                }
            }
        }
```

ZÁVĚR

Tato bakalářská práce přibližuje čtenáři knihovnu SFML a její místo ve světě vývoje her. Jednoznačně definuje SFML. Popisuje výhody a nevýhody této knihovny. Srovnává je s ostatními nástroji používanými pro vývoj her. A na praktickém příkladu názorně předvádí všechny podstatné metody různých modulů.

Teoretická část práce krátce popsala SFML knihovnu a její moduly doplněné o ilustrační kód, zmínila i doplňující knihovnu rozšiřující SFML o chybějící GUI, porovnala tuto knihovnu s ostatními nástroji na trhu. Výsledkem porovnání byla velká implikace pro využití tohoto nástroje pro osobní projekty, prototypizaci a výuku herního vývoje. Dále se v této části rozebírá teoretické fungování metod použitých v praktické části.

Praktická část následně informace popsané v teorii implementovala do několika specializovaných aplikací, které názorně předvedli funkcionalitu jednotlivých modulů. Tyto aplikace se staly základním stavebním kamenem pro následný vývoj funkční aplikace. Jednoduchá logika aplikace načte stavový manažer, který se následně stará o vyobrazení jednotlivých vrstev. Prvně se načte stav menu s tlačítky, odkud se uživatel může přesunout do stavů Game, MpGame a Settings či aplikaci jedním stisknutím tlačítka vypnout. Stav Game řeší inicializaci objektů, jejich vykreslení a herní logiku, jejíž práce je aktualizovat pozici objektů a vyhodnocovat kolize či skóre. Při stisku klávesy Escape se vyvolá stav Pause, který se stará o banální menu se dvěma tlačítky (návrat do hry či návrat do menu), tento stav neaktualizuje celé pozadí, díky čemuž je stav Game na pozadí zmražen. Stav Settings se stará o velikost okna a hlasitost audia. Stav MpGame přesouvá uživatele do stavu, jehož úkonem je propojit dvě instance aplikace do jedné hry dvou hráčů. Tento úkon je řešen pomocí UDP a TCP protokolů, kde první z nich vysílá zprávu do sítě a pokud se najde jiné zařízení, které je ochotné tuto zprávu přijmout, pak se pomocí TCP protokolu vytvoří spojení. Hostitelské zařízení řídí veškerou logiku hry a posílá údaje zabalené jako balíček (packet) na klientské zařízení, jež tyto data zobrazí na své obrazovce a vrátí hostitelskému zařízení vstup od svého uživatele. Největším problémem hry více hráčů byla latence, která bránila efektivnímu hraní hry bez zvýhodňování uživatele hostitelského zařízení. Tento problém je částečně řešen pomocí dvou vláken, kde jedno se stará o odesílání a druhé o příjem dat. Dále bylo implementováno vícero metod, které se snaží latenci redukovat.

SEZNAM POUŽITÉ LITERATURY

[1] BARBIER, Maxime. SFML Blueprints: Sharpen Your Game Development Skills and Improve Your C++ and SFML Knowledge with Five Exciting Projects. Birmingham: Packt Publishing, 2015. ISBN 978-1-78439-577-3.

[2] SFML Development Team. Time – Playing with time values [online]. SFML tutorials. [cit.2025-06-02]. Dostupné z: https://www.sfml-dev.org/tutorials/3.0/system/time/

[3] SFML Development Team. Window – Opening a window [online]. SFML tutorials. [cit. 2025-06-02]. Dostupné z: https://www.sfml-dev.org/tutorials/3.0/window/window/

[4] SFML Development Team. Graphics – Shape and color [online]. SFML tutorials. [cit. 2025-06-02]. Dostupné z: https://www.sfml-dev.org/tutorials/3.0/graphics/shape/

[5] SFML Development Team. Audio – Loading and playing a sound [online]. SFML tutorials. [cit. 2025-06-02]. Dostupné z: https://www.sfml-dev.org/tutorials/3.0/audio/sounds

[6] SFML Development Team. Network – TCP vs UDP [online]. SFML tutorials. [cit. 2025-06-02]. Dostupné z: https://www.sfml-dev.org/tutorials/3.0/network/socket

[7] SFML + ImGui. imgui-sfml [online]. GitHub. [cit. 2025-06-02]. Dostupné z: https://github.com/SFML/imgui-sfml

[8] SDL. Simple DirectMedia Layer – Homepage [online]. [cit. 2025-06-02]. Dostupné z: https://www.libsdl.org

[9] Wikipedia contributors. Godot (game engine) [online]. Wikipedia, The Free Encyclopedia. [cit. 2025-06-02]. Dostupné z: https://en.wikipedia.org/wiki/Godot_(game_engine)

[10] Wikipedia contributors. GameMaker [online]. Wikipedia, The Free Encyclopedia. [cit. 2025-06-02]. Dostupné z: https://en.wikipedia.org/wiki/GameMaker

[11] Wikipedia contributors. Unity (game engine) [online]. Wikipedia, The Free Encyclopedia. [cit. 2025-06-02]. Dostupné z: https://en.wikipedia.org/wiki/Unity_(game_engine)

[12] MRLvsb. Herní smyčka – SDL [online]. MRLvsb.github.io. [cit. 2025-06-02]. Dostupné z: https://mrlvsb.github.io/upr-skripta/c/aplikovane_ulohy/sdl/herni_smycka.html

[13] INDUWARI, Paramee Supipi. Finite State Machines in Game Development [online]. Medium, 2023. [cit. 2025-06-02]. Dostupné z: https://medium.com/@parameesupipiinduwari/finite-state-machines-in-gamedevelopment-3c12dc4d667a

[14] Wikipedia contributors. Interpolation [online]. Wikipedia, The Free Encyclopedia. [cit. 2025-06-02]. Dostupné z: https://en.wikipedia.org/wiki/Interpolation

[15] HALLER, Jan, HANSSON, Henrik Vogelius, MOREIRA, Artur. SFML Game Development. Birmingham: Packt Publishing, 2013. ISBN 978-1-84969-684-5.

[16] PUPIUS, R. SFML Game Development By Example. Packt Publishing, 2015. ISBN 9781785287343.

[17] SHEKAR, Siddharth, 2019. C++ Game Development By Example. Sonar Publishing. ISBN 9781789537345.

[18] CHANDLER, Heather a Rafael CHANDLER, 2011. Fundamentals of Game Development. 3rd ed. Jones & Bartlett Learning. ISBN 9780763778958. 0763778958.

[19] MARTIN, Robert C., 2009. Clean Code: A Handbook of Agile Software Craftsmanship. 2nd ed. Prentice Hall. ISBN 9780132350884.

[20] Wikipedia contributors. Interpolation [online]. Wikipedia, The Free Encyclopedia. [cit. 2025-06-02]. Dostupné z: https://en.wikipedia.org/wiki/Corrupted_Blood_incident

[21] CG Spectrum. *Difference Between AAA vs Indie Game Studio* [online]. CG Spectrum Blog, 12. srpna 2022. [cit. 2025-06-02]. Dostupné z: https://www.cgspectrum.com/blog/difference-between-aaa-vs-indie-game-studio

[22] Mendelova univerzita v Brně – Fakulta pedagogická. Multimediální pomůcka z audiovizuální kultury – Herní žánry [online]. [cit. 2025-06-02]. Dostupné z: https://is.muni.cz/elportal/estud/pedf/js08/avk/ucebnice/lekce14.htm

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

SFML Simple and Fast Multimedia Library

API Application Programming Interface

GUI Graphical User Interface

SDL Simple DirectMedia Layer

GML GameMaker Language

UDP User Datagram Protocol

TCP Transmission Control Protocol

LAN Local Area Network