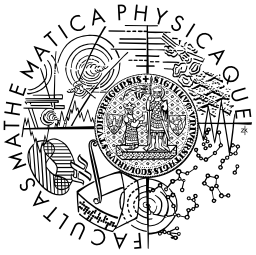


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Milan Kotva

2D Game editor

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D.
Studijní program: Informatika

Praha 2024

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne 9. 5. 2024

Rád bych tímto upřímě poděkoval panu RNDr. Martin Pergelovi, Ph.D vedoucímu mé práce za odborné rady, podporu a pomoc při vzpracování této práce.

Dále bych rád poděkovali své rodině a nejbližším přátelům, za jejich neustálou podporu a povzbuzení.

Speciální poděkování patří mé partnerce Tereze Kotlabové za neocenitelnou podporu a péči.

Název práce: 2D Game editor

Autor: Milan Kotva

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: V této práci představujeme editor 2D her poskytující prostředí pro tvorbu jednoduchých her, jako jsou například FlappyBird nebo JumpKing. Předkládáme tak alternativu k jiným editorům, které jsou buďto placené (např. Construct 3), nebo vyžadují vyšší úroveň znalostí a zkušeností s programováním (např. GameMaker nebo Godot). Náš editor nabízí možnost použití vlastní grafiky, hudby či úpravu fyzikálních pravidel změnou hodnot vlastností objektů. Pro pokročilejší uživatele je navíc k dispozici jednoduchý skriptovací jazyk, pomocí něhož lze tyto vlastnosti také měnit. Použití editoru demonstrujeme na příkladu tradiční hry Mario a již zmiňované hry FlappyBird.

Klíčová slova: Herní editor, 2D hra, Vývoj her, Skriptování

Title: 2D Game Editor

Author: Milan Kotva

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Martin Pergel, Ph.D., Department of Software and Computer Science Education

Abstract: In this thesis, we introduce 2D game editor providing environment for creation of simple computer games, such as FlappyBird or JumpKing. We present an alternative to other existing game editors, which are either paid (such as Construct 3), or require higher proficiency level and programming skills (such as GameMaker or Godot). Our editor offers interesting features such as usage of custom graphics, music, or customization of laws of physics. For advanced users, it provides simple scripting language, which can be used to modify those properties as well. The usage of the editor is demonstrated on the examples of the traditional game Mario and already mentioned game FlappyBird.

Keywords: Game editor, 2D game, Game development, Scripting

Obsah

Úvod

Počátkem nového tisíciletí začal vznikat trend velkých herních studií vytvářejících známé herní tituly. Tyto velké tituly ovládly většinu herního trhu na další dvě desetiletí. V dnešní době se ale začíná stále více rozvíjet trend menších a jednodušších her, jelikož velké množství současných velkých herních titulů nepřináší mnoho inovací a spíše spoléhá na osvědčené herní mechaniky.

„Hlad"po inovacích motivuje nemalé možství vývojářů vymýšlet vlastní mechaniky, příběhy či styly her. Asi každý hráč počítačových her zažil moment, kdy si přál, aby jeho hra měla více herních prvků nebo aby se dala hrát s přáteli. Napsat či upravit hru není tak snadné, jak se může zdát. To může spoustu hráčů, kteří si chtějí pouze zahrát jednoduchou zábavnou hru podle svých představ, odradit. Většina současných nejběžněji používaných nástrojů pro vývoj her vyžaduje pokročilé znalosti programování a zkušenosti s vývojem softwaru. Příkladem mohou být třeba herní enginy **Unity**, **Unreal Engine**, nebo **Godot**.

Cílem této práce je umožnit běžným uživatelům jednoduše vytvářet netriviální množství jednoduchých platformových her a zároveň jim poskytnout možnost osvojit si základní principy vývoje her a programování, a to bez nutnosti se dlouhé hodiny učit používat komplexní nástroje pro vývoj her.

V průběhu vývoje se zaměříme na práci s jednotlivými herními prvky, na jejich vytváření a upravování jejich vlastností, jako je vzhled, pohyb nebo interakce mezi jednotlivými herními objekty. K tomuto účelu navrhneme jednoduchý skriptovací jazyk, který nám umožní všechny tyto prvky definovat podle našich požadavků. Jelikož je cílem práce vytvořit co nejjednodušší prostředí pro vývoj her, bude tento jazyk opravdu minimalistický a jeho hlavním účelem bude dát uživateli možnost spustit předpřipravené funkce v okamžiku, který si uživatel určí.

V rámci zvýšení pohodlí uživatele bude součástí práce i velmi jednoduchá analýza průchodnosti herní úrovně a možnost testování hry přímo v editoru. Jako součást práce vytvoříme i prostředí, ve kterém budeme hry hrát. Dále se zaměříme na již vytvořené populární platformové hry a jejich mechaniky, například **Super Mario** nebo **FlappyBird** a pokusíme se je replikovat.

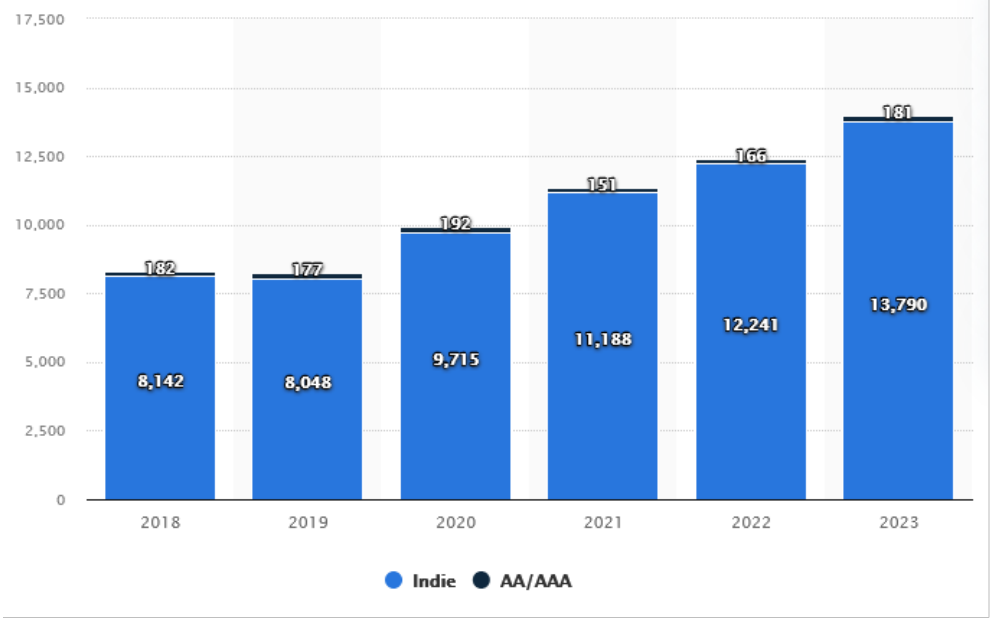
Fenomén vytváření her

Jak jsme již zmínili v úvodu, herní trh byl v posledních dvou dekádách dominován převážně velkými herními studii. Toto můžeme snadno demostrovat na sérii **Call of Duty**, od studia **Activision Blizzard**. Hry z této série se v posledních dvaceti letech jedenatřicetkrát umístily mezi deseti nejprodávánějšími hrami roku a dvanáctkrát byly dokonce nejprodávanější na americkém trhu [1].

Toto však není ojedinělý případ velkých herních sérií ovládajících herní trh po dlouhé roky. Dalším příkladem může být herní série **Grant Theft Auto**, od studia **Rockstar North**, jejíž nejúspěšnější hra **Grant Theft Auto V** se stala jednou z nejprodávanějších her v historii, skoro s dvěstě miliony prodaných kopií [2].

Na následujícím grafu pak můžeme pozorovat počet vydaných **Indie** a **AAA her** od roku 2018. Pojem **AAA hry** je neformální klasifikací videoher vytvářených nebo distribuovaných středními nebo velkými vydavateli. Typicky jsou tyto hry

vytvářené velkým počtem vývojářů s větším rozpočtem, obvykle dosahující až stovek milionů dolarů. Většinou je také odlišuje nejen kvalita produkce, ale také cena, která se zpravidla pohybuje okolo šedesáti dolarů při vydání. [3] Naproti tomu, pojem **Indie hry** označuje videohry vyvíjené jednotlivci či malými herními studii s počty zaměstnanců v řádu nízkých desítek, a také většinou s omezeným rozpočtem. Narozdíl od **AAA her** se vývojáři **Indie her** snaží vyvážit nedostatek financí inovacemi v obsahu a formě her. [4]



Obrázek 1 Počet her publikovaných na herní platformě **Steam** [5]

Jak můžeme nahlédnout, počet nově vydaných **Indie her** v posledních šesti letech stoupl o sedmdesát procent. Ve srovnání s tím se počet vydaných **AAA her** meziročně liší v jednotkách kusů a nevykazuje rostoucí trend. Tato skutečnost naznaznačuje rostoucí zájem o jednodušší, ale inovativnější hry, a s tím spojený zájem o tvorbu her tohoto typu [6].

1 Rešerše existujících platforem

Jak jsme již nastínili v úvodu , máme na náš editor určité požadavky, které si nyní trochu více rozebereme. Naším cílem je vytvořit prostředí, ve kterém lze naprogramovat netriviální podmožinu her, tedy aby se v něm dala vytvořit skupina různorodých her, ale aby bylo zároveň příjemné na používání.

1.1 Rešerše existujících editorů

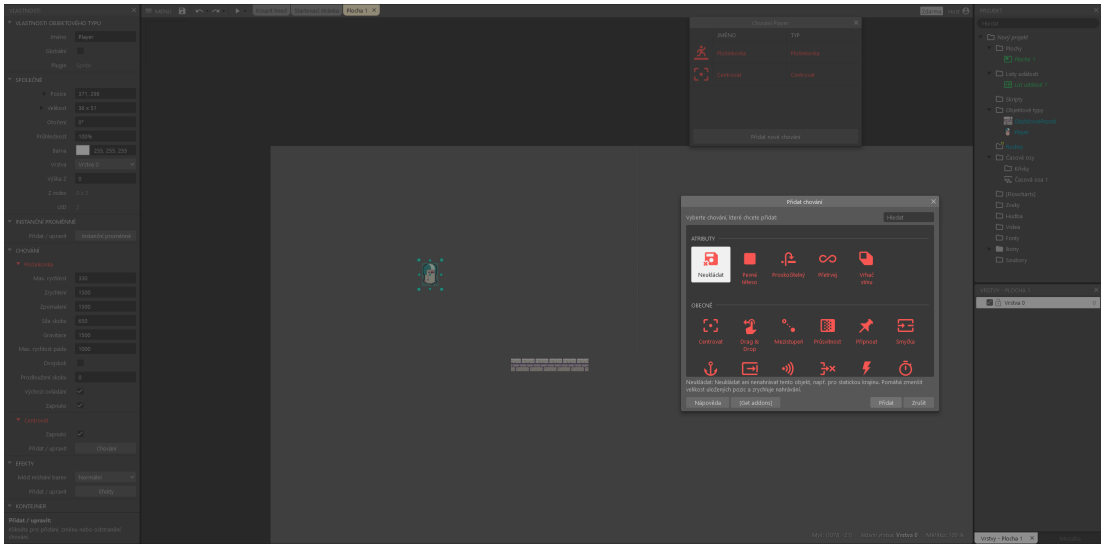
Nyní si ukážeme tři příklady editorů her, ve kterých je možné vytvářet platformové hry a podíváme se, zda a jak splňují naše nároky, případně jak byly implementovány. Tyto nároky budeme definovat z pohledu uživatele neprogramátora:

- **vhodnost pro začátečníky** - z definice nás tedy bude zajímat míra zkušeností s programováním, které jsou pro práci v editoru vyžadovány;
- **vytváření objektů** - jakým způsobem funguje vytváření herních objektů a jejich konfigurace;
- **rozšiřitelnost** - jakým způsobem lze vytvořit herní funkce, mechaniky nebo komponenty, které nejsou součástí výchozího editoru;
- **funkce** - jaké další zajímavé funkce editor poskytuje;
- **využitelnost** - jaké druhy her jsme schopni v daném editoru vytvořit;
- **licenční podmínky a poplatky** - jakými poplatky a licenčními podmínkami je zatížen vývoj v editoru.

1.1.1 Construct

Herní editor **Construct** je komerční editor, jehož první verze byla vydána v roce 2007, [7] a je zaměřený na co nejsnazší vytváření a výuku tvorby **2D her** [8].

Editor umožňuje velice snadné vytváření herních prvků s možností nastavování vlastních obrázků, animací a zvuků z již předpřipravených zdrojů, či přímo z počítače uživatele. Akce a vlastnosti jednotlivých objektů jsou pak nastavovány z předpřipravených možností a zbavují tak uživatele jakékoli nutnosti programovat. Mezi tyto vlastnosti patří například to, zda má objekt pevné hranice nebo jestli se na něj vztahují fyzikální zákony. Přidávání těchto vlastností pak funguje na pricipu připojování jednotlivých komponent, tedy pokaždé, když chceme rozšířit objekt, přidáme komponentu, kterou nastavíme podle našich požadavků, a uložíme ji do seznamu komponent daného objektu.



Obrázek 1.1 **Construct** - Ukázka nastavení akcí

Pokud ale uživatel chce vybočit z akcí předpřipravených prostředím, například je upravit nebo vytvořit vlastní, je zapotřebí znalosti jazyka **JavaScript**, ve kterém se píší všechny požadované doplňky.

Mezi další možnosti tohoto editoru patří přidávání grafických efektů, jako například barevné filtry, které uživateli umožnují třeba vytvořit tématicky laděné hry v černobílem stylu bez nutnosti upravovat jednotlivé obrázky, nebo přidání optických zkreslení jako je lom světla přes čočku, případně zkreslování audio stop pro umělé zvýšení tóniny.

Cena za použití tohoto editoru je rozlišena v rámci níže uvedených kategorií [9]:

Typ využití	**Roční cena**
Soukromé	2 899,99 CZK
Podnikání	10 899 CZK
Výuka	789,99 CZK

Tabulka 1.1 Přehled cen licencí **Construct 3**

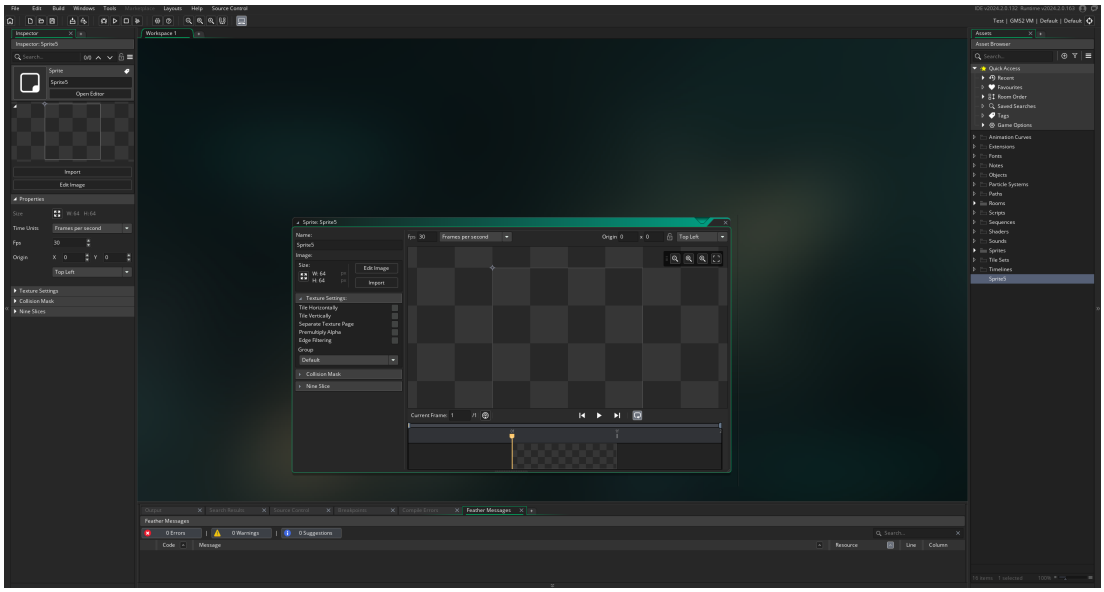
Editor proto splňuje podmínku ***vhodnost pro začátečníky***. Negativním aspektem editoru je nutnost zakoupit licenci i pro soukormé účely.

1.1.2 GameMaker

GameMaker je herním editorem ideálním pro nováčky, kteří se chtějí naučit něco více o vytváření počítačových her a programování ve zjednodušené formě [10].

Editor disponuje velkou řadou funkcí, od vytváření her pomocí návrhových vzorů pro jednotlivé kategorie, jako třeba skákačky nebo střílečky, až po upravování

jednotlivých textur objektů či nejdetailnější upravování akcí jednotlivých prvků pomocí již zmíněného skriptovacího jazyka.



Obrázek 1.2 GameMaker - Upravování textur

Narozdíl však od předchozího příkladu (1.1.1) tento editor vyžaduje větší odhodlání k překonání počáteční neznalosti prostředí. Tato počáteční bariéra je zapříčiněna větším množstvím funkcionalit a možností oproti zmíněnému **Construct**, které ho činí komplikovanější.

Příkladem, na kterém můžeme ukázat větší komplexitu GameMakeru, může být pouhé ***vytvoření herního objektu***. Narozdíl od předchozího příkladu můžeme objektu vytvořit textury či animace na míru, upravovat jednotlivé pixely a nebo přidat objekt do různých vrstev v herním levelu. To vše ale vyžaduje netriviální znalost editoru, narozdíl od předešlého příkladu, kde sice nebylo tolik možností, ale editor byl jednodušší na ovládání.

Základní verze tohoto editoru je zdarma pro nekomerční využití. V ostatních případech si uživatel musí zakoupit buďto jednorázovou licenci, nebo platit roční členství, které mu umožní používat prémiové funkce [11].

Typ licence	**Roční cena**
Jednorázová	$99.79
Roční	$79.99

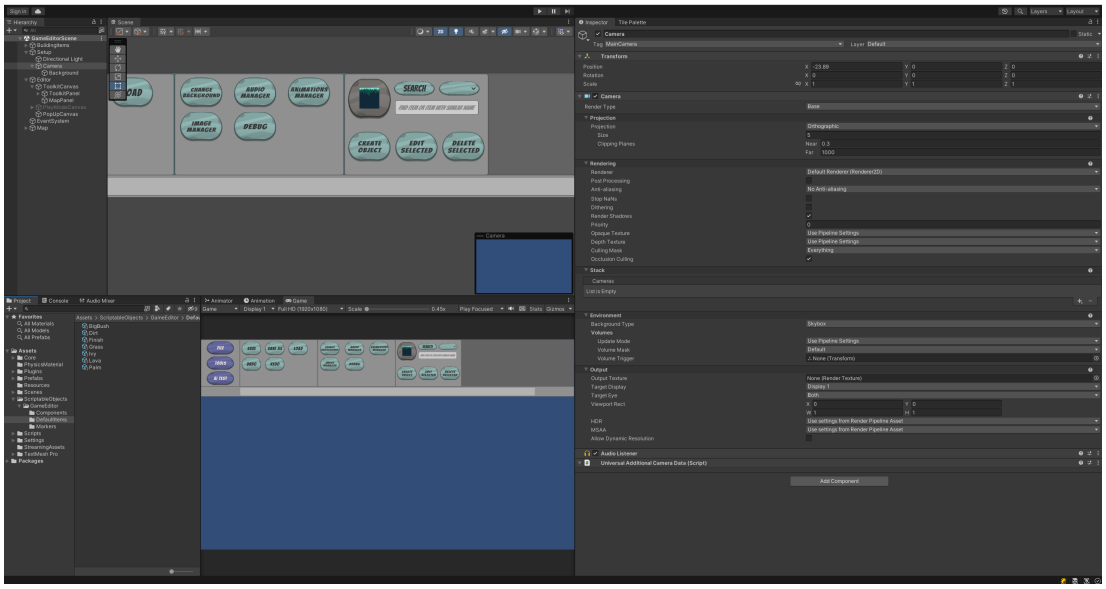
Tabulka 1.2 Přehled cen licencí **Game Maker**

O tomto editoru tedy můžeme říct, že ne zcela splňuje nárok ***vhodné pro začátečníky***, ale splňuje nárok na ***využitelnost***.

1.1.3 Unity

Posledním ukázkovým herním editorem bude velmi populární prostředí pro tvorbu her, a to **Unity** [12]. První verze Unity byla publikována v roce 2005 a jejím hlavním cílem bylo umožnit vývoj her širší komunitě [13]. Engine poskytuje velkou škálu funkcí pro tvorbu her s možností přidávat komunitou vyvíjené nástroje a rozšíření.

Stejně jako v předchozích editorech jsou herní objekty a jejich vlastnosti utvářeny přidáváním jednotlivých nastavitelných komponent. Ovládací prvky jednotlivých objektů i celkovou herní logiku pak vývojáři píší za pomoci frameworku napsaného v jazyce **C#**.



Obrázek 1.3 Komponenty herních prvků v **Unity**

V tomto herním enginu bylo (a stále je) vyvíjeno značné množství různorodých her od velkých herních titulů až po nejmenší **Indie hry** [14]. Pro uživatele nezkušené v programování může být používání tohoto editoru výzvou, jelikož, jak již bylo zmíněno, prostředí vyžaduje znalosti jazyka **C#** spolu se znalostí komplexního prostředí.

Stejně jako u **GameMakeru** je základní verze zdarma, a to i pro komerční využití, kde celkový roční výnos z vytvořených her nepřesahuje $100.000. Dále pak umožňuje zakoupení licencí, rozšiřujících funkcí prostředí a jiné poskytnuté služby [15].

Typ licence	**Roční cena**
PRO	$185.00
Industry	$450.00

Tabulka 1.3 Přehled cen licencí **Unity**

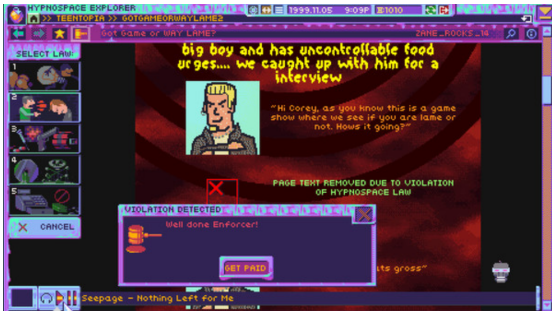
I přes velké množství výukových pomůcek a zdrojů [16] nesplňuje náš nárok ***vhodnost pro začátečníky***. Jeho ***využitelnost*** a ***rozšiřitelnost*** je na druhou stranu nezpochybnitelná.

1.2 Příklady her

Jelikož jsme v předchozí podkapitole rozebrali příklady konkrétních herních editorů, bylo by na místě podívat se i na hry, které v nich byly vytvořeny. Tyto hry z jednotlivých editorů pak můžeme porovnat podle jejich úspěšnosti na herním trhu.

1.2.1 Hypnospace Outlaw

Pravděpodobně jedna z nejúspěšnějších her vytvořených v herním editoru **Concept** nabízí hráči možnost stát se moderátorem alternativní verze internetu v roce 1999. Technologie v této alternativní realitě umožnuje lidem brouzdat internetem, zatímco spí. Náplní hry je řešení případů porušení různých nastavených pravidel pomocí pouze částečných informací a nápověd.

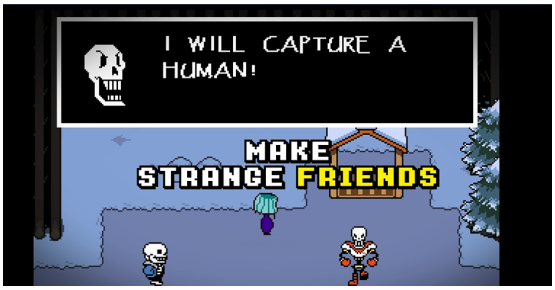


Obrázek 1.4 Ukázka ze hry **Hypnospace Outlaw**

Tato **2D** hra získala mnoho ocenění, včetně ceny za nejlepší design či umělecký styl v žánru **Indie her** v roce 2019 a na herním trhu Steam drží hodnocení devadesáti sedmi procent [17].

1.2.2 Undertale

2D hra vytvořená americkým vývojářem Tobym Foxem je jednou z nejlepších her vytvořených v herním editoru **GameMaker** [18]. Byla dokonce nominována na cenu za nejlepší *RPG* hru roku. Hráči nabídne prostředí alternativní země po válce mezi lidmi a monstry, kdy cílem hry je zachránit hlavní postavu a najít cestu z území monster.



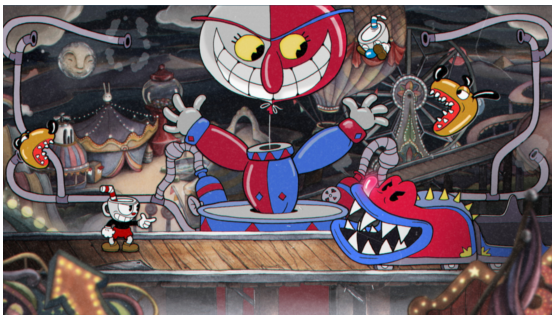
Obrázek 1.5 Ukázka ze hry **Undertale**

V průběhu hry může hráč interagovat s monstry a přijít na způsob, jak z nich udělat své přátele, nebo může naopak zvolit cestu násilí. Aktuální hodnocení této hry na platformě Steam je devadesát šest procent [19].

1.2.3 Cuphead

Jak vývojáři sami uvádějí v popisu své hry, tak **Cuphead** je: *Klasická akční střílečka, která klade velký důraz na boje s bossy. Grafika a ozvučení se inspirovaly kreslenými groteskami z 30. let 20. století a vznikly za pečlivého využití dobových technik, takže se můžete těšit na tradiční ručně kreslené animace, vodová pozadí a originální jazzové nahrávky* [20].

Jedná se o jednu z nejúspěšnějších her, která kdy byla vytvořena v editoru **Unity** [21].



Obrázek 1.6 Ukázka ze hry **Cuphead**

V roce 2017 byla oceněna za nejlepší umělecké provedení a jako nejlepší Indie hra. Na platformě Steam drží taktéž hodnocení devadesáti šesti procent [20].

2 Analýza

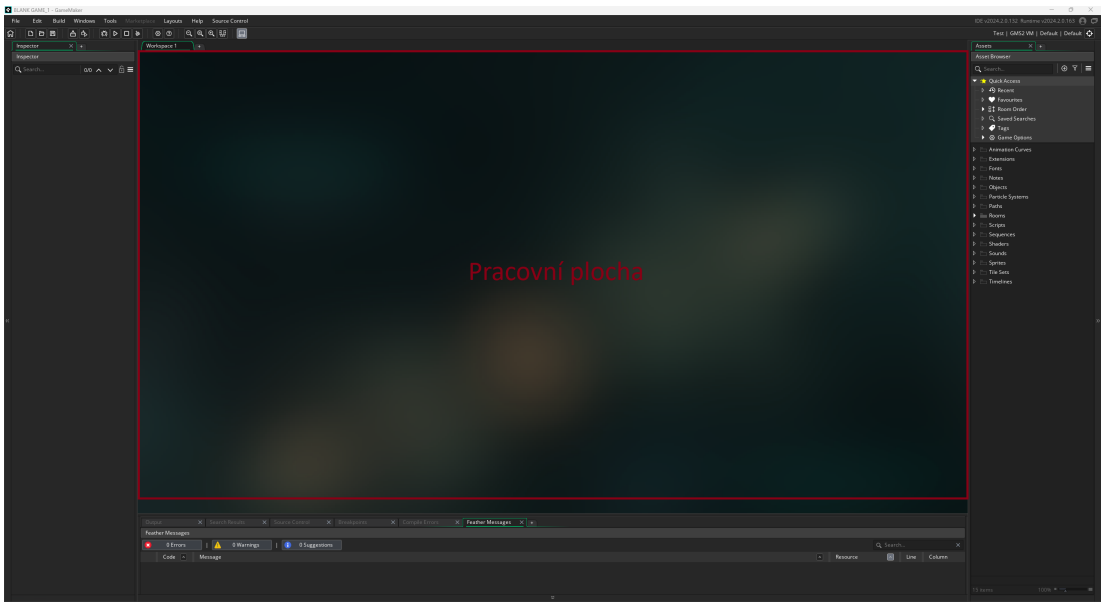
Jak již bylo zmíněno v úvodu, náplní této práce je vytvořit jednoduchý herní editor pro tvorbu platformových her. Analýza se tak nejprve zaměruje na to, jaké funkce by naše implementace měla nabízet. Dále se věnuje tomu, jaká specifika oproti jiným hrám obsahují hry platformové. K tomu nám jako inspirace poslouží nejprve již zmiňované konkurenční editory, a následně rozbor několika platformových her. Nejprve je ale nutné zaměrit se na to, jaké funkce chceme nebo musíme v našem editoru podporovat. Dále je pak potřeba vzít v potaz, jaké jsou nezbytné vlastnosti platformových her a umožnit jejich implementaci v našem projektu. K tomu použijeme uvedené příklady jednotlivých editorů, ve kterých se můžeme inspirovat.

2.1 Funkcionality editoru

Editory v předchozí sekci (1) jsme primárně rozlišovali podle rozsahu jejich funkcionality a komplexnosti. Nyní se naopak pokusíme zaměřit na jejich společné rysy a vybereme ty, které se budou hodit pro náš projekt. Začneme od nejzákladnějších pozorování:

Pracovní plocha

Stejně jako třeba textové editory, drží si i programy pro vývoj her určité standardní rozdělení plochy. Každý z těchto nástrojů má oddělené prostory pro nástroje a pro samotnou editační plochu. To můžeme pozorovat i na následující ukázce, kde uprostřed máme nadpisem označenou pracovní plochu.



Obrázek 2.1 Ukázka rozdělení panelů v **GameMakeru**

Nástroje a jednotlivé panely s nastavením jsou pak seskupeny po stranách editační plochy. Při každé možnosti jsou tyto komponenty dále seskupeny do

logických skupin tak, aby je bylo snadné najít a použít. Příkladem může být oddělení funkcí pro manipulaci s daty, jako je ukládání a načítání, a funkcí pro manipulaci s herními objekty v editační ploše. Tyto funkce jsou dalším společným rysem, který můžeme pozorovat.

Funkce pro manipulace s herními objekty

Je asi poměrně přirozené, že uživateli umožníme přidávat a odstraňovat objekty na editační ploše. Manipulace s objekty pouze za pomocí těchto dvou funkcí může být nepraktická v případech, kdy chceme jednu z těchto funkcí aplikovat na větší množství objektů najednou. Z tohoto důvodu je vhodné rozšířit tyto funkce na práci s vybranou plochou či skupinou objektů o funkci výběru.

Ukládání akcí uživatele

V průběhu používání editorů se můžeme ocitnout v situaci, kdy bychom se rádi vrátili do některého z předchozích stavů projektu. Toho jsme nepochybně schopni dosáhnout skrze již zmíněné funkce pro ukládání a načítání. To by však vyžadovalo, aby uživatel průběžně svoji práci ukládal, ideálně po nebo před každou změnou. Většina editorů tak navíc implementuje funkce pro odstranění a znovuprovedení uživatelských akcí. Jelikož chceme umožnit návrat do stavu projektu v konkrétní fázi, je nutné uložit stav po dokončení každé jednotlivé akce. Zapisování do souborů může být však časově náročné. Stejného efektu můžeme dosáhnout, pokud budeme ukládat pouze záznamy o akcích uživatele a k nim určené protiakce. Pokud tak například do projektu vložíme objekt na určitou pozici, editor uloží záznam o vytvoření objektu a jako protiakci zvolíme odstranění objektu, oboje na uživatelem určené pozici. Tyto záznamy tak tvoří řetězec akcí a protiakcí, pomocí kterého se v našem projektu můžeme pohybovat do minulých stavů a zpět. Pokud ale uživatel provede změnu ve stavu, do kterého se navrátil, tak akce, které zvrátil, již nenavazují, a proto je odstraníme a nahradíme novými.

Tvorba a upravování herních objektů

Manipulace s herními objekty vyžaduje v první řadě vytvoření prototypů herních objektů, na jejichž základě je budeme vytvářet. Tyto prototypy pak budou obsahovat informace o konkrétních vlastnostech objektů, které s jejich pomocí vytvoříme. Zde se nabízí možnost definovat množinu různých základních prototypů, které uživateli poskytneme. To však značně omezuje možnosti prostředí, jelikož ne vždycky musí námi vytvořené objekty postačovat, a to z různých důvodů. Jelikož chceme uživateli dát možnost vytvořit si hru podle svých představ, musíme mu také poskytnout možnost definovat vlastní prototypy herních objektů. S tím je pak spojená potřeba určit zmíněné vlastnosti těchto prototypů.

Komponenty a jejich výběr

Součástí tvorby *herních prototypů* je přiřazování a následné nastavování jejich vlastností. Tyto vlastosti můžeme kategorizovat do různých skupin, komponent nebo podle využití. Jak jsme mohli pozorovat ve vzorových editorech, můžeme při vytváření herních objektů takovéto komponenty, které jsou poskytnuty prostředím,

přidávat či ubírat podle požadovaných vlastností výledného objektu. Na to, jaké komponenty by náš editor měl poskytovat, se zaměříme v následující podkapitole (2.2.1).

Používání vlastních audiovizuálních zdrojů

Vytváření nových herních objektů s sebou přináší nový problém. Součástí tvorby nových objektů by nesporně mělo být i nastavení audio a vizuální části objektu. Pokud bychom umožnili výběr těchto vlastností pouze z předpřipravené množiny, narazili bychom na stejný problém, jaký jsme nastínili už při vytváření prvků. Tedy umožníme uživateli tyto prvky dynamicky nahrávat v průběhu tvorby z externích zdrojů, stejně jako to dělají námi pozorované editory (1). Je však zbytečné, abychom načítání prováděli při tvorbě každého objektu, jelikož některé tyto prvky může chtít uživatel použít u většího množství objektů. Z tohoto důvodu vytvoříme centrální systém spravující tyto prvky tak, abychom je mohli po načtení znovu použít při dalších operacích v editoru.

Funkce pro práci s daty

Při práci na projektech nemusíme být vždy schopni práci dokončit najednou. Z tohoto důvodu je vhodné uživateli umožnit svoji práci uložit a případně zpětně načíst. Načítání souborů pak vyžaduje možnost přístupu ke konkrétním souborům zvolených uživatelem. V návaznosti se pak zdá žádoucí dát uživateli možnost při ukládání vybrat jméno pro soubor s daty, aby je pak byl schopen snadno identifikovat. Pokud tuto naši myšlenku zkusíme nalézt v pozorovaných editorech, zjistíme, že tyto funkce nejen implementují, ale například disponují i možností rychlého uložení dat za použití předdefinovaného jména a čísla pořadí. Dále pak při těchto operacích umožňují procházet souborový systém uživatele a dávají mu tak možnost vybrat si, odkud data načte nebo kam je uloží.

Pohyb po pracovní ploše

Součástí práce v editoru je umisťování objektů na uživatelem zvolenou pozici. Pokud bychom uživateli omezili pracovní plochu projektu na pevně stanovenou velikost, omezili bychom také možnost pro přesné umístění. Proto uživateli umožníme pohybovat se po pracovní ploše tak, jako kdyby okno pracovní plochy bylo pouze průzorem hledícím na větší plochu před ní.

Režim ladění

Jednotlivé herní prvky a jejich interakce se nemusí vždy chovat předvídatelně. Například, když vytvoříme herní mapu pro skákací hru, nemusí nám být při tvorbě jednoznačně jasné, zda bude hráč schopen tuto mapu projít. Některé platformy mohou být například pro skok moc vzdálené, nebo někteří nepřátelé až příliš nároční. Proto by se nám v našem editoru hodila možnost si hru otestovat v průběhu vývoje, aniž bychom byli nuceni ji opustit. Z tohoto důvodu, stejně jako v ostatních editorech, budeme podporovat režim ladění, ve kterém budeme testování provádět. Součástí tohoto režimu bude možnost spusit hru v aktuálním stavu projektu a interagovat s jejím prostředím tak, jako by se jednalo o spuštěnou

hru. Jelikož je ale cílem této funkcionality testovat náš projekt, přidáme možnost pro pozastavení hry, abychom mohli například vyhodnotit aktuální stav hry. Po ukončení režimu ladění se pak všechny herní prvky musí uvést do stavu, ve kterém byly před spuštěním.

2.2 Analýza komponent platformových her

Funkce editoru jsme si již uvedli. Nyní je třeba zjistit, jaké nároky na náš editor bude mít tvorba platformových her. V této části se zaměříme na společné rysy platformových her. U jejich analýzy budeme vycházet z následující definice:

Definice 1. *Platformové hry neboli plošinovky, je žánr her takových, kde cílem hráče je dostat se z jednoho místa na druhé, většinou za pomoci skákání či šplhání, a to někdy i navzdory překážkám či nepřátelům [22], [23].*

Tato definice nám sama o sobě poskytuje náznak toho, jaké komponenty budeme potřebovat.

2.2.1 Audiovizuální efekty

- Jednotlivé objekty by měly být vizuálně odlišitelné. Pokud bychom měli všechny objekty stejné, nebylo by možné kupříkladu odlišit hráče od nepřítele nebo překážky. Musíme tedy u herních objektů umožnit nastavení jejich textury.

- Jelikož hry většinou obsahují dynamické prvky, jako například pohybujícího se hráče, bylo by vhodné uživateli umožnit tuto dynamiku vizuálně vyjádřit. V případě pohybu by mohlo být například žádoucí, aby postava pohybovala nohama. Z tohoto důvodu umožníme u našich objektů nastavit nejen textutu, ale i animaci. Animací pak chápeme sérii textur s definovaným časem, který určuje, jak dlouho bude textura zobrazena, než bude nahrazena další.

- Vizuální stránka ale není jediná forma komunikace s hráčem, kterou ve hrách běžně nacházíme. Pokud například chceme zdůraznit kolizi dvou herních prvků, můžeme to udělat pouze výrazným zvukem. Budeme tedy při tvorbě her podporovat i přehrávání zvuků. S tím přichází potřeba nastavit vlastnosti jednotlivých zvukových stop, jako například jejich hlasitost. Zvukové efekty pak budeme přiřazovat k jednotlivým akcím či událostem.

2.2.2 Pohyb a kolize

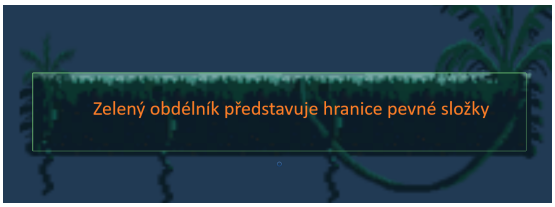
- Jak jsme mohli nahlédnout z definice (2.2), součástí skákacích her je přesun objektů z jedné lokace na druhou. To znamená, že uživateli umožníme k vytvářeným objektům přidat a nastavit pohybové akce, jako je například chůze, skok či létání.

- Pohyb objektu je reprezentován posunutím jeho pozice o jednotku vzdálenosti za čas ve směru pohybu. Jeho rychlost pak bude vyjádřena přenásobením směrového vektoru. To ale nemusí vždy stačit. Například můžeme mít dvě oddělené platformy a hráče stojícího na jedné z nich. Pokud budeme skok

z jedné platformy na druhou reprezentovat pouze tímto způsobem, bude se objekt pohybovat pouze po přímce směrového vektoru. [Obrázek]. Aby skok měl tvar křivky, musíme při pohybu vzít v úvahu gravitační zrychlení a hmotnost objektu. Gravitaci objektů využijeme i v jiných případech, třeba pro simulaci pádů.

Pokud hráč pohyb přeruší, přestane kupříkladu držet patřičnou klávesu, posun objektu se zastaví. To by ale mohlo vypadat poněkud nepřirozeně, jelikož by objekt okamžitě zastavil. Z tohoto důvodu přidáme i simulaci tření, které objekt po ukončení pohybu zpomalí lineárně.

- Spolu s dynamicky se pohybujícími objekty přichází i problém ošetření kolizí dvou prvků. Abychom byli schopni zjistit, že se dva objekty srazily, musíme nejprve znát jejich hranice. Srážku jsme pak schopni detekovat v případě, že se hranice dvou objektů protínají. Tyto hranice jsou uzavřeným tvarem, představujícím pevnou složku objektu, která nemusí velikostí či tvarem nutně odpovídat textuře objektu. Složitost počítání průniku dvou objektů je pak odvozena od tvaru hranice. Pokud například zvolíme obdélníkové hranice, bude výpočet průniku jednodušší, než pro mnohoúhelník. Naproti tomu mnohoúhelník by reprezentoval vizuální hranice objektu lépe, než zmíněný obdélník.



Obrázek 2.2 Ukázka pevné složky

Zvolit složitější tvary hranic může být žádoucí například v případech, kdy chceme lépe reprezentovat fyzikání vlastnosti objektů. Příkladem může být objekt představující kuličku. Jelikož se nacházíme ve dvorozměrném prostoru, nastavíme této kuličce kruhovitou hranici, abychom umožnili její kutálení.

- Pro některé akce by bylo vhodné mít možnost definovat podmínky, za kterých mohou být provedeny. Kupříkladu pokud se postava hráče nenachází nad pevným objektem, tedy nestojí na platformě, nemusíme vždy chtít, aby měl možnost učinit akci skoku.

2.2.3 Speciální objekty

- Jak vyplývá z definice (2.2), součástí platformových her mohou být i různé speciální objekty, jako například zmíněné pasti. Tyto objekty pak budou reagovat na kolizi s hranicemi objektů. Například pokud se postava hráče dotkne objektu pasti, můžeme jí odebrat část jejího zdraví. Tyto akce však nemusíme chtít uplatnit na všechny objekty, které mohou s pastí přijít do kontaktu. Proto k nastavení těchto akcí přidáme možnost určení množiny prvků, na které je budeme chtít při kolizi aplikovat.

- Příklady akcí, které by tyto objekty mohly podporovat, jsou například:

Akce	**Příklad využití**
Změna textury	Vyjádření změny stavu
Přehrání audiostop	Zvuk při kolizi
Nahrání další úrovně	Dosažení cílové pozice levelu
Sebedestrukce	Simulace praskajících platforem
Změna pozice	Teleportace hráče
Změna osvětlení	Navození hororové atmosféry

Tabulka 2.1 Příklady akcí a jejich využití

2.2.4 Nepřátelé

Překážky a jiné speciální objekty nejsou jediné prvky herního levelu, které mohou jeho průchod zkomplikovat. Nepřátelé jsou herní objekty s množinou přiřazených akcí, které nejsou řízené uživatelem, nýbrž předdefinovaným chováním, které se spustí v reakci na určené podněty. Například pokud se hráč přiblíží do určité vzdálenosti od hypotetického objektu, spustí se akce pro útok. Při tvorbě objektů tedy umožníme přidání komponenty, která bude tento objekt řídit. Součástí nastavení této komponenty bude i přiřazení pohybových akcí, které tento objekt může provést.

2.2.5 Hráč

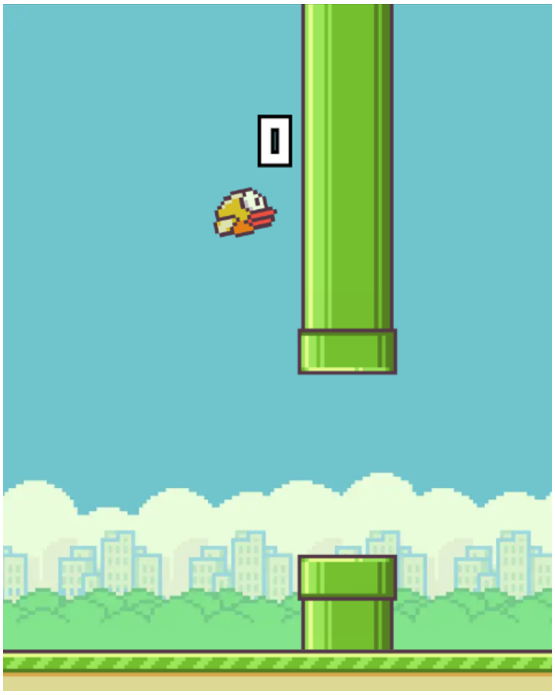
Stejně jako u objektů nepřítele, objekt hráče vyžaduje přiřazení jeho pohybových akcí. Narozdíl však od objektů nepřátel, které řídí předdefinované chování, objekt hráče musí být schopen reagovat na podněty uživatele. Tudíž přidáme komponentu pro hráčský objekt, kde při přidávání pohybových a jiných akcí umožníme uživateli zvolit kombinace kláves, při jejichž stisknutí budou provedeny.

2.3 Rozbor vzorových her

V předchozí části jsme se soustředili na platformové hry obecně. Nyní k naší analýze použijeme vzorové hry, jejichž adaptace budeme později implementovat, tedy **FlappyBird**, **JumpKing** a **Mario**.

2.3.1 FlappyBird

Původně pouze mobilní hra z roku 2013 vytvořená programátorem Dong Nguyenem, která se již na konci roku 2014 stala nejvíce stahovanou bezplatnou hrou na telefonech s operačním systémem **iOS** [24].



Obrázek 2.3 Ukázka ze hry **Flappybird** [25]

Cílem této hry je navigovat hráčem řízeného ptáčka Fabyho skrze serii překážek, co nejdéle je v možnostech hráče. Hráč přitom může pouze ovlivnit pohyb směrem vzhůru, hra samotná pak kontinuálně posouvá ptáčka směrem doprava. Pokud se ptáček dotkne některé z překážek, hra končí. Pro vytvoření této hry musíme být tedy schopni detekovat kolizi s překážkami a podporovat výše zmíněné pohybové akce.

Komponenta	**Výskyt ve hře**
Hráč	Je ovládán pouze pomocí kliknutí
Vizuální efekty	Animovaná textura hráče Fixní překážky
Audio efekty	Kolize s překážkou Mávnutí křídel
Pohyb	Kontinuální let vpravo Impulz směrem vzhůru
Kolize	Sloupy a podlaha levelu
Speciální prvky	———

Tabulka 2.2 Výčet pozorovaných herních komponent hry **Flappy Bird**

2.3.2 Jumpking

Hra vytvořená v roce 2019 programátorem pod přezdívkou Nexile, která se stala kvůli své obtížnosti populární na streamovací platformě **Twitch** [26]. Hráč v této hře kontroluje postavu rytíře, snažícího se zachránit princeznu na vrcholu vysoké věže.

Obrázek 2.4 Ukázka ze hry **Jumpking** [27]

Aby se hráč dostal na vrchol, musí zdolat množství od sebe vzdálených plošin a nerovných povrchů. Rytíř se přitom může pohybovat pomocí chůze a skoků s proměnlivou intezitou. Síla skoku se pak odvíjí od délky stisku klávesy určené pro akci skoku. Pokud se hráči nepodaří správně určit intenzitu skoku a platformu mine, propadne se o část zpět, někdy až na začátek.

Pro tuto hru budeme muset opět umět detekovat kolize mezi prvky a navíc podporovat zmíněné pohybové akce. Oproti předchozí hře zde navíc máme pozadí s pohyblivými prvky, jako jsou ptáci či mlha.

Komponenta	Výskyt ve hře
Hráč	Chůze pomocí směrových šipek Mezerníkem je ovládán skok
Textury	Animovaná textura hráče Fixní překážky
Audio efekty	Interakce s prostředím
Vizuální efekty	Animace postavy při pohybu
Pohyb	Horizontální chůze Skok pro vertikální pohyb
Kolize	S prostředím
Speciální prvky	Žádné

Tabulka 2.3 Výčet pozorovaných herních komponent hry **Jump King**

2.3.3 Super Mario Bros.

Hra z roku 1985 vydaná společností Nintendo, kde hráč přebírá kontrolu nad postavou instalatéra Maria při cestě za záchranou princezny Peach z houbového království [28].

Obrázek 2.5 Ukázka ze hry **Super Mario Bros**. [29]

Během hraní musí hráč překonat mnoho překážek a nepřátel, aby se dostal k červené vlajce a dokončil tak herní level. Součástí jednotlivých herních levelů

jsou i speciální objekty, které hráči zvedají skóre či uvedou Maria do vylepšeného módu. Jednotlivé herní objekty se pohybují chůzí, skoky a případně letem.

Abychom byli schopni tuto hru implementovat, musíme opět splnit všechny nároky, jako u předchozích her. Navíc musíme být schopni vytvářet nepřátele, speciální objekty a jejich akce. Dále pak tato hra umožňuje přítomnost druhého hráče. Toho můžeme vytvořit jako druhý objekt, kterému přiřadíme jiné klávesy k jednotlivým akcím.

Komponenta	**Výskyt ve hře**
Hráč	Směrové šipky či W,A,S,D Mezerník pro skok
Textury	Animované textury Fixní překážky Pozadí
Audio efekty	Hudba Interakce s prostředím Interakce s nepřáteli Skok
Vizuální efekty	Pohyb Speciální objekty Interakce s objekty Smrt
Pohyb	Chůze do stran Skoky Chůze
Kolize	S objekty S nepřáteli
Speciální prvky	Bonusové mince Magické houby Hvězda dávající nezranitelnost

Tabulka 2.4 Výčet pozorovaných herních komponent hry **Super Mario**

2.4 Analýza akcí

Jak již bylo zmíněno (2.2.2), u speciálních objektů, hráčů a nepřátel umožníme uživateli přidat různé akce reagující na určené podněty. Nyní se na základě předchozích rozborů pokusíme analyzovat, jak tyto akce implementovat v našem editoru.

2.4.1 Přiřazování akcí

Stejně tak jako v případě vytváření objektů (2.1) by nebylo u přiřazování akcí k objektům vhodné omezit tyto akce pouze na námi vybranou předdefinovanou množinu. Opět bychom tím totiž omezili množství akcí, které bychom byli schopni u vytvářených objektů definovat. Zároveň by u některých akcí mohlo být vhodné mít možnost tyto akce podmínit nejen spouštěcí událostí, ale třeba také aktuálním stavem objektu. U jiných funkcí by se nám zase mohlo hodit je spustit několikrát za sebou. Z těchto důvodů by se nám hodilo vytvořit si jednoduchý programovací jazyk, s jehož pomocí by uživatelé, kterým nedostačuje množina základních funkcí, mohli tyto akce spojit do větších celků.

Součástí tohoto programovacího jazyka by byla tvorba podmínek, cyklů, základních proměnných a volání našich funkcí.

2.4.2 Podmíněné bloky

98,jPři nastavování akcí se uživatel může dostat do situace, kdy by se na základě stavu objektu či hry hodilo provést jinou akci než tu, která je přiřazena k události. Příkladem může být například střet postavy hráče s pastí. Pokud má postava dostatek bodů na přežití střetu, odebereme část těchto bodů a hra pokračuje. Pokud by ale měla být hodnota zdraví po střetu záporná, hru ukončíme, jelikož hráč zemřel. Pokud bychom se tento problém pokusili vyřešit definováním dvou akcí reagujících na stejný spouštěč, docílíli bychom pouze toho, že by se tyto akce při vyvolání události provedly obě. Potřebujeme tedy určit, jakou akci spustit v závislosti na úrovni zdraví hráče. K tomuto účelu umožníme v našem jazyce definovat podmíněné úseky kódu.

Abychom se při tvorbě podmíněných bloků pro náš jazyk vyhnuli problému *Dangling Else* (5.0.4), budeme podmínku a s ní související blok kódu ohraničovat klíčovými slovy **if** a **fi**.

Program 1 Ukázka konstrukce podmíněného bloku

```
1    if podminka
2        #Podmineny blok kodu
3    fi
```

Jak můžeme vidět na našem příkladu se zdravím, kód pro ukončení hry se spustí pouze v jednom případě. Za jiných okolností pouze ubereme část bodů. Proto by se nám hodila možnost definovat blok kódu, který se provede pokaždé, když není splněná podmínka. Tento blok nazveme **Else**, stejně jako v jiných programovacích jazycích.

Program 2 Ukázka podmíněného bloku s konstrukcí **Else**

```
1    if podminka
2        #Splneni podminky
3    else
4        #Nesplneni
5    fi
```

2.4.3 Cykly

Některé události mohou vyžadovat několikanásobné aplikování jejich přiřazené akce. Pokud se například hráč dostane do kontaktu se speciálním objektem pro doplňování bodů zdraví, můžeme zdraví navýšit tolikrát, kolik má hráč nastřádaných bodů skóre. Každá akce přidání života ubere část skóre. Takovouto akci by bez umožnění podmíněného cyklení nebylo možné vytvořit.

Podmíněné cykly v našem programovacím jazyce umožníme implementací cyklu **While**. Klíčová slova, kterými jej a blok kódu s ním související budeme označovat, jsou **while** a **end**. Cyklus se pak bude provádět do doby, kdy přestane platit podmínka. V našem jazyce tedy budeme muset ošetřit případ, kdy uživatel vytvoří podmínku, která nebude nikdy neplatná. V takovém případě by se totiž náš kód dostal do nekonečné smyčky.

Program 3 Ukázka konstrucke cyklu **while**

```
1    while podminka
2        #Blok smycky
3    end
```

2.4.4 Proměnné

V našem jazyce budeme dále umožňovat vytváření proměnných. Pro vytvoření proměnné pak budeme vyžadovat typ hodnoty, její jméno a hodnotu. U těchto hodnot pak budeme vyžadovat, aby odpovídaly typu proměnné. Jakmile už proměnou jednou definujeme, nebudeme při dalším použití vyžadovat uvádění jejího typu. Nyní si uvedeme jaké typy budeme podporovat.

- Jak již z názvu vyplývá, podmíněné bloky a cykly jsou provedeny pouze tehdy, pokud je splněna určená podmínka. Takovéto podmínky pak chápeme jako logické hodnoty spojené příslušnými operátory do větších celků. Tyto hodnoty, které budeme označovat klíčovým slovem **bool**, pak umožníme získat buďto z logických operátorů pro porovnávání, anebo přímou definicí. Proměnné typu bool pak budeme moci nastavit buďto hodnotou **true** pro pravdu nebo **false** pro nepravdu.

Program 4 Ukázka přiřazení logického výrazu do proměnné **bool**

```
1    bool jmeno = true ^ 1 < 2 || false
```

- Dalším typem proměnné, kterou budeme v našem jazyce podporovat, je číslo. Čísla se nám budou hodit například v případech, kdy budeme chtít cyklus provést přesně *n-krát*. Vytvoříme si tedy hodnotu určující počet vykonaných cyklů, kterou před provedením každého cyklu porovnáme s hodnotou **n**. Typ těchto proměnných budeme označovat klíčovým slovem **num** a jejich základní hodnoty budou celá či desetinná čísla.

Program 5 Ukázka přiřazení aritmetického výrazu do proměnné **num**

```
1       num jmeno = 1.234 + 2 * 7
```

- Posledním typem podporovaných proměnných bude text, který nám poslouží například k porovnávání jmen. Tuto proměnnou budeme označovat klíčovým slovem **string** a její hodnoty budou jakékoliv znaky, ohraničené znaky uvozovek.

Program 6 Ukázka přiřazení textového výrazu do proměnné **string**

```
1       string jmeno = "Hello" + " " + "World"
```

2.4.5 Definované akce

Všechny tyto konstrukce nám pouze umožňují ovlivnit, jak a kdy se provedou námi definované akce. V této části se pokusíme určit, na základě předchozí analýzy, jaké funkce bychom měli podporovat.

Pohybové akce

U vzorových her jsme mohli pozorovat, jaké pohybové akce implementují. Těmito akcemi byly chůze, skok, skok s proměnlivou intenzitou a let. Tyto akce se pak pokusíme rozdělit do jednotlivých podakcí, u kterých bychom byli schopni určit směrový vektor.

- Pro chůzi máme jednoduché rozdělení na pohyb vlevo a vpravo. Rychlost pohybu pak určíme tak, jak jsme popsali zde (2.2.2).
- U skoku už bude rozdělení poněkud komplikovanější. Základní směry jsou stejné, jako pro chůzi, ale jelikož chceme uživateli umožnit nastavit tvar křivky a ovlivnit tak sílu skoku, musíme do nastavení této akce přidat intenzity horizontálního a vertikálního pohybu. Směrový vektor skoku pak získáme součtem vektoru základního směru přenásobeného intenzitou horizontálního pohybu a vektoru směřujícího vzhůru přenásobeného vertikální intenzitou.
- Skok s proměnlivou intenzitou pak budeme implementovat stejně jako běžný skok, až na určení intenzit jednotlivých pohybů. U nich si určíme minimální a maximální hodnotu. Dále pak přidáme maximální čas, po který budeme skok nabíjet. Při začátku nabíjení nastavíme intenzity na minimální hodnoty. S roustoucí dobou nabíjení porostou hodnoty až ke svým maximům.
- Poslední podporovanou akcí bude let. Let budeme implementovat jako volný pohyb v prostoru, při jehož nastavování určíme pouze rychlost pohybu. Jednotlivé směrové vektory, tedy podakce, pak určíme jako vektory v rozestupu 45 stupňů počínaje nulovým úhlem.

Akce objektů

Cílem tohoto druhu akcí bude upravovat jednotlivé komponenty a vlastnosti herních objektů (2.2.1). Příkladem takovýchto funkcí může být nastavení velikosti pevné složky objektu, změna textury nebo animace. Dále pak implementujeme funkce pro získání důležitých hodnot, jako je například pozice nebo směr pohybu.

Akce prostředí

Podobně jako funkce nastavující herní objekty, funkce v této kategorii budou nastavovat vlastnosti prostředí hry. Jako příklad můžeme uvést změnu textury pozadí nebo přehrávané audio stopy.

2.5 Stanovení cíle práce

Na základě předchozích kapitol jsme získali představu o vlastnostech, které by náš editor měl splňovat. V rámci implementace tedy vytvoříme herní editor podle těchto kritérií (1).

Jelikož pro hry vytvořené v tomto editoru neexistuje prostředí, ve kterém by je bylo možné hrát, bude součástí tohoto projektu i jeho vytvoření. Tento editor zapracujeme do projektu jako součást hry. Styl editoru tak pojmeme více v herním stylu, než je pro takovýto nástroj běžné.

Dále pak implementujeme pro náš editor vlastní programovací jazyk umožňující sestavování předdefinovaných akcí do spustitelných bloků. Popis tohoto jazyka můžeme najít zde (2.4.1)

Na závěr se pak pokusíme vytvořit adaptace vybraných vzorových her (2.3).

2.6 Plán implementace

Pro vývoj toho projektu použijeme již zmíněné prostředí pro tvorbu her **Unity**(1.1.3). Toto prostředí nám totiž postkyne následující výhody [30]:

- Pokud výdělek z tvorby v tomto prostředí nepřesahuje sto tisíc dolarů ročně, je běžná verze prostředí zdarma. To tedy umožňuje širší veřejnosti implementovat případná rozšíření tohoto projektu, a to bez nároků na finanční stránku.
- **Unity** má rozsáhlou kominutu programátorů, čítající okolo dvou a půl miliónu vývojářů. Tato komunita pak může být díky oficiálním diskuzním fórům nápomocná nejen při řešení našich problémů týkajících se vývoje, ale i pro již zmíněné potencionální vývojáře pracující na rozšířeních.
- Další užitečnou vlastností je podpora velkého množství operačních systémů, jako jsou například **Windows**, **Android** nebo **iOS**. To rozšíří potencionální pole působnosti pro náš editor.
- Vývoj nám dále usnadní **Unity Asset Store**, ve kterém můžeme nalézt velké množství bezplatných rozšíření a balíčků s texturami. Tyto položky převážně vytváří zmíněná komunita a většinou je možné je nalézt v placené, či bezplatné verzi.

- Jelikož **Unity** je samo o sobě herním editorem, poskytuje velké *API* implementující některé ze zmíněných prvků našeho editoru či herních objektů. To nám tak například usnadní práci s fyzikou objektů.

Nyní si rozvrhneme, po jakých krocích během vývoje budeme postupovat. Plán zapíšeme zjednodušenými body:

- Rozvrhneme a následně implementujeme základní vzhled editoru a panelů nástrojů (2.1).
- Vytvoříme sadu základních testovácích objektů.
- Implementujeme základní funkce pro manipulaci s herními objekty (2.1), které otestujeme na již vytvořených základních objektech.
- Otestujeme funkce pro manipulaci a přidáme ty pro práci s daty (2.1)
- Přidáme možnost vytváření objektů (2.1) a funkce s tím spojené, tedy například úpravu či mazání již existujících objektů. Součástí přidání těchto funkcí bude i vytvoření sady komponent, které budeme moci těmto objektům nastavit. Dále pak upravíme vzhled editoru tak, aby umožnil zobrazovat seznam existujících objektů.
- Vytvoříme a otestujeme prostředí pro režim ladění.
- Implementujeme prostředí pro hraní vytvořených her.
- Editor a prostředí pro hry spojíme do jednoho celku.
- Pokusíme se vytvořit vzorové hry.

3 Uživatelská dokumentace

V průběhu této kapitoly se zaměříme na popis používání jednotlivých částí našeho projektu. K vytvoření složitějších grafických komponent, jako jsou tlačítka nebo textury základních prototypů, byly použity volně dostupné balíčky z **Unity Asset Store** [31] [32]. Z tohoto zdoje také pochází panel pro procházení souborů na disku [33] a základní pozadí projektu [34].

Následující části projektu jsou vytvořeny pro práci v rozlišení **Full HD**, tedy **1920x1080**.

3.1 Úvodní menu

Menu načtené po spuštění programu, které dává uživateli na výběr mezi spuštěním editoru nebo hraním již vytvořené hry.

Obrázek 3.1 Ukázka úvodního menu

- Tlačítko **Play** zobrazí průzkumníka souborů, jehož pomocí uživatel určí hru, kterou si přeje hrát. Pokud načtění proběhne v pořádku, zobrazí se plocha herního prostředí. V opačném případě je vypsána chybová hláška do panelu ve spodní části ukázky.
- Stiskem **Map editor** se načte herní editor, ve kterém můžeme vytvářet jednotlivé hry.
- Celou hru pak můžeme ukončit pomocí tlačítka se symbolem křížku v pravém horním rohu.

3.2 Herní prostředí

Po úspěšném načtení konkrétní hry v hlavním menu, je v tomto prostředí zobrazena vybraná hra i pozadí. V horní části obrazovky se pak nachazí textové pole, do kterého můžeme pomocí definovaných akcí vypisovat námi zvolené hlášky. Těmi tak můžeme například podávat informace o změně aktuálního skóre hráče.

Při stisku klávesy **Escape** se hra pozastaví a je zobrazeno menu, které můžeme vidět na ukázce. Dále si rozebereme funkce jeho jednotlivých prvků.

Obrázek 3.2 Ukázka menu v herním prostředí

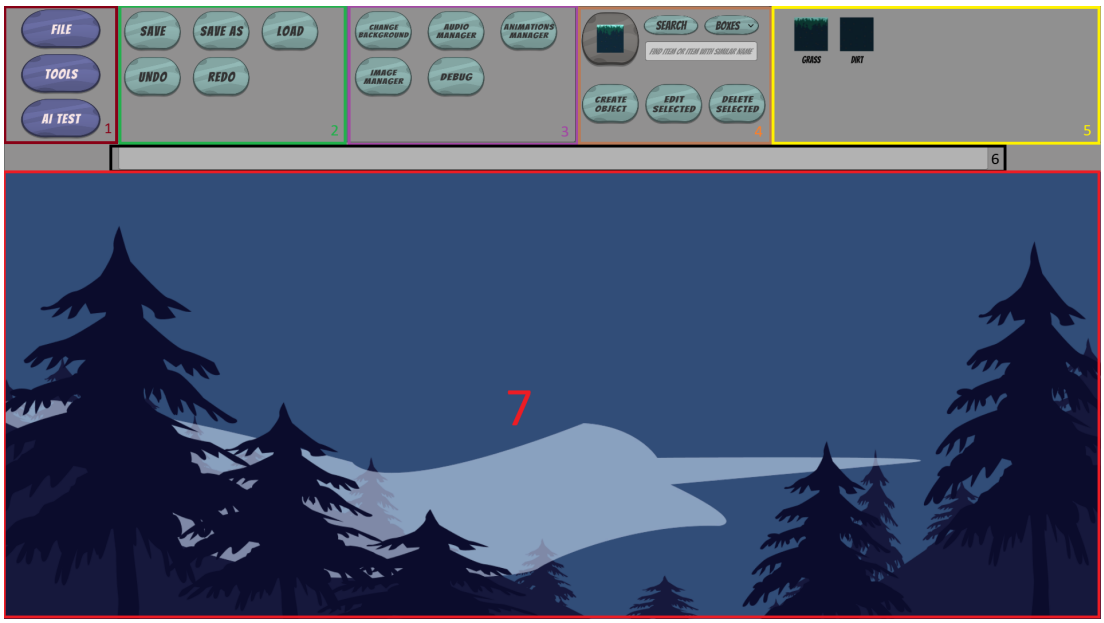
- **Resume** - zruší zobrazení tohoto menu a znovu spustí hru;
- **Restart** - načte hru do úvodního stavu;
- **Exit** - ukončí aktuálně hranou hru a vrátí nás do úvodního menu;
- **Volume** - tímto posuvníkem můžeme nastavit celkovou hlasitost herního prostředí.

3.3 Editor

Tato část obsahuje popis jednotlivých částí herního editoru a jejich použití.

3.3.1 Hlavní panel editoru

Začneme popisem základního rozdělení našeho editoru podle následujícího obrázku. K odlišení jednotlivých sekcí používáme barevně odlišené a očíslované rámečky, na které se budeme odkazovat.



Obrázek 3.3 Ukázka plochy editoru

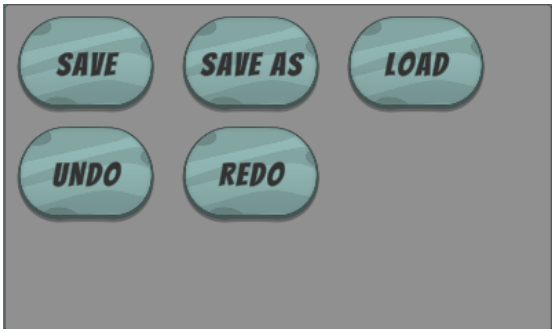
V červeném rámečku, označeném číslem sedm, je ohraničená pracovní plocha editoru (2.1). Ve zbylé části obrázku se nachází hlavní panel nástrojů, opět rozdělený do jednotlivých sekcí:

- Tmavě červený panel, označený číslem jedna, obsahuje tlačítka pro výběr jednotlivých panelů nástrojů označených v zeleném rámečku s číslem dva. Vybrat pak můžeme z následujících tří skupin: funkce pro práce s daty, manipulaci s herními objekty a pro testování.
- Panel s číslem tři obsahuje tlačítka pro zobrazení jednotlivých manažerů *assetů*, nastavení pozadí a spuštění módu ladění.
- Číslem čtyři je označený panel pro práci s prototypy a jejich vyhledávání. Zároveň obsahuje tlačítko pro výběr zobrazované skupiny protypů. Vybraná skupina se pak zobrazuje na panelu označeném číslem pět.
- Poslední panel, označený číslem šest, slouží jako výstupová konzole pro zachycené chybové hlášky (4.5).

3.3.2 Panely nástrojů

Nyní se zaměříme na konkrétní skupiny nástrojů a rozebereme si funkce jednotlivých tlačítek.

Funkce pro práci s daty

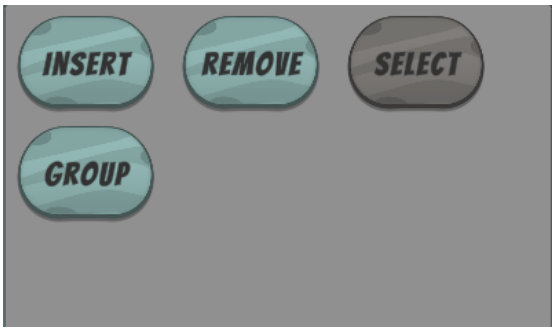


Obrázek 3.4 Ukázka panelu funkcí pro práci s daty

- **Save** - slouží k rychlému uložení projektu. Ten se uloží do složky Maps pod vygenerovaným jménem (2.1);
- **Save as** - umožňuje uložení projektu pod jménem, které si uživatel zvolí;
- **Load** - umožňuje načtení vybraného souboru, neuložený projekt bude ztracen;
- **Undo** - odstraní akci provedenou uživatelem;
- **Redo** - znovu provede odstraněnou akci, pokud taková je. Pokud uživatel po odstranění provedl novou akci, akci již nelze obnovit.

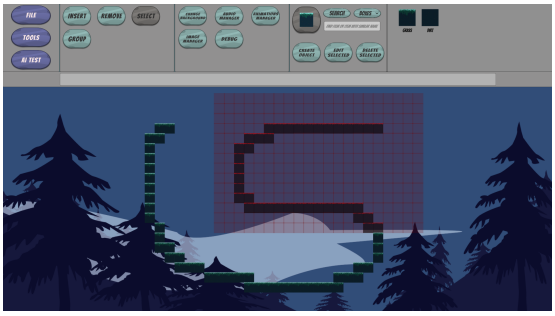
Funkce pro manipulaci s objekty

Ze skupiny následujících funkcí, můžeme vždy vybrat pouze jednu aktivní funkci, se kterou budeme pracovat. Tyto funce pak aplikujeme stisknutím levého tlačítka myši na pracovní ploše (2.1). Funkce se aplikuje do doby, kdy uživatel přestane držet zmíněné tlačítko.



Obrázek 3.5 Ukázka vybrané funkce **Select**

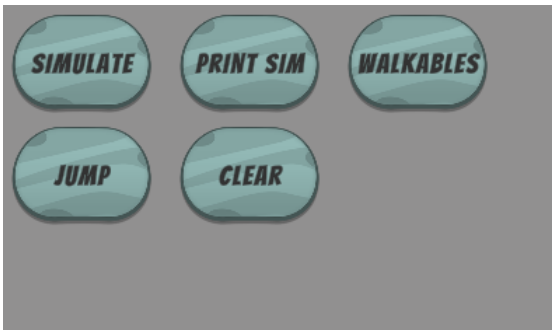
- **Insert** - Podle aktuálně zvoleného prototypu vytváří nové herní objekty na pracovní ploše. Pokud funkci aplikujeme na pozici vybranou funkcí **Select**, tak se akce aplikuje na všechny vybrané pozice.
- **Remove** - Odstraní existující herní objekt na vybrané pozici. Stejně jako u funkce pro vytváření objektů, pokud vybraná pozice je součástí pozic vybraných funkcí **Select**, je tato funkce aplikovaná na všechny objekty v tomto výběru.
- **Select** - Funkce pro výběr pozic a objektů na pracovní ploše. Navíc podporuje výběr čtvercové oblasti, pokud při stisku levého tlačítka myši navíc držíme klávesu **Shift**.
- **Group** - Označí všechny objekty vytvořené ze stejného prototypu, jako objekt na pozici stisku.
- **Posun** - Pokud uživatel nevybere žádnou z uvedených funkcí, je vybrána funkce pro posun pracovní plochy. Při stisknutí a následném posunutí kurzoru posune plochu ve směru pohybu. Pokud je pozice stisku zahrnuta ve výběru funkce **Select**, slouží k posunu objektů v této skupině.



Obrázek 3.6 Ukázka aplikování čtvercového výběru

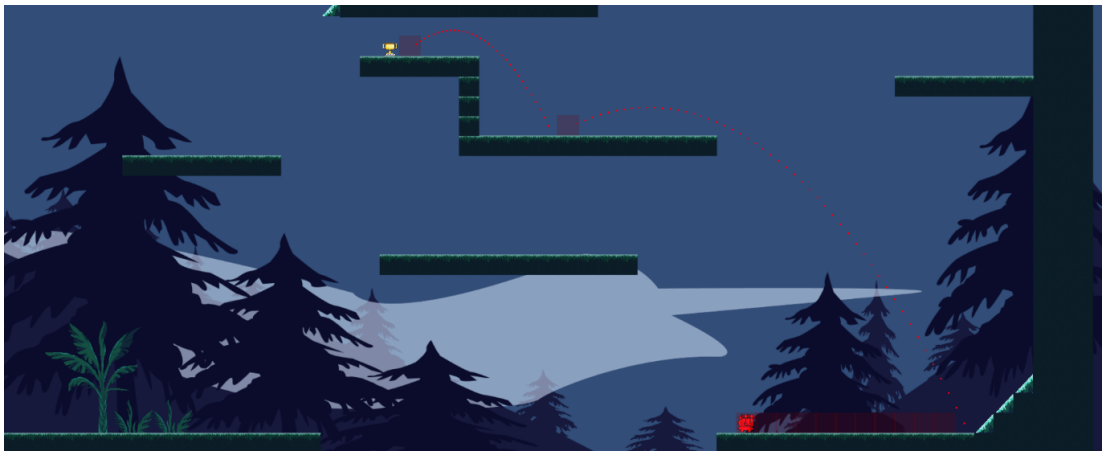
Funkce pro testování

Následující funkce, vyjma funkce **Walkables**, pracují s objekty obsahující komponentu hráče nebo nepřítele. Z nich získají jim přiřazené pohybové akce a následně simulují jejich průběh. S tím přichází i jistá míra nepřesnosti výpočtu a tedy tyto funkce slouží spíše pro získání rámcového přehledu o možnostech testovaného objektu. Ten vybereme pomocí funkce **Select**.



Obrázek 3.7 Panel funkcí pro testování

- **Simulate** - Pokusí se najít cestu z pozice vybraného prvku na pozici speciálního objektu cíle se jménem **Finish**, pomocí pohybových akcí přiřazených k tomuto objektu. Pokud takováto cesta existuje, provede simulaci sekvence pohybů použitých k jejímu nalezení.
- **Print Sim** - Funguje stejně jako funkce **Simulate** s výjimkou provedení simulace. Namísto toho označí jednotlivé pozice, které objekt na této cestě navštíví.



Obrázek 3.8 Ukázka funkce simulace

- **Walkables** - Označí všechny pozice nacházející se nad objekty obsahující pevnou část. Jedná se pouze o orientační přehled pozic, na kterých může pohyblivý objekt stát.
- **Jump** - Pokud označený objekt disponuje pohybovou akcí skoku, vykreslí všechny jeho možné trajektorie.
- **Clear** - Odstraní všechny značky vyrobené funkcemi z této skupiny.

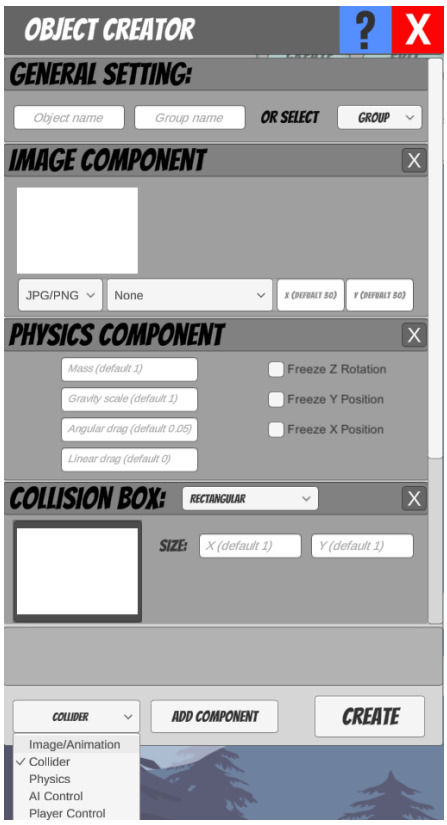
3.3.3 Nástroje pro práci s prototypy

V této sekci se zaměříme na používání nástrojů pro manipulaci s prototypy herních objektů a s tím související funkce pro výběr aktuálně používaného prototypu. Všechny tyto funkce můžeme vidět označeny pod čísly čtyři a pět, v ukázce hlavního panelu (3.3).

Vytváření a upravování

Na následujícím obrázku můžeme vidět okno pro vytváření prototypů. Nastavení jednotlivých vlastností prototypu pak probíhá přiřazením jednotlivých komponent (2.2.1). Rozbalovací menu, jež můžeme vidět v levém spodním rohu obrázku, slouží pro výběr konkrétní komponenty z předpřipravené množiny. Výběr potvrdíme stiskem tlačítka **AddComponent**. Ve středu okna se nachází posuvná plocha zobrazující panely s nastavením jednotlivých přidaných komponent.

Každý panel přidané komponenty, s výjmkou **General Setting**, obsahuje v pravém horním rohu tlačítko se symbolem ***x*** umožňující její odstranění.



Obrázek 3.9 Ukázka panelu pro vytváření prototypů

Pro finální vytvoření prototypu stiskeneme tlačítko **Create** ve spodní části ukázkového obrázku. Panel pro vytváření je stejný jako pro upravování aktuálně vybraného prototypu (3.3.3). Při zobrazení tohoto panelu se v něm zobrazí již existující komponenty a jejich nastavení.

Nastavení komponent

Zde se zaměříme na panely nastavení jednotlivých komponent a rozebereme si jejich nastavitelné vlastnosti (2.2.1). Začneme od povinné komponenty každého vytvářeného prototypu.

- **General Setting** - Slouží k nastavení jména protypu a jeho skupiny. Jméno prototypu nastavované v textovém poli zcela vlevo musí být narozdíl od jména skupiny unikátní. (4.3.2) To můžeme vybrat v rozbalovacím menu v pravé části. Pokud chce uživatel vytvořit novou skupinu pro tento objekt, může k tomu využít textové pole.

Obrázek 3.10 Komponenta **General Setting**

- **Image** - Přidává a nastavuje objektu jeho vizuální stránku. Na následující ukázce vidíme v levé části náhledový obrázek, který se tomuto prototypu při vytvoření přiřadí. Náhled se mění s aktuálním výběrem *assetu*.

  Ve spodní části máme panel, ve kterém můžeme vybrat typ vizuálního prvku, tedy zda se jedná o texturu či animaci. Po volbě typu z rozbalovacího menu můžeme vybrat jméno již existujího *assetu*, které objektu chceme přiřadit. Příklad vybraného jména vidíme u panelu textem **grass**.

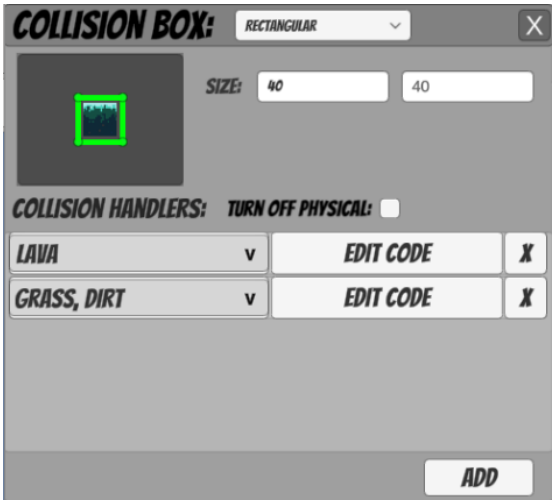
  Pokud bychom místo použití již existujícího *assetu* raději vytvořili nový, zvolíme ve výběru jména možnost **Create**. Ta otevře okno pro vytváření *assetů* vybraného typu (3.3.6). Po ukončení vytváření se nám jméno nového *assetu* přiřadí jako aktuálně vybrané.

  Další vlastností, kterou můžeme nastavit, je zobrazovaná velikost vybraného *assetu*. Tu specifikujeme ve dvou textových polích umístěných v pravé části panelu.

Obrázek 3.11 Komponenta **Image**

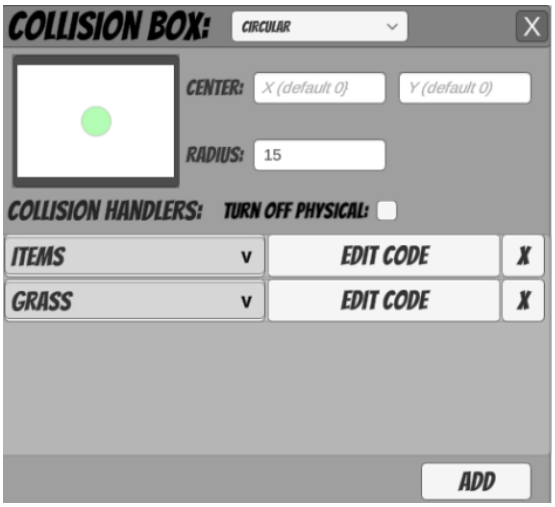
- **Collision box** - Nastavuje velikost a tvar pevné složky objektu a zároveň umožňuje vytvořit akce reagující na kolizi s vybranou množinou prvků (3.3.3). Nyní si uvedeme nastavení různých tvarů, které můžeme vybrat v horní části panelu pomocí rozbalovacího menu. To můžeme na jednotlivých obrazcích vidět v horní části vedle názvu komponenty.

  – **Obdélník** - Umožňuje nastavení šířky a výšky pomocí dvou textových polí, které můžeme vidět na následujícím obrázku.



Obrázek 3.12 Nastavení obdélníkového tvaru pevné složky

– **Kruh** - Spolu s nastavením poloměru umožňuje i nastavení středu. To vše pomocí tří vstupních polí, které opět můžeme vidět na následujícím obrázku.

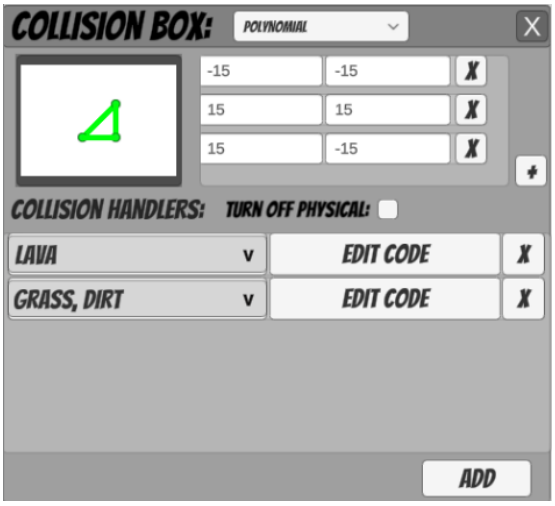


Obrázek 3.13 Nastavení kruhového tvaru pevné složky

– **Polygon** - V tomto případě není nastavení už tak jednoznačné jako u předchozích tvarů. Okrajové body polygonu, jež můžeme nastavit pomocí jednotlivých souřadnic v seznamu, který lze pozorovat v následující ukázce, na sebe navazují, počínaje nejvrchnější souřadnicí v seznamu.

  Jednotlivé panely souřadnic můžeme do seznamu přidat pomocí tlačítka se symbolem plus v levé spodní části. Odstranit je pak můžeme pomocí tlačítka se symbolem křížku v pravé části každého panelu souřadnic, pokud je počet souřadnic větší než tři. Pokud bychom měli méně než tři body, nejednalo by se již o polygon.

Vedle jednotlivých nastavení tvarů můžeme vidět panel náhledu, ve kterém se nám během úprav průběžně zobrazuje tvar pevné složky, a pokud obsahuje komponentu **Image** (3.3.3), tak i náhledový obrázek.

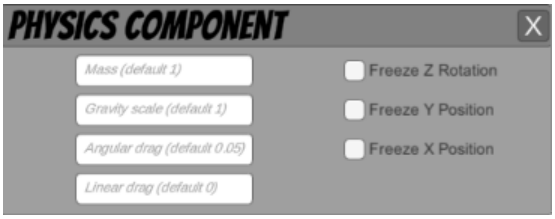


Obrázek 3.14 Nastavení pevné složky ve tvaru polygonu

Ve spodní části ukázkových obrázků (3.12)(3.13)(3.14) vidíme panely nastavující jednotlivé množiny prvků a přidělené akce. Výběr množiny probíhá zaškrtnutím jednotlivých prvků v rozbalovacím menu. Akci vytvoříme kliknutím na tlačítko **Edit Code**, které nám zobrazí okno editoru kódu.

Každý z panelů nastavení této komponenty navíc obsahuje zaškrávací políčko **Turn off physical**, které při zaškrtnutí nastaví pevnou složku tak, aby na případnou srážku nereagovala jako pevný objekt, ale pouze vyvolala přiřazené akce.

- **Physics** - Přidává objektu fyzikální vlastnosti a umožňuje jejich nastavení. Následuje ukázka této komponenty a tabulka vysvětlující jednotlivé nastavitelné položky.



Obrázek 3.15 Nastavení fyzikálních vlastností

- **AI controller** - Umožňuje nastavení a vytvoření akcí, které budou objekt automaticky řídit. Dále umožňuje nastavení pohybových akcí (2.4.5), které z již zmiňovaných vytvořených akcí budeme moci ovládat. Ty můžeme vybrat v rozbalovacím menu vyobrazeném následujícími ukázkami.
  - **Move** - Reprezentuje pohyb vlevo či vpravo, pro nějž umožňujeme nastavit akceleraci a maximální rychlost. Navíc obsahuje zaškrtávácí políčko **Grounded Only** zamezující vykonávání pohybu ve volném prostoru, tedy pokud nestojí na pevném povrchu.
  - **Jump** - Jednoduchý skok, u kterého umožňujeme nastavení horizontální a vertikální síly, čímž ovlivníme tvar a délku skoku. V následujícím

Obrázek 3.16 Ukázka nastavení pohybu **Move**

obrázku můžeme v horní části vidět zaškrtávací políčko *Chargable Jump*, které nám umožní skok přepnout na variantu skoku s proměnlivou intenzitou. Obě varianty skoku také obsahují zaškrtávací políčko **Grounded Only** se stejnou funkcí jako u pohybové akce **Move**.

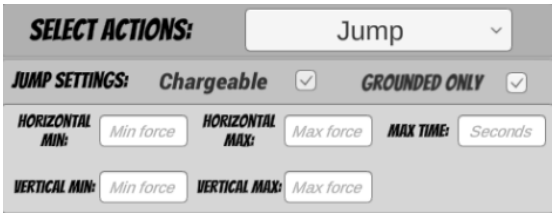
Obrázek 3.17 Ukázka nastavení pohybu **Jump**

- **Chargable jump** - Pro skok s proměnlivou intenzitou umožňujeme nastavení minimálních a maximálních sil v obou směrech - tedy horizontálním i vertikálním. Dále lze nastavit maximální délku chargování skoku.
- **Fly** - Umožní objektu létání, tedy pohyb ve všech základních směrech. Stejně jako u pohybu **Move** umožnujeme akceleraci a maximální rychlost.

Nad nastavením těchto akcí můžeme vidět dvě rozbalovací menu označená jako **On Create** a **Perpetual Update**, která reprezentují akce provedené

Název hodnoty	**Význam**
Mass	Hmotnost objektu
Gravity scale	Intenzita gravitačního zrychlení
Angular drag	Úroveň tření pro rotaci
Linear drag	Úroveň tření pro pohyb
Freeze Z rotation	Zakáže rotaci objektu
Freeze Y rotation	Zakáže pohyb po ose Y
Freeze X rotation	Zakáže pohyb po ose X

Tabulka 3.1 Nastavitelné hodnoty komponenty **Physics**



Obrázek 3.18 Ukázka nastavení skoku s proměnlivou intenzitou

Obrázek 3.19 Ukázka nastavení letu

při vytvoření objektu nebo při průběžné aktualizaci hry pomocí funkce *Update* (5.0.4). Jednotlivé akce pak vytvoříme, když v menu vybereme možnost **Create**, která zobrazí okno editoru kódu.

- **Player controller** - Umožňuje nastavit objekt tak, aby reagoval na specifické akce uživatele (4.4). Jednotlivé nastavitelné vlastnosti jsou stejné jako u předchozí komponenty, ale navíc umožňuje přiřazení jednotlivých akcí ke kombinacím kláves.

Obrázek 3.20 Ukázka nastavení komponenty hráče

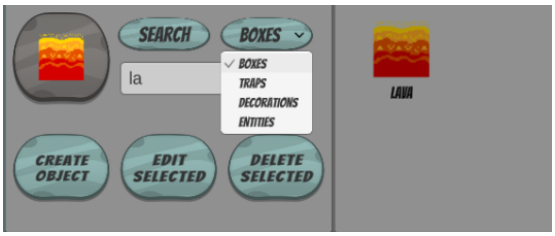
Na ukázkovém obrázku (3.20) můžeme ve spodní části vidět oblast pro nastavení jednotlivých akcí přidáváním a odebíráním jednotlivých panelů. Tyto panely můžeme přidat pomocí **Add** a odebrat pomocí tlačítka s křížkem, které se nachází v pravém rohu každého panelu.

Tyto panely obsahují tlačítka s názvem **Bind** a rozbalovací menu s výběrem akcí. Po stisku **Bind** začne systém registrovat jakékoliv stisky kláves či tlačítek myši. První registrovaná kombinace se nastaví k dané akci, pokud již není použitá jinde.

Druh akcí, které lze v rozbalovacím menu vybrat, se odvíjí od vybrané pohybové akce. Výběr také ovlivní základní směry pohybu, které můžeme ovládat. (2.4.5) Dále je přítomna funkce **Create**, která umožní vytvoření zcela nové akce.

Výběr existujících prototypů

V rámci práce s jednotlivými prototypy, umožňujeme v editoru zobrazení jejich náhledů po jednotlivých skupinách a jejich rychlé vyhledávání. Výběr aktuálně zobrazovaných skupin probíhá přes rozbalovací menu, které můžeme vidět na následujícím obrázku.



Obrázek 3.21 Panel výběru prototypů

Pokud zvolíme některý ze zobrazených náhledů, automaticky se nastaví jako aktuální prototyp a zobrazí se v šedém rámečku na levé straně ukázky.

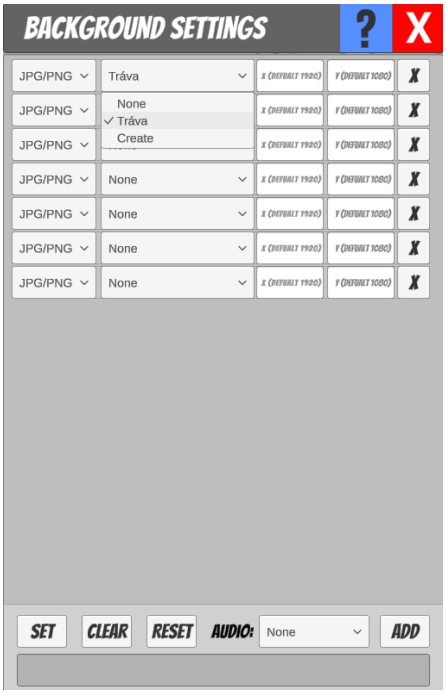
Abychom usnadnili vyhledávání ve větším množství prototypů, implementujeme pro to funkci. Ta najde prototyp, jehož jméno je nejpodobnější tomu, které uživatel zadal do textového pole při stiknutí tlačítka **Search**.

Odstranění prototypů

Jednotlivé vytvořené komponenty umožňujeme odstranit pomocí tlačítka **Delete selected** nacházejícím se na zmíněném panelu čtyři (3.3). To pak po stiknutí odstraní aktuálně vybraný prototyp.

3.3.4 Nastavení pozadí

Zde se zaměříme na funkci pro nastavení audiovizuální stránky pozadí, kterou spustíme stisknutím tlačítka **Change background** na panelu s číslem tři, v ukázce hlavního panelu (3.3). Ta vytvoří nové okno, jehož součástí je nastavení jednotlivých vrstev, u kterých umožňujeme přiřazení či vytvoření vizuálních *assetů*, stejně jako u komponenty **Image** (3.3.3).



Obrázek 3.22 Okno pro nastavení prvků pozadí

Tyto vrstvy pak přidáváme pomocí tlačítka **Add**, které můžeme vidět ve spodní části ukázky. Stejně jako u panelů akcí pak panely vrstev obsahují tlačítko se symbolem kříže, které je při stisku odstraní.

Pomocí sousedních tlačítek **Clear** a **Reset** umožňujeme odstranění všech přidaných vrstev a obnovení základního nastavení pozadí.

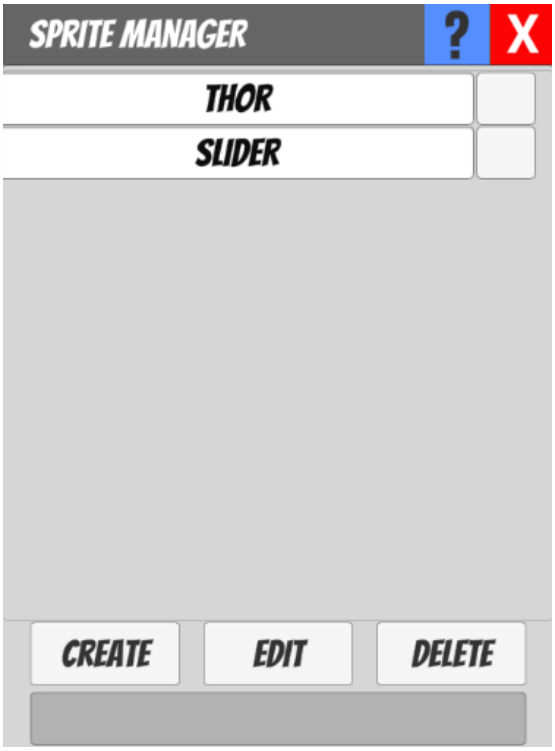
Kromě změny vizuální části můžeme nastavit či vytvořit i audio stopu, která se bude opakovaně přehrávat jako hudba na pozadí. Její výběr či vytvoření pak umožníme pomocí rozbalovacího menu označeném nadpisem **Audio**, podobně jako při výběru vizuálního *assetu*.

3.3.5 Panely manažerů

Všechny načtené herní *assety* můžeme spravovat pomocí panelů manažerů příslušných vždy jednomu konkrétnímu typu (4.2.2). Spustit je lze pomocí tlačítek umístěných v panelu číslo tři, v ukázce hlavního panelu (3.3). Každý z nich poskytuje sadu základních funkcí společnou všem manažerům a funkce specifické příslušnému typu *assetu*.

Image manager

Pro popis zmíněných společných funkcí použijeme manažer obrázků, jelikož ten vzhledem ke spravovanému typu nepodporuje jiné. Tyto funkce slouží k vytvoření nového *assetu*, jeho modifikaci a odstranění. Tlačítka spouštějící tyto funkce nalezneme na následující ukázce.



Obrázek 3.23 Ukázka panelu manažeru obrázků

Funkce pro úpravu existujícího *assetu* pod tlačítkem **Edit** vyžaduje, aby uživatel určil *asset*, který chce upravit. Odstranění funguje na stejném principu, ale navíc umožňuje odstranění více *assetů* zároveň.

Jednotlivé existující *assety* jsou ve všech manažerech zobrazeny v posuvném seznamu nacházejícím se uprostřed panelu tak, jak můžeme vidět na ukázkovém obrázku. Jednotlivé posuvníky se objeví až ve chvíli, kdy by se jednotlivé panely reprezentující existující *assety* nevešly do tohoto posuvného seznamu.

Označení jednotlivých *assetů* pro zmíněné funkce umožňují jejich příslušné panely pomocí zaškrtávácích polí v pravé části.

Animation manager

Panel manažeru animací, který můžeme vidět na vzorovém obrázku, implementuje spolu se základními funkcemi i funkce specifické nejen pro animace, ale i pro audiostopy. Následující popis jejich fungování tedy můžeme aplikovat i na manažera audiostop.

Obrázek 3.24 Ukázka panelu manažeru animací

Na obrázku (3.24) můžeme pozorovat novou sadu tlačítek **Play**, **Pause**, **Resume** a **Reset** reprezentujících funkce pro přehrání, pozastavení, znovuspuštění a resetování vybrané množiny animací. Ty vybereme stejně jako v případě panelů obrázků z předchozího příkladu.

Audio manager

Vizuální vzhled i funkce tohoto manažeru jsou stejné jako pro panel animací. Jednotlivé manažery odlišují pouze panely pro načítání a upravování *assetů*, které vytvářejí společné funkce **Create** a **Edit**. Na tyto panely se zaměříme v následující sekci.

3.3.6 Panely načítání a upravování *assetů*

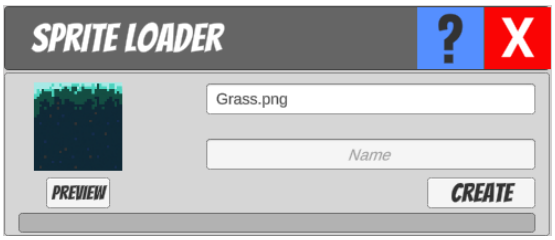
Na následujících příkladech předvedeme fungování jednotlivých panelů načítajících nové *assety* (4.2). Tyto panely jsou stejné jako panely pro upravování, s výjmkou určení jména *assetu*, jelikož to je doplněné z již existujícího *assetu*. Provedené úpravy se pak projeví na všech místech, kde byl upravovaný *asset* použit.

Stejně jako u manažerů mají i panely pro načítání jisté společné vlastnosti. Například pro načtení nových *assetů* potřebujeme adresu vedoucí k příslušným souborům. Tato adresa pak musí mít buďto formát **URL**, pokud chceme soubory stahovat pomocí **HTTP**, nebo jméno a typ souboru. V takovém případě pak tento soubor hledáme ve složce **StreamingAssets** umístěné mezi soubory hry.

Dále pak musíme načítaným *assetům* přiřadit jméno, které je unikátní v rámci daného typu *assetu*. Tímto jménem se pak na vytvořený *asset* budeme později odkazovat. Nyní si rozebereme panely načítající jednotlivé typy *assetů*.

Obrázky

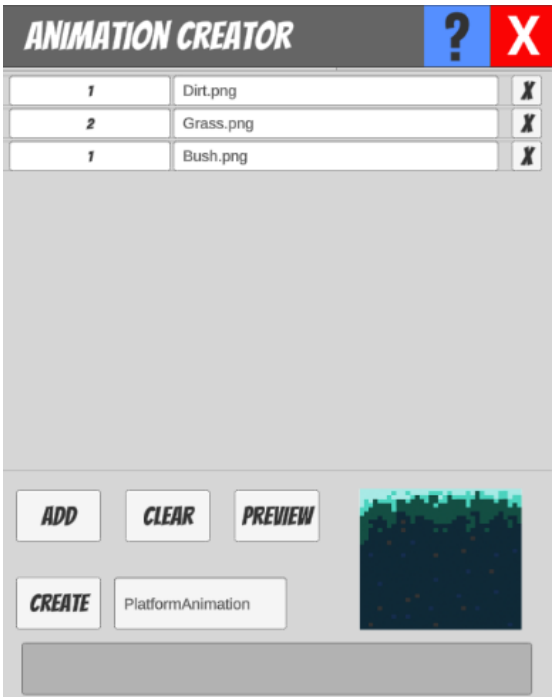
Panel pro načítání obrázků/textur obsahuje, vyjma textových polí pro zadání adresy a jména, ještě panel náhledu stahovaného obrázku. Tento náhled zobrazíme stiskem tlačítka **Preview**. Načtení vybraného *assetu* pak potvrdíme stiskem tlačítka **Create**.



Obrázek 3.25 Panel načítání obrázků

Animace

Jak již bylo zmíněno v předchozích kapitolách (4.2.3), animace nemůžeme stahovat. Namísto toho tedy umožňujeme vytváření animací pomocí následujícího panelu.



Obrázek 3.26 Panel vytváření animací

Ve vrchní části můžeme vidět řádky obahující adresu obrázku a vlevo od nich textové pole, do kterého nastavíme dobu, po kterou se mají tyto obrázky v průběhu animace zobrazovat. Čas je uváděný v sekundách a umožňuje přiřazení desetinných

čísel. Jednotlivé řádky reprezentují posloupnost zobrazovaných obrázků, počínaje nejvrchnějším. Tlačítko **Add** pak přidá jeden řádek na konec posloupnosti.

Odstranění konkrétního řádku pak opět umožňuje tlačítko se znakem křížku umístěné v pravém kraji každého z nich. Pokud chceme odstranit všechny přidané řádky, stiskneme tlačítko **Clear**.

Stejně jako u předchozího panelu můžeme vytvářenou animaci zobrazit v náhledové ploše umístěné v pravém dolním rohu stiskem tlačítka **Preview**.

Vytvoření animace pak potvrdíme stiskem tlačítka **Create**.

Audio stopy

Načítání audio stop také vyžaduje adresu souboru a unikátní jméno. Tento panel ale navíc umožňuje nastavení jednotlivých vlastností stažené stopy a upravení způsobu přehrávání pomocí posuvníků, které můžeme vidět na následující ukázce v horní části panelu.

Obrázek 3.27 Panel načítání audio stop

Na pravé straně od těchto posuvníků můžeme vidět textové pole s aktuální hodnotou reprezentovanou pozicí posuvníku. Pokud chceme posuvníku nastavit přesnou hodnotu, můžeme ji zadat pomocí těchto polí. V následující tabulce si uvedeme významy těchto hodnot.

Název hodnoty	Význam
Priority	Určuje prioritu zvuku při přehrávání více stop současně
Stereo	Určuje intenzitu zvuku na konkrétní straně sterea
Pitch	Upravuje rychlost nahrávky, a tím i tóninu
Volume	Hlasitost přehrávané nahrávky

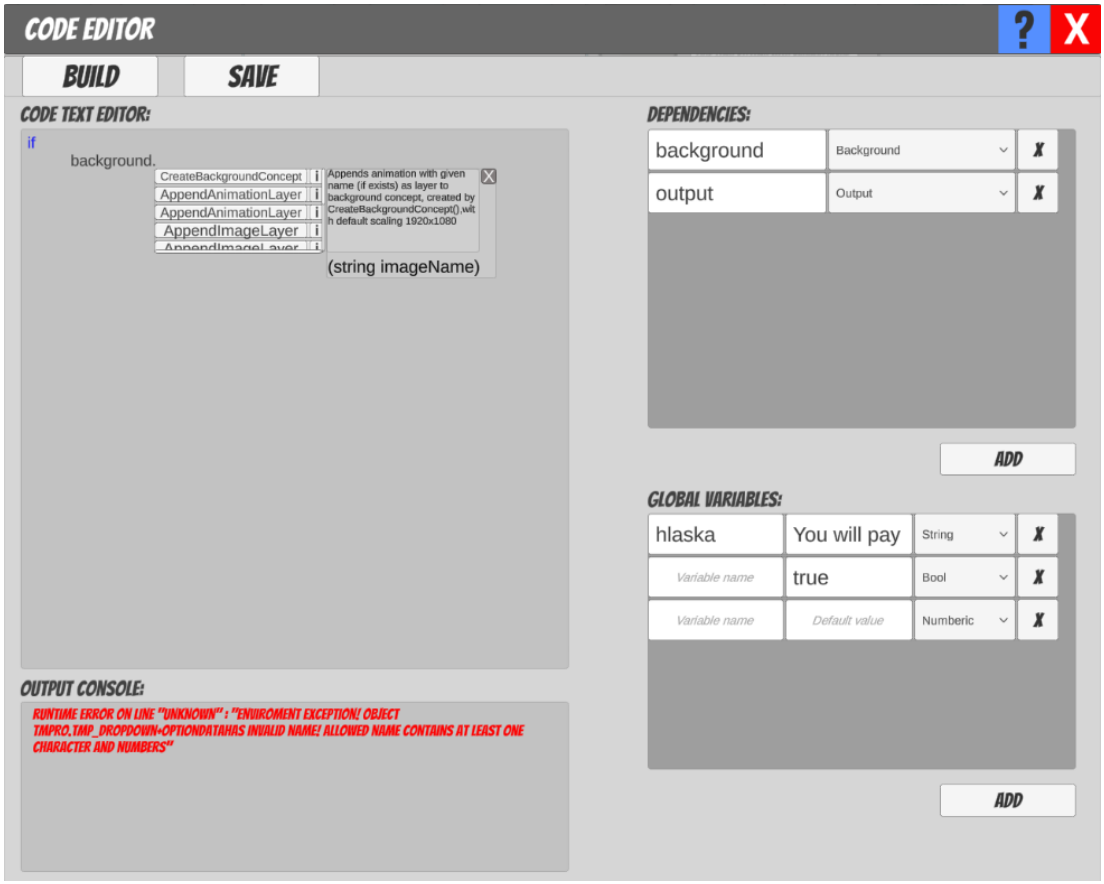
Tabulka 3.2 Nastavitelné hodnoty vytvářené audio stopy

Poslední částí nastavení je možnost určit, zda má být načtená stopa přehrávána ve smyčce pomocí zaškrtávácího políčka **Loop**.

Panel dále obsahuje tlačítka **Play**, **Stop** a **Save**, která slouží k přehrání, pozastavení a uložení načtené audio stopy.

3.3.7 Editor kódu

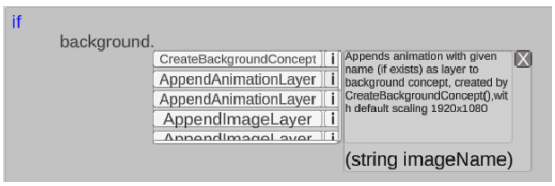
Zde se zaměříme na používání editoru našeho kódu, který použijeme ke tvorbě jednotlivých akcí objektu (2.4.1). Na následujícím obrázku vidíme ukázku našeho editoru, kterou nyní rozebereme podle jednotlivých pojmenovaných částí.



Obrázek 3.28 Ukázka editoru kódu

Code Text Editor

V panelu s tímto nadpisem budeme vytvářet náš kód, k čemuž nám pomůžou zabudované funkce pro zvýraznění klíčových slov syntaxe (4.4.4) a **IntelliSence** (4.4.4), která nám zobrazí nabídku dostupných funkcí.

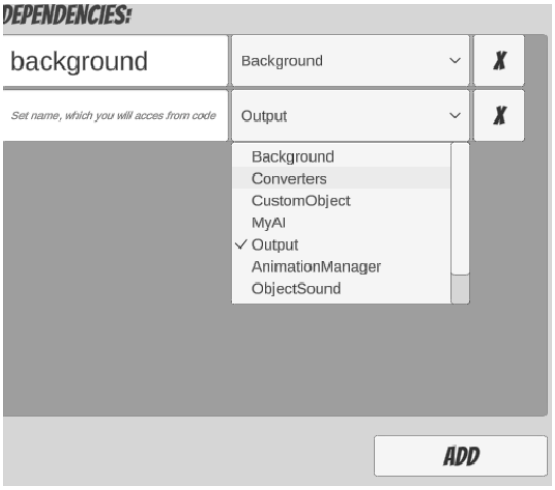


Obrázek 3.29 Ukázka fungování **Intellisence**

Na ukázkovém obrázku (3.29) vidíme panel funkce intellisence, který se sestává z posuvného menu se seznamem dostupných metod na připojeném objektu. Jednotlivé položky seznamu pak obsahují tlačítko se jménem metody, které při stisku automaticky toto jméno doplní do kódu. Dále pak obsahují tlačítko se symbolem *i* sloužící k zobrazení popisu dané metody. Součástí tohoto popisu je i seznam parametrů, které metoda ke svému provedení vyžaduje. Ten můžeme pozorovat ve spodní části panelu nápovědy. Panel pak zavřeme stisknutím tlačítka se symbolem křížku.

Dependencies

Pomocí tohoto panelu můžeme do našeho kódu přidávat skupiny metod po jednotlivých třídách poskytnutých prostředím. Těmto třídám pak přiřadíme jméno, pomocí kterého se na ně budeme odkazovat z našeho kódu. Jméno a příslušnou třídu pak uvedeme do jednotlivých řádků tak, jak můžeme vidět na ukázce (3.30).

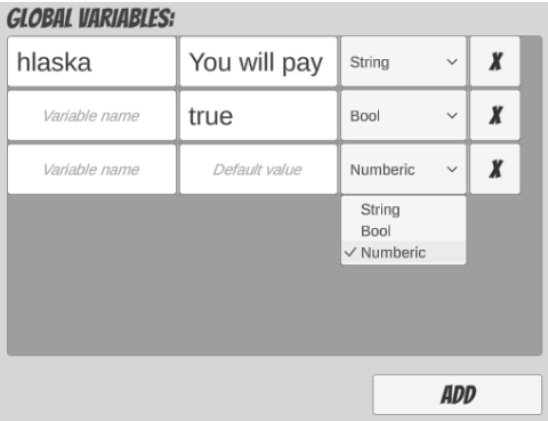


Obrázek 3.30 Ukázka přidání třídy

Tyto řádky pak do posuvného seznamu přidáváme stisknutím tlačítka **Add** a odstraňujeme pomocí tlačítka se symbolem křížku umístěném v pravém rohu každého z nich.

Global variables

Tento panel nám umožní přidávat a nastavovat globální proměnné, na které budeme moci odkazovat z našeho kódu. Ty budeme používat k přenesení stavu kódu, mezi jednotlivými provedeními. V panelu jsou pak reprezentovány jednotlivými řádky v posuvném seznamu. Jejich vytváření a odebírání pak funguje stejně jako v panelu **Dependencies**.



Obrázek 3.31 Ukázka vytvoření globálních proměnných

Do těchto řádků pak uživatel vyplní unikátní jméno proměnné, výchozí hodnotu a vybere z rozbalovacího menu typ proměnné. Pokud jméno není unikátní nebo není vyplněné, je při kompilaci vyvolána výjimka, která se zobrazí v panelu **Output console**.

Do textového pole pro výchozí hodnotu nemusí uživatel nastavit žádnou hodnotu. V takovém případě se pak použije hodnota nastavená systémy, jejichž přehled můžeme vidět v následující tabulce (3.3). Ta dále obsahuje podmínky pro vyplnění určující hodnoty, kterou může uživatel zadat jako výchozí. Pokud uživatel zadá neplatnou hodnotu, systém vypíše chybovou hlášku.

Typ hodnoty	**Hodnota systému**	**Podmínky vyplnění**
String	Prázdný text	Jakákoliv sekvence znaků
Bool	False	**true** nebo **false**
Number	0	Celá nebo desetinná čísla s tečkou

Tabulka 3.3 Přehled typů a základních hodnot globálních proměnných

Output console

Slouží k vypisování chybových hlášek vybarvených červenou barvou a informativních hlášek vybarvených černou barvou při kompilaci a ukládání programu.

3.3.8 Tvorba akcí

Prostředí pro tvorbu kódu nových akcí jsme již popsali. Nyní se zaměříme na pravidla, která během psaní kódu musíme dodržovat.

Používání operátorů

V kapitole analýzy jsme uvedli typy proměnných, které náš jazyk bude podporovat (2.4.4). Pro každý typ proměnné pak podporujeme sadu binárních a unárních

operátorů, které uvedeme v následujících tabulkách. Návratovou hodnotou **True** budeme označovat pravdivou logickou hodnotu, tedy **bool**.

- **Společné**:

Operátory	**Funkce**	**Výsledná hodnota**
==	Porovná dvě hodnoty	**True** pokud se rovnají
!=	Porovná dvě hodnoty	**True** pokud se nerovnají

Tabulka 3.4 Výčet společných operatorů

- **String**:

Operátory	**Funkce**	**Výsledná hodnota**
+	Spojí dva texty	Spojený string

Tabulka 3.5 Výčet operátorů pro typ **string**

- **Num**:

Operátory	**Funkce**	**Výsledná hodnota**
+, -, *, /	Aritmetické operace pro čísla	Výsledná hodnota **num**
%	Aritmetická operace modulo	Výsledná hodnota **num**
«, »	Bitový posun doleva a doprava	Výsledná hodnota **num**
<, >	Ostře menší nebo větší	True pokud platí
<=, >=	Menší nebo větší rovno	True pokud platí
&, ,||	Bitové operace AND, OR, XOR	Výsledná hodnota **num**
-	Unární operátor negace	Negovaná hodnota **num**

Tabulka 3.6 Výčet operátorů pro typ **num**

- **Bool**:

Operátory	Funkce	Výsledná hodnota
&&, ||	Logické AND a OR	True pokud platí
!	Unární operátor **NOT**	Negovaná hodnota **num**

Tabulka 3.7 Výčet operátorů pro typ **bool**

Tyto operátory pak můžeme použít ke konstrukci složitějších výrazů, jak můžeme vidět na následujícím příkladě. Unární operátory umisťujeme vždy napravo od hodnoty a binární mezi dvě hodnoty daného typu. Pro unární operátor **!** navíc platí, že musí být od negovaného výrazu oddělen alespoň jednou mezerou.

Program 7 Ukázka vytvořeného výrazu **bool**

```
1       (2 * 5) + 5 > x % 2 || (x / 10) * -5 > 20 && ! true
```

Jednotlivé typy hodnot mají své operátory, pomocí kterých s nimi můžeme pracovat. Pokud ale operátor není podporován danou hodnotou, prostředí se pokusí o převedení. Pokud i tento pokus selže, je vyvolána výjimka.

Vytváření proměnných a přiřazení hodnot

Ukázali jsme si, jaké operace můžeme s hodnotami provádět. Zde se zaměříme na to, jak výslednou hodnotu uložit do proměnné pro pozdější využití. Abychom ale byli schopni ukládat hodnoty do proměnných, musíme je neprve definovat. U té se řídíme pravidly, které jsme již stanovili v sekci analýzy (2.4.4).

Pro přiřazení do existující proměnné stačí pouze její jméno, operátor přiřazení a hodnota. Na následující ukázce můžeme vidět vytvoření proměnné **x** a přiřazení nové hodnoty.

Program 8 Ukázka přiřazení do proměnné **x**

```
1       num x = 5
2       x = (2 * 5) + 5
```

Stejným způsobem pak můžeme přiřazovat hodnoty do globálních proměnných, které vytváříme v prostředí editoru kódu (3.3.7).

Volání funkcí

Připojené třídy **Dependencies** nám poskytují možnost volat prostředím definované metody. Těmto třídám jsme pak v rámci připojování definovali unikátní jméno, kterým se na ně budeme v rámci kódu odkazovat (3.3.7). Tedy například mějme třídu se jménem **output** obsahující metodu **Print** pro výpis hlášky. Její volání by pak vypadalo následovně:

Program 9 Ukázka volání funkce **Print**

```
output.Print("Hello")
```

Jménem třídy určíme množinu volaných metod a jménem metody pak konkrétní metodu. Tyto dvě jména pak oddělujeme tečkou.

Těmto metodám pak předáváme požadovaný počet argumentů konkrétních typů v kulatých závorkách. Jednotlivé argumenty oddělujeme čárkou. Následující příklad funkce **PrintSum** přijímá tři argumenty v pořadí **string**, **num**, **num**:

Program 10 Ukázka volání funkce **PrintSum**

```
output.PrintSum("The sum of A and B is :", a, b)
```

Jako agrumenty funkce můžeme definovat výrazy obsahující konstanty a lokální či globální proměnné. Argumenty obsahující volání funkcí však nejsou v našem jazyce podporovány.

Podmíněné bloky a cykly

Abychom mohli ovlivnit běh kódu, například na základě stavu hry, implementujeme podmíněné bloky kódu a cykly tak, jak jsme definovali v kapitole analýzy (2.4.1). Podmínky těchto bloků pak definujeme následovně:

Program 11 Ukázka volání funkce **Print**

```
if a == b
   #Code
elseif b <= c
   #Code
else
   #Code
fi
```

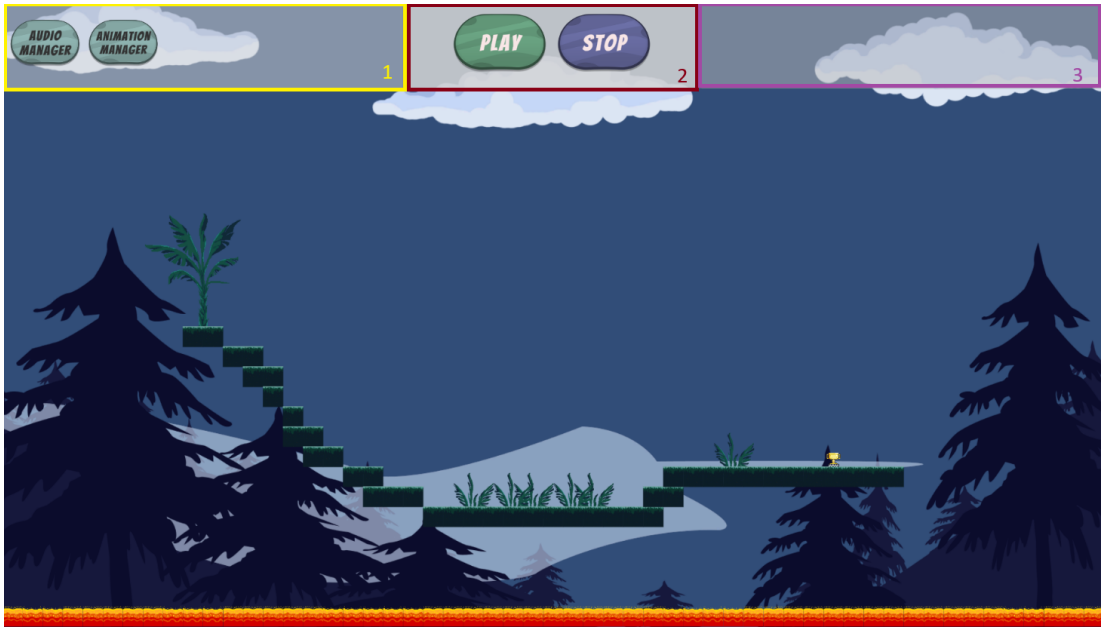
Jak můžeme vidět, podmínky jsou od klíčových slov vždy oddělené alepoň jednou mezerou. Samotná podmínka je pak výraz, jehož výsledkem je **bool**. Stejná pravidla platí i pro cyklus **while**:

Program 12 Ukázka volání funkce **Print**

```
while a > b
   #Code
end
```

3.3.9 Mód ladění

Na závěr se zaměříme na rozbor módu ladění (4.7.2), který při své aktivaci nahradí hlavní panel (3.3), jak můžeme vypozorovat z ukázky.



Obrázek 3.32 Ukázka módu ladění

V horní části se nachází panel, pomocí kterého můžeme kontrolovat průběh ladění. Jednotlivé sekce jsou stejně jako u ukázky hlavního panelu označeny očíslovanými barevnými čtverci, na které se budeme nyní odkazovat.

- V části označené číslem jedna se nacházejí funkce, které nám umožní přístup k manažerům animací a audio stop. Díky tomu můžeme například pozastavit přehrávání konkrétní audio stopy.
- V prostřední části panelu s číslem dva se nachází tlačíta **Play** a **Stop**, která řídí běh tohoto prostředí. Při stisku **Play** se aktivují všechny objekty na herní ploše. To znamená například spuštění všech přiřazených animací, aktivování fyziky pro jednotlivé objekty či registrování stisků jednotlivých kláves. Zároveň se tlačítko změní z **Play** na **Pause**. Při stisku tlačítka **Pause** se pozastaví všechny spuštěné komponenty hry, jejíž příklady jsme si uvedli.

Obrázek 3.33 Ukázka změněného tlačítka **Play**

  Tlačítko **Stop** pak vypne mód ladění a uvede všechny herní objekty do pozice a stavu před spuštěním.

- Poslední sekce hlavního panelu označná číslem tři slouží pro výpis jakýchkoli hlášek, které jsou během ladění vyvolány.

4 Programátorská dokumentace

V této kapitole se zaměřujeme na konkrétní implementace jednotlivých důležitých částí našeho projektu. Při jejich popisování se snažíme postupovat podle našeho plánu, který jsme si v průběhu analýzy vytvořili (2.6). Ne vždy to je ale možné, jelikož některé části jsou provázané.

4.1 Pracovní plocha

Jak jsme již uvedli v sekci analýzy, pracovní plocha je prostor, kde budeme manipulovat s herními objekty. Operace na pracovní ploše jsou pak řízeny skrze třídu **EditorCanvas**, která je definována jako *singleton*. Tato třída má na starost vytváření a mazání objektů na určené pozici. Při vytváření objektů pak ukládáme jejich instanci spolu s pozicí a identifikátorem, unikátním pro každý objekt. Tímto způsobem si třída udržuje přehled o všech vytvořených prvcích na pracovní ploše. Díky těmto datům pak třída implementuje i funkce pro dotazy na existenci objektů na určené pozici či pro smazání všech instancí se stejným identifikátorem.

Dále tato třída registruje stisknutí kláves či tlačítek myši za podmínky, že je kurzor v hranicích pracovní plochy. Údaje o těchto stisknutých klávesách a pozici kurzoru v době stisku se pak předají třídě vykonávající aktuálně zvolenou akci pro manipulaci s objekty. Pokud žádná akce není zvolená, je použita základní akce, která zodpovídá za pohyb po pracovní ploše.

Kdybychom pozice těchto stisků reprezentovali přesně, bylo by pro uživatele velice obtížné vytvořit mezi prvky jakékoliv zarovnání. Z tohoto důvodu je pracovní plocha překryta pomyslnou mřížkou. Díky této mřížce jsme pak schopní z pozice určit příslušnou buňku mřížky. Z této buňky pak vezmeme pozici jejího středu, a tu pak předáme místo původní pozice stisku.

4.1.1 Funkce pro manipulaci s objekty

Do této kategorie řadíme funkce pro vytváření, mazání, vybírání a posouvání objektů na pracovní ploše. Tyto funkce jsou implementovány třídami dědícími od rodičovské třídy **EditorActionBase**. Ta poskytuje základní strukturu *virtuálních metod* společných pro všechny akce.

Ve chvíli, kdy uživatel v nástrojích vybere jednu z těchto funkcí, nastaví se jako aktuální akce editoru. Aktuální akci pak můžeme mít vybránu vždy pouze jednu. Při předání informace o stisku klávesy si funkce nastaví všechna potřebná data. Samotná operace se pak bude volat s každým voláním funkce *Update* ve třídě **EditorCanvas**, která jí pokaždé předá aktuální pozici kurzoru. Akce se provádí, dokud uživatel nepřestane držet příslušnou klávesu. Při dokončení každé akce se pak uloží záznam o akci a příslušné protiakci (2.1), kterou uložíme do deníku akcí.

4.1.2 Deník akcí

Deník implementovaný třídou **Journal** uchovává zmíněné záznamy o akcích v podobě dvojic instancí tříd **JournalActionDTO** reprezentující akci a protiakci.

Každá instance pak uchovává *delegáta* na příslušnou metodu akce a parametry pro tuto metodu v podobě *stringu*. Záznamy jsou uchovávány ve dvou datových strukturách typu *zásobník* (5.0.4), kde jeden slouží pro uchování provedených akcí a druhý pro akce odstraněné. Zásobník provedených akcí má pak z důvodu paměťové optimalizace nastavenou maximální velikost, kterou kontrolujeme při přidávání záznamů. Ve chvíli, kdy této kapacity dosáhneme, se dolní polovina zásobníku vymaže, aby uvolnila místo novým záznamům.

Při odstraňování či znovuobnovování akcí přesouváme záznamy z jednoho zásobníku na druhý. Pokud uživatel provede novou akci, tedy vytvoří nový záznam, odstraněné akce se smažou. Díky vhodně vybrané datové struktuře máme navíc zajištěno správné pořadí prováděných akcí.

4.2 Načítání a správa *assetů*

K nastavování různých segmentů naší hry implementujeme i funkce pro načítání externích *assetů*. Konrétněji pak obrázků ve formátu **JPG** nebo **PNG**, které pak můžeme složit do animací, a audio stop ve formátu **MP3** nebo **MP2**. O jejich získání se starají příslušné *loadery*. Načtená data jsou pak ukládána příslušnými správci zdrojů, s výjmkou náhledových obrázků, které jsme mohli vidět v uživatelské sekci(3.3.6).

4.2.1 Loadery *assetů*

Jelikož se pohybujeme v prostředí Unity, musíme k načtení jednotlivých zdrojů využívat zabudované metody jím poskytnuté, abychom je pak byli schopni přiřadit jednotlivým objektům. Pomocí třídy **UnityWebRequest** tedy umožníme načítání jednotlivých *assetů* buď ze složky **StreammingAssets**, která je umístěna mezi složkami projektu, anebo pomocí **HTTP**. Metody, které k tomu budeme využívat, nám pak vrátí instance objektů, kterými Unity reprezentuje naše *assety*.

Všechny naše třídy loaderů podporují statické metody pro načtení či přímé přiřazení *assetu* k hernímu objektu. Metody pro přímé přiřazení budeme využívat pouze pro případy zmíněných náhledů, jelikož u těchto *assetů* není jistota, zda budou použity. Ostatním herním objektům pak budeme *assety* přiřazovat pouze skrze příslušné manažery (4.2.2).

Těmto metodám pak při zavolání předáme příslušné zdrojové objekty obsahující informace potřebné k načtení či vytvoření *assetů*. Jelikož animace není možné načíst, ani vytvořit při běhu programu, budeme v rámci metod příslušné třídy volat již vytvořené metody pro načtení obrázků, kterým poté pouze nastavíme jejich délky zobrazení. Ty pak přiřadíme do námi vytvořeného objektu reprezentujícího animaci. Tu budeme přehrávat pomocí tříd animátorů.

V následující tabulkách jsou shrnuty typy loaderů, *assety*, jejich zdrojové objekty a výsledné **Unity** objekty.

Asset	Unity objekt
Obrázek/Textura	Sprite
Zvuková stopa	AudioClip
Animace	—

Tabulka 4.2 **Unity** reprezentace assetů

Loader	*Asset*	Zdrojový objekt
SpriteLoader	Obrázek/Textura	SpriteSourceDTO
AudioLoader	Zvuková stopa	AudioSourceDTO
AnimationLoader	Animace	AnimationSourceDTO

Tabulka 4.1 Výčet loaderů assetů a jejich typů

4.2.2 Správa *assetů*

Z důvodů, jež jsme popsali v kapitole analýzy (2.1), ukládáme všechny načtené objekty do námi vytvořených tříd manažerů. Tyto třídy pak definujeme jako *singletony*, abychom je mohli využít v celém projektu. V následující tabulce si uvedeme jména těchto manažerů a *assety*, které spravují:

Manažer	Spravované *assety*
SpriteManager	Obrázky/Textury
AudioManager	Zvukové stopy
AnimationManager	Animace

Tabulka 4.3 Výčet manažerů assetů a jejich typů

Manažeři pak spravují tyto jednotlivé *assety*, jejich vytvořující objekty a registrují kontrolery objektů (4.2.3), které tyto zdroje využívají. Jelikož je cílem těchto manažerů nenačítat zbytečně již existující zdroje, přiřadíme jednotlivým *assetům* jejich unikátní jména. S těmito jmény poté párujeme registrované kontrolery i zdrojové objekty. Každý z manažerů pak implementuje následující metody:

- **Načtení *assetu*** - Pomocí příslušného zdrojového objektu načte požadovaný *asset* skrze jeden z implementovaných loaderů. Načtený *asset* pak uloží s příslušným jménem do dat manažeru. Spolu s tím uloží i zdrojový objekt.
- **Úprava existujícího *assetu*** - Nahradí existující *asset* a zdrojový objekt s poskytnutým jménem, nově načtěným *assetem* a jeho zdrojem. Dále se jeho kopie nastaví ve všech registrovaných kontrolerech.

- **Odtranění existujícího *assetu*** - Odstraní existující *asset* na základě poskytnutého jména. Zároveň pak odstraní jeho kopie, ze všech registrovaných objektů.
- **Nastavení kontrolerů objektu** - Podle poskytnutého jména nastaví kontroleru objektu kopii příslušného *assetu* a zařadí ho mezi registrované kontrolery.
- **Přidání kontroleru** - Přidá nový kontroler mezi registrované s pomocí poskytnutého jména *assetu*.
- **Odstranění kontroleru** - Odstraní registrovaný kontroler pomocí poskytnutého jména a unikátního *ID* instance kontroleru.
- **Ověření registrace kontroleru** - Na základě poskytnutého jména *assetu* a unikátního *ID* kontroleru ověří, zda je poskytnutý kontroler registrovaný.

Dále pak manažery obsahují své speciální metody, které jsou unikátní pro *assety*, jenž spravují. Tedy například metody pro přehrání, pozastavení či znovuspuštění animací. Tyto metody pak provedou akce na registrovaných kontrolerech přiřazených k poskytnutým jménům.

4.2.3 Kontrolery

K přiřazení a ovládání načtených *assetů* k jednotlivým objektům nám slouží již zmíněné třídy kontrolerů, jejichž instance jsou přímo spjaty s jednotlivými herními objekty. Jinými slovy: každý herní objekt pracující s nějakým typem *assetu* má přiřazenou unikátní instanci příslušného kontroleru. Tyto kontrolery pak implementují metody, přímo ovlivňují kopii *assetu* a komponenty jim přiřazeného objektu. Ke každému typu *assetu* pak implementujeme příslušnou třídu kontroleru.

- **SpriteController** - Tento kontroler je odpovědný za přiřazení a nastavení textury do komponenty **SpriteRender** zodpovědné za její vykreslení. Součástí nastavení je pak například škálování či barva překrytí textury.
- **AudioController** - Přiřazuje audio stopu komponentě AudioSource odpovědné za její přehrání a nastavení jednotlivých jejích vlastností jako jsou hlasitost či rychlost přehrávání. Tento kontroler pak implementuje metody, které umožňují nastavit tyto vlastosti a ovlivnit přehrávání, tedy spustit, pozastavit či znovuspustit aktuální stopu.
- **AnimationController** - Podobně jako AudioController je zodpovědný za přiřazenou animaci a poskytuje metody ovlivňující její přehrávání. Jelikož ale *asset* animace není reprezentována objektem implementovaným v prostředí **Unity**, nemá tento *asset* implementovanou ani komponentu pro přehrávání. Z tohoto důvodu má kontroler k sobě přiřazenou jednu ze tříd animátorů.

4.2.4 Animátory

Jsou třídy implementující rozhraní **IAnimator**, jež jsou zodpovědné za přiřazování textur po určených časových intervalech v nastaveném pořadí. Animátory pak dělíme podle druhu komponent, kterým textury přiřazují. Těmi jsou v našem případě **SpriteRenderer** a **Image**. Animátor komponenty **Image** pak využíváme pouze k animování náhledů v editoru. Náhledy v editoru totiž spadají do kategorie *UI objektů*, které mají přiřazenou právě zmíněnou komponentu [35].

4.3 Tvorba a správa prototypů objektů

V této části se zaměříme na implementaci tříd souvisejících s vytvářením *prototypů* herních objektů. Jak jsme si již uvedli (2.1), u vytváření nových prototypů specifikujeme, jaké komponenty bude tento herní objekt obsahovat. Těmto našim komponentám v průběhu vytváření objektu nastavíme příslušné parametry, od kterých se pak budou odvíjet vlastnosti konkretního herního objektu. Každý vytvářený objekt pak bude obsahovat povinnou komponentu, která specifikuje jméno objektu a skupiny, podle nichž budeme tyto objekty kategorizovat.

4.3.1 Komponenty

V této části si přiblížíme implementace komponent, které poskytujeme při tvorbě prototypů. Jednotlivé komponenty odpovídají komponentám platformových her, jejichž využití jsme již nastínili (2.2.1). Všechny komponenty pak implementují metody z abstraktní rodičovské třídy **CustomComponent**, které využíváme, když z prototypů vytváříme konkrétní instance herních objektů. Tyto metody přiřazují k instancím námi vytvořené kontrolery (4.2.3), jako třeba **SpriteController**, případně odpovídající **Unity** komponenty, které budeme s pomocí těchto kontrolerů ovlivňovat.

Vizuální komponenta

Reprezentovaná třídou **VisualComponent** slouží k nastavení vizuální stránky objektu. Při vytváření komponentě nastavujeme, zda má zobrazovat jednoduchou texturu či animaci, její jméno a velikost. Díky určenému typu a jménu pak při vytváření instance získáme konkrétní *asset* skrze příslušného manažera (4.2.2). Tato komponenta pak objektu přidá příslušný kontroler (4.2.3), kterému *asset* předá spolu s jeho velikostí.

Komponenta kolizí

Odpovídá komponentě pro pevnou složku objektu (2.2.2) a je reprezentovaná serií tříd s rodičovskou třídou **ColliderComponent**. Všechny tyto třídy reprezentují různé tvary pevných složek a nastavují jejich velikosti. Při vytváření herních objektů přidávají kontroler kolizí a odpovídající **Unity komponenty**, které tento kontroler následně spravuje. V následující tabulce (4.4) jsou zobrazeny třídy našich komponent, jim odpovídající **Unity komponenty** a tvary, které reprezentují.

Třída komponenty	Unity ekvivalent	Tvar
BoxColliderComponent	BoxCollider2D	Obdélník
CircleColliderComponent	CircleCollider2D	Kruh
PolygonColliderComponent	PolygonCollider2D	Polygon

Tabulka 4.4 Seznam tříd reprezentující tvary pevných složek

Tato komponenta dále umožňuje přiřazení akcí naprogramovaných v našem jazyce, které se provedou při kolizi s konkrétními skupinami prvků. Ty jsou reprezentovány instancemi tříd **CollisionDTO**, které jsou při vytváření předány přidanému kontroleru. Ten pak při každé kolizi daného objektu s libovolným dalším zkontroluje, zda je jméno nabouraného objektu přiřazeno k nějaké akci a pokud ano, tak akci provede.

Fyzikální komponenta

Komponenta implementovaná třídou **PhysicsComponent**, jejíž žádoucí vlastnosti a důvody k implementaci jsme uvedli v předchozí sekci (2.2.2), přidává vytvářeným objektům fyzikální vlastnosti pomocí **Unity** komponenty **RigidBody2D**. Dále přidává i příslušný kontroler, **PhysicsController**, umožňující ovlivňování této přidané Unity komponenty. Tomu předá sadu hodnot(3.1), které kontroler následně nastaví i zmíněné komponentě.

Komponenta nepřítele

Komponenta nepřítele, reprezentovaná třídou **AIComponent**, přidává vytvářeným objektům kontroler **AIAgent**, do kterého předá seznam uživatelem vybraných pohybových akcí, které objekt nepřítele může vykonávat. Dále předává akce naprogramované v našem zjednodušeném jazyce. Tyto akce určují, co se má s daným objektem stát při jeho vytvoření, při volání funkce **Update** a při jeho zničení. Tento přidaný kontroler pak v průběhu hry řídí objekt nepřítele na základě naprogramovaných akcí. Ty mohou například zahrnovat přesun nepřítele ze současné pozice na jinou, jak uvidíme v následující části (4.4). V takovém případě pak kontroler použije přiřazené pohybové akce (3.3.3).

Komponenta hráče

Stejně jako v případě komponenty nepřítele, komponenta hráče implementovaná třídou **PlayerComponent**, umožňuje nastavení zvolených pohybových akcí do přidávaného kontroleru **PlayerController** a také předává stejnou skupinu naprogramovaných akcí. Dále však předává seznam objektů typu **ActionBindDTO** reprezentující mapování jednotlivých pohybových (3.3.3) či naprogramovaných akcí, které kontroler při stisku příslušné klávesy čí jejich kombinací provede. Jednotlivé kombinace kláves jsou unikátní v rámci této komponenty. Stisknutí kláves je pak registrováno přes **Unity Input**.

4.3.2 Prototypy

Prototyp objektu i s jeho přiřazenými komponentami uchováváme ve třídě **ItemData**. Tato třída navíc obsahuje jméno prototypu a skupiny získané ze společné komponenty. Dále také unikátní identifikátor, prázdnou instanci typu **GameObject**, a pokud existuje, tak i náhledový obrázek.

Jméno prototypu musí být unikátní, jelikož ho využívámě ke generování identifikátoru za pomocí funkce *Hash* implementované na typu *string*. Unikátnost jména využíváme i na jiných místech projektu, například v kontroleru kolizí na identifikování objektů střetu nebo pro funkci vyhledávání prototypu podle jména v panelu pro výběr prototypů (3.3.3). Toto vyhledávání je uskutečněno přes výpočet **Levenstheinovy vzdálenosti** mezi hledaným jménem a jmény všech prvků (5.0.4).

V tomto panelu využíváme i náhledový obrázek, pomocí něhož umožníme uživateli vizuální identifikaci prototypu. Pokud prvek nemá náhledový obrázek, tak se v panelu zobrazí pouze jeho jméno.

V prostředí **Unity** jsou všechny herní objekty typu **GameObject** [36], a tedy všechny námi vytvořené objekty budou stejného typu. K vytváření používáme metodu **Instantiate**, které předáme tento prázdný objekt. Této kopii pak nastavíme všechny komponenty uložené v prototypu způsoby popsanými výše.

4.3.3 Správa prototypů

Abychom se mohli na vytvořené prototypy odkazovat a používat je v rámci projektu, ukládáme je do manažeru prototypů implementovaného ve třídě **ItemManager** definované jako *singleton*. Ta udržuje přehled o všech prvcích a jejich skupinách a umožňuje nám například získat všechny prvky patřící do stejné skupiny. K tomu používáme datové struktury typu *slovník*, do kterých vkládáme jako klíče identifikátory prototypů a jména skupin. Dále v rámci manažeru ukládáme mapování jmen prototypů na jejich identifikátory. V následujícím přehledu si shrneme nejdůležitější metody, které tento manažer implementuje.

- **Přidání prototypu** - Přidá nový prototyp do jednotlivých datových struktur manažeru.
- **Úprava existujícího prototypu** - Najde existující prototyp a odstraní z pracovní plochy všechny herní objekty z něj vytvořené. Ty pak nahradí objekty vytvořenými z nového prototypu, který následně nahradí původní. Tuto funkcionalitu využíváme v případě editování existujícího prototypu, který probereme v následující sekci.
- **Odstranění prototypu** - Odstraní prototyp i všechny objekty z něj vytvořené.

Tento manažer nám tedy usnadňuje práci s prototypy. Součástí tohoto manažeru je pak i proměnná obsahující odkaz na aktuální vybraný prototyp, který bude využívat například manipulační funkce pro vkládání objektů na pracovní plochu (2.1).

4.3.4 Editace existujících prototypů

Součástí funkcí pro práci s prototypy je i možnost upravování vytvořených prototypů. Tu implementujeme tak, že načteme existující prototyp a jeho komponenty do okna v editoru, a umožníme je tak uživateli upravit. Poté, co uživatel provede požadované úpravy a potvrdí ukončení editace, vytvoříme z upravených komponent nový prototyp. Ten pak za pomoci zmíněné funkce manažeru prototypů nahradíme za starý prototyp.

4.4 Akce a jejich vytváření

V této sekci rozebereme implementaci kompilátoru našeho vytvořeného jazyka (2.4.1), jednotlivé funkce, které jeho protřenictvím můžeme volat, a jednotlivé objekty s ním související. V souvilosti s tím se také zaměříme na funkcionality prostředí, ve kterém bude kód vyvíjen.

4.4.1 Zpracování textu

Kód, který uživatel vytvoří, náš kompiler obdrží v podobě obyčejného textu. Analýza tohoto textu je pak provedena třídami syntaktické analýzy, které text rozdělí podle řádku, a na tyto řádky následně uplatní *Regex Pattern Matching*. Ten nám tyto řádky umožní kategorizovat podle jenotlivých akcí jako jsou podmínky, cykly a přiřazování do proměnných. Jednotlivé vzory pak můžeme vidět na následující ukázce.

Program 13 Ukázka použitých regulárních výrazů

```
1       variable = @"((?<![^\s+*\/\-\(\)])(\w*[a-zA-Z]\w*\.)*(\w*[a-
            zA-Z]\w*)(?![^\s+*\/\-\(\)]))"
2       numericValue = "([0-9]+(?:\\.[0-9]*)?)"
3       keyWords = @"(^|\s)(if|while|elseif|else|fi|end|while)(\s|$)
            "
```

Na základě této identifikace můžeme rozhodnout, zda tyto řádky mají obsahovat nějaké aritmetické či logické výrazy. Pokud tedy obsahují takovýto výraz, spustíme na řádku další analýzu, která identifikuje jednotlivé členy těchto výrazů, tedy operandy a operátory. Na konci této analýzy projdeme všechny nalezené členy a pokusíme se mezi těmito jednotlivými výrazy identifikovat volání funkcí a jejich argumenty. Všechny nalezené členy pak reprezentujeme třídami dědícími od rodičovské třídy **Item** a uložíme je do výsledného typu **ActionItem** reprezentujícího zpracovaný řádek.

Třída **ExpressionItem** slouží k reprezentaci běžných členů a ukládá indentifikovaný typ členu, textovou hodnotu, případně také převedenou konkrétní hodnotu, jako je číslo či bool. Pro nalezené funkce pak používáme třídu **MethodItem**, která v sobě ukládá jméno funkce a seznam jejích argumentů, které jsou opět typu **Item**.

4.4.2 Kompilace

Po zpracování jednotlivých členů začneme stavět syntaktický strom, jehož vnitřní uzly budou námi definované podporované operace (2.4.1). Pro jednotlivé výrazy pak použijeme třídu **ExpressionTree**, která z jednotlivých členů výrazu vytvoří příslušný strom. Jednotlivé členy pak budou v tomto stromě reprezentovány následujícími třídami:

- **Operator** - slouží k reprezentaci následujících unárních a binárních operátorů:

Unární operátory	**Význam**
!	Logické NOT
-	Unární mínus, tedy aritmetická negace

Tabulka 4.5 Seznam podporovaných unárních operátorů

Binární operátory	**Význam**
*, /, %	Násobení, Dělení, Modulo
+, -	Sčítání, Odečítání
«, »	Posunutí doleva a doprava
<, <=, =>, >	Operátory porovnání
==, !=	Operátory rovnosti
&	Bitový AND
^	Bitový XOR
|	Bitový OR
&&	Logický AND
||	Logický OR

Tabulka 4.6 Seznam podporovaných binárních operátorů

- **Operand** - reprezentuje hodnotové typy bool, string a float.

Stavba stromu výrazu

Při stavbě stromu výrazu se řídíme náledujícím algoritmem, respektujícím priority jednotlivých operátorů. Tyto priority pak odpovídají pořadí operátorů v předchozí tabulce.

Jednotlivé členy tohoto stromu jsou třídy, dědící od abstraktní rodičovské třídy **TreeNode** vyžadující od svých členů implementaci metody **Evaluate**, jež vrací instanci třídy **Operand**. Struktura tohoto stromu pak vypadá následovně.

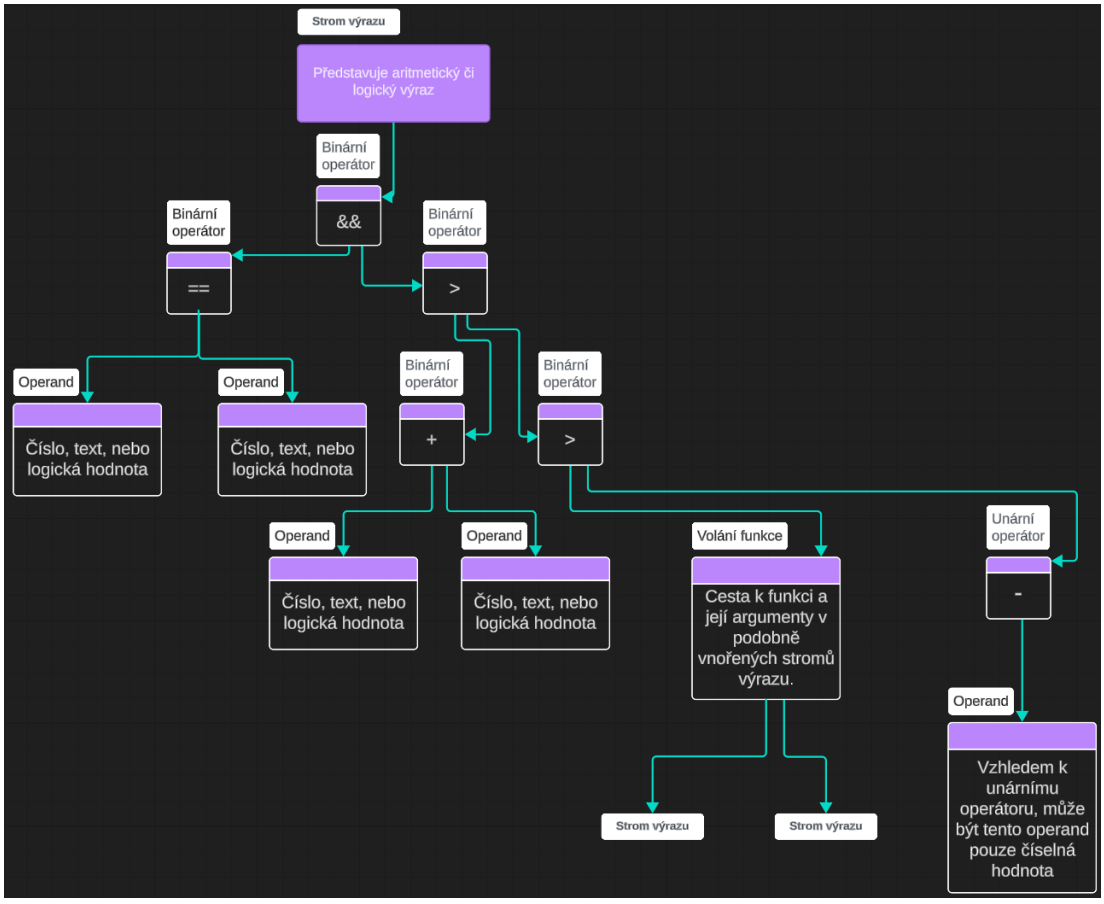
V listech má třídy:

- **OperandNode** - Třída, která při zavolání metody **Evaluate** vrací jedinou uloženou hodnotu typu **Operand**.
- **VariableNode** - Reprezentuje vytvořenou proměnnou či vlastnost připojené třídy (4.4.3) v rámci aktuálního kontextu kódu. Jako výsledek volání **Evaluate** pak tato třída vrací aktuální hodnotu, vlastnoti třídy nebo lokální či globální proměnné, pokud existují.

a ve vnitřních vrcholech:

- **MethodNode** - Obsahuje seznam tříd **TreeNode** reprezentující argumenty této funkce a odkaz na příslušnou metodu. Tento odkaz přiřazujeme již v konstruktoru pomocí poskytnutého jména (4.4.3). V průběhu funkce **Evaluate** se pak tato metodá zavolá a její výsledek se vrací jako **Operand**.
- **OperationNode** - Tato třída reprezentuje binární operaci a obsahuje tedy dvojici instancí typu **TreeNode** a jednu instanci typu **Operator**.
- **UnaryNode** - Reprezentuje operaci unárního operátoru. Obsahuje jednu instanci typu **TreeNode** a jednu instanci **Operator** s unárním operátorem.

Na následujícím obrázku můžeme vidět graf sestaveného stromu a postup jeho vyhodnocení:



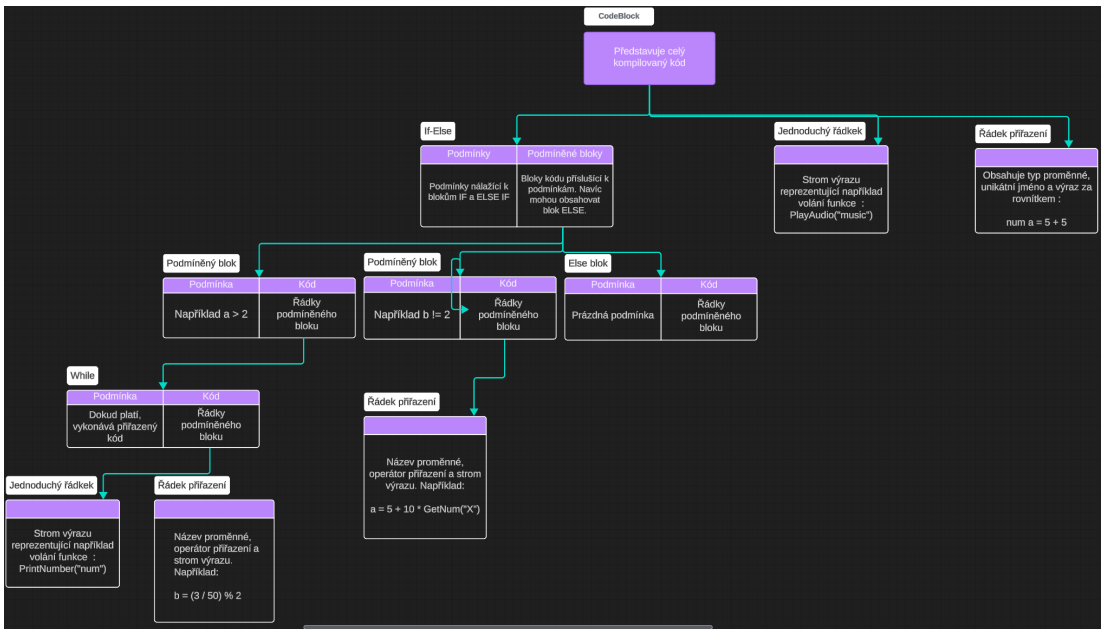
Obrázek 4.1 Ukázka stromu výrazu

Typ proměnné	Operátory
Bool	=
String	=, +=
Num	=, +=, -=, *=, /=

Tabulka 4.7 Seznam podporovaných operátorů přiřazení

Stavba syntaktického stromu

Zpracování jednotlivých výrazů máme tedy již vyřešené. Nyní se podíváme na to, jak stavíme syntaktický strom a jaké třídy k tomu používáme. Při zpracovávání kódu postupujeme odshora dolů, a postupně přidáváme prvky do stromu. Každé zanoření, například do podmíněného bloku či cyklu, pak reprezentuje hladinu stromu, jak můžeme vidět na následujícím příkladu:



Obrázek 4.2 Ukázka syntaktického stromu

Jednotlivé prvky stromu pak implementujeme následujícími třídami:

- **AssignLine** - Představuje řádek s přiřazením do proměnné. Obsahuje jméno přoměnné, do které přiřazujeme vybudovaný strom výrazu a jeden z příslušných operátorů přiřazení. Jednotlivé typy proměnných pak podporují různé operátory:
- **SimpleLine** - Jednoduchý řádek bez přiřazení či jakýchkoliv řídících segmentů. Typické použití je například volání metod.
- **IfElseLine** - Řídící segment reprezentující podmíněný kód (2.4.1). Obsahuje seznam podmínek a příslušných bloků kódu představující **if** a **else if** podmínky, a pokud je specifikovaný, tak i **else** blok. Jednotlivé bloky jsou pak reprezentovány odkazy na vnořené objekty vyjmenovaných tříd.

- **WhileLine** - Představuje podmíněný blok kódu obsahující podmínku cyklu a maximální délku cyklení, která je v základu nastavená na deset tisíc iterací (2.4.3). Přesažení této hodnoty pak vyvolá výjmku upozorňující na nekonečný cyklus.

Kořen tohoto stromu je pak implementován třídou **CodeBlock**.

4.4.3 Přidání závislostí a globálních proměnných

Jak jsme zmínili u stavby stromu výrazů, součástí těchto výrazů je volání existujících metod podle jména a přiřazování do lokálních či globálních proměnných. Všechny tyto prvky si pak udržujeme v instanci třídy **CodeContext** reprezentující kontext kódu.

Lokální proměnné

Lokální proměnné definujeme v rámci vykonávání kompilovaného kódu. Při každé definici takovéto proměnné vytvoříme v kontextu kódu záznam do datové struktury typu *slovník*, kde klíčem je jméno proměnné a hodnotou instance typu **Operand**. Z povahy ukládaných dat je tedy jasné, že jména lokálních proměnných musí být unikátní. Jakmile jednou lokální proměnnou vytvoříme, tak jakákoliv další přiřazení už pouze mění uloženou hodnotu v kontextu. Data těchto proměnných se reinicalizují s každým spuštěním kódu.

Globální proměnné

V některých případech se nám však hodí, aby proměnné přetrvaly i mezi jednotlivými spuštěními. Z tohoto důvodu implementujeme v editoru kódu možnost definovat takzvané globální proměnné. Při jejich vytváření specifikujeme typ, jméno a základní hodnotu. Stejně tak jako lokální proměnné jsou i globální ukládány v datové struktuře typu *slovník* se stejnými hodnotami. Unikátnost jmen je tedy stejná. U globálních proměnných pak ale navíc platí, že pokud mají stejné jméno jako vytvořená lokální proměnná, tak je tato lokální proměnná zastíní.

Vlatnosti a metody

Vlastnosti a metody umožňujeme nastavovat na námi specifikovaných objektech tak, jak jsme již popsali (3.3.7). Tyto jednotlivé prvky jsou pak implementovány třídami, které dědí od rodičovské třídy **EnviromentObject**. Tyto třídy implementují metody označené námi vytvořeným *atributem* **CodeEditorAttribute**, jehož součástí je popis jejich chování a vstupních parametrů. Takto označené metody je pak možné volat z našeho jazyka.

Tyto třídy přidáváme skrze funcionalitu implementovanou v editoru kódu. Každé z tříd pak specifikujeme unikátní jméno, jímž se na ni budeme odkazovat a se kterým bude i uložena v kontextu kódu. Na následujícím příkladě můžeme vidět, jak se pak v našem jazyce můžeme odkazovat na hypotetickou metodu.

Program 14 Ukázka odkazování na metodu připojené třídy **Else**

```
1    odkazovaneJmeno.vlastnost1.vlastnost2.metoda(argument1 ,..)
```

Jak můžeme vidět, tyto odkazy tvoří cestu skrze jednotlivé objekty až k finální metodě. Jako první tedy uvádíme jméno, které jsme třídě přiřadili. Dále se pak skrze sérii vlastností dostaneme až k metodě, kterou chceme zavolat. Procházení této cesty a hledání metod je implementováno pomocí *reflexe* na jednotlivých objektech cesty. Reflexe nám zároveň umožňuje specifikovat hledané objekty podle jejich atributů, což nám umožňuje hledat objekty pouze s naším atributem.

Tímto způsobem můžeme také odkazovat přímo na vlastnost. V případě odkazování na metody pak ještě hledáme takové, které odpovídají počtem a typy argumentů.

4.4.4 Funkce editoru kódu

V rámci funkcí editoru kódu byly implementovány následující funkce usnadňující jeho používání:

Zvýraznění syntaxe

Zvýraznění syntaxe funguje na principu označování klíčových slov našeho jazyka v textu kódu. Tato slova pak hledáme stejným způsobem jako při analýze kódu, tedy pomocí regulárních výrazů. Označení je pak provedeno změnou barvy určitých klíčových slov podle využití. Klíčová slova označující typ definované proměnné, tedy num, bool a string, označíme růžovou barvou. Ostatní klíčová slova pak značíme tmavě modrou.

Obrázek 4.3 Ukázka zvýraznění syntaxe

Jelikož je tato funkce časově náročná, není vhodné, aby se prováděla při každém volání metody *Update*. Označování tedy provádíme jednou za specifikovaný časový úsek, v základním nastavení jednou za vteřinu.

IntelliSence

Jak jsme již rozvedli v předchozí sekci, v rámci našeho kódu můžeme volat metody na připojených třídách. Aby ale uživatel věděl, jaké metody může na aktuálním objektu cesty volat, implementovali jsme mezi funkce editoru interaktivní nápovědu **IntelliSence**, která zobrazí všechny dostupné metody s naším atributem na aktuálním objektu cesty. Toho je opět dosaženo pomocí *reflexe*. Jelikož

jednotlivé atributy v sobě obsahují stručný popis metody (4.4.3), umožníme jeho zobrazení v rámci této nápovědy.

Tato nápověda se pak zobrazí vždy, když uživatel do edioru napíše cestu, a posledním stisknutým znakem je tečka. Ta nám totiž naznačuje, že uživatel má v plánu zahrnout do cesty další vlastnost či metodu nacházející se na posledním objektu již vytvořené cesty. Pokud tečka není částí žádné existující cesty, nápověda se nezobrazí.

4.5 Řešení chyb a vypisování hlášek

V konkrétních částech naší implentace se můžeme potkat se stavy, které nejsme schopni vyřešit. Například pokud načítaný *asset* neexistuje, nebo pokud došlo v průběhu načítání ke ztrátě spojení. Takovouto chybu musíme být schopní ošetřit, ale zároveň musíme inforomovat uživatele o vzniklém problému. Z tohoto důvodu byl v rámci našeho projektu implementován centrální systém registrující chybové hlášky a předávající informace o těchto chybách přihlášeným *posluchačům*. Těmito posluchači pak myslíme odkazy na metody, které na nastalou chybu zareagují. Příklad takovéto metody řídí výpis do panelu výstupní konzole. Nově příchozí zprávy jsou přidávány do fronty, ze které odebíráme po určitém časovém úseku. Odebraná zpráva je pak aktuální zprávou, dokud neuběhne zmíněný časový úsek. Poté je zpráva nahrazena další, pokud existuje.

Tento systém je implementovaný třídou **ErrorOutputManager**, která je definovaná jako *singleton* a pro kterou si nyní uvedeme rámcový přehled jejích metod:

H **Přidání posluchačů přidaných zpráv** - Metody pro přídání posluchače zpráv můžeme rozdělit podle toho, zda chceme přidat posluchače reagujícího na přidání zprávy do fronty, nebo až ve chvíli, kdy ji z fronty odebereme. Jednotlivým posluchačům můžeme také nastavit autory zpráv, na které mají reagovat. Pokud není autor u posluchače uveden, přijímá všechny zprávy.

H **Přidání posluchačů ničení zpráv** - V případech panelů vypisujících tyto zprávy se nám hodí přidat posluchače reagující na zničení aktuální zprávy. To nám umožnuje text těchto panelů vymazat při zničení zprávy.

H **Přidání zprávy** - Tato metoda přidává novou zprávu do již zmíněné fronty. Zároveň umožňuje specifikovat autora těchto zpráv.

4.6 Funkce pro práce s daty

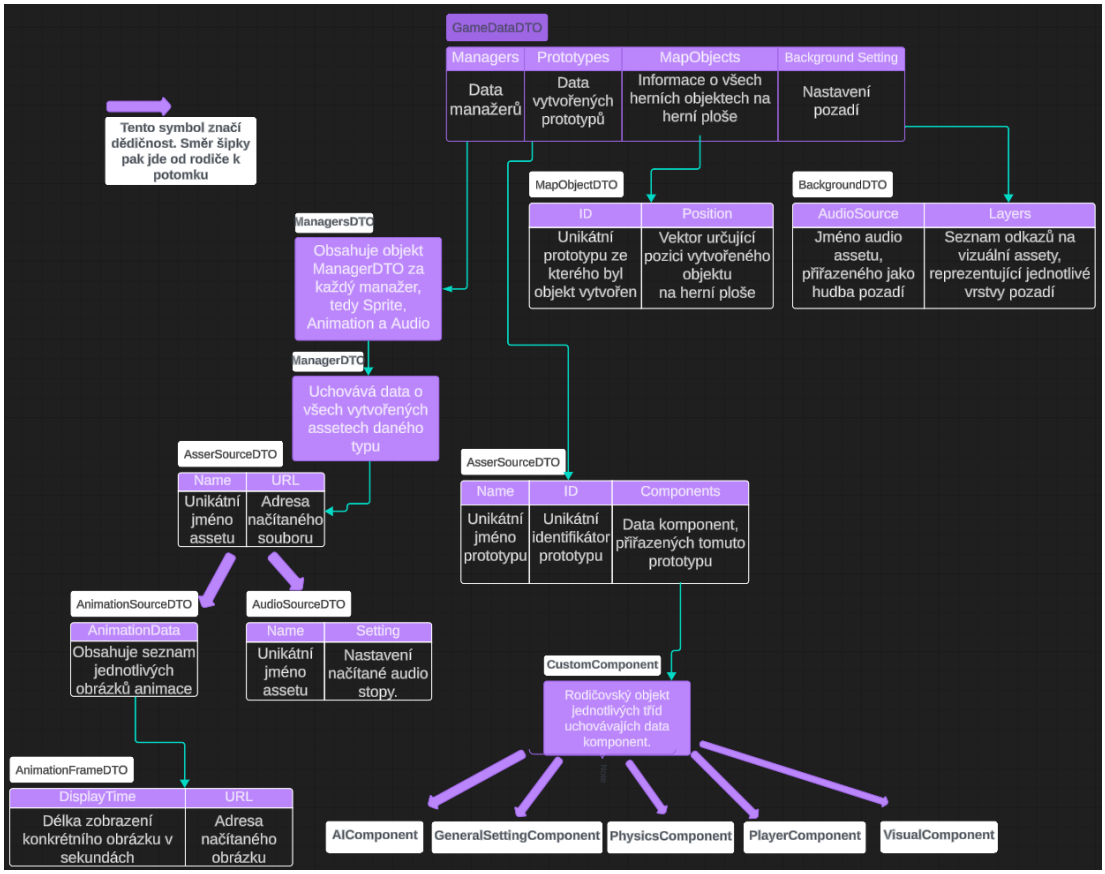
V předchozí kapitole jsme vysvětlili důležitost funkcí pro práci s daty (2.1). Zde se zaměříme na konkrétní implementaci.

4.6.1 Ukládání a načítání objektů

Ukládání aktuálního stavu projektu není pouze o uložení herních objektů na ploše, ale i o ukládání jednotlivých prototypů a zdrojů k načteným *assetům*, abychom byli schopni projekt při načtení obnovit do původního stavu. Pro zápis

těchto dat do souborů a jejich zpětné načtení využíváme knihovnu **Newtonsoft.JSON**. Abychom byli schopni tuto serialiazaci jednoduše provést, vytvořili jsme centrální objekt **GameDataDTO**, do kterého uložíme jednotlivé části našeho projektu. Metody odpovědné za načítání a ukládání projektu z tohoto objektu jsou implementovány ve třídě **GameDataSerializer**.

Strukturu tohoto centrálního objektu můžeme pozorovat na následujícím grafu:



Obrázek 4.4 Ukázka struktury objektu pro ukládání dat

4.7 Ostatní části projektu

4.7.1 Ovládání pozadí

Další částí projektu, jejíž implementaci si rozebereme, bude třída **BackgroundController** ovlivňující pozadí pracovní plochy a hudbu s ním přehrávanou.

Nastavování vizuální stránky pozadí je realizováno pomocí přidávání překrývajících se vrstev. Těmto vrstvám umožňujeme nastavení textur či animací podle uživatelova výběru. Překrývající vrstvy zároveň umožňují vytváření různých efektů, například pohybující se mraky zakrývající další prvky pozadí.

K objektu pozadí je pak přidána i komponenta **AudioSource** pro přehrávání zvukových stop. Zvuková stopa, kterou pozadí nastavíme, se pak bude přehrávat ve smyčce, jelikož tato stopa slouží jako hudba celé hry.

4.7.2 Mód ladění

V rámci implementace módu ladění byly kontrolerům komponent přidávaných v rámci vytváření herních objektů z příslušných prototypů (4.3) přidány metody pro jejich aktivaci a deaktivaci. Jejich aktivace a deaktivace pak závisí na tom, jestli je editor v režimu ladění nebo ne. Kdyby byly jednotlivé komponenty aktivní i mimo režim ladění, mohlo by se například stát, že objekty s přiřazenou fyzikální komponentou by propadávaly pracovní plochou díky aktivnímu gravitačnímu zrychlení.

Úvod stavu projektu do stavu před laděním je pak realizován uložením stavu projektu a opětovným načtením po jeho skončení.

4.7.3 Hledání cesty

Součástí funkcí editoru je i sada nástrojů pro testování, které mají za cíl ověřit, zda existuje cesta z pozice označeného objektu na nejbližší pozici objektu cíle, pokud existuje. K hledání těchto cest byl implementován algoritmus **A-star**, který prohledává stavový prostor, kde jsou jednotlivé pozice určené buňkami překrývající mřížky (4.1).

Algoritmus začne na pozici označeného objektu a pomocí jednotlivých pohybových akcí se snaží najít všechny dosažitelné pozice. Na těchto nalezených pozicích pak aplikuje stejný postup, dokud nenalezne cíl, nebo nevyčerpá nalezené pozice. V takovém případě pak vypíše hlášku o neexistující cestě.

Každá akce má přiřazenu cenu provedení, která určuje prioritu zpracování nalezených pozic. Nové pozice jsou hledány pomocí metod, implementovaných ve třídách pohybových funkcí.

5 Vzorové hry

Nyní provedeme rozbor námi vytvořených adaptací vzorových her z kapitoly analýzy (2.3), kde se zaměříme se na jednotlivé klíčové komponenty her a ukážeme, zda se je povedlo implementovat. Součástí bude i ukázka z každé adaptace. Abychom předešli problémům s autorskými právy, použili jsme pro tvorbu her kombinaci volně dostupných *assetů* z **Unity Asset Store** [37] [38] [39] [40] [41].

Vytvořené hry je možné nalézt ve složce **Maps**, umístěné mezi soubory editoru.

5.0.1 FlappyBird

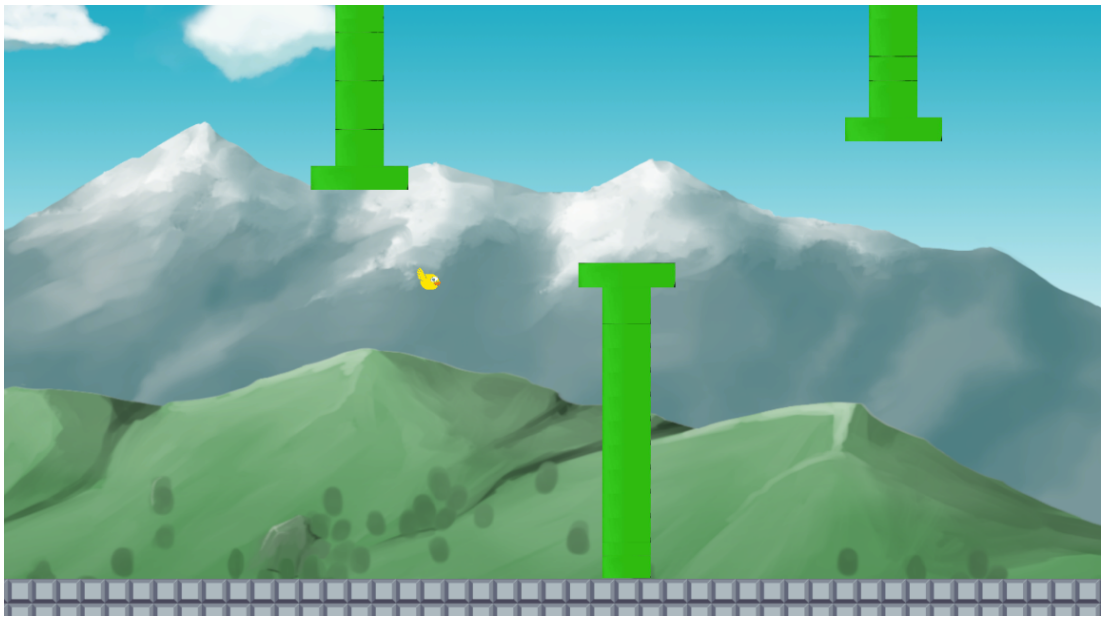
Jak jsme ukázali v rozboru originální hry, základním principem této hry je vyhnout se kolizím s překážkami pouze za pomocí změny výšky hráčské postavy. Pokud dojde ke kolizi, hra je ukončena. Tyto kolize implementujeme pomocí komponenty **CollisionBox** přidané k prototypu hráče, které jsme přidali akci reagující na kolizi s překážkou (3.3.3).

Program 15 Ukázka akce pro kolizi s překážkou

```
1     objectControl.Kill (false, false)
```

Ta při svém spuštění přehraje zvuk kolize a vyvolá proces ukončení hry neúspěchem.

Pohyb postavy je implementován pomocí akce skoku s pevně nastavenou intenzitou. Ta má v hráčské komponentě přiřazený stisk levého tlačítka myši jako spouštěcí akci (3.3.3). Ta zároveň spustí animaci mávnutí křídel a s ní spojený audio efekt.



Obrázek 5.1 Ukázka adaptace hry **Flappy Bird**

Tím jsme splnili implementaci všech požadovaných komponent.

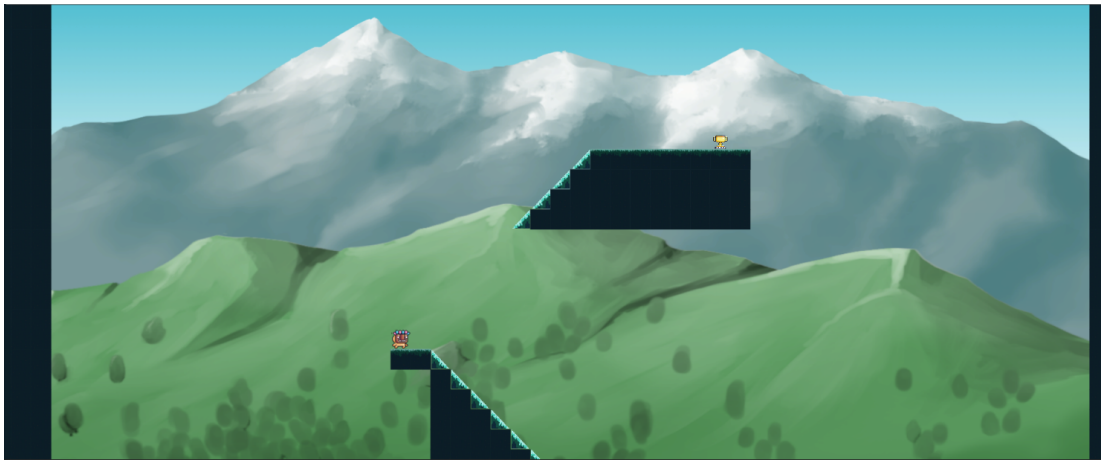
5.0.2 Jumpking

Požadované komponenty k vytvoření této hry se příliš neliší od předchozí adaptace, a proto se v tomto rozboru zaměříme na jejich rozdílné vlastnosti.

Místo skoku s pevnou intenzitou, umožňujeme hráči chůzi a skok s proměnlivou intenzitou (3.3.3). Tyto akce jsou na rozdíl od předchozího příkladu nastaveny tak, aby se spustily pouze v případě, že hráč stojí na platformě. Nabíjení skoku je spuštěno stisky kláves **Q** a **E** pro skok vlevo a vpravo. Ekvivaletně k tomu klávesy **A** a **D** zajišťují chůzi.

Aby byl hráčův postup náročnější, implementuje tato hra i souvislé šikmé plochy. Po těch pak hráč může sklouznout i o několik úrovní níž, například když netrefí cílenou platformu.

Pokud hráč dosáhne cílového objektu, je spuštěna akce ukončení hry úspěchem. Tento cílový objekt pak můžeme vidět na následující ukázce jako zlatý pohárek.



Obrázek 5.2 Ukázka adaptace hry **Jump King**

5.0.3 Super Mario

Poslední vzorovou hrou je adaptace hry **Mario**, která je pravděpodobně nejkomplexnější adaptovanou hrou.

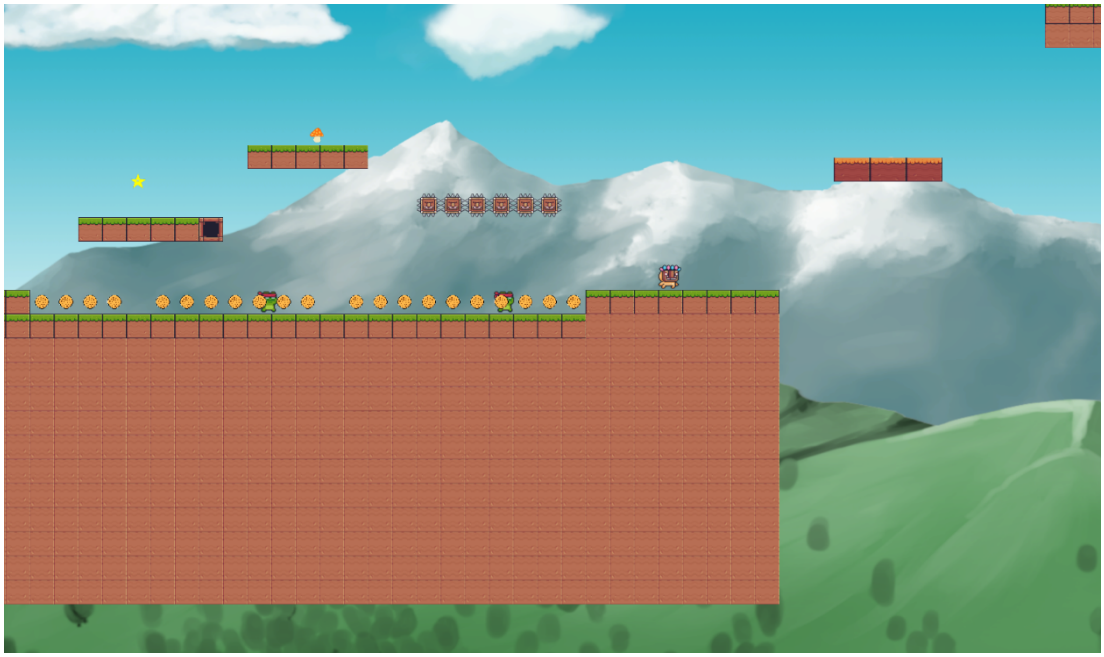
Hráčská postava umožňuje pohyb pomocí chůze a skoku s pevnou intenzitou (3.3.3). V tomto případě však umožňujeme vykonávat pohyb chůze i v případě, že se hráč pohybuje v prostoru. Stejně jako v předchozích adaptacích jsou součástí pohybů animace a audioefekty.

Klávesami **A** a **D** ovládáme chůzi vlevo a vpravo. Pro provedení skoku v těchto směrech používáme klávesy **Q** a **E**

Součástí naší adaptace jsou i nepřátelé, pohybující se po náhodné trase. Pokud se postava hráče dotkne postavy nepřítele jinde než na vršku, hráč ztratí část bodů zdraví. V opačném případě je nepřítel eliminován.

Dalšími speciálními objekty jsou mince, které hráči přidávají skóre, houby doplňující zdraví a hvězdy. Ty poskytují hráči dočasnou nezranitelnost.

Hra navíc obsahuje pohybující se platformy, pomocí kterých se hráč může dostat na nedosažitelné pozice.



Obrázek 5.3 Ukázka adaptace hry **Super Mario**

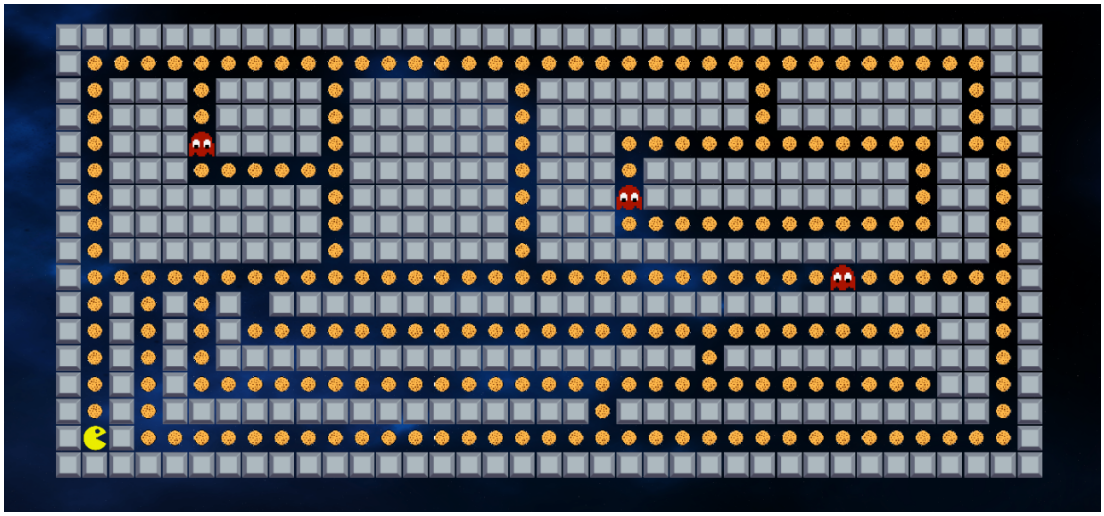
Tyto operace opět implementujeme pomocí vytvořených akcí pro kolize s vybranou skupinou prvků(3.3.3).

Program 16 Ukázka akce pro kolizi s objektem houby

```
1  ob.Score += 30
2  string score = "Score :" + conv.NumToString(ob.Score)
3  sound.SetAudio("coin", true)
4  output.Print(score)
```

5.0.4 Pacman

Tento editor můžeme použít i ke tvorbě jiných druhů 2D her, jak demonstrujeme na následující ukázce z adaptace hry **Pacman**.



Obrázek 5.4 Ukázka ze hry **Pacman**

Pro replikaci pohybu hráčské postavy z originální hry používáme implementovanou pohybovou akci letu (2.4.5). Provedení pohybu umožňujeme pomocí kláves **W**, **S**, **A**, **D** pro směry nahoru, dolu, vlevo a vpravo.

Dále přidáváme fyzikální komponentu, které nastavíme nulové gravitační zrychlení pro všechny všechny nepřátele a hráče.

Stejně jako u hry **Super Mario** implementujeme objekty nepřátel, které můžeme v ukázce vidět jako červená monstra.

Na mapě se vyskytují také objekty sušenek, které hráči zvedají skóre. Ve chvíli kdy hráč posbírá všechny sušenky, hra končí budťo výhrou, nebo načtením další herní úrovně.

Závěr

V rámci této práce jsme provedli rešerši existujících herních editorů. Naprostá většina znich je buďto zpoplatněna, anebo klade příliš velké nároky na jejich uživatele. Na základě této rešerše jsme definovali požadavky na vytvoření vlastního herního editoru. V další kapitole jsme tyto požadavky podrobili důkladné analýze, dále jsme je upřesnili a doplnili.

Podle těchto požadavků jsme naimplementovali vlastní herní editor. Ten poskytuje možnost zjednodušené tvorby her a upravování jejich komponent. Tyto hry můžeme ukládat, načítat, a díky možnosti stahování assetů pomocí **HTTP protokolu** i sdílet.

Rozbor adaptací vzorových her v rámci rešerše nám nastínil, jaké hry jsme schopni pomocí našeho editoru vytvořit. Hierarchická a dokumentovaná struktura jeho kódu navíc nabízí možnost rozšíření o nové funkce. Jako příklad můžeme uvést třídy s metodami, které lze volat pomocí námi nově navrženého skriptovacího jazyka, nebo přidání zbraňových funkcí.

Minimalismus tohoto skriptovacího jazyka poskytuje možnost snadnějšího osvojení základních konstrukcí programovacích jazyků, jako například práci s proměnnými a řídícími bloky. Použití zmíněných tříd metod navíc snižuje nároky na znalosti uživatele, a tím celkově usnadňuje tvorbu her.

Výsledkem je funkční herní editor, který je vytvořen s myšlenkou kritérií jednoduchosti a rychlého osvojení. K editoru jsme sepsali podrobnou uživatelskou a vývojovou dokumentaci. Jeho funkcionalitu jsme demonstrovali na adaptacích známých her FlappyBird, Jumpking, Super Mario a Pacman.

Tím jsme splnili stanovené cíle práce.

Literatura

1. Wikipedia, The Free Encyclopedia. *Best-selling video games in the United States by year* [online]. [cit. 2024-04-18]. Dostupné z: https://en.wikipedia.org/wiki/Best-selling_video_games_in_the_United_States_by_year.
2. Sirani, Jordan. *The 10 Best-Selling Video Games of All Time* [online]. [cit. 2024-03-21]. Dostupné z: https://www.ign.com/articles/best-selling-video-games-of-all-time-grand-theft-auto-minecraft-tetris.
3. Wikipedia, The Free Encyclopedia. *AAA (video game industry)* [online]. [cit. 2024-04-10]. Dostupné z: https://en.wikipedia.org/wiki/AAA_(video_game_industry).
4. Wikipedia, The Free Encyclopedia. *Indie game* [online]. [cit. 2024-04-10]. Dostupné z: https://en.wikipedia.org/wiki/Indie_game.
5. Clement, J. *The 10 Best-Selling Video Games of All Time* [online]. [cit. 2024-01-17]. Dostupné z: https://www.statista.com/statistics/1411839/number-games-released-steam-developer-type.
6. Herd, Jules. *The Global Surge Of Independent Games Development Studios* [online]. [cit. 2023-08-21]. Dostupné z: https://www.forbes.com/sites/forbesagencycouncil/2023/08/21/the-global-surge-of-independent-games-development-studios.
7. Wikipedia, The Free Encyclopedia. *Construct (game engine)* [online]. [cit. 2024-04-03]. Dostupné z: https://en.wikipedia.org/wiki/Construct_(game_engine).
8. Construct. *Make Games with Construct 3* [online]. [cit. 2024-04-23]. Dostupné z: https://www.construct.net/en.
9. Construct. *Buy Construct 3* [online]. [cit. 2024-04-23]. Dostupné z: https://www.construct.net/en/make-games/buy-construct.
10. Famularo, Jessica. *The best game engines for making your own 2D indie game* [online]. [cit. 2017-10-28]. Dostupné z: https://www.pcgamer.com/the-best-2d-game-engines.
11. GameMaker. *Let's make a game!* [online]. [cit. 2024-04-23]. Dostupné z: https://gamemaker.io/en/get.
12. GameFromScratch. *Game Engine Popularity in 2024* [online]. [cit. 2024-01-29]. Dostupné z: https://gamefromscratch.com/game-engine-popularity-in-2024/.
13. Marie Dealessandri, Deputy Editor. *What is the best game engine: is Unity right for you?* [online]. [cit. 2024-05-05]. Dostupné z: https://www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you.
14. Steam. *Games Made With Unity* [online]. [cit. 2024-04-23]. Dostupné z: hhttps://store.steampowered.com/curator/39750107-Games-Made-With-Unity.

15. Unity. *Plans and pricing.* [online]. [cit. 2024-04-23]. Dostupné z: https://unity.com/products.

16. Learn, Unity. *Learning Pathways* [online]. [cit. 2024-04-23]. Dostupné z: https://learn.unity.com.

17. Steam. *Hypnospace Outlaw* [online]. [cit. 2019-03-12]. Dostupné z: https://store.steampowered.com/app/844590/Hypnospace_Outlaw.

18. Mitra, Ritwik. *The Global Surge Of Independent Games Development Studios* [online]. [cit. 2023-12-08]. Dostupné z: https://gamerant.com/best-games-made-using-gamemaker-engine.

19. Steam. *Undertale* [online]. [cit. 2015-09-15]. Dostupné z: https://store.steampowered.com/app/391540/Undertale.

20. Steam. *Cuphead* [online]. [cit. 2024-04-23]. Dostupné z: https://store.steampowered.com/app/268910/Cuphead.

21. Drake, Jeff. *24 Great Games That Use The Unity Game Engine* [online]. [cit. 2023-10-30]. Dostupné z: https://www.thegamer.com/unity-game-engine-great-games.

22. Wikipedia, The Free Encyclopedia. *Platformer* [online]. [cit. 2024-04-18]. Dostupné z: https://en.wikipedia.org/wiki/Platformer.

23. dictionary, Cambridge. *Platform game* [online]. [cit. 2024-04-18]. Dostupné z: https://dictionary.cambridge.org/dictionary/english/platform-game.

24. Wikipedia, The Free Encyclopedia. *Flappy Bird* [online]. [cit. 2024-04-20]. Dostupné z: https://en.wikipedia.org/wiki/Flappy_Bird.

25. bird, Flappy. *Flappy Bird* [online]. [cit. 2024-04-23]. Dostupné z: https://flappy-bird.co.

26. Nexile. *Jump King* [online]. [cit. 2024-05-05]. Dostupné z: https://store.steampowered.com/app/1061090/Jump_King/?curator_clanid=44376191.

27. Bullet, Code. *Jump King* [online]. [cit. 2024-04-23]. Dostupné z: https://code-bullet.github.io/Jump-King/.

28. Nintendo. *Mario history - Mario through the years* [online]. [cit. 2024-03-18]. Dostupné z: https://mario.nintendo.com/history/.

29. SuperMario-game. *SuperMario-game* [online]. [cit. 2024-04-23]. Dostupné z: https://supermario-game.com.

30. G, Gokul P. *Top 15 Reasons to Use Unity 3D for Game Development* [online]. [cit. 2023-12-15]. Dostupné z: https://www.linkedin.com/pulse/top-15-reasons-use-unity-3d-game-development-jklucent-cvhjc.

31. KartInnka. *Buttons Set* [online]. [cit. 2024-06-05]. Dostupné z: https://assetstore.unity.com/packages/2d/gui/buttons-set-211824.

32. SiLena$_A$RT. *2D Pixel art pack - Rousette* [online]. [cit. 2024-06-05]. Dostupné z: https://assetstore.unity.com/packages/2d/characters/2d-pixel-art-pack-rousette-167698.

33. YASIRKULA. *Runtime File Browser* [online]. [cit. 2024-06-05]. Dostupné z: https://assetstore.unity.com/packages/tools/gui/runtime-file-browser-113006.

34. SIMPLEART. *Foggy Mountains Parallax Background* [online]. [cit. 2024-06-05]. Dostupné z: https://assetstore.unity.com/packages/2d/textures-materials/foggy-mountains-parallax-background-142516.

35. DOCUMENTATION, Unity. *Image* [online]. [cit. 2019-01-17]. Dostupné z: https://docs.unity3d.com/2018.2/Documentation/ScriptReference/UI.Image.html.

36. DOCUMENTATION, Unity. *GameObject* [online]. [cit. 2024-04-29]. Dostupné z: https://docs.unity3d.com/ScriptReference/GameObject.html.

37. ASSETS, Super Brutal. *Painted HQ 2D Forest Medieval Background* [online]. [cit. 2024-06-05]. Dostupné z: https://assetstore.unity.com/packages/2d/environments/painted-hq-2d-forest-medieval-background-97738.

38. FROG, Pixel. *Pixel Adventure 1* [online]. [cit. 2024-06-05]. Dostupné z: https://assetstore.unity.com/packages/2d/characters/pixel-adventure-1-155360.

39. LITE, Dynamic Space Background. *DinV Studio* [online]. [cit. 2024-06-05]. Dostupné z: https://assetstore.unity.com/packages/2d/textures-materials/dynamic-space-background-lite-104606.

40. DUSTYROOM. *FREE Casual Game SFX Pack* [online]. [cit. 2024-06-05]. Dostupné z: https://assetstore.unity.com/packages/audio/sound-fx/free-casual-game-sfx-pack-54116.

41. MIANENCZ. *Triadic Ascension* [online]. [cit. 2024-06-05]. Dostupné z: https://mianencz.itch.io/triadic-ascension.

42. RNDR. JAKUB YAGHOB, Ph.D. Ambiguous grammar, Disambiguation. *Compiler principles, Syntax analysis - an introduction*. 2024, roč. 1, č. 1, s. 6–7.

43. CORNER, C#. *C Delegate: Everything You Need To Know About Delegate In C* [online]. [cit. 2023-12-08]. Dostupné z: https://www.c-sharpcorner.com/UploadFile/puranindia/C-Sharp-net-delegates-and-events.

44. VODSLOŇ, František. *Vyhodnocování podobnosti zdrojových textů* [online]. [cit. 2007-05-30]. Dostupné z: https://www1.cuni.cz/~obo/vyuka/projekty/vodslon-ukazkova-bc.pdf.

45. BALA, Gopal C. *C Reflection With Code Example* [online]. [cit. 2023-09-21]. Dostupné z: https://www.c-sharpcorner.com/UploadFile/009ee3/reflection-in-C-Sharp.

46. WIELAND, Rob. *What Is A Role-Playing Game? RPG Types Explained* [online]. [cit. 2024-03-24]. Dostupné z: https://www.forbes.com/sites/technology/article/what-are-rpg-games.

47. MICROSOFT, Learn. *Stack<T> Třída* [online]. [cit. 2024]. Dostupné z: https://learn.microsoft.com/cs-cz/dotnet/api/system.collections.generic.stack-1?view=net-8.0.

48. UNITY. *MonoBehaviour* [online]. [cit. 2024-05-05]. Dostupné z: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html.

49. DOCUMENTATION, Unity. *User interface (UI)* [online]. [cit. 2024-04-29]. Dostupné z: https://docs.unity3d.com/Manual/UIToolkits.html.

50. DOCUMENTATION, Unity. *MonoBehaviour.Update()* [online]. [cit. 2024-04-20]. Dostupné z: https://docs.unity3d.com/ScriptReference/MonoBehaviourUpdate.html.

51. SHEKHAWAT, Sandeep Singh. *Virtual Method in C* [online]. [cit. 2024-01-09]. Dostupné z: https://docs.unity3d.com/2018.2/Documentation/ScriptReference/UI.Image.html.

52. MICROSOFT, Learn. *Dictionary<TKey,TValue> Třída* [online]. [cit. 2024]. Dostupné z: https://learn.microsoft.com/cs-cz/dotnet/api/system.collections.generic.dictionary-2?view=net-8.0.

Seznam obrázků

Seznam tabulek

Rejstřík

- **Assetem** budeme označovat objekty reprezentující textury/obrázky, animace a audio stopy.
- **API** je druh softwarového rozhraní programu, které umožňuje komunikaci s dalšími počítačovými programy.
- **Dangling Else problém** Specifický problém během kompilace a vytváření syntaktického stromu, kdy lze zkonstruovat dva možné syntaktické stromy pro if-then(-else) výraz. Tato nejednoznačnost musí být specificky ošetřena podle definovaného řešení (např: *vždy párovat else s nejbližším if*) [42]
- **Delegát** v jazyce **C#** slouží k předávání odkazů na metody. To může být užitečné například v případech, kdy chceme zavolat množinu metod, čekajících na konkrétní událost, která nastala [43].
- **ID** neboli identifikační číslo, které bude v rámci našeho projektu vždy globálně unikátní.
- **Levenstheinova vzdálenost** Tato metrika udává nejmenší počet znakových operací potřebných ke konverzi jednoho řetězce na druhý. Znakové operace jsou vložení, vymazání a substituce tokenu [44].
- **Reflexe** nám umožňuje pracovat s popisy jednotlivých objektů v jazyce **C#**. Tyto popisy můžeme použít například pro přístup k proměnným popisovaného typu nebo k jeho metodám. [45]
- **RPG hry** označují herní žánr, kde hráč převezme charakter zapadající do příběhu hry a kde jednotlivá rozhodnutí, kterým musí v průběhu hry čelit, ovlivní vývoj příběhu a jeho postavy [46].
- **Slovníkem** označujeme stejnojmenou datovou strukturu, která uchovává jednotlivé elementy jako páry klíčů a hodnot [47].
- **Singleton** v našem projektu je základní třída pro MonoBehaviour [48], která má jedinnou globálně dosažitelnou instanci a může nebo nemusí být prezistentní mezi scénami.
- **UI objekty** slouží k vytváření uživatelského rozhraní v **Unity editoru** [49].
- **Update** je funkce poskytnutá prostředím **Unity**, která se vykonává před vykreslením každého framu [50].
- **Virtuální metody** v jazyce **C#** jsou takové, které třída umožňuje přepsat ze třídy potomka v případě, že se stane rodičovskou třídou [51].
- **Zásobníkem** označujeme stejnojmenou datovou strukturu, jež vrací prvky v opačném pořadí, než ve kterém byly vloženy [52].