VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ROZŠÍŘENÍ VÝVOJOVÉ PLATFORMY UNITY PRO VIZUÁLNÍ PROGRAMOVÁNÍ

EXTENSION OF UNITY DEVELOPMENT PLATFORM FOR VISUAL PROGRAMMING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. LUKÁŠ HORKÝ**
AUTHOR

VEDOUCÍ PRÁCE **doc. RNDr. JITKA KRESLÍKOVÁ, CSc.**
SUPERVISOR

BRNO 2018

Zadání diplomové práce/21098/2017/xhorky21

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Horký Lukáš, Bc.**
Obor: Informační systémy
Téma: **Rozšíření vývojové platformy Unity pro vizuální programování**
Extension of Unity Development Platform for Visual Programming
Kategorie: Softwarové inženýrství

Pokyny:

1. Seznamte se s herní vývojovou platformou Unity. Diskutujte pracovní postupy pro tvorbu her za použití tohoto prostředí.
2. Nastudujte způsob tvorby rozšíření této platformy na úrovni editoru.
3. Navrhněte a implementujte rozšíření pro intuitivní tvorbu a spravování velkého počtu tříd tvořících herní logiku, včetně jejich atributů.
4. Navrhněte a implementujte editor pro vizuální programování v kontextu herní scény spolu se způsobem překladu či interpretace.
5. Dle zvoleného způsobu vytvořte interpret či překladač tvořící vrstvu mezi vizuálním editorem a jádrem Unity. Použitelnost vytvořeného systému demonstrujte na vhodné úloze vybrané po dohodě s vedoucí.
6. Zhodnoťte dosažené výsledky, zejména použitelnost v reálném prostředí a diskutujte možnosti dalšího rozvoje tohoto rozšíření.

Literatura:

- GREGORY, Jason. *Game engine architecture*. Wellesley, Mass.: A K Peters, c2009. ISBN 978-1568814131.
- HOCKING, Joseph. *Unity in action: multiplatform game development in C#*. Shelter Island, NY: Manning Publications Co., 2015. ISBN 978-1617292323.
- SCHELL, Jesse. *The art of game design: a book of lenses*. Boston: Elsevier/Morgan Kaufmann, c2008. ISBN 978-0123694966.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů zadání 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kreslíková Jitka, doc. RNDr., CSc.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Cílem diplomové práce je tvorba rozšíření pro vývojovou platformu Unity, které umožní tvorbu počítačových her od návrhu herní logiky po samotné programování scén na vizuální úrovni. Rozšíření pak tvoří abstraktní vrstvu mezi návrháři, kteří spravují herní prvky na logické úrovni a mezi programátory, kteří se starají o implementaci některých částí, jejichž tvorbu není možno automatizovat. Usnadňuje pak některé návrhové procesy a iterace vývoje eliminací potřeby ručního programování na místech, které automatizovat možno je a umožňuje se soustředit na samotný proces vývoje.

Abstract

The goal of the thesis is to create an extension for Unity Development Platform that will allow creation of computer games from design of game logic to scene programming on visual level. The extension forms an abstract layer between designers, who manage game objects on logic level, and programmers, who are responsible for implementation of parts that can't be created automatically. It then facilitates some designing processes and iterations by eliminating the need of manual programming in places that can by automate and allows to concentrate on the design process itself.

Klíčová slova

Programování, návrh, softwarové inženýrství, herní jádro, Unity, rozšíření, počítačová hra

Keywords

Programming, design, software engineering, game engine, Unity, extension, computer game, videogame

Citace

HORKÝ, Lukáš. *Rozšíření vývojové platformy Unity pro vizuální programování.* Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. RNDr. Jitka Kreslíková, CSc.

Rozšíření vývojové platformy Unity pro vizuální programování

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením paní doc. RNDr. Jitky Kreslíkové, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Lukáš Horký
18. května 2018

Poděkování

Děkuji své vedoucí doc. RNDr. Jitce Kreslíkové, CSc. za její vedení a odborné podněty, které mi při tvorbě této diplomové práce pomohly.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Play is critical to learning. Making „everyday“ things more playful, not only engages people more deeply, but allows them to better learn the systems they are using.

— Jason Della Rocca[1]

Cílem diplomové práce bylo studium ke tvorbě rozšíření herního vývojového prostředí Unity pro usnadnění procesu návrhu a pro snadné vizuální programování a následná implementace tohoto rozšíření dle bodů v zadání. Rozšíření pak tvoří abstraktní vrstvu mezi dvěma rolemi vývojářů – návrhářem, který je zodpovědný za herní logiku a programátorem, který implementuje a píše zdrojové kódy. Rozšíření spoustu věcí automatizuje a umožňuje soustředit se na samotný proces vývoje bez řešení integrace návrhů do herního jádra.

Rozšíření je tvořeno s ohledem na maximální modularitu a rozšiřitelnost. Není tvořeno pouze se zřetelem na jeden typ her, ideálně by mělo být použitelné pro řešení jakéhokoliv projektu, a to i větších rozměrů.

V kapitole 2 je uveden teoretický základ a podrobnější informace o vývoji počítačových her. Jsou zde představeny herní jádra a herní vývojová prostředí a velká část kapitoly je věnována popisu typů počítačových her, které mohou mít v praxi vliv na výběr vhodného jádra.

Kapitola 3 pak pojednává o samotném procesu návrhu, analýze požadavků a o některých teoretických náležitostech souvisejících s rozšiřováním editoru Unity. Součástí této kapitoly je taktéž popis návrhu vlastního rozšíření pro tvorbu grafického uživatelského rozhraní, které bylo dále v diplomové práci využito.

V kapitole 4 je popsán proces implementace rozšíření včetně všech technických náležitostí, schémat a názorných ukázek. Je zde také podrobně rozebrán vytvořený editor herních tříd a editor šablon. Je zde taktéž popsán způsob překladu navržených dat do podoby použitelné editorem Unity na úrovni scény.

Kapitola 5 popisuje různé způsoby testování hotového rozšíření a jeho praktické použití na konkrétní úloze.

Aktuální stav diplomové práce a závěr jsou shrnuty v kapitole 6, kde se mimo jiné nachází i souhrn dosažených výsledků a potenciální plán vývoje projektu do budoucna.

[1] Bývalý výkonný ředitel IGDA, zakladatel Perimeter Partners a spoluzakladatel Execution Labs.

Kapitola 2

Herní jádra

Termín „herní jádro“, neboli též „herní engine“ (anglicky „Game Engine“) vznikl v roce 1993, kdy byla vydána hra *Doom* společností ID Software. Architektura této hry oddělovala softwarové komponenty řídící vykreslování ve 3D prostoru, audio systém či systém pro detekci kolizí, od samotných herních zdrojů (anglicky assets[1]). Velkou výhodou tohoto řešení pak byla znovupoužitelnost jádra pro jiné projekty za použití zdrojů nových, ale bez nutnosti opětovné tvorby celého jádra – to mohlo zůstat stejné či pouze minimálně pozměněné.

Tato výhoda dala vzniknout tzv. módovací komunitě – skupině lidí, kteří hry upravovali či tvořili nové hry zásahem do již existujících her za použití nástrojů, které byly vývojářem originální hry často dostupné. Hry, jako *Quake III Arena* či *Unreal*, které vznikly koncem devadesátých let minulého století, již s „módováním“ počítaly a začaly vznikat i první pokusy o prodej licencí k herním jádrům, která mohly menší společnosti zakoupit a využít pro tvorbu vlastních herních titulů. Tato praktika přežila dodnes a často se nové hry vytváří za použití již existujících herních jader, což je pro vývojáře méně nákladné než výstavba zcela vlastního řešení.

Rozdělení hry od svého jádra je často neurčité. U některých jader je separace na první pohled jasná, jindy tomu tak ovšem není a záleží zcela na návrhu architektury jádra, jak se k tomuto problému staví. Obecně vzato však herní jádra nejčastěji používají architekturu řízenou daty, která je popsána v kapitole 2.1. Rozhodnutí o tom, zda hra má své jádro oddělené od samotné hry není navíc binární; místo toho je možno každé jádro zařadit na stupnici znovupoužitelnosti, která je znázorněna na obrázku 2.1. Obecně vzato však platí, že čím obecnější jádro je, tím méně optimalizované je pro konkrétní hru na konkrétní platformě [8].

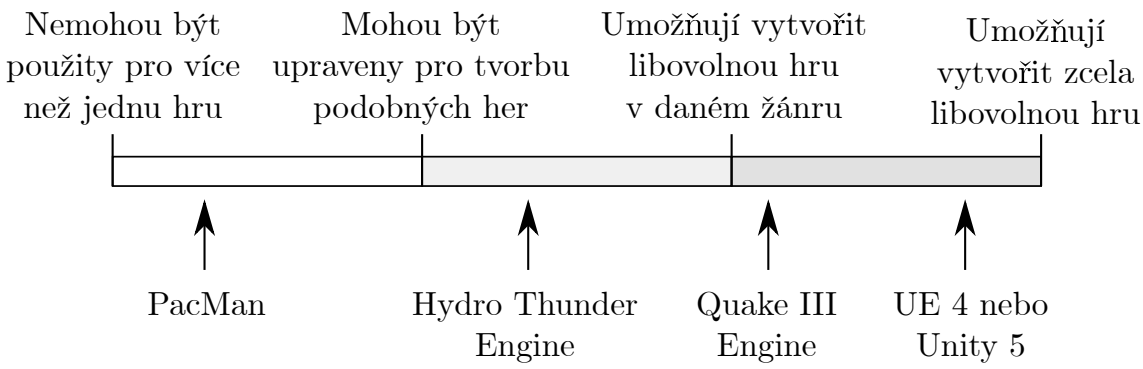
2.1 Architektura řízená daty

Architektura řízená daty (anglicky „Data-driven architecture“), někdy též programování řízené daty („Data-driven programming“), je paradigma, ve kterém je program popsán abstraktními daty a obsahuje pouze generický zdrojový kód, který s těmito daty nakládá. Pro změnu výstupu programu je tedy potřeba změnit vstupní data.

Objekty v této architektuře jsou často tvořeny položením následujících otázek:

1. Co tento objekt reprezentuje?

[1] Herní „asset“ označuje znovupoužitelný zdroj projektu. Může se jednat o grafický prvek – např. textura či 3D model, nebo i o logický prvek – skript, dialog... V některých případech jde o jejich kombinaci, jako jsou celé herní světy, scény, aj.



Obrázek 2.1: Znázornění znovupoužitelnosti herních jader [Inspirováno [8]]

2. Jaké operace může tento objekt použít?

Architektura řízená daty využívá zapouzdření, což zlepšuje možnost, aby byl software opakovaně použit, upraven, testován, udržován či rozšířen. Plné výhody je však docíleno pouze pokud je zapouzdření maximální v době procesu návrhu. Roku 1989 však někteří účastníci konference o objektově orientovaných systémech, jazycích a aplikacích vyjádřili názor, že tento přístup zapouzdření nemaximalizuje, protože se příliš soustředí na implementaci objektů – více v [23].

Alternativou pak jsou architektury řízené odpovědností („Responsibility-Driven“), událostmi („Event-Driven“) či např. aspekty („Aspect-Oriented“).

2.2 Herní žánry a typy počítačových her

Máme-li diskutovat jádra počítačových her a existující vývojová prostředí, je třeba nejprve popsat typy, do kterých je možno hry rozdělit a které často o použitém jádře rozhodují. Tato kapitola popisuje nejčastější žánry, nejedná se však o kompletní výčet a často se navíc tyto druhy prolínají – nejsou vzájemně výlučné.

Dále je vhodné vymezit termín „počítačová hra“. Tímto termínem se nejčastěji označuje podmnožina her v obecném smyslu, která je hráči zprostředkována prostřednictvím počítače. To mimo jiné umožňuje počítačovým hrám vypůjčovat si zábavní techniky z jiných médií, jako jsou knihy, filmy atp. Samotný pojem „hra“ na obecné úrovni je možno popsat jako „umělý konflikt“, jak je definováno v knize *Rules of Play* [16].

Důležitým aspektem každé hry jsou pravidla a cíl. Pravidla jsou definice a instrukce, se kterými hráč souhlasí a kterými se řídí během hraní hry. Budují účel hry a všech aktivit a událostí, které se ve hře vyskytují a pomáhají hráči rozhodnout, jaký postup zvolit pro dosažení cíle hry. Kniha *Fundamentals of game design* od Ernesta Adamse [1] jakožto součást pravidel definuje následující:

- **Sémiotika hry** – definuje významy a vztahy nejrůznějších symbolik, která hra používá. Tyto symboly mohou být čistě abstraktní (např. auty v baseballu), jiné mají základ ve skutečném světe (např. armády v tahových hrách).

- **Hratelnost**[2] – skládá se z výzev (úkolů) a akcí, které hráč může podniknout. Definice se může lišit, např. herní návrhář Sid Meier v knize *Game Architecture and Design* [12] popisuje hratelnost jakožto „sérii zajímavých voleb [hráče]“.
- **Sekvence hraní** – vývoj aktivit, které celou hru tvoří.
- **Cíle hry** – úkoly, které hráč plní; jsou definovány obvykle v základních pravidlech hry.
- **Terminální podmínka** – podmínka, která musí být splněna, aby došlo k ukončení hry (pokud hra takovouto podmínku obsahuje).
- **Metapravidla** – pravidla o pravidlech. Mohou například indikovat, za jakých okolností se mohou některá pravidla měnit či jaké výjimky jsou povoleny.

Dále je možno rozdělit hry do podmnožin dle toho, jak se staví k soupeření a kooperaci hráčů. K soupeření dochází, pokud mají hráči odlišné zájmy, tedy když se snaží dosáhnout cílů, které jsou vzájemně výlučné. Kooperace naopak nastává v případě, kdy se hráči snaží dosáhnout stejného nebo podobného cíle a tedy spolupracují. Hráči, kteří se snaží dosáhnout rozdílných cílů, které však nejsou protikladné, nekooperují ale současně ani nesoupeří. Dá se říci, že hrají odlišnou hru (nebo část hry).

Než se dostaneme k samotným žánrům počítačových her (viz další kapitoly), je dobré uvést možné kombinace, které mohou nastat v závislosti na permutaci soupeření a kooperaci hráčů. Jedná se o následující varianty herních (nejen počítačových) módů:

- **Kompetitivní pro dva hráče** – „ty versus já“ – nejznámější herní model. Typickým příkladem mohou být velmi staré hry, jako šachy či vrhcáby.
- **Kompetitivní pro více hráčů** – „každý sám za sebe“ – model občas též známý pod anglickým termínem „deathmatch“. Jedná se např. o hry, jako je stolní Monopoly, poker či spousta sportovních her.
- **Kooperativní pro více hráčů** – „všichni společně“ – varianta, kdy všichni hráči spolupracují pro dosažení společného cíle. Nejsou časté v klasických stolních hrách, ale v počítačových hrách jsou často zabudovány jakožto alternativa ke klasickým kampaním[3], kdy se hráči snaží společnými silami porazit umělou inteligenci. V kontextu MMORPG her (viz kapitola 2.2.4) se navíc často o tomto modelu hovoří jakožto o PvE – „Player(s) vs Environment“, tedy „hráč(i) proti prostředí“.
- **Týmové** – „my proti nim“ – tento model se vyskytuje v případě, kdy členové týmu kooperují, ale samotné týmy mezi sebou soupeří. Příkladem z reálného světa může být např. fotbal.
- **Pro jednoho hráče** – „já proti situaci“ – do této kategorie spadá většina starých arkádových her, jako je např. Mario od společnosti Nintendo. Uživatelé operačního systému Windows často znají vestavěnou hru Solitaire, která sem taktéž spadá.

[2] Anglicky „Gameplay“ – v tomto případě se nejedná o hratelnost ve smyslu hodnocení kvality hry, ale samotný akt hraní (v dlouhodobějším smyslu) jako takový.

[3] Kampaň je série úkolů či misí s příběhem, která často tvoří hlavní část hry.

- **Hybridní modely** – sem patří hry, které mohou prostřednictvím definovaných pravidel umožňovat přechody mezi výše uvedeným. Hráči tak například soupeří, ale za určitých okolností mohou i spolupracovat pro dosažení menších cílů.

Spousta počítačových her obsahuje více než jeden herní mód. Hráči často před samotným započetím hry volí, jaký mód si přejí hrát, čemuž pak může být uzpůsobena sada pravidel.

2.2.1 Strategické hry

Strategické hry jako takové jsou jedny z nejstarších her vůbec. Mezi nejznámější patří např. desková hra Senet (3500 př.n.l., Egypt), Královská hra z Uru (2500 př.n.l., Uru) či hra Go (2000 př.n.l.). Jedna z nejhranějších deskových her – šachy – se pak datuje k 6. století našeho letopočtu (Indie), avšak moderní verze byla vytvořena až v 15. století v jižní Evropě [10]. Tato kapitola bude dále pojednávat o strategických hrách v kontextu počítačových her, avšak je dobré mít na paměti, že spousta prvků a charakteristik odráží strategické stolní hry z reálného světa a ty pak v různých podobách implementuje a rozšiřuje [1].

Charakteristika strategických her

Strategické hry jsou takové hry, které prezentují úkoly a cíle jakožto sadu strategických konfliktů, které hráč řeší výběrem z možných akcí a tahů, které může v dané chvíli během hraní podniknout. Vítězství je dosaženou prostřednictvím pečlivého plánování a podnikáním optimálních akcí. Náhoda nesmí hrát příliš velkou roli a slouží pouze pro diverzifikaci hratelnosti (pokud se ve hře vůbec vyskytuje).

Strategická hra může obsahovat množinu menších problémů k řešení, jako jsou: Taktické, logistické, ekonomické, aj. Fyzická koordinace nemá tradičně příliš velkou roli, i když pokud se jedná o *strategickou hru hranou v reálném čase* (Rea-Time Strategy), může být reakční doba hráče a schopnost rychle improvizovat vliv na výsledek hry. To nebývá případ „tahových strategických her“ (Turn-Based Strategy).

V kontextu moderních strategických her se často používají dva následující pojmy:

- **Macro řízení** (macromanagement) – schopnost provádět velké množství úkonů v jedné chvíli. Například stavět budovy, provádět výzkum, starat se o ekonomiku, přemisťovat armády...
- **Micro řízení** (micromanagement) – schopnost rychle provádět drobné akce, které se mohou na první pohled zdát irelevantní, ale mohou mít vliv na výsledek hry. Jedná se zejména o kontrolu jednotek během soubojů – odvolávání zraněných jednotek mimo dosah nepřítele, ovládání formací, koncentrace útoku do konkrétních nepřátel atp.

Výše uvedené termíny mají základ v řízení týmů manažery ve skutečném světě, i když význam může být občas přenesený a mírně odlišný. To však není předmětem této práce. Pro další informace o tomto použití termínů vizte např. [4], kde jsou rozebírány záporné charakteristiky micromanagementu.

Ve většině počítačových strategických her hráč vidí část herního světa z ptačího pohledu a přímo či nepřímo kontroluje své jednotky. Tyto hry pak umožňují ovládat více jednotek současně oproti pouze jedné postavě (jak tomu bývá u rolových nebo akčních her). Úkolem pak bývá eliminace nepřátelského hráče (či hráčů) prostřednictvím ekonomické a válečné dominance.

Příklady strategických her

První počítačovou hrou, kterou lze zařadit do žánru strategických her, byla hra *Invasion* pro herní konzoli Magnavox Odyssey, jež byla vydána roku 1972. V roce 1980 pak vyšla první počítačová historická válečná hra *Computer Bismarck* od společnosti Strategic Simulations, která byla inspirována poslední bitvou bitevní lodě Bismarck (1941).

V osmdesátých letech minulého století pak začala vznikat celá řada strategických počítačových her. Za zmínku stojí hra *The Lords of Midnight*, kterou vytvořil roku 1984 Mike Singleton a která získala ocenění *Nejlepší dobrodružná hra roku 1984* (i když se technicky nejednalo o adventuru) časopisu Crash a taktéž získala ocenění *Zlatý joystick*.

Za první strategickou hru hranou v reálném čase (nikoliv tahově) se považuje *Herzog Zwei* vydaná společností Sega pro konzoli Sega Genesis[4] roku 1989, i když některé elementy Real-Time strategických her je možno najít i ve starších titulech, jako *Cytron Masters*, *Utopia*, *Bokosuka Wars* a.j. Žánr strategických her dále popularizovala roku 1992 hra *Dune II*, která kladla velký důraz na aspekt reálného času a nutnost rychlého plánování [21].

V dnešní době se za nejpopulárnější herní titul u strategií považuje série *Total War*, která vznikla roku 2000 společností Creative Assembly. Jen v rozmezí roku 2009-2015 bylo prodáno 11 milionů kopií her této série [14]. Dalšími úspěšnými hrami v tomto žánru jsou: *StarCraft 2* (každý rok hraný na šampionátu společnosti Blizzard Entertainment – Blizzcon), *Civilization* (za posledních 25 let vydáno 13 titulů[5]) či *Age of Empires* (3 hlavní tituly a 7 nástaveb; koncem roku 2017 navíc oznámen 4. díl).

2.2.2 Akční hry

Akční hry jsou takové hry, ve kterých je většina překážek a úkolů prezentována jakožto test fyzických schopností a koordinace hráče. Vyžadují si rychlou reakci na události a hráč často nemá čas pro strategii nebo plánování.

Tyto hry jsou nejčastějším příkladem, který si většina lidí představí pod termínem „počítačová hra“ či „videohra“, a to zejména z historického důvodu – většina prvních videoher byla právě tohoto typu – například arkádové hry, jako *Asteroids*, *Pac-Man* či *Space Invaders*. Ty však nejsou typickým příkladem moderních akčních her a byla jim proto vyhrazena zvlášť podkapitola na konci 2.2.2.

Charakteristika akčních her

Jak již bylo řečeno, akční hry se vyznačují především nutností dobré koordinace mezi „okem a rukou“ a vyžadují si rychlé reakce hráče a motorické schopnosti. Ne všechny akční hry jsou však postaveny pouze na rychlosti. Některé si vyžadují jiné fyzické dovednosti, jako přesné míření, precizní časování či schopnost provádět „komba“ – komplikované sekvence příkazů.

Akční hry většinou nemívají příliš strategických aspektů v rámci hlavních mechanik, avšak můžou pro zpestření obsahovat různé hádanky a taktické konflikty. Obecně vzato ale hry tohoto žánru nastavují jednoduchá, snadno pochopitelná pravidla a cíle a poskytují zřejmou cestu, jak těchto cílů dosáhnout. Samotné jejich dosažení pak je obtížné dle schopností hráče.

[4] Herní konzole Sega Genesis je též známá pod názvem Mega Drive v regionech mimo severní Ameriku.

[5] Jedná se o 6 hlavních titulů (Civilization – Civilization VI) a 7 samostatných odnoží. Společnosti stojící za vznikem v průběhu let: MicroProse, Firaxis a Take Two.

Nejvíc zastoupené jsou dnes v tomto žánru *střílečky* (Shooter Games) [24]. V jejich případě hráč provádí akce na dálku pomocí střelných zbraní. Klíčovým elementem je míření a hra má často omezení, jako je počet nábojů ve zbrani. Hráč dále musí mít povědomí o svém cíli, ale také o oblasti kolem něj (sám se často může stát cílem). V případě 3D střílečeк (v dnešní době nejčastější typ) bývá kladen důraz na věrohodnost prostředí. Tyto hry kopírují reálný svět – funguje gravitace, zvuk je prostorový, objekty vrhají stíny a mají relativně realistické kolize...

Dále je dobré zmínit *bojové hry* (Fighting Games). Ty jsou velmi odlišné oproti jiným akčním hrám, protože téměř nikdy neobsahují hádanky či prvky prozkoumávání. Místo toho je veškerý důraz kladen jen na reakční dobu hráče a časování. Tyto hry většinou simulují boj na blízko a hráči provádí série útoků a blokování pro poražení nepřítele.

Příklady akčních her

Za první známou akční hru lze považovat *Space Invaders* z roku 1978, která je označována jakožto počátek zlaté doby arkádových video her [22]. Arkádovým hrám je však věnována samostatná podkapitola na konci 2.2.2.

Prvními hrami, které zutilizovala střelbu ve 3D prostoru, jsou *Maze War*[6] a *Spasim*. Ty však nejsou obecně známy natolik, jako *Wolfenstein 3D* od společnosti id Software z roku 1992, který oproti předchozím titulům obsahuje více násilných prvků. Tatáž společnost o rok později vydala světoznámou hru *Doom* na podobných hororových sci-fi motivech. Hra *Doom* se mimo jiné roku 2016 dočkala restartu.

Za zmínku pak stojí některé další známé akční tituly, jako Quake, Half-Life, Counter-Strike či Call of Duty. Předmětem této práce však není kompletní výčet ani jejich analýza, pro podrobnější informace vizte [1].

Co se týče bojových her, roku 1984 vznikly hry *Karate Champ* a *Kung-Fu Master*, které položily základ ostatním hrám tohoto žánru. Známější je však pravděpodobně hra *Street Fighter* společnosti Capcom, jenž vznikla o 3 roky později. Tento titul má k roku 2018 6 hlavních sérií a celou řadu sérií vedlejších.

Subžánr: Arkádové hry

Arkádové hry jsou speciálním případem akčních her. Dá se říci, že jsou předchůdcem akčních her tak, jak je známe dnes. Arkádové hry se taktéž soustředí na rychlost, přesnost a reflexe hráče, rozdílem ale je, že se nepokouší být realistické a pohlcující. Často nemívají téměř žádný příběh či zápletku, oproti tomu kladou důraz na svižnou hratelnost a kompetitivní stránku hry.

Arkádové hry často bývají velmi obtížné (buď od začátku, nebo se obtížnost stupňuje). Původně vznikaly pro herní automaty a jejich účelem bylo vydělávat porážením slabých a nezkušených hráčů. Pouze znalí hráči byli schopni hrát tento druh her delší dobu bez prohry. Principiálně jednoduché herní mechaniky a audiovizuální obsah navíc umožňoval hrát tyto hry i na méně výkonných strojích. Dnes se tyto hry občas vyskytují na starších mobilních telefonech či webových prohlížečích.

Ikonickými příklady těchto her jsou *Space Invaders*, *Asteroids*, *Pac-Man* či *Robotron: 2084*.

[6] Někdy též známo pod jménem *The Maze Game*, *Maze Wars*, *Mazewar* či *Maze*.

2.2.3 Dobrodružné hry

Dobrodružné hry (adventury) jsou hry, ve kterých hráč zaujímá roli hlavního protagonisty v příběhu, kterým postupuje prostřednictvím prozkoumávání světa a plněním nejrůznějších hádanek. Důraz není kladen na obtížnost či složité herní mechaniky, jako spíš na samotný zážitek z hraní a příběhu.

První dobrodružné hry měly pouze textovou podobu – jednalo se v podstatě o interaktivní vyprávěný příběh. Hráči byl předložen text popisující svět a situaci, ve které se nachází, a bylo na něm, aby z různých možností vždy vybral (pomocí zadání textu, který byl analyzován), jak si přeje pokračovat. Tyto hry fungují na podobných principech, jako tzv. „Gamebook“ knihy[7], ve kterých není čtenář stoprocentně vázán napsaným dějem, ale může si ho do určité míry sám upravovat a vytvářet. Tyto knihy pak položily základ pro hypertextovou fikci [17].

Postupem času se s novější technologií začaly vytvářet dobrodružné hry s grafickým rozhraním. Nejprve uživatelův textový vstup doprovázely obrázky a nákresy, později začaly být populární hry, ve kterých mohl s prostředím interagovat klikáním myší. V dnešní době nejsou výjimkou ani dobrodružné hry s dotykovým ovládáním.

Charakteristika dobrodružných her

Žánr dobrodružných her je charakteristický především důrazem na příběh. Vyprávění a průzkum světa hraje nejvyšší roli. Hra pak neobsahuje žádné (nebo zcela minimální) překážky v podobě soubojů, řízení ekonomiky či akce, to však nutně neznamená, že v těchto hrách není žádný konflikt k řešení – pouze není primární aktivitou a hráč jej řeší nepřímo pomocí konceptuálních překážek a hádanek. Veškerá ekonomika dobrodružných her je pouze interní. Všechny vztahy jsou pouze symbolické, nikoliv numerické. Tím se tyto hry razantně odlišují od rolových her (RPG), které jsou blíže popsané v kapitole 2.2.4.

Spousta her z jiných žánrů mají kvalitní prostředí i příběh, ale pokročilí hráči tento jejich aspekt zcela ignorují. Pro profesionální hráče je irelevantní, že šachy napodobují středověký válečný konflikt nebo že *Quake* vypráví příběh vesmírných mariňáků na cizí planetě. Koncentrují se na nutné základy hratelnosti – na strategii pro vítězství nad oponentem nebo na akci a rychlé ničení nepřátel řízených umělou inteligencí. Tohle však dobrodružné hry eliminují. Svět a vyprávění příběhu přidává hře na hodnotě mnohem víc než v jakémkoliv jiném žánru. Ať už je prostředí temné a depresivní, fantastické a bizarní nebo veselé a zábavné – ve výsledku vytváří iluzi skutečného, fungujícího světa, ve kterém hráč žije a který zkoumá.

Dobrodružné hry jsou téměř výhradně pouze pro jednoho hráče.

Příklady dobrodružných her

Jedna z prvních dobrodružných her pro osobní počítače nesla jednoduché jméno *Adventure*[8]. Neměla však žádnou zápletku – pouze poskytovala možnost průzkumu světa a hádanky pro rozluštění. Autorem byl Will Crowther, který hru napsal pro platformu PDP-10 roku 1976.

Na tomto principu byla později (mezi roky 1977 a 1979) skupinou studentů Massachusettského technologického institutu vytvořena mnohem větší hra *Zork* (vydána ve třech dílech). Ta kromě bohatšího světa nabízela i sofistikovanější textový analyzátor pro vstup,

[7] „Gamebook“ knihy jsou občas též známy pod zkratkou CYOA – „Choose Your Own Adventure“.
[8] Někdy též *Colossal Cave Adventure*, *ADVENT* či *Colossal Cave*.

který dokázal rozpoznat a zpracovat předložky a spojky. Příklad: „`hit troll with Elvish sword`“ (místo klasického „`hit troll`“).

S příchodem výkonnějších osobních počítačů a možností grafických rozhraní pak začaly vznikat další dobrodružné hry. Za zmínku stojí zejména herní série *King's Quest* společnosti Sierra Entertainment či hra *The Secret of Monkey Island* (Ron Gilbert ze společnosti Lucasfilm) nebo *Myst* (bratři Robyn a Rand Millerovi).

2.2.4 Rolové hry

Rolové hry, neboli též hry na hrdiny (anglicky Role-Playing Games – RPG), je typ her, ve kterých hráči zaujímají role fiktivních postav (jedné či vícera), které jsou nejčastěji navrženy samotným hráčem, a ty pak vedou napříč sérií úkolů a překážek.

Počítačové rolové hry byly popularizovány svým stolním protějškem – tzv. PnP (pen-and-paper) rolovými hrami. V těch se menší skupina lidí sejde a hraje prostřednictvím diskuzí. GM (Game Master)[9] popisuje svět spolu s hlavním dějem a samotní hráči pak popisují akce svých postav. Výsledek těchto akcí opět popisuje GM. Někdy je rozhodnut herním systémem a pravidly, někdy samotným GM [18].

Charakteristika rolových her

Rolové hry umožňují hráči zaujmout s jeho postavou roli v řešení důležitého konfliktu. Ernest Adams ve své knize *Fundamentals of Game Design* shrnuje tyto hry citátem: „Jen ty můžeš zachránit svět!“ [1] Cílem hry ovšem nemusí být pouze záchrana světa, ale spousta dalšího (odhalení vraha, záchrana princezny, hledání tajemství, návrat domů po únosu...).

Úkoly mohou být majoritní – nutné pro vítězství v dané hře, či minoritní – pouze poskytující nějaký bonus – většinou zcela dobrovolné. Postava typicky plněním těchto úkolů sílí a získává nové schopnosti či jiné výhody. Překážkami pak mohou být taktické souboje, logistika, ekonomický růst, průzkum světa či řešení hádanek.

Rolové hry často kombinují aspekty z jiných žánrů. Existují například:

- **Válečné RPG** – Obsahuje prvky strategií, kdy hráč ovládá celé armády jednotek, ale současně je velký důraz kladen na růst jednotlivců – hlavních postav v příběhu.
- **Akční RPG** – Mimo jiné testují hráčovy fyzické schopnosti. Kromě taktiky a znalosti hry tak hráč potřebuje i např. rychle reagovat na události ve hře.
- **Dobrodružné RPG** – Poskytují propracovaný svět s kvalitním příběhem (či příběhy) a volností, avšak růst postav je numerický (větší síla útoku, životy...), nikoliv pouze abstraktní, jako u čistých adventur (viz 2.2.3).

Rolové hry mohou být pro jednoho i pro více hráčů. RPG pro jednoho hráče kladou větší důraz na samotné herní mechaniky, zatímco u RPG pro více hráčů je důležitá sociální interakce [18]. RPG pro velký počet hráčů (řádově tisíce na serveru) se nazývají MMORPG.

Příklady rolových her

Prvním komerčně dostupnou RPG byla *Dungeons & Dragons* (D&D či DnD) – stolní hra vytvořena Gary Gygaxem a Dave Arnesonem začátkem sedmdesátých let minulého století, jež byla inspirována fantastickou literaturou. Později začaly vznikat elektronické podoby

[9] GM, neboli Game Master, je vypravěč, který nehraje, ale řídí celou hru pro ostatní.

této hry pod názvy, jako *dnd* či *Dungeon*. Jako alternativa pak roku 1975 vznikla hra *Tunnels & Trolls*, kterou vytvořil Ken St. Andre.

Za nejúspěšnější na PC se v dnešní době považují tituly, jako *The Elder Scrolls* (1994, Beteshda Softworks), který je znám propracovaným světem s vlastní historií, kalendářem, písmem, mytologií a kulturou, futuristické akční RPG *Mass Effect* (2007, BioWare) či *Dark Souls* (2011, FromSoftware), jenž se proslavilo svou nadprůměrnou obtížností, *The Witcher* (2007, CD Projekt), který byl oceněn jako „RPG Game of the Year" magazínem PC Gamer US v roce 2007 (později získal další ocenění) či *Fallout* (1997, Interplay), který se odehrává ve světě zubožením jaderným konfliktem.

Čím dál více jsou však populárnější tzv. MMORPG [15].

Subžánr: MMORPG

Masive Multiplayer Online Role Playing Games (MMORPG) jsou speciálním typem rolových her, které jsou určeny pro velký počet hráčů v jedné chvíli v daném světě (na specifickém serveru). Kladou důraz na sociální interakce mezi hráči a spoustu úkolů ve hře je možno splnit pouze ve skupinách (často až desítky hráčů pro poražení jediného nepřítele). Hráči tohoto žánru často zakládají spolky a nejrůznější komunity. Komunikace je obvykle omezena na chat, některé hry podporují i hlasovou komunikaci přes mikrofon. Hráči ovšem mohou používat i externí programy, jako Skype, TeamSpeak či Discord.

Nejznámější hrou v tomto žánru je pravděpodobně *World of Warcraft*, který vznikl jako pokračování strategických her *Wacraft I – III* společnosti Blizzard Entertainment roku 2004. K roku 2018 existuje 6 nástaveb na originální hru a na Blizzconu 2017 byla oznámena nová expanze – *Battle for Azeroth*.

Dalšími novodobějšími úspěšnými hrami tohoto žánru jsou *The Elder Scrolls Online* (2014, Bethesda Softworks), *Guild Wars 2* (2012, ArenaNet), *Star Wars: The Old Republic* (2011, Electronic Arts, LucasArts) či *Final Fantasy XIV* (2010, Square Enix).

2.2.5 Sportovní hry

Sportovní hry jsou jedny z nejpopulárnějších počítačových her [1]. Spousta lidí hraje nebo sleduje sport v reálném světě a často tak přejdou ke sportovním videohrám s velkým očekáváním, které musí výrobci těchto her naplnit.

Tyto hry pak simulují aspekty reálných, nebo i imaginárních sportů – ať už se jedná o samotné zápasy, řízení týmu či obojí. Zápasy využívají fyzických a strategických překážek, spravování mužstva pak především překážek ekonomických.

Často se do této kategorie řadí i závodní hry. Ernest Adams ve své knize *Fundamentals of Game Design* [1] vyhrazuje těmto hrám oddělenou kapitolu, v rámci této práce jsou ale tyto hry sloučeny s ostatními sportovními hrami pro jejich podobnost.

Charakteristika sportovních her

Většina sportovních her se soustředí na simulaci samotných sportovních utkání, ale některé obsahují i manažerské úkoly, jako řízení týmu či kariéry nějakého sportovce. Důraz je kladen na uvěřitelnost a realističnost a často tyto hry kopírují pravidla her z reálného světa.

Spousta sportovních her obsahují herní oblast rozdělenou do dvou sektorů (dva týmy) s cíli na každé straně a samotní atleti (jednotky, které hráč ovládá) se pak snaží dostat nějaký objekt do cíle v sektoru protivníka. Příkladem může být simulátor basketbalu, ho-

keje, fotbalu či vodního póla. Moderní sportovní hry často používají simulování fyziky pro determinaci chování atletů či jiných objektů (míč).

Co se týče závodních her, ty sdílí některé prvky klasických sportovních her. Také vytváří fyzické a strategické překážky pro hráče, ale často je mnohem větší důraz kladen na reflexi hráče, který ve hře řídí vozidlo nebo jiný prostředek za vysoké rychlosti. Prvek managementu je často v těchto hrách implementován prostřednictvím vylepšování daného vozidla (hráč má omezenou měnu a vydělává prostřednictvím výher). Tyto hry se také snaží být velmi realistické a působivé.

Příklady sportovních her

Nejznámějším titulem sportovního žánru počítačových her je *FIFA*, též známá jako *FIFA Football* či *FIFA Soccer* od společnosti Electronic Arts. První hra této fotbalové série vznikla koncem roku 1993 a k roku 2018 existuje více, než 30 pokračování. Tradičně společnost vydává nový díl každý rok, přičemž občas v jednom roce vzniknou díly dva nebo výjimečně i tři (2005, 2006).

Z historického hlediska dále stojí za zmínku hra *Championship Manager* (1992, Collyer brothers) – hra postavená kolem řízení fotbalových týmů, kde důležitým aspektem je strategie, či *Arch Rivals* (1989, Midway), která se snažila o vykreslení basketbalu ze satirického pohledu pro komediální efekt.

Mezi zástupce moderních závodních her pak patří např. *Need for Speed* (1994, Electronic Arts) s přibližně 25 díly k roku 2018, *Trackmania* (2003, Nadeo) s 12 díly či *Forza* (2005, Microsoft Studios) s 10 díly.

2.3 Moderní herní vývojová prostředí

Herní vývojové prostředí je software, který obsahuje herní jádro, ale současně i nástroje (nejčastěji editor) pro samotnou tvorbu her využívající toto jádro. Tyto nástroje se pak starají o automatický překlad či interpretaci vytvořených projektů a obvykle obsahují možnosti rychlého spuštění a testování naprogramované hry. V dnešní době se pojmy „herní vývojové prostředí“ a „herní jádro“ často zaměňují a sjednocují pod anglickým názvem „Game Engine“, jelikož málokteré moderní herní jádro neobsahuje editor či jiné nástroje pro její efektivní využití bez nutnosti přídavného software.

2.3.1 Stručný výčet existujících prostředí

Tato kapitola stručně pojednává o moderních herních vývojových prostředích. Následující výčet neobsahuje prostředí Unity, jelikož tomu byla věnována zvlášť kapitola 2.3.2.

FrostBite

FrostBite je herní engine původně vyvíjený společností EA DICE, nyní Frostbite Labs. Podporuje více platforem, převážně je však určen na tvorbu her pro herní konzole, jako je PlayStation 3, PlayStation4 či Xbox One. Podporovány jsou i systémy Miccrosoft Windows.

Příklady her: Battlefield: Bad Company 2, Battlefield 3, Need for Speed: The Run...

Unreal Engine

Unreal Engine je největším konkurentem Unity. Původně se jednalo o jádro, nad kterým byla vystavena akční hra Unreal z roku 1998. Společnost Epic Games však tento engine rozšiřovala a v dnešní době je možno jej použít pro tvarbu her prakticky jakéhokoliv žánru.

Příklady her: Mass Effect Series, Dishonored, Bioshock Infinite, Batman: Arkham Asylum...

Source

Herní jádro Source od společnosti Valve je převážně používáno pro akční hry, nicméně podporuje moduly a ve výsledku je možno tvořit hru prakticky jakéhokoliv žánru. Pro tvorbu her využívajících toto jádro je však nutné nainstalovat Source SDK.

Příklady her: Dota 2, Half Life 2, Counter-Strike: Source, Counter-Strike: Global Offensive, Left4Dead, Left4Dead 2, Portal 1, Portal 2...

CryEngine

Prostředí CryEngine bylo původně vytvořeno pro sérii Far Cry, nicméně jeho vývoj pokračuje a společnost Crytek vydává stále nové verze. Tento engine se používá zejména pro tvorbu akčních her, přičemž existuje i upravená verze pro rolové hry.

Příklady her: Crysis 2, Crysis 3, Kingdom Come: Deliverance...

Hero Engine

Hero Engine společnosti Simutronics je komerční herní engine pro tvorbu MMORPG her. Jeho součástí je i technologie používaná na samotných serverech, nikoliv pouze jádro klientské hry. Umožňuje tzv. „online tvorbu“, kdy vývojáři mohou pracovat přímo uvnitř hry a uvidí navzájem své výsledky v reálném čase.

Příklady her: Star Wars: The Old Republic, Elder Scrolls Online, Exile Online...

2.3.2 Vývojové prostředí Unity

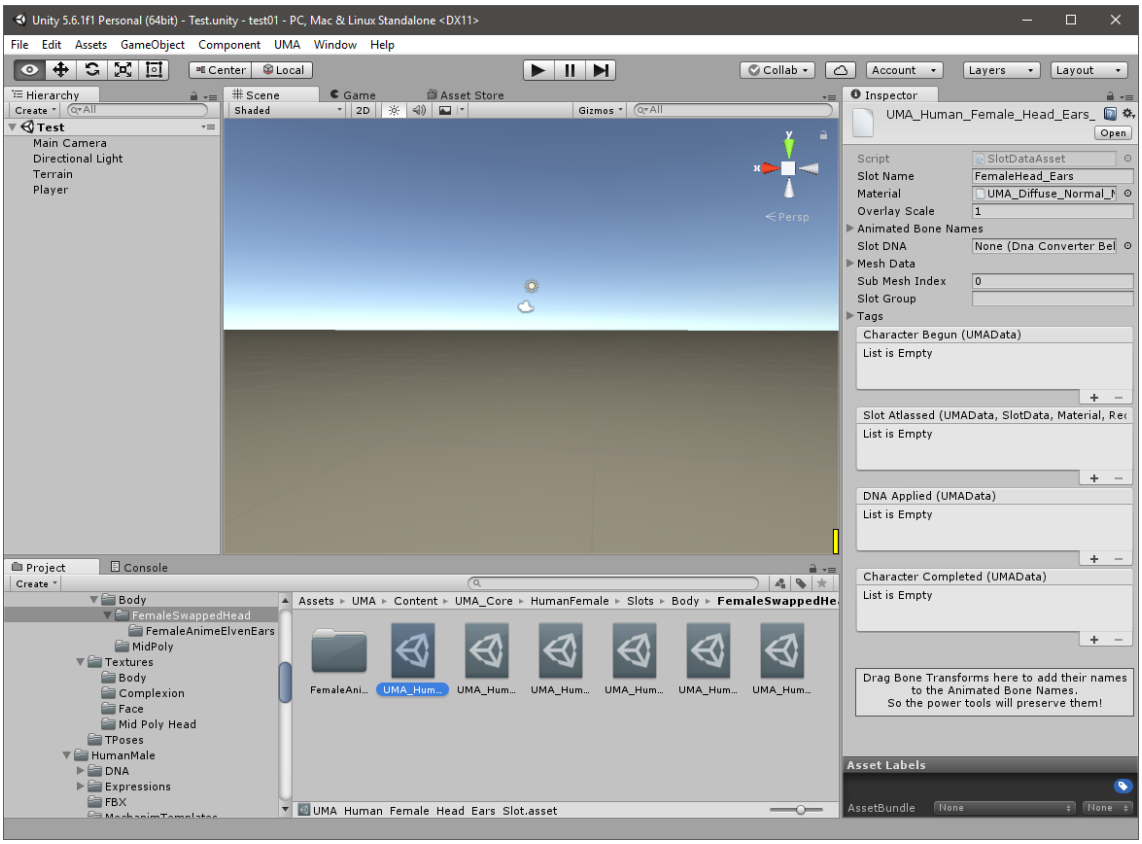
Unity je jedno z nejrozšířenějších herních vývojových prostředí. Jeho silnou stránkou je zejména multiplatformovost, tento engine je možno použít pro vývoj her pod PC, konzolemi, mobily i weby. První verze z roku 2005 byla představena na celosvětové konferenci Applu a podporovala pouze OS X, od té doby ale bylo Unity rozšířeno o více, než 15 dalších platforem.

Unity umožňuje vyvíjet 2D i 3D hry libovolného žánru. Nejčastěji je používáno pro tvorbu akčních her menších společností, ale zdaleka není na tento žánr omezeno. Za pomocí tohoto jádra byly vyvinuty nejrůznější hry, od závodních přes arkádové až po MMORPG [20].

Unity poskytuje spoustu moderních funkcí, mezi něž patří například:

- Simulace fyziky
- Normálové mapy
- Prostorové okluze
- Dynamické stíny

- a spousta dalšího...



Obrázek 2.2: Ukázka vývojového prostředí Unity [Snímek obrazovky]

Pracovní postup vývoje v Unity je poměrně unikátní oproti jiným vývojovým prostředím. Jiné herní nástroje často sestávají z různých oddělených částí a knihoven a vyžadují vlastní specificky nastavené integrované IDE[10]. Unity oproti tomu sestává z vlastního sofistikovaného vizuálního editoru, jenž je tvořen z herních scén a provazuje všechny datové zdroje a kód do interaktivních objektů. Editor je zejména účiný pro rychlé iterace vývoje, prototypování a testování. Spoustu objektů je možno upravovat v editoru i během hraní.

Účelem této práce není poskytnout kompletní návod pro tvorbu her ve vývojovém prostředí Unity ani výčet všech funkcí. Pro podrobnější informace o tomto nástroji vizte [9].

[10]IDE značí Integrated Development Environment, tedy vývojové prostředí (většinou pro psaní kódu).

Kapitola 3

Návrh

Tato kapitola pojednává o procesu návrhu diplomové práce a poskytuje teoretický základ pro samotnou implementaci, která je popsána v kapitole 4.

3.1 Analýza požadavků

Očekává se, že během tvorby hry v Unity budou rozšíření používat dva typy uživatelů.

- **Návrháři** - tvoří herní logiku prostřednictvím grafického rozhraní, jenž rozšíření poskytuje.
- **Programátoři** - programují v C# a rozšiřují funkčnost rozšíření. Implementují věci, jejichž tvorbu není možno automatizovat.

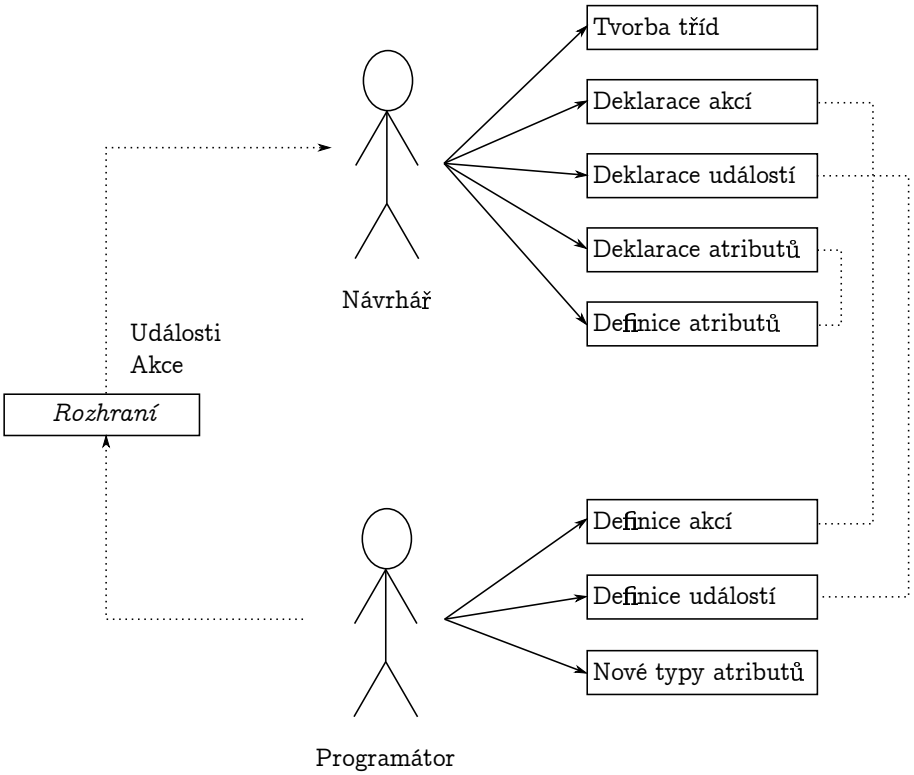
Výše uvedené není nutně vzájemně výlučné. Návrhář může být současně i programátorem dle potřeby. Současně se mohou v týmu vyskytnout i další role, které ovšem nejsou pro rozšíření relevantní a není je tedy třeba uvažovat. Ideální kompozice týmu a projektové řízení není předmětem této práce.

Na obrázku 3.1 je k vidění jednoduchý případ užití znázorňující návrháře i programátora. Jak již bylo řečeno, očekává se, že programátor pracuje na nižší vrstvě – „pod rozšířením“ a návrháři, který tvoří na vyšší vrstvě – „nad rozšířením“ – poskytuje skrz zavedená rozhraní definice, které návrhář deklarovat. Toho nezajímá, že akce *Přesun jednotky* na implementační úrovni spustí hledání cesty pomocí algoritmu A* nebo že se do vnitřní paměti jednotky uloží příkaz do fronty, tyto náležitosti řeší pouze programátor. Návrháři je poskytnuta akce, kterou prostřednictvím grafického rozhraní může použít bez nutné znalosti všech technických náležitostí samotné implementace.

3.2 Herní mechaniky

Při tvorbě počítačové hry je potřeba navrhnout a implementovat *herní mechaniky*. Herní mechaniky jsou nejdůležitější prvky tvořící výsledný produkt. Jedná se o interakce a vztahy, které nám zůstanou, pokud hru oprostíme od veškeré grafiky, estetiky a příběhu.

Herní mechaniky bývají – i v těch nejjednodušších počítačových hrách – poměrně komplexní. Taxonomie herních mechanik bývají nekompletní. Na jednu stranu mají tyto mechaniky jasně stanovaná pravidla, na druhé straně však často obsahují modely, které jsou založené na abstraktních představách. Různí autoři pak mohou používat různou taxonomii.



Obrázek 3.1: Příklad užití se dvěma hlavními účastníky – návrhářem a programátorem [Vlastní zdroj]

Někteří přistupují k této problematice ze striktně akademické perspektivy, jiní používají úhel pohledu praktičtější pro samotný návrh. Jesse Schell ve své knize *The art of game design: a book of lenses* [13] používá následující taxonomii:

- **Prostor** (Space) - Každá hra se odehrává v nějakém prostoru. Jakožto herní mechanika se jedná o matematický konstrukt, pro který jsou důležité tři vlastnosti: počet dimenzí, zda je prostor diskrétní (např. šachy) nebo kontinuální (např. závodní hra) a rozdělení na oblasti, které mohou být, ale nemusí, spojené.
- **Objekty, atributy a stavy** (Objects, Attributes, and States) - Objekty se obvykle nachází v prostoru, ale není tomu tak vždy. Jedná se o postavy, nepřátele či rekvizity, stejně tak ale i o datové objekty, srovnávací tabulky či schopnosti postav. Tyto objekty pak mají atributy – kategorické informace o objektu (např. rychlost, útok...). Každý tento atribut pak má nějaký stav – aktuální hodnotu v definovaném rozmezí.
- **Akce** (Actions) - můžou být operativní, např. *posun figurky vpřed* či *přeskočení jiné figurky* (tedy odpovídají na otázku „Co může hráč udělat?“) či výsledné, např. *donucení oponenta, aby se někam přesunul* či *obětování figurky* (ty mají smysl pouze v širším záběru na celou hru a často se jedná o strategické kroky).
- **Pravidla** (Rules) - nejdůležitější mechanika. Pravidla definují prostor, objekty, akce i důsledky daných akcí, včerně cíle hry. Dobrý cíl pak musí splňovat tři podmínky: být konkrétní, dosažitelný, a prospěšný.

- **Dovednost** (Skill) - každá hra nutí hráče, aby procvičoval určité dovednosti – ať už schopnost rychle reagovat, improvizovat či vymyslet vhodnou strategii. Zde záleží na žánru hry, což bylo probráno v kapitole 2.2.
- **Náhoda** (Chance) - náhoda zajišťuje nejistotu a nejistota je základem překvapení. Hráči bývají rádi příjemně překvapeni a navíc tato mechanika rozšiřuje opakovatelnost hraní, nicméně je potřeba tuto mechaniku používat opatrně, aby zcela nezastínila hráčovi schopnosti. Viz citát níže od Raphaela van Lieropa.

We've been learning that players in general want a game that rewards them for thinking, and doesn't penalize them with meaningless randomness. [...] Players embrace a challenge and don't mind failing as long as the failure seems fair.

— Raphael van Lierop[1]

Během tvorby rozšíření do Unity pro vizuální programování bylo potřeba vzít v úvahu, že návrhář bude tyto mechaniky (ať už přímo či nepřímo) vytvářet, upravovat a dále využívat.

3.3 Vlastní knihovna pro grafické rozhraní

Tvorba vlastního grafického rozhraní pro Unity (např. v rámci vlastního okna rozšíření) je v celku jednoduchá. Stačí vytvořit vlastní třídu, která dědí z `EditorWindow` a následně implementovat metodu `OnGUI`, která se volá při každém překreslení okna. V těle této metody se poté používá statická třída `GUI` nebo `GUILayout`, případně `EditorGUI` a `EditorGUILayout`, jenž obsahuje metody pro zobrazování nejrůznějších komponent.

Problém tohoto postupu je, že se jedná o procedurální přístup k vykreslení, nikoliv deklarativní. Programátor je nucen napsat posloupnost příkazů, která se při vykreslení provede, ale pakliže chce docílit udržování stavu, je nucen ukládat všechna data ručně. Zde je ukázka, jak použít textový box:

```
1  string myString = "Hello World";
2
3  void OnGUI()
4  {
5     myString = EditorGUILayout.TextField("Text Field", myString);
6  }
```

Blok kódu 3.1: Ukázka textového boxu ve vlastním okně

Jak je z tabulky 3.1 patrné, jakožto druhý argument statické metody `TextField` je nutno uvést obsah textového pole ve formátu řetězce. Samotná metoda pak vrací *nový* obsah, pokud došlo ke změně, nebo obsah starý. Programátor je pak nucen samotný textový řetězec udržovat v paměti prostřednictvím ručně zavedené externí proměnné `myString`.

Tohle řešení je vhodné pro jednoduchá grafická rozhraní s minimálním počtem ovládacích prvků. Jakmile ale začne programátor vytvářet komplexní rozhraní s nutností vlastního nastavování pozic a spoustou interních dat, začne být tohle řešení nepraktické.

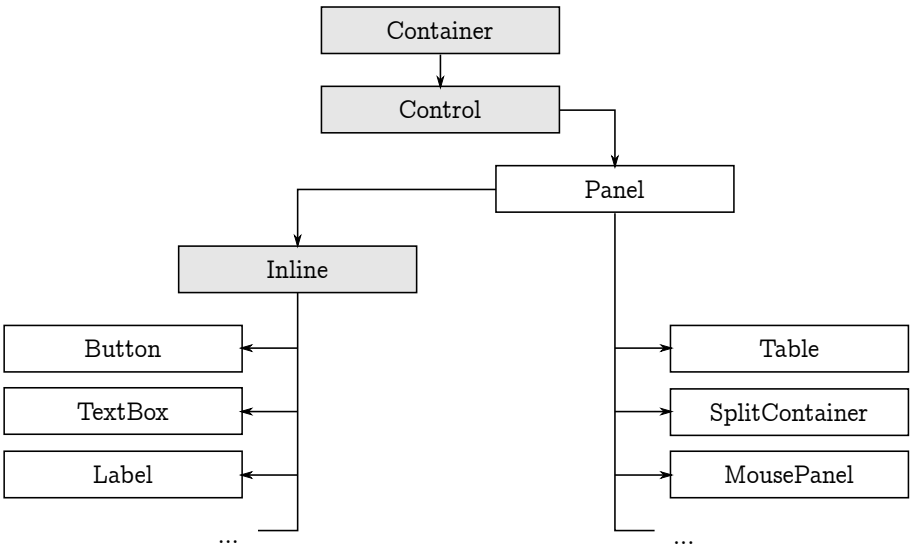
Proto bylo vytvořeno vlastní, objektově orientované řešení pro tvorbu uživatelského rozhraní insprované knihovnou WinForms z .NET Frameworku od společnosti Microsoft.

[1] Raphael van Lierop je zakladatel a ředitel herní společnosti The Long Dark.

Autor této práce se nepokoušel replikovat plnou funkčnost této knihovny, pouze implementoval některé základní třídy dle potřeb projektu a na základě vlastního uvážení. Toto řešení je dostupné pod prostorem jmen `OGUI` (Object GUI) v rámci celého projektu.

3.3.1 Rodičovské třídy OGUI

V rámci řešení OGUI byly implementovány 3 abstraktní třídy pro definici vykreslení a chování uživatelského rozhraní a jedna speciální třída, která abstraktní není, ale taktéž je rodičem mnoha dalších OGUI prvků – třída *Panel*. Na obrázku 3.2 je k vidění dědičnost tříd tvořících celou logiku OGUI řešení.



Obrázek 3.2: Schéma tříd tvořících OGUI, šedě jsou abstraktní třídy [Vlastní zdroj]

Rodičovské třídy z obrázku 3.2 mají následující účel:

- **Container** - nejvyšší komponenta. Implementuje logiku pro hierarchické zanořování komponent a určování (či změnu) jejich pořadí pro vykreslovací frontu.
- **Control** - potomek třídy Container. Implementuje pozicování a velikost komponent, stejně jako dokovací logiku a ukotvení. Neřeší samotné vykreslení, pouze výpočty.
- **Panel** - potomek třídy Control. Technicky není abstraktní třída – je možno vytvářet instance této třídy. Stará se o vykreslování a implementuje některé vlastnosti, které s tím souvisejí (např. barva pozadí).
- **Inline** - potomek třídy Panel. Opět se jedná o abstraktní třídu. Pouze implementuje logiku pro výpočet velikosti komponenty na základě obsahu (např. dle délky textu) a s tím související vlastnost AutoSize.

Z těchto čtyř (respektive v praxi ze třídy `Panel` a `Inline`) se následně implementují samotné grafické komponenty, které jsou blíže popsány v kapitole 3.3.2.

3.3.2 Znovupoužitelné prvky OGUI

Rozšíření ObjectGUI implementuje spoustu znovupoužitelných tříd. Některé implementují jednoduché prvky rozhraní ze statických Unity tříd `GUI` a `GUILayout`, pouze je rozšiřují do objektově orientované podoby a s patřičnými událostmi (např. *OnClick*, *OnChange*, *OnDoubleClick*...). Těmito prvky jsou například `Label`, `Button`, `CheckBox`, `NumBox` či `ComboBox`. Chování těchto komponent je inspirováno komponentami z knihovny .NET Framework.

Dále byly implementovány složitější komponenty pro specifické potřeby projektu. Jedná se o následující:

- **DiagramEntity** - Speciální komponenta, která se používá pro vykreslení entit diagramu tříd. Má v sobě zabudovanou mechaniku pro funkci *Drag&Drop*[2].
- **DialogConfirmationPanel** - Jednoduchý panel ukotvený na spodní části okna obsahující jedno nebo dvě tlačítka (dle potřeby) – pro potvrzení a/nebo pro zrušení dialogu. Součástí tohoto objektu je pak i událost *OnConfirm*.
- **SplitContainer** - Panel obsahující dva menší panely, které vyplňují plnou oblast svého rodiče a jejich poměrovou velikost je možno měnit dynamicky uchycením oddělující oblasti.
- **TabButtons** - Ukotvitelný panel obsahující vzájemně výlučná stavová tlačítka pro přepínání viditelnosti některých panelů v okně.
- **Table** Komplexní komponenta pro vykreslování tabulek. Jako součást této třídy byly dále implementovány pomocné třídy `TableColumn` a `TableRow`. Umožňuje vykreslení hlavičky a obsahuje logiku pro výběr řádku a s tím související události.
- **TreeView** - Komponenta pro vykreslení zadané hierarchické struktury, kdy jeden element je na samostatném řádku a jeho odsazení odpovídá hloubce zanoření. Obsahuje logiku pro otevírání a zavírání potomků vybraného elementu. Jako součást této třídy byly dále implementovány pomocné třídy `TreeViewItem` a `TView` (pouze privátní třída implementující *UnityEditor.IMGUI.Controls.TreeView*).

Jednoduchá ukázka použití prvků OGUI je v tabulce 3.2. Jak je vidno, stačí pouze jednou definovat existenci objektu v uživatelském rozhraní a navázat na objekt příslušnou obsluhu událostí specifikací metod, které se při spuštění události mají provést. Není třeba ručně porovnávat staré a nové hodnoty (či si hodnoty externě udržovat), jako tomu bylo v ukázce 3.1.

```
1  Button btn = new Button("Test");
2  btn.Position = new UPP.Position(10, 10);
3  btn.OnClick += Btn_OnClick;
```

Blok kódu 3.2: Ukázka tvorby tlačítka za použití OGUI

[2]Drag&Drop označuje funkcionalitu, kdy je možno prvek přesouvat pomocí kliknutí a držení levého tlačítka myši.

Současně návrhář nemusí řešit specifikaci vlastního objektu typu *GUIStyle* – většina relevantních vlastností souvisejících se vzhledem komponenty je nstavitelná prostřednictvím veřejných atributů objektu, ze kterých je následně objekt *GUIStyle* vytvořen interně uvnitř instance.

3.3.3 Pomocné třídy a výčtové typy

V rámci rozšíření grafického rozhraní OGUI byly vytvořeny i některé pomocné třídy a výčtové typy, jejichž tvorba instance nemá smysl mimo specifické prvky, ale v rámci těchto prvků často implementují důležitou funkcionalitu, která může být využívána i ve všech potomcích daných prvků.

Některé méně důležité interní třídy byly v rámci textu této práce vynechány pro zachování jednoduchosti a přehlednosti. Důležité třídy pro funkčnost OGUI jsou pak následující:

- **Anchor** - Třída určující ukotvení elementu, což se bere v úvahu při změně pozice a/nebo velikosti rodičovského objektu a je pak potřeba spočítat novou pozici všech potomků. Obsahuje událost *AnchorChanged*, jenž je vyvolána v případě změně ukotvení. Je možno libovolně kombinovat logické atributy *Top*, *Bottom*, *Left* a *Right*, které určují, zda je hrana elementu ukotvena ke stejné hraně svého rodiče.
- **DockStyle** - Nikoliv třída, ale pouze výčtový typ určující pozicování elementu podobně, jako u *Anchor*. Rozdílem je, že při použití *DockStyle* je pozice elementu nastavena na nejbližší možný bod určeného okraje rodičovského prvku a element je dále roztáhnut v protilehlém směru. Pokud je např. *DockStyle* nastaven na Top, pak je pozice automaticky nastavena na hodnotu `[0, 0]` a šířka elementu nastavena na šířku rodiče. Výška elementu zůstává zachována.
- **Position** - Třída popisující pozici elementu. Obsahuje atributy *X* a *Y*, stejně jako rutinu pro nastavování a získávání těchto hodnot a událost, jenž je při změně hodnoty vyvolána.
- **Size** - Třída popisující velikost elementu. Obsahuje atributy *Width* a *Height*. Opět je zde rutina nastavení a získání hodnot včetně patřičné události pro změnu hodnoty.

Dále za zmínku stojí speciální statická třída **ColorSchemes**, která obsahuje definice barev některých grafických komponent – a to pro tmavou a světlou verzi Unity editoru[3]. Vhodnou barvu pak při zavolání statické metody třída vybere automaticky dle toho, jakou verzi editoru právě uživatel používá.

3.3.4 Návrh vlastních oken

Tvorba vlastního okna se v editoru Unity provádí implementací vlastní třídy, která dědí od vestavěné třídy `EditorWindow`.

Okno je možno zobrazit pomocí zavolání metody *GetWindow* statické třídy *EditorWindow*, kdy se jako jediný parametr uvádí typ objektu okna (nikoliv instance). Pokud návrhář chce vytvořit vlastní položku v hlavním menu editoru, která dané okno otevře, je potřeba specifikovat metadatový atribut *MenuItem* k metodě, která se má zavolat při kliknutí na položku v menu (tedy metoda pro vyvolání okna).

[3]Unity editor obsahuje dvě barevné varianty – tmavou (černou) a světlou (stříbrnou). Tmavou variantu je možno používat pouze pokud uživatel vlastní *Pro* licenci Unity.

Dále je možno definovat funkci *OnGUI*, která je zavolána během každé iterace vykreslení okna. Zde se tedy implementují všechny prvky, které se v daném okně nachází.

```
class MyWindow : EditorWindow {
    [MenuItem ("Window/Moje okno")]
    public static void ShowWindow () {
        EditorWindow.GetWindow(typeof(MyWindow));
    }

    void OnGUI () {
        // Zde kod pro vykresleni komponent.
    }
}
```

Blok kódu 3.3: Ukázka tvorby vlastního okna v editoru Unity

Ukázka 3.3 obsahuje kód, který vytvoří novou položku *Moje okno* v menu *Window*. Po kliknutí na tuto položku se otevře okno *MyWindow*. Pokud instance tohoto okna ještě neexistuje, je automaticky vytvořena, jinak se použije poslední známá instance. To zajišťuje mimo jiné i uchování poslední pozice a velikosti okna.

Třída OGUI.Window

Pro potřeby diplomové práce a pro efektivní použití naimplementovaných komponent OGUI se tvorba oken použitím vestavěné třídy *EditorWindow* zdála nedostačující, proto byla vytvořena nová třída *Window* v prostoru jmen OGUI.

Tato třída svým potomkům poskytuje metody *PrepareGUI*, *BeforeDraw* a *AfterDraw*. *PrepareGUI* se zavolá právě jednou, a to při tvorbě instance nového okna. Očekává se, že se zde vytvoří potřebné OGUI objekty. Toto nebylo možné použitím klasického *OnGUI*, které poskytovala vestavěná třída *EditorWindow* bez toho, aby si návrhář externě zavedl proměnnou určující, zda již došlo k vytvoření instance nebo se jedná o první vykreslení okna.

Metody *BeforeDraw* a *AfterDraw* pak poskytují absolutní kontrolu nad vykreslením okna. *BeforeDraw* se zavolá na počátku vykreslení. Následuje pak automatické vykreslení všech komponent, jejichž instance návrhář vytvořil v metodě *PrepareGUI*. Na závěr, až jsou všechny prvky vykresleny, je zavolána metoda *AfterDraw*.

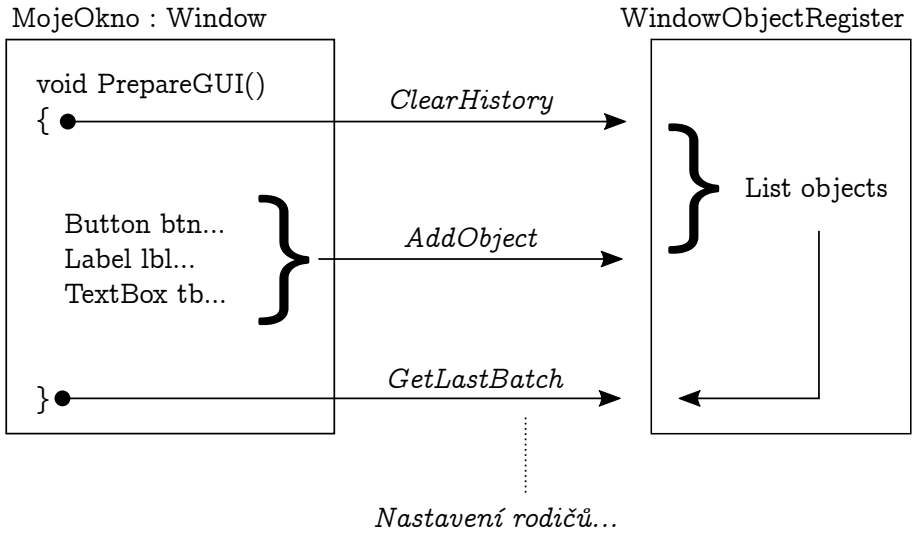
Metody *BeforeDraw* a *AfterDraw* jsou deklarovány jako virtuální, zatímco metoda *PrepareGUI* je abstraktní. To znamená, že *PrepareGUI* je potřeba povinně implementovat ve všech potomcích (vlastních oknech), zatímco *BeforeDraw* a *AfterDraw* implementovat není nutno, pokud obsluhu vykreslení okna návrhář na této úrovni nepožaduje.

Dále třída *Window* obsahuje logiku pro modální okna. Standardně Unity neumožňuje vyvolat modální okno s výjimkou jednoduchých dotazovacích dialogů. Modální okna jsou tedy v rámci OGUI pouze simulována, a to překreslením rodičovského okna poloprůhlednou černou vrstvou a návratem do modálního okna kdykoliv je detekováno zaměření okna rodičovského (událost *OnFocus*). To je pro potřeby rozšíření dostačující chování.

Modální okno je možno nastavit pomocí atributu *ModalChild* nebo *ModalParent*. Není nutno použít oboje – potomkovi či rodiči je druhý atribut do páru nastaven automaticky.

Třída OGUI.WindowObjectRegister

Pro usnadnění práce při vytváření instancí prvků rozhraní v metodě *PrepareGUI* třídy Window (kapitola 3.3.4), byla vytvořena pomocná statická třída *WindowObjectRegister*. Ta se stará o sběr všech nových instancí třídy *Container* (nejvyšší třída OGUI elementů). Na počátku tvorby instance nového okna je provedeno vyčištění interního seznamu objektů a dále je do tohoto seznamu automaticky umístěn každý nový objekt, jenž je v rámci *PrepareGUI* vytvořen. Na závěr metody PrepareGUI jsou pak všechny tyto objekty zkontrolovány a v případě, že některý objekt nemá rodiče, automaticky je jako rodič nastaveno samotné okno. Tím se eliminuje potřeba specifikovat rodiče, pokud je prvek na nejvyšší úrovni okna, což eliminuje zbytečné řádky kódu.



Obrázek 3.3: Schéma funkcionality třídy WindowObjectRegister [Vlastní zdroj]

Na obrázku 3.3 je znázorněna funkcionalita třídy *WindowObjectRegister* za předpokladu, že návrhář vytvoří vlastní okno a naimplementuje metodu *PrepareGUI* s prvky uživatelského rozhraní. Třída *Window* se pak postará o vyplnění prázdných rodičů prvků GUI.

3.4 Serializace dat

Serializace v kontextu počítačové vědy je proces převodu datových struktur či objektů do podoby, kterou je možno uložit (např. do souboru nebo do vyrovnávací paměti) či přenést (např. napříč počítačovou sítí) a následně opět rekonstruovat do původní podoby bez ztráty dat [11].

Během používání rozšíření návrhářem vzniká spousta dat, které je potřeba serializovat. Každá definice třídy, včetně svých atributů, musí být serializovatelná, jelikož pro ukládání dat ve vývojovém prostředí Unity byl v rámci této práce zvolen tzv. *ScriptableObject*.

ScriptableObject je třída, ze které je možno odvodit vlastní třídy, které nepotřebují být připojeny k herním objektům ve scéně. To je nejlépe využitelné pro zdrojové objekty, které slouží jako kontejnery pro ukládání dat [19]. Samotné vytváření těchto objektů v editoru se většinou provádí za použití reflexe, která je dále popsaná v kapitole 3.5.

Aby nedošlo po zavření editoru ke ztrátě dat, jenž jsou udržovány v podobě polí potomka třídy *ScriptableObject*, je potřeba, aby všechna tato pole byla serilizovatelná a Unity tedy dokázalo jejich hodnotu správně uložit a opětovně načíst při příštím spuštění editoru.

Objekty reprezentující navržené třídy a jejich atributy, které jsou popsány v kapitole 4.1, tedy museli být navrženy tak, aby byla možná jejich kompletní serializace. To bylo zařízeno implementací rozhraní `IFormatter`, které předepisuje funkcionalitu pro formátování serializovatelných objektů, konkrétně obsahuje předpis pro metody `Serialize` a `Deserialize`.

3.4.1 Třída Serializer

Během vývoje se často stávalo, že bylo potřeba některá data serializovat či deserializovat ručně, bez použití mechanik, které Unity nabízí v kontextu třídy *ScriptableObject*. Byla tedy vytvořena statická třída *Serializer*, která zapouzdřuje metody tříd *XmlSerializer*, *StreamWriter* a *StreamReader* (včetně vytvoření instancí těchto tříd v případě potřeby).

Samotná třída *Serializer* pak poskytuje následující metody:

- **FromString** - Metoda pro převedení serializovaného textu do objektu v paměti. Je možno použít generickou podobu této metody a typ specifikovat, nebo (v případě, že v době překladu typ není znám) použít obecnou podobu metody, kdy se typ předává pouze jako text prostřednictvím druhého parametru. Serializér se pak sám postará o nalezení typu prostřednictvím reflexe.
- **ToString** - Metoda pro převedení objektu v paměti do textového řetězce. Opět je možno použít generickou podobu metody se specifikovaným typem objektu, nebo použít obecnou metodu se dvěmi parametry, kde druhým parametrem je řetězec popisující typ, který serializér nalezne reflexí.
- **EncapsulateStrings** - Jednoduchá metoda pro převod pole textových řetězců do jednoho velkého, serializovaného řetězce. Místo pole řetězců je taktéž možno jako parametr použít libovolný počet normálních řetězců (implementace klíčovým slovem *params*).
- **DecapsulateStrings** - Převede jeden serializovaný řetězec do pole menších řetězců.

3.5 Reflexe

Reflexe je vlastnost programovacího jazyka (v kontextu objektově orientovaných jazyků) zjistit za běhu údaje o určitém programovém objektu, respektive o programu a jeho syntaktické struktuře.

```
1  // Bez pouziti reflexe:
2  Foo foo = new Foo();
3  foo.PrintHello();
4
5  // S pouzitim reflexe:
6  Object foo = Activator.CreateInstance("complete.classpath.and.Foo");
7  MethodInfo method = foo.GetType().GetMethod("PrintHello");
8  method.Invoke(foo, null);
```

Blok kódu 3.4: Ukázka reflexe v jazyce C#

V tabulce 3.4 je ukázka toho, jak reflexe funguje v jazyce C#, který je používán při programování her ve vývojovém prostředí Unity či při jeho samotném rozšiřování. V tomto příkladu je vytvořena instance `foo` třídy `Foo` a dále je vyvolána její metoda `PrintHello`.

Reflexi je možno použít pro pozorování a úpravu vykonávání programu za běhu. Reflexivně orientovaná programová komponenta může monitorovat konání uzavřeného kódu a může modifikovat sama sebe pro dosažení požadovaných výsledků.

V objektově orientovaných jazycích, jako je C# nebo Java reflexe umožňuje inspekci tříd, rozhraní, polí a metod a také umožňuje tvorbu instance nových objektů a vyvolání metod, jak bylo ukázáno v příkladu 3.4. Dále je možno reflexi v těchto jazycích použít pro přepis pravidel přístupnosti k polím [6].

```
1  Type parent = Type.GetType("RodicovskaTrida");
2  Type[] types = Assembly.GetExecutingAssembly().GetTypes();
3  Type[] inheritingTypes = types.Where(t =>
4    parent.IsAssignableFrom(t));
```

Blok kódu 3.5: Zjišťování potomků v jazyce C# pomocí reflexe

V rámci této diplomové práce se reflexe využívá pro zajištění modularity některých částí rozšiření – například je možno definovat vlastní typy atributů tříd, jak je popsáno v kapitole 4.2.2. Samotné načtení všech možných typů je následně zajištěno pomocí reflexe, kdy se zjistí všechny podtřídy, které dědí z definované abstraktní rodičovské třídy, jak je ukázáno v příkladu v tabulce 3.5. Nutno podotknout, že v tomto případě bude navrácena i rodičovská třída samotná a případně i použitá rozhraní.

Kapitola 4

Implementace

Tato kapitola pojednává o implementaci hlavní části diplomové práce – tedy samotný editor tříd, které tvoří herní logiku, editor šablon, zpracovávání atributů a závěrečný překlad návrhářem vytvořených konstrukcí.

Na závěr kapitoly je popsán způsob, jakým je možno přeložený výsledek použít v rámci scény Unity a případně výsledné třídy dále rozšiřovat.

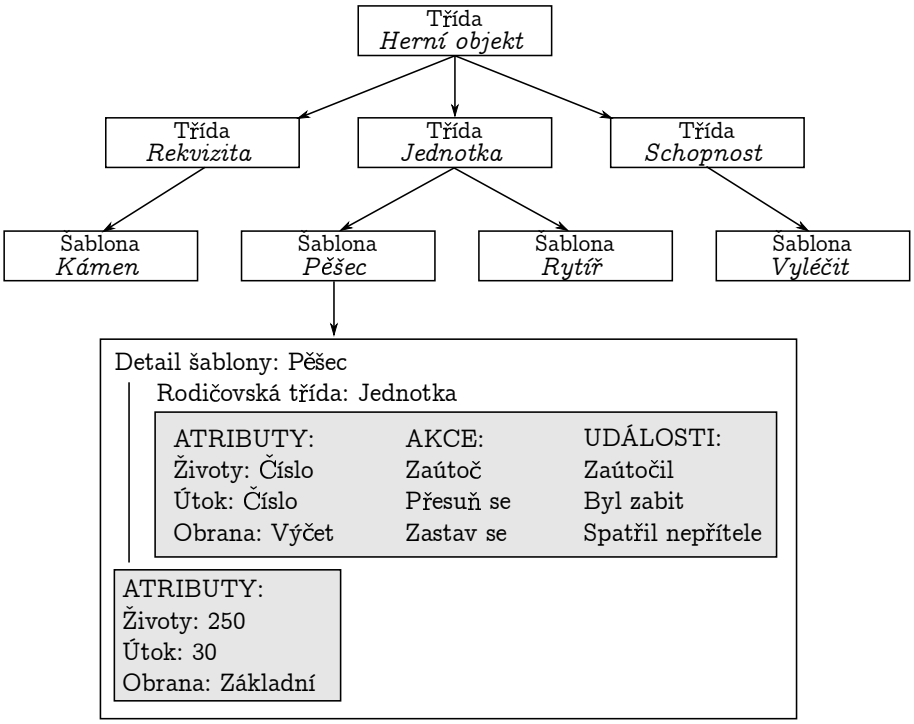
4.1 Třídy herní logiky

Tato kapitola se zabývá implementací rozhraní pro návrh herních objektů, které pak tvoří herní logiku. Očekává se, že návrhář bude schopen vytvářet velkou spoustu abstraktních i neabstraktních tříd, důraz je tedy tvořen na snadnost a přehlednost. Jak již bylo řečeno v kapitole 3.1, je potřeba, aby měl návrhář možnost zobrazit si třídy na globální úrovni včetně jejich závislostí (dědičnost), ale stejně tak je třeba, aby bylo možno zobrazit detail každé třídy včetně veškerých atributů (vlastností) a metod (akcí a událostí).

V rámci tvorby pak návrhář vytváří dva typy entit, které jsou definovány následovně:

- **Třídy** - Entity popisující prvky tvořící herní logiku na konceptuální vrstvě. Nejedná se tedy o prvky, ze kterých by bylo možno vytvářet instance přímo v herní scéně Unity, místo toho se očekává, že z těchto tříd budou vytvořeny a zděděny další podtřídy. Na úrovni objektově orientovaných jazyků se v podstatě jedná o abstraktní třídy. Deklarují atributy a mohou definovat akce a události.
- **Šablony** - Entity popisující výsledné, konkrétní objekty, které často interagují s herní scénou Unity. Na úrovni objektově orientovaných jazyků se jedná o klasické třídy, které nejsou abstraktní a je tedy možno z nich vytvářet instance objektů. Definují konkrétní hodnoty atributů. Akce a události pouze přebírají od svých rodičovských tříd.

Na obrázku 4.1 je k vidění zjednodušený příklad toho, jak by mohla vypadat hra při použití rozšíření. Třídy určují herní logiku a mechaniky, které bude hra využívat. Z hlediska atributů však obsahují pouze deklaraci, nikoliv definici. Samotné hodnoty atributů jsou určeny až v konkrétních herních šablonách. Z těchto šablon je ve finální fázi možno vytvářet samotné individuální instance v herní scéně.



Obrázek 4.1: Zjednodušený příklad návrhu hry prostřednictvím rozšíření [Vlastní zdroj]

4.1.1 Vnitřní reprezentace tříd

Všechny návrhářem vytvořené třídy jsou reprezentovány vlastní instancí třídy *ClassRecord*, která dědí od třídy *ScriptableObject*, jenž byla blíže popsána v kapitole 3.4. Tato třída obsahuje jednoduché atributy, které udržují informace o třídě, jako je její čitelné jméno, strojové jméno či jaký je její rodič.

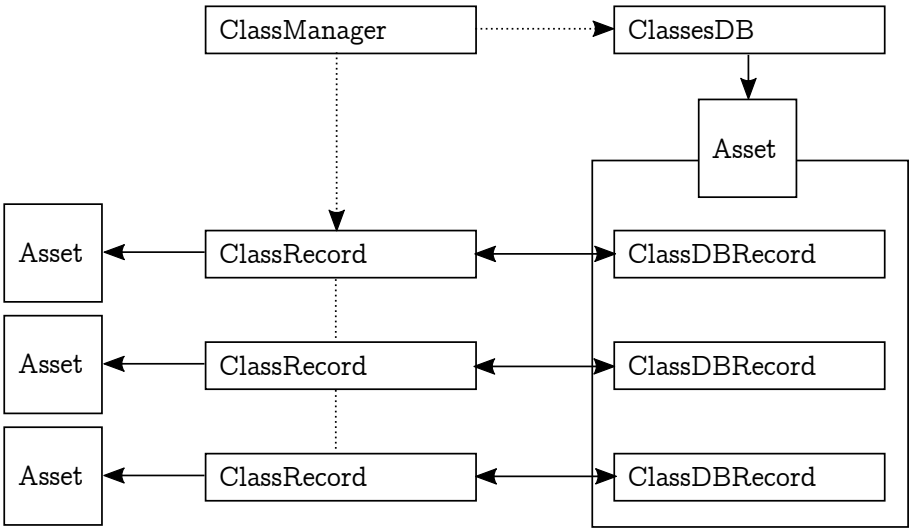
Jelikož se tyto záznamy serializují, není možno používat reference na objekty. Reference na objekt (či ukazatel) je pouze adresa v paměti. Ta může být jiná při příštím zapnutí editoru a deserializaci dat. Proto se všechny údaje ukládají pouze jako jednoduché datové typy (řetězec, číslo...) a konzistence dat je pak v režii manažera herních tříd, který je blíže popsán v kapitole 4.1.2.

Samotné vytvoření instance *ClassRecord* ovšem nezajistí perzistenci této třídy po zavření a znovuotevření editoru. Perzistence je zajištěna vytvořením souboru typu `.asset`, který mimo jiné může sloužit jakožto skladiště serializovaných objektů typu *ScriptableObject*. Vytvoří-li tedy návrhář novou herní třídu *Unit*, automaticky se na předem určeném místě vytvoří soubor *Unit.asset* obsahující serializovaný záznam objektu *ScriptableObject*, který danou třídu popisuje. V případě, že návrhář provede ve třídě změny, ty jsou následně uloženy i do `.asset` souboru.

Pro snadnou manipulaci s vytvořenými třídami bylo potřeba zařídit snadný způsob, jak zjistit výčet všech těchto tříd. Toho by bylo možno dosáhnout pomocí reflexe (zjištění všech instancí třídy *ClassRecord* nebo pomocí vstupně-výstupních operací (zjištění `.asset` souborů v patřičné složce), nicméně jako ideální se jevilo vytvořit interní databázi, která si bude výčet všech vytvořených tříd udržovat formou zjednodušených záznamů. Proto byly vytvořeny třídy *ClassesDB* a *ClassesDBRecord*.

Databáze *ClassesDB* je pouze jednoduchý *ScriptableObject*, který obsahuje generický list objektů *ClassesDBRecord*. Třída *ClassesDBRecord* pak obsahuje čitelné a strojové jméno vytvořené třídy. Databáze je při každé změně uložena a serializována do vlastního, speciálního `.asset` souboru.

Manažer herních tříd, který je popsán v kapitole 4.1.2 pak ke každému `.asset` souboru vytváří i patřičný záznam v databázi *ClassDB*.



Obrázek 4.2: Schéma vnitřní reprezentace tříd [Vlastní zdroj]

4.1.2 Manažer herních tříd

Manažer herních tříd je speciální statická třída, která se stará o tvorbu a správu potřebných instancí záznamových tříd a `.asset` souborů v okamžiku, kdy návrhář vytvoří, edituje nebo smaže herní třídu (nejčastěji prostřednictvím editoru tříd, viz kapitola 4.1.3).

Součástí manažera tříd je mimo jiné i reference na databázi *ClassesDB*. Pokud je tato reference prázdná, manažer se postará o vytvoření nové instance včetně tvorby patřičného `.asset` souboru. K tomu slouží privátní metody *CreateDatabase* a *LoadDatabase*, které jsou zavolány automaticky při pokusu o získání prázdné reference (*get*).

- **LoadDatabase** se pokusí načíst asset na předem definovaném místě. Pakliže se tohle nepodaří (asset neexistuje), je automaticky zavolána metoda *CreateDatabase*. Pokud se načtení podaří, pouze se nastaví reference databáze na instanci načtenou z patřičného `.asset` souboru.
- **CreateDatabase** nejprve vytvoří novou instanci typu *ScriptableObject*, konkrétně v tomto případě *ClassesDB*. Dále pro tuto instanci asociuje vlastní *.asset* soubor na patřičném místě, do kterého se instance serializuje. Následně se reference databáze nastaví na vytvořenou instanci.

Pomocí reference na databázi je pak možno přistupovat ke všem vytvořeným herním třídám, nicméně pouze v podobě objektů typu *ClassDBRecord*, nikoliv *ClassRecord*. Pokud programátor potřebuje přistoupit k podrobnějším údajům o třídě, jako je např. její popis nebo rodičovská třída, je třeba nejprve načíst kompletní záznam *ClassRecord* pomocí strojového jména třídy, který je získán z *ClassDBRecord*.

Manažer herních tříd poskytuje následující veřejné metody:

- **GetClassByMachineName** - načte instanci *ClassRecord* pomocí strojového jména herní třídy. Pokud taková třída neexistuje, vrací `null`.
- **ClassExists** - vrací logickou hodnotu `true` nebo `false` dle toho, jestli herní třída s daným strojovým jménem již existuje, nebo nikoliv. Zde se z bezpečnostních důvodů testuje existence `.asset` souboru, nikoliv pouze záznam v databázi.
- **FindChildren** - Najde všechny potomky specifikované třídy. Pracuje na úrovni načtených záznamů *ClassRecord*, jelikož ve zjednodušených záznamech *ClassDBRecord* se údaj o rodičovské třídě nevyskytuje.
- **Create** - Vytvoří novou instanci záznamu herní třídy a k tomu patřičný `.asset` soubor na předem specifikovaném místě. Dále vytvoří záznam v databázi. Při použití nepovinného atributu *checkForExistence* před tvorbou nejprve ověří, zda již třída s daným strojovým jménem neexistuje a případně vyvolá chybový dialog (nepovažuje se za výjimku).
- **Edit** - Upraví údaje existující herní třídy. Umožňuje i změnu strojového jména – v takovém případě je provedena speciální rutina pro ověření, zda tato třída již neexistuje a manažer upraví i patřičná jména souborů a navíc jsou upraveni všichni potomci dané třídy – jakožto rodič je jim nastaveno nové jméno třídy. Na závěr jsou upraveny záznamy atributů (kapitola 4.2) pro zachování konzistence dat. O to se už ale stará manažer vlastností (*PropertyManager*), kdy je pouze zavolána jeho metoda *RenameClassRutine*.
- **Remove** - Smaže herní třídu a s ní patřičný záznam z databáze i `.asset` soubor. Je možno použít nepovinný parametr *confirmationDialog*, který zařídí zobrazení dialogu pro potvrzení smazání. Dalším nepovinným parametrem je pak *fixChildren*, který zařídí nastavení rodiče potomků smazené třídy na rodiče smazané třídy. Tedy pokud *Třída 3* dědí od *Třída 2* a ta dědí od *Třída 1*, v případě smazání druhé třídy se jako rodič třetí třídy nastaví *Třída 1*.
- **ForceSave** - Vynucení uložení specifikované herní třídy do `.asset` souboru. Většinou tohle není potřeba dělat, uložení je prováděno automaticky. V některých případech ale tato metoda může být žádoucí, pokud je z nějakého důvodu potřeba upravit herní třídu ručně mimo manažera herních tříd.

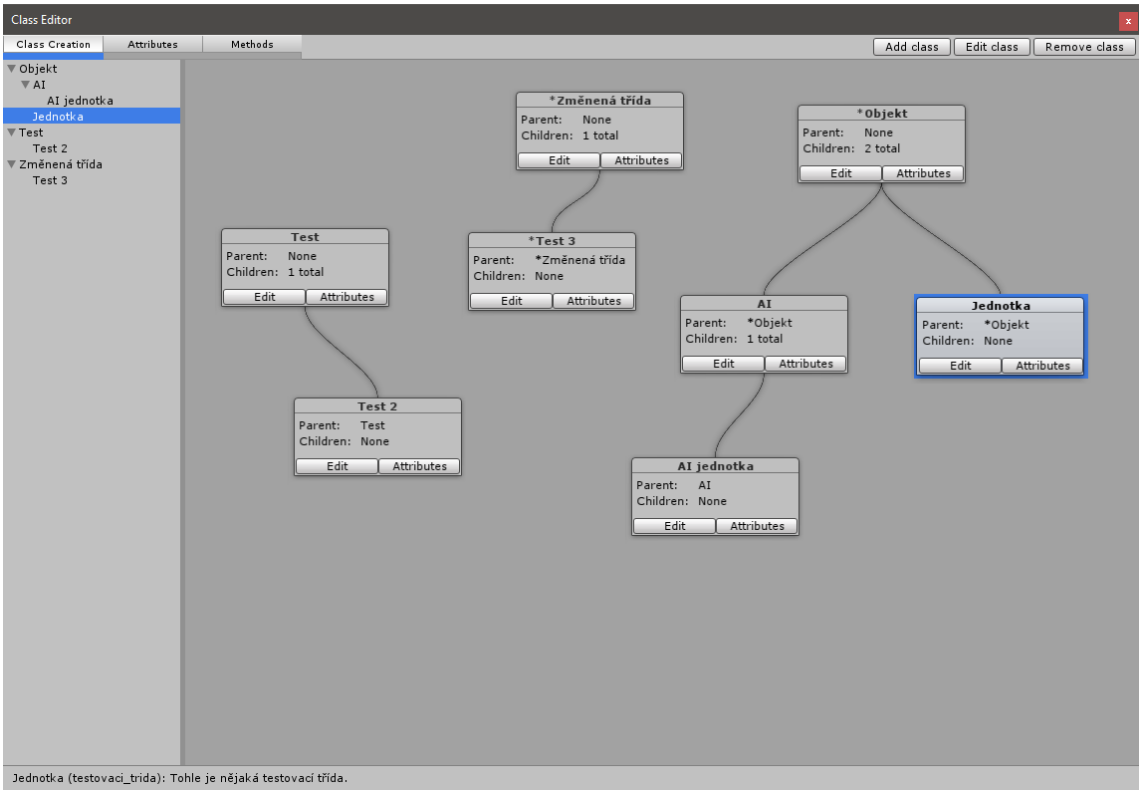
Databáze záznamů herních tříd uvnitř třídy *ClassManager* je založena na principu návrhového vzoru Singleton[1], který byl specificky upraven pro potřeby projektu. Tento návrhový vzor automatizuje tvorbu instance objektu a omezuje počet těchto instancí na jedna.[5]

4.1.3 Editor tříd

Editor tříd je dostupný v nové nabídce „Unity++“ v hlavním menu editoru Unity, a to pod položkou „Class Editor“. Okno editoru tříd je znázorněno na obrázku 4.3.

Editor tříd sestává z několika částí. Horní panel obsahuje záložky pro přepínání aktuálního režimu – návrh tříd, atributů nebo metod. V rámci této kapitoly je popsána záložka samotného návrhu tříd.

[1] Někdy též česky „Jedináček“.



Obrázek 4.3: Ukázka editoru tříd [Snímek obrazovky]

Dále je okno rozděleno do dvou oblastí. Po levé straně se nachází panel s hierarchickou strukturou již vytvořených herních tříd, zatímco v hlavní části se pak nachází plátno s diagramem těchto tříd. Součástí každé třídní entity je její jméno, rodičovská třída a počet potomků. Třídy jsou propojeny dle jejich dědičnosti.

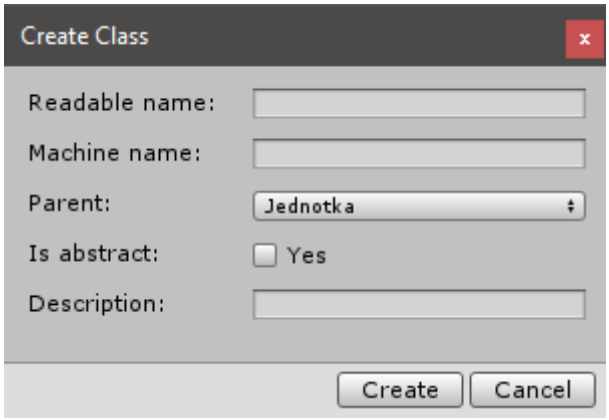
Entity tříd v hlavním plátně je možno libovolně přesouvat. Pozice těchto entit je automaticky ukládána (součást instancí *ClassRecord*), takže i po vypnutí a zapnutí editoru je vše tak, jak to návrhář ponechal.

Vpravo nahoře se nachází tlačítka pro vytvoření nové třídy, editaci a smazání třídy. Třídu je také možno editovat kliknutím na tlačítko „Edit“ patřičné entity, nebo dvojím poklepáním do hierarchivké stromové struktury.

Jak je ukázáno na obrázku 4.4, při tvorbě nebo editaci nové herní třídy je potřeba vyplnit následující údaje:

- **Readable name** - Čitelné jméno třídy. Může obsahovat libovolné znaky.
- **Machine name** - Strojové jméno třídy. Musí odpovídat specifikaci jazyka C# o jméně tříd, tedy nesmí obsahovat speciální znaky, nesmí začínat číslem, atp. Pro ověření správnosti jména byla implementována statická třída *MaskTemplates*, která obsahuje veřejný řetězec *MachineName* sloužící jako regulární výraz[2], kterému musí jméno odpovídat.
- **Parent** - Rodičovská třída. Je možno zvolit „<None>“ pokud třída žádného rodiče nemá.

[2]Konkrétní výraz je: `^[\p{Lu}\p{Ll}\p{Lt}\p{Lm}\p{Lo}\_]\w*$`, kde \p odpovídá třídám znaků.[7]



Obrázek 4.4: Ukázka tvorby nové herní třídy [Snímek obrazovky]

- **Is abstract** - Určuje, zda je možno z této třídy vytvořit šablonu či nikoliv. Abstraktní třídy mohou být pouze použity jako rodič jiných tříd, ale není z nich možno vytvořit šablonu (a tedy ve výsledku ani instanci).
- **Description** - Popis herní třídy.

Další fází po vytvoření herních tříd je obvykle tvorba atributů.

4.2 Atributy tříd

Atributy byly původně navrženy pouze jakožto jednoduchá data reprezentovaná proměnnou určitého typu. To se však ukázalo jako zcela nedostačující. Jak již bylo naznačeno v kapitole 4.1, atributy neobsahují pouze jednu hodnotu, ale i dodatečné informace o této hodnotě, jako je povolený rozsah či výchozí stav.

Jméno: Typ:	Životy Číslo		Jméno: Typ:	Útok Číslo
Minimální hodnota: Maximální hodnota: Výchozí hodnota:	0 1000 100		Minimální hodnota: Maximální hodnota: Výchozí hodnota:	0 200 10
Skutečná hodnota šablony:	250		Skutečná hodnota šablony:	30

Tabulka 4.1: Příklad atributu stejného typu pro různá data

V tabulce 4.1 je znázorněn příklad toho, kdy je jeden typ atributu (číslo) použit pro reprezentaci rozdílných dat – v tomto případě útok a životy. Životy jednotky je v tomto případě možno udržovat v rozsahu 0 – 1000, nicméně útok pouze 0 – 200. Tuto informaci si každý atribut musí taktéž udržovat.

Dodatečné údaje k atributům jsou ovšem jedinečné k danému typu. U čísla dává smysl udržovat si minimální a maximální hodnotu, oproti tomu u řetězce je smysluplnější maximální délka. Výčtové typy pak na úrovni šablon deklarují všechny možné hodnoty, ze kterých pak návrhář na úrovni herních modelů vybírá.

Ve výsledku jsou tato „metadata“ atributů tedy zcela v režii třídy, která daný atribut reprezentuje a není možno navrhnout univerzální řešení pro všechny typy. Očekává se také možnost vytvářet typy zcela nové pro potřeby vývoje konkrétní hry.

Byla tedy vytvořena abstraktní třída `PropertyBase`, která obsahuje deklarace potřebných vlastností a metod, jenž musí samotná třída pro konkrétní typ definovat. Mimo jiné je potřeba definice i samotného grafického rozhraní pro změnu hodnoty (pro řetězec dává smysl dlouhý box, pro číslo posuvník či oblast s šipkami inkrementace a dekrementace, atp.) – to se dělá pomocí metody *GUIForSetting* pro nastavování atributu na úrovni třídy (tedy deklarace) a metody *GUIForValue* pro definici konkrétních hodnot na úrovni šablon. Třída atributu konkrétního typu se pak sama stará o vykreslení relevantního rozhraní pro editaci hodnoty.

Třída `PropertyBase` implementuje rozhraní `ISerializable` pro možnost serializace instancí této třídy. Dále však deklaruje abstraktní metody *Serialize* a *Deserialize*, které musí každá nová třída, jenž z PropertyBase dědí, implementovat. Očekává se pouze serializace hodnot a metadat, která jsou specifická pro daný datový typ. Vše ostatní serializuje samotná PropertyBase.

Ze třídy `PropertyBase` dědí generická třída `Property<T>`. Ta deklaruje navíc některé abstraktní metody, u kterých se očekává, že je znám datový typ, se kterým atribut pracuje. Konkrétně se jedná o metody: *Get*, *Set*, *GetDefault*, *Pack* a *Unpack*.

Při tvorbě vlastního typu atributu, která je popsána v kapitole 4.2.2, se očekává, že bude programátor vycházet z abstraktní generické třídy `Property<T>`.

4.2.1 Manažer atributů

Manažer atributů (či manažer vlastností, *PropertyManager*) je stejně jako v případě manažera herních tříd (viz kapitola 4.1.2) statická třída, která obstarává tvorbu a správu instancí záznamových tříd a `.asset` souborů kdykoliv návrhář vytvoří, edituje či smaže atribut herní třídy.

Manažer vlastností obsahuje referenci na databázi *PropertiesDB* a obdobně, jako se manažer herních tříd stará o tvorbu a načtení této databáze z patřičného `.asset` souboru, i manažer vlastností tohle obstarává pro svou vlastní databázi.

Dále pak manažer vlastností obsahuje následující veřejné metody:

- **GetPropertyByMachineName** - Na základě strojového jména třídy a atributu načte patřičnou instanci *PropertyRecord*. Očekává se, že je možno mít stejná strojová jména dvou různých atributů pakliže se nachází ve dvou různých třídách a tyto třídy od sebe přímo či nepřímo nedědí. Proto je potřeba specifikovat i jméno třídy – pouze strojové jméno atributu nemusí být nutně jedinečné.

- **PropertyExists** - Vrací logickou hodnotu `true` nebo `false` dle toho, jestli atribut herní třídy s daným strojovým jménem již existuje, nebo nikoliv. Opět se z bezpečnostních důvodů testuje existence `.asset` souboru, nikoliv pouze záznam v databázi atributů.

- **PropertiesOfClass** - Najde všechny atributy specifikované třídy a ty pak vrací formou listu dvojic, kdy klíčem je objekt typu *PropertyDBRecord* a hodnotou je objekt typu *PropertyRecord*.

- **FindPropagation** - Najde všechny atributy, které propagují hodnoty od jiného atributu. To se často používá u dvojic atributů, kdy jeden označuje maximum a druhý

pak aktuální hodnotu instance (např. maximální a aktuální počet životů), nicméně využití záleží na návrháři. Vrací list objektů typu *PropertyDBRecord*.

- **RemoveAll** - Smaže všechny atributy patřící specifikované třídě.
- **RenameClassRutine** - Spustí rutinu pro přejmenování třídy. Nejčastěji voláno z manažeru tříd, kdy je použita metoda *Edit* a nové strojové jméno třídy se neshoduje se starým. Pak je potřeba změnit i záznamy všech atributů patřící této třídě, což už je v režii manažera atributů.
- **ClonedProperty** - Vytvoří kopii atributu jakožto nový objekt. Používá se při editaci atributu, kdy návrhář pracuje s kopií a změny se projeví až pokud klikne na tlačítko uložit.
- **Create** - Vytvoří novou instanci záznamu atributu herní třídy a k tomu patřičný `.asset` soubor. Ten je obvykle zanořený v adresáři se strojovým jménem třídy, které atribut náleží, aby nedošlo k problémům v případě stejného jména atributu u různých tříd. Dále vytvoří záznam v databázi. Při použití nepovinného parametru *checkForValidity* navíc kontroluje, jestli už atribut ve třídě neexistuje a jestli třída jako taková (se specifikovaným jménem) je validní, a případně zobrazí chybový dialog (nevyvolá výjimku).
- **Edit** - Upraví údaje existujícího atributu specifikované herní třídy. Opět umožňuje i dodatečnou změnu strojového jména – v takovém případě je provedena rutina pro ověření, zda atribut s novým jménem ve třídě již neexistuje a pokud ne, manažer upraví patřičná jména souborů a navíc jsou upraveny všechny atributy, které z editovaného atributu propagují hodnotu – jakožto rodičovská hodnota je jim nastaveno nové jméno atributu.
- **Remove** - Smaže atribut z herní třídy a s ním patřičný záznam z databáze i `.asset` soubor. Je možno použít nepovinný parametr *confirmationDialog*, který zařídí zobrazení dialogu pro potvrzení smazání atributu. Dalším nepovinným parametrem je pak *fixPropagation*, který upraví atributy, které od smazaného atributu propagují hodnoty, a propagaci v nich zcela vypne.

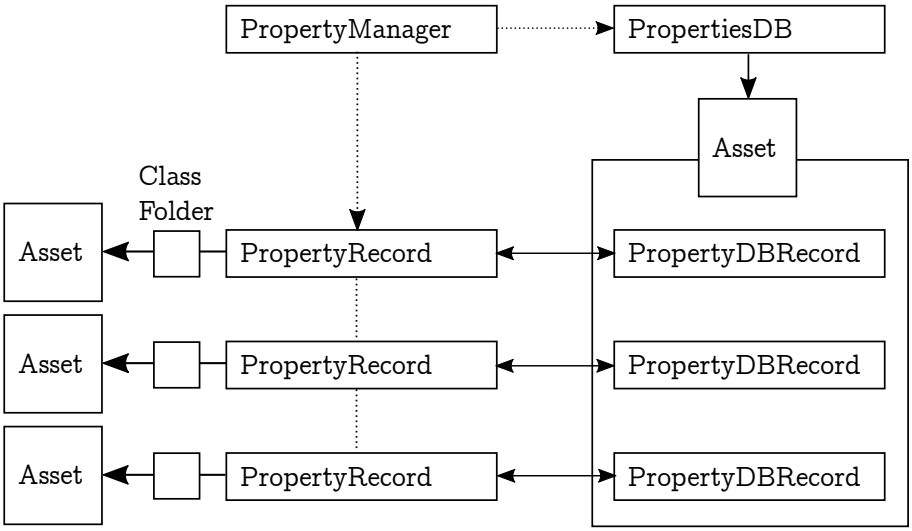
Na obrázku je znázorněna vnitřní reprezentace vytvořených atributů, která je podobná reprezentaci samotných herních tříd. `.asset` soubory jsou pouze zanořeny do adresářů dle strojového jména třídy, které náleží, samotná logika správy záznamů databáze a potřebných souborů je však stejná.

4.2.2 Tvorba vlastních datových typů

Rozšíření je vytvořeno na modulárním principu, který umožňuje tvorbu vlastních datových typů, které je možno použít pro atributy herních tříd.

Pokud si vývojář přeje vytvořit vlastní datový typ, je třeba implementovat novou třídu, která bude dědit z generické třídy `Property<T>`, kde typem je výsledná třída či datový typ, který vzniká po kompilaci. Třída `Property<Int32>` tak např. popisuje atribut herní třídy se všemi metadaty a metodami pro vykreslení uživatelského rozhraní, jehož výsledkem pak po překladu je vlastnost objektu typu `Int32`, tedy 32-bitové celé číslo.

Třída `Property<T>` a s tím související rodičovská třída `PropertyBase` jsou abstraktními třídami, které si vyžadují implementaci následujících metod:



Obrázek 4.5: Schéma vnitřní reprezentace atributů [Vlastní zdroj]

- Základní metody datového typu:
  - **Get** - Návrat hodnoty atributu. Buďto bez parametru, pak se vrací hodnota na indexu 0 (vhodné pokud se očekává, že atribut udržuje pouze jednu hodnotu), nebo právě jeden parametr, a to index.
  - **GetDefault** - Získání výchozí hodnoty atributu. Je možno dát návrháři možnost výchozí hodnotu nastavit prostřednictvím metody *GUIForSetting*, pokud je to žádoucí (např. řetězec typicky neobsahuje možnost nastavit výchozí hodnotu, ale číslo ano).
  - **Set** - Nastavení hodnoty atributu. Druhý nepovinný parametr určuje index, výchozím indexem je hodnota 0.
  - **Pack** - Zabalení všech hodnot do pole řetězců. Pokud atribut obsahuje pouze jednu hodnotu, vrací pole řetězců o velikosti 1. Neřeší metadata.
  - **Unpack** - Rozbalení pole řetězců a nastavení patřičných hodnot atributu. Neřeší metadata.
- Metody pro serializaci a deserializaci dat:
  - **Serialize** - Serializace všech hodnot atributu včetně metadat, jako je výchozí hodnota, minimum, maximum či cokoliv, co pro daný datový typ dává smysl (je v režii autora datového typu).
  - **Deserialize** - Deserializace řetězce a nastavení všech dat a metadat atributu.
- Metody pro grafické rozhraní a komunikaci s návrhářem:
  - **GetDisplayValue** - Převede nastavené hodnoty atributu do čitelné podoby řetězce. Každý datový typ musí podporovat zobrazit svůj stav alespoň částečně řetězcem, např. u barvy dává smysl použít RGB zápis[3], u časového razítka pak např. převedení do čitelné podoby času a data, atp.

[3]RGB zápis vyjadřuje barvu pomocí hodnoty červené, zelené a modré složky.

- **GUIForSetting** - Určuje rutinu, která má být provedena pro deklaraci grafického rozhraní pro nastavení metadat atributu (tedy nikoliv konkrétních hodnot) na úrovni tříd. Zde se např. může jednat o výchozí hodnotu, minimum, maximum, maximální délku, atp.
- **SaveGUIForSetting** - Metoda, která má být provedena při uložení metadat atributu.
- **GUIForValue** - Určuje rutinu, která má být provedena pro deklaraci grafického rozhraní pro nastavení konkrétních hodnot atributu na úrovni šablon. Neřeší se zde přecházení mezi indexy, to je implementováno automaticky v patřičném okně.
- **GUIForValueSetPage** - Metoda, která je zavolána v okamžiku, kdy návrhář v rámci rozhraní pro nastavení hodnot přešel na jiný index a je tedy potřeba načíst novou hodnotu.

- Metody pro překlad:
  - **GenerateSourceAttributes** - Generování zdrojového kódu pro C# metaatributy. Například v případě čísla, které má minimální hodnotu -100 a maximální hodnotu 100, je vhodné vygenerovat následující metaatribut: `[Range(-100, 100)]`, což zajistí omezení hodnoty komponenty na úrovni inspektoru Unity. Pokud pro daný datový typ nedává smysl generovat žádné metaatributy, může tato metoda vracet prázdný řetězec.
  - **GenerateSourceDeclaration** - Generování zdrojového kódu pro samotnou deklaraci (a případně i definici) datového typu pro třídu nebo šablonu. Jako parametr se uvádí logická hodnota určující, zda se jedná o třídu (`true`) nebo šablonu (`false`) a dle toho by metoda měla případně použít klíčové slovo (`new`) pro atributy šablon, které atributy tříd v podstatě přepisují konkrétní hodnotou (to je blíže popsáno v kapitole 4.6). Pokud atribut není editovatelný na úrovni instancí, mělo by se navíc jednat o konstantu.

Samotná vnitřní implementace uchování dat je ponechána na autorovy daného datového typu. Jelikož každý atribut musí podporovat neomezený počet hodnot (nekonečně velké pole), vhodné je např. použití hashovací tabulky nebo slovníku, kdy klíčem je index pole. To umožňuje nastavit hodnotu na libovolnou pozici, aniž by bylo potřeba držet v paměti pole dané délky, kdy většina hodnot ani není nastavena (odpovídá výchozí hodnotě).

Na závěr je potřeba, aby nový datový typ obsahoval C# metaatributy obsahující informace o typu jako takovém, nikoliv o jeho instanci. Používá se metaatribut PropertyInfo, který obsahuje položky pro čitelné jméno atributu, strojové jméno a popisek určující, jak a k čemu tento datový typ použít. Příklad 4.1 znázorňuje metaatribut pro vestavěný celočíselný datový typ. Je také vhodné uvést, že je třída serializovatelná, i když to na funkcionalitu v rámci rozšíření nemá vliv, jelikož se serializace provádí ručně na různých úrovních.

```
1  [Serializable]
2  [PropertyInfo(
3      Name = "Number",
4      MachineName = "UPP_int",
5      Description = "An integer number without a floaing point. Both positive
6        and negative numbers are allowed, unless limited through Minimum
7       and/or Maximum."
```

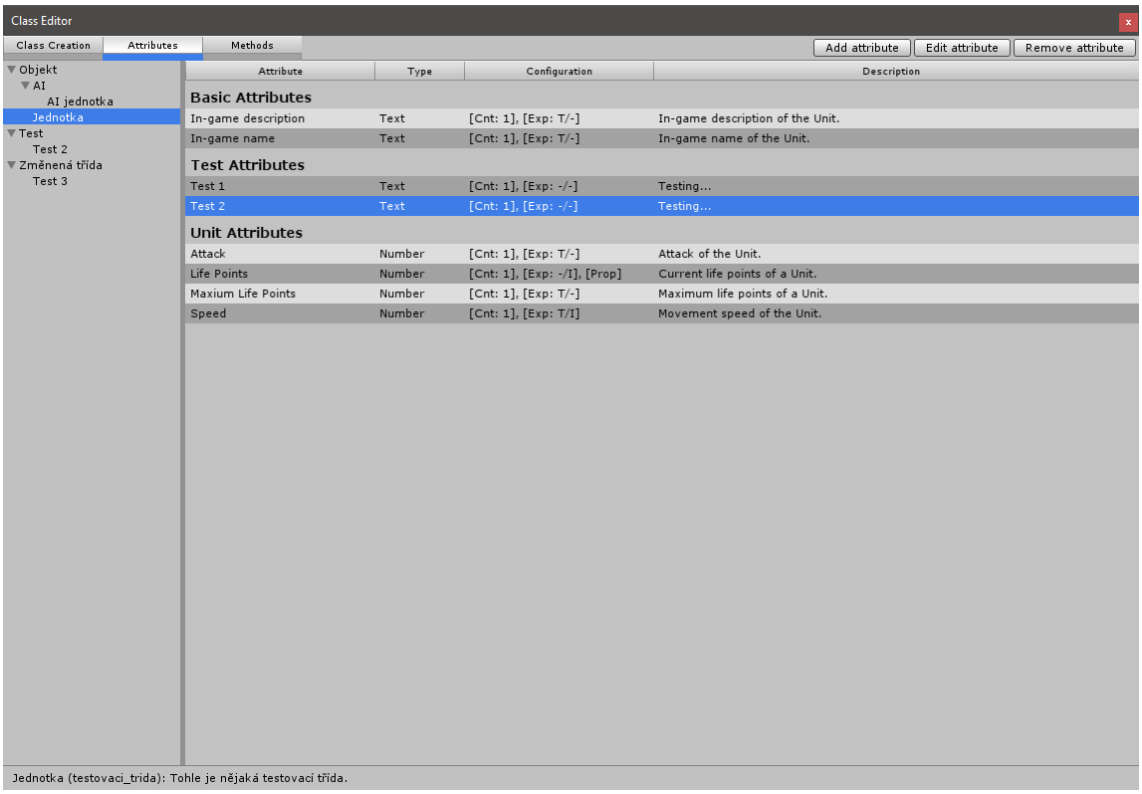
```
8         )]
```

Blok kódu 4.1: Ukázka metaatributu pro číselný datový typ

Vlastní datové typy neřeší obecná metadata, jako je jméno, popis či kategorie atributu. To je implementováno jako součást vyšších abstraktních tříd, jelikož na této úrovni není potřeba znalost typu atributu.

4.2.3 Editor atributů

Editor atributů je součástí editoru herních tříd. Pro přístup k editaci atributů je nutno přepnout záložku okna (tlačítka v levé horní oblasti) na „Attributes“, nebo vybrat herní třídu a v návrhovém režimu kliknout na tlačítko „Attributes“ v rámci entity diagramu tříd.



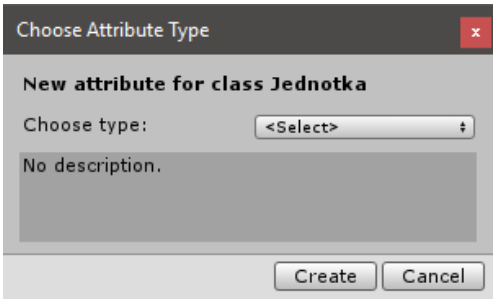
Obrázek 4.6: Ukázka editoru atributů [Snímek obrazovky]

Editor atributů v hlavní části obsahuje tabulku s atributy právě vybrané herní třídy. Tyto atributy jsou pro přehlednost seskupovány dle kategorií. Kategorii může návrhář určit při tvorbě či editaci libovolného atributu. Nejedná se o výčtový typ nebo referenci na objekt, kategorie je pouze textový identifikátor a je tedy možno mít libovolný počet kategorií. Při tvorbě nového atributu je navíc kategorie automaticky vyplněna na kategorii atributu právě vybraného (pokud takový atribut existuje).

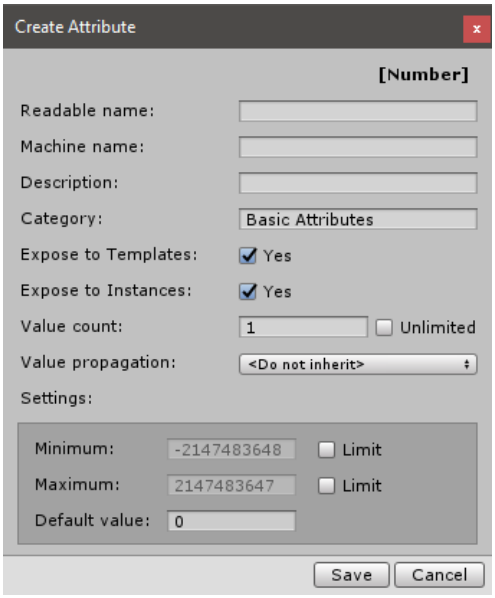
V tabulce se pak v jednotlivých buňkách nachází jméno atributu v čitelné podobě, jeho typ, konfigurace a popis atributu. Konfigurace pak sestává z následujícího:

- **[Cnt: x]**, kde x je povolený počet hodnot atributu, respektive délka pole. Nejčastěji 1 pro jednoduchý atribut, ale může být jiné kladné číslo nebo i `inf` pro neomezené množství hodnot.
- **[Exp: x/y]**, kde x je znak `-` nebo `T` a y je znak `-` nebo `I`. To určuje, zda je atribut viditelný pro editaci na úrovni šablon (první hodnota) a na úrovni instancí (druhá hodnota). Příklad: `[Exp: T/-]` značí, že atributu je možno nastavit hodnotu v šabloně, ale nikoliv pak přímo v instanci objektu.
- **[Prop]** je nepovinná značka určující, že hodnota tohoto atributu dědí (je propagována) z jiného atributu.

V pravé horní části editoru se pak nachází tlačítka pro vytvoření, editaci a smazání atributu.



(a) První krok



(b) Druhý krok

Obrázek 4.7: Ukázka vytvoření nového atributu [Snímek obrazovky]

Na obrázku 4.7 jsou znázorněna okna pro vytvoření nového atributu herní třídy. V prvním kroku se vybírá datový typ, zatímco ve druhém kroku již probíhá samotná konfigurace atributu. V případě, že návrhář upravuje již existující atribut, je první krok přeskočen. Typ atributu z technických důvodů není možno dodatečně měnit. Je však v případě nutnosti možno atribut smazat a vytvořit nový se stejným strojovým jménem.

Existující datové typy pro atributy se v kroku 1 načítají pomocí reflexe, která je blíže popsána v kapitole 3.5. Ve výsledku se tedy jedná o modulární přístup, kdy je možno dodatečně vytvářet nové typy atributů, jak je popsíno v kapitole 4.2.2.

V kroku návrhář konfiguruje atribut dle potřeby. V této fázi se vyplňují následující údaje:

- **Readable name** - Čitelné jméno atributu. Jedinečnost není vynucena, ale je doporučena.

- **Machine name** - Strojové jméno atributu. Stejně jako strojová jména herních tříd musí odpovídat specifikaci jazyka C# pro povolená jména a navíc musí být unikátní v rámci dané herní třídy.
- **Description** - Textový popis atributu.
- **Category** - Kategorie, pod kterou má být atribut zařazen v editoru atributů pro danou herní třídu (a v inspektoru Unity na úrovni prefab objektu a instancí).
- **Expose to Templates** - Určuje, zda má být atribut odhalen pro editaci na úrovni šablony herního objektu. To se hodí pro atributy, které nabívají stejných (často konstantních) hodnot pro všechny instance dané šablony.
- **Expose to Instances** - Určuje, zda má být atribut odhalen pro editaci na úrovni instancí objektů v herní scéně Unity prostřednictvím Unity inspektoru. To se hodí pro atributy, jejichž hodnota je odlišná pro každou instanci a často se mění v průběhu hry.
- **Value propagation** - Umožňuje nastavení hodnoty atributu na hodnotu jiného atributu stejného typu v okamžiku vytvoření nové instance objektu. Často se používá v kombinaci se skrytím atributu na úrovni šablon a odhalením na úrovni instancí.
- **Value count** - Určuje počet hodnot (respektive velikost pole), kterých může atribut nabývat. Je možno použít neomezeně velký datový prostor.
- **Settings** - Je v režii konkrétního datového typu.

Atribut je nejčastěji odhalen pouze do šablon nebo pouze do instancí. Občas se ale může hodit vytvořit interní, pracovní atribut, který není viditelný prostřednictvím grafického rozhraní, ale je možno jej číst nebo nastavovat ve zdrojovém kódu. Toho je docíleno sktytím atributu pro šablony i pro instance. Naopak v ojedinělých případech může být žádoucí specifickému atributu nastavovat hodnotu v šablonách, ale posléze i dodatečně v instancích. Proto rozšíření umožňuje libovolné nastavení viditelnosti atributu.

4.3 Akce a události

Akce a události herních tříd nebyly hlavním předmětem diplomové práce, nicméně bez jejich implementace by použitelnost rozšíření klesla. Proto byly naimplementovány nad rámec zadání.

Události a akce je možno vytvářet v záložce „Methods“ v okně editoru tříd. Oproti samotnému návrhu tříd či datových atributů se ovšem metody tříd razantně liší v jednom aspektu – není možno provést definici metody na zcela vizuální úrovni, pouze deklaraci. Samotné tělo vytvořené metody je však nutno implementovat na úrovni zdrojových kódů jazyka C#. V rámci editoru tříd je pouze specifikováno, jaký formát je od metody očekáván, respektive její jméno, parametry a návratová hodnota (a i to lze později změnit na úrovni zdrojového kódu).

Vytvořené metody nejsou ukládány do žádné databáze, jako je tomu v případě atributů. Jako praktičtější řešení se jevilo umožnit tvorbu přímo v rámci `.cs` souborů a metody do editoru tříd načítat pomocí reflexe. Pokud má být metoda viditelná v editoru tříd, musí obsahovat metaatribut *MethodInfo* z prostoru jmen *UPP.Engine* (nejedná se tedy o

System.Reflection.MethodInfo). Tato metoda musí být implementována uvnitř dané herní třídy, která je označena jako částečná (`partial`) a je tedy možnost implementovat libovolný počet metod napříč různými `.cs` soubory. Jedním z těchto souborů je pak i automaticky generovaný `.cs` soubor obahující navržené aributy pro herní třídu.

```
1   public abstract partial class testovaci_trida : class2 {
2       [UPP.Engine.MethodInfo(
3           Description = "Zrani jednotku, jejiz zivoty pak klesnou.")]
4       public int injure(int damage)
5       {
6           life -= damage;
7           return life;
8       }
9   }
```

Blok kódu 4.2: Ukázka implementace metody (akce) herní třídy

V bloku kódu 4.2 je znázorněno, jak může vypadat jednoduchá akce *injure* herní třídy *testovaci_trida*. Důležité je uvést metaatribut MethodInfo. Metody bez tohoto atributu jsou v rámci editoru herních tříd ignorovány, jelikož se předpokládá, že se jedná o interní či pracovní metody pro implementovanou třídu.

Události se vytváří podobně, jako akce. Pouze je potřeba navíc implementovat delegáta a náledně jej použít pro typ události (`event`).

4.4 Šablony tříd

Vzory či šablony jsou speciální typy herních tříd na nějnižší úrovni, které se nepoužívají pro další tvorbu a dědičnost potomků, ale místo toho slouží pro budoucí tvorbu instancí objektů přímo v herní scéně Unity. Během překladu je ke každému vzoru vytvořen speciální `.prefab` soubor, který slouží jako uložiště *GameObject* objektu spolu s jeho komponentami a vlastnostmi, ze kterého je pak možno vytvářet libovolný počet *GameObject* objektů přímo ve scéně.

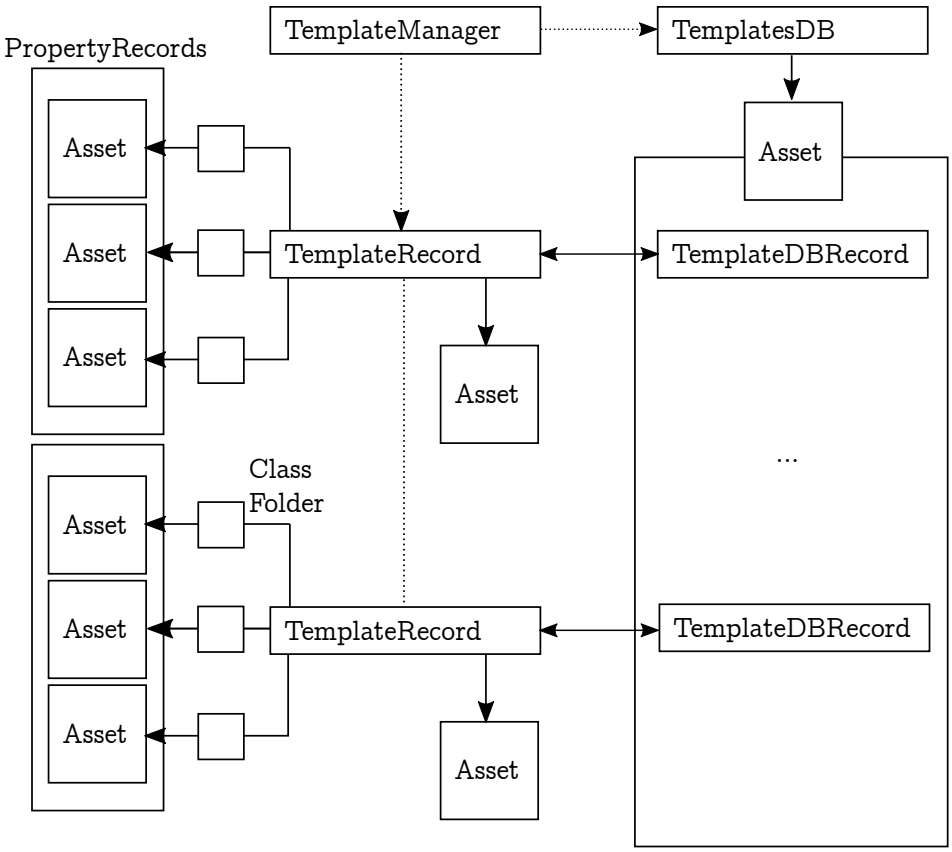
Změna hodnoty vlasnosti uvnitř `.prefab` souboru způsobí změnu ve všech instancích, nicméně je možno hodnoty přepisovat i přímo v samotných instancích – ty pak mají přednost. Inspektor instance navíc umožňuje provést operace revertování změn do podoby originálního prefab objektu či naopak uložení změněných hodnot do prefab objektu. Výjimkou je komponenta *Transform*, která je vždy individuální pro konkrétní instance.[2]

Pro strojové názvy šablon platí stejná pravidla, jako pro herní třídy, a navíc z technických důvodů není možno použít jako šablonu jméno, které již existuje jakožto jméno třídy.

4.4.1 Vnitřní reprezentace šablon

Reprezentace vzorů uvnitř rozšíření kombinuje reprezentaci herních tříd a atributů. V případě vzoru je totiž potřeba nejen uvést data o vzoru jako takovém, ale navíc i definice atributů, které třídy deklarují. Z jediné třídy ovšem může (a očekává se) dědit libovolný počet šablon, a proto není možné hodnoty uložit do stejné instance záznamu atributu, jaký

je provázán z herní třídou. Místo toho každá šablona musí mít vlastní sadu atributů, jejichž metadata vychází z atributů rodičovské třídy, ale samotné hodnoty se pak definují v šabloně.



Obrázek 4.8: Schéma vnitřní reprezentace šablon [Vlastní zdroj]

Na schématu 4.8 je znázorněna vnitřní reprezentace šablon v rámci rozšíření před kompilací. Stejně jako v případě herních tříd a atributů, i v případě šablon existuje `.asset` soubor s databází všech navržených šablon v jednoduché podobě objektů typu *TemplateDBRecord* (samotný *ScriptableObject* pro databázi je pak typu *TemplateDB*). Instance s podrobnými daty a metadaty o třídě jsou pak udržovány prostřednictvím *TemplateRecord* objektů, které jsou při každé změně serializovány do svých vlastních `.asset` souborů.

Dále je však ke každé šabloně vytvořen adresář nesoucí jako jméno strojový název šablony, který následně obsahuje serializované záznamy atributů šablony prostřednictvím *PropertyRecord* objektů uložených do vlastní sady `.asset` souborů. O atributech šablon blíže pojednává kapitola 4.4.2.

4.4.2 Atributy šablon

Atributy šablon vychází z atributů třídy, ze které daná šablona dědí. Deklarace atributů pouze přebírá a nelze ji v šablonách nijak měnit. Šablony pouze definují konkrétní hodnoty, které návrhář specifikuje prostřednictvím editoru šablon (kapitola 4.4.4).

Pro každou novou šablonu je tak potřeba vytvořit novou sadu atributů založenou na rodičovské třídě, a stejně tak v případě změn metadat atributů třídy by se tyto změny měly projevit ve všech šablonách. Zjednodušený příklad:

- Krok 1 - Vytvoří se herní třída `Class1`.
- Krok 2 - Vytvoří se celočíselný atribut pro třídu `Class1` se jménem `Num`. Atribut nemá nastavenou minimální ani maximální hodnotu.
- Krok 3 - Vytvoří se šablona `Temp1` a `Temp2` ze třídy `Class1`. Atributu `Num` je možno individuálně nastavit hodnotu v obou šablonách.
- Krok 4 - Pro šablonu `Temp1` se nastaví hodnota atributu `Num` na -400 a pro šablonu `Temp2` na 650.
- Krok 5 - Změní se deklarace atributu `Num` ve třídě `Class1` a nastaví se povolený rozsah od 0 do 100.
- Závěr - Automaticky by se minimum a maximum mělo projevit ve všech šablonách, který tento atribut používají, a hodnota by měla být upravena na co nejbližší hodnotu původní hodnotě, která je validní. Tedy v případě šablony `Temp1` na hodnotu 0 a v případě šablony `Temp2` na hodnotu 100. Zároveň by při příští editaci atributu v šabloně nemělo být umožněno zadat hodnotu mimo nový povolený rozsah.

K tomuto slouží speciální statická metoda *Blend* v abstraktní třídě `PropertyBase`. Ta přijímá jakožto dva argumenty objekty typu *PropertyRecord* se třetím logickým argumentem, který určuje, zda mají být pro výsledný objekt použity hodnoty atributu ve druhém argumentu či zda se mají použít hodnoty výchozí. Metoda pak z dvou předaných atributů vytváří nový objekt *PropertyBase* dle následujících pravidel:

1. Deserializuj oba objekty do podoby *PropertyBase*.
2. Zjisti typ atributu (řetězec jakožto identifikátor) a pokus se tento typ načíst pomocí *Type.GetType*.
3. Dle třetího argumentu udělej následující:
   (a) Pokud se mají použít výchozí hodnoty atributu, načti pomocí reflexe objekt *MethodInfo* metody se jménem „GetDefault“ v typu, který byl zjištěn v předchozím kroku. Tuto metodu vyvolej nad druhým deserializovaným argumentem. Reflexí pak vytvoř pole předem neznámé velikosti (nastavitelná proměnná určující, kolik hodnot se má vyplnit) a typu a to posléze naplň výchozí hodnotou, který byla zjištěna metodou *GetDefault*.
   (b) Pokud se nemají použít výchozí hodnoty, ale hodnoty existující, načti pomocí reflexe objekt *MethodInfo* metody se jménem „Pack“ v typu, který byl zjištěn v kroku 2. Dále tuto metodu vyvolej nad druhým deserializovaným argumentem. Výsledek ulož do proměnné typu *Array* (bez specifikovaného typu v deklaraci).
4. Načti objekt *MethodInfo* metody se jménem „Unpack“ z typu deserializovaného prvního argumentu.
5. Vyvolej zjištěnou metodu a jako parametr předej výsledné pole zjištěné v kroku 3.
6. Vrať první deserializovaný argument (je nezávislý na své serializované podobě).

Jak je patrné z výše uvedeného, pro sloučení dvou atributů a zajištění konzistence je potřeba spousta reflexe. Reflexe jazyka C# je více popsána v kapitole 3.5.

4.4.3 Manažer šablon

Manažer šablon je statická třída vytvořená na kombinovaném principu manažeru tříd (kapitola 4.1.2) a manažeru atributů (kapitola 4.2.1). Tato třída se stará o tvorbu a správu potřebných instancí záznamových tříd a `.asset` souborů šablon v okamžiku, kdy návrhář vytvoří, edituje nebo smaže šablonu herního objektu. Současně se stará o validaci a udržování konzistence atributů v případě, že dojde ke změně jejich metadat na úrovni herních tříd.

Jak již bylo znázorněno na schématu 4.8, manažer šablon obsahuje databázi typu *TemplateDB*, ve které jsou udržovány zjednodušené záznamy vytvořených šablon v podobě *TemplateDBRecord* objektů. Databáze je při každé změně opět serializována do vlastního `.asset` souboru.

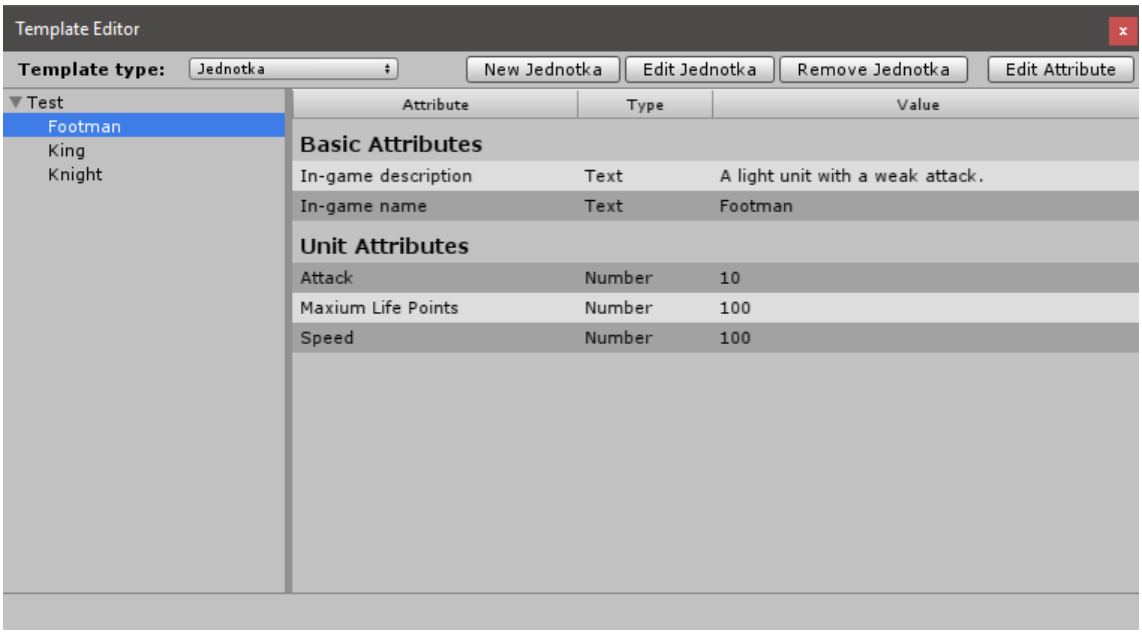
Manažer šablon implementuje následující vřejné metody:

- **GetTemplateByMachineName** - Načte instanci šablony *TemplateRecord* na základě strojového jména. Strojové jméno šablony je vždy jedinečné a nesmí se jednat o strojové jméno herní třídy.
- **TemplateExists** - Otestuje, zda šablona se specifickým jménem již existuje či nikoliv. Z bezpečnostních důvodů testuje existence `.asset` souboru, nikoliv pouze záznam v databázi šablon.
- **TemplateOrClassExists** - Složení metody *TemplateExist* ze třídy *TemplateManager* s metodou *ClassExists* ze třídy *ClassManager*. Vrací `true` pokud třída nebo šablony s daným jménem již existuje, jinak vrací `false`.
- **FindTemplatesOfType** - Nalezne všechny šablony, které dědí ze specifikované herní třídy. Vrací list párů, kdy klíčem je vždy *TemplateDBRecord* a hodnotou pak *TemplateRecord*.
- **FindAllAtributesFor** - Nalezne všechny atributy pro specifikovanou šablonu. Vrací list objektů *PropertyRecord*.
- **PrepareAttributes** - Metoda pro zajištění konzistence atributů mezi herními třídami a mezi šablonami. Pro specifikovaný záznam šablony provede validaci, která sestává ze třech kroků:
  - Smazání zastaralých atributů z šablony, které se již nenachází v rodičovské herní třídě.
  - Přidání nových atributů, které se nachází v rodičovské herní třídě, ale nejsou prozatím v šabloně.
  - Kontrola metadat existujících atributů šablony a případně oprava nastavených hodnot. Používá se statická metoda *Blend* abstraktní třídy *PropertyBase*, která je podrobně popsána v kapitole sec:temp-att.
- **Create** - Vytvoří novou instanci záznamu šablony a k tomu patřičný `.asset` soubor. Dále vytvoří záznam v databázi šablon. Při použití nepovinného parametru *checkForExistence* navíc kontroluje, jestli už herní třída nebo šablona s tímto jménem neexistuje a případně zobrazí chybový dialog (nevyvolá výjimku). Tato metoda nevytváří atributy pro nově vytvořený záznam šablony; očekává se, že bude použita metoda *PrepareAttribute* po úspěšném vytvoření šablony.

- **Edit** - Upraví údaje existující šablony. Umožňuje dodatečnou změnu strojového jména – v takovém případě je provedena rutina pro ověření, zda šablona či třída s novým jménem již v projektu neexistuje a pokud ne, manažer upraví patřičná jména souborů a adresářů a všechny záznamy o šabloně tak, aby data byla konzistentní.
- **Remove** - Smaže šablonu a s ním patřičný záznam z databáze i `.asset` soubor. Dále smaže všechny atributy, které se v šabloně nacházely (nikoliv atributy herních tříd, ze kterých šablona vycházela). Je možno použít nepovinný parametr *confirmationDialog*, který zařídí zobrazení dialogu pro potvrzení smazání šablony.

4.4.4 Editor šablon

Editor šablon se stejně jako editor tříd nachází pod nabídkou „Unity++“ v editoru Unity, konkrétně pak pod položkou „Template Editor“. Okno editoru šablon je znázorněno na obrázku 4.9.



Obrázek 4.9: Ukázka editor šablon [Snímek obrazovky]

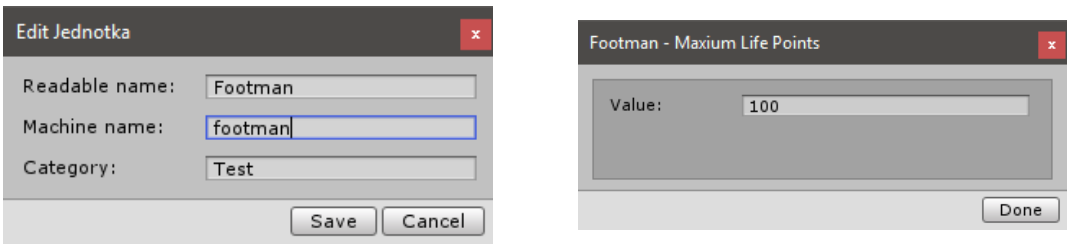
Editor šablon sestává z několika částí. V horním panelu se nachází element pro výběr herní třídy, ze které budou šablony, se kterými návrhář hodlá v dané chvíli pracovat, vycházet. Jsou zde pouze herní třídy, u kterých bylo nastaveno, že nejsou abstraktní. Dále se zde nachází tlačítka pro vytvoření, editaci a smazání šablony. Čtvrtým tlačítkem je pak editace vybraného atributu. V této fázi není již možno atributy vytvářet či mazat, ani jakýmkoliv způsobem měnit jejich metadata. Od toho slouží herní třídy; šablony pouze definují konkrétní hodnoty atributů, který již byly deklarovány.

Hlavní část okna je dále rozdělena do dvou menších panelů, jejichž velikost je možno dynamicky měnit.

V levém panelu se nachází hierarchická struktura s vytvářenými šablonami. Tato struktura má pouze dvě úrovně – kategorie na vyšší úrovni a samotné šablony na úrovni nižší. Kategorie nemají žádný hlubší význam pro překlad, jedná se pouze o řetězec, ke kterému je šablonu během editace možno přiřadit. Jiným způsobem šablony není možno zanořovat

a z hlediska případů užití by to ani nemělo smysl – šablony mají být posledním stupněm v rámci tvorby herní logiky. Zanořování a dědění v podobě, jako se děje v případě herních tříd, tedy u šablon nejde provádět.

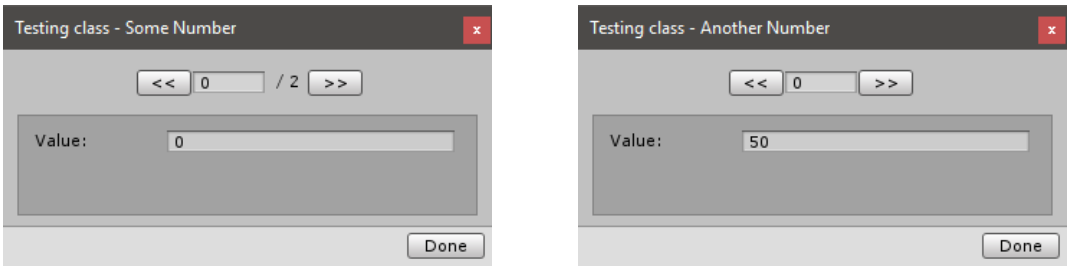
V pravém panelu pak je možno nalézt hlavní oblast pro definici hodnot atributů šablony. Tyto atributy jsou automaticky vytvořeny během prvního načtení nové šablony a o konzistenci dat se stará manažer šablon (jak je popsáno v kapitole 4.4.3).



(a) Editace šablony

(b) Editace jednoduchého atributu šablony

Obrázek 4.10: Ukázka použití editoru šablon [Snímek obrazovky]



(a) Omezený počet hodnot

(b) Neomezený počet hodnot

Obrázek 4.11: Úprava atributů s více hodnotami [Snímek obrazovky]

Na obrázku 4.10 jsou vyobrazena dvě okna. V prvním okně je znázorněna editace existující šablony „Footman“ a ve druhém pak editace celočíselného atributu „Maximum Life Points“ této šablony. Je zde důležité zmínit, že v tomto případě se jedná o nastavení hodnoty v případě, že počet hodnot atributu je v rámci herní třídy nastaven na 1. Pokud by počet hodnot byl vyšší nebo dokonce neomezený, byl by v rámci okna vykreslen i ovládací prvek pro přechod mezi hodnotami dle indexu. Tato situace je znázorněna na obrázku 4.11.

Jakým způsobem je implementováno ukládání hodnot je vždy v režii konkrétního datového typu. U předpřipravených typů rozšíření se používá slovník (*Dictionary*), kdy klíčem záznamu je index a hodnotou pak samotná hodnota atributu na daném indexu. To umožňuje používat vysoké indexy bez nutnosti tvorby pole velké délky, kde většina hodnot by byla výchozí.

4.5 Ukázka použití manažerů v kódu

Předpokládá se, že pro tvorbu tříd a šablon bude návrhář používat připravené editory. Pokud je však z nějakého důvodu proces tvorby potřeba automatizovat, je možno použít statické třídy *ClassManager*, *PropertyManager* či *TemplateManager* přímo ve vlastním kódu. Zjednodušená ukázka tvorby tříd a šablon vlastním kódem je pak k vidění v příkladu 4.3.

```
// Nejprve vytvorime 10 hernich trid:
for (int i = 0; i < 10; i++)
{
   string s = i.ToString();
   string className = "mcClass" + s;
   ClassManager.Create(
      "MC Class " + s,              // Jmeno tridy
      className,                    // Strojove jmeno
      "",                           // Rodic
      "Manual creation test #" + s, // Popis
      false                         // Je abstraktni
      );

   // Ka kazde teto tride dale 10 atributu:
   for (int j = 0; j < 10; j++)
   {
      string t = j.ToString();

      // Nejprve je potreba vytvorit serializovatelny objekt:
      PropertyInt prop = new PropertyInt();
      prop.ValueCount = 1;
      prop.Name = "MC Property " + t;
      prop.MachineName = "mcProp" + t;

      // Nyni vytvoreni skrz manazera:
      PropertyManager.Create(
         prop,                    // Objekt s atributem
         className,               // Herni trida
         prop.GetType().FullName // Typ atributu
         );
   }

   // Nyni vytvoreni 3 sablon ke kazde tride:
   for (int j = 0; j < 3; j++)
   {
      string t = j.ToString();
      string tempName = "mcTemp" + s + "_" + t;
      TemplateManager.Create(
         "MC Template " + s + " / " + t, // Jmeno sablony
         tempName,                       // Strojove jmeno
         className                       // Rodicovska trida
         );
   }
}
```

Blok kódu 4.3: Ukázka automatizované tvorby tříd a šablon

Blok zdrojového kódu 4.3 názorně předvádí, jak je možno vytváření tříd a šablon automatizovat. Jedná se o zjednodušený příklad, který neřeší hierarchické zanořování herních tříd (všechny jsou na nejvyšší úrovni) a některá vedlejší metadata jsou ponechána na výchozí hodnotě, navíc se pak nenastavují hodnoty atributů šablon. Jde pouze o názornou ukázku použití rozšíření na úrovni kódu.

První cyklus obstarává vytváření samotných herních tříd. V ukázce je vytvořeno 10 tříd se strojovým jménem „mcClass0“ - „mcClass9“. Pro každou tuto třídu je pak dále vytvořeno 10 vlastních atributů (tedy 100 atributů celkem) a na závěr jsou ke každé třídě vytvořeny 3 šablony, které z dané třídy dědí.

4.6 Implementace překladače

V rámci diplomové práce bylo potřeba rozhodnout, jestli navržené třídy a šablony překládat do podoby, jenž je srozumitelná pro editor Unity, nebo jestli je vhodnější vytvořit interpret jakožto mezivrstvu mezi rozšířením a samotným editorem Unity. Proto bylo potřeba zhodnotit výhody a nevýhody těchto dvou řešení.

Mezi výhody interpretovaných jazyků patří následující:[4]

- Nezávislost na platformě. To však není příliš relevantní v případě Unity, které je již ve svém základu multiplatformní.[5]
- Dynamické typování, dynamické určování prostoru vázaných proměnných. Není relevantní pro potřeby diplomové práce, kdy návrhář nepíše ve vlastním jazyce, ale pouze navrhuje určité prvky tvořící herní logiku, které by pak potenciálně byly interpretovány.
- Reflexe interpretovaného jazyka. Jazyk C# je však taktéž reflexivní, i když se překládá.

Naopak velkým problémem interpretů je nižší rychlost vykonávání operací. Proto bylo ve výsledku rozhodnuto, že všechny herní třídy a šablony, které návrhář vytváří, budou přeloženy do podoby nativního C# kódu a bude ponecháno na Unity, aby tyto prvky zakomponovala do výsledných scén.

K tomu byla vytvořena statická třída *Compiler*, která se o převod stará. Obsahuje spoustu privátních procedur a funkcí, nicméně veřejnou metodou je pouze metoda *CompileAll*. V rámci rozšíření je pak možno spustit překlad v okně *Compiler*, jenž je možno otevřít v hlavním menu Unity pod volbou „Unity++“. Toto okno je znázorněno na obrázku 4.12.

4.6.1 Převod tříd

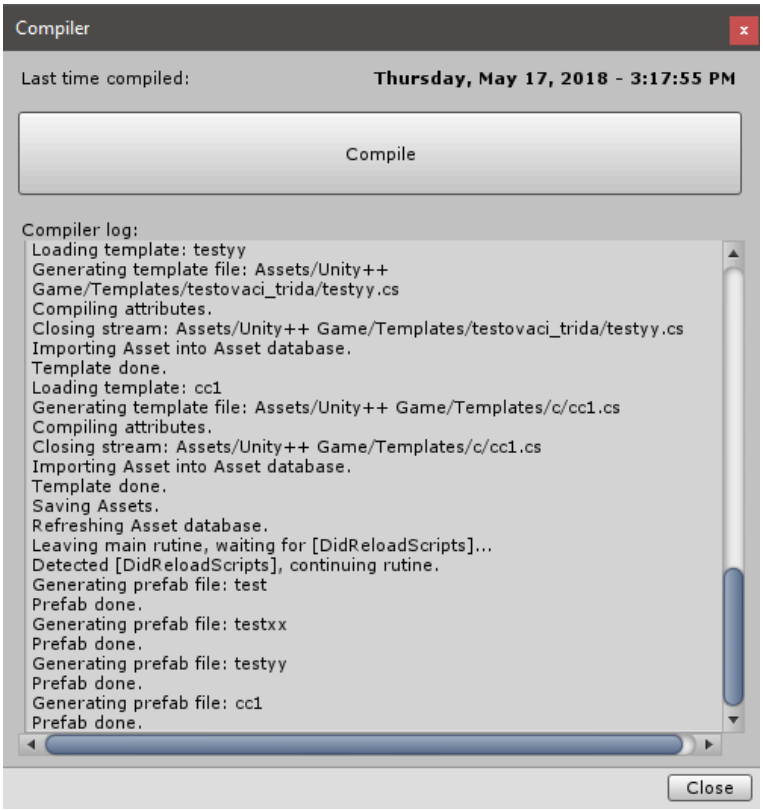
Prvním krokem při překladu přeložení herních tříd, včetně jejich atributů. To se provede průchodem všech záznamů databáze herních tříd – tím jsou získány strojová jména tříd – a následným postupným načítáním plných verzí záznamů herních tříd v cyklu.

Dle definovaných pravidel je pro každý takovýto záznam vytvořen nový `.cs` soubor na určeném místě. Soubor nese jako jméno strojový název herní třídy. Dále je v souboru generováno následující:

- Hlavička s komentářem nesoucí základní informace o třídě.
- Výchozí povinné knihovny, které jsou potřeba ke správne funkcionalitě.

[4] Podrobnější výčet vlastností překladačů a interpretů je dostupný v knize [3].

[5] Unity implementuje vlastní verzi vývojové platformy *.NET Framework* s názvem *Mono*, která je multiplatformní.



Obrázek 4.12: Okno pro překlad [Snímek obrazovky]

- Extra knihovny, které je možno definovat v nastavení rozšíření.
- Prostor jmen (standardně „UPP.Game“).
- XML dokumentace třídy prostřednictvím komentáře s trojitým lomítkem.
- Deklarace třídy s patřičnou dědičností rodičovské třídy. Herní třídy jsou vždy po překladu abstraktní. Očekává se, že instance budou vytvářeny z šablon.
- Samotné tělo třídy.

V rámci těla třídy jsou pak načteny všechny atributy, které byly specifikovány v rámci herní třídy. Ty jsou seskupeny dle kategorií a seřazeny dle čitelného jména. Dále je pro každý tento atribut vygenerováno následující:

- Pokud je detekována nová kategorie, je vygenerován speciální metaatribut `[Space]` pro inspektor Unity. Dále se použije `[Header]` pro samotný nadpis kategorie uvnitř inspektoru.
- XML dokumentace atributu prostřednictvím komentáře s trojitým lomítkem.
- Metaatributy, které generují datové typy pomocí metody *GenerateSourceAttributes*. Můžou ovlivnit, jak se pole zobrazuje v inspektoru.
- Samotná definice atributu, kterou generuje datový typ metodou *GenerateSourceDeclaration*. Na úrovni herních tříd (nikoliv šablon) se pak jako hodnota uvádí výchozí hodnota atributu.

Na závěr se provede uzavření otevřených závorek a soubor se uloží a zavře. Vytvořen v kontextu Unity je pak pomocí *AssetDatabase.ImportAsset*, jelikož standardní vstupně-výstupní operace mohou být v Unity nespolehlivé a/nebo způsobovat chyby.

Jakmile jsou všechny herní třídy přeloženy, provede se uložení a obnovení *AssetDatabase* v Unity.

Je vhodné také uvést, že pokud herní třída nemá ve fázi návrhu rodiče (tedy je sama na nejvyšší úrovni), při překladu je jako rodič uvedena třída `MonoBehaviour`, která umožňuje tvorbu komponenty pro objekt ve scéně Unity a obsahuje některé základní metody pro správu objektu během hry, jako jsou *OnStart* či *OnUpdate*.

4.6.2 Převod vzorů

Druhým krokem překladu je přeložní šablon, které dědí z herních tříd a definují konkrétní hodnoty atributů. Princip je téměř identický, jako překlad herních tříd, pouze se zde pracuje s databází šablon a manažerem šablon. Na úrovni výsledného `.cs` souboru se pak šablony odlišují ve dvou aspektech:

- Výsledné třídy nejsou abstraktní. Je tedy možno vytvářen z nich instance objektů.
- Atributy jsou znovu deklarovány s klíčovým slovem `new`, tedy přepisují atributy uvedené v rodičovské třídě. Samy pak jako hodnotu generují to, co návrhář uvedl v rámci patřičné šablony, nikoliv hodnotu výchozí, jako tomu bylo v případě herních tříd.

Každá šablona má v době návrhu jakožto rodiče herní třídu, nicméně po překladu je rozdíl mezi herní třídou a šablonou minimální. Ve výsledku se jedná o celistvou hierarchii tříd, z nichž některé jsou abstraktní, a jejichž nejvyšším rodičem je třída `MonoBehaviour`, jak již bylo zmíněno v kapitole 4.6.1.

4.6.3 Tvorba prefab objektů

Jakmile jsou herní třídy a šablony přeloženy, zbývá vytvořit `.prefab` soubory pro každou šablonu. Prefab objekty pak umožňují snadné vytváření nových instancí objektů ve scéně Unity prostřednictvím kopírování prefab objektu. Tyto instance pak mají stejné komponenty a vlastnosti, jako prefab, ze kterého byly vytvořeny. Dojde-li ke změně hodnoty atributu prefab objektu, tato změna se projeví ve všech instancích daného typu, pokud tato instance atribut již nepřepsala na vlastní hodnotu. Navíc je možno každou instanci revertovat do podoby jejího prefab objektu či nastavit hodnoty atributů prefab objektu do podoby vybrané instance.

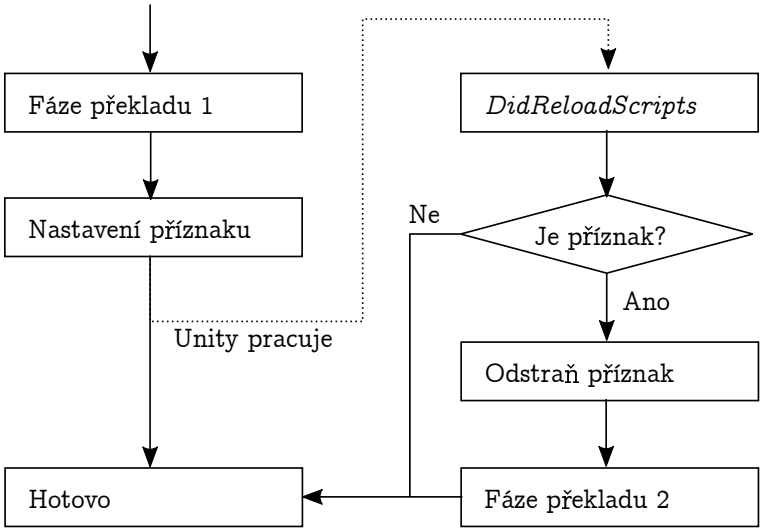
Tvorba `.prefab` souboru pro šablonu je provedena vytvořením prázdného prefab souboru na specifikovaném místě. Dále je dynamicky vytvořen *GameObject*, kterému je dynamicky přiřazena Unity komponenta pomocí specifikace jejího typu. Typem je přeložená třída popisující šablonu, jenž byla vytvořena v předchozím kroku. Na závěr *GameObject* nahradí vytvořený prázdný `.prefab` soubor.

4.6.4 Dvoufázový překlad

Při překladu herních tříd a šablon a následné tvorbě prefabů nastává problém. Tvorba prefab objektu, jak již bylo řečeno v kapitole 4.6.3 požaduje znalost typu pro vytvoření komponenty, která má být objektu přiřazena. Tento typ je získáván pomocí reflexe, nicméně

během překladu typ ještě neexistuje – právě byl vytvořen a editor Unity o jeho existenci až do dokončení překladu neví.

Existují dvě možnosti, jak tohle řešit. První možností je vynucení synchronního importu souboru a jeho okamžitý interní překlad na úrovni editoru. Toho je možno docílit následovně: *AssetDatabase.ImportAsset(fpath, ImportAssetOptions.ForceSynchronousImport | ImportAssetOptions.ForceUpdate);* Tato metoda však není doporučována a může způsobit problémy při překladu. Vhodnější je použití dvoufázového překladu, kdy jsou nejprve vytvořeny potřebné `.cs` soubory herních tříd a šablon a dále se asynchronně počká, až Unity tyto soubory zakomponuje do své interní databáze asset souborů a přeloží je. Po překladu Unity automaticky zavolá všechny metody s metaatributem `[DidReloadScripts]`, čehož lze využít pro spuštění druhé fáze překladu.



Obrázek 4.13: Schéma dvoufázového překladu [Vlastní zdroj]

Na schématu 4.13 je znázorněno provedení dvoufázového překladu. V první fázi se provádí překlad herních tříd a šablon a následně se nastaví příznak a asynchronně se čeká, než Unity zpracuje potřebné změny. Jakmile Unity znovu načte všechny `.asset` soubory a skripty (což může být způsobeno i jinou příčinou než jen první fází překladu), je automaticky vyvolána metoda s metaatributem `[DidReloadScripts]`. V té se zkontroluje existence příznaku z první fáze překladu. Pokud existuje, vymaže se a pokračuje se druhou fází. V opačném případě se nic neprovádí.

Ve druhé fázi je pak již možno bez problému vytvořit prefab objekty pro všechny existující šablony.

4.7 Použití ve scéně Unity

Jakmile je dokončen překlad, který je popsán v kapitole 4.6, je možno výsledné třídy a objekty použít přímo ve scéně Unity. Existují dva způsoby, jak toho docílit.

Prvním způsobem je použití automaticky vytvořených prefab objektů. Ty je možno jednoduše přetáhnout z projektového adresáře do scény či do hierarchie, a tím je automaticky vytvořena nová instance. Prefab objekty jsou připraveny pro všechny navržené šablony herních tříd. Je možno dodatečně měnit hodnoty atributů pomocí inspektoru Unity, a to jak na úrovni prefab objektu, tak i v samotné instanci. Pokud však návrhář upraví hodnotu

atributu v prefab objektu, dojde k revertování této změny při příštím překladu. Pro trvalou změnu hodnoty je potřeba upravit navrženou šablonu pomocí editoru šablon. Pokud je ovšem hodnota atributu upravena přímo na úrovni individuální instance, pak je tato změna zachována i při opakovaném překladu a přepisu prefab objektu.

Obrázek 4.14: Ukázka nové palety nástrojů pro scénu [Snímek obrazovky]

Druhým způsobem je použití palety nástrojů, kterou je možno otevřít v nabídce *Unity++/Scene Tools*. Toto plovoucí okno je možno připnout do libovolné části Unity editoru, nebo je možno jej ponechat v plovoucí podobě. Okno pak obsahuje možnost výběru existujících typů šablon (respektive herních tříd, které nebyly ve fázi návrhu označeny jako abstraktní) a dále pak tlačítka reprezentující existující šablony daného typu. Kliknutím na patřičné tlačítko pak dojde k vytvoření instance uprostřed scény stejným způsobem, jako by byl použit prefab objekt.

4.8 Editor spínačů

Editor spínačů je speciální editor umožňující na úrovni scény provazovat události a akce objektů. Pomocí metod, jejichž tvorba je popsána v kapitole 4.3 pak dochází k tvorbě seskupených logických celků, kdy je možno na omezené úrovni scénu vizuálně programovat. Omezenost je pak v režii programátora, který musí nejprve akce a události naimplementovat, než je možno je v editoru spínačů použít.

Použitelné akce a události jsou získávány pomocí reflexe. Každá použitelná metoda musí definovat metaatribut *MethodInfo* a musí být implementována uvnitř herní třídy (jedná se o částečnou[6] třídu). Není však žádoucí metody implementovat uvnitř stejného `.cs` souboru, který vznikl při překladu třídy. Při příštím překladu by totiž změny souboru byly ztraceny. Proto je potřeba použít vlastní soubor (umístění není pro reflexi důležité).

Jakmile jsou potřebné události a akce naimplementovány, je možno je použít v editoru spínačů. Nejprve je potřeba vytvořit nový spínač (tlačítko v pravé horní části). Každý spínač pak umožňuje definovat libovolný počet událostí, které spínač vyvolají, pokud alespoň jedna

[6] Částečné, nebo-li `partial` třídy umožňují implementaci třídy ve vícero souborech.

nastane, a libovolný počet akcí, které se při spuštění spínače provedou v chronologickém pořadí.

Každý spínač může být provázán pouze na jednu konkrétní scénu. Scény se vybírají pomocí tlačítka v levé horní části editoru. Jakmile scéna obsahuje alespoň jeden spínač, je v patřičné scéně vytvořen speciální objekt se jménem „Unity++ Scene Manager“. Tento objekt nesmí být smazán, jinak spínače pro danou scénu nebudou fungovat a potenciálně může dojít k jejich úplné ztrátě.

Kapitola 5

Testování

Tato kapitola pojednává o testování rozšíření a výsledcích, které byly z testů vyvozeny.

Rozšíření bylo testováno třemi způsoby. Prvním způsobem je zátěžový test, kdy byl automaticky vytvořen větší počet herních tříd, atributů a šablon a testovala se rychlost vytvoření a odezvy celého rozšíření. Druhým testem pak bylo testování na úrovni grafického uživatelského rozhraní a třetím testem pak byl pokus o vytvoření jednoduché hry za použití rozšíření.

5.1 Tvorba dat prostřednictvím manažerských objektů

Standardně se očekává, že návrhář bude pro tvorbu herních tříd a šablon používat připravené grafické rozhraní. Jak však bylo ukázáno v kapitole 4.5 a příkladu 4.3, tvorbu těchto entit je možno automatizovat za pomocí ručního použití statických tříd *ClassManager* a *TemplateManager* (případně též *PropertyManager*).

Prvním testem tedy byla tvorba herních tříd, atributů a šablon ručně z kódu ve velkém množství a měření doby splnění operací. Naměřené hodnoty jsou k dispozici v tabulce 5.1.

Počet herních tříd	Počet atributů na třídu	Počet šablon na třídu	Celkem `.asset` souborů	Doba
1	1	1	3	méně než sekunda
1	10	10	21	6 sekund
10	10	3	140	49 sekund
100	0	0	100	35 sekund
1	100	0	101	34 sekund
1	0	100	101	35 sekund

Tabulka 5.1: Naměřené doby trvání zpracování nových dat

Jak je z tabulky 5.1 patrné, tvorba velkého množství entit pomocí výše specifikovaných manažerů není příliš efektivní. Doba trvání operace je téměř zcela závislá na počtu vytvořených `.asset` souborů. Důvodem neefektivity je pak fakt, že manažeři tříd, atributů a šablon automaticky provádí uložení a znovunačtení celé Unity Asset databáze, s čímž pak může souviset i serializace a deserializace některých objektů. Rozšíření totiž podporuje tvorbu

velkého množství tříd popisující herní logiku včerně nadměrného počtu atributů, nicméně neočekává hromadnou tvorbu těchto věcí.

Za normální situace je uložení a znovunačtení Asset databáze po vytvoření některé herní entity žádoucí pro aplikaci provedených změn, nicméně z testu vyplývá, že by do budoucna bylo vhodné vytvořit i efektivnější způsob hromadného zpracování dat, kdy se všechny změny projeví až na závěr všech prováděných operací.

5.2 Tvorba dat skrze uživatelské rozhraní

Druhým způsobem testování rozšíření byla práce na úrovni grafického rozhraní a provádění různých permutací úkolů, na závěr čehož se kontrolovala validita výsledků (vytvořených dat). Úkoly, které byly v různých podobách prováděny:

- Tvorba nové herní třídy.
- Změna strojového jména herní třídy.
- Změna jiných dat herní třídy.
- Nastavení rodiče herní třídy.
- Smazání rodiče herní třídy.
- Smazání třídy, která má potomky.
- Smazání třídy, která obsahuje atributy.
- Smazání třídy, ze které dědí šablony.
- Deklarace nového atributu.
- Změna strojového jména atributu.
- Změna jiných dat atributu.
- Nastavení propagace hodnoty atributu.
- Zrušení propagace hodnoty atributu.
- Smazání atributu, ze kterého jiné atributy propagují.
- Smazání atrubutu, ze kterého nic nepropaguje.
- Tvorba nové šablony.
- Změna strojového jména šablony.
- Změna jiných dat šablony.
- Definice hodnoty v atributu o velikosti 1.
- Definice hodnot v atributu s větší velikostí.
- Definice hodnot v atributu neomezené velikosti.
- A některé další...

Během testování nebyl nalezen žádný závažný problém a výsledná data vždy odpovídala očekávaným výsledkům.

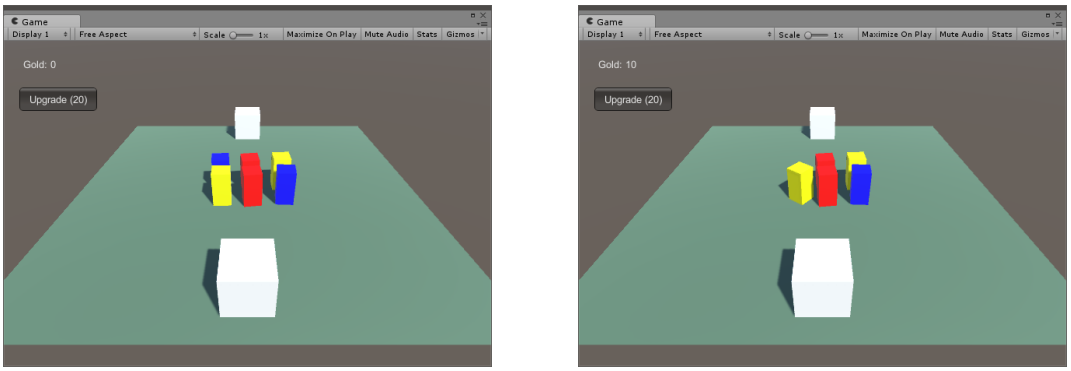
5.3 Tvorba jednoduché hry

Pro otestování rozšíření v praxi byla vytvořena jednoduchá pseudo-strategická hra obsahující několik typů jednotek dvou nepřátelských týmů, které se přetlačují na bojišti. Každá jednotka má určitý počet životů, útok a rychlost. Jednotky jsou pak následující:

- **Červená** - 100 životů, útok 10, rychlost 15.
- **Modrá** - 150 životů, útok 15, rychlost 20.
- **Žlutá** - 200 životů, útok 10, rychlost 20.

Každý tým má svou hlavní budovu, u které se periodicky jednotky objevují. Budovu je možno vylepšit, což vede k rychlejší produkci jednotek. Vylepšení však stojí zlato, které se získává eliminací nepřátelských jednotek.

Vítězem je tým, který zničí nepřátelskou budovu.



(a) První ukázka (b) Druhá ukázka

Obrázek 5.1: Ukázka jednoduché hry [Snímek obrazovky]

Během implementace této jednoduché hry nebyly zjištěny žádné problémy při použití rozšíření. Návrh herních tříd i šablon proběhl bez problémů a výsledek byl úspěšně přeložen. Některé herní aspekty, jako samotná grafika, nebyly vytvořeny za použití rozšíření (k tomu ani neslouží), ale za pomocí standardních Unity nástrojů.

Kapitola 6

Závěr

Cílem této diplomové práce bylo vytvoření rozšíření pro vývojové prostředí Unity, které by usnadnilo proces návrhu a programování herní logiky na úrovních, jenž je možno automatizovat. Rozšíření úspěšně vytváří abstraktní vrstvu mezi dvěma rolemi vývojářů – návrhářem, který je zodpovědný za herní logiku a za samotná data a programátorem, který implementuje a píše zdrojové kódy. Rozšíření umožňuje soustředit se na samotný proces vývoje bez řešení integrace návrhů do herního jádra.

Při implementaci byl kladen důraz na rozšiřitelnost a modularitu. Je možno deklarovat vlastní datové typy atributů, bezkonfliktně rozšiřovat přeložené třídy či ručně implementovat metody uvnitř herních tříd. Rozšíření není tvořeno pouze se zřetelem na jeden typ her, ale je možno jej použít pro řešení jakéhokoliv projektu, a to i větších rozměrů.

V rámci práce byl vytvořen editor herních tříd umožňující snadný návrh herní logiky na úrovni třídních diagramů, včetně deklarace atributů prostřednictvím uživatelského rozhraní. Dále byl naimplementován editor šablon pro specifikace instanciovatelných tříd a definice hodnot atributů. Dvoufázový překladač pak všechna navržená data převádí do podoby `.cs` souborů, které jsou pro editor Unity srozumitelné. Dále usnadňuje práci automatickým vytvářením `.prefab` souborů pro snazší tvorbu objektů ve scéně.

Nad rámec zadání bylo dále implementováno vlastní řešení pro tvorbu grafického uživatelského rozhraní založené na objektově orientovaném přístupu místo přístupu deklarativního, které Unity standardně na úrovni editoru používá.

Testování proběhlo ve třech podobách a nebyly zjištěny žádné problémy. Za pomoci rozšíření byla i naimplementována jednoduchá hra demonstrující použitelnost projektu na praktickém příkladu. Potenciální budoucí vývoj by dále měl být směrován do rozšiřování editoru spínačů a usnadnění tvorby událostí a akcí pro herní třídy.

Všechny formální požadavky na diplomovou práci byly úspěšně splněny.

Literatura

[1] Adams, E.; Rollings, A.: *Fundamentals of game design*. New Riders, 2010, ISBN 9780321643377.

[2] Blackman, S.: *Beginning 3D game development with Unity 4: all-in-one, multi-platform game development*. Apress, 2013, ISBN 978-1430248996.

[3] Brown, P. J.: *Writing Interactive Compilers and Interpreters*. Wiley, 1981, ISBN 978-0471100720.

[4] Chambers, H.: *My way or the highway: the micromanagement survival guide*. Berrett-Koehler Publishers, 2004, ISBN 9781576752968.

[5] Erich, G.; Helm, R.; Johnson, R.; aj.: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995, ISBN 978-0201633610.

[6] Forman, I. R.; Forman, N.: *Java reflection in action*. Pearson Education, 2005, ISBN 1932394184.

[7] Friedl, J. E. F.: *Mastering Regular Expressions*. O'Reilly Media, 2006, ISBN 978-0596528126.

[8] Gregory, J.: *Game Engine Architecture*. A K Peters/CRC Press, 2014, ISBN 9781466560017.

[9] Hocking, J.: *Unity in action: multiplatform game development in C#*. Manning Publications Co., 2015, ISBN 9781617292323.

[10] Murray, H. J. R.: *A history of chess*. Benjamin Press, 1985, ISBN 0936317019.

[11] Parashift: *Serialization and Unserialization*. Dostupné na: http://www.parashift.com/c++-faq-lite/serialize-overview.html, [cit. 2017-12-28].

[12] Rollings, A.; Morris, D.: *Game architecture and design*. New Riders, 2004, ISBN 9780735713635.

[13] Schell, J.: *The art of game design: a book of lenses*. Elsevier/Morgan Kaufmann, 2008, ISBN 9780123694966.

[14] Segabits: *Total War series sells 11 million within the last 6 years*. Dostupné na: http://segabits.com/blog/2015/03/09/total-war-series-sells-11-million-within-the-last-6-years/, [cit. 2018-01-02].

[15] Techspot: *Top 10 Best MMOs*. Dostupné na: https://www.techspot.com/article/1434-the-best-mmos/, [cit. 2018-01-03].

[16] Tekinbas, K. S.; Zimmerman, E.: *Rules of Play: game design fundamentals*. MIT Press, 2003, ISBN 9780262240451.

[17] Tresca, M. J.: *The evolution of fantasy role-playing games*. McFarland & Co., 2011, ISBN 9780786458950.

[18] Tychsen, A.; Hitchens, M.; Brolund, T.; aj.: *The Game Master. Second Australasian Conference on Interactive Entertainment, the*, 2005: s. 78–79, 215–222.

[19] Unity Documentation: *ScriptableObject*. Dostupné na: https://docs.unity3d.com/ScriptReference/ScriptableObject.html, [cit. 2017-12-28].

[20] Unity Showcase: *Made with Unity*. Dostupné na: https://unity3d.com/showcase/gallery, [cit. 2018-01-04].

[21] Weiss, B.: *Classic home video games, 1972-1984: a complete reference guide*. McFarland, 2012, ISBN 9780786469383.

[22] Whittaker, J.: *Cyberspace handbook*. Routledge, 2004, ISBN 041516835X.

[23] Wirfs-Brock, R. J.; Wilkerson, B.: *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*. 1989.

[24] Wolf, M. J.: *Encyclopedia of video games: the culture, technology, and art of gaming*. Greenwood, 2012, ISBN 031337936X.

Příloha A

Obsah přiloženého paměťového média

Adresářová struktura:

- `Documentation` - Dokumentace diplomové práce.
  - `PDF` - Vysázená zpráva v čitelné podobě.
  - `LaTeX` - Zdrojové soubory dokumentace pro vysázení programem LaTeX.
- `Source` - Zdrojové soubory diplomové práce.
- `Logo` - Logo Unity++ ve vektorové podobě.
- `Screenshots` - Snímky obrazovky důležitých oken rozšíření.

Příloha B

Metriky kódu

- Velikost nezabalených zdrojových kódů: 23,9 MB
- Velikost .ZIP archívu se zdrojovými kódy: 4,77 MB
- Počet .META souborů: 89
- Počet .CS souborů: 68
- Napsáno řádků kódu: cca 8300 bez komentářů

Příloha C

Logo Unity++

Obrázek C.1: Logo projektu Unity++ (inspirováno originálním logem Unity)