Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Vývoj 2D herního enginu

DIPLOMOVÁ PRÁCE

Autor: **Bc. David Konečný**
Studijní obor: **Aplikovaná informatika, kombinované studium**

Vedoucí práce: **Ing. Karel Petránek**

Hradec Králové 2017

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval samostatně a za použití zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Děkuji Ing. Karlu Petránkovi za odborné vedení diplomové práce, za udílení cenných rad a připomínek, za bezproblémovou spolupráci na diplomové práci.

Děkuji své úžasné manželce za podporu, pomoc a trpělivost.

Anotace

Cílem diplomové práce bylo vytvořit engine vhodný pro 2D plošinovou hru. Byly stanoveny tři hlavní cíle, které musí engine splňovat s ohledem na vyvíjenou hru. Následoval průzkum napříč současnými herními enginy a zjištění, že žádný z nich není ideálním kandidátem pro splnění stanovených cílů. Vlastní řešení je vytvářeno v jazyce C#. Kromě některých cizích součástí (fyzikální engine, audio-vizuální framework) stojí také na vlastní obecné znovupoužitelné knihovně „Corpus“. Samotný engine je poměrně úzce spjat s hrou, pro kterou je vyvíjen. Díky tomu je možné mít v něm zakomponovanou konkrétní funkcionalitu, která není lehce použitelná v jiných herních žánrech. Engine je nyní plně funkční a splňuje stanovené cíle. Hra, pro kterou byl vyvíjen, na něm úspěšně staví a odpovídá současným standardům herního průmyslu.

Synopsis

The goal of this thesis was to create a game engine suitable for a 2D platformer game. Three main goals closely linked to the developed game were set. Follow-up research of contemporary game engines revealed that none of them is an optimal candidate for an accomplishment of these goals. An own solution is created in C# language. It is based on 3rd party components (physical engine, audio-visual framework) as well as on re-usable „Corpus“ library of own design. The engine itself is fairly closely connected to the game for which it is developed, thanks to which it contains some functionality unusable in other game genres. The engine is fully functional and meets all set requirements. Game for which this engine was developed is successfully built on top of it and meets all contemporary standards of a game industry.

Klíčová slova: Herní engine; Počítačová grafika; 2D; XNA; Farseer

Keywords: Game engine; Computer graphics; 2D; XNA; Farseer

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

1 Úvod

Tato diplomová práce se zabývá vývojem herního enginu pro použití ve 2D plošinové hře. Dané téma bylo zvoleno s ohledem na stále větší popularitu počítačových her jako média. V posledních letech se také velmi často mluví o herních enginech, které tyto hry pohání. Nezabývají se jimi již pouze specializované kanály, ale i média určená pro širokou veřejnost.

Následující kapitoly shrnují požadavky zmíněné 2D hry na herní engine a implementace takového enginu, včetně způsobu dosažení všech požadavků. Diplomová práce, její teoretická část, se také věnuje aktuální situaci enginů v herním průmyslu. Teoretická část popisuje několik nejznámějších aktuálně používaných herních enginů, jejich vlastnosti a historii.

2 Cíle

Cílem diplomové práce je vytvořit herní engine pro hru Other Inside. Jedná se o 2D kreslenou plošinovou hru, která v sobě kombinuje prvky logických her a adventur [1].

Nebude se tedy jednat o univerzální engine. Jeho znovupoužitelnost bude na pomezí herní série a herního žánru (viz. 3.2.1).

Engine musí splňovat některé základní vlastnosti, plynoucí ze stylu hry, pro kterou vzniká. Těmto vlastnostem se věnují následující podkapitoly.

2.1 Vykreslování 2D grafiky

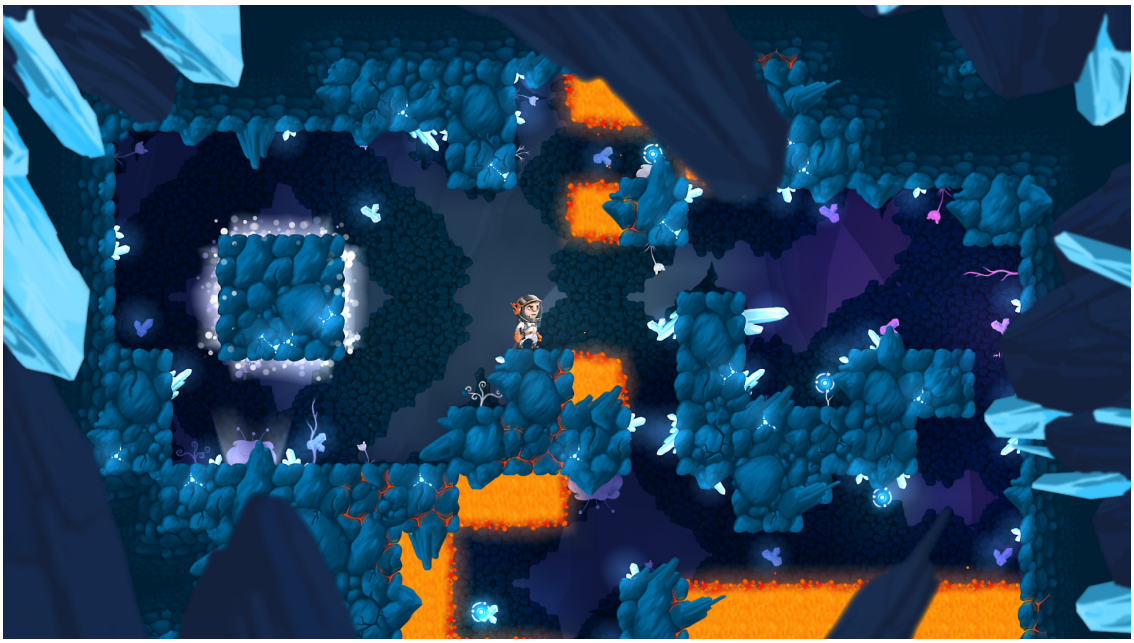
Hlavním úkolem enginu je starat o vykreslování herních scén. Bude optimalizovaný pro 2D grafiku ve vrstvách. Nicméně je zde jedna podstatná změna oproti většině 2D plošinových her. Tento typ her v naprosté většině případů nezobrazuje plochu podlahy, po které se pohybují postavy (příklad na obrázku č. 1). Tento přístup usnadňuje vykreslování, protože postavy nikdy nepřekrývají (ani nejsou překrývány) grafikou, jenž tvoří podlahy a stěny. To umožňuje mít všechny tyto grafické prvky v jedné vrstvě.

Hra Other Inside využívá ale odlišný způsob, kdy pracuje s perspektivou. Tento přístup je spíše známý z kreslených adventur. V takových hrách lze vidět podlaha, po které postavy chodí (příklad na obrázku č. 2) a tvůrci musí skládat grafické elementy do vrstev podle toho, co je v popředí a co v pozadí. Největší problém nastává v situaci, kdy má jeden grafický element v některých místech postavy překrývat a jindy by měl být v pozadí (naznačeno na obrázku č. 3, kde kvádr uprostřed je jedna 2D textura). Tento problém je větší, než může na první pohled vypadat. Především pokud jsou použity poloprůhledné textury. Toto bude muset vyvíjený herní engine řešit.

Dále bude kladen důraz na možnosti využívání různých postprocessing efektů, ať už na jednotlivé textury, nebo celou scénu.

2.2 Paralelní světy

Hra Other Inside má jedno specifikum v tom, že se hráč dynamicky přepíná mezi dvěma verzemi herního světa. V každé verzi herního světa mohou figurovat jiné herní



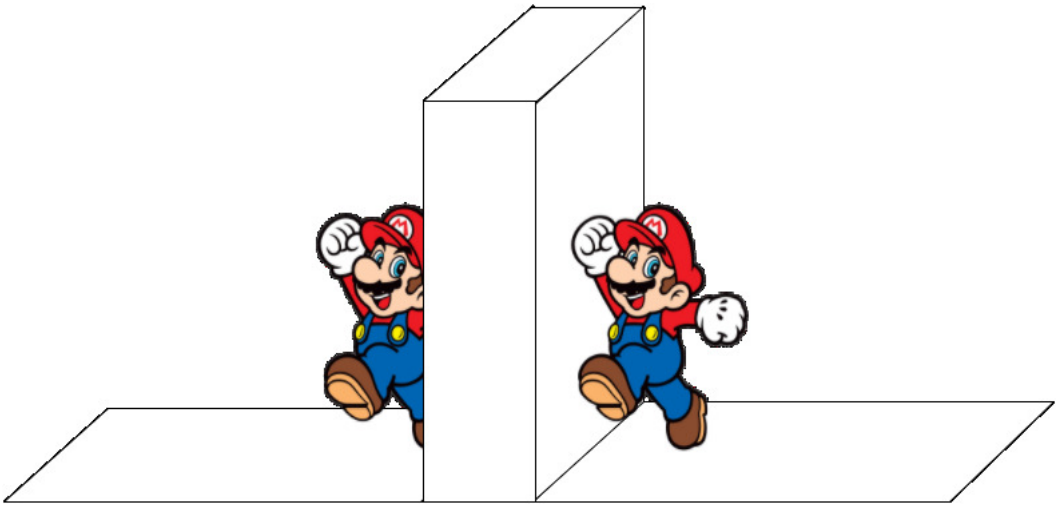
Obrázek 1: Screenshot ze 2D plošinovky Blackhole.

Obrázek 2: Screenshot z adventury Machninarium.

objekty a přepnutí bude možno prakticky v libovolný okamžik. Na tomto principu budou velmi často založeny logické rébusy ve vyvíjené hře.

Zároveň bude gameplay spoléhat na reálně se chovající simulaci fyzikálních těles. Bude se jednat o různé přesouvání či přemisťování a tahání předmětů, simulace lan a řetězů, ovlivňování gravitace a další.

Každý herní objekt musí tedy vědět, do které verze světa patří a jeho chování tomu musí odpovídat. Zároveň kombinace paralelních světů a simulace fyziky s sebou



Obrázek 3: Demonstrace 3D perspektivy ve 2D hře.

přináší další komplikace. Určité fyzikální těleso, se kterým právě probíhá interakce, může náhle zmizet. Případně se může jiné těleso objevit na nevhodném místě (v jiném tělesu). Tyto situace musí umět engine řešit.

2.3 Větvení

Hra je založena na reflektování hráčových voleb. Z tohoto důvodu musí engine podporovat různé formy větvení. Cílem je vytvořit systém, který dokáže vzít v potaz aktuální stav herního světa i minulé hráčovy volby. Ty, pomocí sady logických pravidel, vyhodnotit a vykonat určitou akci na základě výsledku.

3 Herní engine

Na úvod je třeba definovat některé pojmy, které se budou v následujícím textu vyskytovat.

3.1 Pojem hra

Pojem "hra"nemá jasnou a ustálenou definici. Johan Huizinga ve své práci Homo Ludens definuje hru jako "*dobrovolnou činnost, která je vykonávána uvnitř pevně stanovených časových a prostorových hranic, podle dobrovolně přijatých, ale bezpodmínečně závazných pravidel, která má svůj cíl v sobě samé a je doprovázena pocitem napětí a radosti a vědomím jiného bytí, než je všední život*" [2].

Důležité body zmíněné definice jsou, že hra je **dobrovolná** činnost se **stanovenými pravidly**. Hra se hraje pro potěšení ze samotné hry.

Další vlastnosti, které splňují všechny nebo převážná většina her jsou: jsou umělé, obsahují konflikt, mají cíle, zahrnují tvorbu rozhodnutí a mají nejistý výsledek.

3.1.1 Videohra a herní žánry

Videohra je podmnožinou her. Jedná se o elektronické hry, se kterými hráč přichází do interakce skrze televizní, počítačovou nebo jinou obrazovku. Do této kategorie

spadají počítačové hry, konzolové hry, mobilní hry, handheldové hry a další.

Pro účely tohoto textu je pojem "hra"zaměnitelný za pojem "videohra".

Počátky videoher sahají do roku 1958. Tehdy v amerických jaderných laboratořích Brookhaven National Laboratory muž jménem William A. Higenbotham připojil k počítači několik analogových ovladačů, jejichž prostřednictvím chtěl na osciloskopu simulovat pohyb letícího míčku. Svůj prototyp zdokonalil a vytvořil tak jednoduchou tenisovou hru pro dva hráče. Tuto hru předvedl na místní univerzitě, kde zaznamenal značný úspěch [3].

V současné době jsou videohry právoplatnou součástí kultury. Do herního průmyslu se investuje obrovské množství prostředků a odbyt herních titulů nezřídka bývá v milionech kusů. V roce 2016 bylo vydáno několik systémů pro herní virtuální realitu, což kladně vypovídá o rozvoji tohoto odvětví. Rovněž Evropská unie a některé vlády již před několika lety zařadily do svých programů pro podporu audiovizuální tvorby i videohry.

V posledních letech se často diskutuje o tom, zda je možno chápat hry jako umění. Od svého počátku urazily videohry velkou cestu a čím dál tím více se objevují tituly vyzdvihované kritiky za jejich umělecký přínos. Společnost se postupně transformuje a učí se videohry akceptovat jako novou formu umění. Nicméně tento přerod bude ještě několik let trvat.

Videohry můžeme dělit podle žánrů. Nejznámější a nejrozšířenější jsou [4]:

- **Simulace**: patří sem simulátory **sportů** (*série NHL*), **vozidel** (*Need for Speed, Microsoft Flight Simulator*), **výstavby a managementu** (*Sim City, Roller Coaster Tycoon*) a další.

- **Strategie**: kladou důraz na myšlení a plánování. Často se dále dělí na **Real Time Strategie** (*Warcraft, Command & Conquer*) a **Turn Based Strategie** (*Heroes of Might & Magic*).

- **Akce**: hry rychlejšího tempa, od hráče vyžadují rychle řešení netriviálních situací. Jsou založeny na postřehu. Známými příklady jsou hry *Call of Duty*, *Gears of War* nebo *Super Meat Boy*.

- **Role-playing**: snaží se o vtažení hráče do role herní postavy, založeny na příběhu. Role herní postavy může být pevně daná (série titulů *Sherlocke Holmes*) nebo pouze naznačena a hráč si ji vytváří sám (*The Elder Scrolls*). Do této skupiny spadá žánr **adventur** (*Broken Sword, Polda*) a dále tradiční **RPG** hry vycházející ze stolních her na hrdiny (**Dungeons & Dragons**).

Na tomto místě by bylo vhodné zmínit některé další žánry, které si často berou některé prvky z již zmíněných. Jsou to například **logické hry** (*Hledání min, Bejeweled*), které kladou důraz na myšlení, stejně jako strategie, ale mohou si brát i elementy z akčních her (postřehové) nebo simulací. Podobně by se dalo mluvit o **hudebních hrách** nebo tzv. **open-world hrách**. Herní žánry se také často kombinují, příkladem toho je dnes velmi populární žánr **akčních adventur**.

Na závěr je třeba zmínit dělení her dle způsobu jejich vnímání. Dělení by mohlo vypadat takto:

- **Textové hry**: herní situace je reprezentována pouze textem.

- **1st person hry**: hra je hrána z pohledu hlavní herní postavy.

- **3rd person hry**: hlavní herní postava nebo postavy jsou viditelná na obrazovce, hráč může mít určitou kontrolu nad virtuální kamerou, která snímá herní svět. Lze zde zařadit strategie viděné z ptačího pohledu, side-scroller, plošinové hry, a další.

3.2 Význam enginu

Herní engine je sada softwarových nástrojů sloužících k vývoji hry. Tyto nástroje již mají zabudovanou potřebnou nízkoúrovňovou logiku a jsou znovupoužitelné [5]. Než se začneme plně věnovat herním enginům, je třeba ujasnit si pojmy knihovna a framework [6]. Tyto pojmy se v praxi velmi často zaměňují.

Knihovna (library) je znovupoužitelná kolekce kódu a dat. Knihovna dává k dispozici prostředky pro provádění operací z určité specifické domény. Může se jednat o vykreslování grafiky, přehrávání hudby, simulaci fyziky a podobně. Příkladem knihoven používaných při vývoji her jsou *Box2D* nebo *FMOD*.

Framework je navzájem propojená kolekce knihoven, určená pro konkrétní skupiny úloh (například tvorbu her). Knihovny mohou pocházet jak z cizích zdrojů, tak přímo od tvůrců frameworku. Příkladem frameworku jsou *XNA* nebo *OGRE*. Jedná se stále o poměrně obecný celek, který pouze dává k dispozici prostředky, ale většinou nestanovuje, jak je přesně používat.

Engine staví na frameworku, na kterém buduje svou vlastní architekturu. Engine především poskytuje strukturu pro uspořádání všech herních prvků, **celého herního světa**. Obsahuje funkce pro stavbu, správu, ukládání a načítání herního světa. Dle úrovně specializace může engine také nabízet určitou specifickou funkcionalitu, například engine určený pro 3rd person střílečky může mít již implementovanou kameru pro tento typ her.

Herní enginy zpravidla obsahují moduly pro práci s:

- Grafikou (vykreslování)
- Zvukem (přehrávání)
- Vstupy (klávesnice, myš, gamepad, dotykový display, ...)
- Assety (soubory obsahující hudbu, zvuky, grafiku, ...)

Dále většinou obsahují alespoň některé z modulů podporujících:

- Fyzikální simulaci
- Umělou inteligenci
- Simulaci částic
- Uživatelské rozhraní
- Síťovou komunikaci
- a další

3.2.1 Univerzálnost enginu

Herní enginy lze rozdělit podle univerzálnosti. Hlavní příspěvek samostatného enginu tkví v jeho znovupoužitelnost. Nicméně vytvořit skutečně univerzální engine je velmi náročné a je třeba dělat mnoho kompromisů. Platí zde pravidlo, že čím univerzálnější má engine být, tím náročnější jeho tvorba a správa bude. Stanovme si tedy několik úrovní univerzálnosti herních enginů:

1. **Zabudován do hry**: engine je pevně spojený se samotnou hrou a nelze snadno oddělit. Znovupoužitelnost je komplikovaná a bude vyžadovat značný refactoring.

2. **Znovupoužitelný v rámci herní série**: engine byl vyvinut pro jednu konkrétní hru, nicméně je oddělitelný a lze jej opět použít, například v pokračování hry. Engine je natolik specifický, že jako celek nemůže být využit v jiné hře. Takový engine umožňuje snadný vývoj pokračování herního titulu a je průběžně rozvíjen a vylepšován.

3. **Znovupoužitelný v rámci herního žánru**: Obdobné jako v předchozím případě, nicméně engine je možno znovu využívat ve větší množině her. Jedná se například o některé enginy ze závodních her - takový engine jen těžko najde uplatnění ve strategické hře, je ale vysoce znovupoužitelný v jiném závodním titulu.

4. **Skutečně univerzální**: Engine je možno zcela oddělit od hry a žánru a lze na něm postavit jakoukoliv hru. Velmi náročný na vývoj, takových enginů je minimum a jsou většinou vyvíjeny přímo za tímto účelem.

Jak již bylo nastíněno, redukce univerzálnosti a vyšší navázání enginu na konkrétní hru nebo žánr má také svá nevyvratitelná pozitiva. Tento přístup umožňuje rychlejší vývoj enginu, protože jeho funkcionalita je omezena výběrem žánru nebo hry. Dále je takový engine zpravidla více optimalizovaný. Funkcionalita, kterou by bylo třeba u univerzálnějšího enginu stavět nad již vybudovanou architekturou, je zde možné začlenit do samotného jádra a přistupovat tak k některým zdrojům mnohem příměji.

V praxi je při vývoji enginu vždy potřeba najít správnou míru univerzálnosti a často kompromis mezi oběma přístupy.

3.2.2 Významné příklady

V současné době se o herních enginech mluví více než kdy dříve. Herní komunitu otevřeně zajímá zákulisí vývoje her a vývojářská studia uvolňují informace o svých enginech (často jako formu marketingu). Vedle toho se v posledních letech změnil přístup k již zavedeným enginům, ke kterým se nyní velmi jednoduše dostává i široká veřejnost. Důvodem jsou především změny licenčních politik, jež jsou vstřícné k nezávislým herním vývojářům.

Několik nejvýznamnějších enginů současnosti je zmíněno na následujících stránkách. Výčet rozhodně není kompletní, enginů existuje v současnosti obrovské množství, navíc ne všichni vývojáři her sdílí s veřejností takto nízkoúrovňové informace o svých titulech.

3.2.2.1 Unreal Engine

Unreal Engine je herní engine vyvíjený společností Epic Games v jazyku C++ určený především pro 1st person střílečky [7].

Obrázek 4: Screenshot technologického dema Unreal Engine 4.

Engine byl poprvé představen v roce 1998 ve hře *Unreal*. Engine byl od začátku navrhován způsobem, aby mohl být dlouhodobě rozšiřován a vylepšován. Unreal Engine 2 byl vydán v roce 2002 současně se hrou *America's Army*. Jádro a vykreslovací modul byly v této verzi zcela přepsány. Později vznikla verze 2.5 přinášející mnohá rozšíření a vylepšení (například zlepšení vykreslovacího výkonu, byla přidána fyzikální simulace vozidel, podpora 64 bitových systémů a další). Vznikla také specializovaná edice enginu nazvaná UE2X. Tato verze obsahovala optimalizace určené speciálně pro konzoli Xbox. V roce 2004 byl představen Unreal Engine 3, který hojně využíval programovatelné shadery grafických karet. Engine měl poměrně přísnou licenční politiku, což komplikovalo především nezávislým vývojářům vydávání her. Odpovědí na tento problém byl Unreal Development Kit vydaný roku 2009. Jedná se o volně dostupný Unreal Engine 3 včetně vývojových nástrojů. Pro nekomerční účely je zcela zdarma. V roce 2014 byla vydána zatím poslední verze Unreal Engine 4. Původně byly všechny zdrojové kódy a nástroje k dispozici za $19 měsíčně + 5% z hrubých výdělků za produkty postavené na tomto enginu. Od roku 2015 je engine zcela zdarma. Zůstává pouze 5% z hrubých výdělků, pokud výdělky přesáhnou $3 000 za čtvrtletí.

V současnosti poslední verze Unreal Enginu podporuje všechny rozšířené počítačové platformy (Windows, Linux, OS X), herní konzole (Xbox One, Playstation 4) i mobilní platformy (iOS, Android). Mezi nejvýznačnější prvky enginu patří podpora DirectX 11 a DirectX 12, volný přístup ke kompletním zdrojovým kódům, podpora umělé inteligence, postprocessing efektů, animací postav a mnoho dalších [8]. Na obrázku č. 4 je ukázka z technologického dema této verze enginu.

I když byl engine vyvinut primárně pro 1st person střílečky, úspěšně se využívá v řadě dalších herních žánrů. Mezi významné hry používající Unreal Engine patří *Gears of War*, *Bioshock Infinite*, *Batman: Arkham Knight*, *Dishonored*, *Borderlands*

nebo série *Mass Effect*.

3.2.2.2 Unity 3D

Unity je multiplatformní herní engine vyvinut společností Unity Technologies. Cílí především na univerzálnost [9].

Obrázek 5: Screenshot technologického dema Unity 5.

Engine byl původně představen v roce 2005, od té doby bylo vydáno již 5 verzí. Existují 2 edice vývojářských balíčků, Personal a Professional. Edice Personal je zdarma pro herní vývojáře, jejichž zisk nepřesahuje $100 000 ročně. Původně tato edice nenabízela všechnu funkčnost Professional edice (chybělo například, v dnešní době téměř nezbytné, vykreslování do textury). Pravděpodobně jako reakci na licenční podmínky Unreal Engine 4 (viz. 3.2.2.1) byla Personal edice v roce 2015 rozšířena a nyní nabízí téměř kompletní funkcionalitu Professional edice [10].

Unity umožňuje v současnosti vyvíjet hry pro více než 15 platforem. Do výčtu patří Windows, OS X, Linux, Xbox 360, Xbox One, Playstation 3, Playstation 4, PlayStation Vita, Wii, Wii U, Nintendo 3DS, iOS, Android, Windows Phone a jsou poskytovány i pluginy do webových prohlížečů. Engine se tedy zaměřuje prakticky na všechny platformy, na kterých je v současnosti možno hrát hry. I z toho důvodu se těší velké oblibě.

Engine obsahuje nástroje pro vývoj 2D i 3D her a mezi jeho hlavní přednosti patří skriptování ve třech jazycích (C#, JavaScript, Boo), simulace fyziky, snadný deployment na různé platformy, inverzní kinematika a další [11]. Na obrázku č. 5 lze vidět screenshot z technologického dema poslední verze enginu. Mezi herní tituly vyvinuté v enginu Unity se řadí například *FireWatch*, *Ori and the Blind Forest*, *Hearthstone: Heroes of Warcraft*, *Gone Home* nebo *The Room*. Jak lze vidět, jedná se hlavně o tituly nezávislých herních tvůrců. Především ti si tento engine velmi oblíbili.

3.2.2.3 CryEngine

CryEngine je herní engine určený původně především pro 1st person střílečky vyvinut německým vývojářským studiem Crytek [12]

Obrázek 6: Screenshot z připravované hry Kingdom Come.

Engine byl poprvé využit ve hře *Far Cry*, kterou vyvinul samotný Crytek původně pouze jako technologické demo, a která vyšla v roce 2014. Engine byl původně využíván především společností Crytek, která na něm vyvíjela své vlastní tituly (série *Crysis*). Až třetí verze enginu uvedená v roce 2009, CryEngine 3, začala být více licencována dalšími herními studii. V roce 2013 Crytek přestal číslovat nové verze CryEnginu, od té doby jsou nové verze bez číslovky na konci. V současnosti je engine dostupný pro kohokoliv za cenu od $8,33 měsíčně (sleva za půl roční předplatné) skrze platformu Steam [13], případně lze dohodnout i standardní licencování.

Engine se pyšní především: grafickými schopnostmi, poskytovanými nástroji, možnostmi animace postav, podporovanými herními platformami (Windows, Playstation 4, Xbox One, Linux, Oculus Rift), možnostmi audia a výkonem [14]. Mezi hry, které používají tento engine, patří mimo již zmíněné například *Evolve*, *Everybody's Gone to the Rapture* nebo právě nyní vyvíjena česká hra *Kingdom Come: Deliverance* (ukázka na obrázku č. 6).

3.2.2.4 id Tech

id Tech je série enginů vyvinutých společností id Software pro jejich 1st person střílečky [15].

První hrou, ve které byl použit engine id Tech 1 byla hra *Doom* od id Software, proto se původně engine nazýval "Doom engine". Tato hra byla vydána v roce 1993, o 6 let později, v roce 1999, byl engine uvolněn pod GLP licencí. Engine vykresluje 3D svět podle dvourozměrného půdorysu. Zdi a podlahy musí být v herním světě kolmé a nelze vytvářet vícepodlažní struktury. Mimo *Doom* pohání tento engine také například hry *Heretic*, *Hexen* nebo *Strife* [16].

id Tech 2 engine byl představen světu v roce 1996 společně s hrou *Quake*. Oproti předchozímu je zde implementováno skutečné 3D vykreslování a podpora multipla-

Obrázek 7: Screenshot ze hry Doom.

yeru. Později byl engine rozšířen, aby mohl pohánět hru *Quake II*. Roku 2001 byl engine opět uvolněn pod svobodnou licencí GPL.

Pro svou následující hru *Quake III Arena* vyvinula společnost id Software engine id Tech 3. Hra vyšla v roce 1999 a ve své době engine konkuroval především Unreal Enginu (viz. 3.2.2.1). Narozdíl od Unreal Enginu je id Tech 3 postaven na OpenGL. Vzhledem k multiplayerové povaze *Quake III Arena* klade engine velký důraz na optimalizaci síťové komunikace. V roce 2005 byl engine opět uvolněn pod GPL licencí [17].

Id Tech 4 byl vyvinut pro hru *Doom 3*, vydanou v roce 2004. Engine přinesl především různá vylepšení grafiky, jako jsou bump mapy, normálové mapy, per-pixel nasvícení a další. Dalšími hrami využívající id Tech 4 jsou například *Prey*, *Quake 4* nebo *Brink*. Stejně jako jeho předchůdci byl engine uvolněn pod licencí GPL, a to v roce 2011.

V roce 2011 vyšla hra *RAGE*, kterou pohání id Tech 5. Hlavním prvkem enginu je tzn. MegaTexture technologie, díky níž je možné používat textury o rozlišení až 128 000 x 128 000 bodů. Tyto textury mohou obsahovat i informace pro fyzikální simulaci a další negrafické informace. Engine zatím nebyl vydaný pod GPL licencí, ačkoliv některé dřívější rozhovory s vývojáři naznačují, že se tomu tak v budoucnu stane. Engine nelze licencovat, je využíván pouze pro herní projekty vydavatele ZeniMax, pod nějž spadá i id Software [18].

Poslední verze tohoto herního enginu nese název id Tech 6. Tento engine pohání hru *Doom* (zobrazena na obrázku č. 7), vydanou roku 2016. Zda bude engine dostupný i herním vývojářům mimo ZeniMax, není zatím známo.

3.2.2.5 Dunia

Dunia je 1st person engine určený pro herní sérii *Far Cry* patřící vydavateli a hernímu vývojáři společnosti UbiSoft.

První díl této herní série byl vytvořen společností Crytek a využíval CryEngine verze 1 (viz. 3.2.2.3). Další díly již vyvíjí společnost UbiSoft za použití svého

vlastního enginu Dunia. Engine je založen na CryEngine verze 1, nicméně se odhaduje, že původního kódu zůstaly pouze přibližně 2-3%. Proto lze Dunii považovat za nový engine. Některými vyzdvihovanými prvky enginu jsou simulace šíření ohně, dynamické počasí nebo pokročilá umělá inteligence. Engine nebyl nikdy poskytnut jinému vývojářskému studiu a je pevně svázán s jedinou sérií videoher [19].

3.2.2.6 Anvil

Anvil je 3rd person engine vyvinutý interně společností UbiSoft především pro jejich herní sérii *Assasin's Creed* [20].

V roce 2007 vyšla první hra poháněná enginem Anvil. Dlouhou dobu byl engine svázán pouze se sérií 3rd person akčních adventur *Assasin's Creed*. Až v roce 2015 vychází 1st person taktická střílečka *Tom Clancy's Rainbow Six: Siege*, která je poháněna enginem AnvilNext 2.0. Vypadá to, že po 8 letech vývoje se engine rozšiřuje a zvyšuje se jeho univerzálnost. Engine je používán pouze společností UbiSoft pro jejich projekty, externí licencování nelze.

3.2.2.7 Disrupt

Disrupt je 3rd person engine vytvořený společností UbiSoft pro konkrétní herní titul *Watch Dogs* [21].

Engine velmi dobře využívá faktu, že byl navržen pro jedinou hru (v budoucnu budou pravděpodobně vydány další hry ze stejné série). Engine má zabudované dynamické simulování herního světa, pod čímž si lze představit simulaci osob, počasí a elektřiny, kdy všechny tyto systémy na sebe vzájemně reagují. Dále je zde kladen velký důraz na dopad hráčových akcí na herní svět. Vše výše zmíněné vyznívá jako popis konktrétní hry. Je tomu tak v důsledku těsného svázání enginu s hrou. V neposlední řadě je také engine budován kolem myšlenky bezešvého multiplayeru, kdy je celý herní svět možné velmi optimálně sdílet s dalšími hráči.

Podporovanými platformami jsou Xbox 360, Playstation 3, Xbox One, Playstation 4 a Windows.

Engine jako takový je stejně jako předchozí enginy Anvil (3.2.2.6) a Dunia (3.2.2.5) určen pouze pro interní užití společnosti UbiSoft a nelze ho v současnosti licencovat.

3.2.2.8 Frostbite

Frostbite je engine vyvinutý společností Electronics Arts (konkrétně studiem DICE), původně určený pro 1st person střílečky, především ze série *Battlefield*. Postupem času se jeho univerzálnost rozvinula tak, že nyní na něm staví i závodní hry, strategie, 3rd person RPG a další žánry [22].

Engine byl poprvé použit v roce 2008 ve hře *Battlefield: Bad Company*, po které následovalo několik dalších 1st person střílecích her. Ty většinou těžily především z fyzikální simulace zničitelnosti prostředí, která je v enginu zabudována. Druhá verze enginu, Frostbite 2, byla poprvé využita v roce 2011 ve hře *Battlefield 3*. Tato verze se již nesoustředila pouze na 1st person hry, ale byla na ní postavena i závodní hra *Need for Speed: The Run* a 3rd person kooperativní střílečka *Army of Two: The Devil's Cartel*. Třetí verze enginu vyšla v roce 2013 společně se hrou *Battlefield 4*.

Tato verze byla již natolik univerzální, že byla použita také ve sportovní hře *Rory McIlroy PGA Tour* nebo realtime strategii *Command & Conquer*.

Engine je v současnosti využíván pouze herními studii spadajícími pod společnost Electronic Arts, externí licencování není možno. Mezi podporované platformy patří Windows, Playstation 3, Playstation 4, Xbox 360 a Xbox One. Engine se snaží obsáhnout všechny aspekty vývoje her. Stará se tedy o audio, animace, filmové sekvence, umělou inteligenci, skriptování, simulaci fyziky, vykreslování grafiky nebo různé vizuální efekty [23].

3.2.2.9 HPL Engine

HPL engine je 1st person 3D engine vyvinut švédským nezávislým herním studiem Frictional Games [24].

Obrázek 8: Screenshot ze hry SOMA.

Původně 2D engine vytvořený pro školní závěrečnou práci byl rozšířen o 3D vykreslování a roku 2006 poprvé použit v herní sérii *Penumbra*. Engine byl pojmenován HPL Engine 1 (podle spisovatele H.P. Lovecrafta) a v roce 2010 byl uvolněn pod GPL licencí včetně zdrojových kódů počítačové hry *Penumbra: Overture*. HPL Engine 2 je rozšíření původního enginu o několik technologických vylepšení. Tato verze byla poprvé využita ve hře *Amnesia: The Dark Descent*, která vyšla v roce 2010. Poslední verze, HPL Engine 3, nově podporuje vykreslování otevřených prostranství (především co se týče nasvícení a generování terénu). Tuto verzi herního enginu využívá hra *SOMA* vydaná v roce 2015, zachycená na obrázku č. 8 [25].

Engine je psaný v jazyku C++ a využívá knihovny OpenGL (vykreslování), OpenAL (audio) a Newton Game Dynamics (simulace fyziky). V současnosti je známo, že engine podporuje platformy Windows, Linux, OS X a Playstation 4. Mimo první verze není engine dostupný pro veřejnost a je využíván výhradně pro herní projekty Frictional Games.

Protože je engine vyvíjen pro konkrétní žánr 1st person hororových adventur, jsou tomu jeho schopnosti přizpůsobeny. Mezi jeho přednosti patří práce s nasvícením scény a stíny (slouží k budování hororové atmosféry) a rozšířené využívání

fyzikálního enginu pro interakci s prakticky všemi dynamickými objekty v herním světě. Dále má engine zabudováno například skriptování herních situací nebo nástroje pro správu umělé inteligence nepřátel.

3.3 Volba enginu

Největší zástupci univerzálních volně dostupných enginů jsou CryEngine, Unreal Engine a Unity. Při současném vývoji her se prakticky jiné enginy nepoužívají, pokud si vývojáři netvoří vlastní. Bylo tedy potřeba zvážit vhodnost těchto tří enginů pro splnění stanovených cílů (viz. kapitola 2).

CryEngine a Unreal Engine jsou enginy, jež jsou určeny primárně pro 3D hry (především 1st person a 3rd person hry). Podpora 2D grafiky (vykreslování 2D quadů, sprite animace, apod.) v nich neexisstuje, respektive je velmi omezená.

Unreal Engine sice obsahuje Paper 2D, což je systém pro tvorbu 2D her za použití spritů [26]. Nicméně Paper 2D není ideálním kandidátem pro 2D hru simulující 3D perspektivu, jako je *Other Inside* (viz. kapitola 2.1). Dále je Paper 2D určitou nádstavbou, proto nelze jednoduše využít všech vlastností Unreal Enginu.

Engine Unity oproti tomu umožňuje vývoj 2D i 3D her ve stejném prostředí. Většina nástrojů v tomto enginu (například animation controller) je uzpůsobena tak, aby pracovala s 2D i 3D grafikou. V tomto ohledu je Unity ideální volbou.

Nicméně se jedná o uzavřený engine a není tedy možné (oproti například Unreal Engine) rozšiřovat jeho jádro. Díky tomu nelze zcela optimálně implementovat paralelní herní světy (viz. kapitola 2.2). Přesto se jedná o nejvhodnějšího kandidáta.

Nicméně ani jeden z těchto enginů neřeší již zmíněný problém s 3D perspektivou ve 2D hře.

3.3.1 Problém s perspektivou

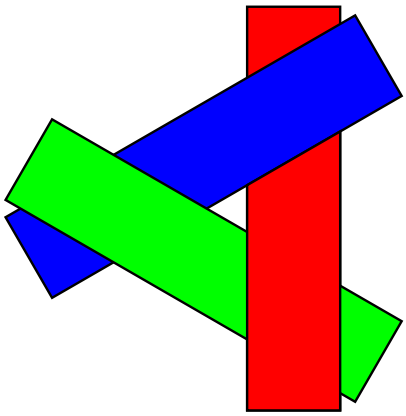
Při vykreslování grafických objektů na počítačovou obrazovku je třeba si uvědomit, že některé objekty budou překrývat jiné. Objekty v popředí, blíže hráči, překrývají objekty dále od obrazovky, v pozadí. Tímto se utváří v herní scéně prostor.

Jedním z nejjednodušších přístupů k této problematice je takzvaný *Painter's algorithm* [27]. Ten funguje na jednoduchém principu, grafické objekty se vykreslují podle vzdálenosti. Nejdříve jsou kresleny nejvzdálenější objekty z pozadí a postupuje se dopředu směrem k hráči. Tímto je zajištěno překrývání pozadí popředím.

Tento algoritmus nicméně selhává ve chvíli, kdy se několik grafických objektů navzájem překrývá způsobem, který je naznačen na obrázku č. 9. Což je přesně problém, který bude často nastávat u vyvíjené hry (viz. kapitola 2.1). Pro řešení tohoto a podobných případů byl vyvinut Z-buffer.

Z-buffering je technika, při které se během vykreslování zapisuje do grafické karty informace o hloubce každého pixelu. Každý další pixel vykreslován na již "obsazenou"pozici, je vykreslen pouze tehdy, pokud je jeho hloubka menší, než vykreslený pixel - je tedy blíže k hráči [28]. Nicméně zásadní vadou Z-bufferingu je, že nelze použít na poloprůhledné textury, čímž se stává pro hru *Other Inside* nepoužitelným.

Pokud nelze použít Z-buffering a očekává se, že budou nastávat situace naznačené na obrázku č. 9, neexistuje žádné univerzální řešení [29]. V tom případě je třeba hledat řešení v závislosti na vyvíjené hře. Pokud je známo, jakým způsobem bude grafika ve hře fungovat a jak bude prezentovaná hráči, je možné připravit engine a herní scény tak, aby bylo vykreslování korektní.



Obrázek 9: Ukázka případu, kdy Painter's algorithm selže

Žádný z nabízených enginů neřeší zmíněný problém, jednoduše z toho důvodu, že obecné řešení neexistuje. Pro optimální implementaci řešení pro tento konkrétní projekt je potřebný přístup k jádru enginu. V opačném případě je řešení komplikované a vyžaduje obcházení některých částí enginu. I v případě dostupnosti zdrojových kódů enginů je ale implementace dosti obtížná, protože se jedná o skutečně nízkoúrovňovou funkcionalitu.

Především z tohoto důvodu byla zvolena cesta vlastního enginu. Tímto způsobem je možno lehce reflektovat vytyčené cíle již v nízkoúrovňových částech kódu a celý engine tvořit tak, aby byl na konkrétní hru vhodně připraven.

3.4 XNA Framework

XNA je volně dostupný framework pro vývoj her od firmy Microsoft. Je postavený na .NET frameworku a umožňuje snadný vývoj her pro platformy Windows, Xbox 360, WindowsPhone a původně Zune [30].

K oznámení XNA došlo v roce 2004. První verze byla vydána v roce 2006, o rok později, v roce 2007, následovala verze 2. Verze 3 vyšla v roce 2007. Poslední verze 4 byla vydána v září 2010. V roce 2013 byl oznámen konec vývoje XNA, nicméně komunita je stále aktivní a soustředila se kolem projektu MonoGame (viz. 3.4.1), který i nadále poskytuje herním vývojářům podporu XNA a původní framework dále rozvíjí [31].

XNA se skládá z nástrojů pro práci s [32]:

- **Audiem**: Přehrávání hudby a zvuků, správa zvukových zdrojů a 3D audia.
- **Herním obsahem**: Pracuje se zvuky, efekty, texturami, modely, fonty, strukturovanými daty libovolného významu a dalšími typy obsahu. Ty kompiluje do interního formátu XNA frameworku, umožňuje k nim přistupovat a spravuje je v paměti počítače.
- **Herními službami (gamer services)**: Zaštituje rozšířenou interakci s hráčem skrze možnosti dané platformy. Jedná se zpravidla o správu herních achievementů, přístup k hráčovu profilu, avataru, seznamu přítel, ...
- **Grafikou**: Frameworku obaluje nízkoúrovňové rozhraní DirectX a umožňuje vykreslovat grafiku za pomocí hardwarové akcelerace. Framework obsahuje 2 použitelné přístupy k vykreslování grafiky:

  - Nízkoúrovňové vykreslování za použití shaderů (v názvosloví XNA *Effect*), transformačních matic, vertexů a podobných základních primitiv.
  - Vykreslování pomocí *SpriteBatch*, což je rozhraní určené pro snadné vykreslování jednoho a více 2D obrázků, tzv. spritů. Oproti předchozímu bodu nabízí méně možností, ale je mnohem jednodušší na pochopení především pro začátečníky.

  Do určité míry lze oba zmíněné přístupy kombinovat.

- **Vstupy**: Zpracování vstupů z klávesnice, počítačové myši, Xbox 360 gamepadu a dotekových obrazovek.
- **Síťovou komunikací**: Nástroje pro implementaci multiplayeru a síťové komunikace. Rovněž jsou zde prostředky pro podporu platformy Xbox LIVE.
- **Úložišť**: Univerzální rozhraní pro ukládání a čtení souborů, například uložených stavů hry. Stará se za herní vývojáře například o správné místo k uložení souborů na rozličných platformách.
- **Médii**: Přístup k playlistům, albům, písničkám nebo obrázkům.

K vývoji her na frameworku XNA je k dispozici prostředí XNA Game Studio, které se integruje do vývojového IDE Visual Studio. K programování je možno využívat jazyků C# nebo Visual Basic.

Od verze 4.0 jsou v XNA dostupné 2 profily pro vykreslování grafiky. **Reach** slouží pro vývoj aplikací na všechny zmíněné platformy, včetně mobilních telefonů. Možnosti vykreslování jsou zde omezeny, rozlišení textur je maximálně 2048 x 2048 pixelů a například některé kombinace color blendingu nejsou dostupné. Profil **HiDef** taková omezení nemá, oproti tomu vyžaduje výkonnější grafickou kartu s podporou alespoň DirectX 10 a shader modelu verze 3 [33].

3.4.1 MonoGame

MonoGame je opensource implementace XNA Frameworku 4.0. Po ukončení vývoje XNA získalo MonoGame mnoho příznivců a stále se aktivně vyvíjí. Díky téměř identickému rozhraní je i pro již existující projekty v XNA přechod na MonoGame poměrně snadný [34].

Hlavním impulsem pro vznik MonoGame byla snaha rozšířit aplikace vyvíjené v XNA na další platformy. V aktuální verzi frameworku je možné vyvíjet aplikace pro Windows, Xbox 360, Windows Phone, Linux, OS X, iOS, Android, Playstation 4, Xbox One nebo Raspberry Pi. Oproti svému originálu nabízí MonoGame i další rozšíření.

Počátky MonoGame sahají do roku 2009, kdy José Antonio Leal de Farias začal pracovat na open source projektu XNA Touch. Cílem projektu bylo umožnit portování jednoduchých 2D her vytvořených v XNA na další mobilní platformy mimo Windows Phone. Ke konci téhož roku vychází první verze XNA Touch, která podporuje platformu iOS. V roce 2011 byl projekt přejmenován na MonoGame a byla přidána podpora pro Android, OS X, Linux a OpenGl vykreslování ve Windows. Na začátku roku 2012 přibyla podpora Direct X 11 a Windows 8. Ta umožnila vydávat hry z XNA na oficiálním obchodu Microsoftu, Windows Store. V roce 2013

využilo MonoGame samotné Microsoft Studios pro vydání několika her na Windows 8 a Windows Phone 8. V roce 2015 byla vydána verze MonoGame 3.4 obsahující podporu univerzálních aplikací pro Windows 10 [35].

4 Implementace

4.1 Volba technologie

Jazykem pro vývoj enginu byl zvolen C#. Hlavním důvodem pro výběr tohoto jazyka je, že se jedná o rapid application development jazyk [36]. Což obecně znamená, že vývoj aplikací v tomto jazyku by měl být rychlejší, než například v jazyku C. Je to způsobeno především těmito vlastnostmi:

- Množství předem zabudovaných softwarových komponent (práce se soubory, sítí, serializací, souborovými formáty, ...)
- Nástroje pro diagnostiku, testování a ladění
- Automatická správa paměti (garbage collector)

Pro práci s grafikou, audiem a vstupy byl zvolen XNA framework (3.4). Hlavními důvody jsou rychlost a funkčnost. Další nezanedbatelnou výhodou je snadný budoucí přechod na multiplatformní framework MonoGame (3.4.1), který je oproti XNA stále v aktivním vývoji.

Mimo tyto základní stavební prvky využívá engine i několik dalších knihoven třetí strany.

4.1.1 Farseer

Farseer je engine pro simulaci fyziky ve 2D prostoru [37]. Je založen na fyzikálním enginu Box2D, což je jeden z nejpoužívanější 2D fyzikálních enginů v jazyku C++. Aktuální verze enginu Farseer je 3.5 a pochází z roku 2013. Což bohužel naznačuje, že engine již není v aktivním vývoji. Nicméně jeho zdrojové kódy jsou volně dostupné.

Mezi hlavní rysy enginu patří:

- Fyzikální simulace těles různých tvarů (podporovány jsou konvexní tvary; konkávní tvary se musí rozdělit na kolekci konvexních tvarů - tuto operaci engine umožňuje provádět automaticky pomocí zabudovaných nástrojů)
- Definice rozličných fyzikálních parametrů (hmotnost, odrazivost, odpor tření, ...)
- Optimalizace (uspávání neaktivních těles)
- Definice fyzikálních spojů (joints)
- Fyzikální controllery (gravitační, silové)

Engine rovněž podporuje framework MonoGame (3.4.1).

4.1.2 XNA Game Console

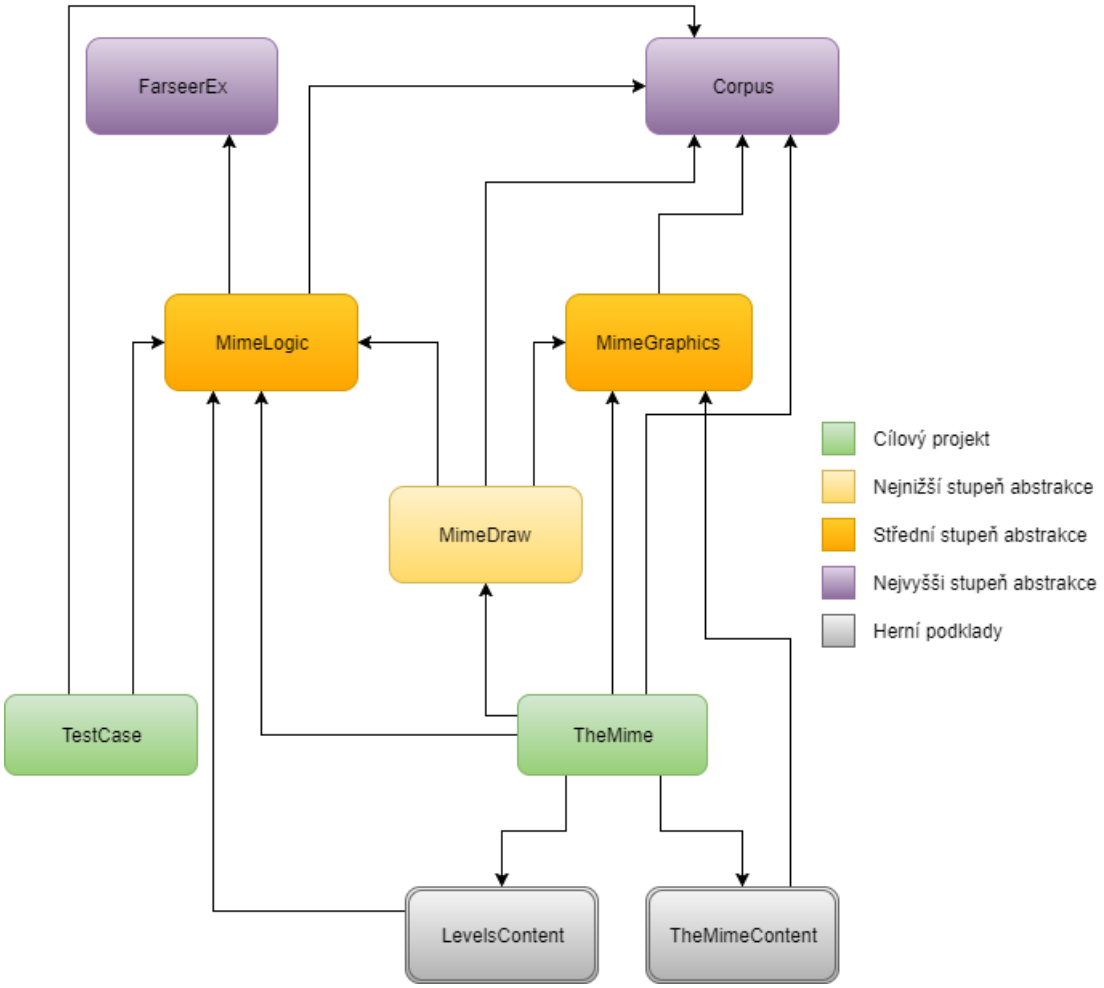
XNA Game Console je open source projekt poskytující jednoduchou ladící konzoly do aplikací postavených na XNA frameworku [38]. Umožňuje v běžící aplikaci zadávat textové příkazy, které mohou jak vypisovat ladící informace, tak i ovlivňovat vnitřní stav aplikace.

Tato knihovna umožňuje snadné definování vlastních příkazů, podporuje našeptávání a její začlenění je velmi snadné i pro již existující projekty.

4.2 Struktura projektu

Herní projekt se skládá z těchto C# projektů: **TheMime**, **TestCase**, **MimeDraw**, **MimeLogic**, **MimeGraphics**, **FarseerEx**, **Corpus** a dvou dalších projektů s herním obsahem **LevelsContent** a **TheMimeContent**. Kromě již zmíněných jsou referencovány ještě C# projekty třetích stran Farseer (4.1.1) a XNA Game Console (4.1.2).

Závislosti mezi jednotlivými projekty jsou zaznačeny na obrázku č. 10. Jejich popis je v tabulce č. 1.



Obrázek 10: Diagram závislostí C# projektů.

Tabulka 1: Popis C# projektů

Název	Popis
TheMime	Samotná hra, která staví na vyvíjeném enginu, jehož funkcionalitu využívá. Jedná se o velmi malý projekt, jehož nejdůležitějším účelem je inicializace herního enginu a načtení výchozí úrovně.
TestCase	Samostatná konzolová aplikace pro testování herní logiky.
MimeLogic	Kompletní logika hry a herního světa. Obsahuje funkcionalitu pro práci s GUI, herními proměnnými, vstupy (klávesnice, myš, gamepad, ...) a především třídy tvořící herní svět (viz. 4.5).
MimeGraphics	Poskytuje funkcionalitu potřebnou k vykreslování herního světa a grafiky. Obsahuje grafická primitiva, funkcionalitu pro podporu postprocessing efektů, debugování a další (viz. 4.3).
MimeDraw	Spojuje logickou reprezentaci světa (**MimeLogic**) s funkcionalitou k jeho vykreslování (**MimeGraphics**). Zajišťuje vykreslování herního světa včetně efektů, jako jsou různé postprocessing shadery nebo nasvětlení scény.
FarseerEx	Rozšíření fyzikálního enginu Farseer (4.1.1) o další obecně použitelnou funkcionalitu (užitečná nejen v rámci tohoto enginu).
Corpus	Obecná knihovna poskytující řadu užitečné funkcionality a rozšíření pro C# i XNA aplikace. Obsahuje systém pro ukládání pozic ve hře, podporu formátovaného textu v XNA, logování chyb, debugovací nástroje, užitečné matematické funkce a další (viz. 4.2.1).
TheMimeContent	Tento projekt obsahuje herní podklady, konkrétně hudbu, zvuky, textury, fonty a podobné. XNA konvertuje tyto materiály do vlastních typů, kompletní kompilace tohoto projektu může trvat na základě obsažených souborů velmi dlouho (v řádu minut).
LevelsContent	V tomto projektu jsou také herní podklady, jedná se především o herní úrovně. Ty nejsou umístěny v **TheMimeContent** z důvodu rychlejšího vývoje. Po každé změně v **MimeLogic** je potřeba projekt **LevelsContent** kompletně přeložit. Pokud by zde byly i grafické podklady, trval by překlad mnohonásobně déle.

4.2.1 Corpus

C# projekt **Corpus** je obecná knihovna poskytující řadu užitečných funkcí. Tyto funkce nejsou svázány pouze s vyvíjeným enginem, ale jsou znovupoužitelné i v jiných projektech.

V následujících kapitolách je popsána nejdůležitější funkcionalita této knihovny.

4.2.1.1 Systém pro ukládání a načítání pozic

Většina současných her umožňuje ukládat hráčův stav a postup (pozice) v herním světě. Díky tomu může hráč odejít od rozehrané hry bez ztráty svého postupu. Při ukládání pozice je třeba uložit především:

- **Stav herního světa**: Dynamické vlastnosti (pozice, rotace, hybnost, ...) každého elementu v herním světě. V zásadě se jedná o dopady všech hráčových akcí na herní svět.
- **Stav hráče**: Stav hráčovy postavy. Jedná se typicky o počet životů, obsah inventáře a podobně.

Pro ukládání pozic ve hře byl zvolen souborový formát XML. Je to z důvodu, že v jazyce C# existuje pro práci s XML soubory velmi dobrá podpora, realizována třídou *XDocument*. Tato třída je velmi obecná a multifunkční. Pro jednodušší práci s XML soubory během ukládání a načítání hráčova postupu byla třída *XDocument* zapouzdřena do vlastní třídy ***SaveGameFile***.

Třída ***SaveGameFile*** je stavová a vždy se nachází právě v jednom XML uzlu. Všechny operace jsou poté prováděny relativně k tomuto uzlu. Operace jsou realizovány metodami třídy a lze je rozdělit do dvou skupin. Operace pro procházení stromovou strukturou souboru (zdrojový kód č. 1) a operace pro zapisování a čtení hodnot (zdrojový kód č. 2).

```
1  // Write start of new element and go to its scope.
2  public void WriteStartElement(string name)
3
4  // Exit actual element scope and go to its parent.
5  public void WriteEndElement()
6
7  // Find children element by name and move to its scope.
8  public bool ReadStartElement(string name)
9
10 // Move to first children elements scope.
11 public bool ReadFirstElementStart()
12
13 // Go to actual element parents scope.
14 public void ReadEndElement()
15
16 // Move to actual element sibling.
17 public bool ReadEndAndStartSibling()
```

Zdrojový kód 1: SaveGameFile - operace pro procházení stromovou strukturou souboru

Díky tomuto zapouzdření je ukládání a načítání hry velmi snadné.

```
// write operations
public void WriteProperty(string name, object value)
public void WriteProperty(string name, Vector2 value)
public void WriteProperty(string name, float value)
public void WriteProperty(string name, bool value)

// read operations
public string ReadPropertyAsText(string name)
public string ReadPropertyAsText(string name, string defaultValue)
public int ReadPropertyAsInt(string name)
public int ReadPropertyAsInt(string name, int defaultValue)
public bool ReadPropertyAsBoolean(string name)
public bool ReadPropertyAsBoolean(string name, bool defaultValue)
public Vector2 ReadPropertyAsVector2(string name, Vector2 defaultValue)
public float? ReadPropertyAsFloat(string name, float? defaultValue)
```

Zdrojový kód 2: SaveGameFile - operace pro zapisování a čtení hodnot

4.2.1.2 Systém pro práci s texty

Cílem bylo vytvořit systém, jenž by umožňoval vykreslování textů do textur. Je to z důvodu, že většina textu ve hře je relativně statická a není třeba je sestavovat při každém vykreslení scény (třeba 30krát za vteřinu).

Dále byl kladen důraz na stylování textů. V jednom textovém řetězci se mohou vyskytovat texty různých fontů, barev, různě zarovnané a podobně.

XNA již obsahuje podporu vykreslování textů. Ta je zakomponována do třídy *SpriteBatch* (viz. kapitola 3.4). Konkrétně se jedná o množinu metod zobrazených ve zdrojovém kódu č. 3. XNA si interně převádí fonty do textur, ze kterých potom dle potřeby vykresluje jednotlivé znaky a vytváří tak souvislý text.

```
public void SpriteBatch.DrawString (SpriteFont, String, Vector2, Color)
public void SpriteBatch.DrawString (SpriteFont, String, Vector2, Color,
    Single, Vector2, Single, SpriteEffects, Single)
public void SpriteBatch.DrawString (SpriteFont, String, Vector2, Color,
    Single, Vector2, Vector2, SpriteEffects, Single)
public void SpriteBatch.DrawString (SpriteFont, StringBuilder, Vector2,
    Color)
public void SpriteBatch.DrawString (SpriteFont, StringBuilder, Vector2,
    Color, Single, Vector2, Single, SpriteEffects, Single)
public void SpriteBatch.DrawString (SpriteFont, StringBuilder, Vector2,
    Color, Single, Vector2, Vector2, SpriteEffects, Single)
```

Zdrojový kód 3: SpriteBatch - metody pro vykreslení textu

SpriteBatch je kvalitním základem pro systém práce s texty. Proto byl zapouzdřen do vlastní třídy ***TextTextureBuilder***. Veřejné rozhraní této třídy pro vykreslování textu je ve zdrojovém kódu č. 4.

Je zde patrný rozdíl mezi přístupem třídy *SpriteBatch* a třídy *TextTextureBuilder*. Druhá jmenovaná má mnohem méně nastavení vizuálních vlastností textu. Chybí například barva nebo efekty. Je to z toho důvodu, že metodám třídy *TextTextureBuilder* se nepředává na vstupu čistý text, ale text s řídícími informacemi o

```
1 public float RenderText(string text, RenderTarget2D target, int
      maxWidth)
2 public Texture2D CreateText(string text, int maxWidth)
3 public Texture2D CreateText(string text, int maxWidth, int maxHeight)
```

Zdrojový kód 4: TextTextureBuilder - metody pro vykreslení textu

tom, jak má vizuálně vypadat.

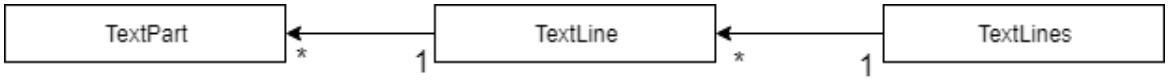
Ukázkový textový řetězec včetně stylování je vidět ve zdrojovém kódu č. 5. Výsledná textura je poté na obrázku č. 11. Jak lze vidět, řídící stylizační informace jsou vždy uvedeny ve složených závorkách.

```
1 {font:Newspaper}With {scale:0.75}Great{scale:1,color:104040}
2   Power{font:Story,n}Comes Great {color:FF0000}Responsibility
```

Zdrojový kód 5: TextTextureBuilder - ukázka zápisu stylovaného textu

Obrázek 11: Text zformátovaný knihovnou TextTextureBuilder

Třída ***TextTextureBuilder*** provádí parsování textu do tříd, které jsou znázorněny v diagramu na obrázku č. 12. Hlavní třídou je *TextLines* reprezentující celý rozparsovaný text. Uvnitř je text reprezentován po řádcích. Je zde tedy kolekce objektů třídy *TextLine* (jeden řádek textu). Řádek textu se dále může dělit na několik částí, pokud obsahuje několik způsobů formátování. Tyto části reprezentuje třída *TextPart*.



Obrázek 12: Vazby mezi třídami pro rozparsovaný text

Třída *TextPart* parsuje řídící informace o formátování textu. Rozpoznávané tagy jsou popsané v tabulce č. 2. Formátování nastavené tagem platí vždy od nastavení až po první změnu nebo konec textu. Zároveň je možno třídu *TextPart* snadno podědit a tagy rozšířit téměř libovolně.

Tabulka 2: Rozpoznávané tagy pro formátování textů

Tag	Popis
color:	Barva textu v hexadecimálním formátu. Lze předat i s nastavením transparentnosti. Příklad: *#50AAFFBB*.
font:	Název fontu, který bude pro text použitý. Třída *TextTextureBuilder* má nastavenou cestu k adresáři, v němž fonty vyhledává podle předaného názvu.
n	Nový řádek. Tag bez parametru, používaný pro zalomení textu. Následující text bude na novém řádku.
scale:	Velikost textu v procentech.
paddingBottom:	Odražení textu od spodní hrany. V pixelech.
paddingTop:	Odražení textu od horní hrany. V pixelech.

4.3 Vykreslování

Projekt **MimeGraphics** (viz.4.2) obsahuje základní třídy určeny k vykreslování herní grafiky.

Jak již bylo zmíněno v kapitole 3.4, XNA nabízí 2 základní přístupy k vykreslování grafiky. V enginu je využíván především nízkoúrovňový přístup, který nabízí mnohem bohatší možnosti práce s grafikou. Základem tohoto přístupu jsou HLSL shadery a pole indexovaných primitiv, vertexů. Shader se skládá z vertex shaderu (transformace vertexů) a pixel shaderu (vykreslování jednotlivých fragmentů, většinou pixelů, na obrazovku nebo do textury). Ukázka C# kódu, jenž vykresluje pole vertexů, za použití efektu, je vidět ve zdrojovém kódu č.6 [39].

```
1  foreach (EffectPass pass in effect.CurrentTechnique.Passes)
2  {
3   pass.Apply();
4
5   GraphicsDevice.DrawUserIndexedPrimitives<VertexPositionColor>(
6      PrimitiveType.TriangleList,
7      vertices,
8      0,  // vertex buffer offset to add to each element of the
9          // index buffer
10     24, // number of vertices to draw
11     indices,
12     0,  // first index element to read
13     12  // number of primitives to draw
14    );
15 }
```

Zdrojový kód 6: Vykreslení skupiny vertexů na scénu

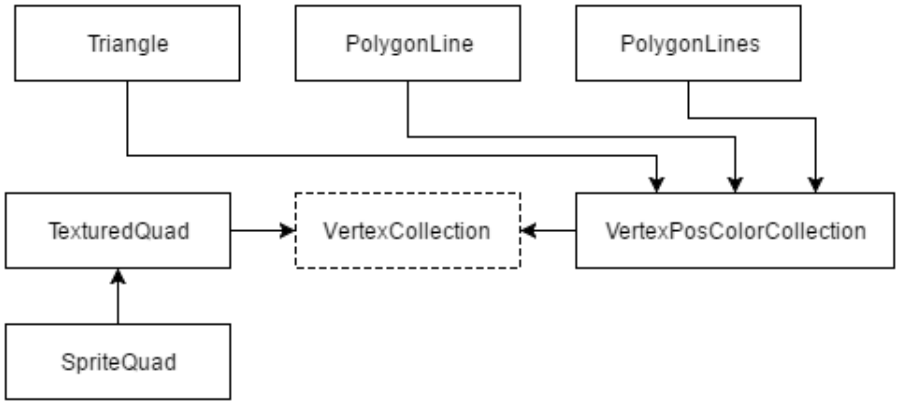
Spouštění shaderů, a tím pádem vykreslování, má na starosti metoda *DrawUserIndexedPrimitives*. Před každým voláním této metody je aplikován jeden průchod efektu (HLSL shaderu). Efekty se mohou skládat z několika průchodů pro dosažení

sofistikovaných výsledků. Samotné metodě *DrawUserIndexedPrimitives* je předáváno pole vertexů (základní primitivum, jeden vertex je jeden bod v prostoru [40]) a jejich pořadí, v jakém se budou zpracovávat. Metoda může zpracovávat vertexy buď jako trojúhelníky nebo úsečky, toto určuje *PrimitiveType*. Samotný vertex může nést více informací, než pouhou pozici, jedná se například o normálový vektor, barvu, mapování textury a podobně.

Pro uložení kolekce vertexů existuje v enginu abstraktní třída ***VertexCollection***. Tato třída má v sobě jak pole vertexů, tak pole indexů určující jejich pořadí. Dále umožňuje provádět s vertexy tyto operace:

- Operace přiřazení celé ***VertexCollection*** (pro snadné klonování objektu)
- Aplikace rotace nebo posunutí
- Změna orientace výsledného primitiva (toto je podstatné pro plochu trojúhelníku, kterou lze vidět jen z jedné strany), čehož je dosaženo změnou pořadí v poli indexů

Z abstraktní třídy ***VertexCollection*** jsou poděděné další třídy, které již většinou reprezentují některá specifická primitiva (úsečky, trojúhelníky) nebo tvary složené z těchto primitiv (quady). Všechny třídy vycházející z ***VertexCollection*** je možné vidět na obrázku č. 13. Jejich popis je v tabulce č. 3.



Obrázek 13: Třídy poděděné z VertexCollection

Tato základní primitiva se dále shlukují do skupin dle užití. Třídy reprezentující tyto skupiny jsou poděděné z abstraktní generické třídy ***Primitives<T>***, jak lze vidět na obrázku č. 14. Jako generikum T se používají třídy poděděné z ***VertexCollection***. Jednotlivé kolekce primitiv jsou popsány v tabulce č. 4.

4.3.1 TexturedQuad

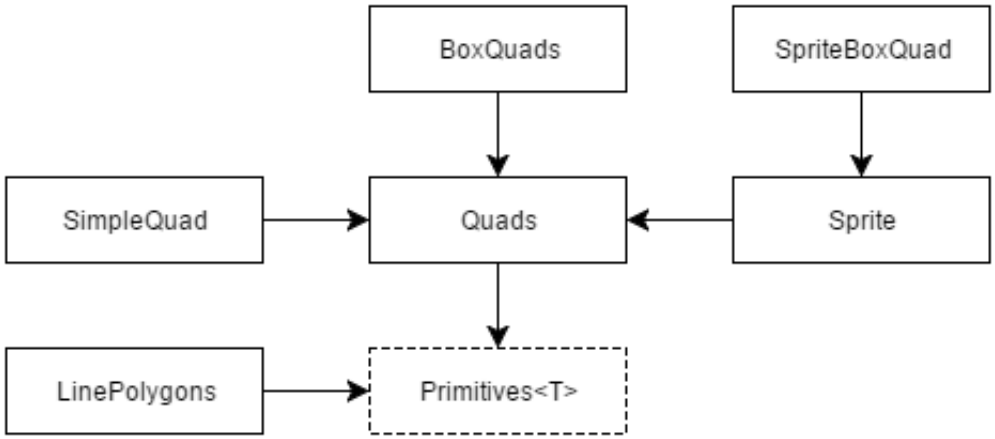
Třída ***TexturedQuad*** reprezentuje nejdůležitější používané grafické primitivum, quad. Ten se používá k vykreslování většiny textur na scéně.

Quad je tvořen 4 vertexy, které vytváří 2 trojúhelníky, ty dohromady vytváří obdélník. Toto schéma lze vidět na obrázku č. 15, kde jsou vertexy označeny čísly 0 až 3. Jeden trojúhelník je tvořen vertexy 0, 1 a 2, druhý trojúhelník vertexy 1, 2 a 3.

Kromě sestavení quadu má třída ***TexturedQuad*** další rozšířenou funkcionalitu:

Tabulka 3: Popis kolekcí vertexů

Název	Popis
VertexCollection	Abstraktní předek popsaný v kapitole 4.3.
VertexPosColorCollection	Kolekce tvořena z vertexů, které nesou informaci o pozici a barvě.
PolygonLine	Prostá úsečka, v konstruktoru se definuje počáteční a koncový bod.
PolygonLines	Soustava úseček definována několika body, může být uzavřená, čímž se vytvoří polygon.
Triangle	Trojúhelník tvořen třemi body. Oproti předchozím nabízí toto primitivum možnost vykreslení celistvé plochy.
TexturedQuad	Vertexy v této kolekci nesou informace o pozici, normálovém vektoru a souřadnici textury. Vertexy jsou organizované do quadu, tato třída je blíže popsána v kapitole 4.3.1.
SpriteQuad	Rozšíření **TexturedQuad** o funkcionalitu sloužící k vykreslování spritů. Podrobnosti jsou v kapitole 4.3.2.



Obrázek 14: Třídy poděděné z Primitives<T>

- Tiling (dláždění) - schopnost vykreslovat na jeden quad jednu texturu opakovaně vedle sebe (horizontálně i vertikálně)
- Clipping (ořezávání) - schopnost oříznout quad včetně textury z jedné nebo více stran

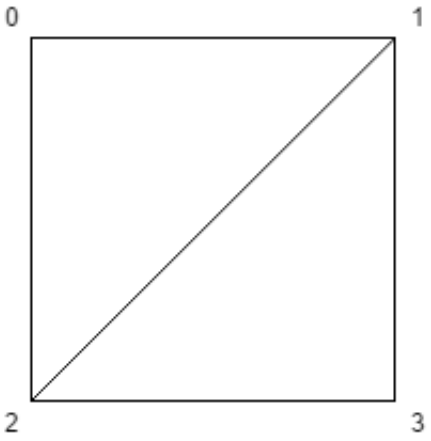
Kromě samotné definice quadu obsahuje třída i cesty k texturám i samotné načtené textury, které se používají k vykreslování daného quadu.

4.3.2 SpriteQuad

Technologie spritů je jedna ze základních společných technik převážné většiny 2D her. Sprity se používají ve hrách již desítky let, příkladem je hra *Super Mario Bros* z roku 1985.

Tabulka 4: Popis kolekcí primitiv

Název	Popis
LinePolygons	Kolekce polygonů tvořených instancemi třídy ***VertexPosColorCollection***. Používané především pro vykreslování ladících informací.
Quads	Kolekce quadů (instance třídy ***TexturedQuad***, nebo poděděných). Používá se především k obejití limitu velikosti jedné textury 2048 x 2048 pixelů (viz. 3.4). V takovém případě jsou jednotlivé quady organizovány vedle sebe, čímž vytváří velký obdélník. Některé metody třídy s tímto rozložení počítají (například clipping).
SimpleQuad	Rozšíření třídy ***Quads*** o funkcionalitu ke snazšímu zacházení a přístupu ke kolekci s pouze jedním quadem.
Sprite	Kolekce s jedním quadem typu ***SpriteQuad***. Třída je připravena k automatické serializaci a obsahuje metody pro nastavení zobrazovaného quadu (celočíselně i v desetinných číslech), jeho velikosti apod.
BoxQuads	Mapování quadů do podoby krychle. Obsahuje funkcionalitu k nastavení velikosti krychle, sklonu a zanoření jednotlivých stran do sebe. Tato funkcionalita je obsažena v samostatné třídě, aby byla znovupoužitelná.
SpriteBoxQuads	Obdobné jako ***BoxQuads***, strany krychle jsou ale sprity.



Obrázek 15: Reprezentace quadu

Sprit je textura obsahující více grafických podkladů, obrázků. Má 2 hlavní využití:

- Paměťová optimalizace - v minulosti grafické karty vyžadovaly textury, jejichž délky stran jsou násobky dvou a pracovali s tím i optimalizace některých procesů. Proto se někdy vyplatí spojit několik textur do jedné a vyplnit tak co největší prostor. Moderní grafické karty již toto omezení nemají.
- Animace - sprity je možné snadno využít na frame-by-frame animace, kdy celá animace může být v jedné textuře, namísto několika samostatných souborů.

Od grafického enginu se vyžaduje schopnost zpracovávání spritových textur. **Vyvíjený engine využívá sprity především jako prostředek pro ukládání animací**.

Jak vypadá takový sprit je možné vidět na obrázku č. 16. Tato jedna textura obsahuje 8 obrázků (2 řádky, 4 sloupce).

Obrázek 16: Ukázka spritu

Vyvíjený engine má pro kreslení spritů speciální HLSL shader, který na vstupu

dostává texturu spritu. Princip tkví v tom, že namísto celé textury shader mapuje jen část této textury na quad (nebo jiné primitivum), jež vykresluje.

Základní pixelshader pro každý vykreslovaný fragment (pixel) vrátí barvu z nějaké souřadnice ve vstupní textuře. Souřadnice ve vstupní textuře určují souřadnice určené pro mapování textur předané společně s jednotlivými vertexy. Pixelshader pro sprity souřadnici na vstupní textuře přepočítává za pomocí **lineární funkce**, tento princip lze vidět ve zdrojovém kódu č. 7. Parametry qX, tX, qY a tY jsou vypočítávány na procesoru a předávány shaderu.

```
1 float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0
2 {
3   float4 texCoord=input.TextureCoordinate;
4   texCoord.x=texCoord.x*qX+tX;
5   texCoord.y=texCoord.y*qY+tY;
6   return = tex2D(TextureSampler, texCoord);
7 }
```

Zdrojový kód 7: Zjednodušený pixelshader pro sprity

Pro výpočet parametrů qX, tX, qY a tY potřebuje engine znát některé informace o spritu. Pro zjednodušení je stanoveno omezení, že všechny obrázky v jednom spritu musí mít shodnou velikost. Jeden sprite se může skládat z více textur (takto je možné obejít omezení na maximální velikost textury 2048 x 2048 pixelů, viz. 3.4). Pro každou texturu potřebuje engine znát následující informace, na základě kterých je schopný vypočíst požadované parametry:

- Počáteční a koncový frame spritové animace obsažené v textuře
- Počet obrázků ve sloupcích a řádcích spritové textury

Tato definice spritu je uložena v XML formátu, jehož ukázka je ve zdrojovém kódu č. 8. Pro ukládání a načítání těchto XML souborů se využívá serializace zabudovaná v XNA frameworku.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <XnaContent xmlns:Quads="MimeGraphics.Quads">
3    <Asset Type="Quads:Sprite">
4      <spriteTextures>
5        <Item>
6          <TextureName>levels/general/crank/body</TextureName>
7          <StartFrame>0</StartFrame>
8          <EndFrame>1</EndFrame>
9          <Dimension>2 1</Dimension>
10       </Item>
11     </spriteTextures>
12   </Asset>
13 </XnaContent>
```

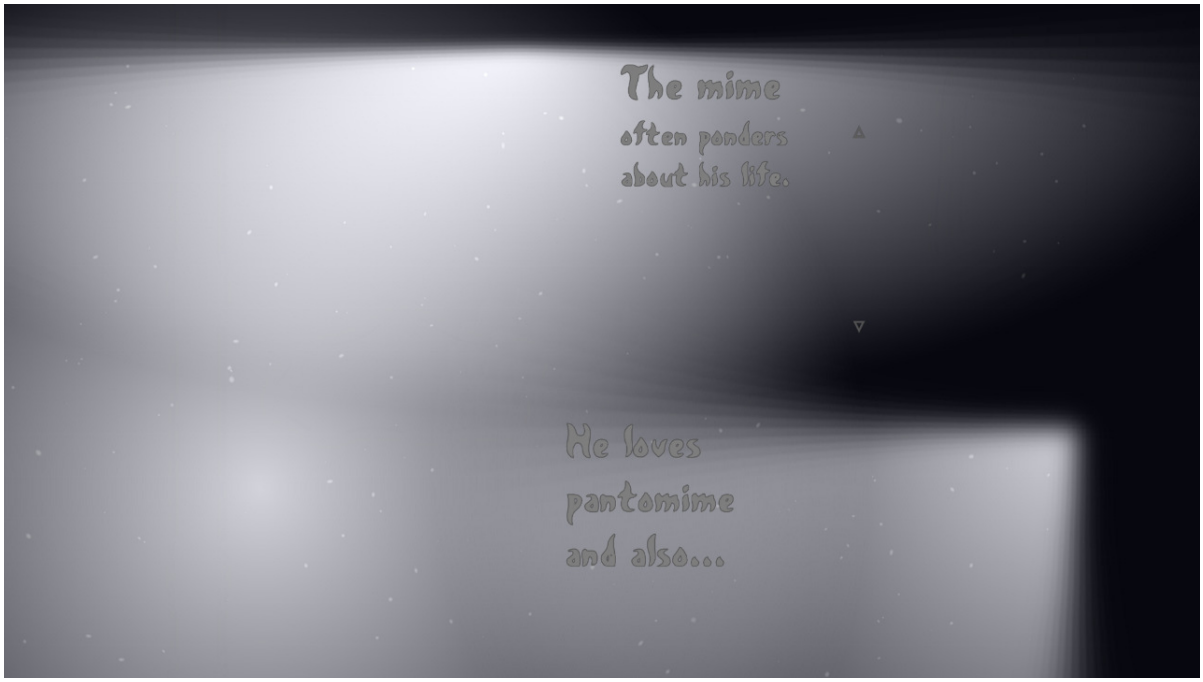
Zdrojový kód 8: XML definice spritu

Logika výpočtu parametrů qX, tX, qY a tY je obsažena ve třídě ***SpriteQuad*** (viz. obrázek č. 13). K přepočtu dochází pouze při změně framu animace, který se má vykreslovat. Tímto je logika kolem spritů výkonově velmi nenáročná.

4.3.3 Osvětlení

Nasvětlení scény je pro vyvíjenou hru důležité, protože pomáhá budovat atmosféru. Postup vykreslování světel je dvoufázový.

V první fází je vytvořena světelná mapa. Toto je textura, do které se postupně, jak se vykresluje celá scéna, vykreslují zdroje světel a stíny které tyto zdroje překrývají. V této fázi nejsou žádná světla vykreslována přímo na scénu. Ukázka takové světelné mapy je vidět na obrázku č. 17. Světlé části jsou osvětlené, tmavé jsou ve tmě.



Obrázek 17: Ukázka světelné mapy

Až ve druhé fázi se tato připravená světelná mapa vykreslí na scénu, ukázka aplikované světelné mapy lze vidět na obrázku č. 18. Pro aplikace světelné mapy se používá speciální color blending. Při vykreslování grafických materiálů nastává prakticky stále situace, kdy jednu texturu vykreslujeme přes již částečně vykreslenou scénu. V takové chvíli je potřeba určit vztah, v jakém se budou ovlivňovat barvy již vykreslených pixelů a nových pixelů. K tomuto právě slouží nastavení color blendingu [41]. Při standardním nastavení je již vykreslená grafika překryta nově vykreslovanými materiály (vstupuje zde ještě samozřejmě alpha kanál). Při aplikaci světelné mapy je ale třeba toto chování změnit. Před aplikací světelné mapy se nastavuje color blending tak, aby světelná mapa upravovala světlost jednotlivých již vykreslených pixelů.

Samotné vykreslování světel do světelné mapy je poměrně náročná operace (1. fáze). Provádí ji speciální HLSL shader se čtyřmi průchody. Dvě světla vykreslená pomocí tohoto shaderu jsou na obrázku č. 19 (jedno volání shaderu vykreslí vždy jedno světlo). Jak lze vidět, světla mají kruhový případně oválný tvar a je možné

Obrázek 18: Ukázka aplikované světelné mapy

nastavit určitou míru vykrojení (skrz vstupní parametr). Dále je možné nastavit intenzitu světla a případně i barevný odstín. Velikost světla určuje quad, na jenž shader vykresluje. Zmíněné 4 průchody shaderu provádí za sebou tyto úkony:

1. **Vykreslení základního tvaru světla** - s vykrojením nebo bez něj. V této fázi má světlo jednolitou barvu.

2. **Horizontální rozostření** - světla nemají jasné okraje, proto je třeba je rozmazat. Pro tento účel se používá modifikované Gaussovo rozostření [42]. Čím dále jsme od středu světla, ohniska, tím větší rozostření je aplikováno.

3. **Vertikální rozostření** - analogicky jako předchozí krok. Rozostření je velmi náročná operace a je ji potřeba rozdělit do dvou průchodů, v opačném případě se naráží na limity shaderů (omezení počtu instrukcí v jednom průchodu).

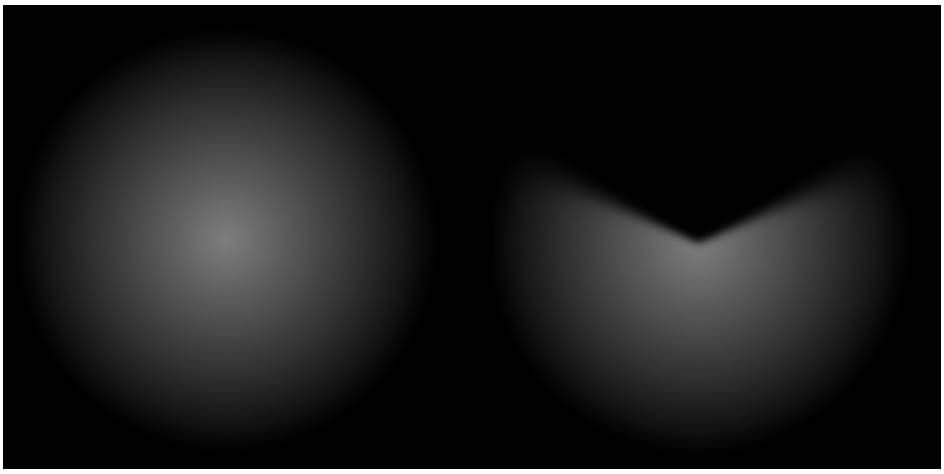
4. **Úprava intenzity světla** - v této fázi se snižuje intenzita světla se zvyšující se vzdáleností od ohniska.

Protože se jedná o velmi náročnou operaci, je každé světlo vykresleno v enginu pouze jedenkrát a to do textury. Poté je již opětovně kresleno z této textury, což je v porovnání s kompletním vykreslení světla velmi nenáročná operace.

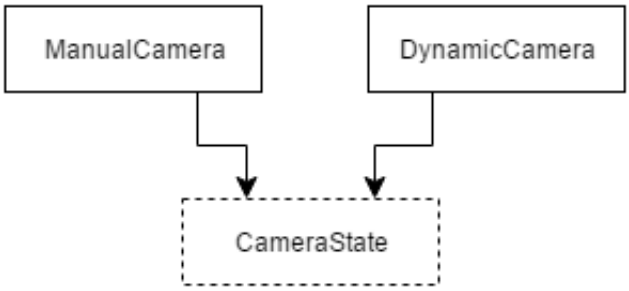
4.4 Kamera

Virtuální kamera ve hrách slouží ke stejnému účelu jako skutečné kamery, dívá se skrz ně na herní svět. Optimální kamera by měla mít vždy v záběru hráčovu postavu a aktuálně podstatnou část herního světa [43].

Funkcionalita virtuálních kamer je umístěna v projektu **MimeLogic** (viz.4.2) ve třech třídách znázorněných na obrázku č. 20.



Obrázek 19: Ukázka vykreslených světel



Obrázek 20: Organizace tříd virtuálních kamer

Abstraktní třída ***CameraState*** v sobě nese základní funkcionalitu. Obsahuje tyto základní parametry:

- Pozice kamery v prostoru (2D souřadnice X a Y)
- Vzdálenost kamery od scény (souřadnice Z) - kamera je vždy umístěna v kladné Z souřadnici a dívá se směrem na souřadnici 0

Z těchto parametrů počítá třída projekční a pohledovou transformační matici. Právě tyto 2 matice se předávají grafické kartě a shadery podle nich transformují scénu dle nastavení kamery.

Třída ***CameraState*** dále umožňuje nastavit hranice kamery, což je oblast, ve které se má kamera pohybovat. Existují 2 typy hranic, tvrdé a měkké. Tvrdé není možné nikdy porušit. Oproti tomu měkké hranice je možné krátkodobě porušit, pokud to vylepší plynulost pohybu kamery.

Poslední důležitou funkcí této třídy je výpočet oblasti, která je aktuálně kamerou vidět. Toto slouží především k optimalizaci hry. Grafické prvky, jenž jsou umístěny mimo viditelnou část scény, nejsou totiž vůbec vykreslovány. Základem tohoto výpočtu je vrhnutí paprsku (ray-casting) do levého horního okraje scény. Díky 2D charakteru scény je poté dopočet celé viditelné obdélníkové oblasti triviální.

Třída ***ManualCamera*** rozšiřuje třídu ***CameraState*** jen velmi jemně. Tato kamera nemá žádnou obsáhlou vnitřní logiku ani dynamiku pohybu. Jak se nastaví, tak okamžitě funguje. Používá se především v editoru úrovní.

Oproti tomu je třída ***DynamicCamera*** velmi obsáhlá. Tato kamera se používá v samotné hře a má tuto funkcionalitu:

- **Dynamické pohyby** - Při změně nastavení kamery (pozice, vzdálenost od scény, hranice) se tato změna neprojeví hned, ale postupně v čase. Rychlost transformace je nastavitelná, stejně jako způsob transformace. Kamera rozeznává dva způsoby:
  - **Lineární** - Rychlost transformace je spočítána podle delty a požadované doby transformace. Poté se rychlost již nemění a v každém cyklu se aplikuje až do dokončení transformace.
  - **SmoothDamp** - Využívá se zde algoritmus chovající se podobně jako simulace pružinových tlumičů [44]. Transformace je velmi hladká a přirozená.
- **Následování hráče** - Pokud se kameře explicitně nenastaví pozice, následuje ve svém výchozím nastavení postavu hráče. Hráč má určitý prostor, ve kterém se může volně pohybovat, pokud jej opustí, kamera ho následuje [45]. Dále kamera pracuje s natočením postavy hráče (doprava/doleva) a ukazuje standardně větší část scény ve směru, kam se hráč dívá.
- **Fronta nastavení** - Často se stává, že se setkají protichůdné požadavky na nastavení kamery, jak hráč prochází herními úrovněmi. Pro tento případ jsou všechna nastavení uchovávána v kolekci typu fronta a aplikuje se vždy první z nich. Jak se postupně nastavení ve frontě deaktivují (například když hráč opustí určitou oblast), tak se obsah fronty mění.

4.5 Organizace herního světa

Způsob uchopení herního světa je jedna ze základních přidaných hodnot herního enginu (viz. 3.2). Jedná se o způsob, jakým jsou vnitřně uspořádány prvky tvořící herní svět, jaké budou mít možnosti navzájem se ovlivňovat a jak bude řešeno jejich rozšiřování o novou funkcionalitu.

Pro elementy, ze kterých se skládá herní scéna, bylo zvoleno stromové uspořádání. Tento přístup usnadňuje organizaci scény a umožňuje implementovat i určité výkonnostní optimalizace.

Abstraktním předkem pro každý element na herní scéně je třída ***Element***. Tato třída obsahuje rozhraní, proměnné a kód společný všem elementům. Mezi hlavní funkcionalitu, kterou tato třída poskytuje, patří:

- Poskytování názvu - každý element má unikátní název sloužící jako identifikátor
- Poskytování absolutní pozice, rotace a měřítka elementu (i negrafické elementy mají tyto vlastnosti z důvodu vykreslování a manipulace v ladícím módu)
- Poskytování *WorldMatrix* pro renderování - transformační matice, která v sobě obsahuje pozici, rotaci a měřítko elementu
- Poskytování hloubky elementu na scéně - elementy se vykreslují v pořadí podle této hodnoty od nejnižší hloubky po nejvyšší
- Poskytování seřazené kolekce podřazených elementů (potomků) a reference na nadřazený prvek (pokud je) - **toto realizuje stromovou strukturu**

Tabulka 5: Výkonnostní testy ElementsCollection

	Řazení s binárním vyhledáváním	Řazení s lineárním vyhledáváním
Vložení 1 000 elementů	1 ms	17 ms
Vložení 50 000 elementů	580 ms	74 303 ms

- Funkcionalita specifická pro editor úrovní a ladění (diagnostické zobrazení elementů, ...)
- Funkcionalita specifická pro hru jako takovou (verze herního světa, viz. 2.2)
- Funkcionalita pro ukládání a načítání stavu elementu (ukládání/načítání rozehrané hry)
- Virtuální metoda *Update* volaná v každém herním cyklu, slouží k implementaci kontinuální vnitřní logiky elementu

Pro udržování kolekce elementů existuje třída ***ElementsCollection***. Primární funkcí této třídy je zajištění pořadí elementů v ní. To je zajištěno při vkládání elementů, které jsou vždy zařazeny na správnou pozici (index). Tato pozice je určena pomocí binárního vyhledávání (je možno realizovat, protože pole je v každém okamžiku seřazeno). Implementací binárního vyhledávaní získala kolekce skvělou výkonnostní charakteristiku, jak lze vidět v tabulce č. 5.

Se změnou hloubky již zařazených elementů současná verze enginu nepočítá, tato operace není podporována.

Všechny elementy jsou zařazeny v nějaké instanci třídy ***ElementsCollection*** a každý element má své podřízené elementy ve své vlastní instanci třídy ***ElementsCollection***. Tímto způsobem je implementována stromová struktura.

Jak již bylo řečeno, ze třídy ***Element*** dědí všechny objekty v herní scéně. Ty se dají rozdělit do tří hlavních kategorií.

- Elementy zajišťující grafickou reprezentaci (vykreslení obrázku, spritu, efektů, ...), viz. 4.7
- Elementy reprezentující fyzikální tělesa (polygony, kruhy, provazy, ...), viz. 4.8
- Element bez grafické reprezentaci i fyzikálního tělesa (různé logické vazby mezi tělesy, knihovny textů, animátory, ...), viz. 4.9

Tyto kategorie se dají dále kombinovat díky stromové struktuře (například fyzikálnímu tělesu je možné přiřadit podřízený element, jenž bude zajišťovat grafickou reprezentaci).

Samotný herní svět je zapouzdřený ve třídě ***World***. Tato třída obsahuje kolekci typu ***ElementsCollection*** s kořenovými elementy (elementy nemající nadřazený element). Jedna instance třídy ***World*** je jedna herní úroveň (herní svět). Třída obsahuje:

- **Content manager** - Content manager (manažer obsahu) je třída mající na starosti načítání herních materiálu (HLSL shadery, zvuky, textury, ...). Třída

rovněž cachuje již načtené objekty, což výrazně optimalizuje načítání stejných materiálů. Herní svět má pro sebe exkluzivní instanci této třídy, skrz kterou načítá veškerý obsah potřebný pro herní úroveň. Content manager je hernímu světu přidělen herním managerem (viz. 4.6).

- **Výchozí kameru** - Instance hlavní kamery nahlížející na scénu. Jedná se o instanci třídy ***DynamicCamera***, které je předána reference na herní svět, ve kterém následuje hráče (viz. 4.4).
- **Fyzikální simulaci světa** - Instance třídy reprezentující fyzikální svět (vit. 4.1.1). Periodicky se volá metoda pro aktualizaci fyzikální simulace. Všechna fyzikální tělesa mají v tomto světě svou reprezentaci.
- **Barvu a intenzitu tmy** - Využíváno při nasvětlování scény (viz. 4.3.3). Barva neosvětlených částí a míra intenzity tmy (zda zcela zakrývá scénu, zčásti, nebo vůbec) je dána parametrem herního světa.
- **Funkcionalitu pro ukládání a načítání stavu světa** - Metody pro ukládání a načítání aktuálního stavu světa. Používá se pro uložení rozehrané hry. Využívá se zde systém pro ukládání a načítání pozic z projektu **Corpus** (viz. 4.2.1.1).
- **Podporu paralaxního scrollingu** - Paralaxní scrollování je metoda, kdy se v závislosti na pohybu kamery pohybují různé vrstvy scény různou rychlostí. To dodává celé scéně pocit hloubky [46]. Jednotlivé paralaxní vrstvy jsou definovány v herním světě. Každá z nich má koeficient přepočtu pohybu proti kameře (zvlášť horizontální a vertikální koeficient). Reprezentace světa přepočítává pozice vrstev pozadí podle těchto koeficientů a aktuální polohy kamery.
- **Optimalizaci vykreslování** - Celá herní scéna obsahuje v zásadě velké množství prvků. Často je třeba vykreslovat jen malý zlomek z nich. Herní svět obsahuje funkcionalitu pro lokalizování elementů, které jsou viditelné na scéně a jejich linearizaci ze stromové struktury do jednoúrovňového seznamu. S takovouto kolekcí se poté pracuje velmi snadno, samotné vykreslování je efektivní a bez rekurze.
- **Řízení postprocessing efektu** - V závěrečné fázi vykreslování je na celou scénu aplikován postprocessing shader. Tento shader je psaný na míru konkrétní hře a skládá se z těchto efektů:
  - Bloom - Efekt simulující chování skutečných kamer. Jeho principem je narušení jasně daných hranic světelných zdrojů a světlých částí scény [47]. Efekt je poměrně náročný, kvůli využití gaussova rozostření vyžaduje několik průchodů grafickou kartou [42].
  - Ztmavení okrajů - Efekt přispívající atmosféře hry. Samotné okraje vykreslované scény jsou vždy velmi tmavé, což tvoří určitý rám kolem scény. Využívá se zde exponenciální funkce.
  - Efekt aktivního světa - Ve hře je možné přepínat mezi dvěmi verzemi herního světa (viz. 2.2), z nichž každá verze má odlišnou vizuální reprezentaci. Tuto odlišnou vizuální reprezentaci má na starosti tento efekt, který mění saturaci scény a přidává zrnění pro simulaci starých televizních obrazovek. Pro optimalizaci je textura zrnění vygenerována pouze

jednou a poté je v každém průchodu posunuta na jinou pozici. To vyvolává dojem náhodného zrnění při velmi nízkých nárocích na výkon.

Herní svět předává tomuto shaderu parametry v závislosti na aktuálním stavu světa.

- **Management vstupních bodů** - Každá herní lokace může mít několik bodů, kterými může hráčova postava do lokace vstoupit (na nichž bude hráčova postava umístěna po nahrátí úrovně). Třída herního světa tyto body shromažďuje v kolekci, kdy každý bod má unikátní index, jímž se identifikuje. Kromě informace o pozici obsahuje každý bod i výchozí natočení herní postavy (doleva/doprava).

4.5.1 Serializace herního světa

Každá úroveň hry, herní svět, je uložena v XML souboru. Tento XML soubor má strukturu rozpoznatelnou vnitřním serializátorem XNA frameworku (intermediate serializer). Načítání struktury světa je kvůli tomu velmi jednoduché, jak lze vidět na zdrojovém kódu č. 9. Tento jednoduchý kód vytvoří kompletní stromovou strukturu světa, všechny elementy včetně jejich nastavení. Nicméně většinou je potřeba ještě načíst další podklady, jako jsou textury apod. (viz. 4.5.2).

Samotný jazyk C# obsahuje také serializer, ten ale není dostačující. Oproti intermediate serializeru z XNA si nedokáže poradit například s kolekcí objektů různých typů.

```
1 MimeLogic.World.World world;
2 world = contentManager.Load<MimeLogic.World.World>(name);
```

Zdrojový kód 9: Deserializace herního světa

XML soubory s definicí herního světa je možné psát ručně, nebo je možno již existující svět (vytvořený například v editoru) nechat serializovat do požadovaného formátu (zdrojový kód č. 10).

```
1 XmlWriterSettings settings = new XmlWriterSettings();
2 settings.Indent = true;
3
4 using (XmlWriter writer = XmlWriter.Create(file, settings))
5 {
6   IntermediateSerializer.Serialize(writer, world, null);
7 }
```

Zdrojový kód 10: Serializace herního světa

Do automatické serializace a deserializace vstupují všechny veřejné proměnné a vlastnosti objektu. To je často žádoucí ovlivnit, k čemuž slouží atributy *ContentSerializer* a *ContentSerializerIgnore* [48]. Tyto atributy lze nastavovat libovolným vlastnostem a proměnným. Je pomocí nich možno ignorovat specifické veřejné vlastnosti nebo serializovat privátní proměnné.

U serializovaných proměnných a vlastností lze také upravit způsob, jakým s nimi bude serializace zacházet. Jedná se například o povolení/zakázání NULL hodnot, změna názvu ve výsledném XML, úprava práce s kolekcemi nebo definování, zda je vlastnost/proměnná povinná či volitelná.

4.5.2 Životní cyklus elementu

Elementy, ze kterých se skládá herní svět, prochází různými fázemi od prvotního vytvoření až po uvolnění. Tento životní cyklus vypadá následovně:

1. **Vytvoření objektu** - Vytvoření samotného objektu elementu. Je běžně prováděno během deserializace (viz. 4.5.1), případně ručně v herním editoru.

2. **Nastavení vlastníka** - Elementu je nastavena reference na herní svět. Tím element rovněž dostane instanci třídy *ContentManager*, skrz kterou si může načíst potřebný herní obsah (shadery, textury, různé struktury, audio apod.).

3. **Deserializace** - Po serializaci elementu mohou být některá data neúplná. Automatická deserializace nedokáže vždy doplnit vše, co je potřeba. V zásadě se jedná například o dohledání dalších elementů na scéně, na než element odkazuje.

4. **Načtení uloženého stavu** (volitelné) - Pokud je herní svět načítán z uložené pozice, je v této fázi obnoven uložený stav elementu. To může obnášet nastavení polohy na scéně, vnitřních proměnných nebo libovolné jiné vlastnosti. Viz. 4.2.1.1

5. **Inicializace** - V tomto kroku se vytváří některé další objekty potřebné pro fungování elementu ve hře. Například entity (viz. 4.8) zde vytváří tělesa fyzikálního enginu. Tímto krokem se některé vlastnosti elementu zamknout k editaci, buď proto, že jejich editace není žádoucí z hlediska výkonu (příliš výkonově náročné, nutná alokace dalších objektů, ...), nebo proto, že není jednoduše možná (např. pozici fyzikálních těles, entit, není možné takto přímo měnit, je potřeba využít jointů). V editoru úrovní se do tohoto stavu elementy nikdy nedostanou.

6. **Vlastní běh** - Běh samotné hry. V každém cyklu je volána metoda elementu *Update*, ve které je implementována vnitřní logika elementů (je-li potřeba).

7. **Uložení aktuálního stavu** (volitelné) - Nastává při manuálním uložením hry nebo při uvolňování herního světa, který má být perzistentní (pokud ho znovu hráč navštíví, je obnoven naposledy uložený stav). Viz. 4.2.1.1

8. **Uvolnění zdrojů** - Probíhá před uvolněním světa. V této fázi se například zastavují přehrávané zvuky, hráčovi se vrací ovládání, bylo-li zamknuto, a podobně.

9. **Uvolnění objektu** - Finální uvolnění objektu provádí automatický garbage collector jazyku C#.

Kromě již zmíněných existuje ještě jedna speciální fáze používaná v editoru úrovní. Jedná se o **Serializaci**. Během této fáze se element připraví na automatickou serializaci, například naplněním některých vnitřních serializovaných proměnných.

Tabulka 6: Popis stavů třídy GameManager

Název	Popis
NoWorld	Žádný herní svět není načten, výchozí stav manageru.
Loading	Probíhá načítání herního světa. Nyní se toto děje synchronně, nicméně tento stav je příprava do budoucna na asynchronní načítání.
Design	Herní svět je načten, ale není inicializován (viz. 4.5.2). V tomto režimu funguje editor úrovní.
Running	Standardní běh hry.
Paused	Zapauzovaný standardní běh hry, použitelné například pro in-game menu. Celý herní svět je dostupný, ale neaktualizuje se, neprobíhá simulace fyziky apod.

4.6 Herní manager

Třída ***GameManager*** je nejdůležitější třídou enginu. Spravuje herní svět, stará se o jeho načítání a uvolňování. Třída rozlišuje několik stavů, ve kterých se herní manager může nacházet. Stavy jsou popsány v tabulce č. 6.

Veřejné rozhranní třídy je velmi prosté (zdrojový kód č. 11). Jsou zde metody na načítání herních světů *LoadWorld* (používá se v editoru úrovní, kde není žádoucí svět inicializovat, viz. 4.5.2) a *LoadAndInitWorld* (používá se v samotné hře). Dále je zde metoda *Update* pro aktualizaci herního světa. V této metodě probíhá nejdůležitější herní logika, simuluje se herní svět a podobně. Na závěr je zde funkctionalita pro správu automaticky uložené pozice. Metoda *AutoSave* vytvoří v předem definovaném umístění soubor s aktuálním stavem hry, tento soubor může být pouze jeden. Metoda *LoadAutoSave* soubor přečte a postará se o kompletní načtení herního světa a obnovení jeho stavu (pokud je stejný herní svět již načtený, je do jisté míry recyklován pro rychlejší načítání).

```
1 // game world related logic
2 void LoadWorld(string name, int? entryPointID = null);
3 void LoadAndInitWorld(string name, int? entryPointID = null);
4 // autosave related logic
5 void LoadAutoSave();
6 void AutoSave();
7 void DeleteAutoSave();
8 // world updating logic
9 void Update(GameTime gameTime);
```

Zdrojový kód 11: Základní veřejné rozhraní třídy GameManager

Třída dále obsahuje funkcionalitu společnou celé herní seanci, jež se přenáší mezi herními světy, úrovněmi. Jedná se o tyto části:

- **Správa hudby** - Instance třídy *MusicPlayer*, skrz kterou je přehrávána hudba na pozadí. Třída využívá přehrávač médií zabudovaný v XNA frameworku (viz. kapitola 3.4).

- **Správa herních nastavení** - Instance třídy *GameSettings* obsahující různé konfigurovatelné vlastnosti hry. Libobolná část enginu má k této instanci přístup a může podle ní upravit své chování.

- **Správa uživatelskcýh vstupů** - Instance třídy *Controls*, v níž jsou nadefinovány jednotlivé ovládací prvky. Ovládací prvky mohou být dvojího typu:

  – Tlačítko - Prvky rozlišující stav zapnuto/vypnuto, patří sem například skok nebo běh.

  – Analog - Prvek chovající se jako analogová páčka na gamepadu. Stavem prvku je v tomto případě 2D vektor, který ukazuje směr a intenzitu. Používá se například pro pohyb hráče.

  Pro každý ovládací prvek jsou nadefinovány skutečné vstupy (tlačítko na klávesnici/gamepadu) i grafická reprezentace pro zobrazení ve hře. Jsou rozlišovány 2 zdroje vstupů, klávesnice a gamepad. Oba mohou fungovat zároveň, ve hře se mění grafické reprezentace ovládacích prvků dle aktuálně používaného.

- **Správa jasu** - Je možno ovlivnit míru jasu vykreslované scény. Tento parametr vstupuje do postprocessing HLSL shaderu, kde ovlivňuje výslednou scénu.

- **Přístup k herní konzoli** - Instance herní konzole pro ladění aplikace, viz. kapitola 4.1.2.

- **Správa ladícího režimu** - Pro účely ladění lze zapnout vykreslování různých ladících informací. Jedná se o:

  – Fyzika - Vykreslování skutečných fyzikálních těles, se kterými pracuje fyzikální engine (viz. kapitola 4.1.1). Znázorní, zda je těleso aktivní, nebo bylo uspáno (optimalizace fyzikálního enginu).

  – Vertexy - Vykreslování samotných vertexů, z nichž se skládají vykreslovaná grafická primitiva (viz. kapitola 4.3).

  – Vazby mezi akcemi - Zobrazení vazeb mezi akcemi (viz. kapitola 4.9.1). Díky těmto informacím lze dobře vidět, jaké elementy se navzájem ovlivňují.

  – Ladící texty - Některé elementy nabízí textové vyjádření svého stavu (například zda nějaký zvukový efekt právě hraje či nehraje). Zde je možno zapnout jejich zobrazování.

  – Oblast grafických elementů - Každý element dědící z *DrawableElement* (viz. kapitola 4.7) má vlastnost *AbsoluteRadius* určující poloměr kruhu, který obsáhne celý element na scéně. Tato informace se využívá k optimalizaci vykreslování. Elementy, jejichž kruh nezasahuje do viditelné oblasti (tuto oblast poskytuje kamera, viz. kapitola 4.4), se nevykreslují.

4.7 Elementy zajišťující grafickou reprezentaci

Tyto elementy implementují interface ***IDrawable***. Interface poskytuje všechny potřebné prostředky pro proces vykreslování.

Tabulka 7: Popis nejdůležitějších interfaces

Název	Popis
IDrawable	Vykreslování elementu na scéně, podpora nasvícení (viz. 4.3.3)
IPositionable	Manipulace s umístěním elementu na scéně
IScaleable	Manipulace s měřítkem elementu na scéně
IInvertable	Překlopení (horizontální i vertikální) objektu na scéně

Dvě nejdůležitější metody tohoto interface jsou vidět ve zdrojovém kódu č. 12. V zobrazených metodách se implementuje samotné vykreslování objektů na scénu. Metoda *Draw* má za úkol samotné vykreslení elementu, zatímco metoda *DrawShadow* vykresluje stín elementu. Stín je jednobarevný jednolitý tvar, který používá systém pro vykreslování světel, například pro zakrytí zdroje světla (viz. 4.3.3). Do těchto dvou metod vstupují jako vstupní parametry 2 transformační matice (projekční a pohledová). Tyto matice poskytuje kamera, skrz kterou je na scénu nahlíženo, viz. 4.4.

Kromě výše zmíněných metod implementuje interface i několik vlastností:

- *Visible* - boolean, element je viditelný na scéně, má se kreslit
- *CastingShadow* - boolean, pokud je element umístěný před světlem, překryje ho
- *AvoidShadow* - boolean, element je vyňat z nasvětlení scény, světlo ani tma se na něj neaplikují
- *Quads* - grafické primitivum reprezentující vykreslovaný element, používá se především pro účely ladění

```
1 void Draw(Matrix projection, Matrix view);
2 void DrawShadow(Matrix projection, Matrix view, Vector4 shadowColor);
```

Zdrojový kód 12: Metody interface IDrawable

Ačkoliv interface ***IDrawable*** může implementovat jakýkoliv element, je zde jeden abstraktní element, již implementující toto rozhraní, z nějž dědí téměř všechny standardní vykreslované objekty. Třída se jmenuje ***DrawableElement*** (viz. 4.7.1) a kromě již zmíněného interface implementuje i interface ***IPositionable***, ***IScaleable*** a ***IInvertable***. Jejich popis je v tabulce č. 7.

Důvodem pro oddělené rozhraní k určitým vlastnosti elementů do samostatných interfaců je usnadnění manipulace s těmito vlastnostmi v editoru úrovní.

4.7.1 Základní grafické elementy

Naprosto základním úkolem grafických entit je prosté vykreslení textury na scénu na zadané souřadnice. K takovému vykreslování slouží abstraktní element ***AbstractImage*** rozšiřující třídu ***DrawableElement*** o specifickou funkcionalitu.

Hlavním rozšířením je práce se specifickou skupinou HLSL shaderů. Převážná většina shaderů používaných ve hře má společné vstupní parametry, jenž získávají includováním HLSL kódu umístěným v souboru *Common.fx*. Tyto parametry lze vidět ve zdrojovém kódu č. 13. Dále shadery includují HLSL kód pro vizuální zobrazení mechaniky přepínání mezi světy (viz. 2.2). Kód je umístěn v souboru *MimeView.fx* a lze jej vidět ve zdrojovém kódu č. 14.

```
#ifndef Common
#define Common

float4x4 World;
float4x4 View;
float4x4 Projection;
// texture of effect
texture EffectTexture, Mask;
bool UseMask;
float2 QuadSize, QuadUpperLeft;

float Alpha;  // transparency
float4 OverColor; // rendering will be in this color (alpha == 0 => no
    overColor)
float Brightness;

#endif
```

Zdrojový kód 13: Společné parametry HLSL shaderů

```
#ifndef MimeView
#define MimeView

#include "Common.fx"

float mimeViewRadius;
float2 mimeViewCenter;

float otherWorldAlpha; // alpha visibility in other world

float rand, intensity;
texture noiseTexture;

#endif
```

Zdrojový kód 14: Parametry HLSL shaderu pro přepínání mezi světy

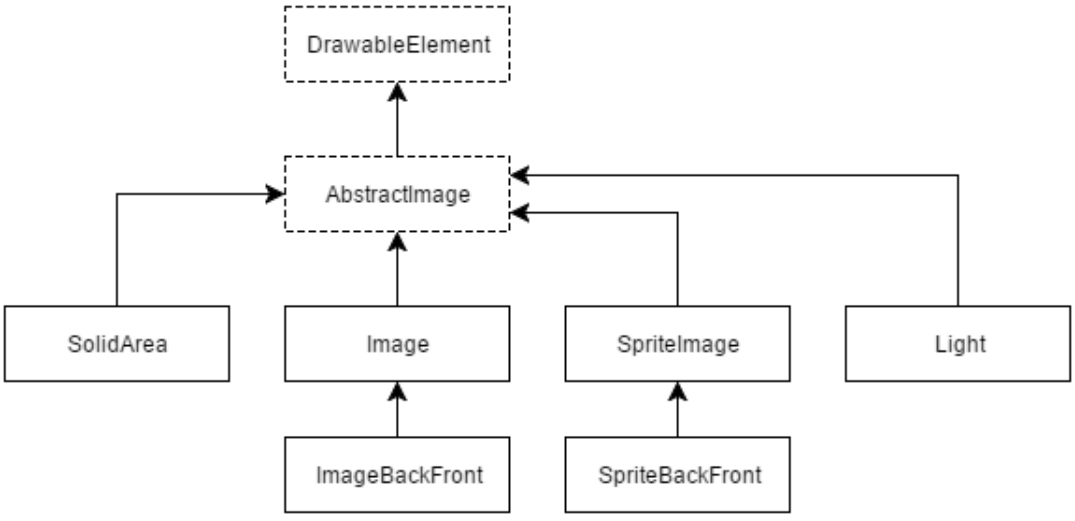
Třída ***AbstractImage*** tedy nastavuje tyto parametry shaderu a její funkcionalita je o to náležitě rozšířena. Třída obsahuje jmenovitě funkcionalitu k nastavování světlosti textury, masky (umožňuje skrýt části textury) a odstínu textury. Dále předává shaderu parametry spojené s přechodovým efektem mezi dvěmi verzemi herního světa (viz. 2.2) a informaci o tom, do kterého světa element patří (podle toho může být vykreslován nebo nemusí).

Tabulka 8: Popis základních grafických elementů

Název	Popis
SolidArea	Jednolitá oblast definována quadem (viz. 4.3.1) a barvou, kterou se vykresluje.
Image	Základní vykreslování textury na scénu. Vykrelsovaná oblast je definována instancí třídy *Quad* (viz. tabulka č. 4).
ImageBackFront	Stejné vlastnosti jako **Image**, nicméně umožňuje vykreslovanou oblast rozdělit vertikálně nebo horizontálně na 2 oblasti s rozdílnou absolutní hloubkou na scéně. Takto může být jedna textura rozdělena na dvě části, z nichž se jedna vykresluje za hráčem a druhá před hráčem. Tímto způsobem se řeší práce s perspektivou nastíněna v kapitole 2.1.
SpriteImage	Element zaštiťující vykreslování spritů, viz. kapitola 4.3.2.
SpriteBackFront	Totožné s **ImageBackFront**, pouze pro sprity.
Light	Vykreslování zdroje světla, viz. kapitola 4.3.3.

Dále třída implementuje metodu *DrawShadow* pro vykreslování stínů pro systém nasvětlení scény (viz. 4.3.3). Je zde využita schopnost překrýt kompletní texturu určitou barvou. Tato schopnost je společná celé skupině shaderů.

Z abstraktní třídy ***AbstractImage*** dědí již konkrétní elementy. Nejvýznamnější z nich jsou vidět na obrázku č. 21. Jejich popis je v tabulce č. 8.



Obrázek 21: Nejdůležitější třídy dědící z AbstractImage

4.7.2 Vykreslování deště

Vykreslování efektu deště je třeba řešit chytře. Kvůli rozsahu tohoto efektu (kompletní obrazovka) je možno při špatné implementaci ztratit velké množství výkonu.

V enginu je efekt implementován způsobem, kdy využivá velkou mírou shadery grafické karty. Shader *Rain.fx* dostává na vstupu kromě standardních parametrů (viz. kapitola 4.7.1) ještě dvě další informace:

- Dešťovou mapu - jedná se o texturu nesoucí informace o každé dešťové kapce na scéně
- Úhel deště - natočení kapek deště, pokud neprší přesně vertikálně

Dešťová mapa je textura o výšce 1 pixel a šířce rovné polovině velikosti backbufferu scény. Texturu vytváří a aktualizuje element ***Rain***, který efekt deště zapouzdřuje. Každý pixel dešťové mapy na horizontální ose určuje svou intenzitou vzdálenost kapky na odpovídající horizontální souřadnici scény od horního okraje (ukázka na obrázku č. 22). Intenzita je uložena v červeném kanálu a při každém volání metody *Update* (viz. kapitola 4.5.2) entity ***Rain*** je intenzita zvýšena s tím, že při překročení 100% se vrací na 0%.

Obrázek 22: Ukázka dešťové mapy

Ukázka efektu deště ve hře je vidět například na obrázku č. 26.

4.7.3 Vykreslování textu

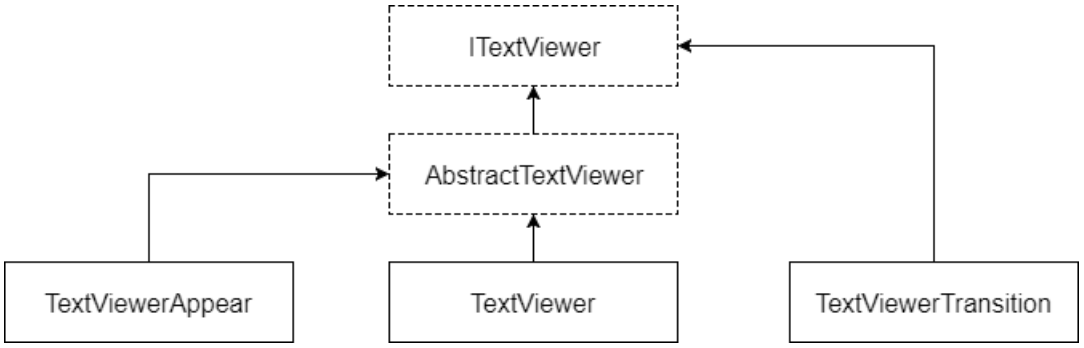
Pro zobrazení textů v herním světě byla vytvořena skupina grafických elementů. Všechny tyto elementy implementují jednoduchý obecný interface *ITextViewer* zobrazený ve zdrojovém kódu č. 15.

```
1 public interface ITextViewer
2 {
3   /// <summary>
4   /// Text to be displayed
5   /// </summary>
6   string Text { get; set; }
7 }
```

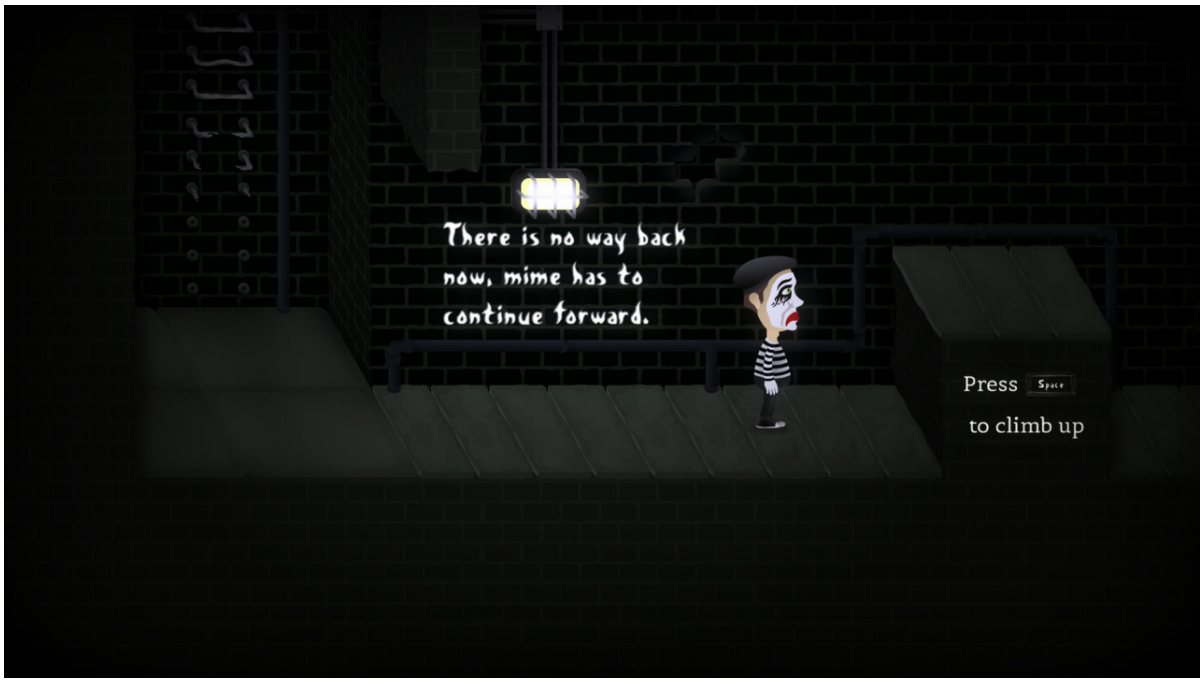
Zdrojový kód 15: Interface IText

Třídy implementující interface *ITextViewer* jsou dále zobrazeny na obrázku č. 23. Nejzákladnější z nich je třída *TextViewer*, ta jednoduše zobrazí zadaný text. Třída *TextViewerAppear* oproti tomu implementuje grafický efekt při zobrazení textu (fade nebo dissolve). Třída *TextViewerTransition* implementuje oproti tomu i grafický efekt při změně textu (toho je docíleno zapouzdřením více instancí třídy *TextViewerAppear*).

K samotnému vykreslování textů se využívá třída *TextTextureBuilder* z knihovny *Corpus*. Viz. kapitola č. 4.2.1.2. Ukázka vykresleného textu v herním světě je na obrázku č. 24.



Obrázek 23: Třídy implementující interface ITextViewer



Obrázek 24: Ukázka textu v herním světě

4.8 Elementy reprezentující fyzikální tělesa

Pro tyto elementy bylo v enginu ustáleno označení "entita". Společným předkem všech entit je třída ***Entity***. Entita ve své inicializační fázi (viz. 4.5.2) vytváří jedno nebo více fyzikálních těles (voláním metod fyzikálního enginu, viz. 4.1.1). Vytvářeným tělesům se nastaví reference na entitu, která je vytvořila, stejně tak entita má referenci na všechna tělesa, jež vytvoří. Tak je zajištěno provázání elementu herního enginu a fyzikálního tělesa enginu Farseer.

Entita nastavuje fyzikálnímu tělesu základní vlastnosti:

- Typ tělesa (dynamické, statické, kinematické)
- Míra tření
- Míra odrazivosti
- Hmotnost
- Působení gravitace na těleso (zapnuté/vypnuté)

Tabulka 9: Popis základních fyzikálních entit

Název	Popis
Polygon	Polygon, jehož tvar a rozměry mohou být libovolně definovány. Fyzikální engine Farseer (viz. 4.1.1) dokáže pracovat pouze s konvexními polygony, konkávní polygony je třeba rozdělit na sadu konvexních polygonů. O toto se entita stará ve své inicializační fázi (viz. 4.5.2) a používá k tomu algoritmy zabudované ve fyzikálním enginu.
Circle	Jednoduchý kruh s definovatelným poloměrem.
Sensor	Polygon, který nemůže s ničím kolidovat a reaguje na přítomnost jiné entity v něm. Může reagovat na všechny entity, pouze na herní postavy (hráč, NPC) nebo na všechny entity kromě herních postav. Pokud nastane očekávaná událost, je aktivována akce, jež může být navázána na téměř libovolnou činnost (viz. 4.9.1)
DummyEntity	Fyzikální těleso bez tvaru. Nepůsobí na něj žádné kolize, může být pomocí jointů spojeno s jinými fyzikálními tělesy na scéně. Do podřízených elementů entity lze zařadit libovolné elementy a ty takto spojit s fyzikální simulací.
DeadZone	Sensor, který automaticky zabije hráče, je-li v něm detekována jeho fyzikální reprezentace. Umisťuje se například do bezedných jam.

- Kolizní kategorie (používá se k určení, do které verze herního světa entita spadá, viz. 2.2)

Dále každá entita vytváří jeden neaktivní joint (typu RevoluteJoint), pomocí něhož lze zamknout pozici fyzikálního tělesa a pohybovat s ním po scéně. K tomuto účelu slouží metody *LockPosition* a *UnlockPosition*.

Z dalších metod je třeba jmenovat *IgnoreCollisionsWith* a *RestoreCollisionsWith*. Tyto metody slouží k dynamickému vypínání a zapínání kolizí mezi dvojící entit.

Entita se také stará o řešení situací, kdy se kolize mezi dvěmi tělesi povolí ve chvíli, kdy jsou tato tělesa do sebe zanořená. Tato situace může nastat po zavolání metody *RestoreCollisionsWith* nebo po přepnutí verze herního světa (viz. 2.2). V těchto situacích zůstanou kolize dočasně vypnuté do chvíle, než tělesa přestanou sdílet stejný prostor.

Základní poděděné entity jsou popsány v tabulce č. 9.

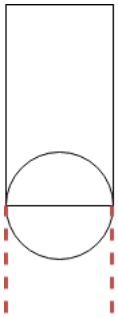
4.8.1 Reprezentace hráče

Jedna ze základních otázek u hry používající fyzikální engine pro simulaci kompletního herního světa je otázka, jakým způsobem bude reprezentován hráč. Hráče není možno od fyzikální simulace oddělit, ani z něj udělat pevné těleso v prostoru. Uvedený přístup by působil nepřirozeně. Například pokud by do hráčovy postavy narazil mnohonásobně těžší předmět, buď by se tento odrazil nebo hráčem prošel bez zpětné vazby v podobě kolize.

Z těchto důvodů je potřeba vytvořit dynamickou fyzikální reprezentaci, jež bude

reagovat na svět kolem, jejíž velikost bude zhruba odpovídat grafické reprezentaci (správná reakce na kolize) a zároveň ji půjde ovládat (chůze, skákání...). Jako první nápad se nabízí jednoduchý obdelník, na který bude působeno kontinuální silou pro pohyb. Nicméně toto řešení má některé nedostatky. Největším problémem jsou hranaté okraje, těleso se bude zasekávat i o ty nejmenší nerovnosti. Rovněž je tu problém se sešikmenou podlahou, na které je třeba obdelník zrotovat, což nemusí být vždy žádoucí, pokud chceme grafickou reprezentaci udržet ve vzpřímené poloze. Pro některé typy her může být tento přístup dostačující, pro vyvíjenou hru to nestačí.

Elegantním řešením všech zmíněných vad je fyzikální těleso vytvořené spojením kruhu a obdelníku. Koncept je zobrazen na obrázku č. 25. Obdelník má uzamknuté rotace, čímž je zaručeno, že vždy bude ve zpřímené poloze. Kruh je k obdelníku připojen pomocí revolute jointu ve středu kruhu. Joint zde má dva účely, spojuje fyzikální tělesa a zároveň funguje jako motor. Revolute jointu lze nastavit točivý moment a kvůli uzamknutí rotací obdelníku toto přirozeně vede k rotaci kruhu a pohybu celého tělesa (při nastavení vhodné úrovně tření kruhu). Princip pohybu je stejný jako například u jízdního kola. Jedná se o velmi jednoduché a elegantní řešení, které zároveň působí velmi přirozeně [49].



Obrázek 25: Fyzikální reprezentace hráče

Skákání je v tomto případě realizováno aplikací silového vektoru ve vhodném směru.

Dále je fyzikální reprezentace rozšířena o detekci vzdálenosti od země. Toto je řešeno kontinuálním (v každé update fázi, viz. 4.5.2) raycastingem dvou paprsků ze základny. Tyto dva paprsky jsou vyznačeny na obrázku č. 25 červenou přerušovanou čárou. Kromě detekce vzdálenosti od země je možná i detekce hran ve světě.

Tato kompletní funkcionalita je implementována ve třídě ***Walker***, která dědí ze třídy ***Entity*** (viz. 4.8). Samotnou třídu je možné použít nejen k reprezentaci hráče, ale i libovolných NPC (počítačem ovládané postavy) [50].

Do třídy ***Walker*** je dále zaimplementována podpora žebříků. Jedná se o detekci situace, kdy fyzikální reprezentace má nějaký žebřík v dosahu a metody pro chycení se žebříku, pohybování se po žebříku a puštění se žebříku (viz. 4.8.2).

Entita ***Walker*** se může nacházet v několika různých stavech definovaných výčtovým typem *WalkerState*:

- walking - běžná chůze, standardní stav
- running - běh, rychlější pohyb
- climbing - lezení po žebříku (viz. 4.8.2)

- midair - fyzikální reprezentace se nedotýká země, volný pád nebo skok
- jumping - fáze těsně před skokem, v této fázi se může například přehrát animace výskoku apod.
- interaction - rozšířená interakce s prostředím, možnosti záleží na konkrétní implementaci (v poděděné třídě), může se jednat o různé tlačení či tahání předmětů apod.
- ledge - fyzikální reprezentace se nachází v oblasti, která je definována jako římsa. V této oblasti jsou omezeny pohybové možnosti a situaci je třeba promítnout do grafické reprezentace (příklad na obrázku č. 26)

Všechny tyto stavy upravují chování fyzikální reprezentace (rychlost pohybu, omezení či rozšíření možností) a měly by se promítat i do grafické reprezentace.

Obrázek 26: Ukázka chůze po římse

Element pro reprezentaci hráče dědí z třídy ***Walker*** a rozšiřuje ji o grafickou reprezentaci, zpracování vstupů z klávesnice/gamepadu (ovládání fyzikální reprezentace) a další specifickou funkcionalitu (speciální schopnosti postavy apod.).

4.8.2 Žebříky

Entita ***Ladder*** simuluje v enginu žebříky. Umožňuje elementům poděděných z entity *Walker* (viz. kapitola 4.8.1) pohybovat se v určitých částech herního světa plynule vertikálně nebo i horizontálně.

Jsou rozlišované tři druhy žebříků (módy fungování této entity):

- ladder - běžný žebřík, pohyb je možný pouze po vertikální ose

- pipe - stejné vlastnosti jako u předchozího druhu, může se lišit animace při šplhání
- wall - pohyb je možný po vertikální i horizontální ose, simuluje různé živé ploty, po nichž se ve hrách dá často šplhat

Entita ***Ladder*** je poděděná z entity *Sensor* (viz. tabulka č. 9) a aktivně monitoruje elementy v její oblasti. Entity poděděné z entity *Walker* notifikuje o tom, že se nachází v oblasti žebříku. Reference na ostatní druhy entit v oblasti žebříku jsou uloženy v kolekci a jsou využívány pro správné chování kolizí. Pro správné fungování ***Ladder*** deaktivuje kolize mezi instancí entity *Walker*, která žebřík právě využívá, a ostatními entitami skrz které žebřík prochází. Tak lze například prolézat skrz patra herního světa.

Jeden žebřík může využívat vždy pouze jedna entita typu *Walker*. Nicméně žebříky je možno skládat za sebe s tím, že logika uvnitř elementu *Walker* se stará o přechod z jednoho žebříku na druhý automaticky na pozadí.

4.9 Element bez grafické reprezentaci i fyzikálního tělesa

Jedná se o elementy, které běžně nelze na scéně přímo zpozorovat. Většinou nějakým způsobem ovlivňují další elementy.

4.9.1 ElementAction

Jeden ze základních elementů této kategorie je ***ElementAction***.

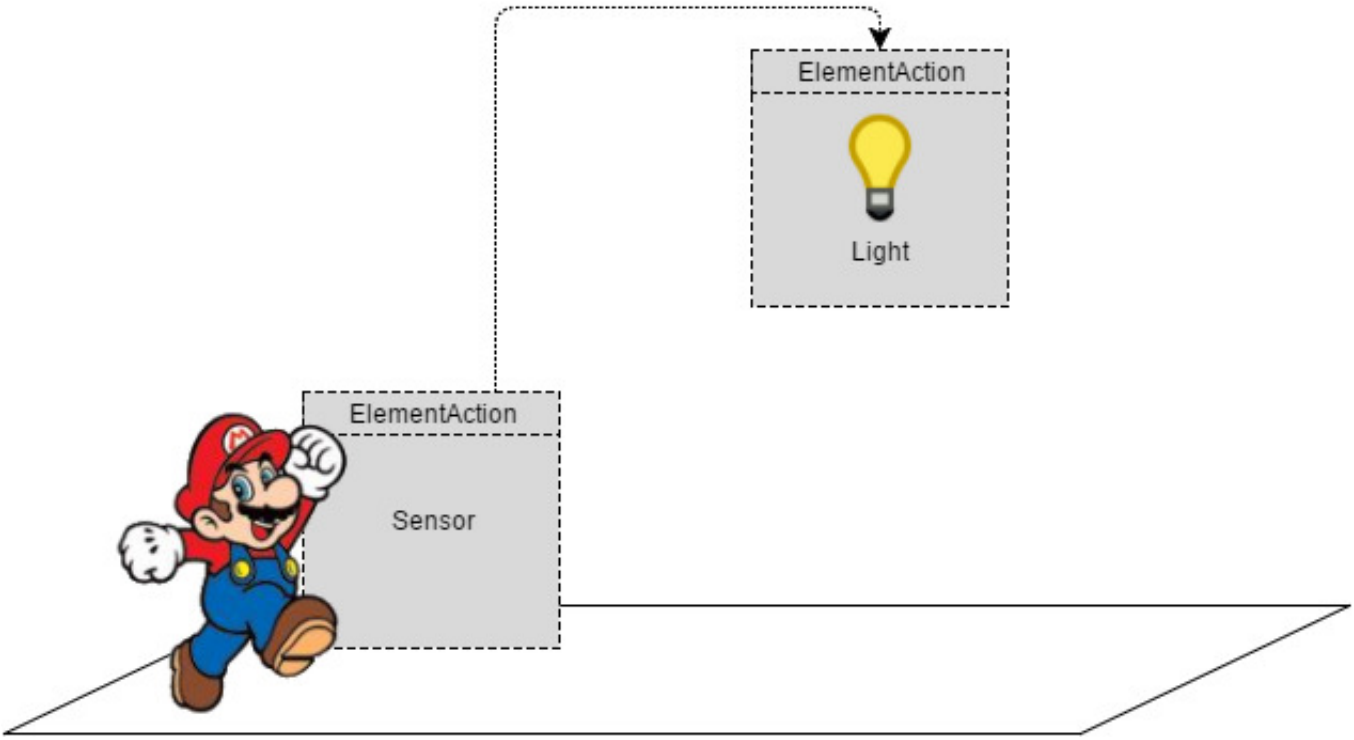
Jedná se o stavový objekt, jenž může být v jednu chvíli buď **aktivní** nebo **neaktivní**. Na tento stav může být navázána celá řada akcí, stejně tak změnu stavu může vyvolat téměř cokoliv. Vždy záleží na konkrétním použití.

Instance třídy ***ElementAction*** se na sebe mohou řetězově napojovat. Jedna instance může být navázána na libovolné množství dalších instancí, kterým mění stav při každé změně svého stavu. Dále je na změnu stavu možno navázat událost.

Příkladem využití elementů ***ElementAction*** (dále "Akce") je rozsvícení světla v přítomnosti hráče. Tento scénář je nastíněný na obrázku č. 27. Fyzický senzor (viz. tabulka č. 9) obsahuje instanci třídy ***ElementAction***, která se v tomto případě aktivuje ve chvíli, kdy do oblasti senzoru vstoupí hráč. Na tuto Akci bude navázána Akce zdroje světla (viz. tabulka č. 8). Zdroj světla se vykresluje jen tehdy, je-li jeho Akce aktivována. Proto bude světlo vykreslováno jen za podmínky detekce hráče senzorem.

Vlastní chování Akce může být značně modifikováno nastavením tohoto elementu. Základní nastavovanou vlastností je samotný typ Akce. Jsou definovány tyto typy:

- **Switch** - Výchozí typ. Akce při aktivaci zůstane aktivována až do chvíle, kdy dojde k deaktivaci.
- **TurnOn** - Jakmile je Akce jednou aktivována, je tento stav permanentní a není možné ji znovu deaktivovat.
- **TurnOff** - Jakmile je Akce jednou deaktivována, je tento stav permanentní a není možné ji znovu aktivovat.



Obrázek 27: Příklad využití ElementAction

- **Event** - Akce nezůstává dlouhodobě aktivována. Při aktivaci je tato informace běžně zpropagována (zavolání události, aktivace napojených Akcí), poté je stav Akce změněn zpět na "deaktivováno".
- **Change** - Akce reaguje na změnu vnitřního stavu. Nezáleží na tom, zda je aktivována nebo deaktivována, vždy při změně je zpropagován stav "aktivováno"(zavolání události, aktivace napojených Akcí).

Dále je možné invertovat výsledný stav Akce. K tomu slouží vlastnost pojmenována *NOT*. Pokud je vlastnost nastavena, stav "deaktivováno"se navenek jeví jako "aktivováno"a naopak.

Další možnost nastavení je nastavení zpoždění. Při aktivaci nebo deaktivaci Akce nemusí tato operace proběhnout ihned, ale může být odložena o přesně definovaný počet milisekund.

Existují i některé speciální akce poděděné z třídy ***ElementAction***, ty jsou popsané v tabulce č. 10.

4.9.2 Animátory

Animátory jsou akce (viz. kapitola 4.9.1), které když jsou aktivní, tak ovlivňují vlastnosti vybraných elementů. Může se jednat o změnu pozice, barvy, rotace a podobně.

Abstraktní třída ***Animator*** obsahuje obecnou funkcionalitu, kterou využívají podědené konkrétní animátory. Jedná se o serializaci a deserializaci seznamu ovlivňovaných elementů, kontrolu typu těchto elementů (zda obsahují vlastnosti, které chce animátor ovlivňovat) a vyhodnocování, zda je animátor aktivní.

Popis základních animátorů je v tabulce č. 11.

Tabulka 10: Popis speciálních akcí

Název	Popis
CounterAction	Akce je aktivována až po určitém předem definovaném počtu aktivací.
RandomAction	Akce je náhodně samovolně aktivována dle nastavení (interval, pravděpodobnost).
ProgressAction	Akce závisí na určitím progressu (viz. kapitola 4.9.3). Nastavuje se jí mez, při které se aktivuje.
ElementActionFalse	Akce rozšířena o instanci další akce pojmenované *FalseAction*. Využívá se u akcí, jejichž vyhodnocení mohou ovlivňovat ještě jiné skutečnosti. Při aktivaci takových akcí nemusí nutně dojít k jejich skutečné aktivaci (pokud nejsou splněny některé další podmínky). V takovém případě je aktivována akce *FalseAction*. Jedná se o abstraktní třídu, jejíž využití je vidět například v kapitole č. 4.9.4.
DecisionAction	Slouží k větvení. Může mít na sebe navázáno 0 až N akcí, z nichž se aktivuje určitá podmnožina. Tato podmnožina je vybrána pomocí rozhodovací mapy (viz. kapitola 4.9.5).

4.9.3 Progress

V některých případech je potřeba mezi elementy propagovat více než jen binární informace (viz. kapitola 4.9.1). Z tohoto důvodu existují elementy umožňující propagaci informace o procentuálním postupu. Standardně tedy nabívají hodnot **0.0 až 1.0**, nicméně tento interval lze možné upravovat.

Jádrem poskytování informace o postupu je abstraktní třída ***AbstractProgress***. Z této třídy dědí jak elementy poskytující informaci, tak spracovávající informaci, jak lze vyčíst z obrázku č. 28.

Samotná třída ***AbstractProgress*** obsahuje funkcionalitu pro:

- Nastavení a dodržování mezí intervalu postupu.
- Abstraktní metody pro nastavování a zjišťování aktuální hodnoty postupu.

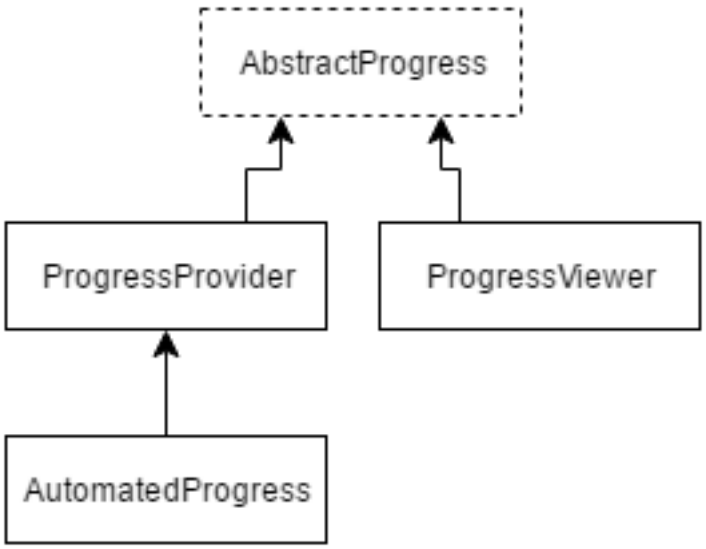
Element ***ProgressProvider*** obsahuje samotnou hodnotu postupu a umožňuje její nastavování a čtení.

Element ***ProgressViewer*** oproti tomu neobsahuje samotnou hodnotu postupu, nicméně má referenci na instanci typu ***ProgressProvider*** (poskytovatel postupu), jejíž hodnotu využívá. Díky podědění z ***AbstractProgress*** může mít nastavené jiné meze, než poskytovatel postupu.

Element ***AutomatedProgress*** rozšiřuje ***ProgressProvider*** o funkcionalitu pro autonomní změnu vnitřní hodnoty. Způsob, jakým se bude postup měnit, se nastavuje těmito parametry:

Tabulka 11: Popis základních animátorů

Název	Popis
Visibility	Nastavení vlastnosti viditelnosti (viz. kapitola 4.7) vybraných elementů.
Rotate	Rotace vybraných elementů. Je možno nastavit rychlost rotace (v radiánech za vteřinu).
SpriteAnimator	Animování vybraných spritů (viz. kapitola 4.3.2). Je možno nastavit počáteční a koncový snímek, rychlost animace a zda se má animace opakovat.
Fade	Animování alpha transparentnosti vybraných elementů. Lze nastavit cílovou procentuální transparentnost a rychlost animace.
Enabler	U vybraných elementů nastavuje, zda jsou zapnuté či vypnuté. Hodí se například u fyzikálních těles, entit (viz. kapitola 4.8), které nejsou vůbec simulovány, jsou-li vypnuty.
BlurAnimator	Animace Gaussova rozostření u elementů, na které je rozostření aplikováno. Lze nastavit cílovou míru rozostření a rychlost transformace.
OverColorAnimator	Umožňuje animovat barvu překrývající nějaký grafický element (viz. kapitola 4.7.1). Lze nastavit cílovou barvu překrytí. Postup transformace se získává z elementu typu ***ProgressViewer***, což umožňuje mnohem větší míru variability (viz. kapitola 4.9.3).
Path	Animátor ovlivňuje pozici vybraných elementů. Animátor obsahuje několik klíčových bodů, z nichž vytváří cestu, po kterých pohybuje elementy. Cesta může být otevřená nebo uzavřená. Kromě samotné cesty je možno nastavit i zda se má projít jen jednorázově nebo opakovaně a dobu, jak dlouho má trvat jedno její kompletní projití.
ProgressPath	Obdobné jako **Path**, pouze pro zjištění aktuálního postupu po cestě se využívá element typu ***ProgressViewer*** (viz. kapitola 4.9.3).
EntityType	Změna typu entity, tedy fyzikálního tělesa. Pomocí tohoto animátoru je možné v průběhu hraní udělat například z dynamického tělesa statické nebo naopak.



Obrázek 28: Třídy poděděné z AbstractProgress

- Absolutní rychlost přírustku.
- Druh přírustku (lineární, exponenciální ...).
- Druh opakování (bez opakování, opakování s vynulováním ...).

Příkladem využití je například zvedání brány navíjením řetězu. Navíjecí mechanismus poskytuje instanci třídy ***ProgressProvider***, kterou konzumuje vhodný animátor (viz. kapitola 4.9.2) skrz instanci třídy ***ProgressViewer***. Tento animátor bude nastavený tak, aby pohyboval objektem brány.

4.9.4 Práce s proměnnými

Aby bylo možno dosáhnout skutečného větvení herního obsahu (viz. kapitola č. 2.3), je potřeba mít možnost uchovávat si krátkodobě i dlouhodobě různé hráčovy volby a mít možnost dle těchto uložených dat kdykoliv rozhodovat. Tyto informace jsou ukládány do proměnných, které se poté serializují a deserializují.

Samotné proměnné jsou spravovány třídou ***Variables***, jenž využívá návrhového vzoru singleton. Proto vždy existuje pouze v jedné statické instanci pro celou aplikaci [51]. Tato třída obsahuje 5 kolekcí typu ***VariablesCollection***, jenž obsahují samotné proměnné. Každý z těchto kolekcí má jiný kontext, do kterého spadají její proměnné. Jsou rozlišovány tyto kontexty:

- **Session** - Jedno hraní hry. Proměnné jsou platné pro aktuálně spuštěnou instanci aplikace, po ukončení jsou ztraceny. *Příklad: datum a čas, kdy byla aplikace spuštěna*
- **Level** - Jedna herní úroveň, herní svět. Proměnné jsou platné v rámci herní úrovně, při načtení nové úrovně jsou ztraceny. Při uložení rozehráté úrovně jsou proměnné uloženy k ní. *Příklad: hráčovo rozhodnutí ve hře, které má vliv na jinou část stejné úrovně*

- **Game** - Jeden průchod hrou, platné napříč úrovněmi. Při zahájení nové hry jsou ztraceny. Při uložení rozehráté hry se proměnné ukládají. *Příklad: hráčovo rozhodnutí ve hře, které má vliv na nějakou část libovolné úrovně ve hře*
- **User** - Uživatel hry. Hra bude umět rozeznávat různé uživatele, kteří mohou mít paralelně rozehráto několik her (každý z uživatelů). *Příklad: nastavení herní obtížnosti*
- **Global** - Persistentní proměnné napříč všemi uživateli, hrami, úrovněmi a podobně. *Příklad: nastavení rozlišení hry*

Le vyčíst, že takto implementované proměnné mají velkou škálu možných využití. Používají se například i pro herní inventář.

Veřejné rozhraní třídy ***Variables*** se skládá z metod a parametrů pro přístup k jednotlivým kolekcím a serializaci a deserializaci těchto kolekcí.

Třída ***VariablesCollection*** je ve svém základu kolekce typu slovník. Jednotlivé proměnné mají tedy unikátní textový identifikátor, podle kterého je možno velmi rychle dohledávat [52].

Samostatné proměnné jsou poté zapouzdřena ve třídě ***VariableValue***. Využívá se zde toho, že v jazyku C# je vše objekt. Středobodem této třídy je privátní proměnná typu *object*. Veřejné rozhraní třídy ***VariableValue*** obsahuje metody pro snadnou konverzi mezi datovými typy (viz. zdrojový kód č. 16).

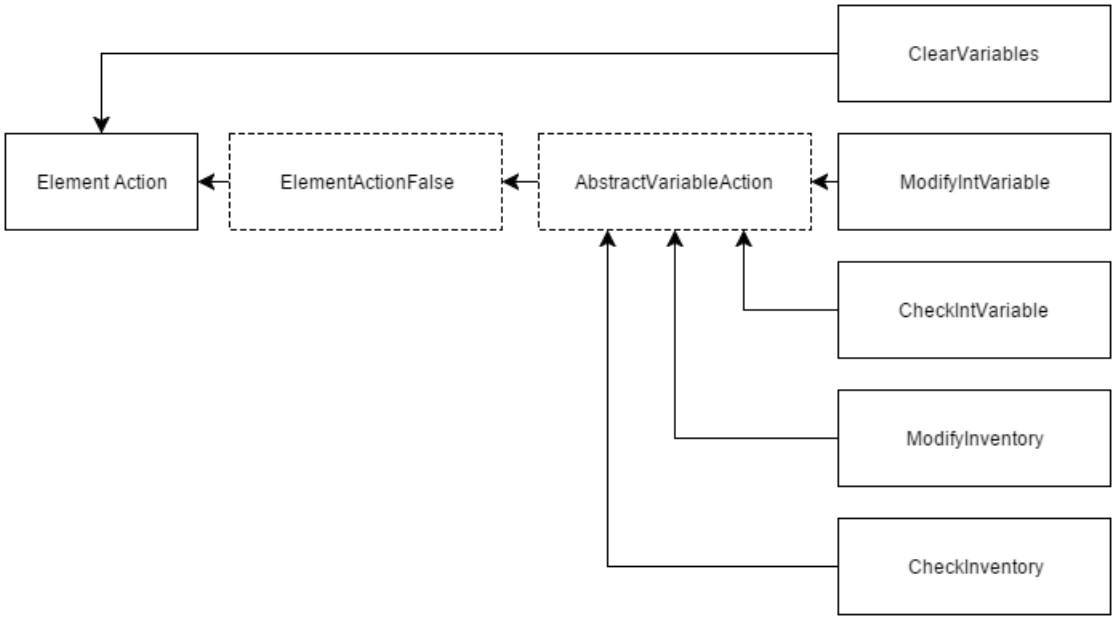
```
1  // properties
2  public bool IsSet { get; }
3  // methods
4  public int GetInt();
5  public float GetFloat();
6  public bool GetBoolean();
7  public string GetString();
8  public object GetObject();
9  public void SetValue(int number);
10 public void SetValue(float number);
11 public void SetValue(bool boolean);
12 public void SetValue(string text);
13 public void SetObject(object obj);
14 public void Serialize(SaveGameFile saveGameFile);
15 public void Deserialize(SaveGameFile saveGameFile);
```

Zdrojový kód 16: Veřejné rozhraní třídy VariableValue

Pro práci se systémem proměnných existuje několik elementů poděděných z třídy *ElementAction* (viz. kapitola 4.9.1). Elementy a dědičnost mezi nimi jsou vidět na obrázku č. 29.

Element ***AbstractVariableAction*** zapouzdřuje základní funkcionalitu nutnou pro práci s proměnnými. Obsahuje vlastnosti pro nastavení kontextu a názvu proměnné, kterou bude číst nebo nastavovat.

Jak lze vidět, vše je řízeno akcemi. Při nastavování proměnných dojde k požadované činnosti při aktivaci elementu. Čtení proměnných je rovněž prováděno při aktivaci elementu, nicméně to, zda bude element skutečně aktivován, závisí na proměnné (v tomto případě se využívá funkcionalita třídy *ElementActionFalse* popsaná v tabulce č. 10).



Obrázek 29: Organizace elementů pro práci se systémem proměnných

Popis jednotlivých koncových elementů pro práci se systémem proměnných je v tabulce č. 12.

4.9.5 Rozhodovací mapa

Rozhodovací mapa je konstrukt sloužící ke zjednodušení práce s komplexními větveními příběhu (viz. kapitola 2.3). Element ***CustomDecisionMap*** je zjednodušeně implementace rozhodovacího stromu [53]. Obsahuje jednotlivé uzly (rozhodnutí) a vazby mezi nimi, přičemž jeden uzel může vést do libovolného počtu dalších uzlů a tímto způsobem je realizován strom. Jeden z uzlů je vždy **počáteční** a 1 až N uzlů je **koncových**.

Samotné uzly rozhodovacího stromu jsou realizovány elementy typu *ElementAction* (viz. kapitola 4.9.1). Z toho důvodu je tedy velmi snadná implementace rozhodování podle herních proměnných (viz. kapitola 4.9.4), podle procentuálního postupu uloženého v herních elementech (viz. kapitola 4.9.3) a podobně.

Velmi důležitým elementem je generická třída ***ElementProvider***, jejíž definici lze vyčíst ze zdrojového kódu č.17.

```
1 public abstract class ElementProvider<T>:Element where T : Element
2 {
3   public CustomDecisionMap DecisionMap { get; set; }
4   public List<T> SelectedElements { get; }
5
6   public void CreateLink(ElementAction action, T element);
7   public List<T> Provide();
8 }
```

Zdrojový kód 17: Definice a veřejné rozhraní třídy ElementProvider

Jak lze vidět, třída ***ElementProvider*** vystavuje 0 až N objektů dědících z třídy

Tabulka 12: Popis elementů pro práci se systémem proměnných

Název	Popis
ModifyIntVariable	Nastavování proměnných celočíselného typu. Umožňuje nastavit proměnnou nebo upravovat hodnotu již existující proměnné (přičítání, odečítání).
CheckIntVariable	Vyhodnocování proměnných celočíselného typu. Nastavená proměnná je zkontrolována podle jednoho z těchto pravidel: • Undefined - proměnná není vůbec nastavena • Defined - proměnná je nastavena na nějakou hodnotu • Equal - hodnota proměnné je rovna specifikované hodnotě • LowerThan - hodnota proměnné je nižší než specifikovaná hodnota • HigherThan - hodnota proměnné je vyšší než specifikovaná hodnota • Differeint - hodnota proměnné je odlišná od specifikovaná hodnoty Kontrola je prováděna při aktivaci této akce. Aktivace proběhne jen v případě kladného výsledku kontroly.
ModifyInventory	Přidávání a odebírání položky z inventáře. Inventář je vnitřně reprezentován proměnnými celočíselného typu.
CheckInventory	Kontrola, že určitá položka existuje v inventáři.

Tabulka 13: Elementy poděděné z ElementProvider

Název	Popis
ActionProvider	Poskytuje elementy typu *ElementAction* (viz. kapitola 4.9.1). Element ***ActionProvider*** využívá akce ***DecisionAction***, která po aktivace dále aktivuje elementy dle aktuálního stavu referencované rozhodovací mapy.
LibraryProvider	Poskytuje elementy typu *Library* obsahující kolekci textů, viz. kapitola 4.9.6.

Element. K tomuto využívá rozhodovací mapu (element typu ***CustomDecisionMap***). Rozhodovací mapu referencuje v property *DecisionMap*. Ke koncovým uzlům této mapy jsou namapovány elementy (funkce *CreateLink*) a podle aktuálního vyhodnocení referencované mapy jsou poskytovány pouze elementy namapované k aktivnímu koncovému uzlu mapy. Některé poděděné elementy z třídy ***ElementProvider*** jsou vidět v tabulce č. 13.

4.9.6 Knihovna

Element ***Library*** obsahuje kolekci po sobě jdoucích textů. Slouží k reprezentaci různých dialogů nebo krátkých textových průpovídek ve hře. Element je poděděný z elementu *ElementAction* (viz. kapitola 4.9.1). Při každé aktivaci se zobrazí následující text. Pro samotné zobrazení textu v herním světě se využívá kolekce 1 až N elementů implementujících interface *ITextViewer* (viz. kapitola 4.7.3).

Samotné texty jsou v elementu uloženy následujícím způsobem. Jeden textový řetězec určený k zobrazení právě v jednom vieweru (objekt třídy implementující interface *ITextViewer*) je uložen ve třídě *TextUnit* (lze vidět ve zdrojovém kódu č.18).

```
1 public class TextUnit
2 {
3   public int ViewerIndex { get; set; }
4   public string Text { get; set; }
5
6   public void Clear();
7 }
```

Zdrojový kód 18: Veřejné rozhraní třídy TextUnit

Instance třídy *TextUnit* se dále shlukují ve třídě *TextSet*. Třída *TextSet* reprezentuje jednu sadu textů, které jsou vždy zobrazeny ve stejnou chvíli. Kolekce těchto instancí je spravována třídou ***TextSetsList***, jenž tvoří základ elementu ***Library***.

Závěr

Podařilo se úspěšně vytvořit funkční herní engine splňující všechny stanovené požadavky. Třemi klíčovými požadavky bylo vytvořit funkční řešení pro: vykreslování grafiky (s ohledem na falešnou perspektivu v herních scénách), přepínání mezi herními světy (především s ohledem na fyzikální simulaci) a větvení příběhu (znázorňování dopadů hráčových voleb na herní svět).

Správné vykreslování grafiky s falešnou perspektivou byl pravděpodobně nejzásadnější bod z výše zmíněných. Především kvůli tomuto požadavku nebylo snadno možné použít jakýkoliv dostupný existující herní engine. Problém nemá obecné řešení, v enginu byl zvolen systém úzce související s návrhem herních scén a formátem grafických podkladů. Implementované řešení je zcela funkční a vyvíjené hře dostačuje. Dále má engine po grafické stránce již zabudovanou funkcionalitu pro animované sprity, podporu velkých textur, vykreslování deště a mnohem více.

Přepínání mezi dvěma verzemi herního světa je další funkcionalita úzce spojena s vyvíjenou hrou. Ačkoliv se podobné herní mechaniky v některých existujících hrách již vyskytují (např. *Quantum Conundrum*), nejedná se ani zdaleka o natolik častou mechaniku, aby byla již zabudována v dostupných enginech. Řešení tohoto požadavku je implementováno v nejnižších vrstvách enginu. Proto má každý objekt na scéně vždy definováno, do které verze herního světa patří a chová se dle toho. Také zde bylo potřeba vyřešit problém s fyzikálními tělesy na scéně. Pro jejich dynamickou povahu se může lehce stát, že během přepínání mezi světy se jedno fyzikální těleso objeví v jiném. Tuto problematiku engine bezezbytku řeší a nedochází k žádným nežádoucím jevům.

Větvení příběhu a reakce na hráčovy volby se dnes stává již relativně častou mechanikou v počítačových hrách (např. *Life is Strange*). Nicméně neexistuje standardizovaný způsob, jak funkcionalitu řešit. Implementace ve vyvíjeném enginu používá již existující logické elementy pro vytvoření rozhodovacích stromů. Tento přístup zajišťuji univerzálnost a snadnou rozšiřitelnost a umožňuje rozhodovací stromy snadno navázat na téměř libovolný herní mechanismus.

Úspěšné vytvoření enginu a vyřešení všech požadavků dokazuje i počítačová hra *Other Inside*, kterou engine pohání. Hra *Other Inside* odpovídá dnešním standardům herního průmyslu, což mimo jiné potvrzuje i jisté mediální pokrytí, které již hra dokázala získat [54][55].

Nicméně použití enginu pro žánrově odlišné tituly není dost dobře možné, a to právě z důvodu jeho svázání s požadavky konkrétního herního titulu. Tvorba zcela univerzálního enginu ale nikdy nebyla cílem. Naproti tomu má engine vůči existujícím univerzálním enginům relativní výhodu. Některé jeho vlastnosti nelze snadno a optimálně implementovat v dostupných enginech.

Kromě toho, že engine obsahuje vše potřebné pro vytvoření hry, jeho architektura současně umožňuje velmi snadné rozšiřování. Nezáleží na tom, zda se jedná o novou funkcionalitu nebo třeba migraci na více platforem.

Literatura

[1]. *OtherInside* [online] [cit. 2016-06-23]. Dostupné z: http://otherinside.com/.

[2]. HUIZINGA, Johan. *Homo ludens: o původu kultury ve hře*. Vyd. 2. Praha: Dauphin, 2000. ISBN 80-7272-020-1.

[3]. JIRKOVSKÝ, Jan. *Game industry: vývoj počítačových her a kapitoly z herního průmyslu*. Vyd. 1. Praha: D.A.M.O., 2011. ISBN 978-809-0438-712.

[4]. APPERLEY, Thomas H. *Genre and game studies: toward a critical approach to video game genres* [online]. University of Melbourne, 2006 [cit. 2016-01-23]. Dostupné z: http://trac.assembla.com/CommanderAssembler/export/32/docs/Genre%20and%%20tom-apperley.pdf.

[5]. JAKUBEC, Martin. *Vývoj herního editoru na platformě Flash* [online]. 2014 [cit. 2016-01-23]. Dostupné z: http://is.muni.cz/th/256318/fi_m/. Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno. Vedoucí práce Barbora KOZLÍKOVÁ.

[6]. *Gamedev glossary: library vs. framework vs. engine* [online]. 2015 [cit. 2016-01-23]. Dostupné z: http://www.gamefromscratch.com/post/2015/06/13/GameDev-Glossary-Library-Vs-Framework-Vs-Engine.aspx.

[7]. *Unreal Engine* [online] [cit. 2016-01-25]. Dostupné z: https://en.wikipedia.org/wiki/Unrea

[8]. *Unreal Engine Features* [online] [cit. 2016-01-25]. Dostupné z: https://www.unrealengine.com/unreal-engine-4.

[9]. *Unity (game engine)* [online] [cit. 2016-01-25]. Dostupné z: https://en.wikipedia.org/wiki/

[10]. BOŘÁNEK, Roman. *Vyvíjejte zdarma a ještě vám zaplatíme, herní enginy bojují o vývojáře* [online]. 2015 [cit. 2016-01-25]. Dostupné z: http://www.root.cz/clanky/vyvijejte-zdarma-a-jeste-vam-zaplatime-herni-enginy-bojuji-o-vyvojare/.

[11]. *Unity 5 engine overview* [online] [cit. 2016-01-25]. Dostupné z: https://unity3d.com/unity/engine-features.

[12]. *CryEngine* [online] [cit. 2016-01-25]. Dostupné z: https://en.wikipedia.org/wiki/CryEngi

[13]. *CRYENGINE* [online] [cit. 2016-01-27]. Dostupné z: http://store.steampowered.com/app

[14]. *CryEngine: Features* [online] [cit. 2016-01-27]. Dostupné z: http://www.cryengine.com/fe

[15]. *id Tech* [online] [cit. 2016-01-27]. Dostupné z: https://en.wikipedia.org/wiki/Id_Tech.

[16]. *Doom engine* [online] [cit. 2016-01-27]. Dostupné z: https://en.wikipedia.org/wiki/Doom_

[17]. *id Tech 3* [online] [cit. 2016-01-27]. Dostupné z: https://en.wikipedia.org/wiki/Id_Tech_

[18]. *id Tech 5* [online] [cit. 2016-01-27]. Dostupné z: https://en.wikipedia.org/wiki/Id_Tech_

[19]. *Dunie Engine* [online] [cit. 2016-01-27]. Dostupné z: http://gutenberg.us/articles/Dunia_

[20]. *Anvil (game engine)* [online] [cit. 2016-01-27]. Dostupné z: https://en.wikipedia.org/wiki/

[21]. LEWIS, Anne. *The Secrets Behind Watch Dogs' Next Gen Experience* [online]. 2013 [cit. 2016-01-29]. Dostupné z: http://blog.ubi.com/watch-dogs-disrupt-engine-multiplayer/.

[22]. *Frostbite (game engine)* [online] [cit. 2016-01-31]. Dostupné z: https://en.wikipedia.org/wiki/Frostbite_(game_engine).

[23]. *Frostbite: This is Frostbite* [online] [cit. 2016-02-02]. Dostupné z: http://www.frostbite.com/about/this-is-frostbite/.

[24]. *Frictional Games: About* [online] [cit. 2016-01-31]. Dostupné z: https://www.frictionalgames.com/site/about.

[25]. *HPL Engine* [online] [cit. 2016-01-31]. Dostupné z: http://amnesia.wikia.com/wiki/HPL_

[26]. *Unreal Engine: Paper 2D* [online] [cit. 2017-07-01]. Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Paper2D/.

[27]. *Painter's algorithm* [online] [cit. 2017-07-01]. Dostupné z: https://en.wikipedia.org/wiki/

[28]. *Z-buffering* [online] [cit. 2017-07-01]. Dostupné z: https://en.wikipedia.org/wiki/Z-buffering.

[29]. BAKER, Steve. *Alpha-blending and the Z-buffer* [online] [cit. 2017-07-01]. Dostupné z: https://www.sjbaker.org/steve/omniv/alpha_sorting.html.

[30]. *Microsoft XNA* [online] [cit. 2016-01-25]. Dostupné z: https://en.wikipedia.org/wiki/Mic

[31]. ROSE, Mike. *It's official: XNA is dead* [online]. 2013 [cit. 2016-01-25]. Dostupné z: http://gamasutra.com/view/news/185894/Its_official_XNA_is_dead.php.

[32]. *XNA Framework Class Library* [online] [cit. 2016-02-02]. Dostupné z: https://msdn.microsoft.com/en-us/library/bb203940.aspx.

[33]. KALANDRA, Petr. *Renderování vektorové grafiky v XNA frameworku* [online]. 2013 [cit. 2016-02-02]. Dostupné z: http://theses.cz/id/iigbm1/. Bakalářská práce. Univerzita Palackého v Olomouci, Přírodovědecká fakulta, Olomouc. Vedoucí práce Ph.D. MGR. EDUARD BARTL.

[34]. VALDAUF, Zdeněk. *Vývoj aplikací pro Windows Phone "Mango"[online]* [online]. 2013 [cit. 2016-02-02]. Dostupné z: http://theses.cz/id/9w3cu4/. Bakalářská práce. Jihočeská univerzita v Českých Budějovicích, Pedagogická fakulta, České Budějovice. Vedoucí práce Ph.D. PAEDDR. PETR PEXA.

[35]. *MonoGame: About* [online] [cit. 2016-02-02]. Dostupné z: http://www.monogame.net/ab

[36]. *Rapid application development* [online] [cit. 2016-07-02]. Dostupné z: https://en.wikipedia.org/wiki/Rapid_application_development.

[37]. *Farseer Physics* [online] [cit. 2016-07-02]. Dostupné z: https://farseerphysics.codeplex.co

[38]. *xnagameconsole* [online] [cit. 2016-07-05]. Dostupné z: https://code.google.com/archive/

[39]. *Creating a Custom Effect: XNA Game Studio 4.0* [online] [cit. 2016-07-12]. Dostupné z: https://msdn.microsoft.com/en-us/library/bb203872(v=xnagamestudio.40).aspx.

[40]. *Vertex* [online] [cit. 2016-07-13]. Dostupné z: https://cs.wikipedia.org/wiki/Vertex.

[41]. *What Is Color Blending?* [online] [cit. 2016-07-23]. Dostupné z: https://msdn.microsoft.com/en-us/library/bb976070.aspx.

[42]. *Gaussian blur* [online] [cit. 2016-07-23]. Dostupné z: https://en.wikipedia.org/wiki/Gaus

[43]. *Virtual camera system* [online] [cit. 2016-07-23]. Dostupné z: https://en.wikipedia.org/wiki/Virtual_camera_system.

[44]. *Unity documentation: Mathf.SmoothDamp* [online] [cit. 2016-07-24]. Dostupné z: http://docs.unity3d.com/ScriptReference/Mathf.SmoothDamp.html.

[45]. *Cameras in 2D platformers* [online]. 2012 [cit. 2016-07-24]. Dostupné z: http://www.imake-games.com/cameras-in-2d-platformers/.

[46]. *Parallax scrolling* [online] [cit. 2016-07-30]. Dostupné z: https://en.wikipedia.org/wiki/Pa

[47]. *Bloom (shader effect)* [online] [cit. 2016-07-30]. Dostupné z: https://en.wikipedia.org/wiki/Bloom_(shader_effect).

[48]. HARGREAVES, Shawn. *Everything you ever wanted to know about IntermediateSerializer* [online]. 2008 [cit. 2016-07-30]. Dostupné z: https://blogs.msdn.microsoft.com/shawnhar/2008/08/12/everything-you-ever-wanted-to-know-about-intermediateserializer/.

[49]. *BADLOGIC GAMES: Box2D Platformer Character Controls* [online] [cit. 2016-07-18]. Dostupné z: http://www.badlogicgames.com/wordpress/?p=2017.

[50]. *NPC* [online] [cit. 2016-07-19]. Dostupné z: https://cs.wikipedia.org/wiki/NPC.

[51]. *Singleton pattern* [online] [cit. 2016-09-17]. Dostupné z: https://en.wikipedia.org/wiki/Si

[52]. *Dictionary – třída* [online] [cit. 2016-09-18]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/xfhwa508(v=vs.110).aspx.

[53]. *Decision tree* [online] [cit. 2016-11-19]. Dostupné z: https://en.wikipedia.org/wiki/Decisio

[54]. *Other Inside je česká logická plošinovka o vzpomínkách jednoho mima* [online] [cit. 2017-07-30]. Dostupné z: https://games.tiscali.cz/oznameni/other-inside-je-ceska-logicka-plosinovka-o-vzpominkach-jednoho-mima-284808.

[55]. *Česká hra Other Inside míří do Greenlightu* [online] [cit. 2017-07-30]. Dostupné z: http://www.czechgamer.com/24650/ceska-hra-other-inside-miri-do-greenlightu.html.

A Obsah přiloženého CD/DVD

bin/
: Engine spustitelný přímo z CD/DVD. Adresář obsahuje i všechny runtime knihovny a další soubory potřebné pro bezproblémový běh programu z CD/DVD.

doc/
: Text práce ve formátu PDF. A všechny soubory potřebné pro bezproblémové vygenerování PDF dokumentu textu (v ZIP archivu), tj. zdrojový text textu, vložené obrázky, apod.

src/
: Kompletní zdrojové texty enginu se všemi potřebnými (příp. převzatými) zdrojovými texty, knihovnami a dalšími soubory potřebnými pro bezproblémové vytvoření spustitelných verzí programu.

readme.txt
: Instrukce pro spuštění a kompilaci enginu, včetně všech požadavků pro jeho bezproblémový provoz.

U veškerých cizích převzatých materiálů obsažených na CD/DVD jejich zahrnutí dovolují podmínky pro jejich šíření nebo přiložený souhlas držitele copyrightu. Pro všechny použité (a citované) materiály, u kterých toto není splněno a nejsou tak obsaženy na CD/DVD, je uveden jejich zdroj (např. webová adresa) v bibliografii nebo textu práce nebo v souboru `readme.txt`.

B Zadání práce

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor:	**Bc. David Konečný**
Studium:	I1300386
Studijní program:	N1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název diplomové práce:	**Vývoj herního enginu pro 2D plošinovou hru**
Název diplomové práce AJ:	2D Game Engine Development
Garantující pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	Ing. Karel Petránek
Datum zadání závěrečné práce:	1.12.2013

FIM UHK
UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Rokitanského 62, 500 03 Hradec Králové, tel: 493 331 111, fax: 493 332 235

Zadání k závěrečné práci

Jméno a příjmení studenta: **David Konečný**

Obor studia: Aplikovaná informatika (2)
Jméno a příjmení vedoucího práce: **Karel Petránek**

Název práce:
Vývoj 2D herního enginu

Název práce v AJ:
2D Game Engine Development

Podtitul práce:

Podtitul práce v AJ:

Cíl práce: Porovnat existující přístupy k vývoji 2D herních enginů. Navrhnout vlastní řešení s ohledem na stabilitu, rozšířitelnost a využití hardwarové akcelerace. Diskutovat výhody a limitace navrženého enginu.

Osnova práce:

1) Seznámení s problematikou herních enginů a vývojem her

2) Průzkum existujících metod, technologií a frameworků a jejich popis

3) Volba a implementace metod

4) Testování metod, zhodnocení dosažených výsledků

Projednáno dne:

Podpis studenta Podpis vedoucího práce