VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

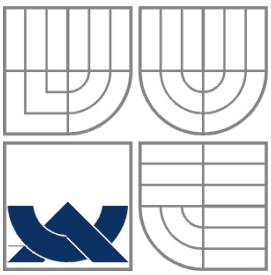
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

HERNÍ ENGINE PRO 2D HRY

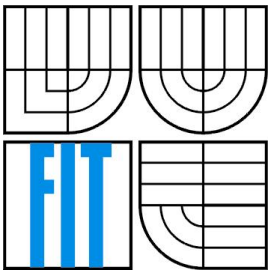
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JAKUB ŠTANGLICA
AUTHOR

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

HERNÍ ENGINE PRO 2D HRY

GAME ENGINE FOR 2D GAMES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAKUB ŠTANGLICA

VEDOUCÍ PRÁCE
SUPERVISOR

ING. MICHAL ZACHARIÁŠ

BRNO 2012

Abstrakt

Tato bakalářská práce se zabývá tvorbou 2D herního enginu pro Windows Phone 7. V první části práce je představen framework Microsoft XNA, který byl při tvorbě enginu použit, dále pak platforma Windows Phone 7 a specifika vývoje s tímto spojená. Také je obecně vysvětlen pojem herní engine. V dalších částech práce je podrobněji popsán návrh a implementace samotného enginu a jeho částí, a na závěr také vývoj demonstrační aplikace, dokazující použitelnost enginu, kterou je jednoduchá 2D hra, přičemž je kladen důraz právě na to, jak aplikace využívá možností tohoto enginu.

Abstract

This bachelor's thesis deals with development of 2D game engine for Windows Phone 7. In the first part of this thesis is introduced the Microsoft XNA framework, which was used for engine development, and Windows Phone 7 platform and some development specifics that goes with it. Also term game engine is briefly described. Other parts describe design and implementation of engine and its parts in more detail and finally development of demonstration application, simple 2D game, which should prove usability of engine.

Klíčová slova

Herní engine, XNA, Windows Phone 7, 2D hra

Keywords

Game engine, XNA, Windows Phone 7, 2D game

Citace

Jakub Štanglica: Herní engine pro 2D hry, bakalářská práce, Brno, FIT VUT v Brně, 2012

Herní engine pro 2D hry

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Michala Zachariáše. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jakub Štanglica
Datum 11. května 2012

Poděkování

Na tomto místě bych chtěl poděkovat Ing. Michalu Zachariášovi za jeho ochotný a vstřícný přístup a mnoho užitečných rad a připomínek při tvorbě této práce.

Obsah

1 Úvod

Tématem této bakalářské práce je vývoj 2D herního enginu pro Windows Phone 7 a demonstrační aplikace využívající tento engine. Toto téma jsem zvolil, protože jsem se chtěl věnovat některé z moderních mobilních platforem. O framework XNA jsem se zajímal už dříve a vím, že Microsoft poskytuje kvalitní dokumentaci a podporu vývojářům, proto jsem se rozhodl právě pro Windows Phone.

Hlavním cílem této práce bylo vytvořit herní engine, jež může být použit při vývoji 2D her pro systém Windows Phone 7, využívajících framework XNA. Dalším cílem pak je dokázat použitelnost enginu vytvořením jednoduché hry, která jej bude využívat.

V první kapitole bude nejprve představena mobilní platforma *Windows Phone 7* a její typické rysy. Řekneme si, jaká jsou specifika vývoje pro tuto platformu, a jaké technologie a postupy se zde používají.

Jako další si v kapitole *XNA* přiblížíme framework XNA, na němž je engine postaven. V této kapitole se dozvíte co to vlastně XNA je, k čemu všemu se využívá a jak pracuje. A to především z pohledu vývoje pro Windows Phone 7.

Následovat bude kapitola *Herní engine* věnující se herním enginům. Tento pojem bude podrobněji vysvětlen, a řekneme si, jak používání enginů ovlivňuje vývoj her. Také se podíváme na příklady současných herních enginů.

V nejobsáhlejší kapitole s názvem *Tvorba enginu* bude podrobně popsán navrh a implmentace enginu. Nejprve se budeme věnovat enginu jako celku a poté podrobně jednotlivým částem z nichž je sestaven. Také zde bude popsáno využití frameworku XNA Game Studio 4.0 na němž je tento engine postaven.

Poslední kapitola *Tvorba hry* je zaměřena na vysvětlení návrhu a implementace 2D hry, sloužící jako demonstrační aplikace, jež má prokázat použitelnost enginu. Touto hrou bude jednoduchá 2D akční plošinovka pro jednoho hráče, obsahující mimo jiné jednoduchou umělou inteligenci a systém power-upů. Stejně jako v předchozí kapitole, se i zde nejprve budeme věnovat hře jako celku a poté budou podrobně popsány jednotlivé části. Důraz bude kladen především na využití možností použitého enginu a demonstraci toho, jaké výhody použití enginu přineslo.

Po přečtení této práce by měl čtenář vědět co je, jaké části obsahuje a k čemu slouží herní engine a získat představu jak vývoj jednoduchého 2D enginu probíhá a jak se takový engine dá využít při tvorbě konkrétní hry. Také získá základní informace o paltformě Windows Phone 7 a frameworku Microsoft XNA.

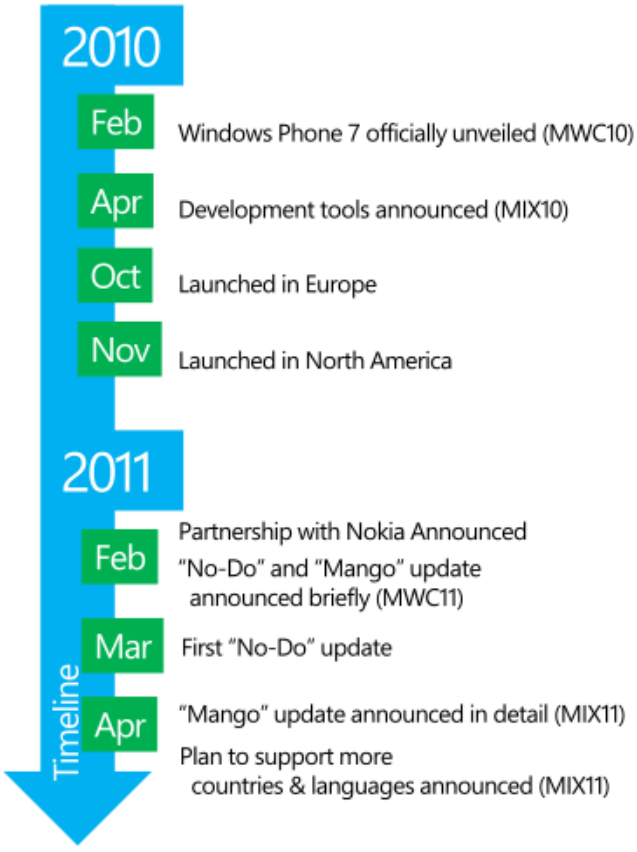
2 Windows Phone

Windows Phone 7 je operační systém vyvinut společností Microsoft pro takzvané chytré mobilní telefony. Na trh byl uveden ve druhé polovině roku 2010 jako nástupce předchozí mobilní platformy Microsoftu, kterou byl Windows Mobile. Na rozdíl od svého předchůdce, jež byl zaměřen především na potřeby podnikatelů, se však Windows Phone snažil oslovit i běžné uživatele, což se s rozšířením Windows Phone 7.5 stalo jeho hlavním cílem.

2.1 Historie

Práce na novém mobilním operačním systému byly zahájeny v roce 2008 a 15. února 2010 byl systém s názvem Windows Phone 7 oficiálně odhalen na konferenci v Barceloně. [1] Uvedení na trhy probíhalo postupně, koncem roku 2010 v Evropě a Severní Americe a na začátku roku 2011 v Asii. První velký update tohoto operačního systému s označením Windows Phone 7.5 byl vydán 27. září 2011 a přinesl až 500 nových funkcí a rozšíření oproti původní verzi. [2]

Původně bylo ohlášeno 10 zařízení s operačním systémem Windows Phone 7, a to od výrobců HTC, Samsung, LG a Dell. V únoru roku 2011 pak bylo ohlášeno nové partnerství mezi společnostmi Microsoft a Nokia, díky němuž se Windows Phone má stát hlavním operačním systémem pro chytré telefony Nokia. Zároveň má dojít ke sloučení služeb těchto společností, např. Nokia maps se spojí s Bing maps. [3]



Obrázek 2.1: Časová osa vývoje systému Windows Phone.

2.2 Hlavní rysy

2.2.1 Uživatelské rozhraní

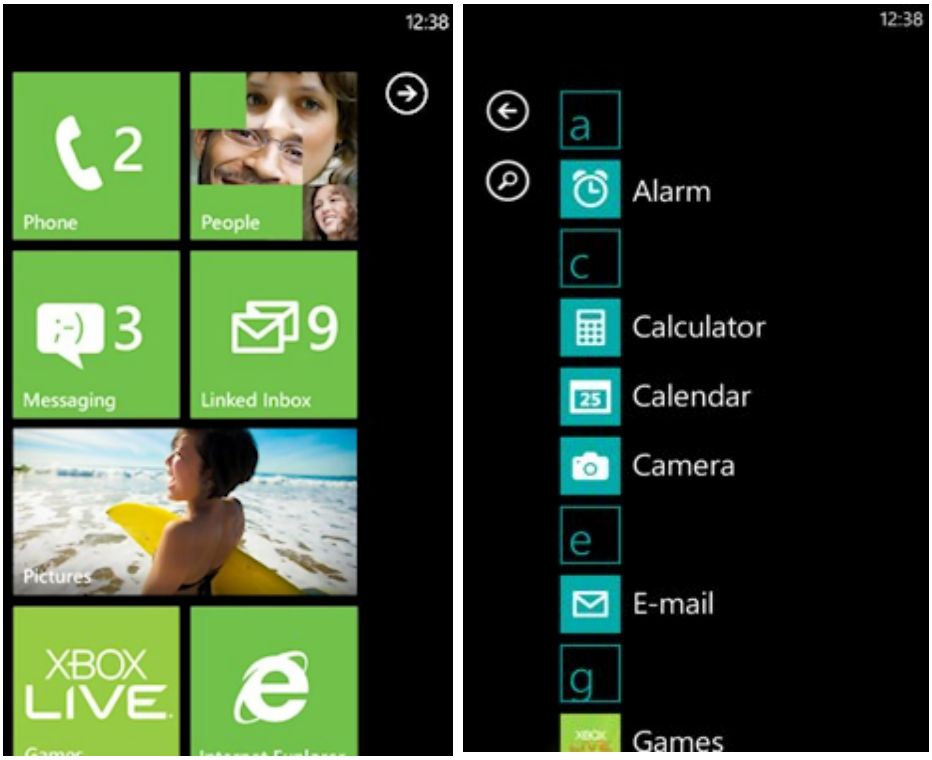
Windows Phone používá typický designový styl zvaný Metro, z něj vychází i většina prvků uživatelského rozhraní. Tento designový styl byl primárně vyvíjen právě pro systém Windows Phone a jeho hlavní principy jsou následující: [4]

- **Typografie** – Texty jsou nejen vizuálně atraktivní, ale také funkční a mohou pomáhat tvořit hierachrii obsahu.
- **Pohyb** – Velký důraz je kladen na vlastní animace a přechody.
- **Obsah** – V Metru se nachází minimum nadbytečných dalších prvků a obsah se tak stává středobodem rozhraní.
- **Autentičnost** – Díky rychlosti a jednoduchosti se uživatelům dotekových zařízení snaží přinést autentický digitální zážitek.

Zajímavá je hlavní obrazovka zvaná Start Screen, která se skládá z takzvaných Live Tiles dlaždic, což jsou jakási spojení s aplikacemi, kontakty, webovými stránkami apod. Tyto dlaždice se aktualizují v reálném čase.

Některé služby se sdružují do takzvaných *hubů,* kde je možné kombinovat lokální i on-line obsah, například mezi obrázky najdeme jak fotky z fotoaprátu telefonu, tak alba z Facebooku. Další takovéto *huby* jsou třeba multimédia (Zune) nebo hry.

Co se týká vlastního uživatelského vstupu se Windows Phone příliš neliší od jiných mobilních systémů, textový vstup je zajištěn pomocí virtuální klávesnice a ovládání je dotykové, s podporou technologie multi-touch.



Obrázek 2.2: Úvodní obrazovka Start Screen s typickými dlaždicemi(vlevo) a menu aplikací(vpravo).

2.2.2 Hry a Marketplace

Vzhledem k tématu této bakalářské práce se budeme oblasti her na Windows Phone věnovat trochu více. Hry jsou sdružovány do *Games hubu,* a spadají pod Xbox Live. Díky tomu můžeme i na telefonech najít některé fukce známé z konzole Xbox 360. Uživatelé Windows Phone jsou také plnohodnotnou součástí Xbox Live komunity. Velkým omezením v oblasti her je v současné době nemožnost real-time multiplayeru, tato možnost má však být přidána v budoucnu.

Distribuci her, stejně jako ostatních aplikací zajišťuje Windows Phone Marketplace, k němuž mohou uživatelé přistupovat pomocí *Marketplace hubu*. Marketplace slouží především k distribuci aplikací vyvíjených třetími stranami – vývojaři a společnostmi nezávislými na Microsoftu. Aplikace může vyvíjet, spravovat a prodávat každý registrovaný vývojář, než je však aplikace uvolněna k prodeji na Marketplace, musí nejprve projít schvalovacím procesem Microsoftu.

Obrázek 2.3: Windows Phone Marketplace.

2.3 Vývoj aplikací pro Windows Phone 7

Všechny aplikace pro Windows Phone musí být založeny na XNA nebo Siverlight (zvláštní verzi pro Windows Phone) nad .NET Compact Framework. Speciálně pro vývoj mobilních aplikací Microsoft vydal rozšíření pro Visual Studio 2010, a jeho Express verzi, zvané Windows Phone SDK. Součástí vývojářského balíku je například i Windows Phone Emulator, umožňující testovat aplikace i vývojářům, kteří nevlastní zařízení Windows Phone. Pro vývoj uživatelských rozhraní Microsoft zdarma nabízí nástroj Expression Blend. [5]

Microsoft také provozuje portál pro komunitu Windows Phone a Xbox 360 vývojářů zvaný App Hub. Zde je možné získat vývojové nástroje, registrovat se jako vývojář, umisťovat své aplikace na Marketplace atd. Můžeme tam také najít kompletní dokumentaci k vývojovým prostředkům, mnoho návodů a tutoriálů pro začínající vývojáře a v neposlední řadě také fórum, kde je možné diskutovat věci spojené s vývojem aplikací s ostatními vývojáři.

Vývoj Windows Phone aplikací se dělí na dva hlavní směry, vývoj běžných aplikací a vývoj her. Každý druh aplikací se primárně vyvíjí pomocí jiného frameworku, Silverlight pro normální aplikace, a pro hry je to framework XNA. Také je možné tyto dva přístupy do jisté míry kombinovat.

3 XNA

Microsoft XNA je vývojářská platforma od firmy Microsoft, jedná se o sadu knihoven a nástrojů pro tvorbu her. XNA je nádstavbou frameworku .NET Compact Framework pro Xbox 360 a .NET Framework pro PC, běží nad verzí CLR mírně upravenou pro potřeby her, která je dostupná na systémech Windows XP, Windows Vista, Windows 7, Windows Phone 7 a Xbox 360. Ačkoliv je teoreticky možné programovat v jakémkoliv jazyce kompatibilním s .NET frameworkem, tak oficiálně je podporován pouze jazyk C#. XNA spolupracuje s množstvím dalších nástrojů z oblasti vývoje her, například XACT a obsahuje rozsáhlou podporu pro vývoj 2D a 3D her. [6] [7]

Framework XNA zapouzdřuje nizkoúrovňové technické aspekty vývoje a díky tomu také pokrývá rozdíly mezi různými platformami, což usnadňuje meziplatformní přenos her. Ačkoliv by předchozí věta mohla naznačovat opak, XNA jako takové ještě nelze považovat za herní engine, jeho knihovny a třídy jsou spíše obecného charakteru, aby je bylo možné využít v co nejširším spektru projektů. Spíše by se tedy dalo říct, že se jedná o jakýsi obecný základ na němž lze engine, nebo přímo hru, vystavět.

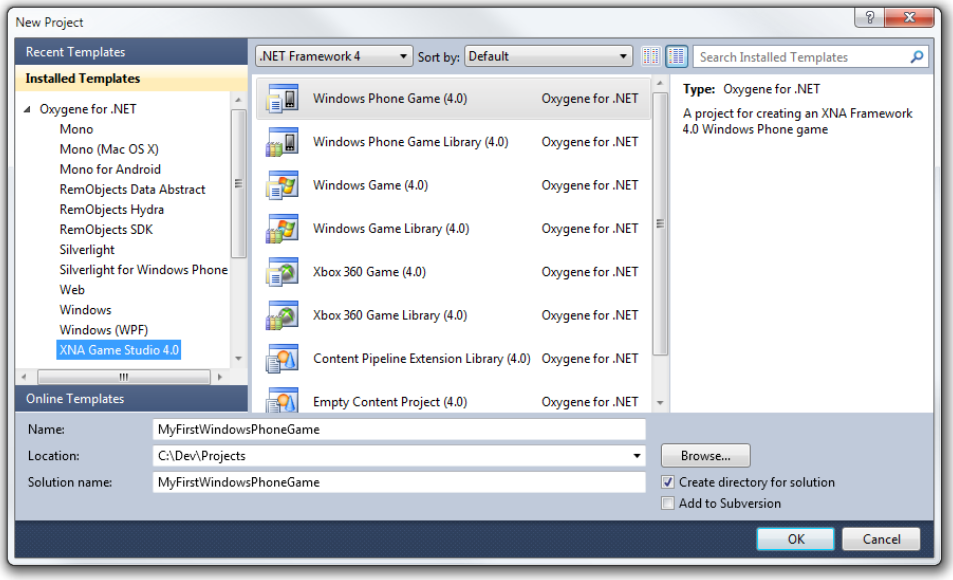
3.1 XNA Game Studio

XNA Game Studio je vývojové prostředí určené pro vývoj her nad frameworkem XNA. Je to rozšíření IDE Microsoft Visual Studio a je možné jej zdarma získat na stránkach společnosti Microsoft. Zatím byly vydány čtyři verze XNA Game Studio, přičemž některé z nich se dočkali takzvaného refreshe.

První verze byla vydána koncem roku 2006, byla zaměřena především na nezávislé vývojáře a studenty. Poskytovala takzvané starter kity pro rychlý vývoj her klasických žánrů jako FPS a strategie na platformy Xbox 360 a Windows. Tato verze, stejně jako následující, staví na DirectX 9. Oficiálním jazykem Game Studia je od počátku C#. [8]

Téměř přesně o rok později bylo uvedeno XNA Game Studio 2.0, spolupracující s Visual Studiem 2005. Jeho nejvýznamnějším rozšířením bylo síťové API přinášející možnost tvorby multiplayerových her. [9]

XNA game Studio 3.0 pro Visual Studio 2008 přineslo rozšíření možnotí využívat prvků Xbox Live Community a hlavně umožnílo vývoj pro další platformu, kterou je Microsoft Zune. [10]

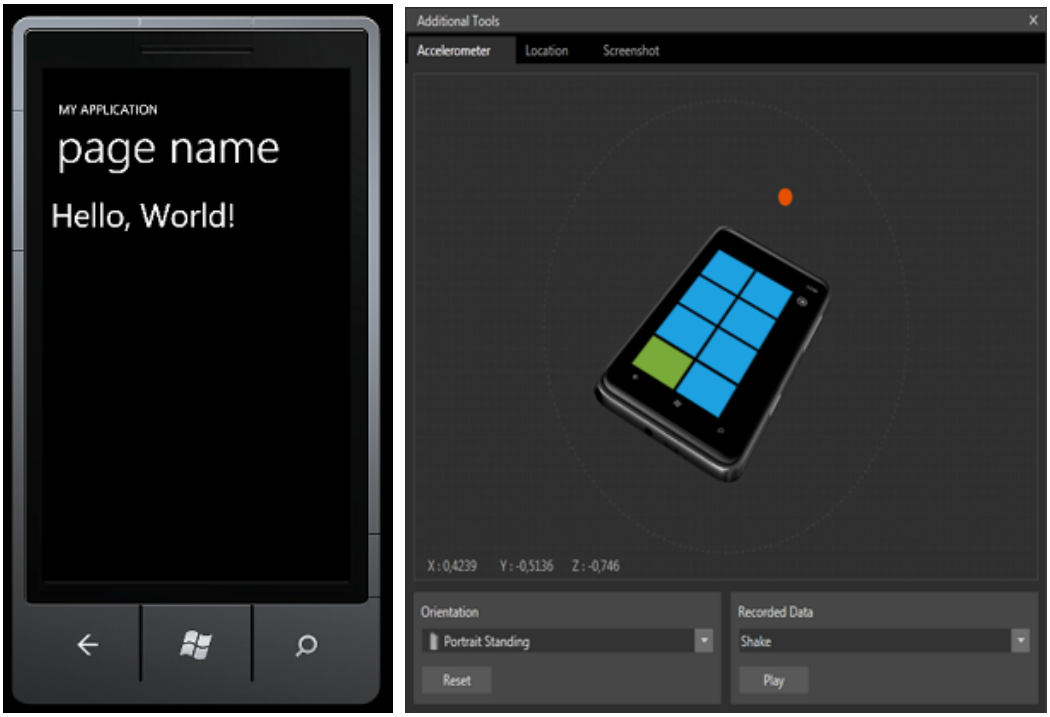


Obrázek 3.1: XNA Game Studio.

3.1.1 XNA Game Studio 4.0

Z pohledu vývoje pro Windows Phone, a tudíž i této bakalářské práce, je nejvýznamnější právě zatím poslední čtvrtá, verze Game Studia, která byla vydána 16. září 2010. Od této verze je možno vyvíjet hry i pro platformu Windows Phone a to včetně využití multi-touch ovládání nebo třeba hardwarové 3D akcelerace. Součástí Game Studia 4.0 je i Windows Phone Emulator, zmíněný v předchozí kapitole. Z dalších vlastností můžeme zmínit třeba podporu práce s XML soubory a jejich využití pro ukládání herních dat nebo sjednocené API pro vstupy různých platforem. [11]

XNA Game Studio 4.0 se dočkalo i refresh verze, která přinesla například podporu dalšího jazyka, Visual Basic, nebo možnost interoperability mezi XNA a Silverlight. Od této verze je také možno využívat síťovou komunikaci přes TCP/UDP sockety. Vylepšení se dočkal i Windows Phone Emulator, jeho funkce byly rozšířeny o simulaci akcelerometru nebo GPS. [12]



Obrázek 3.2: Windows Phone Emulator.

3.2 Vývoj her s XNA

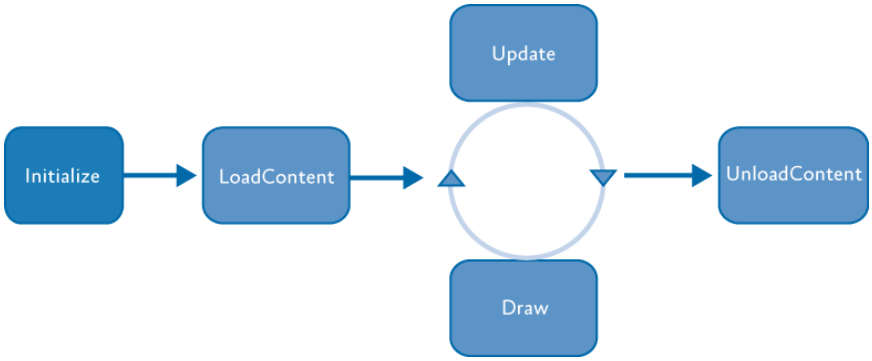
3.2.1 Struktura

Hry postavené na XNA jsou typické svou strukturou, která je tvořena několika základními metodami definovanými ve třídě Game, což je jakási hlavní třída představující prostředí hry. Jedná se o následující metody: [13]

- **Initialize** – Tato metoda slouží k inicializaci hry, například nahrání různých (negrafických) zdrojů, přihlášení se k používání služeb apod.
- **LoadContent** – Volána z metody Initialize, slouží tato metoda k načtení grafických zdrojů hry. Kromě toho je volána pokaždé, když potřebujeme načíst grafické zdroje, třeba po resetu grafického zařízení.

- **Update** – Toto je metoda v níž se odehrává logika hry, spolu s metodou Draw jsou volány v nekonečné smyčce až do ukončení hry.
- **Draw** – Metoda sloužící k vykreslení obrazu.
- **UnloadContent** – Je to opak metody LoadContent, volá se vždy, když je potřeba uvolnit grafické zdroje, typicky při ukončení hry.

Metody jsou volány v daném pořadí přičemž metody Update a Draw tvoří typickou smyčku jakožto jádro hry. Tato základní struktura stačí ke kompletní implementaci jednoduchých her a i u složitějších projektů tvoří jakousi základní osu.



Obrázek 3.3: XNA herní smyčka.

3.2.2 Herní komponenty a služby

Významnou součásti XNA je možnost tvorby takzvaných herních komponent, ty mohou představovat jakési moduly zajišťující nějakou funkcionalitu potřebnou pro vytvářenou hru, jako například správce uživatelského rozhraní. Komponenty lze navíc snadno přenést do dalších projektů a tak přispívají k zvýšení znovupoužitelnosti kódu.

Herní komponenta je třída, která dědí z jedné z XNA tříd: GameComponent nebo DrawableGameComponent. V případě DrawableGameComponent je navíc přidána možnost, aby komponenta přímo vykreslovala do obrazu, jako například v případě výše zmíněné komponenty pro uživatelské rozhraní. Tyto třídy obsahují stejné metody jako třída Game1, které jsou po zaregistrování komponenty automaticky volány. [14]

Další významnou částí jsou služby. Jedná se o rozhraní třídy, které může tato třída nabízet ostatním. Hlavní třída Game tvoří prostředníka, u nějž se služby registrují a kterákoliv jiná třída může naopak třídu Game požádat o poskytnutí některé z regitsrovaných služeb bez jakékoliv znalosti třídy jež službu poskytuje. Dalo by se tedy říci, že služby poskytují mechanismus komunikace mezi různými třídami a komponentami. [15]

3.2.3 XNA a Windows Phone

Vzhledem k rozdílům mezi zařízeními jako PC/Windows a Xbox 360 oproti Windows Phone je nasnadě otázka, jak se liší vývoj pro mobilní zařízení oproti těmto „velkým“ platformám. Odpověď může být překvapivá, ale moc rozdílů nebo omezení zde není.

Nejvýznamější rozdíly plynou ze samotného faktu, že se jedná o mobilní zařízení. Rozdílem je především omezení rozlišení, které v současné době může být pouze 800x480 pixelů a také metody vstupu. Jako další bych zde zmínil to, že v současné době Windows Phone nepodporuje programovatelné shadery, náhradu za ně mají poskytovat konfigurovatelné efekty, které jsou optimalizované pro běh na mobilním GPU. [16]

4 Herní engine

Pojmem herní engine označujeme soubor nástrojů pro usnadnění vývoje her. Herní engine většinou poskytuje funkce pro obstarávání základních částí většiny her jako například fyzika, vstupy od uživatele, zvuk, umělá inteligence, správa scény, renderer a tak dále. Engine může také poskytovat určité šablony pro často používané objekty (záleží na typu hry). [17]

Z těchto částí se skládá téměř každá hra a bylo by samozřejmě nevýhodné a značně časově náročné je pro každou novou hru tvořit znovu, proto se používají herní enginy, které je možno znovupoužít při vývoji mnoha různých her a tak šetří vývojářům čas a práci.

Herních enginů existuje velké množství, liší se v rozsahu poskytovaných funkcí, zaměření na určitý žánr her nebo platformu a v neposlední řadě také v ceně. Engine může být vytvořen pro vlastní potřebu a usnadnění práce jeho vývojářů jako ten, kterým se zabývá tato bakalářská práce. Může se ale také jednat o velmi rozsáhlé komerční dílo a cenný ochodní artikl, generující jeho tvurcům zisk v miliónech dolarů jako třeba v případě Unreal Enginu.

Obrázek 4.1: Obrázek ze hry Doom používající engine id Tech 1.

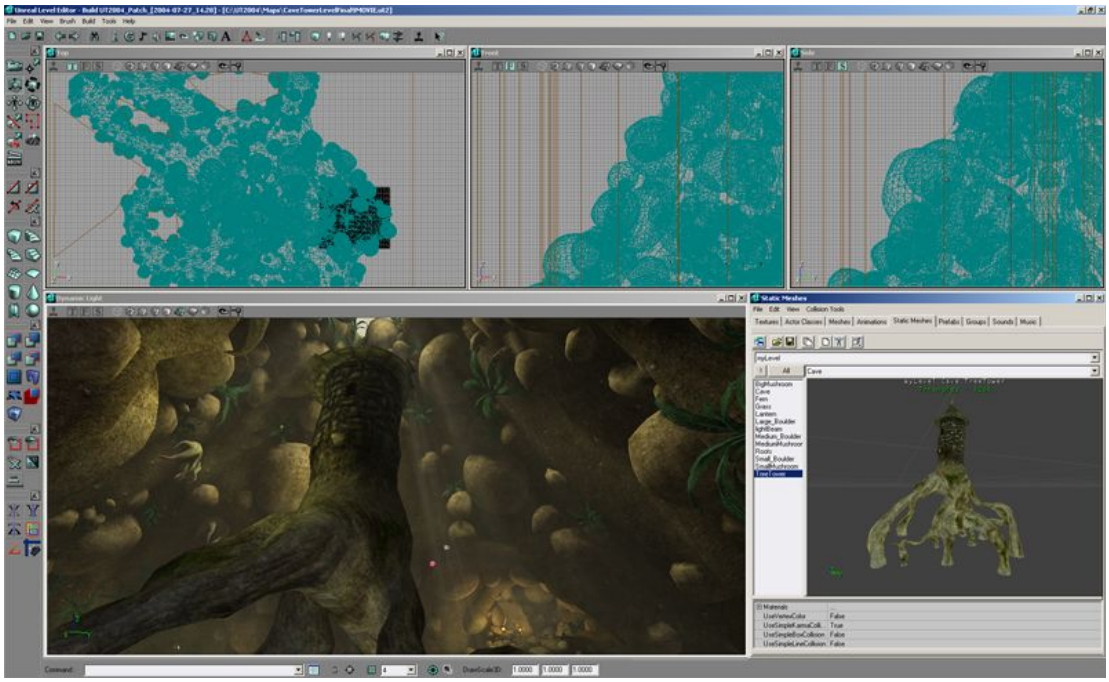
4.1 Unreal Engine 3

Jako příklad moderního herního enginu může sloužit Unreal Engine 3, který je pokračovatelem už zmíněného Unreal Engine od Epic Games. Tento engine umožňuje vývoj pro široké spektrum platforem, od konzolí jako Xbox 360 a Playstation 3 přes například Android, iOS nebo Mac OS X až po PC (Windows, Linux).

Jeho renderer může zobrazovat jak s OpenGL tak DirectX (9 a vyšší), přičemž podporuje nejmodernější technologie jako programovatelné shadery (3.0), dynamické stínování, HDRR nebo per-pixel osvětlení. Zatímco jádro enginu je napsáno v C++, většina ostatních součástí je napsána ve speciálním jazyce UnrealScript což při různých úpravách snižuje nutnost zásahu do jádra na minimum.

Unreal Engine 3 také obsahje velké množství podpůrných nástrojů jako například level editor UnrealEd, Sound Cue editor pro práci se zvukem, UnrealCascade pro částicové systémy a mnohé další.

Mezi hry používající Unreal Engine 3 patří třeba Unreal Tournamet 3, série Gears of War nebo Bioshock. Kromě FPS her však našel uplatnění třeba v MMORPG TERA: The Exiled Realm of Arborea. [18]

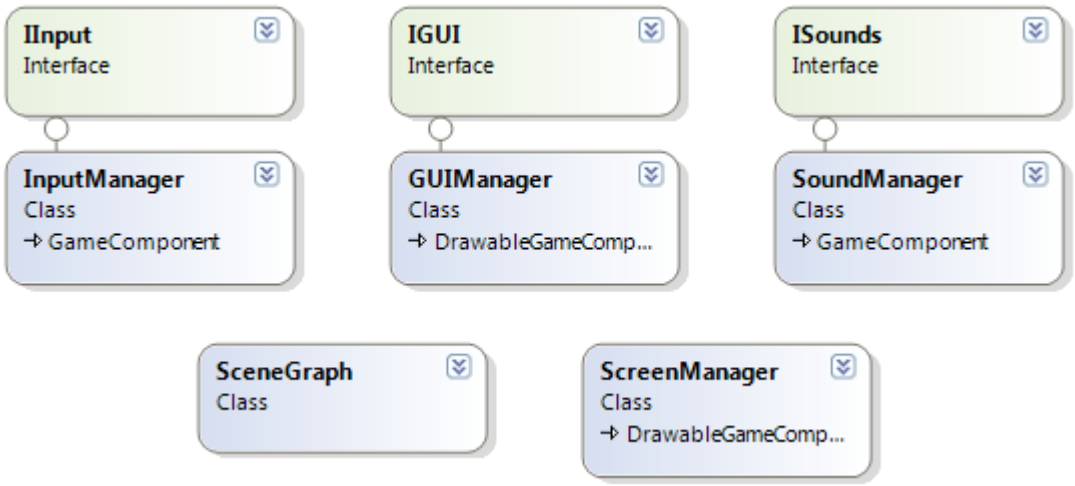


Obrázek 4.2: Nástroj UnrealEd pro tvorbu levelů.

5 Tvorba enginu

Vytvoření 2D herního enginu bylo hlavním cílem této práce, engine byl tvořen jako knihovna pro framework XNA, poskytující základní funkce pro vývojáře 2D her. Skládá se z několika částí poskytujících funkce pro specifické oblasti vývoje her. Protože je tento engine určen pro vývoj her na platformu Windows Phone 7, je založen na frameworku XNA a psán v jazyce C#. To, že základem je framework XNA, tvorbu enginu dosti usnadnilo. Díky využití XNA nebylo třeba pracovat přímo s nejnižšími vstrvami nad hardware a řešit problémy a optimalizace s tímto spojené, tak jako tomu je u většiny enginů pro jiné platformy. Místo toho tento engine zcela využívá abstrakci, poskytovanou frameworkem XNA, stejně jako jeho správu zdrojů a další funkce. Tento engine zde tedy spíše působí jako další vrstva nad XNA poskytující konkrétní, pro vývoj her potřebné a často používané funkce.

Před začátkem vývoje bylo důležité stanovit si rozsah poskytované funkcionality, ačkoliv je engine primárně určen především pro hry typu akční plošinovky (angl. Platformer), snažil jsem se při vývoji postupovat tak, aby byla zachována co největší generičnost a engine mohl být uplatněn i při vývoji jiných typů 2D her. Samotný engine se skládá z několika logických částí, jež pokrývají specifické oblasti vývoje her. Těmito částmi jsou správa zvuku, správa ovládání, dále vykreslování a s tím spojený především graf scény, základní fyzika, game state management a také jednoduché uživatelské rozhraní. Téměř všechny tyto části je možno použít pro libovolný typ 2D hry. V porovnání s jinými herními enginy je možno říci, že co se týká hlavní funkcionality pokrývající grafiku, zvuk a ovládání, zde popsaný engine obsahuje všechny tyto oblasti. Stejně tak game state management bývá častou součástí enginů, protože se bez něj většina her neobejde, v jiných enginech je možno také často nalézt nějakou část zabývající se tvorbou levelu. Tuto oblast jsem se rozhodl ve svém enginu nepokrývat, protože by to bylo na úkor generičnosti, nicméně jednoduchý systém tvorby levelu jsem implementoval v rámci demonstrační hry (Kapitola 6). Jako další častou část enginu v rámci správy scény můžeme zmínit také kameru, tu jsem ze stejného důvodu také nezahrnul přímo do enginu ale implementoval ji až v rámci hry.



Obrázek 5.1: Přehled součástí enginu.

Jednotlivé části enginu jsem se rozhodl implementovat jako herní komponenty, případně vykresovatelné herní komponenty, což znamená, že dědí z příslušných tříd obsažených v XNA frameworku. Toto řešení zajišťuje, že jednotlivé části běží nezávisle na sobě, na pozadí samotné hry a mohou tedy být kdykoliv přístupné z libovolného místa aplikace, což je například v případě správy uživatelských vstupů zcela nezbytné. Vyjímkou z tohoto přístupu je správce uživatelského rozhraní, který využívá ke své prácí správce uživatelských vstupů, bez nějž není schopen fungovat.

Většina těchto komponent je přístupna přes rozhraní, které nabízí v podobě služeb a jsou tak snadno dostupné pro každou třídu s přístupem k hlavní třídě Game, u níž jsou tyto služby

zaregistrovány. Toto řešení mimo jiné zvyšuje znovupoužitelnost jednostlivých částí, a také usnadňuje vzájemnou komunikaci mezi těmi částmi enginu, které to vyžadují. Tento přístup by měl také vývojářum umožnit například použití jen některých částí tohoto enginu a pro pokrytí ostaních oblastí si vytvořit vlastní komponenty nebo využít jiné moduly, takže mají větší volnost a nejsou nuceni se spolehnout pouze na funkce enginu.

Vyjímkou, pro niž není implmentováno rozhraní, je část starající se o game state management, pro ten jsem se rozhodl rozhraní neimplementovat z důvodu, že se jedná o jakýsi základ hry, v rámci nějž by měly běžet všechny části hry a proto mi zde přijde užší provázání na místě. Navíc možnosti interakce s Game state managerem jsou dosti omezené, této části se budu blíže věnovat v kapitole 5.3. Také část starající se o správu scény a jeji vykreslování neposkytuje rozhraní a navíc není ani vytvořena jako komponenta. Rozhodl jsem se zde takto postutpovat proto, že tato část je použita z její podstaty pouze na jedné herní obrazovce.

V pozdější fázi vývoje jsem sice přišel na to, že by bylo možné a možná i výhodnější i tuto část implementovat jako komponentu poskytující služby, bohužel se tak ale stalo až ve chvíli, kdy byl vývoj zcela u konce a bylo by třeba předělat podstatnou část projektu. To by bylo příliš náročné, především z časových důvodů.

5.1 Správce uživatelských vstupů

V této části bude popsán správce uživatelských vstupů. Oblast ovládání a vstupů od uživatele je poměrně dobře pokryta už v XNA frameworku, čehož jsem se snažil v co největší míře využít při tvorbě této části a vytvořit spíše jakousi nádstavbu zjednodušující práci se vstupy a poskytující funkce zaměřené přímo na využití této oblasti při tvorbě her.

Vzhledem k tomu, že engine je určen pro zařízení Windows Phone, tak hlavními způsoby ovládání jsou především dotykový display (touchpanel), a ovládací tlačítka, z těch se však pro potřeby her, a aplikací obecně, většinou používá pouze tlačítko Zpět. Zařízení Windows Phone poskytuje i jiné možnosti vstupu od uživatele jako obraz (prostřednictvím fotoaparátu či kamery) nebo zvuk (mikrofon), tyto typy vstupu se však v běžných hrách využívají velmi zřídka a proto jsem se rozhodl je v tomto jednoduchém enginu nepodporovat.

5.1.1 Implementace

Správce uživatelských vstupů je reprezentován třídou *InputManager* a je vytvořen jako herní komponenta. K jeho funkcím je přistupováno přes rozhraní *IInput*, které nabízí formou služby.

Protože je zaměřen na ovládací prvky touchpanel a tlačítka, jsou v tomto správci uchovávány informace o dotycích a gestech detekovaných na dotykovém displayi zařízení a informace o stavu tlačítek zařízení. Detekovaná gesta jsou uchovávána v seznamu `Gestures` k němuž je možno také přistupovat přes rozhraní třídy, a tak umožňuje vývojáři zjistit, jaká gesta byla detekována. V samotném modulu se s nimi už dále nepracuje, jsou uchovávana pouze z důvodu úplnosti poskytovaných informací vývojáři, který je může vyžít pro vlastní účely.

Z pohledu této části nejdůležitějším ovládacím prvkem jsou doteky, ty jsou uchovávány v seznamu `Touches` typu TouchCollection. Datový typ TouchCollection je součástí XNA a reprezentuje množinu doteků na touchpanelu. I tyto informace jsou pro potřeby vývojáře přístupny přes rozhraní *IInput*. Informace o detekovaných dotecích jsou však na rozdíl od výše zmíněných gest využity i dále v tomto modulu, kdy slouží k určení stavu touchpanelu a tak umožňují rozpoznat akci uživatele, například zda pouze někam klikl, nebo zda stále ještě drží prst na touchpanelu.

Třetím prvkem, o němž uchovává správce informace, je už zmíněné tlačítko Zpět, pro tuto činnost je možné zcela využít XNA framework, a tak jediným rozšířením v této oblasti je, že je uchováván aktuální i minulý stav tlačítka, čehož je využito pro další funkce správce, které budou popsány dále.

Správce vstupu detekuje stav touchpanelu a tlačítek v každém cyklu hlavní herní smyčky, klíčový kód se tedy nachází v metodě `Update()`, jež je automaticky volána z hlavní smyčky hry v rámci obsluhy herních komponent. Na začátku každého cyklu jsou vynulovány a aktualizovány informace o dotecích a gestech, poté se uloží poslední platný dotek a poslední stav tlačítka Zpět do příslušných proměných a nakonec se zjistí aktuální stav jak tlačítka Zpět, tak doteku na touchpanelu.

V puvodním návrhu jsem počítal s řešením třístavovou detekcí, od které jsem si sliboval snažší odlišení doteku od držení. Bohužel však nejen, že tento způsob nepřispěl k lepšímu řešení problému, ale celkově pro potřeby správce vstupů přinášel více komplikací. Rozhodl jsem se tedy nakonec vrátit k dvoustavové detekci, která funguje dobře.

Ke zjištění stavu tlačítka Zpět je použita metoda `Gamepad.GetState()`, ta je součástí modulu Input XNA frameworku. Zjištění stavu doteku probíhá vybráním doteku z kolekce detekovaných doteků zjištěných na začátku obslužné smyčky, pro zjednodušení je detekován pouze jeden, první, dotek. Ačkoliv by se mohlo zdát, že toto znemožňuje multitouch chování a tedy omezuje uživatele tím, že nebude například možno používat dvě virtuální tlačítka naráz, není tomu tak. Smyčka se oproti uživatelovu chování aktualizuje velmi rychle a je tak možno detekovat i více doteků aniž by uživatel pocítil nějaké omezení.

Třída *InputManager* obsahuje také metody pro jednoduchou detekci uživatelových základních akcí. Sem patří metoda `BackPressed()`, která na základě aktuálního a předchozího stavu určuje zda uživatel stiskl tlačítko Zpět a metody `TouchTap()` a `TouchHold()`, které také na základě aktuálního a minule detekovaného stavu touchpanelu určují, zda uživatel klikl nebo stále drží stisklý prst na touchpanelu.

```
public bool BackPressed()
{
    return (ButtonActual.IsButtonDown(Buttons.Back) &&
                ButtonLast.IsButtonUp(Buttons.Back));
}
```

Zdrojový kód 5.1: Implementace medody BackPressed.

Tento modul plní spíše podpůrnou funkci a jeho hlavní využití je především jako podpora modulu spravujícího uživatelské rozhraní, je však volně přístupný a proto může být libovolně využit vývojářem. Při návrhu jsem však počítal, že při vývoji bude využito především detekce stisknutí tlačítka Zpět, zatímco v ostatních případech se vývojář v otázce ovládání spolehne z větší části na využití správce uživatelského rozhraní.

5.2 Správce zvuku

Zvuková stránka bývá také podstatnou součástí většiny her, protože se jedná o dobrý způsob dokreslení atmosféry, a tak nějakou hudbu a zvukové efekty můžeme nalézt snad v každé hře nehledě na platformu či žánr. Z toho důvodu se podpora zvukové stránky vyskytuje i v herních enginech a ani tento engine není vyjímkou.

I v oblasti zvuků poskytuje XNA dobrý základ, kterého jsem při tvorbě správce zvuku využil. Hlavním cílem v této části bylo zapouzdřit tyto funkce v XNA už obsažené a usnadnit přímočařejší používání zvukových zdrojů, především se zaměřením na použití ve hrách.

5.2.1 Implementace

Správce zvuku, který představuje třída *SoundManager*, uchovává zvukové efekty a hudbu hrající na pozadí herních obrazovek. K načtení těchto zdrojů využívá ContentManager, který je součástí XNA frameworku. I tato třída je implementována jako herní komponenta.

S prací se zvukem na zařízení Windows Phone 7 jsou však spjaty některé požadavky Microsoftu, ty musí být splněny, aby aplikace získala certifikát a mohla být distribuována na Marketplace. [19] Aby mohl být engine využíván pro tvorbu takových aplikací bylo tedy důležité aby tyto podmínky práce se zvukem splňoval.

Hudební stopa je uchovávána v proměnné `BgMusic`, která je typu Song. V tomto enginu jsem se rozhodl uchovávat pouze jednu píseň současně, což by podle mého názoru mělo být pro tvorbu jednoduchých her dostačující. Pro přehrávání hudby velmi dobře posloužila třída MediaPlayer, která je součástí XNA frameworku a je určena právě pro tyto účely.

Zvukové efekty jsou pak uchovávány v kolekci typu Dictionary a to jako dvojice název efektu, jako string a samotný zvukový efekt, který je reprezentován instancí třídy SoundEffect, jež je opět součástí XNA frameworku. Tento přístup jsem zvolil pro snažší práci a orientaci ve zvukových efektech, jakmile je tedy efekt přidán se zvoleným názvem, tak se s ním dále pracuje na základě tohoto názvu.

Správce zvuku samozřejmě umožňuje zakázat jak přehrávání hudby, tak i zvukových efektů a to nezávisle na sobě. Také je v něm možno měnit hlastitost přehrávání hudby, tyto možnosti mimo jiné zajišťují spnění jedné z certifikačních podmínek a to podmínku 6.5.2 v [19]. Stav povolení k přehrávání hudby je reprezentován proměnou `musicEnabled` a stav povolení zvukových efektů proměnou `soundsEnabled`, jež jsou obě typu bool. Nastavování hlasitosti přehrávání hudby je zajištěno přímo manipulací s třídou MediaPlayer.

Důležitou funkci plní příznak `mediaplayerEnabled`, který určuje, zda není MediaPlayer využíván systémem, jeho hodnota je nastavena zavoláním metody `IsMPAvaible()`. Tento příznak je pak kontrolován v každé metodě pracující s MediaPlayerem a pokud se zjistí, že aplikace nemá oprávnění s ním manipulovat, akce se neprovede. Tím je zajištěno splnění certifikační podmínky 6.5.1 v [19].

K přidávání zvukových efektů slouží metoda `AddSound()`. Ta má dva parametry, název pod kterým bude efekt uložen a název zdrojového souboru z nejž ContentManager tento efekt načte a uloží jej do kolekce zvukových efektů k příslušnému názvu. Podobně pracuje i metoda `LoadBackgroundMusic()` jež slouží k načtení hudby, ta příjmá název zdrojového souboru jako svůj jediný parametr. V obou těchto metodách jsou také odchytávány případné výjimky, které jsou poté předány dále.

Přehrání zvukového efektu zajišťuje metoda `PlaySound()`, jíž je jako parametr předán název zvukového efektu, který má být přehrán. V rámci této metody se také kontroluje, zda je přehrávání zvukových efektů povoleno, a pokud není, tak efekt samozřejmě není přehrán. Tato metoda je schopna generovat výjimku typu KeyNotFoundException v případě, že zadaný název efektu neexistuje.

```
SoundEffect sound;
if (Sounds.TryGetValue(name, out sound) == true)
      sound.Play();
else
      throw new KeyNotFoundException("There is no Sound Effect
with that name!");
```

Zdrojový kód 5.2: Kód pro přehrání zvukového efektu.

Dale je zde několik metod pro ovládání přehrávání hudby, které zajišťují základní akce jako pozastavení přehrávání, zastavení, zahájení a pokračování v přehrávání. Tyto metody pouze volají příslušné medoty třídy MediaPlayer, přičemž však zohledňují nastavení správce zvuku jako povolení přehrávání hudby a oprávnění přistupovat k MediaPlayeru. Metoda `PlayBackgroundMusic()` slouží k zahájení přehrávání. Kromě této akce ještě navíc automaticky nastavuje, aby se hudba přehrávala ve smyčce stále dokola.

Mezi metody zajišťující ovládací funkce patří ještě metody mající na starost hlasitost přehrávané hudby. Sem patří metoda `GetMusicVolume()`, která vrací aktuální nastavenou

hodnotu hlasitosti získanou přímo ze třídy MediaPlayer. Tato hodnota je v rozmezí 0-1, později byla metoda upravena tak, že tuto hodnotu vrací pouze pokud je povoleno přehrávání hudby a MediaPlayer je aplikaci k dispozici, pokud jsou tyto podmínky porušeny vrací vždy hodnotu 0. K této změně jsem přistoupil především, aby nedocházelo ke zbytečnému zmatení uživatele protichůdnými informacemi. Pak je zde metoda pro vlastní nastavení hlasitosti, jedná se o metodu `AdjustBackgroundMusicVolume()`, ta opět nastavuje přímo hodnotu hlasitosti v MediaPlayeru, ovšem pouze v případě splnění všech výše popsaných podmínek.

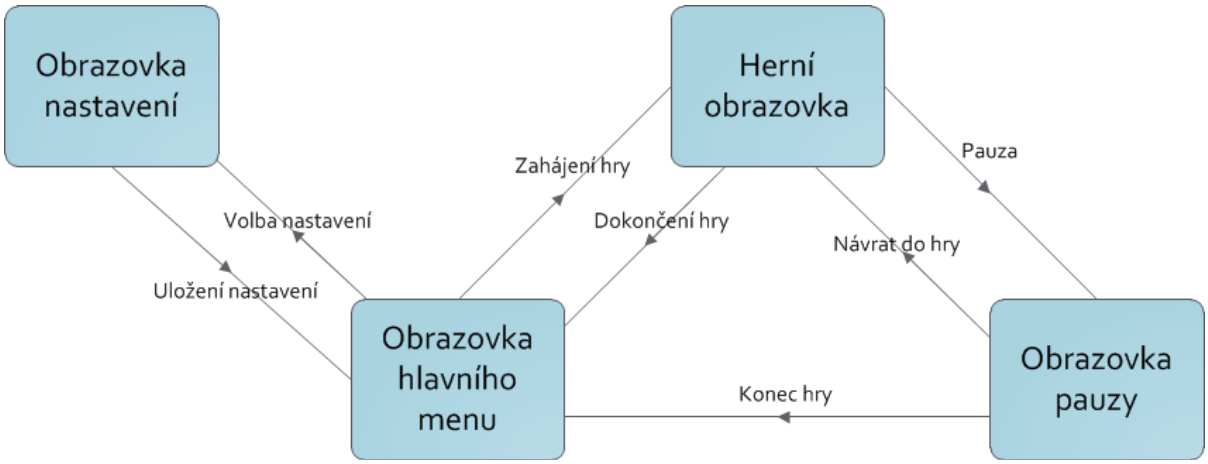
Ke zjištění, zda je povoleno přehrávání hudby, slouží metoda `GetMusicEnabled()` a ke stejnému účelu, pouze týkající se zvukových efektů, je metoda `GetSoundsEnabled()`. V původním návrhu pro povolení a zakázání přehrávání hudby sloužila jediná metoda, při jejímž zavolání se změnil stav povolení na opačnou hodnotu, a podobná metoda plnila stejnou funkci při manipulaci s povolením zvukových efektů. Při používání enginu se však ukázalo, že toto řešení není příliš šťastné, a tak jsem do tohoto modulu doimplementoval zvláštní oddělené metody pro povolení a zakázání hudby, a stejně tak pro zvukové efekty.

Jak již bylo zmíněno výše, z hlediska dodržení certifikačních požadavků je důležitá metoda `IsMPAvaible()`, která nastavuje příznak značící, zda má aplikace povolení používat MediaPlayer. Tato metoda je volána v konstruktoru třídy SoundManager a tak hned po vytvoření instance správce zvuku nastaví příznak, který dale ovlivňuje používání metod spjatých s přehráváním hudby.

Správce zvuku implementuje rozhraní *ISounds*, jež je nabízeno formou služby a usnadňuje tak přístup k funkcím třídy *SoundManager*.

5.3 Správce herních obrazovek

V této kapitole bude popsána část enginu obstarávající správu herních obrazovek. Na úvod si nejprve trochu přiblížíme tuto problematiku. Každá hra, snad jen s vyjímkou těch úplně nejjednodušších se skládá z několika obrazovek. Kromě obrazovky, na níž se odehrává hra samotná, je zde většinou jěště nějáké hlavní menu, obrazovka s nastavením, případně nějaké další menu při pozastavení hry a další obrazovky. Počty obrazovek se mohou u různých her, různých žánrů lišit, téměř vždy je jich však více než jen jedna a proto je potřeba nějak řešit pořadí jejich zobrazování, předávání řízení, přechody mezi obrazovkami a další problémy spojené s tímto řízením. Právě tuto oblast má na starost správce herních obrazovek nebo také stavů hry – anglicky Game state manager, který je součástí většiny her. A také mnoha herních enginů.



Obrázek 5.2: Ukázka typického toku jednoduché hry.

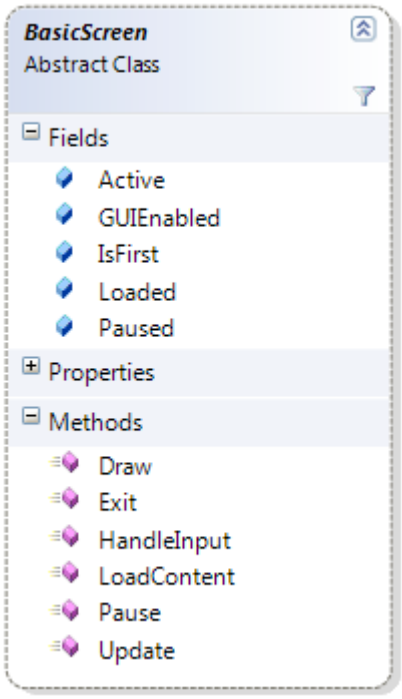
Jak je patrné z úvodního odstavce, jedná se o velmi důležitou součást, která je společná hrám bez ohledu na žánr, a proto jsem se ji rozhodl zahrnout do tohoto enginu a poskytovat vývojářům funkce s touto oblastí spojené. V mé implementaci jsem se rozhodl zaměřit především na základní důležité funkce, nezabýval jsem se tedy věcmi, které nepovažuji za důležité z hlediska vlastního fungování, jako například různé efekty přechodů obrazovek nebo zvláštními obrazovkami pro čekání na nějaké nahrávání. Protože se v této části jedná o problematiku na vyšší úrovni abstrakce než v předchozích popisovaných součástech, tak jsem se zde nemohl opřít o žádný základ v rámci XNA frameworku.

Samotný modul správce herních obrazovek jsem se rozhodl také vytvořit jako herní komponentu, protože však na rozdíl od předchozích dvou modulů obstarává i vykreslování, tak byla jako základ, z nějž třída *ScreenManager* představující tohoto správce dědí, použita třída DrawableGameComponent. Zároveň jsem při návrhu počítal s tím, že se bude jednat o jakousi hlavní komponentu, která tvoří základ hry a vlastně ji obaluje. Hra pak bude představována několika obrazovkami běžícími v rámci tohoto správce.

Základem pro všechny herní obrazovky je abstraktní třída *BasicScreen*, s níž správce pracuje, a proto musí všechny herní obrazovky ve hře používající tuto součást enginu právě z této třídy dědit, což zajistí potřebnou konzistenci. Třída *BasicScreen* je samozřejmě také součástí enginu.

Než se budu věnovat popisu vlastního správce obrazovek, nejprve zde popíšu právě třídu *BasicScreen* aby byl zajištěn potřebný kontext.

5.3.1 Třída BasicScreen



Jak už bylo zmíněno výše, jedná se o abstraktní třídu, jež slouží především jako základ pro budoucí obrazovky samotné hry, obsahuje důležité atributy a metody využité pro správnou funkci správce herních obrazovek.

Při návrhu této třídy jsem se snažil postupovat tak, aby bylo na vývojáře kladeno co nejméně omezení v pozdější tvorbě vlastních herních obrazovek. Zároveň jsem se snažil na herní obrazovku pohlížet jako na jakousi malou herní komponentu, a také brát v úvahu to, že na jedné z obrazovek se bude odehrávát vlastní hra, proto jsem se při návrhu obrazovek nechal dosti inspirovat hlavní třídou Game. Třída *BasicScreen* tedy předepisuje klasické metody `LoadContent()` pro načtení zdrojů a inicializaci obrazovky, `Update()` pro aktualizaci stavu obrazovky, případně logiky hry a `Draw()` pro vykreslování. Navíc je zde metoda `HandleInput()`, ta slouží ke zpracování uživatelských akcí, bylo totiž nutné oddělit aktualizaci herní logiky a zpracování uživatelových podnětů. Proč jsem zvolil toto řešení bude blíže popsáno v další části věnující se vlastnímu správci herních obrazovek. Všechny výše zmíněné metody jsou virtuální a slouží pouze jako předpis pro vývojáře hry, který si jejich funkcionalitu vytvoří sám podle svých potřeb.

Dále tato třída poskytuje metodu `Exit()`, která slouži k ukončení činnosti a smazání herní obrazovky. A metodu `Pause()`, ta pozastaví činnost herní obrazovky a umožní tak například vyvolání vyskakovacího okna. Tyto dvě metody jsem zahrnul do třídy *BasicScreen* jako jakési pomocné, jejichž účelem je obstarávat podle mě často používané a užitečné akce, a tak by mělo jejich využívání vývojaři usnadnit práci a šetřit čas.

Důležitou součástí této třídy je atribut `ScreenManager`, jenž uchovává odkaz na správce obrazovek ovládající tuto obrazovku, což mimo jiné umožňuje obrazovce přístup k hlavní třídě Game. A jako další z hlediska práce správce herních obrazovek jedny z nejdůležitějších jsou atributy

reprezentující skupinu příznaků informujících o aktuálním stavu obrazovky. Tyto atributy jsou typu bool a nabývaji tedy logických hodnot.

Jedná se o následující atributy:

- `Active` – Značí, zda je obrazovka právě aktivní, a tedy viditelná pro uživatele, aktualizující svou logiku a reagující na vstupy od uživatele.
- `IsFirst` – Určuje, zda se jedná o obrazovku viditelnou ihned po spuštění hry.
- `Loaded` – Signalizuje, zda má obrazovka načteny všechny potřebné zdroje a může být používána.
- `Paused` – Pokud nabývá hodnoty true, znamená to, že obrazovka je pozastavena a nemá se tedy aktualizovat a reagovat na vstupy, zůstává však ve frontě správce.

Kromě těchto důležitých příznaků je zde ještě jeden, který plní spíše pomocnou roli a to atribut `GuiEnabled`, ten má sigalizovat, zda se na této obrazovce má zobrazovat grafické uživatelské rozhraní. Jeho využití je však čistě na vývojáři a sám o sobě žádnou určenou funkci nemá.

5.3.2 ScreenManager

V této podkapitole bude popsána třída *ScreenManager*, představující vlastního správce herních obrazovek.

Všechny obrazovky odvozené ze třídy *BasicScreen* jsou uchovávány v seznamu `screen`, z tohoto seznamu jsou kopírovány do seznamu `updateScreens`, z níž jsou postupně během jednoho cyklu vybírány a obslouženy. Tyto dva seznamy, jedná se o kolekce typu List, tedy tvoří jakési jádro správce obrazovek. Obsluha obrazovek probíhá v rámci metody Update jenž bude popsána níže.

Pro jednoznačné určení, která obrazovka je aktivní a zajištění exkluzivity obsluhy, jsem použil navíc ještě princip žetonu, který si obrazovky předávají. Stav žetonu, zda je volný nebo jej právě vlastní některá obrazovka je určen boolovskou proměnou `TokenTaken`, ta nabývá hodnoty true, pokud je právě nějáká obrazovka obsluhována, a tudíž není žeton k dispozici žádné jiné. Dalším takovýmto příznakovým atributem je `popUpOnTop`, ten signalizuje, že je obsluhována pop up obrazovka, která překrývá jinou obrazovku pod ní, prekrývaná obrazovka je ve stavu pauzy. Tento příznak si nastavují samy obrazovky, pokud je u nich volána metoda `Pause()`.

Protože je třída *ScreenManager* schopná vykreslovat na obrazovku, obsauje také `SpriteBatch`, ten od ní navíc mohou získat i vlastní herní obrazovky pro vlastní potřeby vykreslování.

Třída *ScreenManager* dědí ze třídy DrawableGameComponent, a proto dodržuje její strukturu, najdeme zde tedy metodu `Initialize()`, dále pak `LoadContent()`, v níž se pro všechny obrazovky zavolá jejich metoda LoadContent, čímž je jim umožněno načíst si potřebná data a podobně pracuje také metoda `Draw()`, v níž je opět volána metoda Draw ovšem pouze u aktivních obrazovek, aby se mohly vykreslit.

Nejdůležitější metodou je pak metoda `Update()`, v níž probíhá samotná obsluha jednotlivých herních obrazovek. Nejprve je vyčištěn seznam `updateScreens`, do nějž se poté načtou všechny obrazovky ze seznamu `screens`. Pokud byla některá obrazovka přidána dodatečně a nemá tedy ještě načteny potřebné zdroje, tak se zde před jejím přidáním do `updateScreens` zavolá její metoda LoadContent, aby se tato situace napravila. Poté následuje cyklus while, v němž je procházen seznam `updateScreens` a každé obrazovce je nejprve dána možnost získat žeton, a tedy aktivitu pro sebe. To je realizováno vyhodnocením několika příznaků, jak samotné obrazovky, tak i správce obrazovek. Pokud jsou požadavky splněny, obrazovka získává žeton. Nakonec, pokud je nalezena aktivní obrazovka, jsou volány její metody `Update()` a `HandleInput()`, takže obrazovka může vykonávat svou logiku a reagovat na podměty od uživatele. Reakce na uživatelské podměty by mohla být také součástí metody Update, rozhodl jsem se však tyto dvě části odělit, protože si dokáži představit případ v němž by mohlo být žádoucí, aby obrazovka dále aktualizovala svou logiku, ale nemohla reagovat na uživatelský vstup.

Správce obrazovek také samozřejmě obsahuje metody zajišťující přidání nové herní obrazovky `AddScreen()`, a pro zrušení a odebrání některé herní obrazovky `RemoveScreen()`. Také jsou zde pomocné metody `GetToken()` a `ReturnToken()`, jež slouží k získání a vrácení žetonu při předávání aktivity mezi obrazovkami. Dále zde ještě najdeme metody nastavující režim zobrazení obrazovek, jedná se o metody `DisplayNormal()` pro klasické zobrazení na výšku a `DisplayPanorama()` pro zobrazení na šířku.

5.4 Správce uživatelského rozhraní

Uživatelské rozhraní je v nějaké formě přítomno v každé hře. Poskytuje uživateli informace o stavu hry, a také mu dává možnost reagovat a ovládat hru. V případě chytrých telefonů, platformu Windows Phone nevyjímaje, kdy oproti jiným platformám není přítomno žádné zvláštní zařízení pro vstup uživatele jako klávesnice či gamepad, je tato situace často řešena pomocí použití nějakých virtuálních ovládacích prvků. To je důvod, proč právě zde nabývá uživatelské rozhraní ještě o to větší důležitosti, a také důvod, proč jsem se rozhodl správce uživatelského rozhraní zahrnout do tohoto enginu.

Stejně jako u většiny ostatních částí enginu jsem i správce uživatelského rozhraní navrhoval jako herní komponentu, a protože obstarává i vykreslování prvků uživatelského rozhraní na obrazovku, tak je opět jejím základem třída DrawableGameComponent. Toto řešení tedy znamená, že správce uživatelského rozhraní běží na pozadí hry bez ohledu na to, v jakém stavu se hra právě nachází a jednotlivé herní obrazovky jej mohou využívat podle své potřeby. Komunikace se správcem uživatelského rozhraní probíhá přes rozhraní *IGUI*, jež tato komponenta nabízí formou služby.

Původně jsem počítal s objektem typu správce uživatelského rozhraní jako součástí každé herní obrazovky, později jsem však tento koncept zavrhl, protože ne každá obrazovka musí potřebovat uživatelské rozhraní, nebo by mohla být tvořena hra jež se zcela obejde bez využití služeb tohoto správce, v těchto případech by pak byl tento objekt zcela zbytečnou zátěží. Současná implementace tedy počítá s takovým přístupem, že každá obrazovka si při svém vzniku vytvoří a sestaví vlastní uživatelské rozhraní z dostupných prvků, a to pak správce uživatelského rozhraní zobrazuje a zajišťuje reakce na podněty. Při rušení obrazovky si tato pak po sobě rozhraní uklidí.

Jak vyplývá z předchozího odstavce, tak samotné uživatelské rozhraní se skládá z určitých prvků. Základem pro tyto prvky je abstraktní třída *GUIElement* z níž jsou odvozeny některé základní předvytvořené prvky jako tlačítko, label nebo kontejner, jež jsou také součástí enginu. Vývojář si také může vytvořit vlastní prvek, který však vždy musí z této třídy dědit.

5.4.1 Třída GUIElement

GUIElement je abstraktní třída, tvořící základ pro tvorbu prvků uživatelského rozhraní, obsahuje tedy hlavně předpisy metod a důležité atributy jež musí každý prvek implementovat.

Při návrhu této třídy jsem se opět snažil postupovat tak, aby měl vývojář při tvorbě vlastního prvku co největší volnost. Obsahuje tedy pouze několik atributů vymezujících vlastní prvek tak, aby s ním bylo možno snadno pracovat. Sem patří především pro 2D prvky typické vlastnosti jako pozice, která se uchovává jako dvourozměrný vektor, tedy je typu Vector2, dále pak výška a šířka prvku nebo odkaz na správce uživatelského rozhraní, jenž tento prvek řídí. Kromě těchto je zde ještě atribut `type`, určující typ prvku, ten nabývá hodnot výčtového datového typu GUIElementType, který byl vytvořen pro tento účel, jedná se spíše o pomocný atribut, jenž umožňuje správci uživatelského rozhraní snáze určit, jaký typ prvku právě obsluhuje. Jeho využití však není v případě tvoření vlastního prvku nutné. Posledním atributem, který bych zde chtěl zmínit je atribut `parent`, obsahující odkaz na rodičovský prvek, pokud takový existuje. Tento atribut je využit jen u zanořených prvků, například při sdružování prvků do kontejnerů kteréžto bude popsáno dále.

Virtuální metody obsažené v této abstraktní třídě jsou `Draw()`, která je předpisem pro metodu obsahující kód pro vykreslení prvku a metoda `HandleInput()`, jež slouží pro detekci aktivace prvku. V původním návrhu nebyla obsažena metoda `Update()`, neboť jsem počítal pouze se statickými prvky uživatelského rozhraní. Pozdější jsem se ji však rozhodl zahrnout za účelem, poskytnout vývojáři možnost tvořit i dynamické prvky. Jako poslední je zde metoda `Hitbounds()`, která vrací Rectangle ohraničující prvek, tato metoda má kromě vykreslování hlavní využití především při určování, zda byl prvek na obrazovce zasažen dotekem.

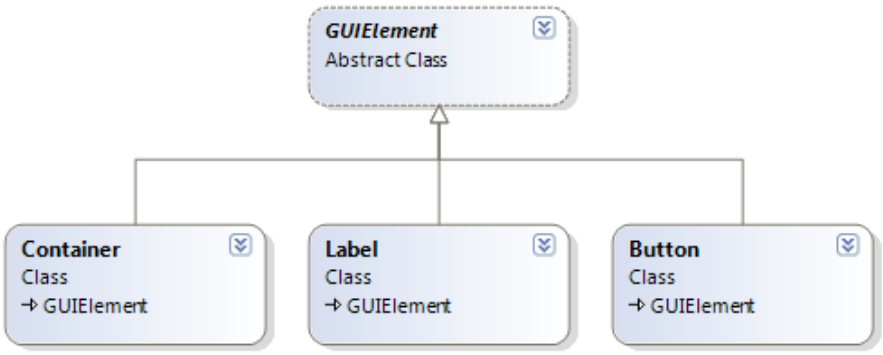
5.4.2 Třída GUIManager

Nyní se dostáváme ke třídě *GUIManager* reprezentující vlastního správce uživatelského rozhraní. Ta je, jak už bylo zmíněno implementována jako vykreslovatelná herní komponenta, tudíž dědí ze třídy DrawableGameComponent, pro vykreslování používá vlastní SpriteBatch, a protože v rámci uživatelského rozhraní může pracovat i s nápisy, tak má také atribut definující používaný font. Ten je samozřejmě možné nastavit. Mezi atributy najdeme také atribut `GUIEnabled`, jehož nastavení umožňuje povolit nebo zakázat vykreslování uživatelského rozhraní na obrazovku. Toto bylo původně řešeno pomocí atributu `Visible` zděděného z nadřazené třídy DrawableGameComponent. To se však příliš neosvědčilo a proto jsem přistoupil k vlastnímu řešení. Klíčovou součástí této třídy je seznam `Elements`, v němž jsou uchovávány všechny prvky uživatelského rozhraní.

Třída GUIManager nabízí své rozhraní jako službu, tu registruje u hlavní třídy Game hned ve svém konstruktoru, defaultně také na tomto místě nastaví atribut `GUIEnabled` na false, čímž zamezí vykreslování uživatelského rozhraní dokud to není povoleno vývojářem. Ke svému fungování správce uživatelského rozhraní využívá jako jediná část enginu také služeb jiné části, a to správce vstupů (viz. Kapitola 5.1), o jehož službu žádá při své inicializaci. Dále zde najdeme metodu `Update()`, v níž je volána metoda Update jednotlivých prvků, tato část byla přidána až později kvůli podpoře dynamických prvků, o níž jsem se zmiňoval v předchozí kapitole.

Nejdůležitější metodou tvořící hlavní logiku správce uživatelského rozhraní je pak metoda `HandleInput()`. Ta funguje tak, že z informací o aktuálních dotecích na obrazovce, jež získává ze správce vstupů, zjišťuje, zda některý z doteků směřuje na nějaký prvek uživatelského rozhraní. To je realizováno procházením seznamu prvků, kdy je u každého prvku zavolána jeho metoda `Hitbounds()` a je tak získán hraniční Rectangle, za pomoci jehož metody Contains se zkontroluje, zda souřadnice doteku směřují na tento prvek. Aby mohl být tento postup použit, je nejprve třeba převést reprezentaci doteku z typu TouchLoaciton na typ Point.

Důležitá je také metoda `Draw()`, zajišťující vykreslování jednotlivých prvků. To opět probíhá tak, že je volána metoda Draw samotných prvků. Ačkoliv se zde nabízí na první pohled jednodušší možnost, a to nechat správce uživatelského rozhraní přímo vykreslovat prvky, tak jsem se rozhodl nakonec přenechat vykreslování sebe sama až na vlastní prvky, a to především z důvodu snažší implementace možnosti zanořování prvků za použití kontejnerů.



Obrázek 5.4: Předdefinované komponenty GUI.

Dále třída GUIManager implementuje několik metod k přidávání prvků, jako `AddElement()` pro přidání jakéhokoliv libovolného prvku. A dále metody pro přidání předdefinovaných prvků `AddButton()` pro přidání jednoduchého tlačítka, a její přetížená varianta, která umožňuje přidat tlačítko s nápisem a zvolit jeho barvu, metoda `AddLabel()` k přidání prvku typu Label a nakonec je zde metoda `AddContainer()` pro přidání prvku typu kontejner. Těmto předvytvořeným prvkům se budu blíže věnovat v další části. Poté už jsou zde jen metody pro odebírání prvků, a to `RemoveElement()` pro odstranění jednoho konkrétního prvku a `ClearGUI()` pro odstranění všech prvků.

V souvislosti s uživatelským rozhraním jsem se rozhodl do enginu zahrnout i několik prvků uživatelského rozhraní, jež považuji za užitečné a ve hrách často používané, a předpokládám, že jejich využití vývojářům usnadní práci. Jedná se o prvky tlačítko, label a kontejner.

5.4.3 Prvek Label

Tento prvek, reprezentován třídou *Label*, slouží k zobrazování popisků a různých textových informací. Mezi jeho atributy tedy patří především text, jenž je zobrazen a barva, v níž je tento text vyveden. Jako všechny prvky je schopen se sám vykreslovat za použití SpriteBatche a fontu, které získává z rodičovského správce uživatelského rozhraní. V případě, že je zanořen v nějakém jiném prvku, tak se automaticky zarovnává podle polohy svého rodiče.

5.4.4 Prvek Button

Tlačítko považuji za nejdůležitější z běžných prvků uživatelského rozhraní. Proto jsem jej také zahrnul mezi prvky, které jsou součástí enginu. Prvek typu tlačítko je schopen reagovat na dva typy podnětů, a to kliknutí, a nebo souvislé držení. Typ podnětu, na který bude tlačítko reagovat se vybírá při jeho instancializaci, a je uchováván v atributu typu ActionType, což je výčtový typ vytvořen pro tyto účely. Tlačítko má texturu, jež představuje jeho vizuální reprezentaci, a může být navíc rozšířeno o nápis, k tomuto účelu je zde metoda `AddText()`, která nastaví příznak `hasText` a přidá vlastní text jenž se má zobrazovat. Tlačítko ma navíc také vlastní SpriteFont, což umožňuje použít jiný font než ve zbytku správce.

Akce, která má být při použití tlačítka vykonána, je s ním svázana pomocí EventHandleru `onAction` a vyvolává se z metody `HandleInput()`, v této metodě se pracuje s informacemi ze správce vstupu, odkaz na jeho rozhraní je v tomto případě předáván jejím parametrem.

5.4.5 Prvek Container

Posledním, avšak velmi důležitým předvytvořeným prvkem, je kontejner. Ten umožňuje sdružovat prvky do logických celků a vytvářet tak oddělené součásti uživatelského rozhraní a především je díky němu možné zanořování prvků. Zanořování prvků jsem původně řešil tak, že bylo možné u všech prvků, takže například bylo možné vytvořit tlačítko v tlačítku. Toto řešení jsem však shledal nepříliš logickým a zbytečně komplikovaným. Nakonec jsem se rozhodl zanořování omezit pouze na kontejnery. Nyní je tedy možné zanořovat prvky tím způsobem, že se do kontejneru přidá další kontejner a do něj prvky nižší úrovně a tak dále, přičemž úroveň zanoření není omezena.

Kontejner do jisté míry představuje zmenšenou verzi správce uživatelské rozhraní. Také obsahuje seznam prvků, které jsou v něm obsaženy, a ty poté obsluhuje stejným způsobem, jaký byl popsán v kapitole věnující se třídě GUIManager. Kontejner však může být jako běžný prvek vidět na displayi, a proto obsahuje vlastní texturu představující pozadí kontejneru. Výhodou použití kontejneru také je, že když do něj sdružíme prvky a vytvoříme tak nějakou ucelenou část uživatelského rozhraní, můžeme pak tuto část přesouvat jako celek, například při snaze vytvořit optimalní uspořádání uživatelského rozhraní. Jelikož je kontejner u vnořených prvků považován za rodičovský prvek, po jeho přesunutí se všechny vnořené prvky samy zarovnají na dané místo a není

tedy třeba přesouvat každý zvlášť. Později byla také přidána možnost nastavovat viditelnost kontejneru pomocí atributu `Visible`, což v případě rozdělení uživatelského rozhraní na části s použitím kontejnerů umožní například některou část zviditelnit až na základě určité akce apod.

Třída Container obsahuje také několik metod pro přidávání a odebírání prvků. Ty jsou shodné jako ve třídě *GUIManager*.

5.5 Graf scény a kolize

Vykreslování jako takové je na platformě Windows Phone v režii XNA, čehož je také v tomto enginu a jeho částech využito. Proto jsem se zaměřil spíše na organizaci scény a rozhodl jsem se implementovat graf scény.

5.5.1 Graf scény

Grafy scény jsou konceptuální nástroje pro reprezentaci virtuálního světa v aplikacích počítačové grafiky. Graf scény je hierarchická struktura obsahující uzly spojené hranami. Uzly scény spravují data popisující virtuální scénu a hrany, jež uzly spojují, popisují vztah mezi nimi. Uzly jsou ideálně uspořádány hierarchickým způsobem, tak aby sémanticky a prostorově korespondovaly s modelovaným světem.

Uzly grafu scény můžeme rozdělit do tří kategorií. Kořenový uzel, seskupovací neboli skupinové uzly a listové uzly nacházející se na konci větve. Kořenový uzel je prvním uzlem a všechny další uzly jsou s ním přímo, či nepřímo spojeny. Vnitřní uzly mohou mít mnoho vlastností, například v grafice časté transformace jako rotace, posunutí, zkosení nebo změna měřítka, a popisují pozici a orientaci objektu, nebo jeho stav v rámci světa. [20]

Graf scény se používá především ve 3D grafice pro reprezentaci virtuálních světů, kde je objekt reprezentován skupinou uzlů s různým účelem. Já se jej však rohodl použít i v tomto enginu pracujícím pouze s 2D grafikou. Zde může plnit podobný účel, ačkoliv graf nebývá tak komplikovaný. Jeho použití v mém případě spočívá především v seskupování dílčích objektů v celek představující jeden logický objekty scény, například postava může být tvořena kromě vlasního spritu s texturou postavy i spritem s texturou zbraně, a případně k ní přidruženým nějakým efektem.

Vzhledem k tomu, že graf scény obsahuje přehlednou reprezentaci všech objektů ve scéně rozhodl jsem se informací v něm obsažených využít také k implementaci jednoduchého systému kolizí.

5.5.2 Třída SceneGraph

Jak již bylo zmíněno, graf scény je tvořen stromovou strukturou, proto je většina funkcionality implementována ve třídě *Node*, jež představuje uzly tohoto grafu. Tato třída bude popsána níže. Třída *SceneGraph* tvoří jakýsi obal a zároveň kořen tohoto stromu. Najdeme zde několik metod pro správu první úrovně grafu, která je nejdůležitější, protože uzly v této úrovni představují vlastní objekty ve scéně. S touto úrovní je možno si při tvorbě nejjednodušších her, kdy je jeden objekt reprezentován pouze jedním uzlem, zcela vystačit.

Pro indetifikaci uzlu jsem se rozhodl používat číselné ID, proto třída *SceneGraph* obsahuje proměnou `idCounter`, jejíž hodnota se ukládá do každého uzlu při jeho vkládání zavoláním metody `AddChild()`. Tato metoda pak po vložení uzlu vrací právě hodnotu jeho ID, aby bylo možno jej uchovat v rodičovském objektu za účelem další manipulace s uzlem, který je s ním svázán. Po vložení uzlu se hodnota `idCounter` zvýší, aby se zajistila její jednoznačnost.

Na základě ID uzlů je k nim pak možno přistupovat a to pomocí metody `GetNodeByID()`, která vrací přímo odkaz na uzel se zadaným ID, nebo metody `GetNodeIndexByID()` jež vrací

index uzlu v rámci seznamu potomků. Najdeme zde i metodu na odstranění konkrétního uzlu `RemoveChild()`.

Dále třída *SceneGraph* obsahuje metodu `Update()` pro aktualizaci stavu uzlů, což je podobně jako u správce uživatelského rozhraní řešeno voláním příslušné metody jednotlivých uzlů. Další metodou této třídy je `Draw()`, které pracuje na stejném principu jako `Update()`, a umožňuje, aby se uzly vykreslily.

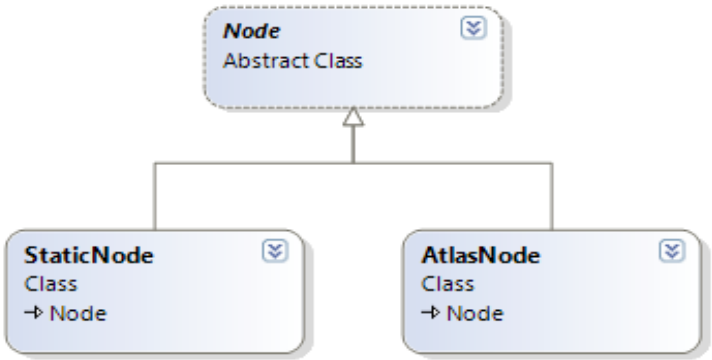
Navíc je zde ještě medota `Gravity()`. Ta nesouvisí s prácí s grafem scény, rozhodl jsem se ji však vytvořit, protože si dokážu představit její využití v mnoha typech her a především v plošinovkách pro které je tento engine hlavně vyvíjen. Tato metoda, jak už její název napovídá, simuluje gravitaci a to tak, že uzly v grafu scény a tím pádem i objekty s nimi spojené, nutí klesat dokud neodjde ke spodní kolizi. Tato funkce je zde pouze jako jakýsi bonus a vývojář ji nemusí využívat nebo si místo ní může vytvořit vlastní.

5.5.3 Třída Node

Třída *Node* je základem implementce grafu scény, představuje jeden uzel ve stromové struktůře grafu scény. Tvoří základ pro třídy *StaticNode* a *AtlasNode*.

Potomci tohoto uzlu jsou uchovávání v seznamu `Children`, což zajišťuje možnost pokračování ve stromové struktuře. Každý uzel má své ID, jehož funkce je popsána v předchozí kapitole a obsahuje odkaz na graf scény pod nějž spadá, což zároveň znamená kořen stromu. Také je zde atribut Type, jež určuje, zda uzel reprezentuje statický či animovaný prvek scény, tento atribut nabývá hotnot zvláště vytvořeného výčtového typu NodeType. Velmi důležitý je atribut `ConnectedObject`, obsahující odkaz na objekt, který je uzlem reprezentován. Další atributy jsou pro potřeby 2D reprezentace typické, patří sem pozice, šířka a výška a dále textura, a také atribut `offset`, ten slouží k určení odsazení od rodičovského uzlu. Při návrhu jsem se potýkal s problémem, jak určit otočení prvku, tedy například při reprezentaci postavy určit směr jejího pohledu. To jsem nakonec vyřešil opět vlastním datovým typem FaceDirection, který nabývá dvou hodnot podle nichž se určuje, zda je objekt směřován doprava či doleva. V rámci třídy *Node* je tato informace uchovávána atributem `faceDirection`.

Přidávání a rušení potomků je stejně jako v případě třídy *SceneGraph* řešeno metodami `AddChild()` a `RemoveChild()`. Z hlediska vykreslování a především kolizí je důležitou metodou `GetBoundBox()`, která vrací Rectangle ohraničující prvek scény reprezentovaný uzlem. Nejrozsáhlejší metodou třídy *Node* je metoda `UpdateChildrenPosition()`, ta se stará o aktualizaci polohy přímých potomků uzlu, přičemž je bráno v úvahu především natočení uzlu, a také se zde dopočítává výsledná poloha potomka na základě jeho atributu `offset`. Pokud je navíc potomkem animovaný prvek, tak se v rámci této metody také aktualizuje jeho animace. Zde vypočítané změny se pak zpětně promítnou i na vlastní objekt, jenž je upravovaným uzlem reprezentován. Ve třídě *Node* je obsažena i virtuální metoda `Draw()`, jejíž implementace je však ponechána až na dědící třídě.



Obrázek 5.5: Rozšiřující třídy abstraktní třídy Node.

Dále je zde několik metod pro detekci kolizí, ty však budou popsány ve zvláštní kapitole.

5.5.4 Třída StaticNode

Tato třída představuje uzel v grafu scény reprezentující statický prvek ve scéně. Rozšiřuje třídu *Node* a pouze doplňuje implementaci metody `Draw()`, ta zohledňuje především natočení objektu, přičemž opačný směr zobrazení je dosažen pomocí efektu FlipHorizontally, který způsobí přetoční textury podle vertikální osy. Také se zde bere při vykreslování v úvahu pozice kamery, podle níž se při vykreslení prvek příslušně posune. V této metodě se zároveň i zajistí vykreslení potomků uzlu voláním jejich metody Draw.

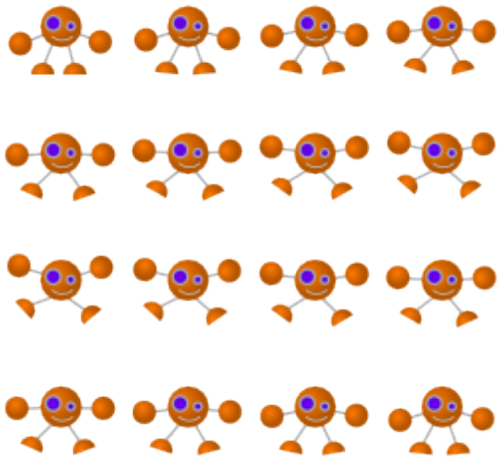
5.5.5 Třída AtlasNode

Druhou třídou rozšiřující třídu *Node* je třída *AtlasNode*, ta slouží k reprezentaci animovaných prvků. Animace je dosaženo použitím takzvané Atlas textury. To je textura skládjící se z několika okének, přičemž v každém je objekt v jiné fázi animace (viz. Obr 5.6), při vykreslování se pak namísto celé textury používají postupně tyto části a tak vzniká animace.

Aby bylo možno s pomocí atlas textury animovat, je potřeba znát počet těchto okýnek – framů, ten se většinou udává jako počet sloupců a řádků, aby se mohlo určit jejich umístění. Tyto informace jsou zadávány už při tvorbě uzlu a potřebné informace, jako rozměry jednoho framu, jsou pak dopočítány s použitím dalších údajů, například šířka a výška celé textury.

Vlastní animování, tedy posun po jednotlivých framech zajišťuje metoda `UpdateFrame()`, ta posune animaci pokaždé, když detekuje změnu pozice. Pokud není detekována změna pozice zůstává počítadlo framů reprezentované proměnou `currentFrame` na hodnotě 0. Tím se zajistí, že objekt nezůstává například v prostřední fázi animace, což by působilo nesmyslně.

Stejně jako *StaticNode* i tato třída má vlastní implementaci metody `Draw()`, v níž stejně jako *StaticNode* počítá s otočením prvku a pozicí kamery. V této metodě je však také hlavní logika zajišťující atlas animaci, kde se podle počítadla framů zjistí, která část textury se má použít a vypočte se její výška a šířka. Poté se pomocí těchto informací sestaví zdrojový Rectangle, jež je použit při vykreslování k výběru správné části atlas textury.



Obrázek 5.6: Atlas textura.

5.5.6 Kolize

Z oblasti fyziky jsem do tohoto enginu zahrnul pouze jednoduchý systém kolizí založený na hranicích objektů tvořených pomocí hraničního Rectangle. Protože graf scény sdružuje reprezentace všech

objektů ve scéně na jednom místě, tak jsem se toho rozhodl využít a implementovat zde i systém kolizí.

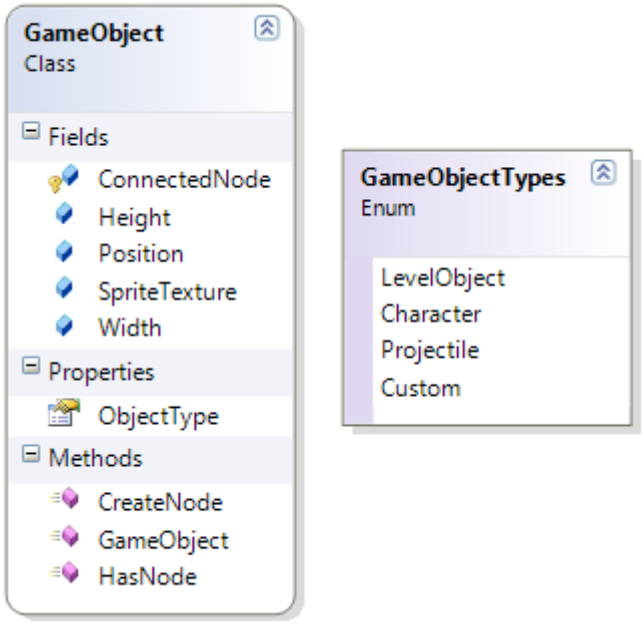
Detekci kolizí jsem založil na na průniku hraničních Rectanglů pomocí metody Intersects, jež je součástí třídy Rectangle. Pro získávání hranic prvků slouží metoda `GetBoundBox()`, která byla popsána v kapitole věnující se třídě *Node*. Kolize se řeší pouze v první úrovni grafu scény, kde jsou uzly reprezentující hlavní objekty ve scéně.

Po vyzkoušení různých variant jsem kolize implementoval způsobem, při kterém je kolem objektu vytvořena jakási kolizní zóna, čehož je dosaženo mírným zvětšením hraničního Rectanglu a kolize se tedy detekují už v této zóně dříve, než by se protly vlastní objekty. Toto řešení pomohlo vyhnout se problémům zejména při současné detekci více kolizí.

V rámci detekce kolizí jsem vytvořil několik metod, které buď pouze kontrolují zda je objekt v kolizi, nebo detekují kolize z určité strany objektu. Každá metoda má dvě varianty a to jednu s návratovým typem bool, která pouze signalizuje výskyt kolize pro použití v jednodušších řešeních, a druhou verzi, jež vrací odkaz na objekt, s nímž objekt, pro nějž je prováděna kontrola koliduje, pro použití v náročnějších případech, kdy je potřeba nějaká interakce mezi kolidujícími objekty.

Všechny tyto metody fungují podobně a to tak, že prochází všechny uzly v první úrovni grafu scény a kontrolují, zda hranice některého z nich protínají hranice uzlu patřícího k objektu, pro nějž se provádí kontrola, přičemž se samozřejmě vynechává objekt, z kterého je metoda volána.

5.6 Třída GameObject



Obrázek 5.7: Třída GameObject.

Poslední částí enginu je třída *GameObject*. Tato třída tvoří základ pro všechny objekty, jež se budou vyskytovat ve hře postavené na tomto enginu jako součásti herní scény. Tomuto řešení jsem se v počáteční fázi velmi snažil vyhnout, protože jsem měl obavy z přílišného omezení vyvojářů. Nakonec jsem však tyto úvahy přehodnotil a rozhodl jsem se přece jen základ pro herní objekty do enginu zahrnout s tím, že budu při návrhu této třídy postupovat maximálně obezřetně s ohledem na omezení vývojářů. Důvodem k tomuto rozhodnutí bylo především použití vazby na vlastní objekty v uzlech grafu scény, a také pozdější změna mého přístupu k problematice volnosti vývojáře, kdy jsem si uvědomil, že stanovit jakýsi základní rámec pro vývojáře pracující s tímto herním enginem může být naopak z hlediska jeho používání prospěšné.

Při návrhu této třídy jsem se tedy, jak už bylo zmíněno, snažil postupovat s velkým důrazem na to, abych co nejméně omezil vývojáře, proto třída obsahuje jen několik atributů, jež považuji u objektu za nezbytné. To je například pozice, výška a šířka, a také textura, ta je zde i přes to, že jak vyplývá z předchozí kapitoly, jsou objekty vizuálně reprezentovány svými příslušnými uzly v grafu scény, a to proto že jsem nechtěl v rámci samotných uzlů grafu scény pracovat s načítánim zdrojů, což mi připadalo neobratné. Rozhodl jsem se tedy pro přístup, kdy je textura načtena v rámci objektu a do uzlu, který jej reprezentuje, je pak už jen předána při jeho tvorbě.

Kromě těchto atributů jsou zde ještě dva, jež je možno považovat za ne tolik standardní, a to především atribut `ConnectedNode`, představující druhou stranu vazby mezi objektem samotným a jeho uzlem v grafu scény, jenž je reprezentován svým ID. I v tomto případě jsem si v pozdější fázi vývoje uvědomil, že jsem mohl postupovat jinak a vazbu realizovat přímo odkazem na objekt typu Node, bohužel to však bylo až ve fázi kdy byl vývoj téměř u konce a změna by vyžadovala příliš rozsáhlé úpravy.

Posledním atributem je atribut `ObjectType`. Ten nabývá jedné z hodnot výčtového datového typu GameObjectTypes, který jsem pro tento účel vytvořil a vložil do něj hodnoty představující některé základní typy objektů o kterých se domnívám, že by mohly najít ve hrách své využití, jako například postava, objekt v levelu nebo projektil. Kvůli zachování volnosti vývojáře je zde pak také možnost Custom značící libovolný objekt, který není ve výčtu zahrnut.

Dále tato třída obsahuje už jen metodu `HasNode()`, která na základě atributu `ConnectedNode` zjišťuje, zda je daný objekt reprezentován nějakým uzlem. Je zde také virtuální metoda `CreateNode()`, ta tvoří pouze předpis pro implementaci metody zajišťující tvorbu a připojení uzlu reprezentujícího objekt ve scéně. Původně tato metoda byla plnohodná, a danou funkci plnila už v rámci této třídy. Od tohoto řešení jsem se však rozhodl ustoupit, protože jsem to shledal z hlediska vývojáře příliš omezujícím.

6 Tvorba Hry

Druhou významnou částí této práce bylo vytvořit 2D hru založenou na mém enginu a tím demonstrovat jeho funkčnost a použitelnost. Demonstrační hra a její tvorba bude popsána v této kapitole, přičemž se především budu soustředit na to, jak finální hra využívá funkce poskytované enginem, a jaký vliv mělo použití enginu na její tvorbu.

Již od počátku tohoto projektu jsem počítal s tím, že engine bude primárně sloužit pro hry žánru akční plošinovka, a proto jsem pro demonstraci enginu zvolil hru právě tohoto žánru. Také bych chtěl hned zde zpočátku uvést, že při vývoji hry bylo hlavním cílem demonstrovat možnosti enginu a ne vytvořit zábavnou a konkurenceschopnou aplikaci. I přes to jsem se však snažil tomuto cíli co nejvíce přiblížit.

6.1 Návrh a námět hry

Při hledání námětu hry jsem se nechal inspirovat klasickými plošinovkami z devadesátých let, respektive jejich remaky, jež jsou v dnešní době mobilních zařízení opět populární. Jako konkrétní příklad mohu uvést hru Retro Warfare 2. [21]

V mé hře jsem se však chtěl více zaměřit na příběh a atmosféru. Hra se odehrává v postapokalyptickém prostředí, kdy svět ovládli nepřátelští mimozemšťané a hráč v roli vojáka, jenž právě procitl z kómatu na zničené základně, se vydává bojovat o přežití.

Hlavním prvkem hry měla být kombinace skákání a střílení, čehož hráč využívá k překonání překážek a zničení nepřátel. Dále jsem se rozhodl hru obohatit několika jednoduchými power-upy a souboji se zvláštními protivníky, takzvanými bossy. Po grafické stránce jsem se rozhodl držet jednoduchého, kresleného stylu s použitím pouze základních barev.

6.2 Herní obrazovky

Jak bylo zmíněno v kapitole 5.3, každá hra se skládá z několika obrazovek a tato není výjimkou, zde budou popsány jednotlivé obrazovky a jejich účel.

- **SplashScreen** – Úvodní obrazovka zobrazující logo hry a enginu.
- **MainMenuScreen** – Obrazovka hlavního menu, slouží jako rozcestník, lze z ní přejít na obrazovku s natavením, do hry nebo ukončit aplikaci.
- **OptionsScreen** – Obrazovka nastavení, umožňuje uživateli měnit nastavení zvuku nebo resetovat postup ve hře. Vrací se z ní do hlavního menu.
- **GameplayScreen** – Obrazovka na níž se odehrává hra, tvoří nejdůležitější část aplikace, lze z ní přejít do menu pauzy.
- **BriefingPopUp** – Jedná se o úvodní obrazovku levelu obsahující informace pro uvedení hráče do příběhu, je vyvolána nad pozastavenou hlavní herní obrazovkou na začátku levelu a po svém potvrzení zmizí a začíná hra.
- **PausedScreen** – Obrazovka s menu pauzy, je vyvolána nad hlavní herní obrazovkou v případě pozastavení hry, umožňuje pokračovat ve hře nebo se vrátit do hlavního menu.

Tyto obrazovky tvoří základní celky hry a jsou reprezentovány stejnojmenými třídami, jež jsou vždy rozšířením třídy *BasicScreen*. K řízení herních obrazovek je ve hře využit správce heních obrazovek poskytovaný enginem, jehož instance je vytvořena ihned po spuštění hry a hned poté je do něj vložena úvodní splash obrazovka, jakožto vstupní bod aplikace. Další změny toku hry jsou pak řízeny samotnými obrazovkami, které ukončují svou činnost zavoláním metody `Exit()` a typicky i

vkládají do správce obrazovku, jež má následovat. O vlastní přepnutí a změny s ním spojené se pak už postará správce herních obrazovek.

```
if (input.BackPressed())
{
	SaveOptions(SoundEnblState);
	ScreenManager.AddScreen(new MainMenuScreen(),false);
	gui.ClearGUI();
	Exit();
}
```

Zdrojový kód 6.1: Kód pro přepnutí obrazovek.

Kromě samotného správce obrazovek, využívají všechny obrazovky služeb ještě jedné části enginu, a to správce vstupů. Ten jim většinou slouží ke zjišťování, zda bylo stisknuto tlačítko Zpět, což je akce, na niž všechny obrazovky nějakým způsobem reagují. U většiny obrazovek je reakcí ukončení obrazovky a návrat na předchozí obrazovku, většinou do hlavního menu. V případě hlavní herní obrazovky pak pozastavení hry a vyvolání obrazovky pauzy.

6.3 Menu

Menu jsou důležitou součástí hry umožňující uživateli základní volby a ovládání toku hry. V této hře jsem se snažil, aby byla menu co nejjednodušší, skládají se tedy z tlačítek a jednoduchého pozadí reprezentovaného obrázkem, v případě menu nastavení je pak menu rozděleno graficky na několik částí.

Všechna menu jsou vytvořena s pomocí další části enginu, a to správce uživatelského rozhraní. Položky většiny menu jsou tvořeny prvky typu tlačítko s nápisem. Vyjímkou je menu nastavení, v němž jsou použity složitější prvky a kterému bude věnována zvláštní podkapitola. Použití správce GUI a předvytvořených prvků obsažených v enginu se ukázalo být výrazným zjednodušením, které přispělo k významné úspoře času při vývoji. K vytvoření a sestavení menu slouží na každé obrazovce metoda `SetupGUI()`.

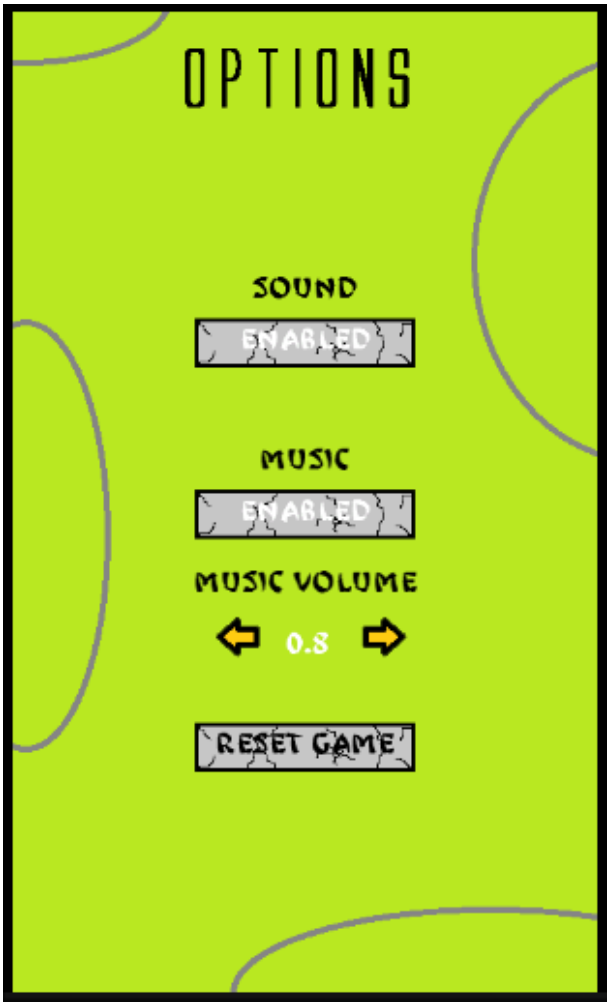
6.3.1 Menu nastavení

Menu nastavení jsem se rozhodl věnovat zvláštní podkapitolu, protože se jedná o nejsložitější menu ve hře, a protože se většina nastavení týká zvuku, je zde také využita další část enginu a to správce zvuku.

V tomto menu může uživatel povolit nebo zakázat zvukové efekty hry nebo přehrávání hudby, popřípadě obojí, dále může v případě, že je povoleno přehrávání hudby, nastavovat její hlasitost, a také může v tomto menu resetovat svůj postup ve hře a začít od začátku.

Pro prvky spojené s manipulací se zvukovými efekty jsem vytvořil zvláštní kontejner abych je logicky oddělil od prvků pro manipulaci s hudbou, jež jsou také umístěny ve vlastním kontejneru. Povolení a zakázání hudby a zvukových efektů pracuje na stejném principu. Jedná se o tlačítko po jehož stisku je zavolána metoda, která voláním metod správce zvuku zjistí aktuální stav povolení a změní jej na opačný, zároveň s tím se také změní nápis na tlačítku, aby reflektoval současný stav povolení. Aktuální stavy povolení jak hudby, tak zvuků jsou uchovávány i ve zvláštních proměnných za účelem ukládání stavu. V případě kontejneru ovládání hudby je zde ještě další skupina prvků, které slouží k ovládání hlasitosti, ty jsou umístěny ve zvláštním zanořeném kontejneru, jenž je zobrazen pouze v případě, že je povoleno přehrávání hudby. V opačném případě nemá nastavování hlasitosti smysl, a proto tento kontejner není viditelný. Prvky ovládání hlasitosti jsou tři, tlačítka pro snížení a zvýšení hlasitosti a label informující o aktuální hlastisti. Metody přiřazené těmto prvkům opět k vyvolání změn hlasitosti využívají funkcí správce zvuku.

Při opouštění obrazovky s nastavením se všechny provedené změny uloží do k tomuto účelu sloužícímu souboru, a nastavení je tak uchováno i po ukončení aplikace. Ukládání a načítání bude věnována samostatná kapitola.



Obrázek 6.1: Obrazovka nastavení.

6.4 Hra

6.4.1 Herní obrazovka

Herní obrazovka je hlavní částí této aplikace, zajišťuje mnoho pro hru důležitých funkcí, jako načítání postupu ve hře, přehrávání hudby a zvuků, ošetřuje různé stavy hry a přechod mezi nimi a hlavně zobrazuje scénu hry a vytváří ovládací uživatelské rozhraní.

Uživatelské rozhraní hry je opět tvořeno pomocí správce uživatelského rozhraní poskytovaného enginem, metoda v níž je sestaveno se stejně jako u dalších obrazovek jmenuje `SetupGUI()`. Samotné uživatelské rozhraní se skládá ze čtyř tlačítek a z jednoduchého HUDu informujícího hráče o stavu hitpointů a armoru. Tlačítka sloužící k ovládání pohybu do stran, skákání a střelbě a akce s nimi spojené volají příslušné metody objektu typu Player. Aktualizaci HUDu má na starosti metoda `UpdateHUD()`, jež je volána vždy na konci každé update části herní smyčky a opět zobrazuje informace získané z objektu Player.

Obrázek 6.2: Ukázka uživatelského rozhraní ve hře.

Důležitou funkcí herní obrazovky je také řízení stavů hry, výskyt jednotlivých stavů je kontrolován v každém cyklu herní smyčky v metodě `Update()` a na základě detekovaného stavu je pak provedena odpovídající akce. Stavy mohou nastat následující:

- **Návrat z režimu pozastavení hry** – Tento stav nastává vždy po návratu do hry, jež byla z nějakého důvodu pozastavena, což je signalizováno nastaveným atributem `musicPaused`.
- **Požadavek na zobrazení brífinku** – Tento stav se vyskytuje na začátku každého levelu a je reprezentován nastavením atributu `briefingFlag`, reakcí je pozastavení hry a vyvolání obrazovky briefinku zavoláním metody `ShowBriefing()`.
- **Ukončení hry** – Požadavek na ukončení hry je signalizován nastavením atributu `quitFlag`, pokud nastane tento stav, herní obrazovka zastaví hudbu, vyčistí uživatelské rozhraní a zahájí přechod do hlavního menu.
- **Postup do dalšího levelu** – Tento stav nastane pokud je nastaven atribut `LevelComplete` v objektu reprezentujícím level, po detekování tohoto stavu je zavolána metoda `NextLevel()`, která uloží postup ve hře a zahájí znovunačtení herní obrazovky, už s dalším levelem.
- **Restart levelu** – V případě smrti hráče, signalizované atributem `IsDead` objektu Player, je zavolána metoda `Restart()`, ta zahají znovunačtení herní obrazovky čímž restartuje současný level.
- **Pozastavení hry** – Posledním stavem je pozastavení hry, tento stav je reakcí na stisk tlačítka Zpět a je detekován v metodě `HandleInput()` za pomoci správce uživatelských vstupů poskytovaného enginem. V reakci na tento stav je pozastaveno přehrávání hudby, nastaven atribut `musicPaused` a vyvolána obrazovka pauzy.

Jak už bylo několikrát zmíněno výše, herní obrazovka se také stará o zvukovou stránku hry, k tomu je plně využito možností další části enginu a to správce zvuků. Veškeré ovládání zvukových efektů a hudby je pak zajitěno pouze voláním příslušných metod správce zvuku.

Poslední, avšak možná nejdůležitější částí o níž bych chtěl v této podkapitole psát, je vykreslování scény. Ve vlastní režii herní obrazovka vykresluje pouze texturu pozadí celého levelu, jež získává z objektu Level, o vykreslování všeho ostatního se stará engine. Součástí herní obrazovky

je instance třídy *SceneGraph* dodávané enginem, odkaz na tuto třídu představující graf scény je pak předáván všem objektům vyskytujícím se ve hře, které do něj mohou vložit své uzly a být tak reprezentovány ve scéně. Samotného vykreslení scény je pak dosaženo voláním metody `Draw()` instance grafu scény.

6.4.2 Level

V této kapitole se, jak už její název napovídá, budu věnovat reprezentaci levelu v demonstrační hře, již představuje třída *Level*. Tato třída je důležitá ze dvou důvodů. Zajišťuje vlastní načtení levelu ze souboru XML, v němž je definován, a především jsou v této třídě sdruženy informace o všech objektech vyskytujích se ve hře a ty jsou z této třídy řízeny.

V této třídě jsou samořejmě také uchovávány informace o samotném levelu, jako je výška a šířka tohoto levelu, textura pozadí nebo hudba hrající v levelu, dále zde najdeme prvky briefingu, jež je zobrazen na začátku levelu, a to pozadí obrazovky briefingu a vlastní text sloužící k uvedení hráče do děje.

Další obsažené informace se týkají vlatní hratelnosti. Mezi ně patří síla gravitace působící v levelu, úroveň pod níž hráč nesmí klesnout jinak zemře – například pro implementaci pádu z platformy, příznak značící dokončení levelu a s tím spojené atributy definující podmínky vítězství, jež mohou být pro různé levely různé, a případně souřadnice cílového místa jehož dosažením hráč dokončí level.

Název	Popis
Útěk z lokace	Hráč musí dosáhnout určeného místa v úrovni.
Porážka bosse	Hráč musí zabít speciální typ nepřítele.

Tab 6.1: Typy podmínek vitězství.

Samotný level se skládá ze čtyř základních prvků, jež jsou klíčové pro celou hru, jedním z nich je samozřejmě hráč, dále pak nepřátelé, objekty levelu tvořící herní prostředí jako platformy, překážky, dveře a nakonec power-upy. Tyto skupiny objektů jsou uchovávány v seznamech daných objektů, jedinou výjimkou je hráč, jenž je pouze jeden, a proto k jeho uchování stačí obyčejná proměnná.

Třída *Level* má vlastní instanci ContentManageru, jenž je používán k načtení potřebných zdrojů na základě informací z XML souboru popisujícího level, dále pak uchovává odkaz na kameru, čehož využívá k nastavení kamery na správné místo, na němž začíná hráč, a také je zde odkaz na graf scény, ten sice v této třídě přímo využit není, předává se však jednotlivým objektům při jejich tvorbě, aby do něj mohly přidat své uzly.

Aktualizace stavu všech objektů obsažených v levelu probíhá v metodě `Update()`, jedná se o klíčovou činnost, většina průběhu herní logiky je delegována na vlastní objekty, a toto je místo, odkud je jejich stav a tím celý průběh hry aktualizován. V rámci metody `Update()` je voláním metody `Update()` objektu Player aktualizován stav hráče, a také je zde kontrolováno dosažení podmínky dokončení levelu, dále jsou zde volány metody zajišťující aktualizaci stavu nepřátel a stavebních prvků levelu.

Protože objekty tvořící prostředí levelu jsou statické, případně se jejich stav mění asynchronně na základě nějaké akce hráče, metoda `UpdateObjects()` pouze kontroluje stav a odstraňuje odkazy na ty nepoužívané. V metodě `UpdateEnemies()` je pak kromě rušení odkazů na neaktivní objekty také aktualizován stav nepřátel voláním jejich metody `Update()`, v níž je obsažena jednoduchá UI ovládající nepřátele.

Poslední metodou třídy *Level* je metoda `LoadLevel()`, ta má na starosti vytvoření levelu a incializaci všech objektů. Levely pro tuto hru jsou reprezentovány XML souborem s pro tyto účely

stanovenou sturkturou, tvorba levelů je tedy představována vytvářením popisu levelu v XML souboru za použití této struktury. Metodě `LoadLevel()` je jako parametr zadáno číslo levelu, jenž má být načten, podle tohoto údaje je vybrán správný XML soubor, který je parsován, a na základě získaných informací jsou pak vytvořeny požadované objekty.

Za účelem parsování XML souborů popisujících levely jsem přímo v rámci metody `LoadLevel()` vytvořil jednoduchý parser, vzhledem k jednoduchosti struktury popisu se mi toto řešení zdálo dostatečné.

```
<PowerUp>
  <PowerUpType>2</PowerUpType>
  <PositionX>1570</PositionX>
  <PositionY>570</PositionY>
  <TextureName>Armor1</TextureName>
</PowerUp>

        ...
switch (reader.Name)
{
        case "PowerUpType":
                if (reader.Read())
                {
                        powerupType = reader.ReadContentAsInt();
                }
        break;
        ...

ArmorBody ArmorFromXML = new ArmorBody(texture, pos, SceneGraph);
PowerUps.Add(ArmorFromXML);
```

Zdrojový kód 6.2: Ukázka načtení objetku z XML pomocí parseru. Reprezentace objektu v XML(nahoře), část parsovacího kódu(uprostřed), výsledný objekt(dole).

6.5 Bázové třídy

Hra obsahuje pět bázových tříd, které tvoří základ pro objekty použité ve hře. Zároveň by se dalo říci, že představují jakési kategorie objektů, jež jsou ve hře použity. Tyto třídy rozšiřují základní třídu *GameObject* poskytovanou enginem a obsahují metody implementující funkcionalitu společnou všem odvozeným objektům dané kategorie. V těch jsou pak už řešeny jen drobné odchylky a nastaveny konkrétní parametry. Jedná se o třídy *LevelObject*, *Character*, *Weapon*, *Projectile* a *PowerUp*, každé z nich bude věnována stručná podkapitola.

6.5.1 LevelObject

Tato třída je základem pro všechny objekty představující prostředí levelu, jako vlastní platformy, překážky apod. Zároveň se jedná o jedinou z bázových tříd, která je přímo používána k instancování objektů. Protože objekty levelu samy o sobě žádnou aktivní činnost nevykonávájí, je tato třída velmi jednoduchá, obsahuje pouze dva typy konstruktorů, v nichž jsou nastaveny potřebné vlastnosti objektu. Dále metodu `GetBounds()`, jež vrací Rectangle ohraničující tento objekt a metodu `CreateNode()`, jež se stará o vytvoření uzlu reprezentujícího objekt a jeho vložení do grafu scény. Jedinou odvozenou třídou rozšiřující *LevelObject* je třída *Door*, tu si přiblížíme později.

6.5.2 Character

Jak už její název napovídá, jedná se o třídu tvořící základ pro postavy ve hře, tedy hráče a nepřátele. Obsahuje tedy atributy jako aktuální a maximální počet hitpointů a armoru, rychlost pohybu, sílu skoku, směr pohledu/střelby nebo atribut indikující, zda je postava živá či mrtvá. Také zde najdeme odkaz na drženou zbraň. Tato třída také obsahuje metodu pro tvorbu a přidání uzlu do grafu scény, navíc však, protože postavy jsou jako jediné reprezentovány animovaným uzlem, tato třída ve své metodě `Update()` i řídí animaci svého uzlu. V souvislosti s uzly grafu scény je zde ještě metoda `GetNode()`, pomocí níž lze získat odkaz na uzel objektu pro použití vně tohoto objektu. Dále tato třída obsahuje několik variant metod ovládajících pohyb postavy, a to v modifikaci zohledňujících kolize nebo přímo určující objekt do kolize zapojený. Třída *Character* obsahuje také metodu implementující smrt postavy a metodu simulující zásah postavy projektilem.

6.5.3 Weapon

Třída *Weapon* představuje základní třídu pro zbraně. Opět obsahuje množství atributů, z těch zajímavějších bych zde chtěl zmínit atribut offset, jenž ve formě dvourozměrného vektoru určuje pozici zbraně ve vztahu k jejímu držiteli. Dále pak atributy `ShootTimer` a `ShootDelay`, určující prodlevu mezi výstřely a nakonec seznam `Projectiles`, v němž jsou uchovávány projektily z této zbraně vystřelené. Zbraň se tedy stará o projektily, jež z ní byly vystřeleny, jejich stav je aktualizován v metodě `Update()`, z této metody je také volána metoda `UpdateDelay()`, která ovládá čítač určující prodlevu mezi výstřely. Toto opatření jsem se rozhodl do hry zahrnout především kvůli zlepšení hratelnosti zvýšením rozdílů mezi zbraněmi a také k omezení schopnosti nepřátel, kteří by jinak byli schopni střílet mnohem rychleji než hráč. Samozřejmě je v této třídě obsažena i metoda `Shoot()` pro vlastní vystřelení zbraně, její implementace je však ponechána až na konkrétní odvozené třídy.

6.5.4 Projectile

Tato třída tvoří základ pro objekty představující různé druhy střeliva zbraní. Opět zde najdeme několik atributů určujících důležité vlastnosti projektilu a to především rychlost, maximální dolet a způsobené poškození. Nejdůležitější metodou této třídy je metoda `Update()` zajišťující vlastní let projektilu a především ošetřující kolize s různými druhy objektů. Dále je zde metoda zapouzdřující akce spojené se zánikem projektilu po kolizi a metoda `ActivateProjectile()`. Tato metoda aktivuje neaktivní projektil, zvolil jsem totiž řešení, kdy se místo rušení a opětovného vytváření projektilů vytváří pro každou zbraň pouze omezené množství projektilů a ty se poté recyklují, čímž se šetří zdroje.

```
switch (Hit.ObjectType)
{
       case GameObjectTypes.Character:
              Character target = (Character)Hit;
              target.GotHit(damage);
              RemoveProjectile();
              break;
       case GameObjectTypes.LevelObject:
              ...
```

Zdrojový kód 6.3: Ukázka řešení reakce projektilu na kolize.

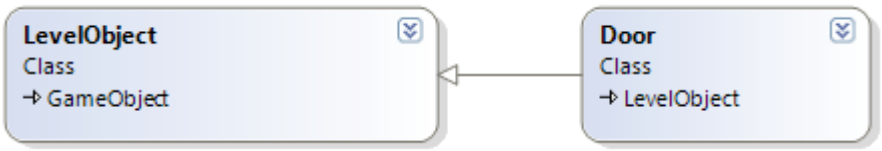
6.5.5 PowerUp

Poslední bázovou třídou je třída *PowerUp*, jedná se zároveň o jedinou třídu reprezentující custom objekt z pohledu enginu (viz. Kap 5.6). Tato třída je velmi podobná třídě *LevelObject* a slouží především k logickému oddělení power upů od obyčejných objektů. Obsahuje dva druhy konstuktorů v závislosti na tom, zda je třeba určovat směr natočení objektu či nikoliv a metody pro přidání uzlu do grafu scény a pro odstanění objektu ze scény po jeho aktivaci hráčem.

6.6 Finální třídy

Tyto třídy jsou přímo použity ve hře, protože jich je poměrně velké množství a většinou se třídy se společným základem liší především různými parametry, rozhodl jsem se zde popsat vždy celou skupinu tříd a na zajímavé části určitých tříd poukázat v rámci těchto podkapitol. Pouze třídy herních postav budou popsány v samostatných kapitolách, protože oproti ostatním jsou mnohem složitější.

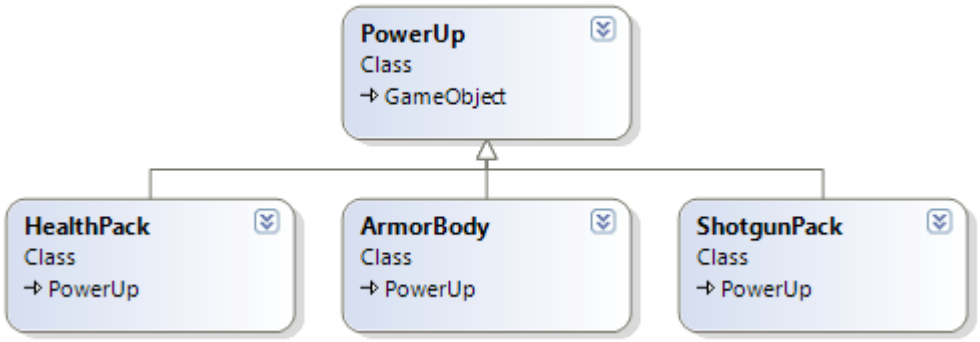
6.6.1 Zničitelné objekty



Obrázek 6.3: Třídy rozšiřující LevelObject.

Jedná se o jedinou třídu s názvem *Door*, která je jednoduchým rozšířením třídy *LevelObject*. Tato třída obsahuje pouze jedinou metodu, která zajišťuje odstanění objektu ze scény v případě zásahu.

6.6.2 Power Upy



Obrázek 6.4: Třídy rozšiřující PowerUp.

Ve hře jsou tři druhy power-upů, a to lékárnička obnovující hitpointy, armor přidávající armorpointy a brokovnice, kterou je možno vyměnit za základní zbraň (viz dále). Lékarnička a armor se liší pouze hodnotou atributu určujícího kolik přidají k hodnotě daného atriburu hráče. Brokovnice pak navíc obsahuje textury potřebné pro vytvoření nové instance zbraně.

6.6.3 Zbraně

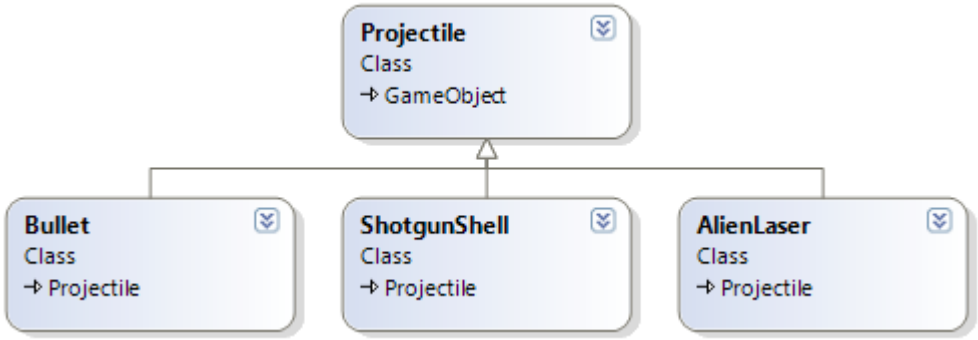
Obrázek 6.5: Třídy rozšiřující Weapon

Třídy představující konkrétní zbraně jsou rozšířeními bázové třídy Weapon a liší se především hodnotou atributu `ShootDelay`, a tím jaký typ projektilu generují. Každá má vlastní implementaci metody `Shoot()`, ta u všech pracuje podobně a to tak, že nejprve určí na základě otočení vlastníka a offsetu zbraně místo, na kterém se má projektil poprvé objevit a poté, v závislosti na počtu už vytvořených projektilů, buď generuje nový, nebo reaktivuje starý projektil. Přehled zbraní je uveden v tabulce:

Název zbraně	ShootDelay	Typ projektilu
9mm pistol	10	Bullet
Shotgun	20	Shotgun shell
Robot arm weapon	15	Alien laser

Tab 6.2: Typy zbraní.

6.6.4 Projektily



Obrázek 6.6: Třídy rozšiřující Projectile

Projektily jsou rozšířením třídy *Projectile*, nepřidávájí žádnou funkčnost, pouze jsou pro ně nastaveny vlastnosti typické pro konkrétní projektil, ty budou opět popsány v tabulce.

Název	Poškození	Rychlost	Dolet
Bullet	20	5	400
Shotgun shell	50	5	200
Alien laser	33	6	400

Tab 6.3: Typy projektilů.

6.6.5 Hráč

Postava hráče je představovaná třídou *Player*, jež je jednou ze dvou tříd rozšiřujících bázovou třídu *Character*. Pro hráče jsou oproti ostatním postavám dostupné dvě hlavní akce, jejichž implementací se hlavně zabývá tato třída, jedná se o schopnost skákat a možnost sebrat power-upy.

Začneme druhou jmenovanou akcí, ta je implementována v rámci metod pro pohyb doleva a doprava, kdy jsou volány příslušné metody třídy Character, vracející v případě kolize objekt s nímž ke kolizi došlo. Na základě typu tohoto objektu se pak rozhodne o další akci. Reakce na jednotlivé typy power-upů jsou implementovány ve zvláštních metodách, které mají na starosti úpravy příslušných atributů, a v případě metody pro změnu zbraně pak odstranění staré a vytvoření instance nové zbraně a její připojení k hráči.

Základem implementace skákání je několik pomocných atributů pro určení zda je hráč ve fázi skoku či na zemi a metody `PmoveUp()`, která na základě těchto informací provádí vlastní pohyb postavy nahoru a kontrokuje kolize. Nastavování výše zmíněných atributů i volání metody pro pohyb nahoru je uskutečněno z metody `Update()`.

6.6.6 Nepřátelé

Druhým typem postav hry jsou nepřátelé, ty představuje třída *NPC*, jež je opět rozšířením třídy *Character*, různé druhy nepřátel se liší pouze hodnotami základních atributů jako hitpointy a armor, proto stačí tato jediná třída.

Jejím hlavním účelem je implementovat jednoduchou umělou inteligenci, zajišťující chování nepřátel, aby to bylo možné, musí nepřítel mít možnost detekovat jak okolní objekty levelu, tak vlastního hráče, a proto tato třída obsahuje odkaz na objekt hráče i na seznam všech objektů v levelu. Samotné chování jsem rozdělil na tři základní části, pohyb, detekci hráče a detekci nebezpečných míst. Většina této logiky je založena na vyhodnocování množství podmínek, nebudu zde tedy konkrétně popisovat jednotlivé úkony a pokusím se jen přiblížit základní myšlenku těchto činností.

Pohyb nepřítele je řízen z metody `Update()`, je využito metod pro pohyb implementovaných ve třídě *Character* a detekce kolizí, chování se liší také v reakcí na to, zda je či není detekován a zaměřen hráč.

Detekce hráče probíhá na základě rozdílů pozic hráče a nepřítele, jsou při ní však brány v úvahu objekty levelu, proto nepřátelé takzvaně nevidí přes zdi.

Poslední detekovanou věcí jsou nebezpečná místa, tedy taková, kde by mohlo dojít ke smrti nepřítele. Typicky se jedná o propasti mezi platformami. Toto je řešeno tak, že při každém kroku nepřítel kontroluje bod na úrovni země o několik kroků dopředu a vyhodnocuje zda tam ještě je platforma, či nikoliv.

Název	Zdraví	Armor	Zbraň
Zombie voják	50	0	9mm pistol
Mutant voják	100	0	9mm pistol
Mutant civil	100	0	9mm pistol
Alien robot	100	100	Robot arm weap.

Tab 6.4: Typy nepřátel.

6.7 Třída Game1

Stejně jako ve všech na XNA frameworku založených aplikacích, i tato má hlavní třídu *Game1*. V případě této hry však třída Game1 slouží spíše jako vstupní bod aplikace. V konstruktoru této třídy jsou vytvořeny všechny komponenty představující jednotlivé části enginu, na němž je tato hra postavena, a také je na tomto místě vložena do správce obrazovek úvodní obrazovka.

V metodě `Initialize()` je pak načteno uložené nastavení hry a na jeho základě jsou provedeny potřebné akce tak, aby stav hry odpovídal poslednímu uloženému. To se týká především nastavení zvuku a postupu ve hře. V metodě `LoadContent()` jsou do správce zvuků načteny zvukové efekty, v případě této hry se jediný, a to zvuk střelby. Více využití třída *Game1* v této aplikaci nemá, neboť většinu funkcí zajišťují součásti enginu.

6.8 Ukládání a načítání

V kapitole věnované hře bylo několikrát zmíněno ukládání a načítání nastavení a postupu ve hře. Pro zajištění této důležité činnosti jsem použil jedinnou součást, která není mojí prací. Jedná se o modul *EasyStorage*, který je volně poskytován a zajišťuje právě funkce spojené s ukládáním za použití zařízení IsolatedStorage, tento modul je popsán v [22].

V této hře jsou ukládány dva typy informací, a to nastavení a postup ve hře. Rozhodl jsem se tyto informace oddělit, a proto jsou ukládány do oddělených souborů, což modul *EasyStorage* umožňuje. Nastavení je ukládáno do souboru `YourGame_Options` a postup ve hře do souboru `YourGame_Game`, názvy těchto souborů jsou nastaveny ve třídě *Saving*, jež je k tomuto účelu vytvořena v rámci této aplikace.

Vlastní načítaní, jak postupu ve hře, tak nastavení probíhá ve třídě *Game1* voláním metod `LoadOptions()` a `LoadGameProgress()`. Ukládání pak probíhá na příslušných obrazovkách jak bylo popsáno v kapitole 6.2.

7 Závěr

7.1 Zhodnocení výsledků projektu

V této práci byla stručně představena mobilní platforma Windows Phone 7 a framework XNA, používaný k vývoji her pro tuto platformu. Dále byl čtenář uveden do problematiky herních enginů a jejich využití při vývoji her. Následně byl popsán vývoj jednoduchého 2D herního enginu pro platformu Windows Phone 7 a jeho využití při vývoji hry.

Demonstrační aplikace prokázala, že engine vyvinutý v rámci této práce je funkční a použitelný, což je možno považovat za uspokojivý výsledek, vzhledem k tomu, že všechny jeho části jsem vytvořil sám.

To samé se dá prohlásit i o demonstrační aplikaci, kterážto jako jednoduchá hra plní svůj účel a poskytuje dva zcela funkční dokončitelné levely.

Při její tvorbě se pak nesporně ukázaly výhody použití herního enginu, které i v takto jednoduché hře přineslo výraznou časovou úsporu a zjednodušení práce. V souvislosti s tím bych chtěl vyzdvihnout především usnadnění tvorby menu a uživatelského rozhraní, a také možnost využití poskytovaného systému kolizí.

7.2 Další vývoj projektu

Ačkoliv je engine funkční a pro jednoduché hry použitelný, pro zvýšení jeho použitelnosti a konkurenceschopnosti by bylo třeba provést ještě mnoho optimalizací a úprav, a to především z důvodu, že se jednalo o můj první projekt v této oblasti a především také první větší projekt psaný stylem objektově orientovaného programování. V průběhu vývoje jsem získal cenné zkušenosti a v současné době bych u vývoje některých částí enginu postupoval jinak.

Pokud se jedná o konkrétní návrhy dalšího vývoje, tak bych uvedl například tyto:

- Úprava správce herních obrazovek a rozšíření jeho funkcí například o přechody mezi obrazovkami.
- Přidání možnosti více vrstev scény v případě grafu scény.
- Podpora složitějších gest ve správci ovládání.
- Schopnost automaticky zarovnávat prvky GUI za použití různých layoutů.
- Implementace Level editoru v rámci enginu.

Pokud bychom se bavili o rozšíření demonstrační aplikace na úroveň konkurenceschopné hry, pak by se jednalo především o úplné přepracování grafické stránky a to někým, kdo má v této oblasti potřebný talent. Bylo by také třeba optimalizovat hratelnost.

Na tento projekt bych chtěl navázat dalším využíváním upraveného enginu k tvorbě jednoduchých, ale plnohodnotných her, které by bylo možno umístit na marketplace a dosáhnout tak jejich rozšíření mezi hráče.

Literatura

[1] Microsoft.: Microsoft Unveils Windows Phone 7 Series [online]. 15.2.2010 [cit. 17.1.2012]. Dostupný z WWW: <http://www.microsoft.com/presspass/press/2010/feb10/02-15MWC10PR.mspx>

[2] Microsoft.: Microsoft zahájil distribuci aktualizace Windows Phone 7.5 (Mango) [online]. 27.9.2011 [cit. 17.1.2012]. Dostupný z WWW: <http://www.microsoft.com/cze/presspass/msg/20110927_news1.mspx>

[3] Microsoft.: Nokia and Microsoft Announce Plans for a Broad Strategic Partnership to Build a New Global Mobile Ecosystem [online]. 11.2.2011 [cit. 17.1.2012]. Dostupný z WWW: <http://www.microsoft.com/presspass/press/2011/feb11/02-11partnership.mspx>

[4] Microsoft.: Metro Design Language of Windows Phone 7 [online]. 2012 [cit. 25. 4. 2012]. Dostupný z WWW: <http://www.microsoft.com/design/toolbox/tutorials/windows-phone-7/metro/>

[5] Petzold, Ch.: *Programming Windows Phone 7*. Redmond, Microsoft Press, 2010, ISBN 978-0-7356-4335-2

[6] Carter, Ch.: *Microsoft XNA Unleashed: Graphics and Game Programming for Xbox 360 and Windows*. Indianapolis, Sams, 2008, ISBN 0-672-32964-6

[7] Walker, M.: What is the XNA Framework [online]. 25.8.2006 [cit. 17.1.2012]. Dostupný z WWW: <http://blogs.msdn.com/b/xna/archive/2006/08/25/724607.aspx>

[8] Microsoft.: XNA Game Studio Express [online]. 2012 [cit. 17.1.2012]. Dostupný z WWW: <http://msdn.microsoft.com/en-US/library/bb200104(v=xnagamestudio.10).aspx >

[9] Microsoft.: What's New in This Release [online]. 2012 [cit. 17.1.2012]. Dostupný z WWW: <http://msdn.microsoft.com/en-US/library/bb417503(v=xnagamestudio.20).aspx>

[10] Microsoft.: What's New in This Release [online]. 2012 [cit. 17.1.2012]. Dostupný z WWW: <http://msdn.microsoft.com/en-US/library/bb417503(v=xnagamestudio.30).aspx>

[11] Microsoft.: What's New in XNA Game Studio 4.0 [online]. 2012 [cit. 17.1.2012]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/hh221584.aspx>

[12] Microsoft.: What's New in XNA Game Studio 4.0 Refresh [online]. 2012 [cit. 17.1.2012]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/bb417503.aspx>

[13] Gravelyn, N.: Life of an XNA Game [online]. 23.11.2008 [cit 18. 1. 2012]. Dostupný z WWW: <http://blog.nickgravelyn.com/2008/11/life-of-an-xna-game/>

[14] Microsoft.: Game Components [online]. 2012 [cit. 18.1.2012]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/bb203873.aspx#ID4EMAAC>

[15] Microsoft.: Game Services [online]. 2012 [cit. 18.1.2012]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/bb203873.aspx#GameServices>

[16] Hargreaves, S.: XNA Game Studio on Windows Phone [online]. 10.3.2010 [cit 18.1.2012]. Dostupný z WWW: <http://blogs.msdn.com/b/shawnhar/archive/2010/03/10/xna-game-studio-on-windows-phone.aspx>

[17] Zerbst, S., Duvel, O.: *3D Game Engine Programming.* Boston, Course Technology PTR, 2004, ISBN 1-59200-351-6

[18] Epic Games Inc.: Game Engine Technology by Unreal [online]. 2012 [cit. 18.1.2012]. Dostupný z WWW: <http://www.unrealengine.com/>

[19] Microsoft.: Additional Requirements for Specific Application Types [online]. 2012 [cit. 26.3.2012]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/hh184838(v=vs.92).aspx>

[20] Woolford, D.: Understanding and Using Scene graphs [online]. 15.6.2003 [cit. 26.3.2012]. Dostupný z WWW: <http://itee.uq.edu.au/~comp4201/scene_graphs_dsaw.pdf>

[21] Saijo, G.: Quick look at Retro Warfare 2 a retro platformer [online]. 27.8.2011 [cit. 3.4.2012]. Dostupný z WWW: <http://www.bestwp7games.com/quick-look-at-retro-warfare-2-a-retro-platformer.html>

[22] EasyStorage [online]. Aktualizováno 29.8.2010 [cit. 3.4.2012]. Dostupný z WWW: <http://easystorage.codeplex.com/>

Seznam příloh

Příloha 1. Obsah CD

- Diagramy tříd
  - Diagramy tříd enginu
  - Diagramy tříd demonstrační aplikace
  - Kompletní diagram tříd celého projektu
- Dokumentace enginu ve formě html stránek
- Plakát k projektu
- Text práce v elekronické podobě
- Výsledné produkty této práce
  - Engine ve formě dll knihovny
  - Demonstrační aplikace ve formě balíku xap
- Zdrojová data použitá v této práci
- Zdrojové kódy
  - Samostatné zdrojové kódy enginu ve formě projektu pro Visual Studio 2010
  - Zdrojové kódy celého projektu ve formě projektu pro Visual Studio 2010
- Návod k instalaci aplikace
- Uživatelská příručka k aplikaci