Ekonomická fakulta
Faculty of Economics

Jihočeská univerzita v Českých Budějovicích
University of South Bohemia in České Budějovice

Jihočeská univerzita v Českých Budějovicích
Fakulta ekonomická
Katedra aplikované matematiky a informatiky

Bakalářská práce

Vývoj aplikace s využitím XNA Frameworku

Vypracoval: Petr Špringer
Vedoucí práce: Mgr. Radim Remeš
České Budějovice 2018

JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH
Fakulta ekonomická
Akademický rok: **2016/2017**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Petr ŠPRINGER**
Osobní číslo:	**E15876**
Studijní program:	**B6209 Systémové inženýrství a informatika**
Studijní obor:	**Ekonomická informatika**
Název tématu:	**Vývoj aplikace s využitím XNA frameworku**
Zadávající katedra:	**Katedra aplikované matematiky a informatiky**

Zásady pro vypracování:

Cílem práce je vytvořit zvolenou aplikaci. Aplikace bude vyvíjena s využitím XNA frameworku.

Metodický postup:

1. Studium odborné literatury.
2. Publikace výsledků rešerše.
3. Návrh, popis vývoje a implementace aplikace.
4. Zhodnocení, vypracování doporučení a závěrů.

Rozsah grafických prací: **dle potřeby**

Rozsah pracovní zprávy: **40 - 50 stran**

Forma zpracování bakalářské práce: **tištěná**

Seznam odborné literatury:

1. **Drumm, L. (2012).** *Microsoft XNA 4.0 game development cookbook: over 35 intermediate-advanced recipes for taking your XNA development arsenal further.* **Birmingham: Packt Pub.**
2. **Nagel, C. (2016).** *Professional C# 6 and .NET Core 1.0* **USA: Wiley/Wrox.**
3. **Nathan, A. (2013).** *WPF 4.5 Unleashed.* **Indianapolis, IN, USA: Sams.**
4. **Sharp, J. (2015).** *Microsoft Visual C# Step by Step. 8.* **vydání. Redmont, WA, USA: Microsoft.**
5. **Troelsen, A., & Japikse, P. (2015).** *C# 6.0 and the .NET 4.6 Framework.* **7. vydání. New York, USA: Apress.**

Vedoucí bakalářské práce: **Mgr. Radim Remeš**
Katedra aplikované matematiky a informatiky

Datum zadání bakalářské práce: **16. ledna 2017**

Termín odevzdání bakalářské práce: **15. dubna 2018**

doc. Ing. Ladislav Rolínek, Ph.D.
děkan

L.S.

RNDr. Jana Klicnarová, Ph.D.
vedoucí katedry

V Českých Budějovicích dne 16. ledna 2017

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47 zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to - v nezkrácené podobě vzniklé vypuštěním vyznačených částí archivovaných Ekonomickou fakultou - elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

Datum: 10. 4. 2018

Podpis studenta

(v písemné verzi vlastnoruční podpis ve všech kopiích!)

Poděkování

Rád bych poděkoval vedoucímu své bakalářské práce, panu Mgr. Radimu Remešovi, za všechny rady a připomínky při tvorbě bakalářské práce. Dále bych pak chtěl poděkovat rodině a přítelkyni za podporu, které se mi dostává při studiu na vysoké škole.

Obsah

Úvod

Cíl této práce je zaměřen na vývoj aplikace s využitím XNA frameworku. Práce není zaměřená pouze na XNA framework, ale také se zaměřuje na vysvětlení problematiky obecného programování her a to z důvodů, že framework XNA v dnešní době již není dále podporovaný. Na začátku práce se čtenář seznámí s .NET frameworkem a jeho kompilací. Dozví se obecné informace o programování a podpoře daného frameworku.

Dále se čtenář seznámí s programovacím jazykem C# a jeho historií. Práce se bude věnovat také metodice objektově orientovaného programování s uvedením konkrétních příkladů z reálného světa. Pozornost bude věnována nejenom základnímu popisu tříd, metod a instancí. Ale také se čtenář dozví informace o zapouzdření, kde bude poučen o identifikátorech přístupu a jejich vhodném použití. Pozornost bude také věnována dědění a polymorfismu, který často bývá vnímán jako velice složitý mechanismus, když ve skutečnosti se nejedná o nic složitého. Následná kapitola seznámí čtenáře s obecným pohledem na vývoj her. Předá informace o tom, co všechno by měl vývojář hry umět předtím, než si vybere konkrétní programovací jazyk a framework. V rámci kapitoly se také čtenář dozví základní informace o frameworku XNA a jeho základních metodách, které budeme využívat při vývoji naší hry. Například se hráč dozví, jakým způsobem se do hry načítají zvuky a grafika. Dozví se, v jaké metodě může vykreslovat herní obrázky a animace, kde může například provést odečtení aktuálních životů hráče, nebo změnit jeho herní pozici.

Po seznámení s obecnými přístupy a XNA frameworkem se čtenář dozví konkrétní příklady návrhu aplikace, bude vědět, co je dobré udělat před tím, než se vrhne do samotného programování hry, na co si dát pozor, popřípadě čeho se vyvarovat. Hlavní částí je pak rozbor samotného zdrojového kódu. Zde se čtenář dozví podrobné informace o jednotlivých třídách a jejich metodách.

1 .NET Framework

K tomu, abychom mohli začít programovat, potřebujeme programovací jazyk a vývojové prostředí (IDE), ve kterém budeme moci psát náš vlastní kód. K tomu účelu je vytvořeno velké množství jazyků, vývojových prostředí, knihoven a různých frameworků. K tomu účelu nám bude sloužit .NET framework. Jedná se o obsáhlou softwarovou platformu určenou pro vývoj velkého množství rozdílných aplikací. S pomocí .NET frameworku jsme schopni vytvořit běžné aplikace pro Windows, webové aplikace nebo aplikace pro android.

Součástí frameworku není jen sada nejrůznějších knihoven, ale takzvané „běhové prostředí“ (CLR – Common Language Runtime), které se stará o funkčnost a kompilaci aplikací. Obrovské množství funkcí, které pro nás obstarává CLR, zajistí to, že se nemusíme zabývat psaním běžných věcí, které se neustále opakují, jsou stereotypní a náchylné na chyby. CLR zaručí, že naše aplikace budou rychlé a bezpečné. .NET framework je pohodlný a za pomoci chytře zvoleného programovací prostředí i méně chybový (Richter, 2003).

1.1 Kompilace

Součástí samotného programování je také kompilace napsaného kódu. To nám pomůže zjistit, jestli v našem kódu nejsou drobné překlepy, možné chybějící středníky a závorky. Pro lepší pochopení si více vysvětlíme pojem kompilace.

1.1.1 Klasická kompilace

Klasická kompilace například u C++ probíhá tak, že výstupem kompilace je samotný strojový kód, který je přímo závislý na platformě operačního systému, na kterém aplikace poběží. Jedná se o velmi rychlý proces, protože výstup kompilátoru je pro proces prakticky připravený, avšak obsahuje také různé nevýhody.

Hlavní nevýhodou je větší pravděpodobnost chyby v samostatné aplikaci, což je zdrojem bezpečnostních rizik. Například u C++ se často pracuje s pointery, to znamená, že kodér může psát na libovolné místo a to vede k častým chybám, které mohou způsobit pád systému (Herceg, 2009).

1.1.2 Kompilace v .NET Frameworku

V případě, že kompilujeme aplikaci, kterou jsme napsali v .NET frameworku, tak proces kompilace je lehce odlišný. Výsledkem kódu totiž není samotný strojový kód, ale tzv. CLI (Common Intermediate Language). Hlavní rozdílem a samotnou výhodou kompilace .NET frameworku je to, že výstup se stává nezávislým na platformě a tak je možné ho spustit kdekoliv, kde je nainstalované běhové prostředí pro .NET framework.

„*Kód přeložený do jazyka CIL se spolu s přidanými datovými soubory (obrázky atd.) zabalí do tzv. Assembly, což je soubor s příponou exe či dll. Na první pohled výsledné soubory vypadají stejně jako výstup kompilátoru v C++.*“ (Herceg, 2009)

Rozdíl nastává až při samotném spuštění, kdy se provede částečná kompilace Assembly, která kompiluje jenom to, co je pro spuštění aplikace důležité a následně aplikaci spustí. To vede k tomu, že výsledkem je aplikace, která má lepší výkon, než kdyby se kompiloval celý kód. Kompilace většinou probíhá na úrovni metod, takže pokud nastane situace, že je zapotřebí zavolat určitou metodu, která ještě není zkompilována, použije se JIT (Just-in-time) překladač, který ji při prvním spuštění přeloží a uchová data pro další možné použití. To umožňuje překládat pouze tu část kódu, která je momentálně zapotřebí (Herceg, 2009).

2 C#

C# je moderní, univerzální, objektově orientovaný programovací jazyk. V Csharpu jsme schopni napsat mnoho různých aplikací. C# se používá při vývoji moderních aplikací běžících na stolních počítačích a dokonce se může využít pro tvorbu sofistikovaných back-endů webových aplikací (“C# Developer Job Description Template”, ©Copyright2010-2018).

Mimo vysoké podpory tvorby komponentových systémů je C# navržen tak, aby zvládal objektově orientované programování. Popisu objektově orientovaného programování se budeme zabývat níže.

C# převzal prvky z jazyku C++, kterými jsou například přetěžování operátorů, uživatelem definované konverze, skutečná obdélníková pole a sématika předávání odkazem. V Csharpu ovšem chybí vlastní runtimová knihovna a to z důvodů, že využívá knihovny tříd v rámci .NET, včetně využívání práce s konzolí, práce se sítí a soubory (Drayton, Neward, & Albahari, 2003).

2.1 Historie

C# vytvořili autoři Anders Hejsberg, Scott Wiltamuth a Petter Golde. Tvůrci popisovali jazyk jako jednoduchý, moderní, objektově orientovaný a typově zabezpečený, který vychází z jazyků C a C++. Má mnoho syntaktických podobností s jazyky C++ a Java.

První zmínka o novém jazyku připravovaným společností Microsoft se objevila v roce 1998. To co se o jazyku v té době vědělo, bylo to, že má být velice podobný současnému jazyku Java. V roce 2000 Microsoft umlčel veškeré spekulace vydáním konkrétních specifikací o C#. To zapříčinilo také rychlé předběžné vydání verze SDK rámce .NET, což také zahrnovalo kompilátor pro C# (Drayton, Neward, & Albahari, 2003).

V roce 1999 se Microsoft zmiňuje o technologii .NET platformě, která se skládá z tak zvaného CLR (Common Language Runtime) a z knihovny tříd systému .NET Framework (někdy nazývaných Base Class Library - BCL).

V roce 2000 předvedl Bill Gates svou vizi platformy .NET na konferenci Forum 2000. Pár dní poté byl vydán jazyk C# společností Microsoft. V první polovině roku 2001 byl dokončen kompilátor pro jazyk C# a celá jeho platforma. .NET 1.0 byla oficiálně uvolněna pro veřejnost 15. ledna 2002 (partnerům Microsoftu byla dostupná již mnohem dříve, včetně všech vývojových beta verzí). V roce 2002 byly dispozici kompilátory jazyka C# verze 1.1 (např. Visual Studio 2003). (Černohorský, 2003)

V dnešní době je C# velice rozšířený a to hlavně v herním průmyslu. Existuje celá řada frameworků, které jsou vytvořeny pro tento jazyk. Například WaveEngine, CryEngine,MonoGame a Unity 3D.

2.2 Objektově orientované programování (OOP)

Objektově orientované programování označováno taky jako OOP je moderní metodikou vývoje softwaru, kterou můžeme využít u velkého množství programovacích jazyků. Můžeme říct, že se jedná o určitý způsob filozofického myšlení, který můžeme využít nejen při programování. OOP se zabývá otázkou, jakým způsobem probíhá komunikace uvnitř programu. Zajímá se o to, jaký způsobem komunikují jednotlivé části mezi sebou. Využití OOP tedy najdeme při tvorbě jak malé aplikace, tak při navrhování složité databázové struktury. Nejedná se jenom o doporučovanou techniku, nebo konkrétní strukturu programu, ale jedná se o zcela nový způsob myšlení při vývoji nových programů.

Při použití OOP jsme tedy schopni navrhnout a naimplementovat strukturu programu, tak abychom byli schopní znovu použít danou struktury pro budoucí vývoj. Každou komponentu, kterou vytvoříme, můžeme pak dále používat ve vývoji. To nám šetří spoustu času a dává nám prostor pro tvorbu složitějších a náročnějších věcí, bez nutnosti opakovaného programování jednotlivých částí (Čápka, ©2018).

2.2.1 Třída

Základem OOP je v podstatě simulace reality, tak jak jí známe z reálného života. Při návrhu se snažíme odpoutat od toho, jak program může vidět počítač a snažíme se ho napsat spíše z pohledu programátora.

Hlavní jednotkou OOP je třída. Pro každou třídu definujeme datové typy, vlastnosti a metody. V atributu se uchovávají základní informace o třídě, se kterými program pak může pracovat a to pomocí vytvořených vlastností a metod.

2.2.1.1 Metoda

Díky metodám můžeme pracovat s atributy objektu popřípadě provádět různé operace týkající se dané třídy. Ukážeme si to na konkrétním příkladu. Třídou může být člověk. Základní atributy člověka, pak bude jméno, věk, datum narození a barva očí. Vlastnost pak bude například zvýšení věku nebo změna příjmení. Základní metody by pak mohli být, řekni svůj věk, pozdrav, jdi spát.

2.2.1.2 Konstruktor

Jedná se o speciální metodu, která má specifický způsob zápisu. Název metody je shodný s názvem třídy a je vždy veřejně přístupný. Jeho specifickou vlastností je to, že se volá okamžitě po vytvoření instance a využívá se především k nastavení důležitých atributů.

2.2.2 Instance

Instance je konkrétní datový objekt, kde už přesně definujeme atributy a vlastnosti dané vytvořené třídy. U třídy člověk bude instancí třeba Petr, jehož věk bude 20 let a bude svobodný. Díky dobře vytvořené třídě, můžeme pak vytvořit spoustu instancí, které mezi sebou můžou také spolupracovat (Čápka, ©2018).

2.2.3 Zapouzdření

Zapouzdření se používá především k tomu, abychom byli schopní vytvořit bezpečně běžící program, který nebude lehké rozbít z venku. Díky zapouzdření můžeme schovat některé metody a atributy, takže při vytváření instancí s těmito daty nebudeme moc dále přímo pracovat. Upravovat je budou pouze metody, které zpřístupníme z venku, takže tím docílíme toho, že dané atributy se budou měnit pouze pomocí našich bezpečně vytvořených metod a vlastností.

„Nevíme, jak to uvnitř funguje, ale víme, jak se navenek chová a používá. Nemůžeme tedy způsobit nějakou chybu, protože využíváme a vidíme jen to, co tvůrce třídy zpřístupnil.“ (Čápka, ©2018)

Výše uvedený text je zjednodušený popis zapouzdření. Například u třídy člověk, která bude mít atribut datumNarození a plnoletost, tak při změně věku by mohla přestat platit podmínka plnoletosti, takže atribut datumNarození nebude veřejně dostupný, ale pomocí vytvořené metody *ZměnVěk*, budeme schopni atribut datumNarození bezpečně měnit s tím, že vždy bude platit atribut plnoletost, který bude vždy překontrolován v metodě, takže vždy bude mít správnou hodnotu.

„Zapouzdření tedy donutí programátory používat objekt jen tím správným způsobem. Rozhraní (interface) třídy rozdělí na veřejně přístupné (public) a veřejně nepřístupné (private).“ (Čápka, ©2018)

2.2.3.1 Public

Public je identifikátor přístupu, kterým se označují veřejně dostupné metody a atributy tříd. To znamená, že po vytvoření instance k nim budeme moc veřejně přistupovat.

2.2.3.2 Private

Private je identifikátor přístupu, kterým se označují metody a atributy, ke kterým nebudeme moc přistupovat, zobrazovat a ani měnit u dané instance.

Hlavním důvodem proč se využívá blokování přístupu k metodám a atributům z venku je to, že se tím dokáže vytvořit třída, která bude přehlednější a také zabezpečenější. Nebude moc dojít k neoprávněné změně atributu.

2.2.3.3 Protected

Posledním identifikátorem přístupu je protected. Funguje stejně jako identifikátor private s tím rozdílem, že metody a atributy budou přístupné nejenom pro samotnou třídu, ale také pro třídy, které z ní budou dědit. Více o dědičnosti se dozvíte v další kapitole.

2.2.4 Dědění

Dědičnost umožňuje vytvářet nové třídy z aktuálně vytvořených tříd, s možností rozšíření, nebo úprav některých metod, které jsou definované v jiných třídách. Třída, ze které se dědí, se nazývá základní třída a nově vytvořená se nazývá odvozená třída. Odvozená třída může mít pouze jednu přímou základní třídu.

Když definujete třídu odvozenou od jiné třídy, odvozená třída implicitně získá všechny členy základní třídy, s výjimkou konstruktorů. Odvozená třída může takto znovu použít kód v základní třídě, aniž bychom ji museli znovu implementovat. V odvozené třídě můžete přidat další členy. Tímto způsobem se odvozená třída rozšiřuje o funkčnosti základní třídy.

Představme si to na konkrétním příkladu, kde budeme muset naprogramovat jednotlivé pracovní pozice. Bude tam účetní, vedoucí provozu a manager. Všechny role budou mít určitě společné atributy, například jméno, příjmení, věk a pohlaví. Společné metody by mohli být třeba příchod do práce a odchod z práce. Bylo by tím pádem zbytečné vytvářet v každé třídě tyhle stejné atributy a metody. Proto je praktické vytvořit třídu zaměstnanec, ve které nadefinujeme výše zmíněné atributy a metody. Třídy účetní, vedoucí provozu a manager potom budou odvozené třídy z hlavní třídy zaměstnanec (“Inheritance (C# Programming Guide)| Microsoft Docs”, 2018).

2.2.5 Polymorfismus

Polymorfismus je často označován jako třetí pilíř objektově orientovaného programování po zapouzdření a dědičnosti. Polymorfismus je řecké slovo, které znamená "mnohotvárný".

Díky polymorfismu můžeme vytvářet takzvané abstraktní třídy. To je taková třída, která nikdy nebude mít vlastní instanci. V našem příkladu pracovníků by abstraktní třída byla třída zaměstnanec. Důvod proč se tvoří abstraktní třídy, je ten, že v našem programu budeme schopní se všemi různými zaměstnanci jednat stejně. Každý zaměstnanec bude pracovat, ale účetní dělá jinou práci než náš vedoucí. To znamená, že v třídě zaměstnanec vytvoříme metodu *Pracuj*. Když vytvoříme odvozenou třídu účetní a vedoucí pracovník. Obě třídu budou mít stejný název pro metodu *Pracuj* s rozdílnou funkčností. Účetní bude moc účtovat a vedoucí kontrolovat (“Polymorphism (C# Programming Guide) | Microsoft Docs”, 2018).

3 Obecně o vývoji her

Vytvoření vlastní hry je sen každého vášnivého hráče. Ale málo kdo tuší, co všechno je potřeba zvládnout, aby tvorba hry byla úspěšná. Existuje spousta diskuzí, kde se uživatelé ptají, jaký programovací jazyk bych měl umět, abych si mohl naprogramovat vlastní hru. Vtipné je pak i to, že správnou odpovědí může být prakticky jakýkoliv programovací jazyk. Dovolím si poznamenat, že textovou hru jsme schopni vytvořit i v příkazové řádce. Správnou otázkou by mělo být, co všechno musím umět, abych byl schopný vytvořit vlastní hru.

Na začátku je důležité si uvědomit, jakou hru vlastně chceme vytvářet. Má-li to být strategická hra, tahová hra, střílečka, nebo dokonce online hra. Pokud již víme, jaký žánr hry bychom chtěli vytvořit, tak druhou otázku, kterou bychom si měli ujasnit, je to, jaký bude cíl hry.

Pokud jsme schopni si sami sobě odpovědět na výše zmíněné otázky, tak až teprve teď bychom se měli zajímat o to, v čem budeme tvořit naší hru a jaké základní znalosti bychom měli mít. Například při tvorbě online hry bychom určitě měli mít přehled nejen o obecném programování, ale také o databázových systémech. Dalším parametrem je pak výběr programovacího jazyka, kterých je celá řada, například Java, Csharp a nebo Python. Každý jazyk má své výhody a nevýhody a taky záleží na tom, pro jakou platformu chcete hru vytvářet, nebo jaký herní engine chcete použít.

3.1 Návrh hry

Hlavní oblastní vývoje hry je pak samotný návrh, ve kterém by měl být zaznamenaný každý detail hry. To můžou být i umístění a pojmenování obrázku, hudby, zvukových efektů, názvy hlavních i vedlejších objektů. Promyslet si a zaznamenat každý důležitý atribut, vlastnost a funkci jednotlivých tříd. Důležité je také si promyslet základní prvky samotné hry, například základní menu hry a herního prostředí.

3.2 Výběr herního enginu

Když už máte představu, jakou hru chcete tvořit, tak je důležité zvolit si správný herní engine. Před samotným výběrem je vhodné si zadat základní kritéria, která nám pomůžou vybrat správný engine. Například jaká je podpora různých platforem, ve kterých se hry tvoří. Je dobré si zjistit i informace o tom, jak moc je podporovaný a aktuální námi vybraný engine. Posledním vhodným kritériem můžou být hardwarové a softwarové nároky, což se v dnešní době může zdá nepodstatné, ale pořád se najdou někteří hráči, kteří nemají nejnovější herní počítače, takže je důležité myslet i na ně.

3.3 Game design

Vývojáři by měli myslet také na samotný design a estetiku hry. Výborný vzhled hry může způsobit to, že si ji hráč bude chtít zahrát, jen kvůli tomu, jak se mu hra na první pohled líbí. Proto při vývoji hry je nutné myslet nejen na funkčnost, ale také na vizuální vzhled, který je pro většinu hráčů v dnešní době důležitý. To znamená, že při vývoji her, nejsou důležití jenom programátoři, ale také grafici (Intel, ©2018).

4 XNA

XNA framework je platforma pro tvorbu her od společnosti Microsoft, která usnadňuje vývoj moderních her. Je poskytovaná zdarma a umožňuje vývoj her jak pro osobní počítače, tak i pro mobilní zařízení. Skládá se ze dvou základních komponent a to XNA Game Studio Express a XNA Framework.

XNA Game Studio Express je volně dostupné vývojové prostředí pro platformu XNA. Požaduje operační systém Microsoft Windows a nainstalované Visual Studio C# Express. To rozšiřuje funkcionalitu Visual Studia o podporu správy herního obsahu, například zvuků, modelů, textur a efektů a možnost tvorby pro konzoli Xbox360.

XNA Framework je sada .NET knihoven přizpůsobených pro platformu .NET Frameworku 2.0 a DirectX9. Jde hlavně o programové rozhraní, které podporuje práci s grafickým herním obsahem, zvukem a především s uživatelským vstupem (Drumm, 2012).

4.1 Struktura XNA frameworku

XNA framework má svoji specifickou strukturu. Při vygenerování kódu pro tvorbu hry pro Windows se nám vytvoří dvě základní složky. První obsahuje samotnou hru a jmenuje se stejně tak, jak jsme pojmenovali naší hru. V téhle složce budeme ukládat náš zdrojový kód. Druhou vygenerovanou složkou je taková, která bude mít v názvu oproti první složce na konce ještě doplnění „Content“. V téhle složce budeme ukládat veškerou naší herní grafiku, hudbu a také fonty.

4.2 Třída Game1

Třída Game1, která obsahuje již předem vygenerované dvě proměnné graphics a spriteBatch. Graphics je typu GraphicsDeviceManager. Pomocí téhle třídy můžeme volat metody pro změnu velikosti herního okna, zapnutí a vypnutí fullscreenu. Důležitý je znát metodu *ApplyChanges*, kterou musíme zavolat pokaždé, když provedeme nějaké změny velikosti obrazovky. Proměnná spriteBatch je instance třídy SpriteBatch. Díky tomu budeme schopni pracovat s veškerým grafickým obsahem. Budeme pracovat se sprity. Jedná se o 2D grafiku, kterou budeme vykreslovat jak pozadí, tlačítka a dokonce i animace našich postaviček.

4.2.1 Konstruktor

Další automaticky vygenerovanou částí třídy Game1 je bezparametrický konstruktor, ve kterém se vytvoříme instanci proměnné graphics s parametrem instance třídy Game1. Pomocí vlastnosti Content, která je součástí balíčku XNA, nastavíme cestu ke kořenovému adresáři s herním obsahem. Konstruktor si můžete prohlídnout v ukázce kódu 1.

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
}
```

Ukázka kódu 1: Konstruktor třídy Game1

4.2.2 Metoda Initialize

Další metoda, kterou nalezneme při vytvoření projektu pro tvorbu naší hry je *Initialize*, která je zobrazena v Ukázce kódu 2. Můžeme jí přirovnat k takovému dalšímu hernímu konstruktoru. Jedná se o metodu, která se volá hned po vytvoření instance třídy Game1. Zde můžeme nastavit grafické požadavky pro naši hru, kterým může být nastavení rozlišení, fullscreenu a nebo například nastavení viditelnosti kurzoru myši. Na konci metody vždy musíme volat původní metodu předka *Initialize* a to tak, že zavoláme „base.Initialize()“.

```
protected override void Initialize()
{
    graphics.PreferredBackBufferWidth = 1024;
    graphics.PreferredBackBufferHeight = 768;
    graphics.IsFullScreen = false;
    graphics.IsMouseVisible = true;
    graphics.ApplyChanges();
    base.Initialize();
}
```

Ukázka kódu 2: Příklad nastavení grafického rozhraní hry

4.2.3 Metoda LoadContent

Ve třídě Game1 máme taky metodu *LoadContent*. Jak už z názvu vyplývá, metoda nám slouží pro načítání obsahu. Pro nás to tedy bude herního obsahu, kde budou všechny sprity, muzika a text. Také zde probíhá vytvoření instance pro spriteBatch. U větších projektů by však načtení veškerého herního obsahu mohlo znamenat přetečení operační paměti. Proto se tahle metoda taky může využít pouze pro vytvoření instance spriteBatch a načítání spritů a dalšího obsahu obsloužit v jiných třídách tak, aby se načítal vždy obsah dané oblasti hry.

4.2.4 Metoda Update

Update je jedna ze dvou nejdůležitějších metod naší hry. Jedná se o metodu, ve které se bude řešit naše herní logika v reálném čase. Zaznamenává se zde vstup z klávesnice, myši, popřípadě herního ovladače. Řeší se zde pohyb objektů ve hře, případně jejich kolize. Metoda se vykonává 60x za vteřinu. To znamená, že pokud nastavíme změnu vektoru o jeden pixel, naše postavička se posune o 60 pixelů za vteřinu daným směrem. Metoda je vytvořená společně s instancí třídy GameTime, která obsahuje údaje o čase hry. Pomocí metody *ElapsedGameTime*, můžeme zjistit, jaký čas uběhnul od posledního updatu a pokud by nás zajímal celkový čas hry, tak pro to je vytvořená metoda *TotalGameTime*. To nám umožní pracovat s hrou v reálném čase a vytvořit například akci každých 10 sekund. V téhle metodě bychom taky mohli využít metodu třídy Game1, kterou je *Exit*. Například při zmáčknutí tlačítka ESC na klávesnici, bychom zavolali metodu *Exit*, která by hru ukončila.

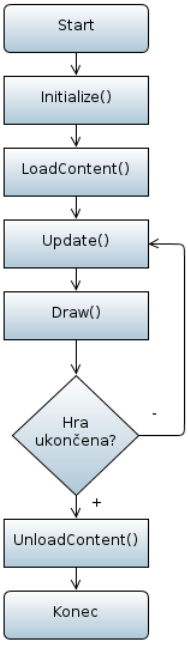
4.2.5 Metoda Draw

Metoda *Update* nám obstarává veškerou logiku naší hry, my však potřebujeme metodu, která zakreslí změny, které se provedli, a právě k tomu nám slouží metoda *Draw*. Vykreslování se provádí pomocí instance třídy spriteBatch. K tomu abychom byli schopní vykreslit nějaký obrázek, je nutné, abychom si ho nejdříve načetli danou texturu, dále musíme nastavit umístnění, které může být uvedeno pomocí třídy Vector2, nebo pomocí třídy Rectangle. Výhodou používání třídy Rectangle je to, že můžeme nastavit pozici obrázku včetně jeho šířky a délky. Při používání třídy Vector2 se obrázek vykreslí v takové velikosti, jakou má původní soubor. Posledním informací, kterou potřebujeme vědět, je barva, kterou nastavíme pomocí struktury Color.

4.3 Herní smyčka

Pro správné pochopení toho, jak naše hra bude fungovat, je dobré si ukázat, v jakém pořadí se budou volat jednotlivé metody. Jak je vidět na obrázku 1 (Čápka, ©2018), po spuštění hry se zavolá metoda *Initialize* a hned po ní *LoadContent*. Poté se již v intervalu 60x za vteřinu střídají metody *Update* a

Draw a to přesně v tomhle pořadí, dokud se hra neukončí. Před tím, než se hra ukončí, tak se provede odstranění herního obsahu z paměti.



Obrázek 1: Herní smyčka

4.4 Načtení obsahu

Načtení obsahu probíhá buď v metodě *LoadContent*, nebo může probíhat i v samotné třídě. Například když se dostaneme do menu, nemusíme zatím načítat všechny animace postaviček, které budou ve hře. Ty můžeme načíst až v samotném spuštění konkrétní hry.

Pro načítání obsahu využíváme vlastnost Content s generickou metodou *Load* s parametrem umístění cílového souboru. Díky tomu, že se jedná o generickou metodu, je nám jedno jestli načítáme texturu, font nebo muziku. V ukázce kódu 3 jsou příklady načtení různých typů souborů.

```
Content.Load<Texture2D>(@"Cesta_k_soubor");
Content.Load<Font>(@"Cesta_k_soubor");
Content.Load<Music>(@"Cesta_k_soubor");
```

Ukázka kódu 3: Příklad načtení obsahu hry

4.4.1 Vykreslení obrázků

Pro vykreslování obrázků využíváme metodu *Draw*. Předtím, než začneme vykreslovat samotné sprity, tak musíme vytvořit blok složený z metod *Begin* a *End*. Mezi ně pak můžeme vkládat kód, který bude vykreslovat naší herní grafiku.

V ukázce kódu 4 si můžete prohlídnout vzorový příklad vykreslení obrázku na vektorové pozici (0,0) s upravenou šířkou a výškou obrázků na 50 pixelů.

```
SpriteBatch.Begin();
SpriteBatch.Draw(texture, new Rectangle(0,0,50,50), Color.White);
SpriteBatch.End();
```

Ukázka kódu 4: Příklad vykreslení obrázku

4.4.2 Přehrání hudby

Pro hudbu využíváme dva různé datové typy SoundEffect a Song. Stejně jako u spritů musíme hudbu nejdřív nahrát s využitím generické metody *Load*. Pro samotné přehrávání hudby typu Song využijeme statickou třídu MediaPlayer s metodou *Play*. U zvukových efektů SoundEffect, pak musíme zavolat na samotnou instanci SoundEffect metodu *Play* bez parametru.

Dalším užitečným parametrem u hudby bude samotná hlasitost. Tu jednoduše změníme zavoláním vlastnosti. Pro hudbu využijeme vlastnost Volume statické třídy MediaPlayer. Pro zvukové efekty třídy SoundEffect využijeme vlastnost MasterVolume. Hlasitost se nastavuje od 0 do 1 pomocí datového typu float.

```
SoundEffect.MasterVolume = 0.1f;
MediaPlayer.Volume = 0.5f;
```

Ukázka kódu 5: Příklad nastavení hlasitosti

4.4.3 Vstup uživatele

Aby mohl hráč naší hru ovládat je důležité nějakým způsobem zaznamenat jeho aktivitu. V XNA máme širokou podporu ovladačů například vstup z klávesnice, myši a gamepadu.

Nejdříve se zaměříme na klávesnici. K tomu abychom byli schopní zaznamenat činnost hráče, potřebujeme znát třídu Keyboard a její statickou metodu *GetState*, která nám bude vracet aktuální stav stisknutých kláves. Další třídu, kterou budeme potřebovat je KeyboardState. Tam budeme ukládat aktuálně stisknuté klávesy. Na instanci třídy KeyboardState pak můžeme využít metody *IsKeyDown* a *IsKeyUp*.

Další možností, jak může hráč komunikovat se hrou, bude myš. S myší se pracuje velice podobně jako s klávesnicí s tím rozdílem, že budeme potřebovat třídy Mouse a MouseState.

5 Praktická část

5.1 Popis aplikace

Základní popis aplikace je důležitým krokem nejen pro nové hráče, ale také pro samotné vývojáře. Už před samotným návrhem aplikace je důležité vědět, co bude cílem hry. V našem případě se bude jednat o 2D střílečku na podobném principu, jako je česká populární hra Bulánci[1]. Hra bude konstruována pro čtyři hráče. Každý hráč si na začátku bude moc vybrat speciálního hrdinu s různými vlastnostmi.

Ovládání hry bude velice jednoduché. Každý hráč se bude moc pomocí šipek pohybovat nahoru, dolů, doprava a doleva. Střílet po ostatních, nebo v případě potřeby, začít přebíjet svojí zbraň. Cílem hry bude porazit všechny oponenty a zůstat poslední na herním poli.

5.2 Návrh aplikace

Při programování větších projektů je důležité si aplikaci rozdělit na menší části a ty jednotlivě programovat. Programování hry můžeme rozdělit na dvě velké části. Jednou částí bude naprogramování samotného menu hry, kde si hráč bude moc spustit hru, opustit hru a nastavit si různé parametry například zvuky, rozlišení a grafiku. V našem případě si ukážeme nastavení zvuků hry.

Druhou částí je pak programování samotného herního prostředí, kde bude probíhat naše vlastní hra. V našem projektu se bude jednat o dvě základní prostředí, kde v jednom z nich si hráči vyberou hrdinu, za kterého budou chtít hrát a následně s daným výběrem vstoupí do druhé části a tou už bude naše vlastní aréna, kde bude probíhat souboj mezi jednotlivými hráči.

1 http://www.bulanci.cz/

5.2.1 Návrh menu

Při návrhu je důležité ujasnit si, jaké prvky chceme v našem menu mít. Důležitým prvkem bude pozadí, které by mělo vytvořit dobrý dojem na hráče, když naší hru spustí poprvé. Dalším prvkem bude logo a umístění tlačítek. Naše menu by mělo obsahovat alespoň tyhle základní tlačítka.

New game bude tlačítko, které přesune hráče do prostředí, kde si budou moc zvolit, za jakého hrdinu chtějí hrát.

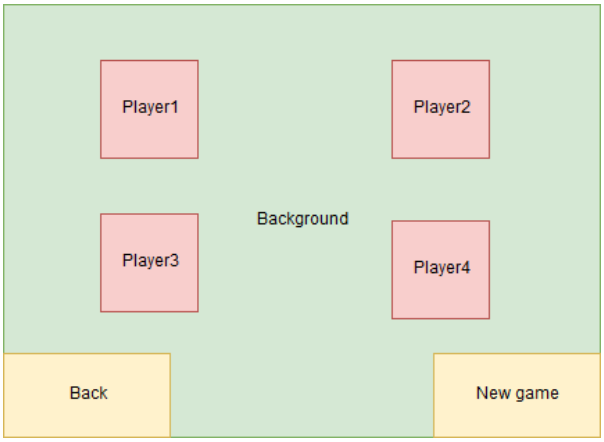
Options přesune hráče na další rozcestník, kde si bude moc vybrat jestli si chce nastavit zvuky nebo změnit grafické rozlišení.

Exit, po vybrání tlačítka hráč vypne hru.

5.2.2 Návrh herního prostředí

Výběr hrdinů bude první činnost, kterou hráči budou muset provést před samotným zahájením samotné herní akce. V našem návrhu je důležité promyslet, jaké všechny prvky budou součástí výběru postavy.

Prvním základním prvkem budou hráči, kteří si budou vybírat svoje hrdiny pro hru. Dalším důležitým prvkem bude pozadí společně s tlačítky *Back* sloužící pro návrat do hlavního menu a tlačítkem *New game*, kterým se dostaneme do hry s aktuálně zvolenými hrdiny. Při návrhu je dobré tvořit si základní nákresy. V návrhu 1 je můj vlastní návrh prostředí.



Návrh 1: Návrh prostředí pro výběr postavy

Druhá část naší hry je již zmíněná aréna, kde bude probíhat souboj mezi hráči. Aréna by měla obsahovat vhodné pozadí, které bude pro hráče příjemné, také je dobré promyslet jaký typ překážek tam umístit a v jakém počtu. Zde se jedná spíše o velice předběžné informace, které se doladí až v průběhu testování hry. Ale je důležité si určit všechny prvky, které by naše aréna v základní verzi měla obsahovat.

Players, bude pole hráčů, které bude evidovat veškeré informace o aktuálním počtu životů, počtu nábojů a rychlosti jednotlivých hráčů.

Boxs, bude pole překážek, které se vyskytují na hrací ploše.

Bullets, bude pole aktuálně letících střel.

5.2.3 Návrh tříd

Využijeme diagram návrhu tříd k tomu, abychom při programování již přesně věděli, jaké proměnné máme vytvořit u jednotlivých tříd. V příloze 9 si můžete prohlídnout diagram tříd, který budeme v naší hře potřebovat se základními atributy včetně jejich datových typů.

5.3 Tvorba grafiky

Většinu grafického obsahu hry jsem vytvářel v programu Gimp, kde jsem vytvořil jednotlivé pozadí pro menu i herní prostředí, vytvořil jsem si jednotlivá tlačítka pro menu a také jsem vytvořil bedny, které ve hře budou představovat překážku, která se bude při kolizi s projektily rozbíjet, až se rozpadne na třísky. Poslední věc, která je důležitá je sada spritů jednotlivých hrdinů, díky kterým budeme moc pomocí našich tříd vytvořit animace chodících postaviček. K tomuto účelu jsem využil online generátor (“Character Generator”, 2018), kde je možné připravit si vzhled postavičky a program již vygeneruje sadu obrázku, které při správném vykreslování vytvoří animaci chůze. Prohlídnout si je můžete v příloze 2.

5.4 Popis zdrojového kódu

V téhle kapitole si ukážeme a vysvětlíme zdrojový kód naší hry. Budeme se zabývat tím, jak jednotlivé třídy spolupracují, jaké mají atributy, vlastnosti a metody. Ukázky zdrojového kódu sem budu vkládat bez komentářů. Veškerý kód budu popisovat a vysvětlovat v každá kapitole, která bude mít stejný název, jako daná popisovaná třída.

5.4.1 Třída Game1

Automaticky vygenerovanou třídu Game1 jsme si již podrobně popsali, takže se můžeme pustit do vysvětlování. Do třídy jsem si přidal čtyři nové datové typy.

První z nich je třída Controls, která nám bude sloužit pro ukládání uživatelského vstupu z klávesnice a myši. V jedné z dalších kapitol si třídu podrobně rozebereme. Dále jsem tam přidal dvě třídy pro obsluhování hudby Music a MusicEffect. Menu je poslední moje přidaná třída, ve které probíhá obsluha celého menu a také obsluhuje přechod k samotné hře.

Než si ukážeme metodu *Initialize*, rozeberu několik pomocných metod, které jsem vytvořil a které nám usnadní život v dalších třídách. Jednou z nich je metoda *Booting*, která slouží k načítání základního herního nastavení ze souboru. Prohlídnout si ji můžete v ukázce kódu 6. Jedná se o parametrickou metodu, kde parametrem je umístnění bootovacího souboru „Start.txt“, který si můžete, prohlídnou v příloze 1. Metoda přečte soubor a nastaví šířku a výšku herního obrazu, zobrazení přes celou stránku a hlasitost hudby a zvukových efektů.

```
private void Booting(string path)
{
    string[] lines;
    char[] separator = new char[] { ';' };
    lines = File.ReadAllLines(path);

    string[] line = lines[1].Split(separator,
        StringSplitOptions.RemoveEmptyEntries);
    graphics.PreferredBackBufferWidth = int.Parse(line[0]);
    graphics.PreferredBackBufferHeight = int.Parse(line[1]);
    graphics.IsFullScreen = bool.Parse(line[2]);

    //nastavení music and effect music
    musicEffect = new MusicEffect(this, int.Parse(line[4]));
    music = new Music(this, int.Parse(line[3]));
}
```

Ukázka kódu 6: Nastavení grafiky a hudby při startu

Další metody, které jsem vytvořil, slouží pro bezpečné načítání textury, fontů, hudby a zvukových efektů. Pokud totiž zadáte cestu k souboru, který není načtený, celá hra spadne. Abychom zabránili dané situaci, vytvořil jsem bezpečné metody pro načítání obsahu. Princip je u všech stejný, proto popíšu pouze načítání textur. Jak můžete vidět v ukázce kódu 7, kód je vložený do bloku „try catch“, který při nenalezení požadované textury načte defaultně připravený obrázek s názvem Error.

```
public Texture2D SetTexture(string path)
{
    try
    {
        return Content.Load<Texture2D>(path);
    }
    catch
    {
        return Content.Load<Texture2D>(@"Sprits\Error");
    }

}
```

Ukázka kódu 7: Bezpečné načítání textur

Teď představím velice užitečné metody, které nám umožní zjistit velikost herního okna, pro snadné vykreslování pozadí, tlačítek a všech textur. *GetScreenSize* metoda s návratovým typem Vector2 nám vrací vektor s hodnotami, které nám poskytnou informaci o šířce a výšce herního okna. To se využije především při změně rozlišení, kde při návrhu vykreslování pozadí nebudeme muset řešit, jakou šířku máme nastavit pro naše pozadí, ale nastaví se automaticky podle daného rozlišení. Využijeme vlastnost instance graphics PreferredBackBufferWidth a PreferredBackBufferHeight. Kód je k vidění v ukázce kódu 8.

```
public Vector2 GetScreenSize()
{
    return new Vector2(graphics.PreferredBackBufferWidth,
                       graphics.PreferredBackBufferHeight);
}
```

Ukázka kódu 8: Zjištění daného rozlišení běžící hry

Konečně si můžeme představit naší metodu *Initialize*, kde vytvoříme instance našich proměnných. Bude se jednat o instanci controls, starající se o to, aby hráč mohl hru spustit a ovládat. Instanci menu, která přebírá instanci třídy Game1 a Controls, ve které bude probíhat veškerá logika naší hry. Nesmíme zapomenout taky na výše popsanou metodu *Booting*, kterou spustíme s požadovanou cestou. Metoda je k vidění v ukázce kódu 9.

```
protected override void Initialize()
{

    //základní nastavení hry
    this.Booting(@"Save\Start\Start.txt");
    graphics.ApplyChanges();

    //Initialization Tools
    controls = new Controls();
    menu = new Menu(this, controls);

    base.Initialize();
}
```

Ukázka kódu 9: Metoda Initialize

Vytvořené instance controls a menu také musíme umístit do metody *Update*, kde se bude vykonávat jejich logika, kterou si popíšeme v dalších kapitolách. A v poslední řadě nesmíme zapomenout na metodu *Draw*, která nám bude vykreslovat veškerý náš herní obsah. Zde vytvoříme pomocí instance spriteBatch blok z metod *Begin* a *End* a mezi ně vložíme metodu instance menu *Draw* s parametrem instance spriteBatch. To nám zajistí to, že v každé další třídě, kde budeme chtít vykreslovat grafiku pomocí instance spriteBatch, nebudeme muset náš kód vkládat mezi blok *Begin* a *End*.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    // TODO: Add your drawing code here
    spriteBatch.Begin();
        menu.Draw(spriteBatch);
    spriteBatch.End();
    base.Draw(gameTime);
}
```

Ukázka kódu 10: Metoda Draw ve třídě Game1

5.4.2 Třída Enums

Jedná se o pomocnou třídu, ve které budeme mít uložené veškeré výčtové typy, které se ve hře budou vyskytovat. Kdybychom vytvářeli výčtové typy v každé třídě, ke které se vztahují, bylo by to méně přehledné i časově náročnější. Také může nastat situace, kde výčtové typy využíváte ve více třídách a následné vyhledání v kódu pro případnou úpravu by mohlo být časově náročné. V ukázce kódu 11 je příklad výčtového typu MyButtonState, který využijeme k vykreslování spritu daného tlačítka v menu.

```
/// <summary>
///  Určuje stav tlačítka
/// </summary>
enum MyButtonState
{
    Base, Collision, Choosen
}
```

Ukázka kódu 11: Ukázka vložení Enums

5.4.3 Třída Controls

K tomu abychom mohli ve hře snadno pracovat s uživatelskými vstupy, vytvořili jsme si třídu Controls, která zaznamenává hráčovu aktivitu. Jelikož naše menu nebudeme ovládat jenom pomocí myší, ale také klávesnice, potřebujeme nejenom zaznamenat činnost klávesnice i myši. K tomu budeme využívat dvě třídy KeyboardState a MouseState. Ke každé třídě si vytvoříme dvě proměnné aktuálně stisknuté tlačítko a tlačítko, které jsme stisknuli těsně předtím.

Vytvoříme hlavní metodu *Update*, kde budeme zaznamenávat zmáčknutá tlačítka a ukládat je do proměnných, jak je vidět v ukázce kódu 12.

```
public void Update()
{
    mouseLast = mouse;
    mouse = Mouse.GetState();

    keyboardLast = keyboard;
    keyboard = Keyboard.GetState();
}
```

Ukázka kódu 12: Metoda Update třídy Controls

Jelikož metoda Update probíhá 60 krát do vteřiny, musíme ošetřit vstupy, které mají být provedeny pouze jednou po stisknutí klávesy. K tomu nám budou sloužit metody *OneClickMouse* a *OneClickKey*. Jednoduše budeme hlídat to, jestli klávesa, kterou hráč stisknul, těsně před tím nebyla stisknutá.

```
public bool OneClickKey(Keys keys)
{
    return keyboard.IsKeyDown(keys) && keyboardLast.IsKeyUp(keys);
}
```

Ukázka kódu 13: Stisknutí klávesy

Také je dobré si vytvořit metodu, která bude vyhodnocovat kolize s tlačítky. K tomu vytvoříme metodu s parametrem Rectangle, kde budeme předávat aktuální pozici tlačítka. Metoda pak sama vyhodnotí, jestli je na stejné pozici, jako je kurzor myši.

5.4.4 Třída Button

Třída, která bude sloužit pro zobrazování a práci s tlačítky v menu. Třída bude obsahovat proměnnou game, kde bude uložená instance třídy Game1, abychom mohli využívat její veřejné metody pro bezpečné načítání textur. Pak budeme potřebovat proměnné pro tři textury. Jedna bude pro tlačítko v běžném stavu, druhá pro tlačítko, na kterém bude najetá myš a poslední bude sloužit pro vykreslení tlačítka, které je již aktivně zmáčknuté hráčem. Také budeme muset uložit hodnotu pozice, pro to využijeme datový typ Rectangle. Do proměnné buttonState datového typu Enum, budeme ukládat stav tlačítka, které nám bude sloužit pro zobrazení hodnoty, v jakém stavu se naše tlačítko právě nachází.

Třída obsahuje tři metody. *Update*, *Draw* a *Click*. Metoda *Update* obstarává funkčnost tlačítka, metoda *Draw* dané tlačítko vykresluje na pozici,

která je určená v proměnné rectangle a metoda *Click* vrací hodnotu true tehdy, když je tlačítko hráčem aktivované. Celá metoda je v ukázce kódu 14.

```
public void Update(Controls controls)
{
    if (buttonState == MyButtonState.Collision
        && controls.mouse.LeftButton == ButtonState.Pressed)
    {
        if (buttonState != MyButtonState.Choosen)
        {
            game.GetMusicEffect(0);
        }
        buttonState = MyButtonState.Choosen;
    }
    else if (controls.GetCollision(rectangle)
        && controls.mouse.LeftButton == ButtonState.Released)
    {
        if (buttonState == MyButtonState.Collision)
        {
            game.GetMusicEffect(0);
        }
        buttonState = MyButtonState.Collision;

    }
    else
    {
        buttonState = MyButtonState.Base;
    }
}
```

Ukázka kódu 14: Metoda Update třídy Button

```
public void Draw(SpriteBatch spriteBatch)
{
    switch (buttonState)
    {
        case MyButtonState.Base:
            spriteBatch.Draw(baseTexture, rectangle, Color.White);
            break;
        case MyButtonState.Collision:
            spriteBatch.Draw(collisionTexture, rectangle, Color.White);
            break;
        case MyButtonState.Choosen:
            spriteBatch.Draw(choosenTexture, rectangle, Color.White);
            break;
        default:
            break;
    }
}
```

Ukázka kódu 15: Metoda Draw třídy Button

Jak můžete vidět v ukázce kódu 15, metoda *Update* nám mění hodnotu proměnné buttonState a třída nám podle ní vykresluje danou texturu obrázku.

5.4.5 Třída MyMouse

Jak název vypovídá, jedná se o třídu, která slouží k speciální textury ukazatele myši. Třída obsahuje proměnné pro texturu, pozici a velikost kurzoru. Také zde můžete najít proměnnou obsahující instanci třídy Game1.

```
class MyMouse
{
    private Game1 game;
    private Texture2D basic;
    private Vector2 position;
    private Vector2 sizeKurzor = new Vector2(20, 20);
    private string path = @"Sprits\Tools\Mouse\Basic";

    public MyMouse(Game1 game)
    {
        this.game = game;
        this.basic = game.SetTexture(path);
        position = new Vector2(0, 0);

    }

    public void Update(Controls controls)
    {
        position = controls.MousePosition();
    }

    public void Draw(SpriteBatch spriteBatch)
    {
        spriteBatch.Draw(basic, new Rectangle((int)position.X, (int)position.Y,
          (int)sizeKurzor.X, (int)sizeKurzor.Y), Color.White);
    }
```

Ukázka kódu 16: Třída MyMouse

V ukázce kódu 16 můžete vidět jednoduchý konstruktor, ve kterém se načte textura a vytvoří se instance pro pozici myši, která se poté v metodě *Update* bude přepisovat podle aktuální polohy kurzoru. Metoda *Draw* následně vykreslí texturu kurzoru na dané pozici v nastavené velikosti.

5.4.6 Třída Menu

Každá hra se hodně zjednodušeně řečeno skládá ze dvou velkých samostatných spolupracujících komponent. Řeč je o samotném herním prostředí, kde probíhá samotná logika hry a pak o prostředí samotného menu, které slouží pro nastavení grafiky, zvuků, spuštění nové hry a ukončení samotné hry. V naší hře si ukážeme nastavení základní obrazovky menu s tlačítky New game, Options a Exit.

Jelikož třída Menu obsahuje mnoho proměnných, které si můžete prohlídnout v příloze včetně jejich datových typů, nebudu zde, popisovat k čemu slouží každá z nich, ale rovnou začnu s popisem konstruktoru. Konstruktor obsahuje pouze dva parametry a to instanci třídy Game a instanci třídy Controls. Následně v konstruktoru probíhá načítání všech požadovaných textur metodou *LoadTexture* instance game. Taky se vytváří dvě nové instance pro myš a zvuk. Také zde zavoláme metodu *SetButton*, kde vytvoříme instance pro všechny tlačítka ve hře. Konstruktor si můžete prohlídnout v ukázce kódu 17.

```
public Menu(Game1 game, Controls controls)
{
    this.game = game;
    background = game.SetTexture(textureBackgroundPath);
    logo = game.SetTexture(textureLogoPath);
    logoRectangle = game.GetPositionRectangle(X_logo, Y_logo, widthLogo, heightLogo);
    this.SetButton();
    mouse = new MyMouse(game);
    audio = new Audio(game, this.GetSettingRectangle());
    this.controls = controls;
    this.PlayMenuMusic();
}
```

Ukázka kódu 17: Konstruktor třídy Menu

Metoda update je velice jednoduchá. Jelikož každé tlačítko je instancí třídy Button a samotné už obsahuje svojí vlastní logiku, tak ve třídě menu budeme volat jenom metody všech instancí tlačítek. První metoda bude *Update*, co ale musíme ohlídat, je to, aby když se budeme nacházet v sekci options, tak aby instance tlačítek New game, Options a Exit, nebyly funkční. To docílíme jednoduše, přes proměnnou menuState, která má datový typ enum a jednoduše pomocí fuknce switch rozdělíme do daných předdefinovaných částí. Názorná ukázka je v ukázce kódu 18. Další metodu, kterou využijeme, bude volání metody instance tlačítek *Click* s parametrem instance controls. Jelikož se jedná o metodu s návratovým typem bool, můžeme to zadat přímo do podmínky a v těle podmínky vyvoláme akci, kterou si představujeme u konkrétních instancí tlačítek. Například v podmínce instance tlačítka Exit, vypneme naší hru.

```
switch (menuState)
{
    case MenuState.Base:
        exitButton.Update(controls);
        newGameButton.Update(controls);
        optionButton.Update(controls);

        if (newGameButton.Click(controls))
        {
            selectionHero = new SelectionHero(game, controls, this);
            menuState = MenuState.SelectionHero;                    }


        if (optionButton.Click(controls))
        {
            menuState = MenuState.Option;
        }
        if (exitButton.Click(controls))
        {
            game.Exit();
        }
        break;
```

Ukázka kódu 18: Použití funkce switch ve třídě Menu

5.4.7 Třída Player

Třída Player je jedna z hlavních tříd, ve které je nastavena logika hráče a veškeré jeho atributy. Třída Player obsahuje informace o instanci třídy Game1, jméno hrdiny za kterého hráč bude hrát, proměnnou move datového typu Enum, kde bude uložený aktuální směr pohybu hrdiny, proměnné typu Keys, sloužící pro uživatelské ovládání. Třídu PlayerDraw, která bude sloužit pro vykreslování všech animací postavy, třídu Gun, která bude evidovat, jakou zbraň daný hrdina používá. Poslední informací, kterou budeme evidovat u jednotlivých hráčů, bude počet aktuálních životů, maximálních životů a rychlost.

Třída obsahuje dva konstruktory. První konstruktor slouží pro vytváření možných hrdinů ve hře, za které bude moc daný hráč bojovat. To znamená, že u takových hrdinů není nutné definovat všechny atributy dané třídy například ovládání, textury a podobně. Jediné, co potřebujeme vědět, je jméno postavy, název používané zbraně, počet maximálních životů a rychlost. V ukázce kódu19 můžete vidět jednoduchý konstruktor, kde předáváme veškeré potřebné informace.

```
public Player(string name, Guns nameGun, int life, int speed)
{
    this.name = name;
    this.nameGun = nameGun;
    this.life = life;
    this.speed = speed;
    this.GetGun(game, this, nameGun);
}
```

Ukázka kódu 19: 1. konstruktor v třídě Player

V ukázce kódu 20 je druhý konstruktor, který už bude sloužit pro vytvoření instance, kterou bude moc hráč využít pro ovládání svého hrdiny. Je zde nutné předat veškeré hodnoty, které jsou vytvořené ve třídě Player.

```
public Player(Game1 game, Rectangle rectangle, string name, int life, int speed,
    Keys up, Keys down, Keys left, Keys right, Keys fire, Keys reloaded, Guns gun)
{
    this.game = game;
    this.rectangle = rectangle;
    this.up = up;
    this.down = down;
    this.left = left;
    this.right = right;
    this.fire = fire;
    this.reloaded = reloaded;
    this.name = name;
    this.life = life;
    this.maxLife = life;
    playerDraw = new PlayerDraw(game, this);
    this.nameGun = gun;
    this.GetGun(game, this, gun);
    this.speed = speed;
}
```

Ukázka kódu 20: 2. konstruktor třídy Player

Dále zde máme několik metod, které si společně popíšeme. Metody *ReadyForBattle* a *Winner*, jsou jednoduché metody, které slouží k spuštění metody instance *playerDraw* pro vytvoření speciálního vykreslení před hrou při výběru postav a následně vykreslení vítězného hrdiny. Dále zde máme metody *GetMove*a *GetGun*, které slouží pouze pro zpřístupnění daných proměnných move a gun, které budeme využívat v hlavní třídě pro to, abychom zjistili směr pohybu hráče a mohli generovat projektily při střelbě. K tomu, abychom byli schopní změnit směr hráčova pohybu, vytvoříme metodu *SetMove* s parametrem Move, kde pouze přepíšeme proměnnou move.

Metoda *Hurt* s parametrem poškození slouží k odečítání zdraví daného hrdiny. Metoda zároveň kontroluje, jestli hráčovo zdraví není již menší nebo rovno 0. Když je daná podmínka splněna, nastaví se proměnná typu bool isAlive na hodnotu false a přehraje se zvukový efekt, který by měl zobrazovat umírání postavy. V ukázce kódu 21 je vidět samotná metoda *Hurt.*

```
public void Hurt(int damage)
{
    life -= damage;
    if (life <= 0)
    {
        isAlive = false;
        game.GetMusicEffect(1);
    }
}
```

Ukázka kódu 21: Metoda Hurt třídy Player

Důležitou metodou cele třídy je *Update*, kde probíhá veškerá logika hráčovi chůze. Zde se detekují srážky z herními objekty. První co metoda *Update* vykonává, je běh metody *Update* instance playerDraw, sloužící pro animace

hrdiny. Dalším prvkem metody je samotné ovládání postavy, kde pomocí podmínek zjišťujeme, jakou klávesu hráč aktuálně drží. Jednou z herních mechanik je chůze. Jelikož každý směr pohybu má své specifické vlastnosti, popíšu všechny čtyři možné směry.

První směr je pohyb nahoru. První co je potřeba nastavit je změna proměnné move. Musíme změnit umístění daného hrdiny a to tak, že v případě pohybu nahoru musíme změnit Y pozici naší proměnné rectangle pomocí vlastnosti „rectangle.Y“ tak, že odečteme hodnotu proměnné speed. Zároveň podmínkou ověříme, jestli naše postavička už není mimo mapu. Jelikož víme, že se budeme pohybovat v kladných hodnotách, stačí se jednoduše zeptat, jestli hodnota Y již není menší než 0 a v případě pravdivosti, nastavit její hodnotu na 0. Nesmíme zapomenout, že v naší hře se můžou vyskytovat různé překážky. Takže musíme ověřit, jestli jsme do nějaké zrovna nenarazili. To zjistíme tak, že si necháme vypsat list všech překážek na mapě a zkontrolujeme, jestli se hráč nenachází na dané pozici, případně vrátit jeho pozici na předchozí pozici.

```
//move Up
if (controls.PressKey(Up))
{
    move = Move.Up;
    rectangle.Y -= speed;
    if (rectangle.Y < 0)
    {
        rectangle.Y = 0;
    }
    foreach (var box in boxs)
    {
        if (box.Rect.Intersects(rectangle))
        {
            rectangle.Y = box.Rect.Y + box.Rect.Height;
        }
    }
}
```

Ukázka kódu 22: Část kódu metody Update ve třídě Player

Pohyb dolů je potřeba také ošetřit a to trošku jiným způsobem. V případě, že hráč drží klávesu pro pohyb dolů, je nutné zvýšit vektorovou pozici Y naší proměnné rectangle pomocí vlastnosti „rectangle.Y“. Pomocí podmínky, kde budeme kontrovat, jestli pozice Y hráče společně s jeho výškou není větší než herní obrazovka. Také zde musí proběhnout kontrola kolize se všemi překážkami na mapě.

Pohyb hráče doleva zajistíme tím, že budeme odečítat naší pozici na ose X a při pohybu doprava přičítat. Poslední operací, kterou budeme v metodě

vykonávat, je kontrola stisknutí klávesy pro přebíjení zbraně. Při stisknutí přebíjecí klávesy zavoláme metodu *GetReloaded*, která je instancí třídy Gun.

```
public void Update(GameTime gameTime, Controls controls)
{
    playerDraw.Update(gameTime, controls);
}
```

Ukázka kódu 23: Metoda Draw ve třídě Player

Jak můžete vidět v ukázce kódu 23, poslední metoda je velice jednoduchá. Pouze volá metodu dané instance. Přesný popis metody si popíšeme v další kapitole.

5.4.8 Třída PlayerDraw

```
private Game1 game;
private Texture2D texture;
private Rectangle rectangle;

private int currentPicture;
private int countPicture = 9;
private Move move;
private float delay = 100;
private float elapsed;

private Player player;
private Bar hp;
private Bar bulets;
private bool IsBattle = false;
```

Ukázka kódu 24: Proměnné třídy PlayerDraw

V ukázce kódu 24 můžete vidět všechny proměnné, které budeme ve třídě využívat. První z nich je naše třída Game1, kterou využijeme k načítání textury. Další je samotná textura a proměnná rectangle, která bude sloužit pro zobrazení konkrétního spritu s naší připravenou texturou, kterou si můžete prohlídnout v příloze číslo 2. Proměnná currentPicture bude sloužit k uchování hodnoty, jaký aktuální sprite naší postavičky se přehrává a countPicture bude určovat kolik je celkem všech spritů, které slouží k vytvoření animace chůze. Move je výčtový typ, který určuje směr naší postavičky. Proměnná delay a elapsed slouží pro určování časové prodlevy mezi přepínáním jednotlivých obrázků. Proměnná player slouží k tomu, abychom byli schopní zjišťovat aktuální pozici hráče ve hře. Hp a bullets nám budou sloužit vykreslení množství aktuálního zdraví a počtu nábojů nad daným hráčem.

Pomocí konstruktoru třídy si načteme danou texturu hrdiny, která budou sloužit pro tvorbu animací chůze. Deklarujeme si proměnné game, player a určíme si počáteční směr chůze. Vytvoříme instance hp a bullets. Konstruktor si můžete prohlídnout v ukázce kódu 25.

```
public PlayerDraw(Game1 game, Player player)
{
    this.game = game;
    this.player = player;
    this.texture = game.SetTexture(@"Sprits\Game\Heroes\" + player.Name);
    move = player.GetMove();
    elapsed = delay;
    hp = new Bar(game, player, game.SetTexture(@"Sprits\Tools\Bar\Hp"), new Vector2(0, 15));
    bulets = new Bar(game, player, game.SetTexture(@"Sprits\Tools\Bar\Bullets"), new Vector2(0, 5));
}
```

Ukázka kódu 25: Konstruktor třídy PlayerDraw

Třída také obsahuje jednu neveřejnou metodu s návratovým typem bool *IsNextFrame* s parametrem třídy GameTime, která slouží pro zaznamenávání určitého časového horizontu. Metoda jednoduše přičítá milisekundy do proměnné elapsed, která se pak porovnává s proměnnou delay. V případě, že elapsed je větší nebo rovno delay, vrátí se hodnota true a nastaví se hodnota elapsed zpátky na nulu.

V metodě *Update*, pak bude probíhat samotná logika animace. V případě, že hráč drží jakékoliv tlačítko pro chůzi, spustí se metoda *IsNextFrame*. Když vrátí hodnotu true, zvýší se currentPicture o jedna a zkontroluje se, jestli již není větší než proměnná countPicture mínus jedna a jestli je, tak se hodnota currentPicture nastaví na nulu. Na konci metody zavoláme metodu *SetRectangle*, která nastaví hodnotu pro proměnnou rectangle. Metodu *Update* si můžete prohlídnout v ukázce kódu 26.

```
public void Update(GameTime gameTime, Controls controls)
{
    if (controls.PressKey(player.Right)
        || controls.PressKey(player.Left)
        || controls.PressKey(player.Up)
        || controls.PressKey(player.Down))
    {
        if (IsNextFrame(gameTime))
        {
            currentPicture++;
            if (currentPicture > countPicture - 1)
            {
                currentPicture = 0;
            }
        }
    }
    else
    {
        currentPicture = 0;
        elapsed = delay;
    }
    this.SetRectangle();
}
```

Ukázka kódu 26: Metoda Update ve třídě PlayerDraw

Metoda *SetRectangle* obsahuje základní proměnné typu integer staticHeight a width. Ty určují, o kolik pixelů se má posunout rám pro zobrazení spritu v textuře spritů. Proměnné right, left, up a down pak určují kolikátý obrázek z vrchu je určený pro daný směr chůze. V případě, že se hráč pohybuje směrem dolů, vynásobí se proměnná staticHeight a right, výsledek uložíme do proměnné height, kde budeme mít současnou pozici Y pro daný sprite. Na konci metody deklarujeme nové hodnoty pro rectangle, kde pozici X získáme vynásobením width a currentPicture, pozici Y máme již uloženou v hodnotě height, výšku a šířku již v width a staticHeight, takže je také využijeme.

Poslední metodou třídy je pak metoda *Draw*, kde vykreslíme část textury, kterou jsme si před chvílí spočítali na konkrétní pozici, kde se aktuálně nachází hráč. Také pokud se hráč nachází v aréně, což nám eviduje proměnná IsBattle, tak vykreslíme také instanci hp a bullets, jak můžete vidět v ukázce kódu 27.

```
public void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(texture, player.Rect, rectangle, Color.White);
    if (IsBattle)
    {
        hp.Draw(spriteBatch);
        bulets.Draw(spriteBatch);
    }

}
```

Ukázka kódu 27: Metoda Draw ve třídě PlayerDraw

5.4.9 Třída Gun

Ve třídě Gun budeme evidovat funkčnost střelby, jako je počet nábojů, rychlost nábojů, poškození, přebíjecí doba a kadence. Všechny tyhle údaje budeme muset uložit do našich proměnných countBullet, maxCountBullet, speed, damage, reloadedDelay, shotDelay, reloadedTime a shotTime. Navíc také budeme muset evidovat typ náboje, instanci hráče, abychom věděli, z jaké pozice máme vygenerovat naší střelu, její výšku a šířku v proměnných bullet, bulletWidth, bulletHeight a player.

Ve třídě využíváme pouze jeden konstruktor, ve kterém budeme předávat instanci naší hry Game1, instanci hráče, množství nábojů, rychlost nábojů, poškození, přebíjecí dobu, kadenci a typ náboje. Jelikož třída slouží pouze k uchování údajů a vyhodnocování logiky střely, není potřebná metoda *Draw*.

Metoda *Update* obsluhuje základní logiku zbraně a určuje, jestli hráč má nabitou zbraň, jestli může vystřelit další střelu a jestli zrovna nepřebíjí. Jako první se zde kontroluje kadence zbraně a to pomocí proměnné isShotReady. V podmínce se kontroluje shotTime a shotDelay. Pokaždé když proměnná typu bool isShotReady vrací hodnotu false přičte se k proměnné shotTime jednička, když je v tu chvíli shotTime větší nebo rovno shotDelay nastaví se hodnota isShotReady na true a shotTime na nulu. Další věc co se musí kontrolovat je přebíjení. V případě, že hráč nepřebíjí, tak se v metodě nic neprovádí, ale v opačném případě se zvyšuje proměnná reloadTime o jedničku a také se kontroluje s proměnnou reloadedDelay stejně jako tomu bylo v předchozím případě. Když je zbraň již nabitá, nastaví se hodnota isRealoaded na false, nastaví se proměnná countBullet na hodnotu MaxCountBullet a vynuluje se proměnná reloadedTime. Metodu *Update* si můžete prohlídnout v ukázce kódu 28.

```
public void Update(GameTime gameTime, Controls controls)
{
    rectangle = SetRectangle();
    if (!isShotReady)
    {
        shotTime++;
        if (shotTime >= shotDelay)
        {
            isShotReady = true;
            shotTime = 0;
        }
    }
    if (isReloaded)
    {
        reloadedTime++;
        if (reloadedTime >= reloadedDelay)
        {
            isReloaded = false;
            countBullet = MaxCountBullet;
            reloadedTime = 0;
        }
    }
}
```

Ukázka kódu 28: Metoda Update ve třídě Gun

Při generování našich střel musíme vypočítat pozici, která bude záviset na konkrétním směru hrdiny. K tomu nám slouží metoda *SetBulletsRectangle*, která vrací hodnotu rectangle. V metodě si vytvoříme dočasnou proměnou tmp typu Rectangle. V případě, že směr našeho hrdiny je nahoru, musíme nastavit hodnotu naší proměné tak, aby se zobrazovala uprostřed naší textury v horní straně. To znamená, že musíme převzít pozici naší instance player, kde akorát změníme hodnotu X na polovinu, vynásobením 0,5. V případě dalších směrů postupujeme podobně a vždy chceme změnit hodnotu té strany, na kterou je náš hrdina právě otočený. Metodu *SetRectangle* si můžete prohlídnout v ukázce kódu 29.

```
private Rectangle SetBulletsRectangle()
{
    Rectangle tmp = new Rectangle();
    if (player.GetMove() == Move.Down)
    {
        tmp = new Rectangle(player.Rect.X +
            (int)(0.5 * player.Rect.Width),
                            player.Rect.Y + player.Rect.Height,
                            bulletWidth, bulletHeight);
    }
    if (player.GetMove() == Move.Up)
    {
        tmp = new Rectangle(player.Rect.X +
            (int)(0.5 * player.Rect.Width),
                            player.Rect.Y - bulletHeight,
                            bulletWidth, bulletHeight);
    }
    if (player.GetMove() == Move.Left)
    {
        tmp = new Rectangle(player.Rect.X - bulletWidth,
                            player.Rect.Y +
                            (int)(0.5 * player.Rect.Height),
                            bulletWidth, bulletHeight);
    }
    if (player.GetMove() == Move.Right)
    {
        tmp = new Rectangle(player.Rect.X + player.Rect.Width,
                            player.Rect.Y +
                            (int)(0.5 * player.Rect.Height),
                            bulletWidth, bulletHeight);
    }
    return tmp;
}
```

Ukázka kódu 29: Metoda SetBulletsRectangle ve třídě Gun

Když už máme vygenerovanou pozici, tak si vytvoříme metodu, která bude generovat naší střelu. K tomu nám slouží naše veřejná metoda *GetBullet*, kde vytvoříme instanci třídy Bullet. Třídu Bullet si detailně popíšeme v další kapitole, teď nám postačí vědět, že třída existuje. Tak ještě předtím, než se v metodě budeme věnovat vytvořením samotné střely, zmenšíme proměnnou countBullet o jedna. Podmínkou si zkontrolujeme, jestli náhodou nám již nedošli náboje a v případě, že by došly, nastavíme proměnnou isRealoaded na true. Dále nastavíme proměnnou isShotReady na false. Následně si zjistíme jaký typ střeli je uložený v proměnné bullet a přehrajeme zvukový efekt daného výstřelu. Na konci vrátíme hodnotu nové instance střely, kde předáme potřebné parametry, které jsou po nás vyžadovány. Zde využijeme naší vytvořenou metodu *SetBulletsRectangle* a vložíme jí na pozici, kde je po nás požadována proměnná datového typu Rectangle.

5.4.10 Třída Bullet

Třída Bullet se nám stará o logiku a vykreslení našich střel. Třída eviduje směr, rychlost a poškození střely, tyhle údaje uložíme do proměnných move, speed a damage. Do proměnné delete datového typu bool uložíme informaci o tom, jestli naše střela má nebo nemá být vymazána z herní plochy. Využijeme toho tehdy, když naše střela zasáhne překážku nebo hráče, tak abychom jí mohli vymazat. Zbytek proměnných game, rectangle, player, bullet, texture a scale nám bude sloužit pro správné vykreslení naší střely.

```
private Game1 game;
private Rectangle rectangle;
private Player player;
private Bullets bullet;
private Texture2D texture;
private float scale;

private Move move;
private int speed;
private int damage;
private bool delete;
```

Ukázka kódu 30: Proměnné třídy Bullet

Pomocí konstruktoru deklarujeme všechny proměnný a zavoláme vlastní metodu *GetBulletTexture*. Metoda vezme proměnnou bullet, která je výčtovým typem Bullets a určí, jaký typ střely má nastat. V našem případě se jedná o střely typu Arrow, FireBall, Spear a Dagger. Pomocí speciální funkce switch, zjistíme o

jaký typ náboje se jedná a načteme daný typ textury a nastavíme proměnnou scale, která určuje velikost projektilu.

V metodě Update, pak probíhá samotný pohyb střely daným směrem, který máme deklarovaný v proměnné move. V případě, že je nastavená hodnota move na Down, což značí pohyb dolů, musíme zvyšovat Y hodnotu proměnné rectangle o velikost proměnné speed. Také po změně pozice zkontrolujeme, jestli se střela nenachází mimo hrací plochu a v případě, že by se nacházela, nastavíme hodnotu proměnné delete na true. Kontrola probíhá stejným způsobem jako ve třídě Player. V případě pohybu nahoru odečítáme Y pozici. Směr doprava zvyšujeme pozici X a doleva zmenšujeme pozici X naší proměnné rectangle.

Abychom jsme nemuseli u každé střeli načítat pro každý směr našeho náboje nový sprite, využijeme speciální úpravu metody *Draw*, která nám umožní rotaci daného spritu. Abychom jsme mohli docílit rotace, musíme metodě *Draw* předat konkrétní parametry. Danou texturu, kterou chceme vykreslit, poté musíme předat umístění daného spritu na hrací ploše pomocí vektorové pozice, následně zadáme null, jelikož nás daný parametr nezajímá, nastavíme barvu na white, což určuje původní barvu textury. Pak nastavíme hodnotu rotace. Další hodnota je velice zajímavá a je to vektorová hodnota, která bude určovat bod rotace objektu, poté zadáme hodnotu velikosti, kterou budeme mít uloženou v proměnné scale. Další dva parametry pro nás nejsou zajímavé, tak proto je nastavíme na defaultní hodnoty. Metodu *Draw* si můžete prohlídnout v ukázce kódu 31.

```
public void Draw(SpriteBatch spriteBatch)
{
    if (move == Move.Right)
        spriteBatch.Draw(texture, new Vector2(rectangle.X, rectangle.Y),
            null, Color.White, 0, new Vector2(texture.Width / 2, texture.Height / 2)
            , scale, SpriteEffects.None, 0);
    if (move == Move.Left)
        spriteBatch.Draw(texture, new Vector2(rectangle.X, rectangle.Y),
            null, Color.White, 3.14f, new Vector2(texture.Width / 2, texture.Height / 2)
            , scale, SpriteEffects.None, 0);
    if (move == Move.Down)
        spriteBatch.Draw(texture, new Vector2(rectangle.X, rectangle.Y),
            null, Color.White, 1.57f, new Vector2(texture.Width / 2, texture.Height / 2)
            , scale, SpriteEffects.None, 0);
    if (move == Move.Up)
        spriteBatch.Draw(texture, new Vector2(rectangle.X, rectangle.Y),
            null, Color.White, 4.71f, new Vector2(texture.Width / 2, texture.Height / 2)
            , scale, SpriteEffects.None, 0);
}
```

Ukázka kódu 31: Metoda Draw třídy Bullet

5.4.11 Třída Box

Třída Box nám bude sloužit pro vykreslování překážek ve hře. Překážka bude reprezentována bednou, která se v průběhu hry bude v závislosti na poškození postupně rozbíjet, až z ní zůstanou jen třísky, přes které sice hráč nebude moc chodit, ale bude moc přes ně střílet.

Budeme zde muset mít proměnnou rectangle, kde budeme mít uloženou pozici a velikost bedny. Dále zde máme proměnné texture, texture2 a texture3. Jelikož zde máme víc textur, které určitě budou ve stejné složce a budou se lišit jen v číslovce na konci, tak si připravíme proměnnou texturePath kam si uložíme cestu a základní společný název textury. Abychom mohli ve hře postupně měnit vzhled bedny v závislosti na jejích životech, budeme potřebovat také proměnnou hp datového typu integer. Poslední proměnná bude datového typu bool s názvem IsBlock a hned si jí deklarujeme na hodnotu true. To využijeme v metodě, kde budeme řešit kolizi s projektily.

```
private Game1 game;
private Rectangle rectangle;
private Texture2D texture;
private Texture2D texture1;
private Texture2D texture2;
private Texture2D texture3;
private string texturePath = @"Sprits\Game\Box\Box";
private float maxHp = 500;
private float actualHp;
public bool IsBlock = true;
```

Ukázka kódu 32: Proměnné třídy Box

V konstruktoru deklarujeme proměnné game a rectangle a načteme všechny textury dané bedny. Také nastavíme hodnotu proměnné actualHp na hodnotu maxHp. Ačkoliv zde máme čtyři různé textury, tak metoda *Draw* je velice jednoduchá. Ve hře totiž vykreslujeme vždy jenom aktuální vzhled bedny. Buď je spravená, nebo nějakým způsobem rozbitá. Takže vždy v metodě *Draw* budeme předávat proměnnou textuře, kterou si můžete prohlídnout v ukázce kódu 33.

```
public void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(texture, rectangle, Color.White);
}
```

Ukázka kódu 33: Metoda Draw třídy Box

Poslední metodou dané třídy je metoda *Hit* s parametrem poškození, kde pokaždé kolizi s danou střelou se zavolá daná metoda a předá se velikost poškození střely. Od aktuálního stavu životů bedny odečteme poškození a pomocí podmínek zkontrolujeme, jestli není potřeba změnit danou texturu. V případě, že je to zapotřebí, nastavíme novou hodnotu textury. Metodu si můžete prohlídnout v ukázce kódu 34.

```
public void Hit(int damage)
{
    maxHp -= damage;
    if (maxHp <= 300)
    {
        texture = texture1;
    }
    if (maxHp <= 100)
    {
        texture = texture2;
    }
    if (maxHp <= 0)
    {
        texture = texture3;
        IsBlock = false;
    }
}
```

Ukázka kódu 34: Metoda Hit třídy Box

5.4.12 Třída Arena

Arena je hlavní třída, ve které probíhá veškerá logika naší hry. Všechny proměnné mají nastavený identifikátor přístupu na private a to z důvodu, že veškerá logika třída probíhá uvnitř. Třída obsahuje dvě veřejné metody *Update* a *Draw*. Jako první co ve třídě máme, jsou naše známe proměnné game a controls, dále pak ještě proměnnou menu. Je to z důvodu, že po skončení souboje se budeme chtít vrátit zpátky do menu. Pro vykreslování potřebného pozadí třída obsahuje proměnnou bg. Abychom mohli snadno řešit kolize objektů, hráčů a projektilů, vytvoříme tři generické listy pro třídy Box, Player a Bullet a rovnou vytvoříme jejich instance. Také si nastavíme velikost hrdinů do proměnných heightPlayer a widthPlayer, kterým rovnou uložíme naší hodnotu. Na konci hry

budeme chtít také vykreslit vítězného hráče a tak si připravíme hodnoty s jeho procentuální velikostí do proměnných heightPlayerWin a widthPlayerWin, také s konkrétní hodnotou. Abychom mohli na konci vykreslit vítěze, tak si také vytvoříme proměnnou winner datového typu bool a nastavíme ji na hodnotu false. Stejným způsobem si vytvoříme proměnnou firstLauch, abychom mohli vykonat jednorázové operace při startu. Její hodnotu nastavíme na true. Do proměnné winnerFont pak uložíme font textu. Také si zde uděláme proměnou třídy MapGenerator, který má jednu veřejnou metodu, která je návratového typu List<Box> a vrací seznam překážek pro mapu.

```
private Game1 game;
private Controls controls;
private Menu menu;
private Texture2D bg;
private Player player;
private MapGenerator mapGenerator;
private List<Box> boxs = new List<Box>();
private List<Player> players = new List<Player>();
private List<Bullet> bullets = new List<Bullet>();
private bool firstLauch = true;
private int heightPlayer = 50;
private int widthPlayer = 50;
private int heightPlayerWin = 30;
private int widthPlayerWin = 30;
private bool winner = false;
private SpriteFont winnerFont;
```

Ukázka kódu 35: Proměnné třídy Arena

V konstruktoru deklarujeme proměnné game, controls, menu, winnerFont. Vytvoříme instanci třídy MapGenerator a zavoláme metodu na naší proměnnou generického listu s třídou Box. Dále načteme texturu pozadí a naplníme náš list s hráčema. Pomocí funkce foreach zavoláme metodu *ReadyForBattle*, každé naší instance Player v listu. Jako poslední pomocí instance třídy Game1 zapneme hudbu arény.

Kromě konstruktoru třída obsahuje pouze dvě veřejné metody *Update* a *Draw*. První si popíšeme metodu *Update*. Jako první spustíme zvukový efekt zahajující bitvu tak, že do podmínky vložíme naší proměnnou firstLauch, kde pomocí instance třídy Game1 spustíme daný zvukový efekt a nastavíme hodnotu proměnné firstLauch na false. Tím docílíme, že se podmínka vykoná pouze při prvním volání metody *Update*. Jak již víme, tak metoda *Update* se přehrává 60 krát za vteřinu, takže by bylo velice nežádoucí, aby se zvukový efekt spouštěl při každém volání dané metody. Dále je nutné zpřístupnit ovládání každého hráče tak,

že projdeme celý náš generický list s našimi hráči a spustíme u každé instance metodu *Update* s parametrem času, ovládání a generickým listem našich překážek. V tuhle chvíli máme funkční chůzi hráčů a jejich kolize s překážkami. Nyní je za potřebí, vytvořit logiku střel. V podmínce si zjistíme, jestli daný hráč nedrží tlačítko, které jsme mu definovali pro střelbu. V případě pravdy musíme zkontrolovat v další podmínce, jestli daný hráč, který se snaží střílet, má proměnnou IsReloaded na hodnotě false, tím pádem víme, že nepřebíjí a může střílet a zároveň jestli hráč má nastavenou proměnnou IsShorReady na true. V případě, že podmínka je vyhodnocená jako true, vytvoříme střelu a přidáme ji do našeho generického listu pro třídu Bullet. Když již máme vytvořené projektily, musíme vytvořit také jejich kolize s překážkami a hráči. Pomocí funkce foreach budeme procházet celí list obsahující naše střeli a spustíme u každé střeli jejich metodu *Update*, abychom zařídili jejich pohyb. Dále zkontrolujeme, jestli v danou chvíli není pozice střely shodná s pozicí nějakého hráče. V případě shody, vykonáme u daného hráče metodu *Hurt* s parametrem poškození dané střely a nastavíme proměnnou delete u dané střely na hodnotu true. Dále stejným způsobem zkontrolujeme kolizi projektilu s překážkami na mapě a v případě shody též nastavíme proměnnou delete na true a vykonáme metodu třídy Box *Hit* s parametrem poškození střely. Dále zkontrolujeme pomocí funkce for, jestli se v listu střel nenachází střela s hodnotou parametru delete na hodnotě true. V případě nálezu dojde k vymazání dané střely z listu. Stejným způsobem zkontrolujeme, jestli se na mapě nenachází nějaký hrdina, který vrací hodnotu true u proměnné isAlive. Také kontrolujeme, jestli list s hráči neobsahuje už jenom jednoho hráče. V takovém případě se ukončí hra a vykreslí se vítězný hráč s textem WINNER.

Poslední metodou dané třídy je metoda *Draw*, ve které vykreslíme pozadí bg. Dále v podmínce, kde v případě, že hodnota proměnné winner je false, tak vykreslujeme všechny objekty generických listů pomocí funkce foreach a to tak, že na instance objektů voláme metodu *Draw*. V případě, že se proměnná winner nastaví na hodnotu true. Dojde k vykreslení pouze posledního hráče s textem WINNER. Metodu *Draw* si můžete prohlídnout v ukázce kódu 36.

```
public void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(bg, game.GetScreenRectangle(),
Color.White);
    if (!winner)
    {
        foreach (var box in boxs)
        {
            box.Draw(spriteBatch);
        }
        foreach (var bullet in bullets)
        {
            bullet.Draw(spriteBatch);
        }
        foreach (var player in players)
        {
            player.Draw(spriteBatch);
        }

    }
    else
    {
        players[0].Draw(spriteBatch);
        spriteBatch.DrawString(winnerFont, "WINNER", game.GetPositionVector(35, 20), Color.White);
    }
```

Ukázka kódu 36: Metoda Metoda Draw třídy Arena

5.5 Testování aplikace

Po samotném naprogramování hry je nutné, jako u každé jiné aplikace otestovat, jestli se doopravdy chová, tak jak jsme při navrhování a programování předpokládali. V rámci práce jsem tedy pomocí mých kamarádů Tomáše Vogeltanze, Martina Kunce a Petry Chvostové hru otestoval.

Při prvním spuštění výsledky nebyly špatné, ale nastavení některých atributů postav bylo velice nevyrovnané, takže někteří hrdinové neměli šanci vyhrát, naopak jeden hrdina byl prakticky neporazitelný. Takže jsem po každé hře upravoval počet životů, rychlost chůze a poškození jednotlivých hrdinů tak, aby každá postava ve hře měla svojí speciální výhodu. Snažil jsem se nastavit atributy tak, aby vždy záleželo na schopnostech daného hráče více, než na tom, jakého si zvolil hrdinu.

Po několika hodinách hraní a zábavy, se mi nám konečně povedlo vybalancovat hru tak, aby každý hrdina byl zábavný a přitom trošku rozdílný, tím bych chtěl poděkovat kamarádům, za pomoc při testováním, zpětnou vazbu a návrhy na úpravy, které mi pomohlo dokončit mojí hru do zdárného konce.

Závěr

Cílem této práce bylo vyvinout aplikaci pomocí XNA frameworku a popsat její vývoj. Na začátku kapitol se čtenář seznámil se základy programování. Dozvěděl se základy objektově orientovaného programování, které bylo využito při vývoji aplikace. Dále byl seznámen s obecným přístupem pro vývoj her, včetně odpovědí na základní otázky každého začínajícího vývojáře hry. Před samotným vývojem aplikace bylo podrobně popsané samotné prostředí XNA. V praktické části jsem pak představil samotnou hru se základním popisem návrhu. Hlavní částí práce byl pak popis samotného zdrojového kódu, kde jsem podrobně popsal hlavní třídy a jejich metody, které jsem v aplikaci využil.

Výsledkem práce je tedy kompletní a funkční hra. Jedná se však o první verzi, takže zde chybí spousty prvků, které byly na začátku předpokládány, například změna rozlišení v nastavení hry. Však návrh tříd je vytvářen tak, že při změně rozlišení se veškeré grafické prvky přizpůsobí danému rozlišení. Grafiku samotné hry jsem vytvářel sám s pomocí různých předpřipravených grafických nástrojů, které se můžou využívat pro nekomerční produkty. Hra byla při prvním dokončením testována a optimalizována.

Na straně hry vidím spousty dalších možností rozšíření, které by se do budoucna mohli implementovat a vylepšit již stávající hru. I přes veškeré chybějící detaily se mi podařilo splnit zadání a výsledný aplikace je plně funkční.

Summary

The topic of this thesis is the programming of application in the XNA framework. The goal is to create a local multiplayer game on one desktop. This is a 2D action game, which uses sprites created by online editor and some picture editor. The theoretical part of the thesis deals with basics of .NET framework, basics of C#, describing of object oriented programming, general programming of games and basic description of XNA framework. The main goal of the practical part is to create working local multiplayer game on one desktop for four players in the XNA framework. In addition to programming itself, there is also an example of game components design. The end of the practical work is also testing of the created game and optimizing of individual game attributes.

Keywords: C#, XNA framework, .NET framework, object oriented programming, sprites, multiplayer

Seznam použitých zdrojů

C# Developer Job Description Template [Online]. (©Copyright2010-2018). Retrieved March 05, 2018, from https://www.toptal.com/c-sharp/job-description

Herceg, T. (2009). Úvod do .NET Frameworku [Online]. *Dotnetportal.cz*. Retrieved from https://www.dotnetportal.cz/clanek/125/Uvod-do-NET-Frameworku

Čápka, D. (©2018). Úvod do objektově orientovaného programování v C# [Online]. Retrieved March 05, 2018, from https://www.itnetwork.cz/csharp/oop/c-sharp-tutorial-uvod-do-objektove-orientovaneho-programovani

Intel. (©2018). DESIGN [Online]. Retrieved March 15, 2018, from https://software.intel.com/en-us/gamedev/journey/design

Richter, J. (2003). *.NET Framework programování aplikací*. Praha: Grada.

Drayton, P., Neward, T., & Albahari, B. (2003). *C# v kostce: pohotová referenční příručka*. Praha: Grada.

Černohorský, P. (2003). Historie a vývoj jazyka C [Online]. Retrieved March 10, 2018, from https://www.fi.muni.cz/usr/jkucera/pv109/2003p/xcernoh1.htm

Developer.microsoft.com: Any Developer. Any App. Any Platform. [Online]. (©2018). Retrieved March 13, 2018, from https://msdn.microsoft.com/en-us/

Inheritance (C# Programming Guide)| Microsoft Docs [Online]. (2018). Retrieved April 10, 2018, from https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/inheritance

Polymorphism (C# Programming Guide) | Microsoft Docs [Online]. (2018). Retrieved April 10, 2018, from https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/polymorphism

Drumm, L. (2012). *Microsoft XNA 4.0 game development cookbook: over 35 intermediate-advanced recipes for taking your XNA development arsenal further*. Birmingham: Packt Pub.

Nagel, C. (2016). *Professional C# 6 and .Net Core 1.0*. Indianapolis, Indiana: Wrox, a Wiley brand.

Nathan, A. (c2010). *WPF 4 unleashed*. Indianapolis, Ind.: Sams.

Čápka, D. (©2018). Struktura XNA hry [Online]. Itnetwork.cz. Retrieved from https://www.itnetwork.cz/csharp/tvorba-her/monogame/csharp-monogame-tetris/xna-tutorial-struktura-hry

Character Generator [Online]. (2018). Retrieved April 08, 2018, from http://gaurav.munjal.us/Universal-LPC-Spritesheet-Character-Generator/#

Seznam použitých obrázků a ukázek kó

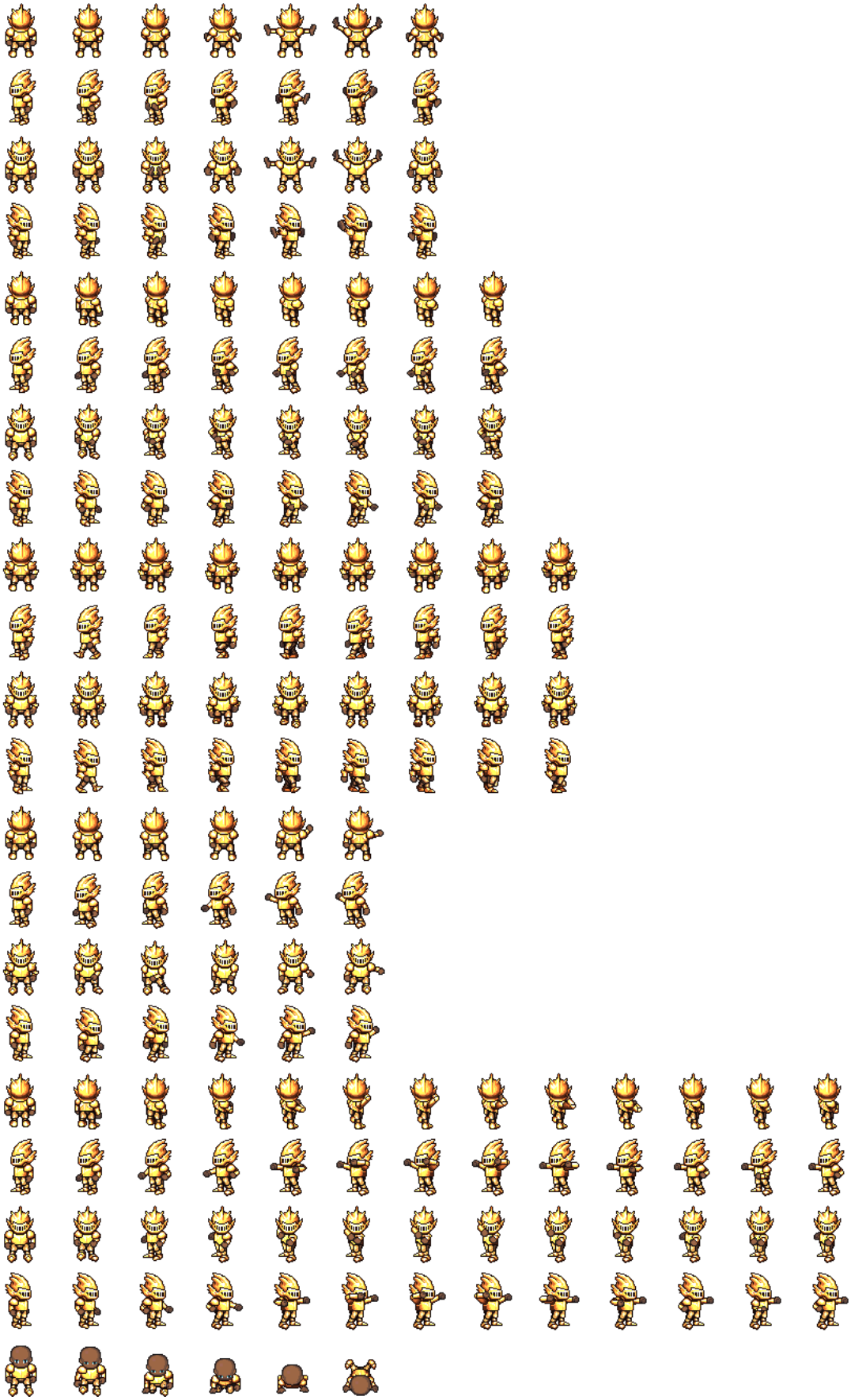
Seznam příloh

Přílohy

Příloha 1: Obsah souboru Start.txt pro načítání hry

```
width ; height ; fullscreen ; music ; music effect
1024 ; 768 ; True  ; 100 ; 100
```

Příloha 2: Sprite pro vykreslení animace hrdiny

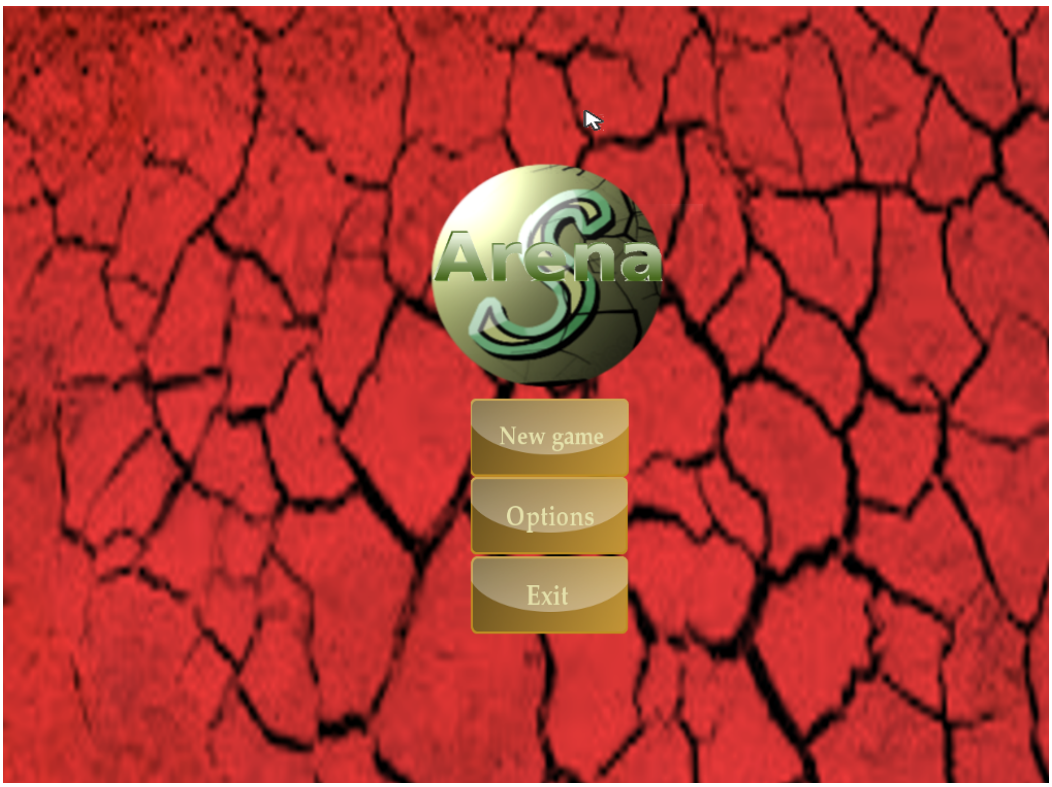


Příloha 1: Proměnné třídy Menu

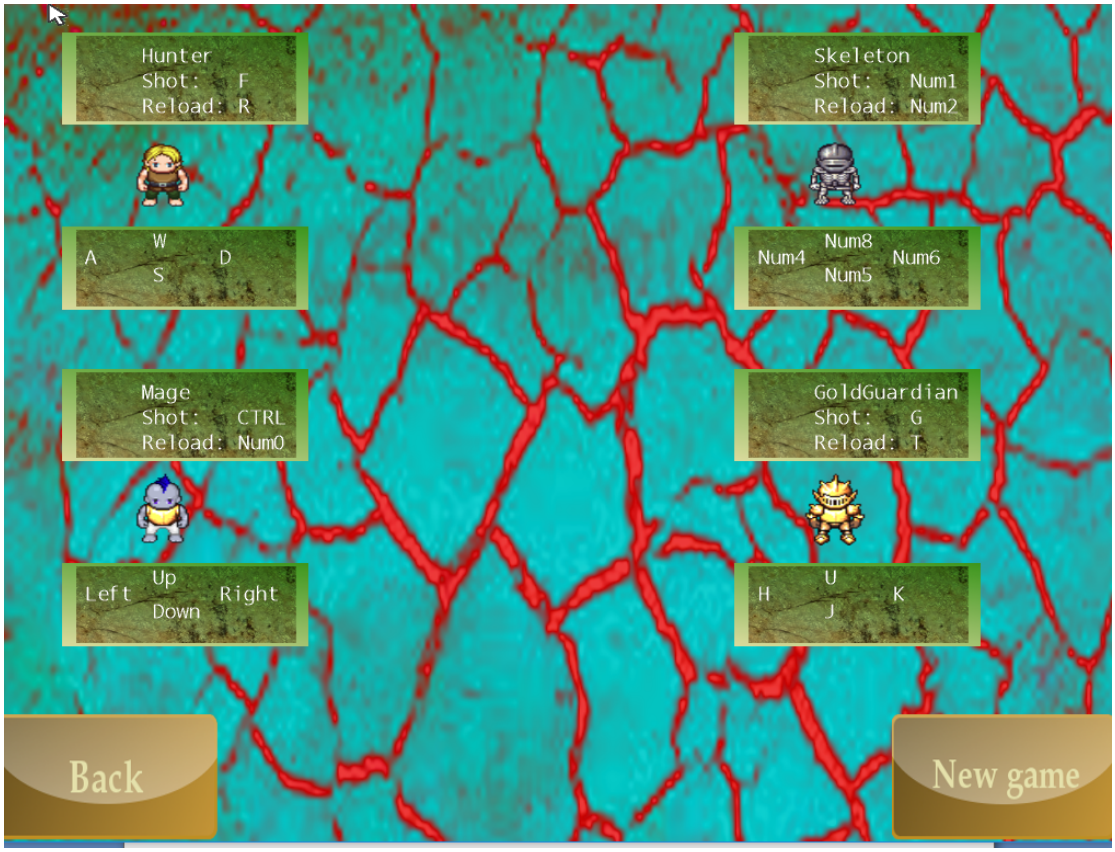
```
    private Game1 game;
    private Controls controls;
    public MenuState menuState = MenuState.Base;
    private OptionState optionState = OptionState.Base;
    private SelectionHero selectionHero;
    private Texture2D background;
    private Texture2D logo;
    private MyMouse mouse; //myš
    private List<Texture2D> te = new List<Texture2D>();
    private int widthButton = 15; //procentuální vyjádření šířky velikosti tlačítka
    private int heightButton = 10; //procentuální vyjádření výšky velikosti tlačítka
    //base
    private Button exitButton;
    private Button newGameButton;
    private Button optionButton;
    //pozice v procentuálním vyjádření tlačítek
    private Vector2 exitPosition = new Vector2(45, 70);
    private Vector2 newGameButtonPosition = new Vector2(45, 50);
    private Vector2 optionButtonPosition = new Vector2(45, 60);
//cesta k texturám tlačítek
    private string exitButtonPath = @"Sprits\Menu\Exit";
    private string newGameButtonPath = @"Sprits\Menu\NewGame";
    private string optionButtonPath = @"Sprits\Menu\Options";
    //option
    private Button volumeButton;
    private Button backButton;
//pozice tlačítek pro nastavení zvuku v Options
    private Vector2 volumeButton = new Vector2(80, 50);
    private Vector2 backButtonPosition = new Vector2(75, 75);
//pozice textur tlačítek v Options
    private string volumeButtonPath = @"Sprits\Menu\Volume";
    private string backButtonPath = @"Sprits\Menu\Back";
    //nastavení tlačítek pro nastavení zvuku
    private Audio audio;
    private int setting_X = 20;
    private int setting_Y = 10;
    private int setting_Widht = 50;
    private int setting_Height = 50;
//nastavení textur pro pozadí a herní logo
    private string textureBackgroundPath = @"Sprits\Menu\Background";
    private string textureLogoPath = @"Sprits\Menu\Logo";
//nastavení pozice a velikosti loga pro zobrazení
    private Rectangle logoRectangle;
    private int X_logo = 37;
    private int Y_logo = 20;
    private int widthLogo = 30;
    private int heightLogo = 30;
```

Příloha 2: Ukázka tlačítek v menu

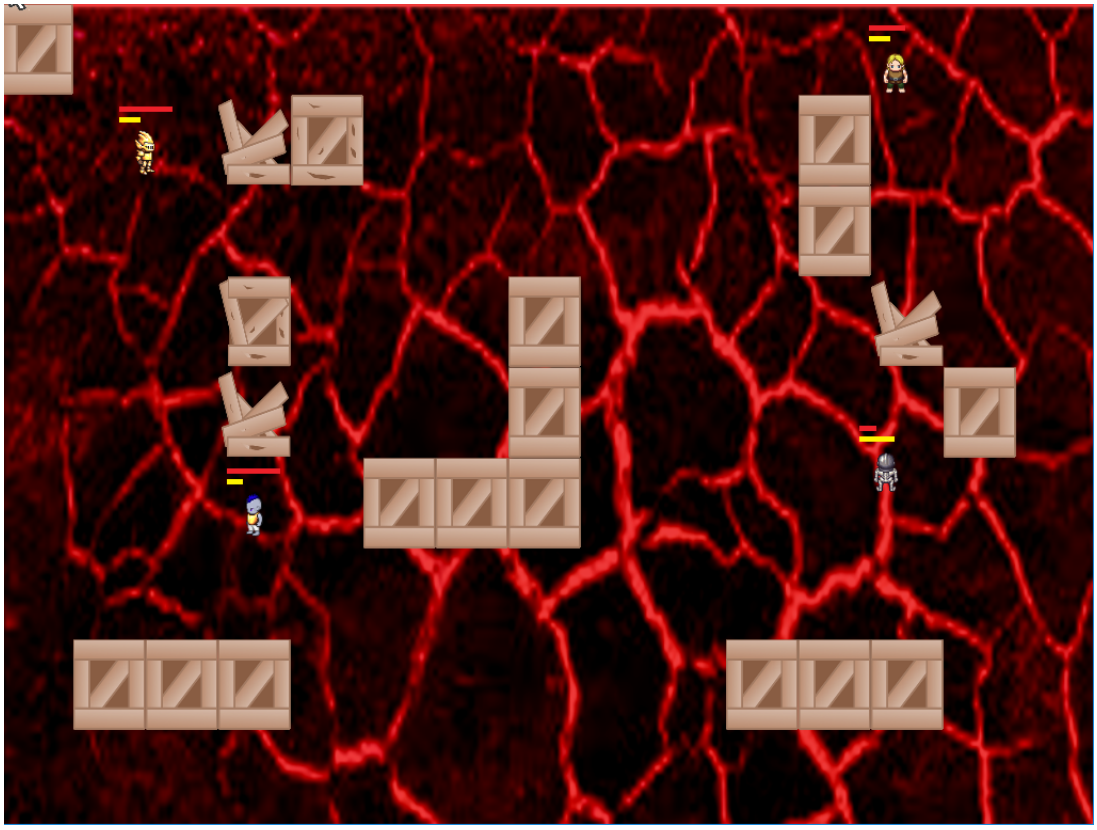
Příloha 3: Ukázka menu



Příloha 4: Ukázka výběru hrdiny

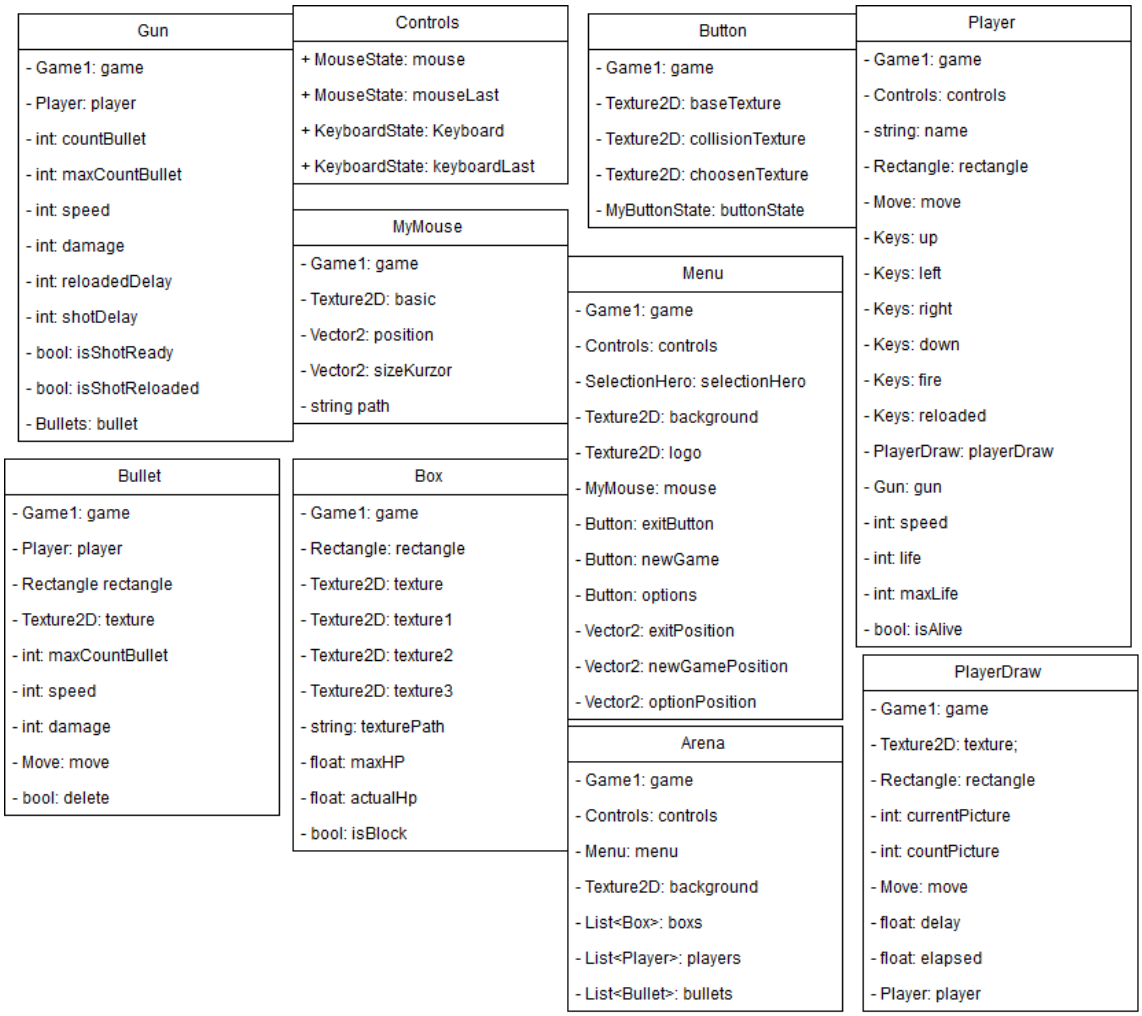


Příloha 5: Ukázka z herního prostředí



Příloha 6: Ukázka vítězného hrdiny

Příloha 9: Diagram návrhu tříd



Příloha 10: Obsah přiloženého CD

K práci je přiložené CD, na kterém je soubor *Zdrojovy_kod.zip*, na kterém jsou všechny hlavní třídy, které se v práci popisují.

Jako další je na disku soubor *Vlastní_hra.zip*, který obsahuje kompletně dokončenou hru, která obsahuje veškerý zdrojový kód, včetně veškerého grafického obsahu a je plně funkční.