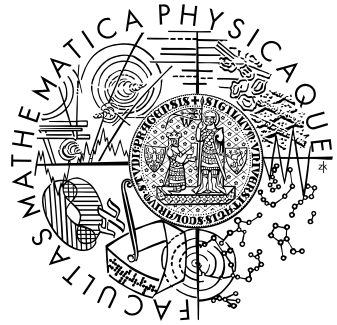
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Michal Staša

Aplikace umělé inteligence v prostředí počítačové hry

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek

Studijní program: Informatika

Studijní obor: Programování

Praha 2011

Velice děkuji Mgr. Pavlu Ježkovi za cenné rady, pomoc a trpělivost při vytváření této práce.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V …….. dne..... Michal Staša

Název práce:	*Aplikace umělé inteligence v prostředí počítačové hry*
Autor:	*Michal Staša*
Katedra:	*Katedra distribuovaných a spolehlivých systémů*
Vedoucí bakalářské práce:	*Mgr. Pavel Ježek*
E-mail vedoucího:	*pavel.jezek@d3s.mff.cuni.cz*
Abstrakt:	*Práce je zaměřená na vytvoření herního prostředí, jehož obsah lze měnit pomocí externích knihoven a systému, který by umožňoval vkládat moduly s umělou inteligencí a podporoval tak její testování. Výsledkem je aplikace napsaná v jazyce C# s využitím knihovny Microsoft XNA Game Studio 3.1. Aplikace obsahuje herní prostředí podobné fotbalu s kouzly a systémem pro vytváření a vkládání nových modulů kouzel a umělé inteligence. Základní modul kouzel a umělé inteligence pak ukazuje, jak moduly vytvářet a vkládat. Umožňuje hru až dvou lidských hráčů a to jak v kooperačním, tak kompetitivním módu.*
Klíčová slova:	*umělá inteligence, XNA, hra*

Title:	*Artificial Intelligence in an Environment of a Computer Game*
Author:	*Michal Staša*
Department:	*Department of Distributed and Dependable Systems*
Supervisor:	*Mgr. Pavel Ježek*
Supervisor's e-mail address:	*pavel.jezek@d3s.mff.cuni.cz*
Abstract:	*The aim of the work is to create a game environment, that has a modular content using external libraries and a mechanism for inserting modules with artificial intelligence as well for testing purposes. The result is an application written in C# programming language with the use of Microsoft XNA Game Studio 3.1 library. The application contains a game environment similar to football with magic and spells and a mechanism for both creating and inserting modules with new spells and artificial intelligence. Basic modules of spells and artificial intelligence shows how to create and insert those modules. It allows up to two human players playing either in the cooperative or competitive mode.*
Keywords:	*artificial intelligence, XNA, game*

Obsah

1 Úvod

Herní projekt „Warlock Football“ (dále zkráceně WF) představuje 2D akční hru s prvky her na hrdiny. Základem jsou klasická, elementární pravidla fotbalu. Dva týmy hrají proti sobě a cílem je porazit soupeře rozdílem vstřelených branek za daný časový úsek. Koncepce umožňuje jak hru pro jednoho hráče, tak pro dva, a to v konkurenčním i kooperačním módu. Nápad skloubit prostředí fantasy se „sportem“ a vytvořit pravidla hry WF vznikl jako součást této práce, která se zabývá zejména návrhem a implementací vnitřních struktur.

V následujících odstavcích této kapitoly jsou popsána pravidla a herní mechanismy hry WF, srovnání s jinými, alespoň z části podobnými projekty, které také sloužili jako inspirace pro její vznik, a hlavní cíle, kterých by práce měla dosáhnout.

1.1 Pravidla a herní mechanismy hry Warlock Football

Jak již bylo řečeno na začátku, v klasickém zápase proti sobě hrají dva týmy složené z jednoho až tří kouzelníků. Jejich úkolem je nastřílet co nejvíce branek. Zápas je časově omezený, v základním nastavení na tři minuty. Tým s vyšším skóre vyhrává. Aby bylo možné plně pochopit herní mechanismus, je nutné vysvětlit, jak vypadá herní svět – mapa, kdo je kouzelník a jak vlastně padne branka.

Zápas se odehrává na mapě s omezenou velikostí, ohraničenou neprostupnými zdmi. Na levé a pravé stěně jsou umístněny branky. Tým začínající vlevo brání svoji levou branku a útočí na pravou branku soupeře a naopak. Hráči se mohou pohybovat po trávníku, zelené ploše, který jim neklade žádná omezení na pohyb a mohou vstoupit i do brankoviště, rovněž bez omezení. Žádný herní objekt, tedy ani hráč, nemůže vystoupit z prostoru daným mapou. Samotná hra samozřejmě takovou akci ani nepřipouští.

Důležitým herním objektem je kulatá hmota energie – „míč“. Míčem lze libovolně pohybovat a to jak nárazem (kopnutím), tak účinky některých kouzel. Standardně však postava utrpí při nárazu vážná poškození. Pakliže se míč dostane celým svým obsahem do brankového pole, padl gól. Hra je pak uvedena do počátečního stavu a skóre patřičně navýšeno.

Postavy, které nejvíce ovlivňují průběh hry, jsou čarodějové[1]. V závislosti na počátečním nastavení je ovládá buď člověk (hráč) nebo umělá inteligence. Každý čaroděj má k dispozici čtyři kouzla, které si před začátkem hry vybral. Kromě kouzel si také před začátkem hry hráč zvolí, v jakých školách magie se jeho čaroděj orientuje pomocí deseti bodů, které mezi ně může libovolně rozdělit. Jejich význam je popsán v dalším odstavci této kapitoly.

Kouzla ovlivňují herní svět mnoha způsoby. Kouzly lze například pohnout míčem, zneškodnit nepřátelského hráče nebo se ubránit útoku. Na vynalézavost použití se již meze nekladou. Každé kouzlo má dobu, kterou je nutné počkat po jeho zakouzlení, než je možné zakouzlit ho znovu. Tento čas se často v angličtině nazývá „cooldown“. Každé kouzlo spadá do jedné ze čtyř tradičních škol magie – oheň, voda, vzduch a země a jeho síla se odvíjí podle čarodějovy znalosti příslušné školy. Při výběru kouzel se hráč dozví, jaké jsou jeho účinky a jakým způsobem jsou ovlivněny vzhledem k jejich původu (škole magie).

1.2 Srovnání s jinými projekty

Projekt WF nevznikl zcela jako originální nápad. Je inspirován některými již známými hrami: Bulánci [1], Machine Ball [2], Warlock Brawl [3], Defense of the Ancients [4]. Nejdůležitější podobnosti ale i rozdíly shrnou stejnojmenné podkapitoly níže.

[1] z anglického archaického výrazu pro čaroděje – „warlock“

1.2.1 Bulánci

Akční hra pro dva až čtyři hráče představuje jednoduchou střílečku, ve které hráči hrají proti sobě, snaží se přežít a při tom zabít soupeře. Bulánci přináší do WF hned několik zajímavých nápadů. Nejdůležitější z nich je 2D svět, ve kterém se odehrává, a pohled přímo shora[1]. Ukazuje, že prostředí je přehledné, jednoduché a hráč se může rychle orientovat a přizpůsobovat. V rozšířené verzi[2] přibylo dalších 5 map a více zbraní. Právě systém zbraní je inspirací pro systém kouzel ve WF a to konkrétně v jednoduchém ovládání – každý kouzelník má jedno tlačítko pro kouzlení a jedno pro výběr a možnosti držet více zbraní najednou. V poslední mapě nazvané Exitus se hráči setkávají s neutrálními duchy, kteří se jim snaží škodit. Nejsou příliš inteligentní, ale účel autorů splňují velice dobře. Po opakovaném hraní se ukazuje, že je možné hru imaginárně pozměnit a hrát kooperačně – hráči se spojí dohromady a snahou se stává zastřelit co nejvíce neutrálních postav, které se náhodně objevují na mapě. Kromě samotného nápadu umělé inteligence tedy Bulánci přináší do WF i kooperační mód.

1.2.2 Machine ball

Futuristický fotbal Machine ball je zábavná hra pro dva hráče kombinující pravidla fotbalu (či spíše hokeje vzhledem k herním mapám, které mají mantinely a nemají auty) a souboje strojů, kde se dva tanky snaží dotlačit míč do brány soupeře. Přínosem této hry byl zejména nápad vytvořit míčovou hru trochu jinak. Hra opět využívá pohledu přímo shora, ačkoliv svět je 3D. Po odehrání pár zápasů nabízí více méně srovnatelnou herní přehlednost jako 2D svět. Kromě podobného herního principu spojuje Machine Ball s WF také fakt, že implementuje v zásadě stejný systém polohování a s tím spjaté i ovládání klávesnicí – dopředu, dozadu a otočit doleva nebo doprava. Rozdíly pak plynou spíše z přirozených fyzikálních rozdílů mezi tankem a kouzelníkem (člověkem).

[1] z anglického „topdown view“
[2] Dnes již není dostupná verze původní.

1.2.3 Warlock Brawl

Projekt se od předchozích liší tím, že neimplementuje vlastní herní engine. Využívá systému skriptovatelných map pro hru Warcraft 3 a mění ji tak z klasické real-time strategie na akční střílečku kouzelníků. Mapa je plně určena pro hru více hráčů a pro kompetitivní účely. Přináší zejména systém kouzel v mírně pozměněné podobě. Hráči si mohou před každým kolem nakoupit nová kouzla či zlepšit ty, co již mají. Je možné rozdělit se do několika týmů a to více méně libovolným způsobem, takže se dá hrát například všichni proti všem, jeden na jednoho, dva na dva, ale i třeba méně standardní jeden proti dvěma. Vítězí vždy ten tým, jehož podmnožina zůstane jako poslední na hrací ploše. Možná je i remíza.

Samotná hra Warcraft 3 nabízí pouze izometrický pohled a dobře tak ukazuje jeho nedostatky pro hru typu Warlock Brawl a jí podobným – čili i WF. Nově příchozím hráčům se hra zdá nepřehledná a je těžké se v ní rychle orientovat, což mapa ale k alespoň obstojnému zahrání vyžaduje. Také implementuje ovládání kouzelníka (hráčské postavy) myší, které je poměrně pohodlné, ale pro účely WF hned ze dvou důvodů nepoužitelné. Možnost hrát WF i ve více hráčů na jednom počítači zabraňuje použití myši, a pokud by si hráč mohl zvolit, dostali bychom se do rozporu s kompetitivním hraním hry, respektive s nutností ovládat hru stejným způsobem v rámci fair-play. Tento druh ovládání by měl smysl pouze pro hru jednoho hráče, který by na něj ale zbytečně přivykl a poté nebyl schopen vyrovnat se těm zvyklým na klasické ovládání na klávesnici.

1.2.4 Defense of the Ancients

Defense of the Ancients (často též zkráceně DotA) je také mapou pro hru Warcraft 3. Je velice dobrým příkladem vysoce kompetitivní hry. Její hráčská báze se momentálně skládá z cca 7 až 10 milionů lidí, nepočítaje Čínu. Ukazuje, že je možné vytvořit virtuální sport, ve kterém lze odlišit mnoho dovedností, které musí hráč ovládnout, aby byl ve hře maximálně efektivní. Rozděluje tak svoji hráčskou základnu

do nespočetného množství podskupin s různou úrovní dosažené herní zkušenosti, šikovnosti. Tím dává velice dobrý příklad, jak by měla vypadat kompetitivní hra.

1.3 Cíle práce

V následujících bodech jsou shrnuty základní cíle práce:

- vytvořit originální, akční, rychlou počítačovou hru
- vytvořit herní engine tak, aby podporoval modularitu obsahu
- vytvořit systém pro vkládání modulů s umělou inteligencí

2 Analýza implementace

Před implementací projektu z návrhu je vždy nutné zvážit možná řešení jednotlivých problémů a rozhodnout se pro ta, která jsou nejvíce vyhovující v dané situaci.

V následujících podkapitolách jsou popsána nejzávažnější rozhodnutí učiněná při přípravě projektu WF: rozdíly v managed a unmanaged kódu vzhledem k programování her, porovnání knihovny XNA Game Studio s jinými knihovnami, možnosti implementace herního a fyzikálního enginu a implementace umělé inteligence.

2.1 Managed kód v herním odvětví

Jádrem herního průmyslu je jednoznačně jazyk C, respektive C++. Všechny větší herní projekty byly postaveny na enginech napsaných právě v tomto jazyce. Je zavedený a tudíž dobře známý s početnou skupinou zastánců. Dává programátorům do rukou možnost plně řídit a optimalizovat[1] běh programu a na začátku vývoje možnost rozmyslet si, jestli by projekt nešel postavit na nějaké již existující knihovně, enginu či frameworku, kterých je s trochou nadsázky téměř nespočetné množství.

Ve větších projektech je možné se setkat s managed kódem zejména při skriptování, což většinou usnadňuje oddělení vývoje herního enginu a herní logiky čili oddělení práce programátorů a designérů[2]. Velice dobrým příkladem takto používaného jazyku je Lua, managed funkcionální jazyk vyvinutý zejména pro skriptování. Sílu tohoto jazyka ukazují zejména reference na projekty, v kterých byl využit. Například světoznámá firma Blizzard jej použila ve svém doposud nejúspěšnějším komerčním projektu World of Warcraft. Skriptovací jazyky, respektive obecně managed jazyky dávají projektu modularitu, možnost zasahovat, upravovat a

[1] Hry se optimalizují zejména v technických částech kódu – grafika, správa paměti a zdrojů (resources), aby byl jejich průběh při hraní co nejplynulejší.

[2] Zde je dobré podotknout, že samozřejmě designér může být programátor a naopak. Jde spíše o rozdělení úkolů v týmu, tedy programátor zde představuje čistě technického vývojáře a designér se zabývá obsahem, logikou a vyvážeností, hratelností hry.

rozšiřovat herní obsah aniž by bylo nutné měnit herní engine a zejména rekompilovat kód.

Menší projekty je možné psát kompletně v managed jazyku. Kromě již zmíněné modularity, mají tyto jazyky i další výhody. Zejména rychlost a jednoduchost psaní kódu. Urychlují vývoj a umožňují vývojářům soustředit více času na jiné problémy, které se nemusí týkat technické části projekt. Jsou tedy vhodné pro menší firmy a nezávislé vývojáře. Dalším přínosem těchto jazyků je větší portabilita. Jazyk C je v tomto ohledu značně těžkopádný a vývoj pro více platforem v něm bývá mnohem obtížnější. V tomto ohledu se těší veliké oblibě jazyk Java. Nejedná se pouze o přenositelnost mezi operačními systémy, ale také o přenositelnost na jiná výpočetní zařízení než jsou stolní počítače, například mobilní telefony, mp3 přehrávače apod. Samozřejmě zásadní nevýhodou managed jazyků je tzv. overhead[1], který souvisí s automatickou správou paměti. S pokrokem technologii je však tento overhead stále zanedbatelnější, zejména pak vzhledem k vývoji menších projektů. Vzhledem ke zrychlení vývoje jsou také managed jazyky žádány z ekonomického hlediska.

Pro projekt WF byl vybrán jazyk C# a to jednak z důvodů uvedených v předchozím odstavci a pak také protože má širokou podporu pro vývoj nezávislých her pro operační systém Windows, jak se ukáže v následující kapitole.

2.2 XNA Game Studio

Od roku 2006 firma Microsoft vydává sadu nástrojů pro vývoj her zvanou Microsoft XNA Game Studio [5]. Od malých výhod jako je jednoduchá instalace a příjemné napojení na Microsoft Visual Studio nabízí studio možnost programovat projekty pro více platforem od Microsoftu, konkrétně Microsoft Windows, Xbox 360, Zune a od verze 4.0 nově „Microsoft Windows Phone 7.

Jádrem XNA Game Studia je tzv. XNA Framework, knihovna založená na .NET Framework 2.0 (.NET Compact Framework 2.0 v případě Xboxu 360), který běží na verzi

[1] Využití větší výpočetní kapacity než by bylo nezbytně nutné.

Common Language Runtime optimalizované pro herní prostředí. Obsahuje třídy pro vytváření a obsluhu základních stavebních kamenů každé hry, například tedy třídy pro základní herní logiku (herní smyčku), grafiku (2D i 3D), vstup (včetně ovladačů a vibrací pro Xbox 360) a herní obsah – textury, zvuky apod. Hry, které běží na XNA Frameworku mohou být z technického hlediska psány v libovolném .NET jazyce, oficiálně jsou však podporovány pouze všechny verze Microsoft Visual Studia 2008, XNA Game Studio Express IDE a jazyk C#.

Výhodou XNA Game Studia je také jeho široká členská základna, velká podpora ze strany vývojářů (Microsoftu) a velice přehledná dokumentace. Pro všechny výše zmíněné výhody a fakta je právě tato knihovna nejvhodnějším adeptem pro implementaci WF.

Existují samozřejmě i jiné knihovny. Většina z nich ale není zdaleka tak obsáhlá a dobře přizpůsobená programování her. Dalo by se uvažovat například o grafické knihovně Managed DirectX. Její vývoj byl ale zastaven. Jejím nástupcem se stala knihovna SlimDX, jež se těší celkem vysoké podpoře, ale je spíše používána pro vývoj multimediálních aplikací pro Windows, respektive XNA má stále lepší podporu pro vývoj her. Jako poslední by ještě bylo dobré zmínit možnost použít nějaký nezávislý projekt. Ty ale zpravidla nebývají natolik robustní a jejich komunity bývají spíše menší, takže je obtížnější najít pomoc při řešení složitějších problémů.

2.2.1 Základní struktura knihovny XNA

Kód knihovny XNA je přehledně strukturován do prostorů jmen. V následujích odstavcích jsou popsány pouze nejzákladnější části knihovny. Detailní popis jednotlivých tříd lze nalézt v online dokumentaci [6].

Kořenovým prostorem jmen je *Microsoft.Xna.Framework* obsahující nejnutnější třídy a struktury, které se běžně používají při programování her. Úplným základem je třída „Game“, která spravuje herní smyčku. Na počátku vytváření hry se obvykle vytvoří třída se jménem hry, například *MySimpleGame*, která od ní dědí. Programátor pak

implementuje virtuální metody *Draw*, kde vykresluje herní svět a *Update*, kde aktualizuje herní logiku. Oběma metodám je v každém běhu herní smyčky předána třída *GameTime* schraňující podrobné informace o uplynulém čase, a to jak herním, tak reálném. Dále se často využívá třída *MathHelper*, ze které lze obdržet některé předpočítané hodnoty, jež se běžně používají, například hodnota pí, a využít její pomocné matematické funkce. Za zmínku pak ještě stojí struktury *Vector2*, *Vector3* a *Vector4* definující vektory s příslušným počtem složek a *Matrix* pro počítání s maticemi.

Velice podstatné jsou pak ještě prostory jmen *Graphics* s třídami pro vykreslování, *Input* umožňující přístup k datům z periferií a *Content* pro načítání herního obsahu, například textur či fontů.

Kromě dokumentace popisuje dobře základy a začátky programování v knihovně XNA první kapitola knihy „Beginning XNA 3.0 Game Programming: From Novice to Professional“ [7].

2.3 Srovnání 2D a 3D her

Z uživatelského pohledu je rozdíl většinou jasně poznat. Konzumní společnost, která má dnes stále vyšší požadavky na grafickou stránku her, nutí velké výrobce pracovat téměř výhradně na 3D projektech. Hodně z nich pak ale končí se špatným hodnocením hratelnosti, protože orientace ve 3D světě je obtížnější než ve 2D a často je nutné mít více ovládacích prvků. Tento problém se úplně netýká tzv. her z vlastního pohledu, tedy her z pohledu samotné hráčské postavy. Přitom precizně zpracovaná 2D grafika může být také velice poutavá. U některých typů her může dokonce být daleko více žádoucí, neboť 2D grafika bývá jednodušší, často přehlednější a hráč se v ní lépe orientuje.

Z technického hlediska je 2D grafika méně náročná jak na výpočetní kapacitu, tak na samotný programátorský vývoj. S počtem dimenzí ve hře také často souvisí fyzikální svět, jímž se zabývá kapitola „Možnosti implementace fyzikálního enginu“. Rovněž výpočty ve 2D fyzikálním světě jsou méně náročné. Menší náročnost na hardware pak

kladně přispívá k portabilitě projektu na jiná zařízení než stolní počítače. Ačkoliv WF není přímo koncipován jako hra pro více platforem, není na škodu vědět, že přenositelnost možná je i když by bylo nutné udělat nějaké změny.

2.4 Herní stavy a jejich správa

Většina herních aplikací sestává z různých částí, ve kterých se hra, respektive uživatel pohybuje – uživatel je v hlavním menu, hraje, nastavuje možnosti zvuku, pozastavil hru apod. Jako nejpřirozenější reprezentace se nabízí konečný stavový automat, jehož princip se pravděpodobně objeví ve všech implementacích. V aplikaci je tedy nutné implementovat konečný stavový automat a podle něj pak řídit její vykreslování a chování.

Asi nejjednodušší implementací je použití výčtového typu a zapamatování si aktuálního stavu. Ten se může změnit při jeho zpracovávání a zároveň je jediný, který může vykreslovat na obrazovku. Implementace není náročná ani na výpočet ani na paměť a pro malé projekty je velice výhodná. Při větším počtu stavů se ale stává méně přehlednou a nedostatečnou.

Často je ale nutné si pamatovat nastavení daného stavu, například po návratu z okna „nastavení“ se očekává, že kurzor v „hlavním menu“ bude na položce „Nastavení“. Dále je dobré si povšimnout, že každá hra má nějaký počáteční stav, například již zmíněné „hlavní menu“. Při přechodu do jiného stavu se buď vracíme směrem k počátečnímu stavu nebo naopak jdeme od něj a překrýváme ho jiným. Tento systém si lze představit jako zásobník. Při přestupu na jiný stav tedy buď odebíráme z vrcholu, nebo na vrchol přidáváme. Možnost vykreslení pak dostává většinou pouze stav na vrcholu zásobníku, který stejně tak zpracovává vstupní události. Výhodou je, že nově vzniklé stavy nezmění nastavení již existujících a k existujícím je jednoduché se vrátit. Systém je jednoduše představitelný jako soustava oken. V takovém případě je pak i rozšiřitelný pro tzv. „pop-upy“, okna překrývající pouze část původního obsahu[1].

[1] Termín „pop-up“ by se na stavech vysvětloval mnohem hůře.

Nutné je pouze modifikovat postup při vykreslování, nejlépe rozdělením oken na viditelná a skrytá a postupem odspoda pak vykreslit všechna viditelná okna. Tím je zaručeno, že okna na vrcholu zásobníku překreslí jejich předchůdce. Možnost zpracování vstupních událostí samozřejmě zůstává pouze nejvýše položenému oknu.

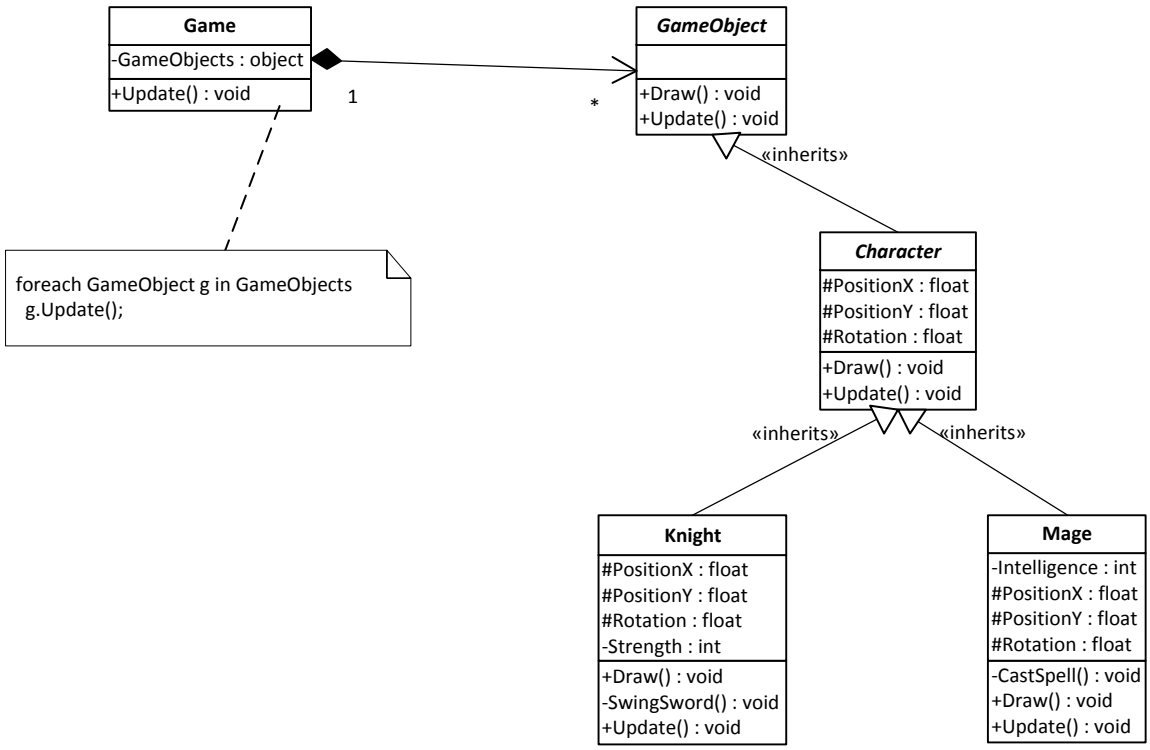
2.5 Koncepty herního enginu

Základem herního enginu je vždy nekonečná smyčka. V ní se většinou řeší jak uživatelské rozhraní, tak událostí odehrávající se ve hře samotné. Tato kapitola se bude zabývat pouze její druhou částí.

Herní svět zpravidla sestává z herních objektů uložených v kontejneru pro rychlý sekvenční průchod. Objekty typicky implementují metody aktualizace a vykreslení, které jsou volány v každém běhu herní smyčky. Následující, dva nejčastěji využívané modely se v této myšlence shodují. Zásadně se však liší právě v implementaci struktury herních objektů, která je popsána v podkapitolách níže. V poslední podkapitole „Srovnání“ jsou pak shrnuty a porovnány výhody a nevýhody obou modelů.

2.5.1 Hierarchický model

Hierarchický model je založen na klasickém principu dědičnosti tříd. Jedná se o jednoduchý, snadno pochopitelný a rychle implementovatelný model. Jeho základem je většinou jedna abstraktní třída představující obecně libovolný herní objekt. Obsahuje pouze nejzákladnější, všem objektům společné vlastnosti a metody, zejména metody pro aktualizaci a vykreslení objektu. Všechny konkrétnější implementace pak dědí od této třídy a rozšiřují ji. Je na nich jasně vidět, jaká je jejich úloha a jaké jsou jejich vlastnosti, což je na jednu stranu přehledné, ale může se ukázat jako omezující.



Obrázek 1 Schéma hierarchického modelu

Obrázek 1 ukazuje jednoduchý příklad. Třída „Game" představuje herní svět a obsahuje pole *GameObjects*[1], což je kontejner pro všechny herní objekty. Metoda „Update" pak při každém běhu smyčky volá na vlastní „Update" na všech obsažených elementech. Důležitou částí je třída *GameObject* nastiňující onu základní, abstraktní třídu, o které byla řeč v prvním odstavci této kapitoly. Z ní budou dědit všechny herní objekty. V tomto příkladu z ní dědí třída *Character*, která je také abstraktní a představuje nějakou postavu v herním světě. Programátor již předpokládá, že taková postava bude mít nějakou pozici a otočení – vlastnosti *PositionX*, *PositionY*, *Rotation*. Ví ale, že takových postav bude více a každá bude mít odlišné vlastnosti a tak v rámci potlačení duplicity v kódu vytváří tuto „mezitřídu". Z ní pak dědí již konkrétní, pravděpodobně koncové třídy *Knight* a *Mage* představující implementaci rytíře a mága. Rytíř má navíc vlastnost *Strength* – síla a „umí" seknout mečem pomocí metody

[1] V jazyce C# by nejjednodušší implementací takového kontejneru byla generická třída „List<T>", kde jako parametr T bychom zvolili třídu „GameObject"

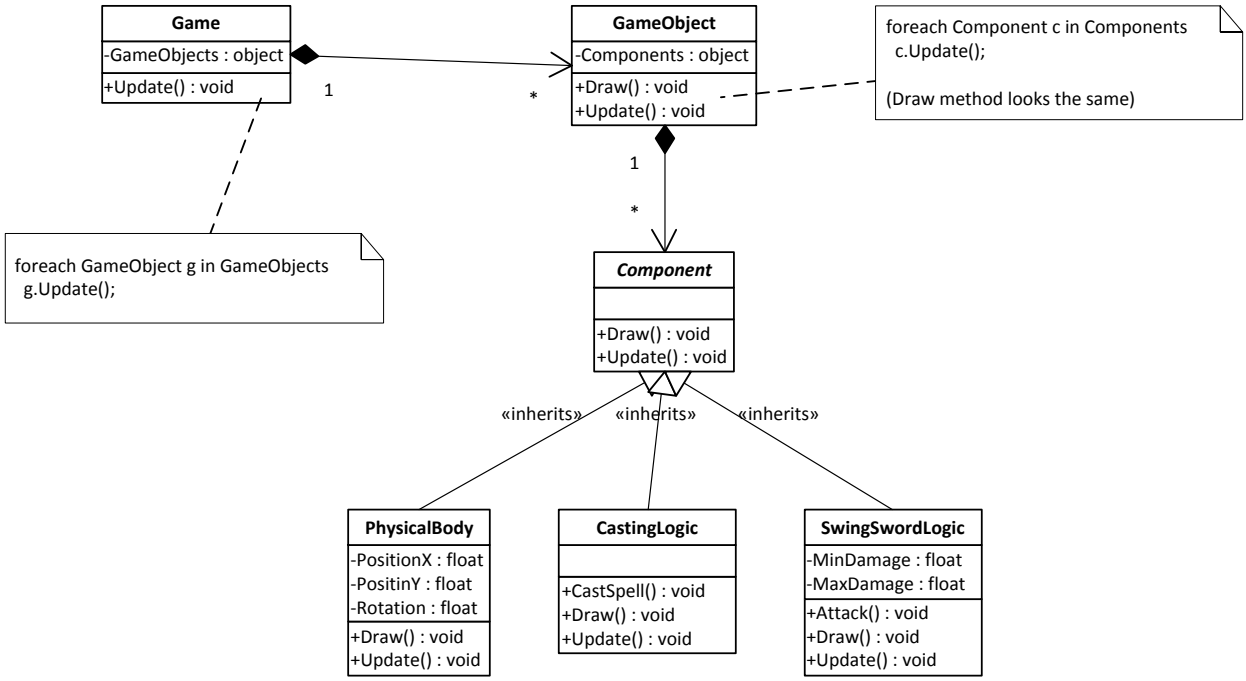
SwingSword. Obdobně má mág vlastnost *Intelligence* – inteligence a kouzlí pomocí metody *CastSpell*.

Hierarchický model je zatím častěji využívám vzhledem k menší výpočetní náročnosti a menšímu overheadu, který při výpočtu vzniká. Ze dvou zmíněných modelů je ten starší, známější a zavedenější.

2.5.2 Komponentový model

Herní objekty v komponentovém modelu jsou koncipovány daleko obecněji. Slouží jen jako kontejner pro komponenty, které až určují jejich vlastnosti. Model v podstatě využívá tzv. principu „duck typing“[1]. Objekty se tak stávají velice flexibilní. Snadno lze měnit jejich funkčnost. Cenou za takovou flexibilitu je menší intuitivnost modelu. Programátor si musí dobře rozmyslet, jak by měly vypadat jednotlivé komponenty, aby nedocházelo k duplicitám v kódu, a musí si dát pozor, aby se jeho komponentový model nezhroutil ve složitější a pomalejší verzi hierarchického modelu.

[1] „Duck typing“ je metoda, při níž objekt není rozpoznáván svým typem, ale svými vlastnostmi, obsahem. Přeloženo z wikipedie [21]: „Jméno konceptu odkazuje na kachní test, připisovaný Jamesi Whitcombovi Rileyi a mohl by být frázován následujícím způsobem: Když vidím ptáka, který chodí jako kachna, plave jako kachna a kváká jako kachna, nazvu ho kachnou.“



Obrázek 2 Schéma komponentového modelu

Obrázek 2 výše představuje jednoduché schéma komponentového modelu. Podobně jako v modelu předchozím obsahuje třídu *Game*, která má i stejnou funkci. První, naprosto zásadní rozdíl je hned ve třídě *GameObject*, která jednak není abstraktní a pak funguje jako kontejner pro třídy obsahující části herní logiky – komponenty. Její metody *Update* a *Draw* volají stejnojmenné metody na všech komponentách, opět při každém průchodu herní smyčkou. Jednotlivé komponenty jsou potomci abstraktní třídy *Component*. V tomto příkladu konkrétně třída (komponenta) *PhysicalBody* reprezentuje fyzikální vlastnosti, třída *CastingLogic* pak dává hernímu objektu schopnost kouzlit a třída *SwingSwordLogic* mu umožňuje sekat mečem. Pakliže by programátor chtěl vytvořit stejné postavy jako v příkladu u předchozího modelu, musel by vytvořit instanci třídy *GameObject* a naplnit ji příslušnými komponentami, tedy pro oba charaktery komponentou *PhysicalBody* a například pro rytíře *SwingSwordLogic*. Je evidentní, že pokud by chtěl vytvořit dokonalou postavu, která umí jak kouzlit, tak sekat mečem, jednoduše by přiřadil obě komponenty do daného herního objektu. Příklad jasně ukazuje flexibilitu komponentového modelu.

Nevýhodou modelu je jeho výpočetní náročnost. Jednak zde vzniká více volání prázdných metod, které je ale poměrně zanedbatelné při menším počtu objektů. Větším problémem je spíše komunikace, závislosti jednotlivých komponent mezi sebou. Ty je částečně možné vyřešit zavedením obecných vlastností pro herní objekt, které jsou sdílené mezi všemi komponentami. Často je ale potřeba volat složitější metody na jiných komponentách. Například komponenta pro pohyb herního objektu nabízí metody dopředu a dozadu a využívá ji komponenta pro ovládání objektu klávesnicí, která už samozřejmě nechce znovu implementovat samotný pohyb. Vzniká tedy závislost mezi dvěma komponentami. Nakreslíme-li si schéma závislostí, pak pokud dvě komponenty závisí jedna na druhé a naopak, je možné tyto komponenty sloučit a závislosti se tak vyhnout. Většinou je ale závislost pouze jednostranná a slučování by vedlo ke zhroucení modelu. Řešením závislostí v modelech se zabývá následující kapitola.

Na komponentovém modelu také vznikají nejrůznější nástroje pro herní vývojáře, například komplexní systém Unity [8].

2.5.3 Srovnání

První z modelů – hierarchický model – využívá stále poměrně značná část větších společností zejména proto, že je známý, praxí dobře prověřený. Pro větší projekty se také hodí více, protože je méně náročný na výpočetní kapacitu.

Komponentový model si pak oblíbili zejména menší vývojáři a tvůrci her, kde je nutná modularita obsahu. Flexibilita a možnost rychle a snadno měnit jednotlivé herní objekty a jejich vlastnosti je jeho zcela zásadní výhodou.

2.5.4 Závěr

Jedním z cílů této práce je právě modularita obsahu. Ani rozsah projektu nijak nenaznačuje, že by měl být pro dnešní techniku náročný na výpočet. Z těchto důvodů byl v této práci vybrán a implementován komponentový model.

2.6 Možnosti implementace fyzikálního enginu

Fyzikální engine je součástí každé hry odehrávající se v reálném čase[1]. Může ale nabývat různých podob. V menších projektech bývá integrovaný přímo v logice hry, ve středně velkých a větších projektech bývá spíše oddělen. Kód je pak přehlednější a lépe se spravuje, případně aktualizuje a zlepšuje. Fyzikální engine nemusí nutně kopírovat fyzikální zákony reálného světa. Může tak činit v zájmu autora hry, například kouzla nemusí nutně podléhat klasickým fyzikálním zákonům, ale zejména tak činí z praktických důvodů. Výpočet komplexního fyzikálního chování by byl příliš náročný pro klasický stolní počítač, konzoly či mobil, jež jsou cílové skupiny herních vývojářů.

Fyzika ve hře reprezentuje pohyb a interakci fyzikálních částí herních objektů, o kterých byla řeč v předchozím odstavci. Důležité je podotknout, že pozice a pohyb zde nemají příliš společného s vyobrazením a zejména pak s animací.[2] Fyzikální svět totiž často používá tzv. herní jednotky[3] jako jednotky míry, které opět nemusí odpovídat žádným jednotkám používaným v reálném světě. Ostatně to už záleží na konkrétních implementacích. Stejně tak se fyzikální enginy liší ve vnímání světa a tím se rozdělují na 2D a 3D. To se samozřejmě projevuje na výpočetní kapacitě, kterou konzumují. 3D fyzikální enginy se jistě dají použít na 2D hry, stačí ignorovat jednu z dimenzí. Programátor ale ztrácí možnost optimalizace pro 2D prostředí a velice pravděpodobně je nucen dělat jednoduché úkony zbytečně složitými způsoby.

V následujících podkapitolách jsou popsány 2D fyzikální knihovny pro C#. Všechny čerpají z projektu Box2D [9] – fyzikální knihovny pro C++. V kapitole srovnání jsou pak zdůrazněny rozdíly mezi těmito enginy a popsány důvody, proč byla knihovna Box2D.XNA vybrána pro projekt WF.

[1] z anglického „real-time game“, čas se v takové hře měří spojitě; opakem je pojem „turn-based game“ – tahová hra, kde se čas měří diskrétně, hráči se v takové hře střídají v jednotlivých tazích jako například v klasických stolních deskových hrách (šach, dáma, go)

[2] v principu, ne v implementaci

[3] z anglického, často používaného spojení „game units“

2.6.1 Box2DX

Knihovna Box2DX [10] je starší projekt a přestože implementuje většinu důležitých funkcí ze svého vzoru, obsahuje spoustu neřešených chyb. Poslední data na jeho domovské stránce naznačují, že poslední aktivita zde proběhla v roce 2008, takže lze předpokládat, že projekt není příliš živý.

2.6.2 Farseer Physics Engine

Fyzikální engine Farseer [11] je nyní plně ve vývoji. Připravovaná verze 3.0, která mimo jiné čerpá také z níže popsaného projektu Box2D.XNA, by měla obsahovat implementaci všech podstatných funkcí včetně tzv. „continous collision detection“[1] (CCD). Aktuální verze 2.1.x čerpá zejména z knihovny Box2DX, je stabilní a nabízí příjemné rozhraní pro práci se světem. Farseer je obecně považovaný za aktivní projekt a má velice dobrou dokumentaci.

2.6.3 Box2D.XNA

Box2D.XNA je kvalitní přenos knihovny Box2D optimalizovaný, jak už jeho název napovídá, pro XNA a technologie, které XNA podporuje, například Xbox 360. Implementuje většinu podstatných funkcí svého předchůdce z C++ včetně CCD. Projekt je vcelku aktivní. Nemá však příliš dobrou dokumentaci a chybí komentáře v kódu. Knihovnu Box2D však kopíruje natolik věrně, že je možné nahlédnout do jejího manuálu a čerpat z něj.

2.6.4 Srovnání

Je lepší volit projekty, knihovny, jejichž komunita je aktivní. Z tohoto prostého důvodu nebyl vybrán první projekt – Box2DX. Srovnání knihovny Farseer s knihovnou Box2D.XNA je náročnější. Komunita a vývojáři projektu Farseer jsou značně aktivnější než v případě druhé knihovny. Farseer nicméně neimplementuje algoritmus pro CCD ve své aktuální, stabilní verzi, který by vzhledem k rychlosti pohybujících se objektů ve hře

[1] Způsob jakým lze zajistit, aby rychle pohybující objekty neprolétávaly jinými fyzikálními objekty. Prolétávání objektů se v anglických textech často označuje názvem „tunneling“.

mohl být podstatný. Proto byla dána přednost knihovně Box2D.XNA. Do budoucna se však počítá s přechodem na projekt Farseer, verze 3.0 (případně vyšší).

2.6.5 Integrace knihovny Box2D.XNA

Má-li dojít k výměně fyzikální knihovny, je nutné projít možnosti jejího napojení do programu.

Nejjednodušším způsobem je používat přímo funkce knihovny všude tam, kde budou třeba. Jednoduchý a na první pohled rychlý a výhodný způsob by se ale mohl projevit jako velice slabé místo při výměně knihovny, protože většina doposavad napsaného kódu by se musela přepsat.

Téměř naprostým opakem je pak vytvoření nové vrstvy mezi programem a fyzikální knihovnou, tak aby implementace této vrstvy zapouzdřila knihovnu. To lze v programovacím jazyce C# udělat pomocí vhodně zvolených abstraktních tříd a rozhraní. Navržení abstraktní fyzikální vrstvy by při výměně knihovny bylo zcela vítané, protože by se nemuselo měnit nic ve stávajícím kódu. Také vzhledem k požadavkům na modularitu obsahu, zejména pokud by nový obsah byl vytvářen třetími stranami, by tato vrstva byla přínosem, ne-li nutností. Nevýhodou takto zvolené cesty je ale velká složitost a režie spojená s vytvářením mezivrstvy. V průběhu vývoje, zejména při vytváření modulu umělé inteligence, se ukázalo, že nelze jednoduše použít strukturu stávající fyzikální knihovny, protože některé části implementace jsou natolik jedinečné, že napojení jiné knihovny by pak bylo velmi složité a krkolomné.

Díky hierarchickému modelu byly v projektu zapouzdřeny jednodušší části fyzikální knihovny. Složitějších částí, jako například hledání fyzikálních objektů v daném obdélníku, bylo ponecháno v původní podobě.

2.7 Umělá inteligence

Jak již Mat Buckland [12] naznačuje ve své knize, umělá inteligence ve hře se značně liší od akademického výzkumu. Zatímco akademici se zabývají zejména optimálním řešením problémů, herní vývojáři se musí zaměřit na výkon výpočtu, zábavnost a prodejnost hry. Technické omezení naznačuje, že je nutné používat rychlejší algoritmy, jejichž výpočet se stihne v jednom průchodu herní smyčky a nezpomalí celkový herní výkon. Míra zábavy je pak odvozena od subjektivního pocitu hráče. Průměrný hráč očekává, že hra bude dostatečně složitá, aby ho zabavila, ale zároveň také očekává, že ji ve většině případů vyhraje[1]. To znamená, že herní vývojáři vlastně ani nechtějí, aby byl algoritmus umělé inteligence optimální. Cílem programátorů by mělo být vytvořit zdání inteligence. Mat Buckland [12] píše: *„…if the player believes the agent he's playing against is intelligent, then it is intelligent.".* Neboli pokud hráč věří, že se soupeř chová inteligentně, pak je inteligentní.

Následující podkapitoly analyzují základní algoritmy často používané při vytváření umělé inteligence a návrh umělé inteligence pro hru WF.

2.7.1 Jednoduchá umělá inteligence

Činnost agentů, entit ovládaných umělou inteligencí, lze často popsat poměrně triviální množinou stavů a popisem přechodů mezi těmito stavy. Jako již v kapitole 2.4 i zde se využívá výpočetního modelu konečného stavového automatu. Vhodným rozvržením stavů a přechodů je možné vytvořit velice reálné chování v poměrně krátkém čase, zejména díky intuitivnosti modelu, podobnosti řešení problémů v reálním světě. Jednoduchá struktura též umožňuje snadnou adaptaci na případné změny. Neposlední výhodou je i rozdělení kódu do přehledných částí, které se dobře ladí. Z technického hlediska jsou konečné automaty nenáročné na výpočet.

[1] Slovo „vyhraje" může být zavádějící. Hráči se musí dařit spíš dobře než špatně vzhledem k tomu, co může v průběhu hry získat nebo čeho může dosáhnout.

Stavové automaty lze různě kombinovat nebo i vnořovat do sebe a vytvořit například stavový automat, kde každým stavem je další stavový automat. Takto je možné dosáhnout značně komplexního chování. Stejně tak je možné udržovat si jak globální data pro více agentů, tak lokální pro jednoho. Nevýhodou je však značný nárůst složitosti a množství kódu.

2.7.2 Cíly řízený agent

Jedním z přístupů jak se dívat na umělou inteligenci agenta je tzv. řízení cíly. Cílem agenta je jakákoliv jeho činnost. Například jedním z úkonů ve fotbalovém zápase je kopnout penaltu, cílem tedy je kopnout penaltu. To znamená rozběhnout se, napřáhnout, zamířit a kopnout do míče. Zamířit například znamená zkontrolovat pozici brankáře, spočítat nejlepší místo kam vystřelit a pak natočit nohu vůči míči tak, aby letěl do spočítané pozice. Jednotlivé cíle lze rozkládat na menší, jednodušší až do doby, kdy už jednotlivé cíle jsou natolik elementární, že nejdou dále rozkládat. V deváté kapitole svojí knihy Mat Buckland [12] pojmenovává velké, rozložitelné cíle složené a nerozložitelné cíle pak atomické. Samotný proces rozkládání cílů nazývá dekompozicí.

Složené i atomické cíle bývají v implementaci reprezentovány hierarchií tříd. Složené cíle pak obsahují frontu podcílů. I cíle implementují stavy, běžně bývají čtyři – cíl je neaktivní (nevykonává se), cíl je aktivní (vykonává se), cíl byl úspěšně dokončen a nebo jeho vykonávání selhalo. U atomických cílů je význam těchto stavů zřejmý. Složené cíle pak většinou přejímají stavy podle průběhu jejich podcílů.

2.7.3 Koncepce umělé inteligence

Při vytváření umělé inteligence je nutné se dívat, co přesně má být ovládáno, jestli jedinci nebo zda-li je možné vyčlenit nějaké skupiny, které by mohli mít podobné chovaní. Programátor musí rozumět světu, pro který vytváří umělou inteligenci a chápat možné pohledy na tento svět a spojitosti v něm.

Ve hře WF proti sobě stojí dva týmy. Tým je tedy jedna ze základních jednotek, o které lze uvažovat. Každý tým je tvořen hráči. Ti jsou dalšími stavebními kameny umělé inteligence. Tým a hráči vytváří určitou hierarchii, jež je možné využít.

V týmovém pojetí problému je tým celistvá jednotka. V rámci týmové strategie rozhoduje, co by měli jednotliví hráči dělat. Buď přímo ovládá jejich chování a nebo jsou hráči silně napojeni na rozhodnutí hlavního týmového agenta. Hierarchii je pak možné chápat tak, že na primární úrovni je tým a sekundární jsou hráči.

Naopak v individuálním pojetí je kladen důraz na hráčské agenty, kteří se sami rozhodují, co je pro tým v danou chvíli nejlepší. Na primární úrovni tedy stojí hráči a tým na sekundární. V takovém modelu je možné, že se sekundární úroveň pouze abstrahuje a je tvořena myšlením individuálních agentů a samozřejmě ji není nutné programovat.

Oba směry chápání problému jsou správné a nelze říci, zda-li je jeden lepší než druhý. Zde pak záleží na implementaci, při níž lze balancovat pomyslný poměr mezi oběma extrémy.

2.7.4 Diverzifikace individuálních strategií

Výběrem jednoho ze způsobů zmíněných v předchozí podkapitole vývoj návrhu umělé inteligence nekončí, spíše naopak. Tak jako v klasickém fotbale i zde lze vnímat jednotlivé rozdíly, role agentů – obránce, záložník, útočník. Jejich cíl je ale společný – zvítězit. Hlavní otázkou je, zda-li jednotlivci budou tohoto cíle dosahovat stejně nebo jinak. Obránce by například měl bránit, pravděpodobně tedy stát poblíž týmové brány a snažit se odvrátit hrozící nebezpečí. Znamená to ale, že se nemůže vůbec vydat do útoku či vstřelit branku? Opět nelze odpovědět jednoznačně a záleží již na konkrétní implementaci. Následující odstavce popisují na příkladech jednotlivé směry, jimiž je možné se ubírat. Vycházejí z předpokladu, že ve hře podobné fotbalu je klíčem k úspěchu útok a obrana. Ve hře WF pak ještě napadání protivníka. V návrhu tedy existují tři role – útočník, obránce a agresor, jehož úkolem je pravě napadání protivníkových

jednotek. I následující příklady jsou ale dovedeny do extrému a opět je nutné podotknout, že záleží pak především na implementaci.

Jedna z možností jak problém chápat je vzít role jako jednotlivé agenty. A přiřadit jim úkoly příslušné dané roli. Obránce stojí blízko brány, odráží míč a kouzlí defensivní kouzla. Útočník naopak stojí na soupeřově straně a snaží se dostat míč do jeho brány, zatímco agresor prochází světem a utočí na protivníky. Každý agent má svůj úkol a nestane se, že by některý z úkolů nebyl vykonáván. Pokud se ale jeden z agentů dostane do situace, kdy by mohl přispět jiné roli, možnost ignoruje.

Druhou cestou je použít role jako možné akce pro libovolného agenta a jeho chování ovlivnit pomocí vhodně zvoleného rozhodovacího algoritmu, který zajistí, aby hráč v situaci kdy je nutné bránit bránil nebo naopak napadl či útočil. Výhodou je, že hráči mohou zastávat libovolné pozice a vždy jsou schopni reagovat na danou situaci. Na druhou stranu se může stát, že se dva agenti rozhodnou provádět stejný úkon a extrémním případě se navzájem vyruší. Dalším positivem je, že z tohoto systému automaticky vyplývá, jak udělat hráče pro souboj jeden na jednoho.

Pro hru WF byl zvolen model, který předchozí příklady částečně kombinuje. Vychází však spíše z druhého. Každý hráč umí vykonávat všechny akce, ale má i roli, která ovlivňuje jeho rozhodovací algoritmus tak, aby spíše tuto roli zastával. Navíc byl vytvořen týmový agent, který volí globální strategii týmu. Ta pak také ovlivňuje rozhodování hráčů.

2.8 Modularita obsahu

Jedním z hlavních cílů práce je modularita obsahu. Tu lze zajistit v různých částech hry. Pro tento projekt postačí možnost výběru kouzel a nastavení umělé inteligence pro jednotlivé týmy, bylo by ale možné vytvořit například ještě výběr herního světa[1]. Ten by pak mohl ovlivnit třeba i pravidla hry - hraje se na počet vstřelených branek či počet smrtí nepřítele apod. V kapitole 0 jsou u jednotlivých částí nastíněna jejich možná

[1] arény, ve které se zápas odehrává

rozšíření, o jejich širších možnostech a dopadu na projekt je pak diskuze v kapitole 4.2. Zde bude popsáno jakým způsobem lze modularity dosáhnout.

O modularitu ve hře je dobré se zasadit téměř vždy, pokud má vidinu rozšiřitelnosti nebo rozsáhlý obsah, na kterém více herních designérů vyvíjí různé části. Ve velkých hrách se často používají skriptovací jazyky[1] jak na obsah, tak právě na umělou inteligenci. Je pak jednoduché přidávat jednotlivé skripty a měnit hru.

Jazyk C# má i charakteristiky skriptovacích jazyků, navíc knihovna obsahuje velice jednoduchý způsob vkládání modulů pomocí knihoven. Použitím tzv. reflexe [13] je možné snadno načíst libovolnou knihovnu za běhu a efektivně využívat její jednotlivé části. Jednou z mála nevýhod, avšak ne nepodstatnou, může být zajištění bezpečnosti kódu v externích knihovnách, například zamezení přístupu na disk. Jednotlivé knihovny je vhodné rozlišit, aby se při načítání nestalo, že by se dvakrát načetl stejný modul. Identifikátory typu jednoduchého čísla nebo řetězce nejsou příliš vhodné. Pravděpodobnost kolize je celkem vysoká. Knihovna .NET obsahuje strukturu *Guid*[2], která reprezentuje spolehlivý identifikátor, díky kterému lze jednotlivé knihovny dobře odlišit. Ve hře WF se používá jak pro rozlišení modulů umělé inteligence, tak pro rozlišení jednotlivých kouzel.

Dalšími možností by pak byl například „.NET Framework Remoting" [14], která by měla zejména výhodu v zabezpečení externího kódu. Její použití je však značně náročné a pokročilé a přesahuje rámec této práce.

2.8.1 Skloubení modularity umělé inteligence a kouzel

Jednotlivé implementace modulů umělé inteligence budou nepochybně chtít, aby jimi ovládané entity měly vybraná kouzla, která jsou, pro připomenutí, velice důležitým, zásadním faktorem hry WF. Různé moduly budou chtít využívat různá kouzla.

[1] viz kapitola 2.1
[2] Globally unique identifier [15], jedinečný identifikátor využívaný v softwarových aplikacích

Z metody rozlišení jednotlivých knihoven pomocí GUID [15] vyplývá jednoduchý způsob, kterým lze zajistit zřejmou selekci kouzel. Tento systém byl v projektu WF použit. Skýtá ale určitá omezení. Svazuje totiž knihovny kouzel a knihovny umělé inteligence.

Zajímavou částí programu by bylo vytvoření obecného systému popisů kouzel, který by umožnil umělé inteligenci vybírat kouzla podle jejich charakteristiky. Například by nevybírala ohnivou kouli, ale vybírala by nějaké dostupné útočné kouzlo. Vytvořit maximálně obecný systém popisu kouzel by bylo velice náročné a přineslo by to pravděpodobně i nevýhody pro jeho uživatele. Ani nejobecnější systém by pravděpodobně nemohl popsat všechny odlišnosti a detaily účinku jednotlivých kouzel. Mohla by se tak ztratit jejich unikátnost a jedinečnost jejich použití.

3 Implementace

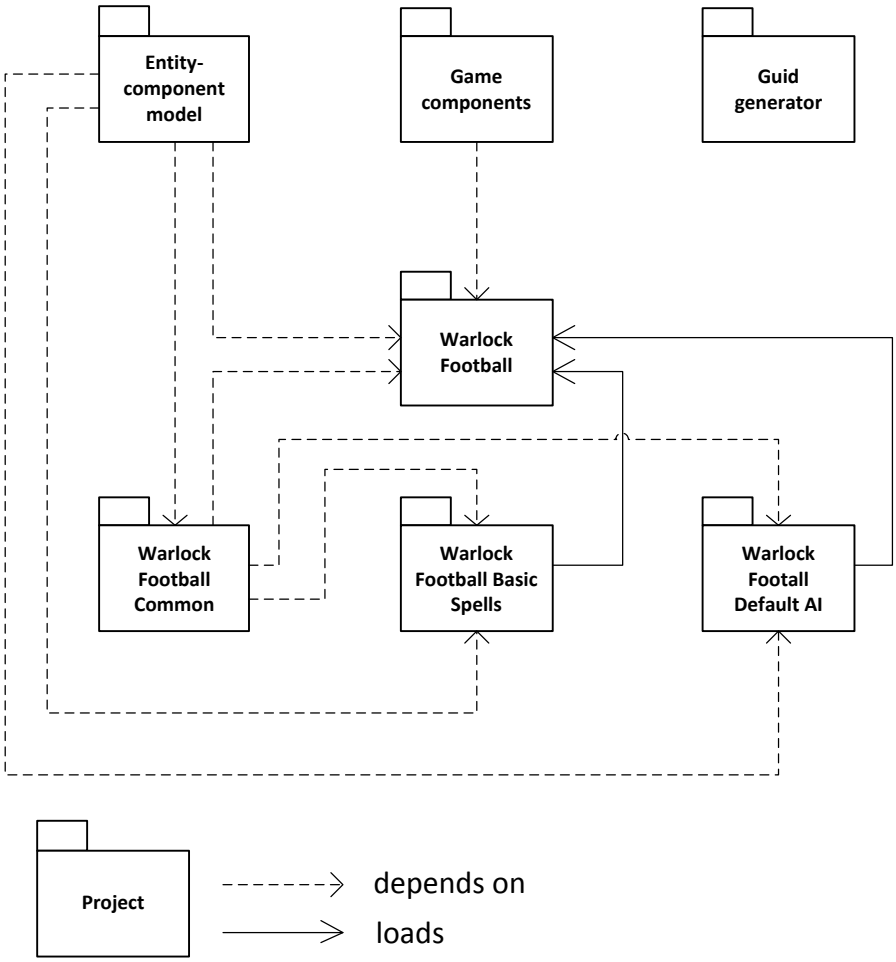
Následující odstavce a kapitoly popisují již konkrétní implementaci výše uvedených problémů a některé další části programu Warlock Football.

V řešení Microsoft Visual Studia pojmenovaném po názvu hry – Warlock Football je celkem sedm projektů:

- **Entity-component model** – knihovna obsahující implementaci komponentového modelu
- **Game components** – knihovna s pomocnými třídami
- **Warlock Football Common** – knihovna zahrnující všechny třídy, které je nutné sdílet mezi hlavní aplikací a knihovnami kouzel a umělé inteligence
- **Warlock Football** – aplikace hry WF, hlavní projekt řešení
- **Warlock Football Basic Spells** – knihovna se základními kouzly
- **Warlock Football Default AI** – knihovna s umělou inteligencí
- **Guid generator** – jednoduchá WPF[1] aplikace pro generování GUID [15]

Jednotlivé projekty spolu samozřejmě souvisejí. Vztahy mezi nimi popisuje diagram na obrázku 3, ve kterém přerušovaná čára značí přímou závislost při kompilaci, například aplikace Warlock Football závisí na knihovně Game Components, a plná čára pak nepřímou závislost označující, že se knihovna načítá až za běhu programu, tedy projekt Warlock Football načítá za běhu knihovnu Warlock Football Basic Spells.

[1] Windows Presentation Foundation [22], knihovna společnosti Microsoft pro vývoj grafického uživatelského rozhraní pro nejen pro Windows



Obrázek 3 Schéma relací mezi projekty

Všechny jmenné prostory začínají prefixem „Santhos“ označujícím jejich tvůrce. Po něm může následovat specifikace použití, ku příkladu „Xna“. Jmenné prostory jsou pak vždy zakončeny názvem projektu. Další části jmenných prostorů pak ve většině případů odpovídají adresářové struktuře projektu. Následují tři příklady vyjmuté z kódu programu:

```
namespace Santhos.Xna.WarlockFootball

namespace Santhos.Xna.WarlockFootball.AI

namespace Santhos.Xna.WarlockFootball.AI.GoalBehaviour
```

Kapitoly níže popisují podrobněji jednotlivé projekty, jejich činnost, obsah a případné důvody pro jejich oddělení od hlavního projektu.

3.1 Projekt Entity-component model

Projekt Entity-component model představuje knihovnu, která obsahuje třídy pro vytváření a správu entit a jejich komponent. Je implementací komponentového modelu popsaného v kapitole 2.5.2 a byl vyčleněn z hlavního projektu, aby byl znovupoužitelný v dalších aplikacích a protože je potřeba tuto knihovnu sdílet při implementaci externích knihoven kouzel a umělé inteligence.

Třída *EntityManager* je kontejner pro jednotlivé entity. Obsahuje metody pro jejich hromadnou správu a registraci. Třída *Entity* představuje jednu instanci libovolné herní entity a je kontejnerem pro komponenty, potomky abstraktní třídy *EntityComponent*. Implementací *EntityComponent* vytváří programátor herní jednotky, jež po vložení dávají jednotlivým entitám konkrétní význam a funkčnost.

3.1.1 Třída EntityManager

EntityManager je statická třída určená pro správu herních objektů, entit, které jsou uloženy v generickém kontejneru *Dictionary*. Každá entita tak dostává unikátní číselný identifikátor, pomocí kterého k ní lze přistupovat. Entitám lze přiřazovat i řetězcové identifikátory, jak bude popsáno v další podkapitole. Registrace entity probíhá při volání jejího konstruktoru, jak bude též popsáno níže. Metodou *Clear* je možné rychle vymazat všechny entity. Jednotlivé entity je možné obdržet voláním prvních dvou metod, všechny herní objekty pak lze obdržet voláním předposlední ze čtyř níže jmenovaných metod:

```
public static Entity GetEntity(int id)

public static Entity GetEntity(string uid)

public static Entity[] GetEntities()

public static void RemoveEntity(Entity ent)
```

K manuálnímu odstranění entity ze systému pak slouží poslední metoda *RemoveEntity*, jejímž parametrem je odstraňovaná entita.

Pomocí třídy *EntityManager* lze také posílat zprávy a to jak všem entitám, tak pouze vybrané. K tomu slouží dvě stejnojmenné metody *DispatchMessage*. K naslouchání globálnách zpráv pak lze použít událost událost *MessageForAllReceived*. Pokud je zpráva posílána pouze jednomu objektu, metoda *DispatchMessage* volá stejnojmennou metodu na této entitě.

K inicalizaci, aktualizaci logiky a vykreslování všech entit pak slouží příslušné metody *Initialize, Update* a *Draw*, které volají stejnojmenné metody na všech herních objektech. Poslední dvě zmíněné se používají v herní smyčce, respektive by se dalo říci, že vytvářejí herní smyčku a průběh hry.

3.1.2 Třída Entity

Třída *Entity* představuje jeden herní objekt, například postavu, svět nebo kouzlo. Je kontejnerem pro jednotlivé komponenty, které určují její vlastnosti a význam ve hře. Kromě metod *Initialize, Destroy, Draw* a *Update*, které volají stejnojmenné metody na všech komponentách, obsahuje metody pro manipulaci s komponentami, konkrétně pro přidávání a odebírání komponent a komponent poskytujících služby a pro vyhledávání služeb.

```
public void AddComponent(EntityComponent entityComponent)

public void AddComponent<T>(EntityComponent entityComponent)

public bool RemoveComponent(EntityComponent entityComponent)

public bool RemoveComponent<T>(EntityComponent entityComponent)

public T FindService<T>()
```

Parametrem *T* při registrování služeb by mělo být rozhraní, které popisuje nabízenou službu. Ty jsou sočástí komunikačního systému komponent v entitě a entit

mezi sebou. Historicky byly jeho první částí. Sdílení informací dále umožňují následující vlastnosti třídy:

```
public string UniqueId { get; private set; }

public int Id { get; private set; }

public HashSet<string> Tags { get; private set; }

public Dictionary<string, object> Properties { get; private set; }
```

Při vytváření entity, voláním konstruktoru, proběhne automatická registrace ve správci entit. Ten ji přiřadí číselný identifikátor *Id*, který je neměnný až do konce života entity. V konstruktoru je také možné přiřadit entitě unikátní řetězcový identifikátor *UniqueId*, který slouží pro vyhledávání jedinečných herních objektů, které by se svým způsobem daly nazývat i „singeltony“[1]. Entitám lze pak přiřazovat jednoznačná označení uložená v instanci generického kontejneru *HashSet* nesoucí identifikátor *Tags*. Libovolné data o entitě je pak možné vkládat do vlastnosti *Properties* reprezentované generickým kontejnerem *Dictonary*. Data se označují řetězcem a ukazuje se na ně přes obecnou třídu *Object*, což je předek všech tříd v jazyce C#. Zejména tato vlastnost umožňuje odlišovat jednotlivé entity od sebe metodou zvanou „duck typing“ popsanou na začátku kapitoly 2.5.2.

Posledním komunikačním nástrojem zejména mezi entitami je pak systém zpráv zmíněný v kapitole výše. Entita zavádí událost *MessageReceived*, na kterou se mohou jednotlivé komponenty navěsit a poslouchat:

```
public event MessageReceivedHandler MessageRecieved;
```

3.1.3 Třída EntityComponent

Třída *EntityComponent* je abstraktní kostrou pro jednotlivé komponenty. Vlastnost *OwnerEntity* ukazuje na třídu entity, do které byla komponenta přiřazena a je inicializována automaticky při přiřazení. V metodě *Initialize* by měl být kód, který

[1] singleton je výraz pro třídu, která je v programu instanciována pouze jednou (nejedná se však o statickou třídu

inicializuje vlastnosti třídy *Entity* a případné vyhledávání služeb, na kterých je komponenta závislá. Neměla by být volána před voláním stejnojmenné metody třídy *EntityManager*, naopak po incializaci všech entit musí být pak volána manuálně. Metody *Update* a Draw jsou volány v herní smyčce. Metoda *Update* by měla obsahovat kód aktualizující logickou část komponenty, *Draw* by pak měla aktualizovat grafické zobrazení komponenty. Poslední metodou je *Destroy*, která by se měla použít v případě, že je potřeba manuálně odstranit nějaké vazby, například poslouchání zpráv od vlastnické entity.

3.2 Projekt „Game components“

Projekt „Game components“ je knihovnou obsahující různé pomocné třídy pro vytváření her. Z hlavního projektu se opět vyčlenil, aby bylo možné ho znovu použít v jiné aplikaci. Třída *Camera* je jednoduchou implementací kamery pohlížející na svět s nastavitelnou pozicí, natočením a přiblížením. V projektu WF se používá pouze staticky. *InputManager* je nadstavbou nad třídami pro vstup z klávesnice v knihovně XNA, která je schopná vracet stav kláves vhodný pro použití při ovládání oken. Třídy *ScreenManager* a *Screen* jsou popsány detailněji v následujících dvou podkapitolách. Jedná se o systém správy oken pro herní aplikace, který byl částečně přepsán z ukázkové aplikace knihovny XNA s názvem „TopDownShooter“ [16].

3.2.1 Třída ScreenManager

V kapitole 2.4, „Herní stavy a jejich správa“, je uvedeno, že každá aplikace prochází určitými základními stavy a ty lze chápat i jako okna. *ScreenManager* představuje správce těchto oken. Obsahuje metody pro jejich přidávání, odebírání a inicializaci. Při konstrukci instance této třídy se inicializuje i tzv. „hlavní menu“, okno, které slouží jako hlavní rozcestí stavů. Je dobré si všimnout, že i samotná hra se pak musí obalit herním oknem. Metoda *Update* se stará o logickou správu jednotlivých oken a podle jejich pozice na zásobníku a některých příznaků nastavuje jejich stav. Tyto

stavy by bylo možné rozšířit tak, aby bylo možné například lépe graficky znázornit jejich inicializaci nebo odstranění.

3.2.2 Třída Screen

Abstraktní třída *Screen* je prototypem implementace jednotlivých tříd oken. Schraňuje informace o stavu okna, vlastnost *IsPopup* udává, zda-li má překrývat okna pod sebou nebo se vykreslovat na ních a její abstraktní a virtuální metody řídí inicializaci a aktualizaci logiky okna a grafického obsahu. Závěrem implementuje rozhraní *IDisposable* dostupné z .NETu, které ukládá třídě implementovat metodu *Dispose* sloužící zejména ke správnému uvolnění alokovaných zdrojů. Protože je ale třída *Screen* abstraktní, metodu pouze nastavuje na virtuální a ponechává její implementaci na potomcích této třídy.

3.3 Projekt „Warlock Football Common“

Knihovna „Warlock Football Common“ obsahuje třídy, které jsou sdílené mezi hlavním projektem hry a knihovnami kouzel a umělé inteligence. Pro tyto externí představuje vstupní bod do celého systému.

Jednotlivé třídy implementují většinu komponent použitých v herních entitách a lze je tudíž použít i v externích knihovnách. Obsahuje také rozhrání pro jednotlivé služby, které nabízejí již zmíněné komponenty a obsahuje též některé další rozhraní, jež mají implementaci v hlavním projektu. Její součástí je i schéma použité pro načítání externích knihoven a předky tříd v nich vyhledávaných.

3.3.1 Třída AdvancedSteeringComponent

AdvancedSteeringComponent je třída, která implementuje některé základní algoritmy chování související s pohybem entity často používané při tvorbě umělé inteligence. Výčtový typ *AdvancedMovementMode* pak nabízí jejich rychlý přehled.

Nastavením vlastnosti *MovementMode* lze vybírat mezi jednotlivými algoritmy. V následujících odstavcích jsou vysvětleny jejich aspekty. Předtím je ale nutné ještě zmínit

dvě další vlastnosti této třídy, jež se nastavují externě a slouží jako cílové destinace: *TargetPosition* (cílová pozice) a *TargetEntity* (cílová entita).

Hold position je klidový stav entity a používá se jako výchozí hodnota. V tomto stavu tedy entita nic nedělá.

Aim je metoda pro míření. Podle vlastnosti *TargetPosition* se entita natočí tak, aby směřovala čelem do daného místa.

Seek a Flee jsou dva antagonistické algoritmy. Oba využívají vlastnosti cílové pozice. První z nich nasměruje entitu maximální rychlostí k tomuto místu. Druhý naopak maximální rychlostí od zadané pozice. Specialitou hry WF je, že při útěku od pozice entita směřuje čelem místu, odkud utíká. Vzhledem k návrhu hry je to pro ni výhodnější. Důležité je poznamenat, že první jmenovaný algoritmus cílovou pozici přejede a pak se k ní opět vrací.

Algoritmus *Arrive* je obdobou algoritmu *Seek* a ve své implementaci ho dokonce částečně využívá. *Arrive* nejprve nasměruje entitu k zadané pozici maximální rychlostí a pak zvolna zpomaluje tak, aby k danému místu dojela. Je zde nastavená určitá tolerance, aby bylo zaručeno, že pohyb vždy skončí. Když je entita na správné pozici, automaticky je nastaven algoritmus *Hold position.*

Zbylé dva algoritmy budou opět uvedeny v páru. Jsou to *Pursue* a *Seek*. Tyto metody využívají druhé vlastnosti – *TargetEntity*. Algoritmus *Pursue* pronásleduje cílovou entitu. Je založen na metodě *Seek* a podle fyzikální komponenty zadané entity počítá její budoucí pozici a snaží se ji dostihnout. *Evade* je pak opět jeho protipólem a snaží se od predikované pozice ujet.

Třída implementuje rozhraní *IAdvancedSteeringComponent*, kterým dají nastavit výše zmíněné parametry. Takto je umožněno umělé inteligenci ovládat entitu.

3.3.2 Třída AnimationComponent

Komponenta *AnimationComponent* představuje zobrazení entity pomocí animace. V konstruktoru dostává třídu *AnimationWrapper*, která zaštiťuje jednotlivé parametry animace. Předpokládá se standardní způsob animace – sekvence za sebou rychle jdoucích obrázků. Pomocná třída obsahuje vlastnosti pro nastavení textury, určení počtu obrázků, nastavení pozice prvního obrázku apod.

3.3.3 Třída CastingComponent

Díky komponentě *CastingComponent* je umožněno entitám kouzlit. Jedna komponenta může obsahovat maximálně čtyři kouzla, která jsou reprezentována potomky třídy *SpellConjuration*, jež bude popsána níže v textu. Komponenta se také stará o zobrazení vybraného kouzla. (Vykresluje text pod entitou.) A při aktualizaci logiky nabízí kouzlům možnost také aktualizovat svůj stav.

Třída implementuje rozhraní *ICastingComponent*. To představuje službu přes, kterou je možno vybírat a čarovat kouzla z jiných komponent.

3.3.4 Třída HealthComponent

Jako první ze jmenovaných komponent třída *HealthComponent* zavádí na objektu entity nové vlastnosti. Ty se týkají životaschopnosti entity. Určují kolik zranění je entita schopná snést, co se stane ve chvíli, kdy dostává zranění a také jaké budou následky v případě překročení maximální hranice.

Ve hře WF se používá pro jednotky kouzelníků a pro zdi, které lze zničit. Nastavením jejích parametrů je možné zajistit, aby se jednotka znovu objevila na hrací ploše, pakliže zemře nebo aby byla zničena.

Třída poslouchá příchozí komunikaci a pakliže přijde zpráva o kontaktu s jinou entitou. Ta pokud má nastavenou vlastnost *ImpactDamage* udělí daný počet zranění této komponentě. Toto je příklad již zmíněné metody „duck typing“ (kapitola 2.5.2).

Komponentu nezajímá s jakou jednotkou se srazila. Podstatná je pouze informace, že jí byl udělen nějaký počet zranění.

Třída pak dále obstarává vykreslení grafického znázornění poměru stávajícího počtu životů a maximálního počtu.

Též implementuje rozhraní *IHealthComponent*, které definuje metodu *ModifyHealth*, která by se měla použít vždy, když má dojít k manipulaci s počtem životů. Implementace této metody je dokonce připravena i na stav, kdy jednotlivé entity budou mít nastaveny odolnosti vůči různým typům zranění.

3.3.5 Třída PhysicalBodyComponent

PhysicalBodyComponent je jedna ze tříd, která obsahuje implementaci fyziky. Je postavena na třídě *Body* z knihovny Box2D.XNA, kterou částečně zabaluje. Na vstupu přijímá struktury této knihovny, které detailně popisují chování entity v simulaci fyzikálního světa.

Komponenta implementuje rozsáhlé rozhraní *IPhysicalBodyComponent*. Toto rozhraní reprezentuje infomační bod, kde jednotlivé komponenty této i jiných entit mohou získat data z fyzikálního světa, například pozici, otočení, směr pohybu a další.

3.3.6 Třída SpriteTextureComponent

Jednou z nejstarších tříd v celém řešení je *SpriteTextureComponent*. Vykresluje danou texturu na základě dat obdržených z fyzikální komponenty, o níž byla řeč v podkapitole výše. (Používá například pozici, rotaci a velikost tělesa.) V konstruktoru dostává třídu *TextureWrapper* obalující ruzné parametry textury.

Textury použité ve hře byly buď vytvořeny speciálně pro tuto hru a nebo měly „volnou“ licenci byly nalezeny v následujících zdrojích: Lost Garden [17], TIGForums [18] a TIGForums [19].

3.3.7 Třída SteeringComponent

Komponenta *SteeringComponent* byla původně zamýšlena jako služba, ke které se budou obracet třídy ovládající pohyb herního objektu. Je nadstavbou fyzikální komponenty a umožňuje čtyři druhy pohybu: dopředu, dozadu, otočit vlevo a vpravo. Zejména pak bylo zamýšleno, že ji bude používat využívat třídy pro ovládání z klávesnice i třídy umělé inteligence, aby měly stejné podmínky a hra tak byla zcel „fér“. Při implementaci se však tento záměr ukázal jako nedosažitelný, přestnost umělé inteligence by pak byla velice nízká, a tak ji využívá dále jen komponenta pro ovládání postavy z klávesnice.

Z předchozího odstavce je patrné, že třída musí implementovat nějakou službu. Konkrétně tedy rozhraní *ISteeringComponent* nabízí službu, kterou lze vypnout a zapnout jednotlivé jednoduché úkony pohybu.

3.3.8 Jmenný prostor DescriptorSchema

Jmenný prostor *DescriptorSchema* obsahuje dvě důležité třídy, které představují obecný systém pro načítání knihoven a umožňují tak modularitu aplikaci WF.

Abstraktní třída *Descriptor* popisuje obecnou implementaci popisu nějaké unikátní funkcionality, kterou je nutné načíst za běhu. Vlastnost *Guid* pak představuje identifikátor, dle kterého lze jednotlivé knihovny, respektive funkcionality, odlišit. Pokud by se identifikátory shodovaly, bude načtena pouze první z nalezených knihoven. O samotné načítání se pak stará generická třída *DescriptorLoader*:

```
class DescriptorLoader<DescriptorClass> where
    DescriptorClass : Descriptor
```

V konstruktoru se nastaví cílový adresář, ve kterém po zavolání metody *LoadDescriptors* proběhne vyhledávání souborů s příponou „.dll“.

```
string[] files=Directory.GetFiles(dllDirectoryFullPath, "*.dll");
```

Knihovna je načtena pomocí reflexe a následně jsou v ní, také pomocí reflexe, vyhledáni všichni potomci třídy *DescriptorClass*, parametru třídy *DescriptorLoader*:

```
Assembly assembly = Assembly.LoadFile(fileName);
foreach (Type type in assembly.GetTypes())
if (type.IsClass && type.IsSubclassOf(typeof(DescriptorClass)))
```

Třída *DescriptorClass* je potomkem abstraktní třídy *Descriptor*. Poté proběhne vytvoření instance, kontrola GUID a registrace nově vytvořené instance v generickém kontejneru, kde klíčem je právě GUID a hodnotou pak instance třídy *DescriptorClass*:

```
(DescriptorClass)Activator.CreateInstance(type)
...
descriptors.Add(descriptor.Guid, descriptor);
```

Tato část by ještě potřebovala vylepšit tak, aby odolala poškozeným knihovnám a pádu a při problému vypsala chybovou hlášku. Zatím se předpokládá, že načítání proběhne v pořádku.

Metodami *GetDescriptors* lze pak obdržet pole načtených deskriptorů.

3.3.9 Jmenný prostor Spells

V jmenném prostoru *Spells* se nachází třídy pro načítání knihoven s kouzly a jejich popis.

Třída *SpellLoader* je obalem generické třídy *DescriptorLoader*. Vyhledává knihovny kouzel v adresáři „Spells“ umístěném v kořenovém adresáři hry.

Potomek třídy *Descriptor*, třída *SpellDescriptor* obsahuje základní vlastnosti popisu kouzla: jméno, popis účinků, typ kouzla. Obsahuje též virtuální metodu *GetSpellConjuration*, kterou by měl implementovat každý potomek. Ta vrací abstraktní třídu *SpellConjuration*.

Třída *SpellConjuration* se vkládá do třídy *CastingComponent* a dává tak entitě možnost kouzlit nějaké kouzlo. Pro lepší představu je třída *SpellConjuration* pergamen, na kterém je napsáno, jak vykouzlit popsané kouzlo.

Komponenta *CastingComponent* je v případě hráčských postav naplněna na začátku hry dle výběru z menu. Knihovny umělé inteligence musí volat manuálně metodu pro obdržení popisů kouzel a pak z ní rovněž manuálně vybrat kouzla pro své hráče.

3.3.10 Jmenný prostor AI

Ve jmenném prostoru *AI* sídlí třídy načítající knihovny umělé inteligence.

Třída *AILoader* je opět obalem třídy *DescriptorLoader*. Z adresáře „AI“ umístěném v kořenovém adresáři hry vyhledává jednotlivé moduly umělé inteligence.

Abstraktní třída *AIDescriptor* popisuje jednoduše modul umělé inteligence. Dává mu jméno a při její implementaci programátora nutí naplnit metodu *CreateAI* kódem, který inicializuje třídy umělé inteligence. Metoda je volána na konci inicializace hry, ale před voláním metody *EntityManager.Initialize*, takže není nutné volat inicializační metody entit manuálně.

3.3.11 Třída GameUnitsConvertor

Třída *GameUnitsConvertor* je převodníkem mezi herními jenotkami a pixely. Obsahuje též konstanty použité při převodech.

3.3.12 Třídy vlastností hry WF

V projektu „Warlock Football Common“ dále sídlí třídy obsahující různé konstanty použité ve hře. Jsou to vesměs statické třídy, které by se daly spojit do jedné velké třídy, ale pro přehlednost tak není učiněno. Jsou to například třídy *EntityProperties*, *GameConstants*, *MapConstants* apod.

3.4 Projekt „Warlock Football“

Ačkoliv je projekt „Warlock Football“ hlavním projektem a představuje „spustitelnou část programu“, větší část logiky obsahuje předchozí část, protože je nutné ji sdílet s externími knihovnami.

Jmenný prostor *Screens* obsahuje jednotlivá okna (globální stavy), do kterých se muže hra dostat, např.: nastavení kouzelníků hráčem, menu nové hry, samotné hraní hry apod.

Projekt pak dále obsahuje některé komponenty entit, ke kterým bylo třeba zamezit přístup, například komponenty vkládané do entity světa. V projektu „Warlock Football Common“ jsou pak obsažena jednotlivá rozhraní k těmto komponentám.

3.4.1 Třída PlayScreen

Třída *PlayScreen* je potomkem abstraktní třídy *Screen* popsané v kapitole 3.2.2 a představuje stav, kdy se odehrává hra WF. Na počátku dostává objekt *PlayScreenDescription*, který se vytvoří v menu nové hry (*NewGameScreen*) a projde všemi dalšími nastaveními. Inicializace tohoto objektu tedy probíhá způsobem valící se sněhové koule, na níž se stále nabalují nové informace. Ty jsou pak použity pro inicializaci jednotlivých herních entit, které jsou ve hře již od počátku. Inicializační metody jsou zejména:

- `Entity CreateWorld() – incializuje entitu světa (herní mapy)`
- `void CreateWarlocks() – incializuje hráče`
- `void CreateAI(...) – incializuje umělou inteligenci`
- `Entity CreateBall() – inicializuje entitu míče`

3.4.2 Třída KeyboardControlComponent

KeyboardControlComponent je třída, která využívá služby nabízené rozhraním *ISteeringComponent* popsané v kapitole 3.3.7. Napojuje entitu na ovládání z klávesnice,

k čemuž využívá třídu *InputManager* z kapitoly 3.2. Vkládá se pouze do hráčem vybraných entit.

3.4.3 Třída PhysicalWorldComponent

Komponenta *PhysicalWorldComponent* je obalem pro třídu *World* v knihovně Box2D.XNA, která slouží k simulaci fyzikálního obrazu světa. Do této simulace je také nutné registrovat jednotlivá fyzikální tělesa (objekty *Body)*, o kterých se píše v kapitole 3.3.5.

Přes rozhraní *IPhysicalWorldComponent* lze přistupovat k metodám, které nabízí Box2D, například vyhledávání v daném obdélníku nebo kolizní „test paprskem“[1].

3.4.4 Třída WorldLogicComponent

Třída *WorldLogicComponent* představuje základní pravidla hry WF. Počítá čas do konce utkání a skóre jednotlivých stran. V metodě *Draw* pak obojí vykresluje na hrací plochu.

Přes rozhraní IWorldLogicComponent lze pak výše popsané údaje jednoduše získat.

3.4.5 Třída TileMapComponent

TileMapComponent je podobně jako předchozí komponenta třídou, vytvářející hranice pro hru WF. Konkrétně představuje herní plochu, mapu, na které se celý zápas odehrává. Mapa je složená z malých dílků, které jsou vedle sebe poskládány tak, aby vytvářely celistvou plochu.

Typy dílků jsou popsány výčtovým typem *TileType*, který je obsažen ve sdíleném projektu „Warlock Football Common“. V inicializační metodě *InitializeTiles* je pak ke každému typu přiřazen objekt *TextureWrapper* nesoucí informace o vzhledu daného dílku.

[1] volně přeloženo z anglického „ray casting“

Systém momentálně automaticky generuje jednoduchou mapu, ale je připraven na případné rozšíření. Popis mapy by pak mohl být přijít ze vstupu a hry by se tak mohly odehrávat v různých prostředích.

3.5 Projekt „Warlock Football Basic Spells“

Projekt „Warlock Football Basic Spells“ je jednou ze dvou externích knihoven, které jsou načítané za běhu. Obsahuje třídy, které popisují základní kouzla hry WF a ukazují, jakým způsobem lze k tvorbě kouzel přistupovat.

Kouzla implementovaná ve hře jsou: ohnivá koule, větrný vír, kamenná zeď, trní a větrný štít. V následujících dvou kapitolách jsou popsány dvě kouzla, jejichž přistup k řešení problému je svým způsobem odlišný.

3.5.1 Ohnivá koule (Fireball)

Ohnivá koule je velice standardní kouzlo. Představuje kouli letící prostorem, která po určité době nebo po nárazu vybuchne a způsobí kolem sebe určité škody.

Třída *FireballDescriptor* je potomkem třídy *SpellDescriptor* popsané v kapitole 3.3.9. Obsahuje inicializaci unikátního identifikátoru GUID a textových polí s vysvětlivky pro uživatele. Implementace virtuální metody *GetSpellConjuration* pak vrací potomka třídy *SpellConjuration*, *FireballConjuration*.

Při zavolání virtuální metody *CreateSpell* se vytváří novou herní entita – ohnivá koule. Ta je v zápětí naplněna novými komponentami: fyzikální reprezentací, animací letu ohnivé koule, animací výbuchu a logikou ohnivé koule.

FireballLogicComponent představuje logiku kouzla – zde určuje její dolet, účinek a chování při nárazu. V momentě výbuchu pak hledá okolní tělesa a ovlivňuje je silou od středu a podle vzdálenosti dopadu jim uděluje zranění.

3.5.2 Větrný vír (Tornado) a kamenná zeď (Stone wall)

Kouzla větrný vír a kamenná zeď jsou velice podobná v implementaci kouzlu ohnivá koule popisovaném v předchozí podkapitole.

Kouzlo větrný vír rovněž sestává ze třech tříd – *TornadoDescriptor*, *TornadoConjuration* a *TornadoLogciComponent* – které mají stejný význam a účel jako třídy popsané u ohnivé koule.

Kouzlo kamenná zeď je mírně odlišné. Efekt kouzla je popsán již vytvořenými komponentami. Třída *StoneWallLogicComponent se využívá pouze ke zničení entity ve chvíli, kdy padne branka.*

3.5.3 Trní (Thorns) a větrný štít (Wind shield)

Ačkoliv mají kouzla trní a větrný štít opět stejnou strukturu, potomky tříd *SpellDescriptor* a *SpellConjuration* a třídu obsahující logiku kouzla, tak se od výše zmíněných značně liší.

Komponenta s logikou kouzla, *ThornsLogicComponent* a *ShieldLogicComponent*, se nyní nepojí na novou entitu. Naopak se napojují přímo na entitu kouzelníka, kde mění jeho vlastnosti v kontejneru *Properties* a po předem stanovené době se pak automaticky z entity zase odstraní a jejich efekt pomine.

3.6 Projekt „Warlock Football Default AI“

„Warlock Football Default AI“ je druhá externí knihovna. Implementuje umělou inteligenci pro tým ve hře WF. Čerpá z poznatků analýzy v kapitole 2.7 a její velká část se zaměřuje na implementaci jedné z individuálních strategií popsaných v podkapitole 2.7.4.

3.6.1 Třída DefaultAIDescriptor

Třída *DefaultAIDescriptor* je potomkem třídy *AIDescriptor* zmíněné v kapitole 3.3.10. V implementaci metody *CreateAI* inicializuje jednotlivé třídy umělé inteligence podle vstupních údajů, počtu hráčů a informace, zda-li je v týmu lidský hráč.

3.6.2 Entita TeamStrategy

První třídou, která je vytvořena při inicializaci umělé inteligence je entita, která obsahuje vlastnosti popisující globální strategii týmu. Při inicializaci spočítá z entity světa, kterou vyhledá přes třídu *EntityManager*, pozici soupeřova středu branky a středu branky týmu.

V průběhu hry pak v metodě *Update* aktualizuje globální týmovou strategii podle vzdálenosti míče od soupeřovi či vlastní branky. Strategie jsou celkem čtyři: výkop, útok, obrana a neutrální strategie. Těmi to strategiemi se pak řídí jednotliví hráči ovládaní umělou inteligencí.

3.6.3 Komponenty entit umělé inteligence

Základní komponentou umělé inteligence je třída *AIComponent*. Komponenta inicializuje některé předem známé vlastnosti a služby, které jsou často používány dalšími třídami, jež mají na třídu *AIComponent referenci.* Komponenta též inicializuje „mozek“, třídu *BrainGoal*, která bude popsána níže a je základem rozhodování jednotlivých entit. A nastavuje pole, jejichž hodnoty modifikují uvažování entity. Metoda *Update* aktualizuje logiku třídy *BrainGoal*. Volá též virtuální metodu *ManipulateTweakers*, kterou implementují její potomci. V ní pak podle globální strategie manipulují s konstantami ovlivňujícími rozhodování entity. Činí tak podle své role – útočník, obránce agresor nebo sólista (viz kapitola 2.7.4).

3.6.4 Jmenný prostor GoalBehaviour

Ve jmenném prostoru *GoalBehaviour* se nachází třídy, které implementují cíly řízeného agenta z kapitoly 2.7.2. V následujících odstavcích budou popsány základní abstraktní třídy tohoto modelu a jejich příklady.

Abstraktní třída *Goal* reprezentuje obecný cíl agenta. Tři abstraktní metody Activate, *Process a Terminate* představují průběh vykonávání cíle – započetí, vykonávání a ukončení výkonu cíle. Ačkoliv je tato třída kořenem této hierarchie (vyjma třídy *Object*), její přímá implementace značí atomický cíl. Ty jsou obsaženy v jmenném prostoru *AtomicGoals*. Atomické cíle představují například třídy *ApproachTargetEntityGoal*, *CastFireballGoal* nebo *TakeAimAtEntityGoal*. Pro reálného člověka by představovaly následující cíle: přiblížit se k cílové entitě, vykouzlit ohnivou kouli, zamířit na cílovou entity. Každý cíl je v jednom ze čtyř stavů:

- cíl je neaktivní – nevykonává se
- cíl je aktivní – cíl se právě vykonává, ale ještě nebyl dokončen
- cíl byl úspěšně dokončen
- cíl selhal – cíl se nepodařilo vykonat

Potomkem třídy *Goal* je, též abstraktní, třída *CompositeGoal*. Její implementace je založená na generickém kontejneru *LinkedList<Goal>*, který představuje oboustranný spojový seznam podcílů, na který je nahlíženo ve většině případů jako na frontu, do které je ale nutné mít možnost rychle přidávat jak na konec, tak na začátek. Kromě tří podstatných metod zděděných po třídě *Goal* obsahuje metody pro jednoduché vkládaní a odebírání cílů a pro zpracování vnitřních cílů. To probíhá tak, že se z počátku fronty odstraní všechny úspěšně splněné úkoly a pokud ještě nějaký úkol zbývá, začne se vykonávat. Stav složeného cíle se pak určuje podle počtu vnitřních cílů a jejich stavu.

Modul umělé inteligence implementuje čtyři složené cíle. Tři z nich odpovídají standardním, již několikrát zmíněným strategiím – útok, obrana a agrese (napadání) – a

skládají se z jednotlivých atomických cílů. Posledním, speciálním cílem, je třída *BrainGoal*, která je popsána v následující podkapitole.

3.6.5 Třídy BrainGoal a GoalEvaluator

Třída *BrainGoal* je speciální případ složeného cíle, který je zároveň arbitrem zbylých tří složených cílů. Obsahuje seznam objektů třídy *GoalEvaluator*, jejichž implementace kopíruje strukturu implementace složených cílů. Úkolem těchto tříd je ohodnotit hlavní strategie (cíle) entity a nabídnout třídě *BrainGoal* hodnoty, podle kterých se může rozhodnout pro nejlepší momentální strategii. Metoda *EvaluateAndClamp* vrací tuto hodnotu v rozmezí od nuly do jedné. Potomci třídy *GoalEvaluator* implementují abstraktní metodu *Evaulate*, ta ohodnocuje cíl, se kterým je evaluátor spjatý, a metodu *GetNewGoal*, která pak tento cíl vrací v případě, že byl vyhodnocen jako nejvhodnější.

3.7 Projekt „Guid generator“

Jednoduchá aplikace s grafickým rozhraním pro generování GUID [15] s možností generovat další identifikátory a snadným kopírováním do schránky.

4 Závěr

Čtvrtá kapitola obsahuje celkové zhodnocení celé práce. V jednotlivých kapitolách je uvedeno, čeho se v práci podařilo dosáhnout a do jaké míry a jaká by se dala udělat případná zlepšení.

4.1 Dosažené cíle

Na počátku práce byly vytyčeny tři hlavní cíle, které se z velké části podařilo splnit. Následující tři podkapitoly nesou jména těchto cílů a detailněji je rozebírají.

4.1.1 Vytvořit originální, akční, rychlou počítačovou hru

V dnešním světě není jednoduché vytvořit originální dílo, navíc originalita je do jisté míry subjektivní vjem. Hra je založena na kombinaci zkušeností z jiných her a fantazie, což tvoří pomyslný limit. Často se originalita posuzuje podle vizuální stránky. Zde je nutno znovu podotknout, že grafika hry je postavena na posbíraných částech, které byly zdarma dostupné. Na druhou stranu kombinace zbylých faktorů, zejména pohledu shora, prostředí, ve kterém se odehrává a dalších drobných prvků tvoří hru přinejmenším netradiční.

Akčnost hry pak vystihuje již její charakter a rychlost je opět subjektivní vjem. Ve hře ale mohou nastávat rychlé zvraty , na které je bezpochyby nutno reagovat rychle. Rychlost by se dala podpořit zlepšením ovládání pro hráče. O něm více v kapitole 4.2.

4.1.2 Vytvořit herní engine tak, aby podporoval modularitu obsahu

Do hry je momentálně možné přidávat knihovny s kouzly, které mohou výrazně měnit ráz hry a ovlivnit způsob uvažování hráčů. Součástí hry je základní knihovna obsahující pět kouzel, která ukazuje jakým způsobem je možné takové knihovny vytvářet.

4.1.3 Vytvořit systém pro vkládání modulů s umělou inteligencí

Hra umožňuje přiřazovat moduly umělé inteligence k jednotlivým týmům. Vždy dochází ke střetu dvou týmů o počtu jednoho až tří členů, přičemž člověk může ovládat libovolnou předem zvolenou postavu. Při hře dvou hráčů mohou oba hráči ovládat jednu z postav na hřišti, samozřejmě ale každý jinou. Hra tedy podporuje celou škálu variant, ve kterých mohou hrát hráči spolu či proti sobě a to i po boku umělé inteligence. Lze také vybrat pouze moduly umělé inteligence a nechat je hrát proti sobě.

4.2 Možná zlepšení a rozšíření

Kapitola pojednává o možných zlepšeních stávajících (nejen herních) mechanismů a o možných rozšířeních.

4.2.1 Zlepšení ovládání

Pro začínající i pro pokročilejší hráče je ovládání hry poměrně náročné a do budoucna by bylo dobré ho zlepšit. Nabízejí se dvě možnosti – vyladění konstant ve fyzikálním světě a nebo změna ovládacích prvků. V rámci zachování „fair play“ by ale oba hráči měli mít přístup ke stejným ovládacím prvkům, tak aby nebyl jeden nad druhým zvýhodněn.

4.2.2 Přechod na fyzikální knihovnu Farseer

Zejména některé odrazy míče jsou při hře nečekané. Přechodem na knihovnu Farseer, která je v aktivním vývoji by se mohli některé chyby fyzikálního světa odstranit. Zároveň s tím by šlo celý fyzikální engine zapouzdřit, tak, jak je popsáno v kapitole 2.6.5. Systém by pak mohl sloužit i pro testování jednotlivých fyzikálních knihoven.

4.2.3 Lépe zabezpečit načítání externích modulů

Pro naprostou publikaci by bylo třeba zajistit bezpečnost kódu v externích modulech použitím složitější technologií, které vymezují určité hranice pro načtené knihovny, jak je popsáno v kapitole 2.8 a v kapitole 3.3.8.

4.2.4 Vytvoření systému popisu kouzel

Vytvoření systému popisu kouzel by byl zajímavý a rozsáhlý projekt, který by modulům umělé inteligence dával nový rozměr, možnost jak vybírat kouzla bez závislosti na jednotlivých knihovnách, tak jak je popsáno v kapitole 2.8.1.

4.2.5 Rozšíření obsahu hry

Rozšíření hry editorem map a vytvořením více herních prostředí by podpořilo hratelnost a i zajímavost hry. Jednotlivé mapy by též mohly určovat pravidla hry a tak by se na některých hrál „klasický Warlock Football“ a jiné by mohly být jen o tom, který tým zabije víckrát protivníka v daném čase.

Mapy lze také rozšířit o další prvky – překážky, interaktivní prvky jako dveře nebo nášlapné bonusy či pasti, tak jak je poznamenáno v původním dokumentu popisujícím design hry, který je dostupný na přiloženém disku (viz Příloha, kapitola 5.1).

Původní dokument také pojednává o nastavení atributů jednotlivých hráčů, které by se promítly do síly jednotlivých typů kouzel. Dobrý příklad podobného systému je ve hře „Leagues of Legends“, tzv. „summoner system“ [20].

4.2.6 Rozšíření o podporu síťového hraní

Pro komerční a kompetitivní účely by bylo nezbytně nutné vytvořit podporu hraní online a s tím i spojené registrace hráčů, týmů, vedení statistik a žebříčky.

4.2.7 Zlepšení prostředí

Hodně i nezávislých her se drží vysoko v žebříčku oblíbenosti zejména díky dobré grafice, pěkným efektům a příjemné hudbě[1]. Obohacením obsahu hry v tomto směru by také výrazně zlepšilo celkový dojem.

5 Příloha

5.1 Obsah disku

Disk obsahuje následující soubory a adresáře:

- Instalace – adresář s balíčky potřebnými ke spuštění hry Warlock Football
- Warlock Football – adresář se zdrojovými kódy (řešení Visual Stuida 2008)
- bachelor_thesis.pdf – text bakalářské práce
- ctime.html – pokyny pro uživatele
- design_document.pdf – design dokument ke hře Warlock Football (pouze v anglické verzi)
- Warlock Football.exe – zabalené binární soubory, po instalaci knihoven z adresáře Instalace lze poklepáním rozbalit do libovolné složky na počítači

5.2 Uživatelská dokumentace

Po nainstalování hry pomocí pokynů na přiloženém disku a jejím spuštění souborem „Warlock Footbal.exe“ se objeví úvodní obrazovka se základním menu. Ovládání menu je následující:

- pohyb v menu – šipka nahoru a dolu
- výběr – tlačítko Enter

[1] Ve smyslu „trefné k prostředí hry“.

Výběrem položek uživatel prochází strukturu menu a na položkách výběru z možností rotuje jednotlivými možnostmi.

5.2.1 Start nové hry

Po zapnutí hry se výběrem položky „Start new match“ dostane uživatel do menu nastavení hry. Zde má následující možnosti:

- vybrat počet hráčů v týmu (warlocks per team) – vždy hrají proti sobě týmy se stejným počtem hráčů
- zvolit si, jestli bude hrát za fialového hráče nebo za žlutého (violet human player, yellow human player) – hráči se kromě barvy liší také ovládáním
- zvolit modul umělé inteligence pro modrý a zelený tým (blue team ai, green team ai)

Jednotlivé položky v menu lze rotovat tlačítkem Enter. Počet hráčů v týmu je jeden až tři. Hráč se volí tak, že si člověk vybere místo které jednotky ve hře nastoupí. Modul umělé inteligence se volí výběrem jeho jména. Dvě pomlčky (--) značí, že položka nemá zvolenou hodnotu.

Při zvolení fialového (violet) nebo žlutého (yellow) hráče po stisku tlačítka „Start game“ vyskočí obrazovka s výběrem kouzel vybraného hráče. Vlevo je seznam jednotlivých kouzel, vpravo pak jejich popis spolu s popisem ovládání hráče. Po vybrání čtyř kouzel se aktivuje tlačítko „Confirm“, po jehož stisknutí se případně objeví nastavení pro druhého hráče nebo započne hra.

Ovládání hráčů ve hře popisuje následující tabulka:

	Fialový hráč (Violet player)	Žlutý hráč (Yellow player)
Dopředu	Šipka nahoru	E
Dozadu	Šipka dolu	D
Otočit vlevo	Šipka vlevo	S
Otočit vpravo	Šipka vpravo	F

Použít kouzlo	L	Q
Vybrat další kouzlo	K	A

Při hraní hry lze stiskem tlačítka Escape a nebo přepnutí okna zapnout pauzu. Objeví jsem herní menu. Stiskem „Resume“ se lze vrátit do hry, „Quit to main menu“ se pak vrátí do počátečního menu a stiskem „Quit game“ je hra ukončena.

5.3 Pokyny pro vývojáře externích knihoven

Pro vývoj externích knihoven je vhodné podívat se do vzorových knihoven v implementaci – projekt „Warlock Football Basic Spells“ (kapitola 3.5) a „Warlock Football Default AI“ (kapitola 3.6).

V první řadě je podstatné přidat referenci knihovny „Warlock Football Common“ (kapitola 3.3) a potom pro knihovnu AI implementovat potomka třídy *AIDescriptor*. Hotovou knihovnu je nutné zkopírovat do adresáře „AI“ v programu WF. To samé platí o kouzlech. Jen je nutné implementovat třídy *SpellDescriptor* a *SpellConjuration*. Knihovnu kouzel je pak nutné zkopírovat do adresáře „Spells“ a případné textury zkompilované do adresáře „Content“.

6 Užité zdroje

1. Combat Pillows (původní název Bulánci). [Online] http://www.combatpillows.com/.

2. Benny Kramek. Machine Ball. [Online] http://benny.kramekweb.com/machineball/.

3. Warlock Brawl. [Online] http://www.warlockbrawl.com/.

4. Defense of the Ancients. [Online] http://www.playdota.com/.

5. Microsoft, MSDN Library. XNA Game Studio 3.1. [Online] http://msdn.microsoft.com/en-us/library/bb200104%28v=XNAGameStudio.31%29.aspx.

6. Microsoft, MSDN Library. XNA Framework Class Library Reference. [Online] http://msdn.microsoft.com/en-us/library/bb203940%28v=XNAGameStudio.31%29.aspx.

7. **Lobão, Alexandre, a další.** *Beginning XNA 3.0 Game Programming: From Novice to Professiona.* Berkeley : Apress, 2009.

8. Unity Technologies. Unity. [Online] http://unity3d.com/unity/.

9. Box2D Physics Engine. [Online] http://www.box2d.org.

10. Google Code. Box2dx. [Online] http://code.google.com/p/box2dx.

11. Codeplex. Farseer physics. [Online] http://farseerphysics.codeplex.com.

12. **Buckland, Mat.** *Programming Game AI by Example.* Plano : Wordware Publishing Inc., 2004.

13. Microsoft, MSDN Library. Reflection. [Online] http://msdn.microsoft.com/en-us/library/f7ykdhsy.aspx.

14. Microsoft, MSDN Library. .NET Framework Remoting. [Online] http://msdn.microsoft.com/en-us/library/kwdt6w2k(v=VS.85).aspx.

15. Wikipedia. Globally unique identifier. [Online] http://en.wikipedia.org/wiki/Globally_unique_identifier.

16. *Microsoft, MSDN Library. TopDownShooter: Creating the Prototype.* [Online] http://msdn.microsoft.com/en-us/library/dd282447(v=XNAGameStudio.31).aspx#GSM.

17. Lost garden. [Online] http://www.lostgarden.com/2006/07/more-free-game-graphics.html.

18. Tigsource, Forums. [Online] http://forums.tigsource.com/index.php?topic=8819.0.

19. Tigsource. Forums. [Online] http://forums.tigsource.com/index.php?topic=8792.0.

20. Leagues of Legends. Summoner information. [Online] http://eu.leagueoflegends.com/learn/gameplay/summoner-information.

21. Wikipedia. Duck typing. [Online] http://en.wikipedia.org/wiki/Duck_typing.

22. Microsoft, MSDN Library. Windows Presentation Foundation. [Online] http://msdn.microsoft.com/en-us/library/ms754130.aspx.