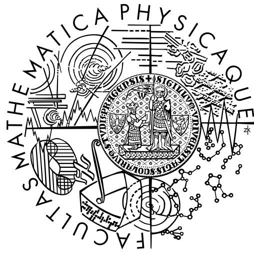


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Jan Kočur

Umělý hráč pro Dotu 2

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: Mgr. Jakub Gemrot, Ph.D.
Studijní program: Informatika
Studijní obor: Softwarové a datové inženýrství

Praha 2018

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) 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 má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V .... dne .... Podpis autora

Rád bych poděkoval vedoucímu práce Jakubovi Gemrotovi za možnost zpracovat zajímavé téma a za mnoho cenných rad, které mi usnadnily vývoj a psaní práce. Zároveň bych rád poděkoval svým rodičům za všechnu jejich podporu.

Název práce: Umělý hráč pro Dotu 2

Autor: Jan Kočur

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: Mgr. Jakub Gemrot, Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: Abstrakt: Dota 2 patří mezi nejoblíbenější strategické počítačové hry žánru Multiplayer Online Battle Arena (MOBA), který je typický svými nároky na týmovou spolupráci a taktické myšlení hráčů. To činí žánr zajímavou platformou pro výzkum umělé inteligence (AI), který se často zabývá tvorbou agentů hrajících hru. Hry žánru neposkytují nástroje umožňující vývoj složitějších agentů. Prvním cílem této práce bylo vytvořit rozhraní, které umožní vývoj agentů pro Dotu 2 v Javě. Druhým cílem bylo vytvořit agenta hrajícího Dotu 2, který prokáže funkčnost rozhraní. Práci jsme rozdělili do dvou částí. Nejdříve jsme vytvořili nároky na rozhraní a popsali jsme jeho architekturu. V druhé části práce jsme analyzovali Dotu 2 z perspektivy AI a implementovali vlastní agenty nad vytvořeným rozhraním. Naši agenti byli schopni prokázat funkčnost rozhraní a zvládnutí základních herních dovedností. Výsledné rozhraní může být použito pro další výzkum umělé inteligence.

Klíčová slova: umělá inteligence, MOBA, Dota 2, mapy vlivu, teorie užitku

Title: Artificial player for Dota 2

Author: Jan Kočur

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Ph.D., Department of Software and Computer Science Education

Abstract: Dota 2 is one of the most popular strategic computer games of the Multiplayer Online Battle Arena (MOBA) genre. MOBA games are based on teamwork and tactical thinking. That makes them an interesting platform for the artificial intelligence (AI) research, that aims to create artificial agents capable of playing the game. However, there does not exist any framework, that would allow the development of complex agents. First, we developed a framework that allows the creation of agents for Dota 2 in Java. Second, we implemented an agent above the framework, that is capable of playing the game. We have divided the work into two parts. First, we have analyzed requirements for our framework and described its architecture. Second, we have analyzed Dota 2 from the AI perspective and implemented agents above our framework. Our agents were capable of playing the game. The framework can be used for further research.

Keywords: artificial intelligence, MOBA, Dota 2, influence maps, utility theory

Obsah

Úvod **3**

1 Dota 2 **5**
1.1 Základy hry . . . 5
1.2 Herní prostředí . . . 5
1.3 Herní mapa . . . 6
1.4 Dynamické objekty . . . 9
1.5 Statické objekty . . . 10
1.6 Polo-dynamické objekty . . . 11
1.7 Shrnutí . . . 12

2 Návrh řešení **13**
2.1 Požadavky agentů . . . 13
2.2 Existující nástroje pro vývoj AI . . . 14
2.3 Dota 2 AI Framework . . . 14
2.4 Nedostatky . . . 16
2.5 Návrh řešení . . . 17

3 Implementace rozhraní **23**
3.1 Addon . . . 23
3.2 Architektura rozhraní . . . 24
3.3 Grafické prostředí . . . 30
3.4 Shrnutí . . . 32

4 Analýza Doty 2 z pohledu AI **33**
4.1 Role hrdinů . . . 33
4.2 Fáze hry . . . 34
4.3 Přístupy k AI . . . 36
4.4 Související práce . . . 38

5 Návrh agenta **40**
5.1 Návrh implementace . . . 40
5.2 Mapy vlivu . . . 41
5.3 Teorie užitku . . . 44

6 Implementace agenta **47**
6.1 Architektura agentů . . . 47
6.2 Mapy vlivu . . . 47
6.3 Teorie užitku . . . 52
6.4 Shrnutí . . . 57

7 Experimenty a výsledky **58**
7.1 Farmení . . . 58
7.2 Vyhrání hry . . . 59

Závěr **60**

Seznam použité literatury **62**

Seznam obrázků **64**

Seznam tabulek **65**

Přílohy **66**

Úvod

Dota 2 je jedna z nejpopulárnějších strategických her žánru MOBA, která byla vyvinuta společností Valve Corporation [1]. Dota 2 patří mezi real-time strategické hry. Hraje se v dynamickém a částečně-pozorovatelném prostředí.

Ve hře spolu soupeří dva týmy hráčů. Každý tým má jednu základnu. Základny jsou umístěny v jiných částech herní mapy a jsou bráněny věžemi. Ze základen vychází jednotky, které se pohybují směrem k nepřátelské základně. Každý hráč ovládá jednoho unikátního hrdinu se specifickými schopnostmi. Cílem hráče je zničit pevnost, která je umístěna v nepřátelské základně.

Pro hráče je nejdůležitější spolupráce s ostatními hrdiny v týmu (viz [15, 16]) a taktické myšlení (viz [11]). Obojí je výrazně ztíženo složitou herní mapou a její dynamičností. Právě tyto nároky činí žánr zajímavým pro výzkum umělé inteligence a vývoj inteligentních agentů.

Studiem a vývojem inteligentních agentů pro žánr MOBA se již zabývalo několik prací (viz [2, 3]). Inteligence většiny agentů ovšem nebyla rovnocenným soupeřem lidských oponentů (viz [4]). Často byl jako příčina uváděn nedostatek výpočetního výkonu nebo nepříznivé prostředí pro vývoj agentů. Špatné nástroje zčásti mohou za to, že je žánr MOBA stále relativně neprozkoumán akademiky.

Pro Dotu 2 existuje sada nástrojů [5], které umožňují tvorbu rozšíření hry pomocí skriptování v jazyce Lua. Pomocí skriptů může uživatel upravovat chování hry a některých herních entit. Skripty hra vykonává po svém spuštění.

Nástroje pro Dotu 2 umožňují snadnou tvorbu jednoduchých agentů. Zároveň s sebou přináší jistá negativa. Prvním negativem je, že skripty jsou vykonávány pouze v jednom procesu. Proto náročnější agenti mohou hru učinit nehratelnou. Další přítěží může být Lua, pro kterou neexistuje tolik knihoven a nástrojů, jako pro běžné programovací jazyky.

Pro překonání obou negativ vytvořil Tobias Mahlmann Dota 2 AI Framework [6]. Framework se skládá z addonu (Lua) a serverové části (Java), které spolu komunikují pomocí protokolu HTTP. Na straně serveru je implementován agent, který používá informace odesílané addonem. Agenti tak mohou být napsáni v Javě a fungovat na více vláknech. Agenti mají k dispozici větší výpočetní výkon a nezpomalují hru.

Bohužel i tento framework má řadu nedostatků. Chybí mu reprezentace mapy, herních předmětů a dalších informací, které mohou být pro agenty důležité. Framework také postrádá jakékoliv grafické rozhraní (dále GUI), které by usnadnilo vývoj agentů a jakékoliv nástroje pro jejich testování. Pro komplexnější agenty je framework nepoužitelný.

Cíle práce

Tato práce má dva hlavní cíle:

1. **Vytvořit rozhraní** ve vyšším programovacím jazyce založené na Dota 2 AI Frameworku. Cílem úprav by mělo být přidat chybějící funkcionalitu a opravit všechny nedostatky, které brání vývoji smysluplných agentů. Rozhraní

[1] http://www.dota2.com

by mělo agentům poskytnout dostatečné informace o světě, který je obklopuje. Mělo by také nabízet GUI, které usnadní jejich testování a případnou konfiguraci.

2. **Navrhnout umělého agenta** v daném rozhraní. Agent by měl předvést vlastnosti rozhraní a jevit známky inteligentního chování. Měl by být schopen prokázat, že zvládá základní dovednosti potřebné k vyhraní hry.

Struktura práce

Práci jsme rozdělili na dvě části. V první části se zabýváme vytvářením rozhraní, které umožňuje tvorbu umělých agentů. Ve druhé části nad tímto rozhraním vyvíjíme umělého agenta.

V kapitole 1 popisujeme prostředí Doty 2. Pokračujeme kapitolou 2, kde se zabýváme analýzou požadavků na naše rozhraní. Kapitola 3 popisuje implementaci rozhraní. Poté přecházíme k umělé inteligenci. Začínáme kapitolou 4, ve které analyzujeme Dotu 2 z pohledu umělé inteligence. V následující kapitole 5 navrhujeme vlastního agenta. V kapitole 6 popisujeme jeho implementaci. V kapitole 7 experimentujeme s implementovaným agentem a v poslední kapitole shrnujeme dosažené výsledky.

1. Dota 2

V této kapitole popisujeme herní prostředí a svět Doty 2. Začínáme popisem herního prostředí a toho, jak hra vypadá z hráčova pohledu. Dále popisujeme herní mapu a základní vlastnosti hry. Končíme popisem herních objektů, které dělíme na dynamické, polo-dynamické a statické. Rozdělení objektů do skupin jsme přejali z práce [3], protože dobře charakterizuje jejich vztah k hráči.

Dynamické objekty jsou pohyblivé a v čase se jejich vztah ke hráči mění. Statické objekty se nehýbou a v průběhu hry nemění svůj vztah k hráči. Polodynamické objekty se nehýbou, ale jejich vztah k hráči se mění. Příkladem polodynamického objektu je věž. Věž se nehýbe, ale k hráči má jiný vztah podle toho, jestli na něj útočí.

Úkolem této kapitoly je především uvést čtenáře do prostředí hry a vytvořit základ, ze kterého čerpá zbytek práce. Dota 2 je velmi složitá, proto jsme popis pro potřeby této práce zjednodušili. Detailnější informace mohou být nalezeny na stránce Dota 2 Wiki [7]. Dotu nerozebíráme z pohledu umělé inteligence, tím se zabýváme v kapitole 4.

1.1 Základy hry

Žánr MOBA je subžánrem Real Time Strategy (RTS) her. Pro své akční prvky bývá žánr MOBA často označován jako akční RTS. Na rozdíl od žánru RTS hráči nestaví budovy a ovládají pouze jednu jednotku. Dota 2 byla vyvinuta jako následovník populárního módu Defense of the Ancients (DotA) [8] pro hru Warcraft 3: Reign of Chaos.

Hráč ovládá jednoho unikátního hrdinu, který je součástí jednoho ze dvou týmů. Týmy jsou složené až z pěti hrdinů. Cílem každého hráče je zničit nepřátelskou pevnost (tzv. prastarého). Pevnost se nachází v centru každé základny. Základny bývají bráněny jednotkami, které z nich vychází a strategickými budovami. Jednotky chodí po každé ze tří cest (tzv. linky), které vedou mezi oběma základnami. Jednotky brání vlastní bázi a na nepřátelskou útočí.

1.2 Herní prostředí

Herní prostředí umožňuje hráčům ovládat své hrdiny a poskytuje jim náhled na část herní mapy. Prostředí poskytuje informace o stavu týmu a jiné informace (viz Obrázek 1.1). Součástí herního prostředí je například tzv. minimapa. Minimapa zobrazuje pozice všech objektů, které jsou viditelné pro hráčův tým.

Hráči interagují se hrou pomocí myši a klávesových zkratek. Pomocí myši se hráči rozhlíží po herním prostředí (posouvají náhledem) a ovládají hrdiny. Například kliknutím na herní mapu mohou hráči pohybovat svým hrdinou. Klávesové zkratky umožňují rychlé použití kouzel a předmětů, které by byly špatně ovladatelné pouze myší.

Obrázek 1.1: Herní prostředí
Herní prostředí. Červeně je zvýrazněna minimapa, zeleně ovládací panel hrdiny.

1.3 Herní mapa

Viditelnost

Hráči mají informace pouze o té části herní mapy, která je pro jejich hrdinu nebo tým viditelná (označováno jako tzv. "fog of war"). Každé hráčovo rozhodnutí je založeno na dostupných informacích, což dělá z konceptu viditelnosti jednu z nejdůležitějších mechanik Doty 2.

Obrázek 1.2: Viditelnost
Obrázek zachycuje hrdinovu viditelnost v noci. Lze si všimnout omezené viditelnosti a stromů, skrze které hrdina nevidí.

Každá herní entita kolem sebe vidí na určitou vzdálenost, která se mění dnem a nocí (viz Obrázek 1.2). Den a noc mají vliv také na vlastnosti některých hrdinů a předmětů. Existuje například hrdina, který se v noci pohybuje rychleji nebo předmět, který se v noci nedá použít.

Entity nevidí na místa blokované neprůhlednými objekty. Neprůhledné jsou například stromy, které jsou součástí lesů a tvoří velkou část mapy. Entity vidí

na místa, která jsou v podobné nebo nižší výšce. To je důležité například v okolí řeky, která má břehy výrazně výše než koryto.

Viditelnost všech entit je sdílena v rámci týmů. Hráč může teoreticky vidět všechno, co vidí jeho tým. Typicky je ovšem limitován náhledem na herní mapu, který zobrazuje pouze její část.

Dota 2 dále aktivně využívá koncept viditelnosti v herních mechanikách. Někteří hrdinové mohou být neviditelní, což může pro hráče představovat velký problém. Také existují předměty (tzv. wardy), které mohou poskytnout viditelnost na nějaké místo na mapě. Existuje také verze, která odhaluje neviditelné jednotky. Použití ward a jejich správné umístění tak může být pro tým klíčové.

Terén

Terén mapy má pro hráče především strategický význam. Prvním faktorem, kterým terén ovlivňuje hru, je výška. Výška ovlivňuje hráčův rozhled — hráči nevidí na místa, která jsou výrazně výše než jejich hrdina. Střelečtí hrdinové mají menší šanci, že se trefí do vyvýšené jednotky. Druhým faktorem jsou neprůchozí místa, která limitují hráčův pohyb.

Oba faktory mohou ovlivnit výsledky soubojů a strategické vlastnosti některých míst na mapě. Výška i neprůchodný terén mohou být klíčovým faktorem při útěku ze soubojů nebo při jejich iniciaci.

Linky

Linky spojují báze obou týmů. Můžeme je rozdělit na horní, dolní a prostřední linku (viz Obrázek 1.3). Každý tým má bezpečnou (safe lane) a nebezpečnou (hard lane) linku. Bezpečná je buď dolní nebo horní linka, jejíž delší část je na straně daného týmu. Linka je bezpečná, protože se hráči na začátku pohybují na své straně mapy. U nebezpečné linky je to naopak. Prostřední linka je stejně dlouhá na obou stranách.

Jednotlivé linky a jejich vlastnosti ovlivňují především způsob, kterým na nich hrdinové hrají. Často se liší počtem hráčů hrajících na lince, který je závislý na bezpečnosti linky. Na bezpečné lince často hraje jeden hráč, kdežto na nebezpečné bývají dva. Na prostřední linku chodí hrdinové, kteří jsou schopni rychle přejít na jinou z linek a pomoci některému ze spojeneckých hrdinů.

Lesy

Další podstatnou částí mapy jsou lesy. Lesy vyplňují prostor mezi linkami (viz Obrázek 1.3) a nejsou přímo ovládány žádnou z frakcí. Hlavní implikací je, že hráči do lesa nevidí a mají o jeho stavu pouze částečné informace. Tím se z něj stává strategické místo, které bývá využíváno při útěku nebo iniciaci útoků.

Dále jsou v lese umístěny tábory neutrálních jednotek, které pro hráče představují alternativní zdroj zlata. Tábory se liší silou neutrálních jednotek a jejich počtem. Les také obsahuje runy, které hráči poskytnou zlato, a fontány, které jej vyléčí. V každém lese jsou umístěny dva obchody. Jednomu obchodu se říká tajný a druhému vedlejší.

Obrázek 1.3: Minimapa s linkami
Linky a báze jsou zvýrazněny. Mimo linky můžeme vidět lesy a řeku. Zdroj původního obrázku: dota2.gamepedia.com/File:Minimap_7.07.png

Řeka

Hlavní dominantou herní mapy je řeka, která mapu dělí na dvě části. Řeka vede přibližně mezi těmi rohy mapy, které neobsahují báze (viz Obrázek 1.3). Představuje nejníže položenou oblast ve hře, což z ní dělá jedno z nejvíce nebezpečných míst na mapě. Nebezpečí představují především vysoké břehy, které hráči v řece omezují výhled a staví jej do nevýhody proti jednotkám na březích.

Řeka obsahuje dvě runy, které hrdinovi po aktivaci poskytnou určitý bonus a Barona. Baron je velmi silná jednotka, kterou jsou hráči schopni porazit pouze v pozdější fázi hry. Po jeho zabití jeden hrdina obdrží speciální předmět, který jej po jeho další smrti oživí.

Řeka bývá nejčastěji využívána pro rychlé přecházení mezi linkami a lesy. V pozdější fázi hry řeka tvoří místo, na kterém probíhá část soubojů například v okolí Barona.

1.4 Dynamické objekty

Hrdinové

Na začátku hry si každý hráč musí vybrat jednoho hrdinu, se kterým bude hrát do konce hry. Hrdinové se liší svými kouzly, základním nastavením atributů, primárním atributem a ve svým vzhledem. Dále má každý hrdina úroveň, která je dána zkušenostmi, které hrdina získal ničením budov a zabíjením jednotek nebo hrdinů.

Kouzla mohou mít stálý efekt (pasivní) nebo je hrdina může použít (aktivní). Kouzla se používají na místo na mapě nebo na herní entity. Za použití většiny kouzel hráč platí tzv. manou. Její nedostatek způsobí, že hrdina nemůže kouzla používat. Mana se podobně jako hrdinovo zdraví časem obnovuje.

Každý hrdina má tři základní atributy, které ovlivňují jeho vlastnosti. Sílu, která ovlivňuje regeneraci životů, jejich počet a hrdinovu odolnost. Obratnost, která každému hrdinovi zlepšuje obranu, rychlost pohybu a útoku. A inteligenci, která zvyšuje poškození udělované kouzly, množství many a její regeneraci.

Jeden ze tří hrdinových atributů je označen jako primární. Vylepšování primárního atributu je o čtvrtinu efektivnější oproti neprimárním atributům a navíc zvyšuje poškození, které hrdina uděluje. Primární atributy ovlivňují herní styl daného hrdiny. Například pro hrdiny s primárním atributem inteligencí je nejvýhodnější používat kouzla, protože kouzla udělují větší poškození a hrdinové mají více many.

K vylepšování kouzel a atributů slouží hrdinovo zlato a zkušenosti. Zlato a zkušenosti se získávají především zabíjením nepřátelských hrdinů, budov a jednotek. Zkušenosti může hrdina použít k vylepšování kouzel. Zlato může být použito k nákupu zbraní, zbrojí a spotřebního zboží.

Předměty zlepšují hrdinovy atributy a mohou mít speciální schopnosti. Většinou jsou složeny z několika částí, které může hrdina nakupovat zvlášť. Spotřební zboží jsou například lektvary, které mohou být spotřebovány na obnovení hrdinova zdraví nebo many.

Pro uložení předmětů slouží hrdinův inventář a batoh. Předměty jsou primárně uloženy v hrdinově inventáři. Pokud se do něj nevejdou, ukládají se do batohu. Hrdina v inventáři unese až šest předmětů a v batohu další tři. Předměty, které hrdina nemá ve svém inventáři nemohou být použity a nemají žádný efekt.

Na začátku hry jsou hrdinové umístěni do svých bází. Pokud některý z hrdinů zemře, znovu se po krátkém intervalu objeví v bázi. Hráč je za smrt svého hrdiny potrestán ztrátou zlata. Délka smrti a množství ztraceného zlata závisí na hrdinově úrovni.

Jednotky

Jednotky mají podobné vlastnosti jako hrdinové. Každá jednotka má přiřazeny jiné atributy, které udávají její sílu. Atributy určují maximální a minimální poškození, které může jednotka udělit. Síla jednotky určuje, kolik zlata a zkušeností může po jejím zabití hráč obdržet. Jednotky také mohou bojovat z větší vzdálenosti (střelecké) nebo zblízka (pěšáci).

Prvním důležitým typem jednotky jsou neutrální jednotky. Ty typicky najdeme v jednom z táborů, které jsou umístěny v lesích. Každý tábor obsahuje několik jednotek, jejichž síla určuje obtížnost daného tábora. K neutrálním jednotkám patří také Baron, což je nejsilnější neutrální jednotka.

Obrázek 1.4: Obrázek ze hry
Zvýrazněny jsou jednotlivé typy jednotek a hrdina. Barva rozlišuje tým.

Druhým typem jsou jednotky, které periodicky vychází z obou bází po každé z linek. Tyto jednotky jsou tří druhů — střelec, pěšák a katapult (viz Obrázek 1.4). Katapult je nejodolnější jednotka, která je typická tím, že uděluje velké poškození věžím. Následují střelec a pěšák. Zatímco střelec a pěšák jsou součástí každé vlny jednotek, katapulty opouští bázi méně často.

Pro hráče mají jednotky na linkách důležitý význam. Jednotky se snaží chránit věže a hrdiny svého týmu. Pokud nepřátelský hrdina zaútočí na spojeneckého hrdinu, pak jednotky v okolí začnou na nepřítele automaticky útočit. Přestanou teprve tehdy, když nepřítel přestane útočit nebo odejde dostatečně daleko.

Poslední důležitou jednotkou je kurýr. Kurýr je pro každý tým maximálně jeden. Jeho hlavním úkolem je doručovat hrdinům předměty z hlavního obchodu. Může být použit také pro nákup předmětů v tajném obchodě nebo pro průzkum mapy. Po třetí minutě hry se z něj stává létající jednotka, která ignoruje terén.

1.5 Statické objekty

Budovy

Budovy jsou základním statickým objektem, který se nachází v každé bázi. Existují budovy, které slouží pouze jako výplň báze. Jejich funkce je taková, že zpomalují postup jednotek bází, které útočí vždy na nejbližší budovu. Dále základny obsahují kasárny. Kasárny jsou dvou druhů — pro střelecké a pro pěší jednotky. Kasárny představují místo, ze kterého vychází jednotky směrem k nepřátelské bázi. Zároveň je jejich úkolem vyvažovat sílu nepřátelských jednotek.

Pokud nepřátelský tým kasárny zničí, začnou z jejich báze vycházet silnější jednotky.

V samotném jádru báze se nachází pevnost. Pevnost nemá žádné zvláštní funkce. Po jejím zničení končí hra vítězstvím nepřátelského týmu.

Další budovou je fontána, která je umístěna v nejzazším konci báze (v rohu mapy). Její první funkcí je doplňovat zdraví a manu hrdinům. Druhou je chránit místo, na kterém se hrdinové objevují po své smrti. Fontána vysílá paprsky po nepřátelích, kteří přijdou příliš blízko.

Obchody

V obchodech mohou hrdinové utratit vydělané zlato. Existuje více typů obchodů. Prvním typem obchodu je hlavní obchod. Ten se nachází v každé z bází a slouží potřebám daného týmu. Dalším typem jsou tajné obchody, které jsou umístěny v obou lesích. Posledním typem obchodu jsou vedlejší obchody, které jsou umístěny na obou postranních linkách.

V hlavním obchodě se nachází většina věcí, které mohou hrdinové potřebovat. Tajný obchod prodává pouze unikátní předměty, které nejsou v žádném jiném obchodě. Tajný obchod bývá nutné navštívit při kupování pokročilejších předmětů. Vedlejší obchod nabízí některé ze základních předmětů a může být využit hráči na nejdelší lince k rychlejšímu nákupu.

Abychom mohli předmět koupit, musíme stát u obchodu, který jej prodává. Jedinou výjimkou je hlavní obchod. Hlavní obchod má skladiště, do kterého můžeme nakoupit předměty i z větší vzdálenosti. Ze skladiště si předměty musíme vyzvednout, nebo k jejich vyzvednutí využít kurýra.

1.6 Polo-dynamické objekty

Věže

Věže se nachází na linkách a před oběma pevnostmi uvnitř obou bází. Věže se liší odolností, která je vyšší blíže k pevnosti. Jejich hlavním úkolem je bránit postupu jednotek a hrdinů. Na každé lince stojí právě tři věže, před pevnostmi další dvě.

Věže jako první útočí na jednotku, která se dostane na jejich dostřel. Pokud je jednotek v dostřelu více, vybírají přednostně jednotky útočící na věž. V druhé řadě cílí na jednotky, které napadají spojenecké jednotky.

Věž může být v agresivním nebo pasivním stavu. V pasivním stavu věž neútočí. V agresivním stavu naopak věž útočí na nějakou jednotku. Hrdina může dále rozlišovat stav věže podle toho, jestli útočí na něj nebo na odlišnou jednotku. Hrdina musí na změny stavu reagovat a přizpůsobovat jim své chování.

Svatyně

Svatyně představují polo-dynamické objekty, které slouží k obnově zdraví a many hrdinů. V každém lese se nachází dvě. Hrdina může svatyni aktivovat. Po aktivaci začne svatyně doplňovat zdraví a manu všem jednotkám ve svém okolí.

Po pěti sekundách je svatyně vyčerpána a hráči musí počkat pět minut, než ji budou moci znovu použít.

Runy

Runy jsou herní objekty, které se objevují na několika místech herní mapy. Jejich význam spočívá v tom, že je hráči mohou aktivovat, aby získali bonus ke svým schopnostem. Runy jsou dvou typů. První typ run hráči poskytuje časově omezené schopnosti. To je například neviditelnost nebo zvýšená rychlost. Tyto runy jsou dvě a jsou umístěny v řece, kde se objevují každé dvě minuty.

Dále existují celkem čtyři runy, které jsou po dvou umístěny v každém z lesů. Pokud hráč aktivuje některou z nich, pak celý hráčův tým dostane zlato. Množství zlata je vyšší, čím déle hra trvá. Tyto runy se ve hře objevují každých pět minut počínaje začátkem hry.

1.7 Shrnutí

Pro lepší představu můžeme ještě vyjmenovat vlastnosti herního prostředí ve vztahu k hráči, které jsme přejali z knihy Russela a Norviga [9]:

- **Částečně pozorovatelné**: Hráči nemají přístup ke všem herním informacím. Např. nevidí celou mapu.
- **Více-agentní**: V prostředí se pohybuje více agentů. Interakce mezi agenty jsou kompetitivní a kooperativní.
- **Nedeterministické**: Ve hře existují nedeterministické akce. Např. poškození je definované maximálním a minimálním poškozením, které může jednotka udělit.
- **Sekvenční**: Akce ovlivňují vývoj hry. Hra není rozdělena do více částí.
- **Dynamické**: Prostředí se mění v čase.
- **Spojité**: Prostředí může být vnímáno spojitě. Jednotlivé stavy jednotek se časem plynule mění.
- **Známé**: Hráči hrají pokaždé se stejnými pravidly a ve stejném prostředí.

2. Návrh řešení

V této kapitole analyzujeme požadavky kladené na naše rozhraní. Nejprve definujeme požadavky umělých agentů na funkcionalitu rozhraní. Poté dáváme do souvislosti již existující nástroje pro vývoj umělých agentů v žánru MOBA a rozebíráme Dota 2 AI Framework, který byl vyvinut nad skriptovacím rozhraním Doty 2. Nakonec navrhujeme vlastní rozhraní, které rozšiřuje Dota 2 AI Framework.

2.1 Požadavky agentů

Při vývoji umělé inteligence agentů se snažíme, aby agenti hráli za stejných podmínek jako jejich lidští protivníci. Pokud by měli podmínky odlišné, nemohli bychom jejich inteligenci porovnávat s tou lidskou.

Pro naše rozhraní to znamená, že musí agentovi poskytovat nejlépe všechny možnosti ovládání, které má k dispozici hráč. To je ovšem obtížné, protože hráčova interakce se hrou je diametrálně odlišná od způsobu, jakým by se hrou měl interagovat agent. Snažíme se tedy poskytnout agentům podobné primitiva, které má k dispozici hráč (např. pohni se).

Nesnažíme se o zachycení obrazu hry, kterou by musel agent interpretovat podobně, jako hráč sledující monitor. Chceme, aby měl agent přístup ke skoro stejným informacím, ke kterým má přístup hráč. Zároveň nechceme, aby mohl agent podvádět. To znamená neměl by být schopen dělat věci, které nemůže dělat lidský hráč (ovládat více jednotek ve stejný čas).

Pro potřeby této práce definujeme následující schopnosti lidských hráčů, které by agenti měli být schopni vykonávat:

- **Pohni se.** Hráč může poslat hrdinu na nějaké políčko mapy nebo mu přikázat, aby následoval spřátelenou jednotku.
- **Zaútoč na jednotku.** Reprezentuje hráčův klik myší na nepřátelskou jednotku. Ten způsobí, že hráčův hrdina na jednotku zaútočí.
- **Použij kouzlo.** Hráč může použít kouzlo na nějaké místo nebo jednotku na herní mapě.
- **Vylepši kouzlo.** Pokud má hráč dostatek zkušeností, může si některé ze svých kouzel vylepšit.
- **Použij předmět.** Hráč může použít předmět z inventáře stejně, jako kdyby používal kouzlo.
- **Prodej nebo kup předmět.** Hráč může nakupovat a prodávat předměty z obchodů.
- **Vezmi všechny předměty.** Hráč může přesouvat předměty ze skladiště do svého inventáře, pokud stojí dostatečně blízko.
- **Použij objekt.** Hráč může aktivovat runu nebo svatyni.

- **Ovládej kurýra.** Hráč může dávat kurýrovi příkazy a ovládat jej.
- **Reaguj na příchozí zprávu.** Hráči si mohou posílat textové zprávy.

Všechny výše definované akce jsou pro agenta důležité. Pokud by agent nemohl jednu z akcí vykonat, pak by za lidským hráčem automaticky zaostával. Například nemůžeme očekávat, že bude agent hrát ve srovnání s člověkem dobře, pokud nebude schopen nakupovat předměty. Rozhraní, které cílí na vývoj inteligentních agentů by mělo podporovat co nejvíce z nich.

2.2 Existující nástroje pro vývoj AI

Kdokoliv, kdo chce vyvíjet umělou inteligenci pro nějakou hru, je závislý na nástrojích, které jsou pro hru dostupné. Většina her nevytváří nástroje určené přímo pro vývoj umělé inteligence, ale poskytuje nástroje, které umožňují hru rozšiřovat (vytvářet tzv. addony).

Dota 2 a Heroes of Newerth patří mezi hry žánru MOBA, které poskytují své vlastní nástroje pro tvorbu rozšíření. Součástí obou nástrojů je Application Programming Interface (API) rozhraní, které umožňuje interakci se hrou pomocí skriptování. Ve skriptech je umožněno upravit chování hrdinů, což bylo úspěšně použito v některých z prací citovaných při analýze umělé inteligence (viz [10]).

Například League of Legends neposkytuje žádné vlastní nástroje. Nicméně pro ni vznikl nástroj Bot of Legends, který hráčům umožňuje vytvářet skripty, které ovládají jejich hrdinu. Nástroj není cílen na vývoj umělé inteligence, ale i tak byl úspěšně použit v jedné z později citovaných prací [4].

Nástroje pro Dotu 2 patří mezi nejvíce používané. Doplácí na špatnou dokumentaci, která je ovšem kompenzována velkou komunitou uživatelů. Umělá inteligence se vytváří formou rozšíření psaných za použití Dota 2 Scripting API ve skriptovacím jazyce Lua.

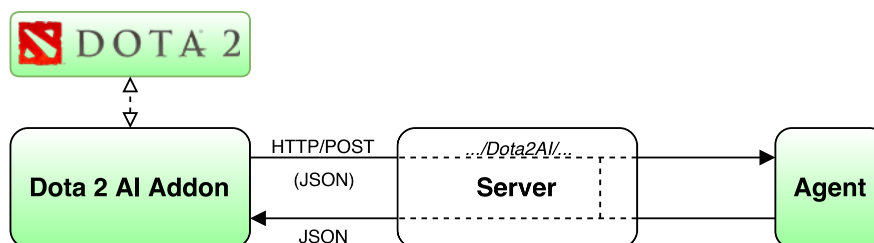
Pro vývoj složitějších agentů je Dota 2 Scripting API nedostačující. Hlavním negativem je, že funguje pouze v jednom procesu. Je teoreticky možné napsat složitější agenty, ale při delších výpočtech na ně musí hra čekat. To hru může učinit nehratelnou. Proto byl nad těmito nástroji vytvořen Dota 2 AI Framework [6].

2.3 Dota 2 AI Framework

Framework se skládá ze tří pomyslných částí — addonu, serverové části a implementace agenta (viz Obrázek 2.1). Tyto části spolu komunikují pomocí protokolu HTTP. Framework umožňuje implementaci právě jednoho agenta, který ve hře ovládá jednoho hrdinu.

Addon je rozšíření Doty 2, které se na straně hry stará o komunikaci se serverem a o vykonání agentových příkazů. Addon řeší, jak zakódovat informace o herním světě, jak je poslat agentovi prostřednictvím serveru a jak obdržet, interpretovat a vykonat agentovy příkazy.

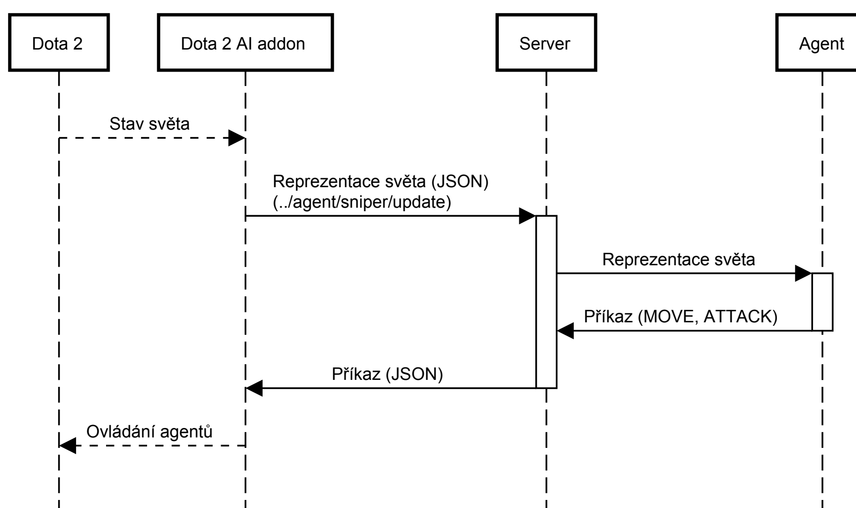
Pro komunikaci je použita metoda POST protokolu HTTP. To je jediný způsob komunikace, který je v Dota 2 Scripting API povolen. To pro komunikaci



Obrázek 2.1: Části Dota 2 AI Frameworku
Framework se skládá z addonu, serverové části a implementace agenta.
Komunikace probíhá prostřednictvím serveru.

mezi serverem a addonem znamená především to, že server může reagovat pouze odpověďmi na požadavky odesílané addonem.

Aby byl addon schopen se serverem komunikovat, rozlišuje framework příchozí dotazy podle URL adresy, na kterou jsou adresovány. Dotazům, kterým je přiřazen nějaký význam budeme dále říkat požadavky. Požadavky addon používá k identifikaci odchozích dat. Například na Obrázku 2.2 je dotaz posílán na adresu serveru, která končí „/update“. Tento dotaz v kontextu frameworku označujeme jako požadavek *update*.



Obrázek 2.2: Ukázka požadavku *update*
Na obrázku je znázorněn tok požadavku *update*. Přes server je poslána reprezentace světa agentovi, který odpovídá příkazem (např. MOVE). Příkaz server předá zpátky addonu, který jej vykoná.

Požadavek *update* je nejdůležitějším požadavkem addonu, protože odesílá stav herního světa. Ten je na straně addonu zakódován do JSONu (JavaScript Object Notation) a odeslán na server. Server JSON přeloží do hierarchie objektů, které odpovídají herním entitám a tuto reprezentaci předá agentovi. Agent se může na svět podívat a sdělit serveru jeden z příkazů, který chce vykonat (např. pohni se). Server pošle příkaz zpátky addonu.

Odpověď na požadavek *update* je addonem interpretována jako příkaz agentovu hrdinovi, který má vykonat ve hře. Například může agent obdržet svět, ve kterém se dozví, že má málo životů. V reakci může addonu odpovědět příkazem,

který jeho hrdinu pohne směrem k jeho bázi.

Příkazy, které framework implementuje, umožňují agentům pohybovat se, útočit na entity, kouzlit, nakupovat a používat některé předměty. Příkazy jsou ovšem dost omezené a některé, ke kterým by měli mít agenti přístup úplně chybí. Například je možné použít pouze pár typů kouzel a nakupovat jde pouze přímo u obchodu.

V Dota 2 Scripting API bychom agenta implementovali na straně addonu. Framework přesunul implementaci addonu do prostředí mimo hru, kde umožňuje jeho implementaci v Javě. Agent tak nesdílí stejný proces se hrou a je schopen pracovat na složitějších výpočtech. Mohou tak být vyvíjeni složitější agenti, kteří potřebují větší výpočetní výkon. Další výhodou je možnost vyvíjet agenty v Javě.

Přesunutí agentů mimo prostředí Dota 2 Scripting API ovšem přináší i negativa. Jelikož agent nepracuje přímo s API, ztrácí funkcionalitu, ke které by měl jinak přístup. Je tedy závislý na abstrakci, kterou mu framework poskytne. Dalším negativem je nutnost komunikace addonu se serverem. Přesun informací vytváří odezvu v reakcích agenta, který nemůže reagovat stejně rychle, jako kdyby reagoval uvnitř API. Agentovo chování je přímo závislé na frekvenci příchozích dotazů.

Původně jsme chtěli vyvíjet agenty nad tímto frameworkem. Rychle se ovšem projevila spousta nedostatků, které nám v jejich vývoji zabránily. Jelikož je nedostatků více, rozhodli jsme se jim věnovat v následující podkapitole. Na nedostatky se později odkazujeme při návrhu našeho frameworku.

2.4 Nedostatky

Problémy frameworku můžeme rozdělit na dvě části. První částí jsou problémy, které vyplývají přímo z nedostatků Dota 2 Scripting API, nad kterým je vybudován Dota 2 AI addon. Druhá část problémů je způsobena zvolenou architekturou a nedokonalou komunikací mezi částmi frameworku.

Dota 2 Scripting API má následující nedostatky:

- **Špatná dokumentace.** API sice poskytuje velké množství funkcí, ale pouze malá část z nich je vhodně dokumentována.
- **Minimální podpora.** API není udržováno vývojáři. Některé funkce nefungují nebo fungují špatně (např. nejde vkládat předmět do inventáře pomocí indexu).
- **Chybějící funkčnost.** Některé funkce, které jsou hráčům dostupné v API chybí. Například nemůžeme aktivovat svatyni.
- **Jeden proces.** Na vykonání jakékoliv synchronní činnosti, která trvá moc dlouho, musí hra čekat.
- **Lua.** Lua nenabízí tolik knihoven a rozšíření jako běžné programovací jazyky.
- **Chybí důležitá místa.** Například chybí reprezentace táborů a obchodů, které nejsou v API přímo dostupné.

Framework, který je vytvořen nad touto API je z velké části omezen jejími nedostatky. Má ovšem i další nedostatky, které ve výsledku znemožňují smysluplný vývoj agentů. Požadavky jsou číslovány a je na ně odkazováno v následujících podkapitolách, ve kterých navrhujeme jejich řešení:

1. **Je umožněn vývoj pouze jednoho agenta.** Ten navíc může hrát pouze v jednom z týmů.
2. **Některé z agentových příkazů nefungují.** Například existují kouzla, které nejdou použít.
3. **Chybí reprezentace mapy.** Hráči nemají informace o výšce a průchodnosti terénu.
4. **Chybí důležitá místa a jejich stav.** Ve frameworku chybí například runy a obchody, bez kterých si nelze chování inteligentních agentů představit.
5. **Není dostupná žádná reprezentace herních předmětů.** Agenti nemají představu, kde se předměty nakupují, kolik stojí a z jakých předmětů se mohou skládat.
6. **Framework nepodporuje testování vyvinutých agentů** — neposkytuje nástroje pro logování a interakci s addonem, chybí mu GUI.

Po analýze frameworku a skriptovacího API jsme došli k názoru, že pro umožnění tvorby agentů bude potřeba framework přepracovat, abychom nad ním mohli vytvářet smysluplné agenty. V další sekci proto navrhujeme vlastní řešení, které z frameworku vychází.

2.5 Návrh řešení

V předchozích podkapitolách jsme definovali problémy Dota 2 Scripting API a Dota 2 AI Frameworku. V této podkapitole navrhujeme řešení, které odstraní co největší množství zmíněných problémů.

Úpravy navrhujeme ve dvou částech. Nejprve navrhujeme úpravy addonu, který musí podporovat všechny agentovy akce, odesílat všechny vhodně reprezentované informace a musí umět správně komunikovat se serverem. Poté navrhujeme úpravy serveru a agenta, kteří musí být schopni s příchozími informacemi správně pracovat. Návrhy úprav využijeme při implementaci našeho rozhraní.

Úpravy addonu

Při analýze frameworku jsme zmínili, že agent posílá své příkazy hře pouze v odpovědi na požadavek *update*. V tomto požadavku jsou posílány informace dostupné celému agentovu týmu (např. pozice všech jednotek a budov). Ze všech příchozích informací agent většinou použije jen své nejbližší okolí.

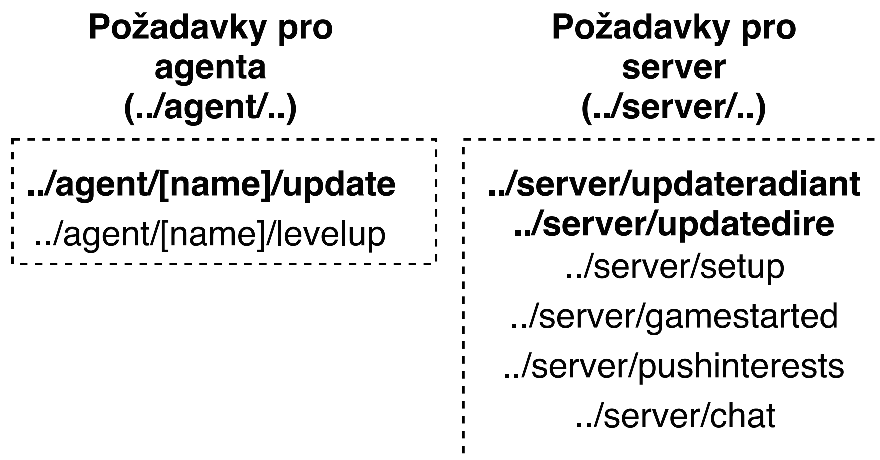
Zpracováním předávaných informací addon i server tráví netriviální dobu. Mezi addonem a serverem tak vzniká odezva, která znamená, že agent nemusí reagovat včas a také zabraňuje častějšímu odesílání požadavků. Pokud na server nepřichází dostatek požadavků typu *update*, je chování agentů viditelně opožděné.

Potřebujeme vylepšit parametry požadavků, na kterých je chování agentů závislé. Nejprve identifikujeme hlavní faktory, které mají vliv na odezvu:

- **Velikost požadavků.** Odesílání velkých požadavků zatěžuje addon i server. Například si nemůžeme dovolit posílat úplnou reprezentaci celého světa.
- **Frekvence požadavků.** Časté odesílání požadavků zpomaluje hru a může znemožnit příjem příkazů.
- **Rychlost odpovědí.** Pokud je na požadavky odpovídáno moc rychle, může být ovlivněn výkon hry. Addon potřebuje dělat jiné věci, než jenom odesílat zprávy.

V ideálním případě by požadavek *update* obsahoval informace o celém světě a přicházel by s co největší frekvencí. Poté by mohl ihned reagovat na jakoukoliv změnu a měl by všechny dostupné informace. Takových podmínek ovšem nemůže být dosaženo například proto, že addon není schopen vytvářet reprezentaci celého světa v rozumném čase. Při návrhu řešení se chceme přiblížit takovým parametrům, které učiní agentovo chování věrohodným vůči chování lidských hráčů.

Začneme tím, že požadavky odesílané addonem rozdělíme do dvou skupin. Každé ze skupin přiřadíme speciální část adresy („../server/..“ a „../agent/..“). Tím budeme moci posílat požadavky určené agentovi a serveru zvlášť a přidělit jim jiné funkce.



Obrázek 2.3: Navržené požadavky
Souhrn všech navržených požadavků addonu. Požadavky odesílající reprezentace světa jsou zvýrazněny.

Dále se zaměříme na zlepšení parametrů požadavku *update*. Původní framework posílá v každém požadavku typu *update* stav celého světa. To není efektivní, protože se ztrácí hodně času na vytvoření a zpracování požadavku. Zároveň se tím ztrácí informace o objektech viditelných pouze jedním týmem.

Vytvoříme dvě varianty požadavku *update*, které budou určeny agentovi a serveru. Agentům budeme posílat pouze informace o jejich nejbližším okolí, které je

pro ně viditelné. Na server budeme naopak adresovat větší požadavky, které budou obsahovat informace dostupné pro jeden z týmů. Podle týmu bude požadavek na straně serveru dále rozdělen do dvou požadavků (*updatedire* a *updateradiant*).

Agentům posíláme pouze informace, se kterými pracují nejčastěji a potřebují, aby byly aktuální. Okolí agentů obsahuje pouze málo objektů, takže jejich posíláním a zpracováním ztratíme minimum času. To nám zajistí malou odezvu na straně addonu i serveru, čímž bude agentova reakce rychlejší. Zároveň nám to umožní posílat tento požadavek daleko častěji.

U požadavků určených serveru nám nevadí delší zpracování. Jelikož posíláme informace odděleně pro každý z týmů, umožní nám to uvnitř rozhraní rozlišit mezi tím, co vidí jednotlivé týmy.

Takto rozdělenou komunikaci můžeme dále využít pro podporu vývoje více agentů (1). K tomu je třeba k adrese určující typ požadavku přidat ještě jeho identifikátor (např. jméno hrdiny „*../agent/sniper/..*“), abychom při komunikaci správně identifikovali agenty a jejich hrdiny.

Jelikož budeme mít agentů více, je třeba ještě vytvořit nastavení agentů a rozložit je do dvou týmů. Nastavení můžeme udělat na straně serveru. Nejlepší řešení je přidat nový požadavek („*../server/setup*“) určený serveru, pomocí kterého se addon před začátkem hry zeptá, jací hrdinové by měli hru hrát. V odpovědi server pošle nastavení agentů a addon hru nastaví.

Dále bude potřeba reprezentovat mapu a herní předměty (3 a 5). Mapa i předměty představují velké množství dat, které se ovšem v průběhu hry výrazně nemění. Tyto informace tedy nemusíme posílat, ale jejich reprezentaci můžeme uložit do souboru, ze kterého si je server načte při svém spuštění.

Důležitá místa (4) je nutné odeslat přímo z addonu. Sice se nejedná o herní entity, jejichž stav by se neustále měnil, ale většina z nich má unikátní identifikátor, který se vytváří na začátku hry. Identifikátor některých entit se v průběhu hry může změnit. Agent identifikátory potřebuje při interakci s entitami (např. zaútoč).

Navrhujeme vytvořit seznam, který bude obsahovat identifikátory, jména míst a jejich umístění na mapě. Seznam bude odeslán na začátku hry serveru pomocí speciálního požadavku (*pushinterests*). Server se musí postarat o reprezentaci těchto objektů.

Některá místa zároveň nestačí poslat na začátku hry, protože se nemusí chovat staticky. To se týká především run a svatyní. Runy po použití zmizí a pokud se znovu objeví, mají nový identifikátor. Naopak svatyni může jeden z týmů zničit, což musí být rozhraní schopno zpozorovat.

K tomuto problému přistoupíme tak, že při posílání reprezentace světa v okolí hráčů, budeme posílat také informace o nedalekých runách a svatyních. Samozřejmě pouze tehdy, když na ně hráč vidí.

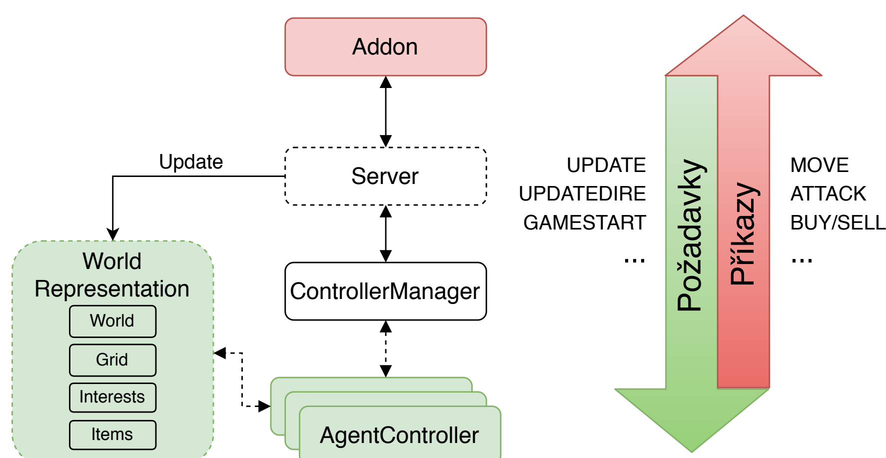
Pro vytvoření agentových příkazů, které ve frameworku chybí (2), je třeba přidat do addonu nové funkce a upravit funkce stávající. Například bude třeba vytvořit funkce, které se budou starat o nákup předmětů na větší vzdálenost a o jejich získání ze skladiště.

Pro ovládání addonu a hry (6) je ještě třeba definovat příkazy, které mohou být vráceny jako odpověď na jakýkoliv požadavek (určený agentům i serveru). Ty použijeme pro ovládání addonu a hry. K tomu nám budou stačit příkazy na pozastavení hry a restartování addonu.

Všechny navržené požadavky jsou shrnuty na Obrázku 2.3. Rozdělení na požadavky dále využijeme při implementaci rozhraní a addonu.

Úpravy serveru a agenta

Navrhli jsme takové úpravy addonu, které eliminují co nejvíce nedostatků frameworku. Nyní je třeba navrhnout serverovou část a agenta tak, aby byl schopen úpravy využít.



Obrázek 2.4: Části rozhraní
Na obrázku jsou znázorněny základní části našeho rozhraní. Vpravo je znázorněn tok požadavků a příkazů rozhraním.

Při úpravě zbytku frameworku se především snažíme o rozšíření serverové části do více komponent (viz Obrázek 2.4), které nám umožní smysluplně pracovat se světem a agenty. Z pohledu uživatele je důležité, jakým způsobem budeme reprezentovat herní informace.

Server musí být nejdříve upraven tak, aby byl schopen přijímat a rozlišovat příchozí požadavky. Na serveru chceme přímo řešit pouze požadavky, které jsou mu určeny a zbytek předávat do dalších částí frameworku. Server by měl většinu práce předat dále a zabývat se pouze řešením problémů spojených s komunikací.

Navrhujeme tedy framework rozdělit do více částí, se kterými bude serverová část komunikovat. Jádrem našeho rozhraní bude GUI, ve kterém bude uživateli umožněno agenty nastavit, nahlížet na stav světa a ovládat tok hry. Dále bude potřeba vhodně reprezentovat svět.

Jak už bylo řečeno v předchozí části, svět bude rozdělen do několika odlišných částí. Nejdůležitější částí reprezentace bude reprezentace herních entit. Té v původním frameworku chyběla především reprezentace inventáře hrdinů a některé důležité atributy herních entit.

Herní entity bude výhodné rozdělit na několik částí, které budou definovány tím, kdo jednotlivé entity může vidět. Skupinám objektů, které jsou viditelné jedním z agentů nebo v rámci jednoho týmu budeme říkat kontext. V implementaci tedy budeme využívat agentův a týmový kontext.

Reprezentace herních objektů se bude neustále měnit. Naopak důležitá místa se budou měnit minimálně. Jejich reprezentace tedy může být o něco kompliko-

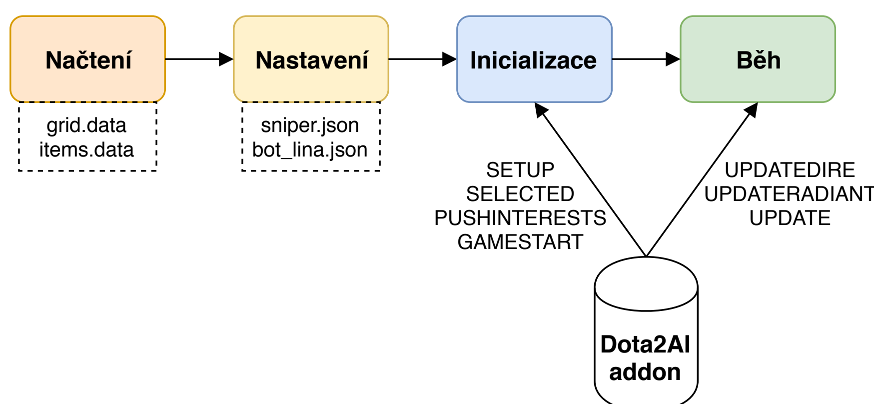
vanější než herních entit. Zde navrhujeme reprezentovat objekty tak, aby reflektovaly jejich herní význam.

Herní mapa může být reprezentována jako pole. Bude pouze třeba, aby měl uživatel přístup k jednotlivým políčkům, které nesmí být moc velké a měly by v sobě ukládat informace o své průchodnosti a výšce.

Informací o předmětech máme malé množství, takže mohou být všechny načteny do paměti a uživatel k nim může přistupovat podle jejich jmen. Také by měly mít přehled o své ceně, komponentách a obchodě, ve kterém mohou být koupeny.

Abychom umožnili vývoj více agentů (1), je třeba vytvořit část frameworku, která je bude spravovat. Ta by měla umožňovat jejich nastavení před začátkem hry, jejich ovládání v jejím průběhu (zahrnuje distribuci požadavků) a měla by umožnit agenty spravovat. Z pohledu uživatele by měla být většina funkcí dostupná pomocí GUI.

GUI bude představovat hlavní část, se kterou bude uživatel rozhraní pracovat, pokud zrovna nebude psát kód. Rozhraní by mělo být jednoduché a mělo by poskytovat nastavení agentů, aktuální stav světa, nástroje pro ovládání hry a nástroje pro komunikaci s agenty (logování a konzole).



Obrázek 2.5: Stavy rozhraní
Navrhujeme několik stavů rozhraní.

Pro přehlednost můžeme ještě lépe definovat stavy, kterými si bude muset naše rozhraní projít. Ty jsou dány stavem hry a informacemi, které má rozhraní v danou chvíli k dispozici. Přechody mezi stavy ovlivňuje uživatel, který grafickou část rozhraní ovládá. Stavy rozhraní jsou vidět na Obrázku 2.5. Stavy vypadají následovně:

- **Načtení**: Při načítání musíme načíst všechny externí data, které addon vyexportoval ze hry. To jsou reprezentace mapy (grid.data) a předmětů (items.data).

- **Nastavení**: Nastavíme jednotlivé týmy a agenty, které v další fázi pošleme addonu (ten je nastaví na straně hry). Agenty budeme nastavovat pomocí konfiguračních souborů, které můžeme reprezentovat pomocí JSONu (sniper.json).

- **Inicializace**: Začneme komunikovat s addonem. Vyměníme si informace o důležitých místech a odešleme addonu nastavení týmů. Nastavování bude

probíhat v reakci na požadavek *setup*. Důležitá místa addon odešle na začátku hry pomocí požadavku *pushinterests*.

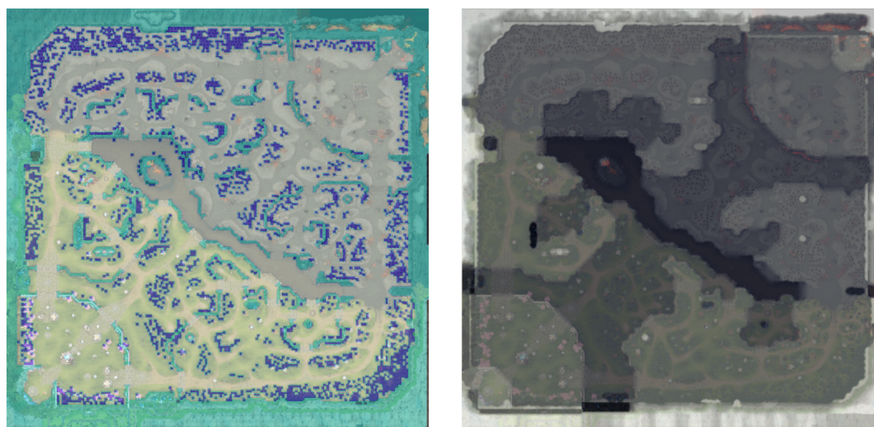
- **Běh**: Aktualizujeme svět a komunikujeme s agenty. Především přijímáme aktualizace světa, které addon rozhraní posílá pomocí požadavků typu *update*, na které odpovídáme příkazy agentů.

3. Implementace rozhraní

V této kapitole popisujeme architekturu našeho rozhraní, která reflektuje návrh vytvořený v minulé kapitole. V první části se zabýváme architekturou addonu na straně Doty 2. V druhé části popisujeme architekturu samotného rozhraní, určeného pro vývoj umělých agentů.

3.1 Addon

Původní addon jsme se rozhodli rozdělit do dvou částí. První částí je jednoduchý samostatný addon, jehož úkolem je ze hry vyexportovat důležité statická data (reprezentaci mapy a předmětů). Druhou částí je dota2ai addon, který jsme rozšířili tak, aby respektoval požadavky zmíněné při návrhu rozhraní.



Obrázek 3.1: Získaná reprezentace mapy
První obrázek obsahuje neprůchozí místa, rozlišené na stromy (fialová) a neprůchozí terén (zelená). Druhý obrázek zachycuje výšku — světlejší oblasti jsou umístěny výše.

Mapu jsme se nejdříve snažili vyexportovat s co největším detailem. Zjistili jsme ovšem, že detailní zachycení výšky a informací o průchodnosti mapy vytváří příliš velké soubory. Ty vedou k velkým paměťovým nárokům na rozhraní, především když používáme více instancí mapy (mapy vlivu). Proto jsme se rozhodli zachytit mapu v menším detailu (viz Obrázek 3.1) a ukládat informace pouze o každém n-tém políčku (defaultně n=8).

Jediná informace, kterou Dota 2 Scripting API poskytuje o předmětech je jejich cena. Pro export předmětů jsme museli ručně vytvořit seznam všech jmen předmětů, komponent ze kterých se skládají a obchodů, ve kterých mohou být koupeny. Addon pouze zjistí jejich cenu a výsledek uloží do souboru.

Vytvoření samostatného addonu pro export map a předmětů je výhodné, pokud se změní mapa nebo ceny předmětů (např. s novou verzí hry). Zároveň nezatěžujeme dota2ai addon, který se exportem nemusí zabývat.

Při návrhu addonu jsme řešili, jak dobře zvolit frekvenci a velikost požadavků. Při implementaci jsme přesně jako v návrhu vytvořili více typů požadavků, které adresujeme serverové části nebo je agentům.

Požadavky určené serveru, které obsahují informace viditelné pro jeden tým, posíláme co sekundu. Addon odesílá informace o všech viditelných věžích, kasárnách, hrdinech a jednotkách.

Požadavky určené agentům, které obsahují výrazně méně informací, posíláme co čtvrt sekundy. Ty obsahují informace pouze o okolí agentů. Dále posíláme také identifikátory některých důležitých míst v agentově okolí (runy a svatyně).

Abychom správně nastavili hru, bylo dále potřeba vytvořit uvnitř addonu reprezentaci jednotlivých agentů. Addon začíná tím, že se rozhraní zeptá, jací hráči by měli hru hrát. Rozhraní addonu vrátí jejich nastavení, které mu řekne, co za hráče hru hraje a pomocí jakých identifikátoru s nimi bude rozhraní komunikovat.

Hráči mohou být ovládání hrou, člověkem nebo agentem. Informace o agentech si addon dále ukládá, aby jimi mohl adresovat požadavky nebo pro jiné akce, které jsou na nastavení závislé.

Dokumentace Scripting API dobře nespecifikuje, jak vytvořit jiného hráče, než takového, kterého ovládá hra (bota). Po dlouhém hledání jsme nalezli způsob, jakým se dá vytvořit tzv. falešný hráč (fake client), kterého můžeme prostřednictvím Doty 2 ovládat.

Problémem je, že i falešný hráč je defaultně kontrolován hrou. Ta definuje vlastní skripty, které hrdiny ovládají. Primárně je ovšem hledá ve složce s addonem. Proto jsme museli zajistit, aby rozhraní po svém nastavení vytvořilo prázdné skripty pro všechny hrdiny, které budou agenti ovládat.

Dalším problémem byl export důležitých míst, který probíhá hned po začátku hry. U většiny míst nám stačí pouze jejich identifikátor a umístění na herní mapě. To se týká například věží, kasáren nebo run.

Největší problém byl, jakým způsobem zjistit příslušnost jednotek k jedné z linek nebo k jednomu z táborů umístěných v lese. Scripting API sice nabízí možnost, jak reagovat na událost vytvoření jednotky, ale jakékoliv pokusy o vyřešení problému na straně addonu selhaly. Proto jsme se rozhodli vyřešit tento problém až na straně serveru porovnáváním vzdáleností.

Poté, co jednotku poprvé spatříme, ji přiřadíme jednomu táboru nebo lince. To není složité, protože jednotka přísluší pouze jedné ze tří linek a tábory řešíme pouze poté, co se k nim hrdinové přiblíží.

Dále bylo třeba addonu přidat další příkazy, které v původním frameworku chyběly. Opravili jsme kouzla, která nešla použít a upravili jsme nakupování předmětů, které nyní respektuje pravidla Doty 2. Zde jsme se opět setkali s nedostatky Sripting API, které neumožňuje vložit předmět do inventáře podle indexu.

Dále jsme zlepšili tvorbu požadavků. Interpretace odpovědi na požadavky také prochází společným místem. Zde se addon ptá, jestli po něm rozhraní nechce, abychom hru pozastavili nebo provedli nějakou jinou činnost, která se nevztahuje ke konkrétnímu agentovi.

3.2 Architektura rozhraní

Pro implementaci rozhraní jsme se rozhodli použít Javu, kterou používal také původní framework. Ta usnadňuje přenositelnost a umožňuje snadno načítat kódy agentů pomocí reflexe. Pro Javu také existuje spousta nástrojů, které nám usnadnily vývoj.

Prvním krokem k vytvoření rozhraní byla implementace HTTP serveru, který zpracovává příchozí požadavky. Původní framework používá *NanoHTTPD*. Ten ovšem způsoboval problémy — při posílání většího množství dat začal vytvářet dočasné soubory, které poté neuměl smazat. Rozhodli jsme se proto použít *Jetty*.

Úkoly serveru jsme specifikovali v minulé kapitole. Především se snaží o zpracování příchozích požadavků a jejich distribuci. Sám řeší pouze ty požadavky, které mu jsou přímo určené. Jako odpověď vrací řídící příkaz nebo příkaz některého z agentů.

Data jsou mezi addonem a serverem odesílány ve formátu JSON. Pro jeho deserializaci jsme použili *Jackson*, který byl použit původním frameworkem. Jackson nám umožňuje přeložit JSON do hierarchie objektů, což se nám hodí při deserializaci herních objektů. *Jackson* dále používáme pro načítání a ukládání nastavení agentů.

O zpracování požadavků se stará třída **RequestHandler**. Ta zpracovává požadavky určené rozhraní. Požadavky určené agentům předává třídě **ControllerManager**, která požadavek předá příslušnému agentovi. Agent odpoví vhodně definovaným příkazem, který má jeho hrdina provést.

Každý požadavek typu *update* aktualizuje svět reprezentovaný uvnitř rozhraní. Svět je složen z částí, které se snažíme reprezentovat několika třídami. Většina z těchto tříd jsou singletony nebo statické třídy, protože musí být uživatelům dostupné z jakékoliv části jejich implementace. Třídy reprezentují mapu, bázi předmětů, herní svět a herně důležitá místa.

Ve zbytku kapitoly popíšeme všechny výše zmíněné reprezentace, reprezentaci herních entit a nakonec agentů samotných. Kapitolu ukončíme popisem vyvinutého grafického prostředí.

Reprezentace mapy a předmětů

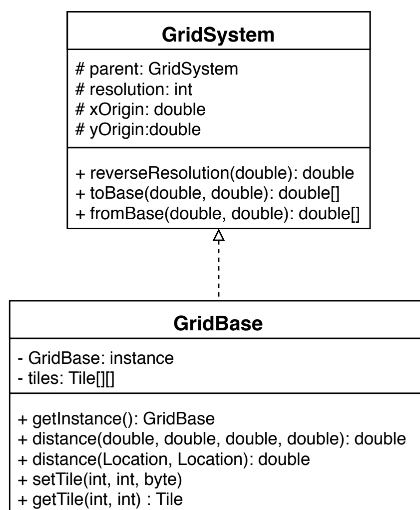
První částí světa, která je načtena ihned po spuštění aplikace je herní mapa. Herní mapu reprezentujeme třídou **GridBase**. Tato třída v sobě ukládá pole, které pro každé políčko obsahuje informace o jeho průchodnosti a výšce. Jelikož každé políčko reprezentuje několik políček herní mapy, museli jsme vyřešit přepočítávání zpět do herních souřadnic.

Z toho důvodu jsme implementovali třídu **GridSystem** (viz Obrázek 3.2), která souřadnice přepočítává v závislosti na počátku souřadného systému a jeho rozlišení. Jelikož se nulové souřadnice nachází uprostřed herní mapy, museli jsme ještě systém transformovat tak, abychom jej mohli dobře reprezentovat pomocí pole.

GridBase nám umožňuje přepočítávat herní souřadnice posunutím do našeho souřadného systému. Zároveň jsme implementovali metody, které umožňují souřadnice přepočítávat se zahrnutím rozlišení nebo bez něj. To se nám hodí ve chvíli, kdy potřebujeme přepočítávat herní vzdálenosti (např. dostřel), na které má vliv pouze rozlišení.

Třídu **GridBase** využíváme také při implementaci map vlivu, které používáme při implementaci agenta. Mapa vlivu je v podstatě část mapy, u která nám nevadí menší rozlišení. Zároveň můžeme chtít souřadný systém posunout nebo zvětšit třeba podle toho, jak daleko agent vidí.

Předměty jsou načteny ze souboru a uloženy ve třídě **ItemsBase**. Tato třída



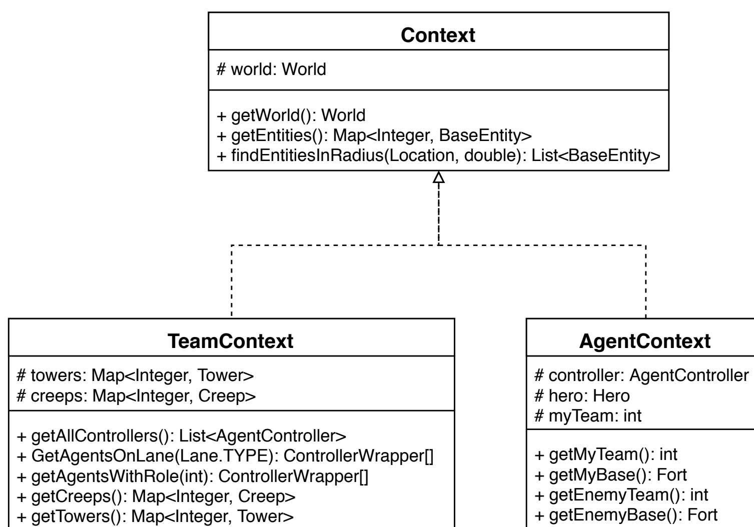
Obrázek 3.2: GridSystem a GridBase
Některé z metod, které poskytují **GridSystem** a **GridBase**.

ukládá mapu předmětů a uživateli k nim umožňuje přístup pomocí jejich názvů. Každý předmět ví o své ceně, o obchodu, ve kterém může být koupen a o částech, ze kterých se skládá.

Pro agenty jsou předměty jejich hrdiny dostupné primárně skrze třídu **Hero**, která obsahuje reprezentaci inventáře.

Reprezentace světa

Při návrhu jsme rozdělili svět do kontextů (viz Obrázek 3.3). Oba typy kontextů pracují s třídou **World**. Ta ukládá entity podle jejich identifikátorů a poskytuje základní funkce, které agentům umožňují s entitami pracovat.



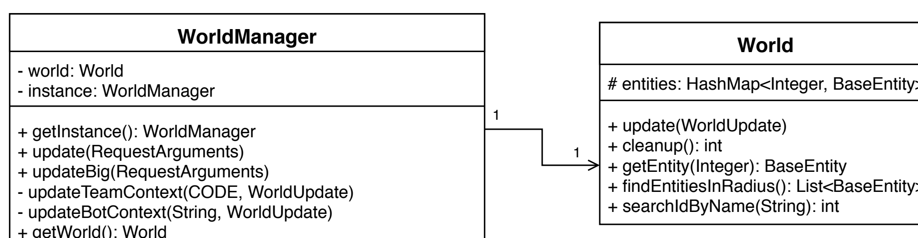
Obrázek 3.3: Context, AgentContext a TeamContext
Zjednodušená struktura kontextů.

Rozdělení světa na dva kontexty je pro nás výhodné také proto, že nám umožnilo definovat v kontextu další funkce a atributy, které se mohou agentům při práci

se světem hodit. Agent tak nemusí definovat tolik vlastních proměnných, které by musel sám aktualizovat.

Agentův kontext je reprezentován třídou **AgentContext**. Tato třída udržuje pro jednoho agenta informace například o tom, jakého týmu je součástí, na které lince hraje nebo kde leží jeho báze. Původně jsme kontext koncipovali tak, aby mohl cachovat některé z agentových složitějších požadavků, nicméně pro to jsme nakonec nenalezli velké uplatnění.

Týmový kontext reprezentujeme pomocí třídy **TeamContext**. Do té ukládáme informace o agentově týmu, o které se může ve hře zajímat. Může jít například o to, jací agenti s ním hrají na lince nebo jaká je jejich role. Informace z tohoto kontextu může využít také na to, aby měl představu o stavu hry.



Obrázek 3.4: WorldManager a World
Obě třídy jsou pro čitelnost zjednodušeny.

Důležitou třídou je třída **WorldManager**. Všechny požadavky typu *update* jí musí projít. Tato třída spravuje reprezentaci světa a ukládá v sobě jednu instanci třídy **World**, do které ukládá všechny příchozí herní entity (viz Obrázek 3.4). Všechny entity v kontextech jsou pouhou referencí na objekt uložený v této třídě.

Tato struktura byla zvolena především proto, abychom mohli řešit životnost objektů pouze na jednom místě a aby měl uživatel přístup ke stavu celého světa. To se může hodit například při použití některých z přístupů k implementaci umělé inteligence (např. strojového učení).

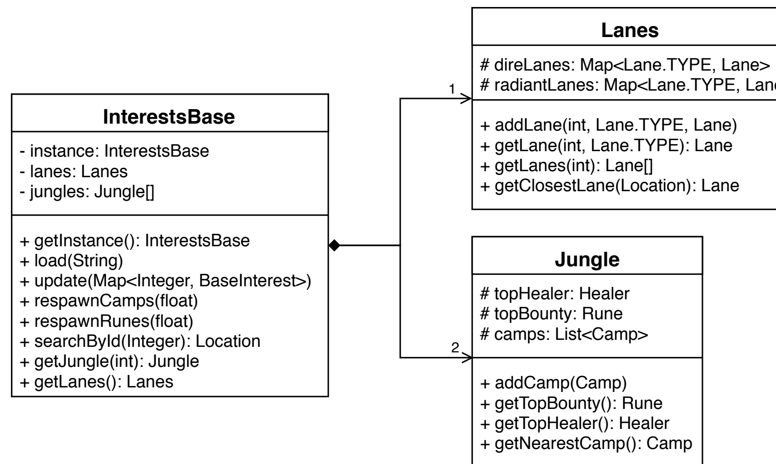
Každá herní entita má svou životnost, která se dá upravit v grafické části rozhraní. Životnost udává, po jaké době entitu smažeme z rozhraní, pokud o ni nedostaneme žádnou informaci (v rámci požadavku). Čas je nastaven vyšší pro statické objekty (>1s) a nižší pro dynamické objekty (<1s) a je udáván v herním čase.

Reprezentace významných míst

Pro reprezentaci významných míst jsme implementovali třídu InterestsBase. Úkolem této třídy je zprostředkovat uživateli přístup ke všem důležitým objektům a místům na mapě.

Objekty uložené v této třídě mohou být uloženy také v reprezentaci světa (např. věže). Tato duplicita je pro nás ovšem výhodná, protože objekty mohou ze světa zmizet, ale hráč se může i tak zajímat o jejich polohu. Např. věž může být zničena, ale zároveň může být použita pro navigaci.

Mezi důležité objekty, které v rozhraní reprezentujeme patří tábory, pevnosti, svatyně, věže, kasárny, obchody a runy.



Obrázek 3.5: Objekty InterestsBase, Lanes a Jungle
InterestsBase reprezentuje důležitá místa na herní mapě. Obsahuje například objekty reprezentující les (**Jungle**) a linky (**Lanes**).

Objekty jsme se rozhodli sdružit do oblastí na základě jejich umístění. Při tvorbě oblastí reflektujeme strukturu mapy Doty 2. Například definujeme les, ve kterém jsou tábory nepřátel, runy, svatyně a obchody (viz Obrázek 3.5).

V průběhu hry je nutné některé z uložených objektů aktualizovat. To se týká především run a táborů. Runy jsou obnovovány v určitém intervalu. Při každém jejich obnovení se změní jejich identifikátor. Naopak v táborech mohou některé jednotky umřít. Nové jednotky mají nový identifikátor a mohou mít i jiné vlastnosti. Bez správného identifikátoru nemůže hrdina s objekty interagovat.

Pro aktualizaci táborů jsme se rozhodli, že nejjednodušší bude jednotlivé objekty aktualizovat, když je dostaneme v některém z požadavků. Ve chvíli, kdy se poprvé setkáme s neutrální jednotkou, tak ji přiřadíme nejbližšímu táboru. Runy aktualizujeme ve chvíli, kdy je uvidíme v požadavku určeném pro některého agenta.

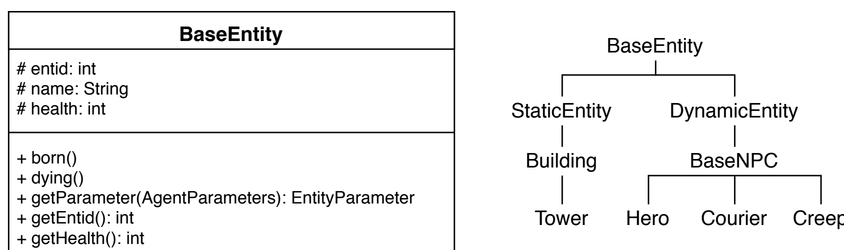
Tábory se obnovují každou minutu, runy každé dvě minuty. Abychom byli schopni zaznamenat, kdy k těmto situacím dojde, vytvořili jsme třídu **TimeManager**. Ten v sobě udržuje čas, který je synchronizován s herním časem po každém požadavku typu *update*.

Tím jsme zaručili, že když bude agent potřebovat informace o významném místě, které je pro něj viditelné, bude mít správné informace. Na delší vzdálenosti ovšem pozná pouze to, jestli může být místo aktivní nebo ne (podle časů obnovení).

Použití třídy **TimeManager** nám také umožní zaznamenat čas, kdy je hra pozastavena. Tak umožníme uživatelům používat čas i uvnitř agentů, který respektuje pozastavení hry.

Herní objekty

Všechny herní objekty, které ukládáme do objektu **World**, rozšiřují třídu **BaseEntity** (viz Obrázek 3.6). Ta reprezentuje všechny jejich společné vlastnosti. To je například zdraví, mana nebo jejich pozice na mapě. Jednotlivé objekty přidávají vlastní metody a atributy, které jsou pro ně typické.



Obrázek 3.6: BaseEntity
Nalevo je třída BaseEntity a některé její metody. Napravo je struktura herních entit reprezentovaných v rozhraní.

Herní objekty načítáme přímo z příchozího JSONu. Každý příchozí objekt má položku „type“, která určuje jeho typ (např. budova). Při deserializaci je vytvořena instance objektu, který odpovídá danému typu. Instance je následně inicializována hodnotami položek z JSONu, jejichž jména odpovídají atributům objektu.

Pokud se s herním objektem setkáváme poprvé, je na něm zavolána metoda **born()**. Pokud je odstraňován, je zavolána metoda **dying()**. Tyto metody využíváme k tomu, aby se objekty samy odstranily z objektů, ve kterých je na ně uložena reference. Konkrétně se třeba jednotka sama odstraní z týmového kontextu, z neutrálního tábora nebo z linky na které se pohybuje.

Hrdina je objekt, který má nejvíce vlastností. Addonem je posílána reprezentace jeho inventáře a všech jeho kouzel. Kouzla jsou samy o sobě entitami, které vznikají a zanikají společně s hrdinou. Pro kouzla si ukládáme informace například o jejich síle a dosahu. Hrdinův inventář reprezentujeme třídou **Inventory**, která v sobě ukládá jednotlivé předměty a je rozdělena na inventář, batoh a skladiště.

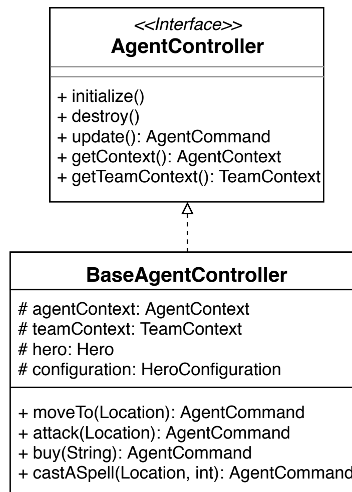
Při implementaci se ukázalo, že potřebujeme být schopni entitám přiřadit parametry. Prvním parametrem je čas, po který může objekt existovat. Tento parametr jsme nastavili jako statickou položku jednotlivých tříd, která se nastavuje v konfiguraci rozhraní.

Dále jsme vytvořili třídu **AgentParameters**, ve které ukládáme parametry tříd, které reprezentují entity. Entity sami vrací své parametry nebo parametry třídy jim nadřazené. Toho jsme využili například při práci s mapami vlivu, pro které jsme potřebovali šířit vliv v závislosti na typu entity.

Agenti

Posledním krokem bylo umožnit uživateli tvorbu vlastních agentů. K tomu vytváříme rozhraní **AgentController**. To definuje základní funkce, pomocí kterých rozhraní komunikuje s agentem. Nejdůležitější metodou je metoda **update()**, která je volána po každé aktualizaci agentova světa a měla by vracet příkaz, který je poslán addonu.

Rozhraní jsme rozšířili třídou **BaseAgentController**. Ta za agenta implementuje některé z funkcí, které může agent potřebovat. Jde například o funkce, které implementují vytváření agentových příkazů, jako jsou pohyb a útočení. Tato třída dává agentovi přístup k objektům **AgentContext** a **TeamContext**.



Obrázek 3.7: AgentController a BaseAgentController
Třídy, které reprezentují agenta. Pro čitelnost zjednodušeny.

Ovladač agenta načítáme do rozhraní prostřednictvím jeho nastavení, ve kterém uživatel specifikuje cestu k třídě implementující ovladač. Rozhraní si vytvoří instanci třídy a uloží ji do třídy **ControllerManager**.

Úkolem třídy **ControllerManager** je především starat se o život instancí agentů a distribuovat požadavky, které jsou jim určené. Zároveň se stará také o to, aby byli agenti dobře nastaveni při spuštění hry.

Aby rozhraní umožnilo lepší práci s ovladači agentů, vytvořili jsme třídu **ControllerWrapper**. Ta obaluje třídu **AgentController** a umožňuje rozhraní a uživateli lépe ovládat agenta. Pomocí této třídy můžeme například zastavit ovladač agenta nebo mu posílat příkazy z konzole.

Další důležitou funkcí třídy **ControllerWrapper** je vytváření loggeru při inicializaci. Ten je vytvořen za použití *Log4j* (viz dále). Logger nám umožní vypisovat zprávy z kódu agentů přímo do grafické části rozhraní.

3.3 Grafické prostředí

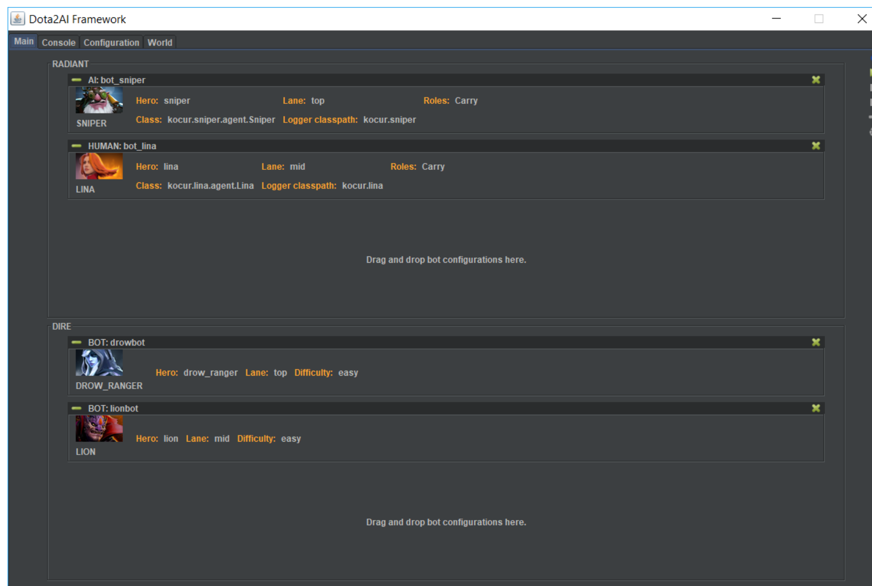
V předchozích podkapitolách jsme se pokusili o vyřešení všech negativ frameworku, které jsme identifikovali v kapitole 2. Posledním negativem původního frameworku je, že nepodporuje testování agentů a postrádá grafické rozhraní.

Při vytváření grafického rozhraní jsme se rozhodli použít *Swing*. Ten je dobře přenositelný a poskytl nám velké množství nástrojů.

Grafické prostředí jsme se rozhodli založit na záložkách. V rámci rozhraní několik záložek definujeme a zároveň umožňujeme uživateli vytvářet si vlastní. Uživatel si tak může vytvořit záložku pro svého agenta, na které může zobrazovat libovolné informace.

Záložky našeho rozhraní umožňují nastavit hru, nastavit jednotlivé agenty, zobrazit si logy agentů nebo si zobrazit aktuální stav světa. Na boku rozhraní také vytváříme jednoduchý ovládací panel, pomocí kterého může agent pozastavit hru nebo hrou krokovat (provést jeden agentův krok a hru pozastavit).

Nastavení hry probíhá tak, že uživatel přetáhne do první záložky soubory, které obsahují nastavení jednotlivých hráčů (viz Obrázek 3.8). Podle nastavení



Obrázek 3.8: Ukázka GUI - nastavení hry
Screenshot GUI při nastavování hry. Na obrázku jdou vidět dva týmy složené ze dvou hráčů.

jsou hráči rozděleni do týmů. Uživatel poté může jejich nastavení měnit a případně zapsat zpátky na disk.

Pro ovládání nebo zobrazování některých informací mohou uživatelé použít také konzoli. Konzole nám umožňuje lépe pracovat s rozhraním a zobrazit informace, které by se špatně zobrazovaly graficky. Například umožňuje vypsat všechny objekty, které se ve světě nachází. Uživateli je umožněno vytvářet vlastní kontrolovatelné třídy implementací rozhraní **Controllable** a registrací ve třídě **ControllersManager**.

Dále jsme uživatelům umožnili graficky zobrazit stav světa. Díky tomu jsme byli schopni otestovat reprezentaci světa a mapy a ukázat uživateli, že je rozlišení našeho světa dostatečně detailní. Zároveň jsou zobrazovány informace dostupné oběma týmům. Tato funkce je dostupná v nástrojích hry, kde ale nefunguje správně, protože nerespektuje vidění agentů.

Poslední záložka se stará o logování. Po pokusech o implementaci vlastního logování jsme se rozhodli použít *Log4j*. Nástroj je hodně známý a umožňuje více způsobů logování. Je tedy lépe použitelný a rozšiřitelný.

Pomocí *Log4j* můžeme logovat do souboru a do konzole. Pro náš účel jsme *Log4j* rozšířili o třídu **QueueAppender**. Ta nám umožňuje přeposílat logy do záložky, ve které s nimi dále pracujeme.

Záložka starající se o logování umožňuje vytvořit záložky zachytávající logy jednoho z loggerů třídy **QueueAppender**. Uživatel může příchozí logy filtrovat podle úrovně logů a jejich obsahu. Také jsme se rozhodli umožnit vytvoření samostatného okna s logy, což uživateli usnadní práci, pokud potřebuje mezi záložkami přecházet.

V Javě je kód rozdělen do balíčků. *Log4j* nám umožňuje definovat kořenové loggery pro celé balíčky nebo projekty. Uživatel je tak schopen logovat z jakéhokoliv místa kódu svého agenta.

3.4 Shrnutí

V předchozích podkapitolách jsme shrnuli nejdůležitější třídy. Další detaily mohou být nalezeny v programátorské a uživatelské dokumentaci, která je součástí přílohy A.

4. Analýza Doty 2 z pohledu AI

V této kapitole analyzujeme Dotu 2 z perspektivy umělé inteligence. V první části se zabýváme rolemi hrdinů. V druhé části hru dělíme na několik fází. Dále zmiňujeme některé z přístupů, které se mohou použít pro implementaci agentů. Nakonec shrnujeme výzkum týkající se Doty 2. Kapitola slouží především jako úvod ke kapitolám 5 a 6.

4.1 Role hrdinů

V rámci jednotlivých týmů mají hráči různé role. Role jsou určené především hrdinou a linkou, které si hráč vybere před začátkem hry. Role definují priority, kterých by se hráči měli v průběhu hry snažit dosáhnout. Hrdina může mít i více rolí.

Následující role jsou typické pro jakoukoliv hru žánru MOBA:

- **Carry** se snaží udílet co největší poškození za cenu malé odolnosti.
- **Tank** se snaží mít co největší odolnost.
- **Ganker** pomáhá hrdinům na jiných linkách.
- **Support** podporuje ostatní hrdiny.
- **Nuker** uděluje velké poškození za krátký čas.

Hráči musí při ovládání hrdinů ovládat tzv. mikromanagement, což je ovládání hry zaměřené primárně na hrdinu a jeho lokální potřeby. Opakem je tzv. makromanagement, který bere v potaz i zbytek týmu a celkový stav hry.

Každá role má vliv na to, jakým způsobem by měl hráč ovládat svého hrdinu. Především jsou role důležité v týmových soubojích a při pohybu po linkách. Na linkách bývá většinou více hrdinů, jejichž role se vzájemně podporují (např. carry a support).

Pro agenty jsou role důležité, pokud hrají v týmu. Agenti by měli mít představu o své roli a o rolích ostatních hráčů. Role ostatních hráčů může agent použít při predikci jejich chování. Agenti by měli hrát podle své role a měli by zastávat unikátní roli v rámci týmu (viz [11]).

Role jsou důležité především v týmových soubojích. Ty bývají iniciovány některým z odolnějších hrdinů. Prioritou bývá snaha vyřadit co nejrychleji ze souboje jednotky, které udělují největší poškození. Klíčovým je také postavení týmu a hrdinů podle jejich rolí.

Většinou bývá souboj iniciován tak, že jeden z týmů zaútočí na nepřátelského hrdinu, který se příliš odhalil. Druhému týmu nezbývá než se mu pokusit pomoct. V takovém případě je výsledek souboje dán především rozestavením týmu a rychlostí reakce jednotlivých hráčů.

Souboj klade velké nároky například na použití kouzel ve správný čas (mikromanagement) a také na správné postavení v rámci týmu (makromanagement). Špatně postavení hrdinové nemohou včas reagovat na hrozby a nemusí být schopni pomoct svému týmu.

4.2 Fáze hry

Dota 2 bývá rozdělována na několik fází. Typicky se fáze dělí podle času hry nebo zlata hrdinů. Chování hráčů a týmů se v jednotlivých fázích hry liší. Mění se i problémy, kterým musí hráči čelit.

Výběr hrdinů

Tato fáze typicky probíhá před začátkem hry. Hráči si vybírají svého hrdinu a linku. Výběr může probíhat naslepo. Preferovanější je ovšem varianta, kdy se hráči při výběru střídají s nepřátelským týmem. V této variantě je také možnost některé hrdiny zakázat.

Někteří hrdinové jsou výrazně silnější než jiní a jejich výběr může mít zásadní vliv na celou hru. Hra teoreticky může být rozhodnuta už při výběru hrdinů.

Agent si musí vybírat primárně z hrdinů, které je schopen hrát. Měl by si vybrat hrdinu s unikátní rolí. Měl by také vybírat hrdinu, proti kterému má nepřítel slabost. Proto potřebuje uhádnout role nepřátelských a spojeneckých hrdinů a linky, na kterých budou hrát.

Analýza rolí spoluhráčů může být ulehčena, pokud agenti se svými spoluhráči komunikují. Pro dobrou analýzu musí mít agent přístup ke znalostní bázi všech hrdinů a aspoň k nějakým informacím o nepřátelském týmu. Bez těchto znalostí vybírá agent pouze podle své preference.

Analyzovat nepřátelskou roli pouze na základě hrdiny je obtížné. Nikdy nevíme, co přesně chce nepřítel s hrdinou dělat. Pro dobrou analýzu hrdiny by znalostní báze musela analyzovat obrovské množství zápasů, ze kterých by statisticky mohla předpovědět chování hrdinů.

Začátek hry

Po výběru se hrdinové objeví ve svých základnách. Každý hrdina na začátku dostane zlato, které utrácí za své první předměty. Už první nákup by měl reflektovat hrdinovu roli na lince. Například carry si kupuje předměty na poškození, aby se mu lépe zabíjely jednotky.

Poté se hráči přesouvají na svou linku, nebo do lesa. V lese se na začátku příští fáze objevuje první runa, kterou mohou vzít. V této fázi se mohou hráči přizpůsobit nepřátelům, aby měli lehčí další fáze hry. Proto je důležité sbírat informace v okolí nepřátelského lesa a linek.

Například může hráč zjistit, že hraje proti silnému hrdinovi, kterého na své lince nečekal. Na poslední chvíli se ještě může prohodit s jiným hrdinou svého týmu a změnit linku.

Umělá inteligence zde řeší podobný problém jako v předchozí fázi. Opět se pokouší analyzovat nepřátelské jednotky, kterým se snaží přizpůsobit první nákup. Ten by měl odpovídat očekávané síle nepřítele a měl by obsahovat předměty, které doplňují zdraví. Špatný nákup může vést ke smrti v další fázi, čímž nepřítel získá velkou výhodu.

Hra na linkách

Tato fáze začíná, když z báze vyjdou první jednotky. V této fázi se hráči pohybují primárně po svých linkách a snaží se vydělat co nejvíce zlata zabíjením jednotek a nepřátelských hrdinů.

Zabíjení jednotek pro zlato se jinak říká tzv. farming. Farming je velkou výzvou této a následující fáze. Pokud je farmení efektivní, představuje větší zdroj zlata než zabíjení nepřátelských hrdinů.

Hráč dostane zlato za zabitou jednotku pouze tehdy, pokud ji udělí poslední úder (tzv. last-hit). Hráč musí dobře odhadnout chvíli, ve kterou na jednotku zaútočit. To je jedna z nejtěžších dovedností, které se hráči musí naučit.

Last-hitting je ztížen mnoha faktory. Jelikož se touto dovedností budeme zabývat při implementaci agenta, tak tyto faktory blíže specifikujeme:

- **Životy a odolnost** — hráč musí útočit teprve tehdy, když má jednotka málo životů.
- **Nepřátelé v okolí** — pokud na jednotku někdo útočí, je složitější odhadnout správnou chvíli k útoku. Zároveň se hráč nesmí vystavovat zbytečnému ohrožení.
- **Poškození** — hráč musí odhadnout, kolik poškození jednotce udělí jedním útokem.
- **Rychlost útoků** — někteří hrdinové útočí velmi pomalu.
- **Pozice** — hráč nesmí stát moc daleko od jednotek (dostřel).

Existují zároveň způsoby, jakými se dají některé z faktorů zneužít. Příkladem jsou techniky, které se používají pro ovládání jednotek. Hrdina může například pomocí jednotek jednoho z táborů odlákat jednotky z linky. Tím nepřátelé přijdou o zlato a zkušenosti, které by jejich zabitím mohli získat.

Agent by měl mít dobrou představu o lince a místě, na kterém má stát. Špatné postavení pro něj může znamenat smrt, které se snažíme především v této fázi vyhnout. Výhody získané v této fázi hry už nebývají v dalších fázích překonány (viz [11]).

Střed hry

Střed hry obvykle nastává poté, co hráči získali základní vybavení a není už pro ně výhodné déle zůstávat na linkách. Hrdinové začnou trávit více času v lese, kde zabíjejí neutrální jednotky.

Pro tuto fázi je typické, že začínají týmové souboje. Cílem soubojů je především zničit nepřátelskou věž na některé z linek a dostat vlastní jednotky co nejblíže nepřátelské bázi.

V této fázi je hlavním problémem agentů strategické a taktické rozhodování. Takticky a strategicky musí jednotky uvažovat při soubojích. Strategicky v rámci týmu nad tím, kde mohou zaútočit nebo kde je třeba obrany.

Souboje samotné jsou obrovským problémem. Agenti musí vnímat svou roli v rámci týmu a místo, na kterém by se v rámci souboje měli držet. Agenti musí

pracovat jako součást týmu a neustále analyzovat situaci, aby mohli ve správný čas reagovat.

Při souboji je potřeba vzít v potaz schopnosti všech zúčastněných hrdinů. Musí být zváženy všechny jejich schopnosti a schopnosti jejich předmětů. Řešení navíc musí být nalezeno ve zlomku času, protože správné načasování je v soubojích klíčové.

Závěr hry

V závěru hry mají hrdinové všechny zbraně a předměty, které potřebují. Linky už bývají bez věží a bojuje se o báze samotné. V této fázi může jeden prohraný souboj rozhodnout celou hru. Proto je tato fáze definována týmovými souboji.

Oproti středu hry není mezi týmy tolik rozdílů. Jelikož už většinou mají všichni většinu potřebných předmětů, tak v soubojích rozhoduje především správné načasování. Aby tým získal výhodu nad nepřátelským týmem, snaží se v této fázi porazit Barona, z kterého jeden hrdina získá život navíc.

Souboje jsou náročnější především kvůli množství předmětů, které hrdinové v této fázi vlastní. Každý může hrdinovi přidat jinou schopnost, kterou hráč musí zvážit.

4.3 Přístupy k AI

Následuje souhrn přístupů, které jsou používány při tvorbě umělých agentů. Při psaní souhrnu jsme se inspirovali prací, která analyzuje umělou inteligenci v žánru MOBA (viz [12]). Detailnější popis všech zmíněných technik lze nalézt v knize Artificial Intelligence for Games [13].

Rozhodovací stromy

Rozhodovací stromy jsou jedním z nejvíce triviálních přístupů.

Pro každého agenta se vytvoří strom. Každý vrchol vyjma listů reprezentuje nějakou otázku. Každá větev vycházející z vrcholu představuje kladnou nebo zápornou odpověď. V listech se nachází akce, které může agent uskutečnit.

Stav světa je reprezentován jako znalostní báze. V každém kroku získáme agentovu reakci tak, že budeme procházet agentův strom a ptát se báze na otázky ve vrcholech. Po každé odpovědi pokračujeme větví, která ji odpovídá. Agentovu reakci získáme z listu, do kterého se průchodem dostaneme.

Výhodou tohoto přístupu je, že se dá rozhodování agenta dobře vizualizovat. Nevýhody jsou zřejmé při rozšiřování, kde může být nutné přepisovat celý strom. Strom zároveň může být moc velký, pokud zachycujeme složité chování.

Rozhodovací strom je velice zajímavé řešení pro menší hry. Hry žánru MOBA jsou pro něj ovšem moc komplexní.

Behaviorální stromy

Behaviorální strom představuje orientovaný acyklický graf, který obsahuje různé typy vrcholů. Vrcholy mohou představovat akce nebo podmínky, které se

týkají okolního světa. Podmínky jsou podobné otázkám rozhodovacích stromů, kterým se behaviorální stromy podobají. Všechny vrcholy mohou mít více rodičů.

Výhody behaviorálních stromů jsou především v tom, že umožňují použit vrchol na více místech zároveň. Proto jsou dobře rozšířitelné a vyžadují méně paměti. Nevýhodou je opět škálovatelnost při velkém počtu akcí. Ta je ovšem výrazně lepší než u rozhodovacích stromů.

Behaviorální stromy bývají často rozšiřovány, aby do svého rozhodování zahrnovaly emoce. To může být použitelné při tvorbě agentů, kteří se chtějí podobat lidskému hráči.

Konečné automaty

Konečné automaty představují o něco lepší přístup. Agentovi jsou přiřazeny stavy, které popisují jeho aktuální vztah k hernímu světu. V rámci každého stavu má hrdina definováno své chování. Mezi těmito stavy je přecházeno v reakci na změny herního světa.

Konečné automaty se velice jednoduše navrhují a nejsou náročné na výkon. Proto bývají velice často používány. Jejich nevýhodou opět je, že se s nimi špatně pracuje při návrhu složitějších systémů. Pro návrh jednodušších agentů jsou použitelné.

Strojové učení

Odlišný přístup k vývoji agentů zaujímá strojové učení. Místo vytváření obrovských sad pravidel je vytvořeno prostředí, ve kterém se agent pravidla naučí „sám“. Obvykle je však nutné věnovat zvýšené úsilí vytvoření prostředí a samotnému učení, které musí být dobře parametrizováno. Většinou je tedy strojové učení složitější než použití některé z předchozích metod. Odměnou mohou být agenti, kteří reagují plynuleji.

Pro žánr MOBA se nabízí například genetické algoritmy. Ty jsou typicky uplatněny na řešení menšího množství problémů, pro které můžeme vytvořit parametrizovaný odhad řešení. Těchto odhadů se vytvoří obrovské množství (populace).

V každém kroku se některá řešení vyberou, zmutují, zkříží a promíchají s původními odhady. Tím vznikne nová množina odhadů, které mohou být o něco lepší. Odhady hodnotíme podle toho, jak dobře řeší zadaný problém. Po provedení mnoha kroků vrátíme nejlepší odhad jako výsledek.

Pro našeho agenta by například dávalo smysl parametrizovat výběr jednotky, na kterou je výhodné zaútočit. Mohli bychom navrhnout váhy, kterými bychom násobili jejich zdraví, sílu a další vlastnosti. Váhy bychom se snažili nastavit tak, abychom udělili poslední úder co nejvíce jednotkám.

Dalším přístupem je tzv. reinforcement learning. Agent je umístěn do prostředí a jsou mu předloženy různé situace, které musí řešit. Každá situace je zhodnocena hodnotící funkcí. Hlavním cílem agenta je maximalizovat hodnotící funkci napříč všemi rozhodnutími. Agentovo chování bývá opět parametrizováno.

Zajímavou metodou strojového učení jsou neuronové sítě. U neuronových sítí je rozhodování rozděleno do libovolného počtu vrstev. Každá vrstva vezme vstupní hodnoty z předchozí vrstvy, které vynásobí vahami, aplikuje na ně něja-

kou funkci a pošle je další vrstvě. Počet výstupních a vstupních hodnot se může lišit.

Pokud neuronové sítě vhodně nastavíme (učením), umožní nám vytvořit vztah mezi vstupními a výstupními hodnotami. Pomocí sítí můžeme vytvořit chování agenta závislé třeba na několika důležitých vlastnostech jeho okolí, které jdou číselně reprezentovat. Síť učíme tak, aby na výstupu vracela například souřadnice místa, na které se má agent pohnout.

Nevýhodou neuronových sítí je, že se v nich špatně hledá chyba. Pokud se chovají špatně, musí se většinou učit jinak nebo se mění jejich struktura. Na druhou stranu může být jistá nepředvídatelnost také výhodou. Neuronové sítě se hodí pro specifické úkoly, které se špatně řeší algoritmicky.

4.4 Související práce

Analýzou rolí hrdinů se zabývali Nuangjumnong a Mitomo [14]. Ve své práci objevili, že existuje korelace mezi vůdčími dovednostmi hráčů a rolí, kterou si ve hře vybírají.

Analýzou kooperace v rámci týmů a jejího vlivu na výsledek hry se zabývalo několik prací. Pobiedina et al. [15] statisticky analyzovali výkony týmů v Dotě 2. Snažili se najít korelaci mezi státy, ze kterých hráči pochází, jejich zkušenostmi a výsledkem hry. Závěrem práce bylo, že týmová kooperace má větší vliv na výsledek hry než soutěživost.

Drachen et al. [16] analyzují vliv umístění hráčů v čase na výsledek hry. Herní mapa je rozdělena na zóny. Jsou sledovány přechody hráčů mezi zónami a jejich vzdálenost od zbytku týmu. Závěrem je, že týmy mají větší šanci vyhrát, pokud využívají celou mapu a drží pospolu. Je navíc ukázáno, že horším hráčům toto chování chybí.

Podobnou práci napsali také Yang et al. [11]. Ta se snaží reprezentovat průběh hry pomocí grafů. Nad grafy se snaží pomocí grafových metrik a rozhodovacích stromů nalézt situace, které pomáhají týmům k vítězství. Práce konstatuje, že na výsledek hry mají vliv všechny role hrdinů a prezentuje situace, které určují výsledek hry.

Analýzou fází hry a problémů, které musí hry žánru MOBA řešit, se zabýval Silva a Chaimowicz [17]. Ve své další práci [4] se zabývali návrhem agenta pro League of Legends. Agent při navigaci používal mapy vlivu. Pokud byl umístěn na mapě sám, byl schopen vyhrát hru beze smrti. Prokázal také zvládnutí farmení.

Batsford [18] se pokusil o nalezení nejoptimálnějších cest po lesích s ohledem na získané zlato. Cesty jsou hledány pomocí neuronových sítí a genetických algoritmů v umělém prostředí, které reprezentuje les Doty 2. Autor byl schopen nalézt optimální cesty, které ovšem neprezentoval v herním prostředí.

Kvůli své jednoduchosti se pro implementaci agentů v skriptovacím prostředí Doty 2 nejčastěji používají konečné automaty. Jako příklad můžeme zmínit tým agentů, kteří se soustředí na rychlý postup linkami (viz [19]). Dota 2 Wiki [7] dále naznačuje, že konečné automaty používají také nativní agenti Doty 2. Žádná z nejznámějších her žánru MOBA ovšem nezveřejnila svůj přístup k inteligenci agentů.

Evoluční algoritmy použil Kolwankar [2] při implementaci agenta pro vlastní zjednodušenou verzi Doty. Evoluční algoritmy byly použity na úpravu parametrů,

které řídily základní chování agenta. Autor konstatuje, že se chování agentů časem zlepšovalo.

Willich [10] použil reinforcement learning při vývoji agentů pro Heroes of Newerth. Agenty hodnotil podle množství získaného zlata. Agenti se pohybovali pouze v malé oblasti mapy a měli výrazně omezen počet akcí, které mohli provádět. Autor dosáhl průměrných výsledků. Konstatoval, že Lua není dobrým jazykem pro strojové učení, a že příčinou může být odezva mezi hrou a algoritmem.

Mezi momentálně nejúspěšnější implementaci umělých agentů patří agenti vytoření společnosti OpenAI (viz [20]). Ti vytvořili jediného agenta, který byl schopen porazit profesionálního hráče. Při implementaci agenta bylo použito strojové učení. Agenti byli hodnoceni na základě zdraví, zlata a jiných klíčových parametrů hry.

Vývojáři bohužel nepublikovali rozhraní, které používali při vývoji agenta a nebyly známé ani detaily implementace. Ovšem nedávno publikovali další článek (viz. [21]), který agenty lépe popisuje. OpenAI se dále pokouší o vytvoření týmu, který bude schopen porazit profesionální tým lidských hráčů. Už nyní sklízí první úspěchy a byli schopni porazit tým průměrných hráčů.

OpenAI staví na Dota 2 Scripting API. Jejich přístup je založen na využití velkého výpočetního výkonu pro dlouhodobé učení agentů (agenti se učí cca 300 let hry za den). Agent samotný by měl být reprezentován jako neuronová síť, která svými výstupy určuje agentovu akci a její parametry.

Aby byli agenti výzvou, bývá potřeba obtížnost agentů měnit v závislosti na hráčově úrovni. Úpravou obtížnosti agentů se zabývali Silva et al. [22]. Navržený systém byl schopen adaptovat obtížnost v reakci na výkon protihráče. Výkon hráčů byl měřen podle počtu zničených věží, úrovně hrdiny a počtu smrtí.

Dota 2 a žánr MOBA byl v posledních letech předmětem několika výzkumů. Jak jde vidět, výzkumů zabývajících se umělou inteligencí je málo. Je patrné, že žánr MOBA je stále mladý jak v herním světě, tak ve světě akademického výzkumu.

5. Návrh agenta

V této kapitole se zabýváme návrhem umělého agenta. Nejdříve se zamýšlíme nad metodami z minulé kapitoly a volíme vlastní přístup k implementaci agenta. Ten je založen na mapách vlivů a teorii užitku, které v této kapitole stručně představujeme. Cílem této kapitoly je vybudovat základ, na kterém budeme implementovat agenta.

5.1 Návrh implementace

Při implementaci agenta se potýkáme se dvěma hlavními problémy. Řešíme agentův pohyb po mapě a jeho přemýšlení.

Rozhraní předává agentovy příkazy addonu, která je vykonává ve hře. Při navigaci agentů to znamená, že nemusíme řešit hledání nejkratší cesty, které za nás řeší hra. Zároveň se nemusíme aktivně starat o vlastnosti prostředí, které by nás při pohybu omezovaly.

Pokud se ovšem chceme pohybovat inteligentně, potřebujeme vhodně reprezentovat mapu ze strategického hlediska. Nad vhodnou reprezentací poté může agent přemýšlet. Důležité je například zachytit nebezpečí, které agentům hrozí na nějakém místě mapy.

Pro reprezentaci mapy, která by nám umožnila strategicky analyzovat prostředí, jsme se rozhodli použít tzv. mapy vlivu. Ty se používají například v RTS hrách a byly použity také v jedné z citovaných prací (viz [4]). Mapy jsou většinou reprezentovány v poli, ve kterém každému políčku přiřazují hodnotu. Hodnota reprezentuje vliv zvolené vlastnosti okolních objektů na dané políčko.

Mapy vlivu se nám mohou hodí, pokud chceme reprezentovat vliv jednotek na základě jejich síly, pohybu nebo životů. Jejich hlavní využití je při strategické a taktické evaluaci, používají se ovšem také k navigaci. Například pokud chce agent zjistit, jak moc je ohrožen nebo do jaké pozice se má pohnout.

Jelikož je žánr MOBA založen na žánru RTS, jsou i mapy vlivu použitelné. V Dotě 2 nám umožní dobře reprezentovat vliv dynamických, polo-dynamických a statických objektů na herní mapu. To je hlavní důvod, proč jsme se je rozhodli použít při tvorbě agentů.

Bez použití map bývá složité prostor analyzovat. Často se provádí složitější analýza na začátku hry nebo je agent závislý na reprezentaci, kterou pro něj vytvoří vývojář. Mapy vlivu nejsou výpočetně složité a jsou schopné pracovat i s hodně objekty v prostoru.

Pro agentovo myšlení jsme mohli použít kterýkoliv z přístupů zmíněných v předchozí kapitole. Nerozhodli jsme se použít rozhodovací stromy, behaviorální stromy, ani konečné automaty. Všechny jsou špatně škálovatelné a jdou jednoduše implementovat ve skriptovacím rozhraní Doty 2.

Strojové učení také nepřichází v úvahu. Učení je závislé na tom, že agenti odehrají velké množství her. Jelikož musíme se hrou komunikovat pomocí požadavků, nejsme schopni hru dostatečně zrychlit. Zároveň postrádáme výkon a prostředky, které by učení vyžadovalo (viz [21]).

Pro myšlení agentů jsme se rozhodli použít teorii užitku. Teorie užitku charakterizuje každé možné rozhodnutí sadou funkcí, které na vstupu pracují s infor-

macemi o herním světě. Výstup jednotlivých funkcích představuje užitek, který je dané informaci přidělen, vzhledem k danému rozhodnutí. Pro vybrání správného rozhodnutí se všechny užitky zkombinují a vybere se „nejužitečnější“ rozhodnutí.

Výhodou teorie užitku je především schopnost dobře zachytit vztahy mezi vstupy a výstupy jednotlivých rozhodnutí. Díky reprezentaci pomocí funkcí nemusí přechody mezi rozhodnutími působit tak deterministicky, jako například u konečných automatů. Teorie užitku se také umí dobře vyrovnat s nedeterministickým světem.

Zároveň nevíme o žádné práci, která by teorii užitku použila v podobné hře. Tím pádem může být její použití zajímavé z hlediska výzkumu AI.

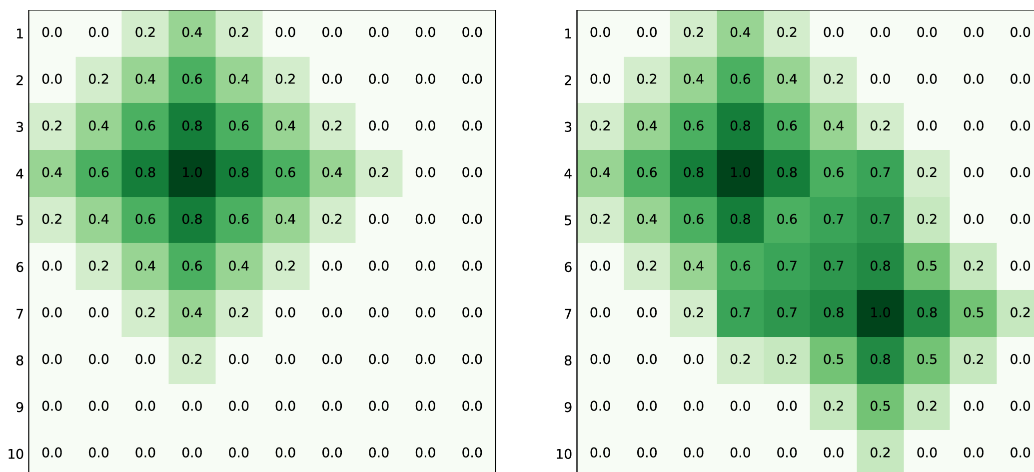
5.2 Mapy vlivu

V této práci nerozlišujeme mezi mapami vlivu (Influence maps) a potenciálovými poli (Potential fields). Oba pojmy bývají často zaměňovány, přičemž v robotice je preferován název potenciálové pole a ve hrách mapy vlivu. V této práci preferujeme označení mapy vlivu.

Pro potřeby této práce bude stačit pouze zjednodušený popis. Detailnější informace, které obsahují popis map, šíření a kombinování vlivu, napsal Mark (viz [23]).

V typické mapě je každé políčko reprezentováno souřadnicemi. Mapy vlivu se snaží tuto reprezentaci rozšířit. Především se snaží do mapy přidat hodnoty, které reprezentují vliv objektů na prostor.

Vlivem rozumíme schopnost objektu uplatnit zvolenou vlastnost v prostoru. Tu se snažíme zachytit pro každé políčko mapy. Typicky volíme právě jednu vlastnost objektů, kterou se snažíme v mapě vlivů co nejpřesněji reprezentovat.



Obrázek 5.1: Vliv dosahu
Mapa vlevo zachycuje vliv jednoho objektu, který za jednotku času urazí až 4 políčka. Mapa vpravo byla získána sečtením vlivů dvou objektů.

Například můžeme chtít zachytit schopnost objektu přesunout se v určitém čase na jiné místo. Políčka s vyšším vlivem budou odpovídat políčkům, na které se může objekt přesunout rychleji. Vliv bude slábnout se vzdáleností od objektu (viz Obrázek 5.1).

Z příkladu je patrné, že jsme závislí na tom, jak vliv reprezentujeme, jak jej šíříme a na tom, jak interpretujeme výsledek.

Mapu vlivu pro zvolenou vlastnost získáme tak, že po poli rozšíříme vliv všech uvažovaných objektů, který zkombinujeme s ostatními vlivy. Pokud pracujeme s dynamickými objekty, typicky bývá mapa v každé iteraci generována znovu.

Reprezentace vlivu

Vliv může být reprezentován v 2D i 3D prostředí, které může být členité a mít více vrstev. Nám bude stačit 2D prostředí, protože se dá herní mapa Doty 2 zachytit jako 2D pole. Pokud by ale mapa obsahovala například tunely, pak bychom museli k mapě i šíření vlivu přistupovat odlišně.

Vliv reprezentujeme pomocí hodnoty, kterou přidělíme každému políčku. Typicky se jedná o reálné číslo, které se může pohybovat v omezeném intervalu. Často také vliv normalizujeme (např. do intervalu [0, 1]). To může být výhodné, pokud kombinujeme více map vlivu dohromady (viz dále).

Existují také dva přístupy k tomu, co do mapy uložit. Můžeme každému políčku přiřadit více hodnot, abychom zachytili více vlastností. Nebo můžeme vytvořit více map, kde každá ukládá jednu hodnotu, jejichž kombinací získáme vliv vzhledem k více vlastnostem.

Šíření vlivu

Pro šíření vlivu se používá několik přístupů. Vždycky se snažíme zachytit šíření vlivu co nejlépe, ale pokud máme méně času na výpočty, musíme vzít také v potaz efektivnost některých metod.

Jedním z nejjednodušších přístupů je postupně procházet všechny políčka mapy. Pro každé políčko počítáme vliv, který na něj mají všechny objekty. Vliv objektů se reprezentuje jako funkce, která bývá založena na vzdálenosti od objektu a snaží se co nejlépe reprezentovat jeho šíření. Nevýhodou je, že musíme počítat funkci pro každé políčko, tudíž i vzdálenost mezi políčkem a objektem. Zároveň uvažujeme i objekty, které mají na políčko zanedbatelný vliv.

Předchozí přístup se často obrací. Místo procházení všech políček procházíme všechny objekty. Vliv šíříme jen v omezeném okolí objektů. Na větší mapě, kde mají objekty vliv na malé vzdálenosti, je tenhle způsob daleko efektivnější. Umožňuje nám lépe pracovat s okolím agentů. Důležitá je ovšem volba velikosti okolí, do kterého vliv šíříme. Pokud je zvolené okolí moc malé, nebude v mapě obsažena potřebná informace.

Dalším přístupem je začít na políčku objektu a z něj se šířit do okolních políček (vlnový algoritmus). Musíme si ukládat políčka, kterými se šíříme, což klade větší nároky na paměť a režii. To se projeví hlavně pokud se šíříme na velkou vzdálenost. Výhodou je, že můžeme vliv šířit s ohledem na neprůchozí terén nebo jiné omezení. Vzdálenost, na kterou vliv šíříme, může být jednoduše upravena.

Poslední často používaný přístup využívá rozmazávání. Na pozice objektů v mapě vložíme jejich maximální vliv. Poté mapu rozmazáváme, jako při práci s obrázkem (na to můžeme použít třeba „Gaussian blur“). Výhodou je, že nemusíme mapu čistit, ale můžeme v každé iteraci rozmazávat všechny vlivy. Vlivy

budou s postupem času slábnout, až vymizí úplně. V okolí objektů budou vlivy vždycky silné.

Hlavní výhodou tohoto přístupu je, že uchovává představu o tom, co se na mapě dělo v minulosti. Nevýhodou je, že se chová stejně ke všem objektům. Z pohledu objektů se dá ovlivnit pouze jejich maximálním vlivem a parametry rozmazávání.

Kombinování map a vlivů

Při šíření vlivu po mapě se musíme zamyslet nad tím, jak vlivy jednotlivých objektů zkombinovat dohromady. Mezi nejčastější metody kombinace patří sčítání, násobení nebo výběr maximální hodnoty. Výběr správné kombinace záleží na použití výsledné mapy.

Sčítání hodnot je výhodné, pokud potřebujeme ve výsledné mapě hledat maximální vliv (viz Obrázek 5.1). Občas může být vliv v některých hrách definován nejsilnější jednotkou v okolí. Poté můžeme preferovat například kombinaci výběrem větší z hodnot.

Kombinováním a šířením vlivu jsme vypočítali mapu, která v sobě uchovává vliv nějaké vlastnosti na prostor. Typicky ale potřebujeme vzít v potaz více vlastností najednou a vytvořit z nich jednu mapu. Většinou se pro všechny uvažované vlastnosti tvoří mapy, jejichž kombinací se vytvoří nová mapa, která zachycuje vliv více vlastností.

Například pro reprezentaci ohrožení můžeme zachytit vliv vlastních jednotek a vliv nepřátelských jednotek, který bude založen třeba na jejich pohybu. Odečtením vlivu našich jednotek od vlivu nepřátel získáme mapu ohrožení vzhledem k oběma vlivům. Znaménko nám bude udávat, kdo má na políčko větší vliv.

Pro kombinace map se používají stejné metody jako pro kombinování vlivů. Často se před kombinací ještě mapy násobí vahami nebo se normalizují do nějakého intervalu. Bývá zvykem mapy kombinovat tak, aby znaménko výsledné mapy určovalo, jestli je pro nás vliv příznivý nebo ne. Nevýhodný vliv by měl být záporný (např. políčko ohrožené nepřítelem).

Shrnutí

Mapy vlivu dobře zachycují vliv objektů v prostoru. Své hlavní uplatnění nachází především ve chvílích, kdy existuje hodně uvažovaných objektů. Stačí nám pouze rozšířit jejich vliv a ten vhodně nakombinovat.

Map vlivu se typicky ptáme na vliv, který se nachází na některých místech mapy nebo na místa s velkým vlivem. Při strategickém myšlení jsou často důležité přechody mezi vlivy (např. změny znaménka).

Mapy vlivu byly úspěšně použity při navigaci v RTS hrách, s ohledem na strategii a taktiku. Hagelbäck a Johansson [24, 25] použili v jednoduché strategické hře mapy vlivu pro vyhýbání se překážkám, útokům na nepřátelské báze a k sběru surovin.

Uriarte a Ontañón [26] použili mapy vlivu při tzv. kitingu. Při něm jednotka zaútočí na nepřítele a uteče rychleji než nepřítel stačí zareagovat. K tomu je potřeba, aby se držela na kraji dostřelu nepřítele. Mapy vlivu byly použity také k tomu, aby se jednotky držely dále od překážek.

Mapy vlivu už byly také použity při vývoji umělé inteligence pro žánr MOBA (viz [4]).

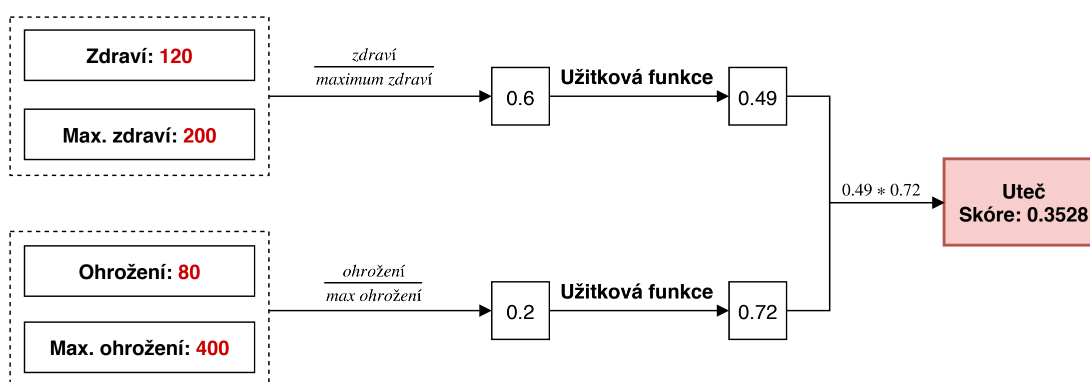
5.3 Teorie užitku

V této práci přistupujeme k teorii užitku na základě dvou přednášek. První přednáška, kterou přednášeli Mark a Dill [27], se zabývala úvodem do teorie užitku. Druhá přednáška, kterou přednášeli Mark a Lewis [28], se zabývala vývojem systému založeného na teorii užitku a mapách vlivu pro hru Guild Wars 2: Heart of Thorns.

Hlavní myšlenkou teorie užitku je charakterizovat každé možné rozhodnutí číselnou hodnotou. Této hodnotě poté říkáme očekávaný užitek. Jejím úkolem je změřit, jak moc je pro nás dané rozhodnutí výhodné vzhledem k aktuálnímu stavu světa.

Při rozhodování se snažíme ohodnotit všechny rozhodnutí, které můžeme uskutečnit. Nakonec vybereme rozhodnutí, které má nejvyšší očekávaný užitek. Tedy takové rozhodnutí, které odpovídá výhodné akci v konkrétním stavu hry.

Rozhodnutí



Obrázek 5.2: Ukázkové rozhodnutí
Rozhodujeme se, jestli utéct. Rozhodnutí je ovlivněno zdravím a ohrožením agenta.

Rozhodnutí představuje akci, o jejímž provedení uvažujeme. Každé rozhodnutí je složeno z tzv. úvah (considerations). Úvahy odpovídají stavům světa nebo agenta, které je při rozhodování nutno uvažovat. Očekávaný užitek rozhodnutí je závislý na užitcích jednotlivých úvah.

Každé rozhodnutí by mělo být složeno z co největšího počtu úvah. Čím více úvah uvažujeme, tím lépe zachytíme užitek rozhodnutí. Na úvahy se také můžeme koukat jako na preferenci. Data potřebné pro úvahy daného rozhodnutí a stav světa nazýváme kontextem rozhodnutí.

Abychom vypočítali užitek, který jednotlivé úvahy přinášejí celému rozhodnutí, definujeme funkci užitku. Funkce užitku vyjadřuje vztah mezi uvažovanou částí kontextu rozhodnutí a užitkem úvahy.

Užitky jednotlivých úvah nakonec vhodně kombinujeme, abychom spočítali očekávaný užitek rozhodnutí. Nejčastěji se užitky úvah kombinují násobením nebo

jejich průměrem. Typicky chceme dostat normalizovaný užitek, se kterým se lépe pracuje.

Například se agent může rozhodovat, jestli uteče zpátky do báze. Při rozhodování uvažuje svůj počet životů a v jakém je ohrožení. Očekávaný užitek útěku bude vysoký, pokud bude agent ohrožen a bude mít málo zdraví. Naopak pokud nebude ohrožen, nemusí s útěkem nutně pospíchat a užitek rozhodnutí bude o něco menší (viz Obrázek 5.2).

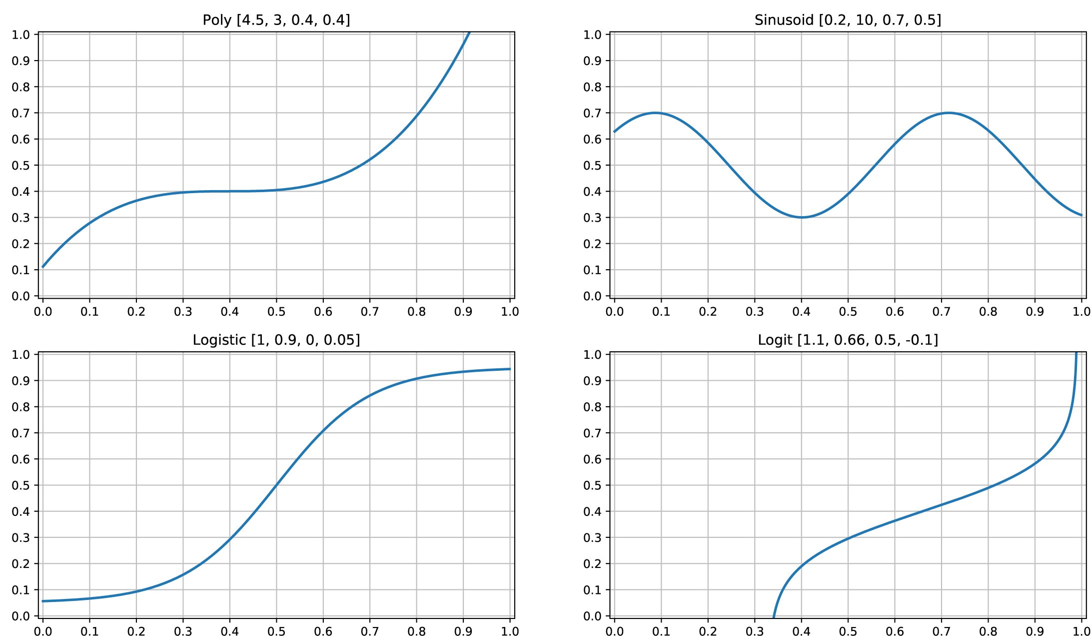
Po ohodnocení všech uvažovaných rozhodnutí vybíráme to nejlepší. Občas se také náhodně vybírá některé z nejlepších řešení. Tím může být rozhodování o něco méně předvídatelným.

Také se zavádí tzv. faktory, které mají význam především pokud jsou vlivy normalizované. Faktor je hodnota, kterou násobíme výsledný očekávaný užitek. To nám umožní preferovat některá rozhodnutí (např. útěk před stáním na místě).

Funkce užitku

Úkolem funkce užitku je spočítat užitek jedné z úvah. Funkce bývají definované vývojářem, který se jimi snaží zachytit užitek úvahy v závislosti na stavu světa.

V jedné z přednášek Mark [28] zmínil, že je výhodné výstupy funkcí užitku normalizovat a jejich předpisy parametrizovat. Úkolem normalizace je převést výstupní argumenty do nějakého předem definovaného intervalu (viz Obrázek 5.2). To nám umožňuje definovat všechny funkce na stejném intervalu. Parametrizace nám umožní funkce snadno definovat a lépe je vizualizovat.



Obrázek 5.3: Základní funkce
Průběh funkcí je zachycen na základním intervalu [0,1]. Parametry odpovídají čtveřici [m, k, c, b].

Funkce se dají parametrizovat pěticí parametrů (viz Obrázek 5.3). Prvním parametrem je typ funkce. Následují čtyři parametry, které mají vliv na tvar křivky a její umístění. Pro každou úvahu nám stačí zvolit vhodný typ funkce

a jejich parametrů. To se může hodit také ve chvíli, kdy definice funkcí ukládáme do konfiguračních souborů, kde parametry zvětšují čitelnost.

Základní parametrizované funkce mohou vypadat následovně:

- Polynomial: $\boldsymbol{m(x-c)^k+b}$
- Sinusoid: $\boldsymbol{m\sin(kx+c)+b}$
- Logit: $\boldsymbol{0.5\log_{100^m}\frac{z}{1-z}+b+0.5}$, kde $\boldsymbol{z=\frac{x}{k}-c}$
- Logistic: $\boldsymbol{k\frac{1}{1+e^{-z}}+b}$, kde $\boldsymbol{z=10m(x-c-0.5)}$

Funkce výše jsou upravené tak, aby se pohybovaly v okolí počátku souřadného systému. Tyto čtyři funkce představují základní tvary křivek, které můžeme pro zachycení užitku potřebovat. Další důležitou funkcí je binární funkce, která vrací pouze jednu ze dvou hodnot v závislosti na tom, do jakého intervalu vstup patří. Ta může představovat klasickou podmínku (if).

Hlavní výhodou parametrizace je, že se dají jednotlivé funkce velmi snadno upravovat. To usnadňuje tvorbu nástrojů, které mohou být použity pro vytváření rozhodnutí. Nástroje, které se dají vytvořit nad parametrizovanými funkcemi, mohou umožnit tvorbu inteligentních agentů v řádu několika minut (viz [28]).

Shrnutí

Teorie užitku je velice jednoduchý nástroj, který nám umožňuje vytvářet rozhodnutí závislé na velkém množství parametrů. Výhodou jsou malé nároky na výkon — počítání funkcí stojí relativně málo času — a důvěryhodné chování, které je možné získat definováním pár rozhodnutí.

Hlavní výhodou teorie užitku jsou lepší reakce na okolní svět. Oproti některým ze zmíněných přístupů nejsou přechody mezi rozhodnutími tvořeny podmínkami, ale jsou závislé na použitých funkcích. Rozhodnutí tak můžou působit plynuleji a méně skokově. To je dáno tím, že výběr rozhodnutí závisí také na užitku ostatních rozhodnutí, který se v čase mění.

Užitkové funkce, faktory a samotné rozdělení na rozhodnutí a úvahy charakterizují jednu vlastnost teorie užitku — je hodně závislá na parametrizaci. Všechny části musí být dobře definovány a musí do sebe dobře zapadat. Občas je složité dosáhnout optimálních výsledků.

6. Implementace agenta

V této kapitole popisujeme implementaci dvou agentů hrajících Dotu 2. Začínáme popisem obecné architektury agentů a jejich interakce s rozhraním. Dále je kapitola rozdělená na dvě podkapitoly, ve kterých rozebíráme implementaci map vlivu a teorie užitku.

Při implementaci jsme se rozhodli implementovat dva agenty, kteří ovládají dva hrdiny. Hrdinové mají v Dotě unikátní jména. První agent ovládá střeleckou hrdinku Lina, která je závislá na používání kouzel, kterými uděluje největší poškození. Druhý agent ovládá střeleckého hrdinu Snipera. Ten se vyznačuje velkým dostřelem a rychlými projektily.

Rozhodli jsme se, že oba hrdinové budou střelci. To nám zjednoduší práci s mapami vlivu, především při implementaci farmení, protože střelečtí hrdinové nemusí chodit do blízkosti nepřátelských jednotek, aby na ně mohli útočit. Tím je pohyb o něco jednodušší, protože se nemusíme pohybovat v místech, které ohrožují nepřátelské jednotky.

Kdybychom museli chodit blízko k nepřátelům, strávili bychom hodně času parametrizací map vlivu, které v takovém případě musí dobře rozlišovat mezi stavem, kdy se k jednotkám přibližujeme a kdy se naopak snažíme držet dále. To pro jednu z citovaných prací (viz [4]) představovalo problém.

6.1 Architektura agentů

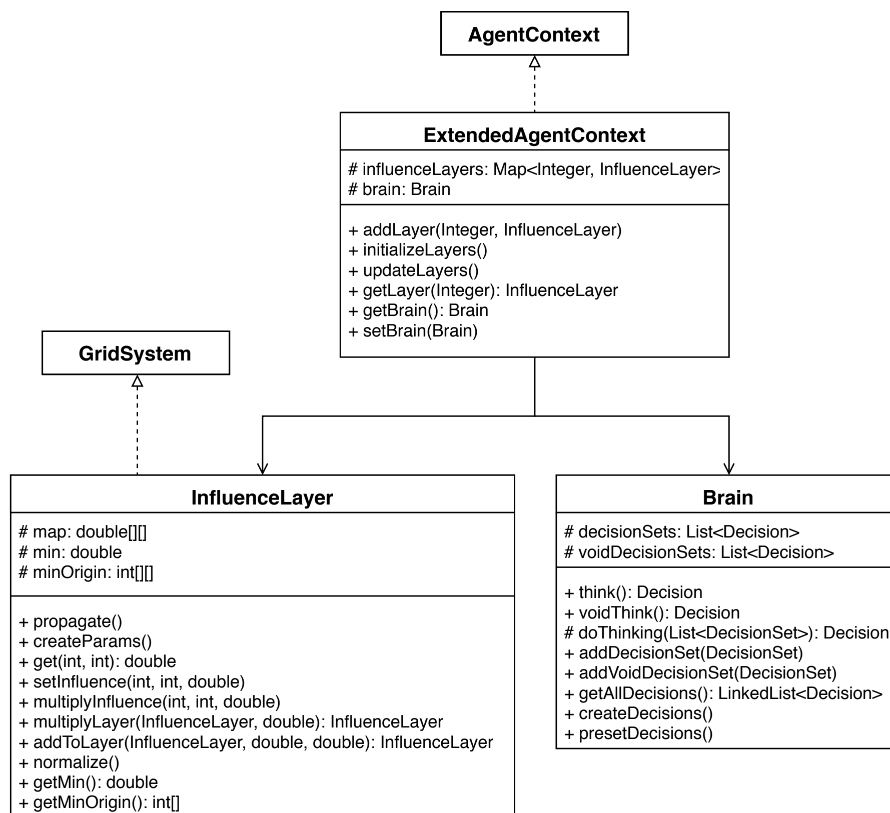
Při tvorbě rozhraní jsme implementovali třídu **AgentContext**. Ta v sobě obsahuje informace o stavu části světa, která je z agentova pohledu důležitá. Prvním krokem bylo rozšířit tuto třídu o reprezentaci map vlivů a agentova rozhodování. Z toho důvodu jsme vytvořili abstraktní třídu **ExtendedAgentContext** (viz Obrázek 6.1), která v sobě ukládá objekty představující mapy vlivu (**InfluenceLayer**) a objekt, který představuje agentovo myšlení (**Brain**).

Třída **ExtendedAgentContext** agentům umožňuje uložit si do svého kontextu vlastní mapy vlivů. Pokud aktualizujeme kontext, aktualizujeme také mapy. To nám umožní kontext dále využít v rozhodnutích, které potřebují znát hodnoty některých map vlivu. Každý agent má více než jednu mapu, přičemž mapy mohou vznikat kombinací jiných map.

Naši agenti jsou reprezentováni třídami, které rozšiřují **BaseAgentController** (viz kapitola 3). Aby agenti nalezli nejvýhodnější akci, kterou mohou provést, používají metodu **think()** třídy **Brain**. Ta reprezentuje myšlení agenta a vrací nejlepší rozhodnutí vzhledem k aktuálnímu kontextu, které interpretujeme jako agentovu akci a předáváme rozhraní.

6.2 Mapy vlivu

Mapy vlivu implementujeme pomocí třídy **InfluenceLayer** (viz Obrázek 6.1). Mapy se snaží co nejlépe reprezentovat původní svět. Kdyby ovšem přesně odpovídaly původnímu světu, kladly by velké nároky na výpočetní a paměťovou složitost operací. Proto využíváme toho, že agentům stačí méně detailní informace (např.



Obrázek 6.1: ExtendedAgentContext
Agentův kontext rozšiřujeme o mapy vlivu (InfluenceLayer) a třídu, která se stará o agentovo myšlení (Brain). Některé metody pro čitelnost vynecháváme.

informace o oblasti, ne bodu). Proto rozšiřujeme třídu **GridSystem**, což nám umožní vytvořit menší mapu a souřadnice mezi mapami přepočítávat.

Při práci s mapami se na ně koukáme jako na vrstvy. Nové vrstvy mohou vznikat kombinací jiných vrstev. Vrstvy spolu můžeme násobit nebo sčítat. K tomu slouží metody **addToLayer** a **multiplyLayer**. Vrstvy musí být definovány nad stejným souřadným systémem, ovšem nemusí mít stejné rozměry ani počátek, protože kombinujeme pouze jejich průnik.

Skládání vrstev nám umožňuje vytvořit složitější vrstvy na základě pár elementárních. Vrstvy také agentům umožňují získávat maximální a minimální hodnoty vlivu a jejich umístění. To používáme především při rozhodování. Například při navigaci vybíráme políčko s největším vlivem v agentově okolí, na které poté agent směřuje (viz dále).

Maximální a minimální hodnoty nám umožňují agentovu mapu normalizovat. Toho využíváme, pokud chceme univerzálně porovnávat vliv s nějakou číselnou hodnotou nebo mapy kombinovat (např. násobením).

Nejdříve jsme se snažili mapy vlivu implementovat pomocí rozmazávání. Úpravou parametrů může tento přístup zachycovat historické informace, které by se nám hodily při taktické analýze celé mapy. Bohužel se rozmazávání ukázalo jako velmi náročné na výkon. To bylo znát především pokud jsme pracovali s velkými mapami.

Dále potřebujeme způsob, jakým rychle změnit okolí nějakých objektů. To je klíčové například v okolí věží, jejichž vliv se musí po změně vztahu k hráči rychle

měnit. Potřebujeme větší kontrolu nad šířením vlivů, kterou nám rozmazávání neposkytuje. Proto jsme se rozhodli použít jiný přístup.

Pokusili jsme se o šíření vlivu v malém okolí objektů a po celé mapě pomocí funkcí vlivu. Při obou metodách jsme byli schopni dobře zachytit vliv objektů na prostor. Funkce nám umožnily vychýlit vliv herních entit směrem k jejich bázi a dobře ovlivňovat klesání vlivu s rostoucí vzdáleností. Přístup byl také dostatečně rychlý a nenáročný pro velké mapy.

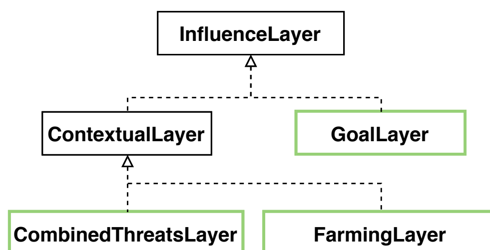
Nevýhodou bylo, že vliv ignoroval neprůchozí políčka, což je problém při navigaci, pokud jde agent vždy ve směru většího vlivu. Oba přístupy totiž vytváří lokální minima v okolí překážek, ve kterých agenti uváznou.

Nakonec jsme se rozhodli šířit vliv po mapě vlnovým algoritmem. Ten nám umožnil správně implementovat navigaci, aniž by se agenti zasekávali v lokálních minimech. U velkých map bylo zřejmé, že je tento přístup o něco pomalejší. Proto jsme se snažili použít mapy vlivu s menším rozlišením, což se ve výsledku osvědčilo.

Během implementace šíření vlivů jsme se pokoušeli o definování použitelných map. Začínali jsme s mapami, které reprezentovali vliv všech objektů na mapě. Výhodou tohoto přístupu bylo, že agent mohl analyzovat stav celé mapy. Zároveň nebylo třeba vytvářet mapy pro jednotlivé agenty, ale stačilo je mezi nimi sdílet. Kvůli velikosti jsme ale museli mapu reprezentovat v malém rozlišení.

Vlastností této reprezentace je, že agenti mají přístup k informacím o celé mapě. Z hráčova pohledu může jejich chování vypadat jako podvádění, pokud prokazují znalost informací, které by neměli mít k dispozici. Například pokud by byli agenti schopni utéct před hrdinou, kterého nevidí. Často se proto reprezentuje informace pouze v agentově okolí.

Rozhraní jsme vytvářeli tak, aby agentovi umožnilo pracovat s informacemi v jeho okolí. Proto jsme se rozhodli, že by toto rozdělení měly respektovat i mapy vlivu. Proto jsme se snažili vytvořit mapy, které pracují pouze s agentovým okolím.



Obrázek 6.2: Používané mapy vlivu
Mapy vlivů, které náš agent používá.

Vrstvu, která pracuje s agentovým kontextem, reprezentujeme abstraktní třídou **ContextualLayer**, která rozšiřuje **InfluenceLayer**. Jejím hlavním úkolem je posouvat mapy vlivů s agentem a případně měnit jejich velikost. Velikost je dána tím, jak daleko agent vidí. Jelikož je mapa vlivu v agentově okolí relativně malá, umožňuje nám zachovat původní rozlišení mapy a uchovávat přesnější informace.

Při implementaci map vlivů jsme se snažili umožnit tři základní agentovy dovednosti: pohyb, farmení a útěk z nebezpečí. Pro každou z dovedností bylo

potřeba vytvořit samostatnou mapu vlivů (viz Obrázek 6.2). Jednotlivé mapy rozebíráme v následujících sekcích.

Navigace

Každý agent má vlastní navigační cíl na mapě, ke kterému směřuje. Navigaci implementujeme jako mapu vlivů **GoalLayer**, která zachycuje vliv cíle. Vliv cíle reflektuje jeho vzdálenost od jednotlivých políček — vliv je nejvyšší v cíli a s rostoucí vzdáleností klesá.

Vliv cílů šíříme po celé herní mapě, abychom byli schopni z každého místa mapy směřovat směrem k cíli. Zároveň mapu reprezentujeme s dostatečně malým rozlišením (1:10). Proto můžeme mapu častěji aktualizovat.

Samotnou navigaci řešíme tak, že agent jde směrem největšího vlivu. K tomu si vybírá místo ve svém okolí, které je od něj dostatečně vzdálené. Pokud bychom vybrali místo příliš blízko, ve hře by se agent na chvíli zastavil. To je způsobeno frekvencí příchozích požadavků typu *update*.

Farmení

Pro správné farmení je potřeba, aby nepřátelské jednotky, které mají málo zdraví, nebyly mimo agentův dostřel. Jednoduchý přístup je postavit agenta do blízkosti jednotky, která je na lince nejdále od naši báze. Tím agenta dostaneme do blízkosti nepřátelských jednotek.

Ovšem tento přístup agenta vystavuje velkému nebezpečí. Pokud hraje proti střeleckému hrdinovi, může být v neustálém ohrožení. Proto je potřeba, aby se hrdina zároveň držel od jednotek co nejdále a nejlépe směrem k naši základně. Je proto potřeba, aby se agent postavil do výhodné pozice a případně odešel z ohrožení. Proto jsme vytvořili mapu vlivu, která se stará o správný pohyb pro optimální farmení.

Při implementaci map farmení jsme se rozhodli použít podobný přístup jako Silva a Chaimowicz (viz [4]). Snažíme se, aby agenti stáli co nejdále od jednotek ve vzdálenosti svého dostřelu. Protože reprezentujeme mapu v menším rozlišení a mezi hrou a rozhraním existuje odezva, rozhodli jsme se agenty postavit o malou konstantu blíže. Tím předejdeme případům, kdy se jednotka posune o malý kousek mimo dostřel a agent nebude schopen zareagovat.

Vliv šíříme do okolí nepřátelských jednotek až na vzdálenost dostřelu našeho hrdiny (bez malé konstanty). Vliv nešíříme na políčka, která jsou ve větší vzdálenosti od naši báze než hrdina. Tím se šíříme pouze do oblasti, která připomíná půlkruh a je orientovaná směrem k naši bázi.

Abychom stáli přednostně u jednotek s nižším zdravím, na které bychom měli útočit přednostně, zahrnujeme do počítání vlivu také počet životů jednotky. Čím méně životů bude jednotka mít, tím větší vliv bude mít na své okolí.

Při počítání normalizujeme vzdálenost d_{tb} políčka od agentovy báze. Při normalizaci používáme maximální $max(d_b)$ a minimální $min(d_b)$ vzdálenosti jakéhokoliv políčka v okolí jednotky, do kterého se šíříme (dostřel). Normalizaci v dalším textu označujeme pomocí čáry nad proměnnou. Vzdálenost políčka od báze nor-

malizujeme následovně:

$$\overline{d_{tb}} = \frac{d_{tb} - min(d_b)}{max(d_b) - min(d_b)}.$$

Vliv jednotky na políčko počítáme na základě životů h jednotky, maximálních životů h_{max} a normalizované vzdáleností $\overline{d_{tb}}$ od agentovy báze. Pro vliv definujeme také jeho maximální hodnotu i_{max}. Vliv na políčko v okolí jednotky spočteme podle:

$$i = \left(1 - \frac{h}{h_{max}}\right)^2 (i_{max} - \overline{d_{tb}}).$$

Pokud sčítáním zkombinujeme všechny vlivy nepřátelských jednotek v okolí našeho hrdiny, dostaneme mapu, jejíž maximum odpovídá ideální pozici, na které by měl hrdina stát. Výsledná pozice bude zároveň orientovaná směrem k naši bázi.

Ohrožení

Pro reprezentaci ohrožení vytváříme dvě mapy. Vlivem zde rozumíme nebezpečnost nepřátelských jednotek a entit, která je dána jejich dostřelem, zdravím a jinými vlastnostmi. Mapou **FriendlyThreatsLayer** reprezentujeme vliv spojeneckých jednotek na agentovo okolí. Ve druhé mapě **EnemyThreatsLayer** ukládáme hrozby nepřátelských jednotek a věží.

Tyto dvě mapy kombinujeme, abychom vytvořili mapu **CombinedThreatsLayer**, která charakterizuje ohrožení jednotlivých políček oběma týmy. Ta vzniká odečtením hrozeb nepřátel od vlivů spřátelených jednotek. Políčka se záporným vlivem jsou ohroženy nepřáteli a naopak. Náš agent se může ptát na znaménko nebo na hodnotu vlivu, která určuje přítomnost a sílu nepřátelských jednotek.

Agent má přístup ke všem třem vrstvám. To využívá například, když se u sebe pohybují jednotky obou týmů. Pak na rozhraní jejich vlivů je z kombinace map původní hodnota ohrožení těžko čitelná, protože se vlivy navzájem vyruší. Agentovi to řekne, že jsou jednotky na obou stranách přibližně stejně silné. Pokud chce ale znát jejich skutečný vliv, může použít jednu z původních vrstev.

Pro správnou implementaci ohrožení musíme rozlišovat mezi jednotkou, hrdinou a věží. Vliv jednotlivých jednotek na prostor se mění podle toho jaký je jejich stav (zdraví, mana) a jaký vztah mají k hráči (útočí nebo neútočí).

Vliv všech tří entit šíříme podobně. Máme jednu funkci, která počítá vliv pro každé políčko v okolí entity, do kterého se vliv šíří. Funkce má podobný tvar pro všechny tři entity. V potaz bere vzdálenost políčka od entity $\overline{d_{et}}$, kterou dělíme maximální vzdáleností d_{max}, do které můžeme vliv šířit (dáno např. dostřelem). Dále uvažujeme normalizovanou vzdálenost políčka od báze $\overline{d_{tb}}$, kterou normalizujeme maximální a minimální vzdáleností od báze, které může políčko dosáhnout v oblasti, do které vliv šíříme.

Pro šíření vlivu také definujeme maximální i_{max} a minimální i_{min} vliv, kterého může políčko dosáhnout. Tyto dvě proměnné jednotlivé entity upravují podle svých vlastností, aby byl vliv dobře reprezentován. Například věže obě hodnoty zvětšují, pokud zrovna útočí na našeho hrdinu. Také můžeme vliv umocňovat pomocí proměnné p, pokud má vliv klesat pozvolněji.

Zde jsou vlivy jednotlivých entit:

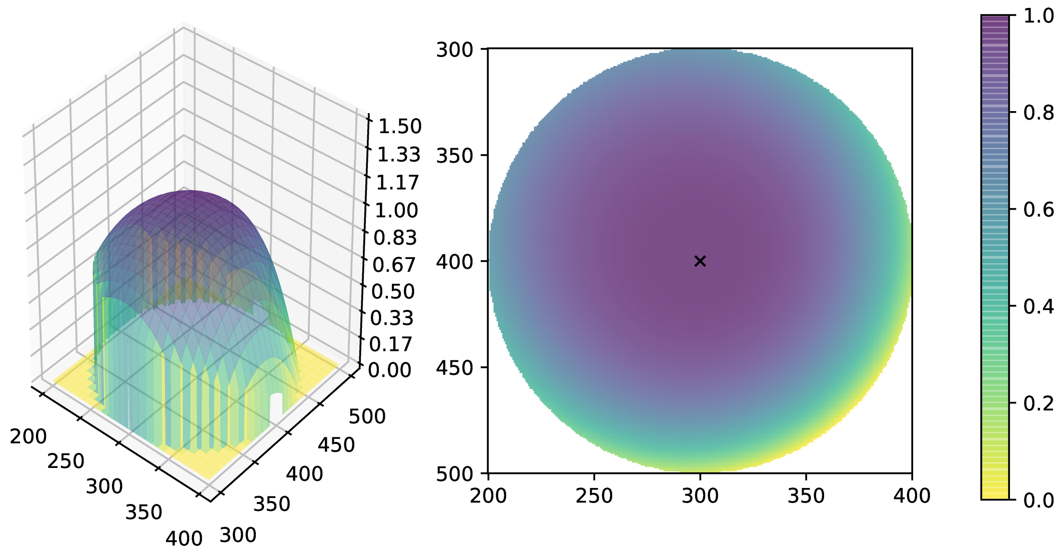
- Hrdinové a jednotky:

$$\boldsymbol{i = \left(i_{max}\left(1 - \frac{\overline{d_{tb}} + \frac{d_{et}}{d_{max}}}{2}\right)\right)^{p}}$$

- Věže:

$$\boldsymbol{i = i_{min} + i_{max} - \left(i_{max}\frac{\overline{d_{tb}} + 3 * \frac{d_{et}}{d_{max}}}{4}\right)^{p}}$$

Šíření vlivu jednotek a hrdinů není složité. Hrdinům nastavujeme větší maximální vliv ($i_{max} = 2$), než jednotkám ($i_{max} = 1$). Maximální vliv přibližně zdvojnásobíme, pokud nepřátelský hrdina nebo jednotka útočí na našeho agenta. To ve výsledku zvedne vliv v okolí agenta, který se dozví, že je v nebezpečí a může se rozhodnout ustoupit.



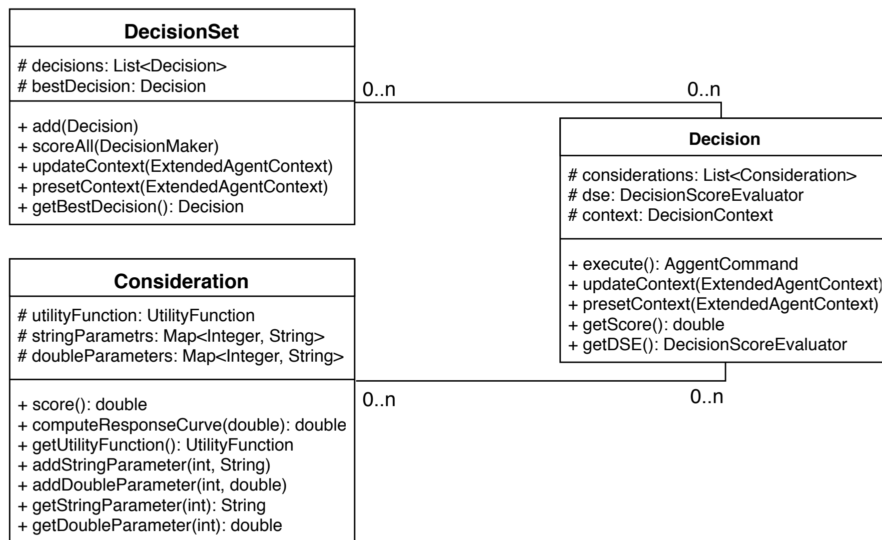
Obrázek 6.3: Ohrožení věží
Báze leží v [0, 0]. Věž leží v [300, 400]. Dále i_{min}=0, i_{max}=1.0, d_{max}=100, kde d_{max} je dostřel věže. Na obrázku vlevo jde vidět, že vliv (ohrožení) věže je vyšší směrem k její bázi. To jde vidět také na obrázku napravo, kde je vliv reprezentován jako obrázek.

Při počítání vlivu věží se nejdříve podíváme, kolik spojeneckých jednotek stojí blíže k věži než náš hrdina. Podle jejich počtu upravujeme i_{max} a i_{min} tak, aby se hrdina mohl přiblížit k věži, pokud stojí před věží aspoň dvě jednotky s dostatečným počtem životů. Pokud je jednotek před věží málo, je vliv věže výrazně vyšší. Nejvyšší je tehdy, pokud věž útočí na našeho hrdinu ($i_{max} = 4$). Pod věž se chceme přiblížit při farmení, pokud pod ní stojí nepřátelské jednotky s málo životy. Základní tvar funkce vlivu pro ohrožení věží je zachycen na Obrázku 6.3.

6.3 Teorie užitku

Teorii užitku implementujeme s abstrakcí, kterou jsme popsali v kapitole 5. Klíčovými jsou rozhodnutí, se kterými pracuje třída **Brain** (viz Obrázek 6.1).

Tato třída ke své práci používá objekty **DecisionSet**, které sdružují rozhodnutí do skupin. Rozhodnutí se lépe definují a sdružují do celků, které řeší podobné problémy. Zároveň nám to umožňuje poskládat (nebo rozšířit) objekt **Brain** z libovolných skupin rozhodnutí, které si agent definuje.



Obrázek 6.4: DecisionSet, Decision a Consideration
Třídy používané při rozhodování. Pro přehlednost byly některé funkce vynechány.

Rozhodnutí představuje objekt **Decision**. Ten je součástí jedné skupiny rozhodnutí a ukládá v sobě objekty třídy **Consideration**, které představují jednotlivé úvahy rozhodnutí (viz Obrázek 6.4).

Uvažování agentů probíhá následovně. Nejdříve se zavolá funkce **think()** třídy **Brain**. Uvnitř se prochází jednotlivé skupiny rozhodnutí. Začíná se aktualizací kontextu rozhodnutí, který je reprezentován objektem **DecisionContext**. Ten v sobě ukládá kontext agenta a zároveň odkazy na objekty, které jsou pro rozhodnutí důležité. To jsou zdroj rozhodnutí a jeho cíl. Zdrojem může být například agent, který útočí na jednotku (cíl).

Poté se začnou vyhodnocovat jednotlivé rozhodnutí. Ty se vyhodnotí zavoláním funkce **getScore()**, která projde jednotlivé úvahy, určí jejich skóre a spočítá celkové očekávané skóre rozhodnutí. Očekávané skóre vypočítáme součinem skóre jednotlivých úvah a vynásobením faktorem rozhodnutí. Nejlepší rozhodnutí napříč všemi skupinami je vráceno jako výsledné rozhodnutí.

Při určování skóre úvah používáme funkce užitku (třída **UtilityFunction**). Třída **Consideration** představuje místo, kde dochází k analýze nějaké číselně dané části světa. Zde je nutno především upravit vstupní hodnoty tak, aby mohly být předány funkci užitku. Proto ve třídě ukládáme také parametry, které jsou použity při normalizaci (např. maximální hodnota vstupu). Příkladem implementované úvahy je třída **ConsiderDistanceToTarget**, která uvažuje vzdálenost k cíli rozhodnutí.

Navigace

Navigace agentů je dána cílem, který agenti uloží do navigační mapy vlivů **GoalLayer**. Proto je potřeba, aby existovaly rozhodnutí, které tuto činnost provádí. V rámci teorie užitku chceme, aby jednotlivé rozhodnutí vykonávaly pouze jednu akci. Rozhodnutí odpovídají agentovým akcím ve hře, ale nastavení cíle se provádí pouze na straně rozhraní.

Faktor	Rozhodnutí	Úvahy	Funkce	Parametry
3	**CreepAsAGoal**	Uplynulý čas	Lineární	[2, 1, 0.6, 0]
		Vzdálenost k cíli	Polynomiální	[8, 3, 0.5, 0.05]
6	**EscapeToBase**	Zdraví	Polynomiální	[7.4, -10, -1, -0.5]
8	**EscapeThreat**	Ohrožení	Sinusoid	[3.6, 1.8, 1.55, 0.2]
5	**FarmingPosition**	Uplynulý čas	Lineární	[2, 1, 0.5, 0]
		Hodnota vlivu	Polynomiální	[-10, 2, 0.5, 1]
		Vzdálenost k cíli	Polynomiální	[-2.2, 2, 0.9, 1.2]

Tabulka 6.1: Navigační rozhodnutí
Void rozhodnutí použité při nastavování navigačních cílů s příslušnými funkcemi užitku a jejich parametry.

Proto jsme vytvořili oddělenou množinu rozhodnutí, jejichž výsledkem není akce ve hře. Těmto rozhodnutím říkáme tzv. void rozhodnutí. Ve třídě **Brain** implementujme metodu **voidThink()**, která z množiny void rozhodnutí vybere nejlepší rozhodnutí, které rovnou vykoná. Tím agent přemýšlí na dvou úrovních — hledá nejlepší akci ve hře (pohni se) a vedle toho hledá také nejlepší akci na straně rozhraní (změň proměnnou).

Void rozhodnutí jsou zachyceny v tabulce 6.1. Faktory nám umožní prioritizovat rozhodnutí, které jsou důležité pro přežití našeho agenta (např. **EscapeThreat**). Tím docílíme například toho, že agent nepůjde na pozici výhodnou z hlediska farmení, pokud by byl v ohrožení.

Farmení

Při farmení pracujeme s mapou vlivu **FarmingLayer**, která zachycuje vliv životů nepřátelské jednotky na její okolí, až do vzdálenosti agentova dostřelu. Nepřímo pracujeme také s mapou **GoalLayer**, která nás naviguje do nejvýhodnější pozice.

Pro nastavování pozic využíváme rozhodnutí **FarmingPosition** (viz Tabulka 6.1). Pokud je vybráno, pak pro navigaci nastavuje jako cíl pozici s největším vlivem, kterou jsme nalezli v mapě **FarmingLayer**. Pozici neupravujeme vždycky, ale snažíme se ji upravovat v delších intervalech, abychom imitovali lidského hráče. Tím dáváme prostor také jiným rozhodnutím.

Zvažujeme také hodnotu vlivu, která se zvětšuje, čím méně životů nepřátelské jednotky mají. Tím se k jednotkám přibližujeme tehdy, když to vede k udělení posledního úderu. Nakonec ještě zvažujeme, jestli není nejlepší pozice příliš blízko našeho agenta. Tím zabráníme moc častému přecházení do výhodné pozice, což by vypadalo nepřirozeně.

Abychom se dostali k jednotkám na lince, definujeme rozhodnutí **CreepAsAGoal**, které nastaví jako cíl jednotku na konci linky. Náš agent tak dojde

do blízkosti nepřátelských jednotek, kde existují užitečnější rozhodnutí, které jej postaví výhodně z hlediska farmení a ohrožení.

Při implementaci rozhodnutí pro udělení posledního úderu jsme nejdříve vytvořili rozhodnutí **AttackCreep** (viz Tabulka 6.2). Předpokládali jsme, že pokud bude agent preferovat jednotky s méně životy, bude udělovat také poslední údery.

Hlavním důvodem, proč se tento přístup neosvědčil jsou naše jednotky. Ty útočí na nepřátelské jednotky a jsou jim schopny nějaké životy ubrat mezi našimi útoky. Například zaútočíme na jednotku, která má dvakrát tolik životů než poškození, které můžeme udělit. Může se stát, že nebudeme schopni jednotce udělit poslední úder, protože ji mezitím spojenecké jednotky zabijí.

Proto jsme vytvořili rozhodnutí **LastHitCreep**. Rozhodnutí **AttackCreep** jsme upravili tak, aby zvažovalo útok na jednotky pouze tehdy, pokud mají více životů než přibližně dvojnásobek našeho maximálního poškození. Tím vznikne jakýsi interval, ve kterém na jednotku s málo životy neútočíme a čekáme na správnou chvíli. Pokud jsme schopni jednotce udělit poslední úder, pak je vybráno rozhodnutí **LastHitCreep**.

Při výběru jednotky, které chceme udělit poslední úder, preferujeme jednotky, za které dostaneme největší odměnu a jednotky, které jsou více ohroženy spojeneckými jednotkami (viz Tabulka 6.2).

Ohrožení

Veškerá mechanika útěku před ohrožením vyplývá z map vlivu. Ty upravují vliv objektů podle toho, jaký mají k našemu hrdinovi vztah. Nám tedy pouze stačí kontrolovat ohrožení na pozici agenta v mapě **CombinedThreatsLayer**. Definujeme void rozhodnutí, které nastaví jako navigační cíl bezpečnější místo, pokud je agent ohrožen.

K tomu slouží rozhodnutí **EscapeToBase** a **EscapeThreat** (viz Tabulka 6.1). Faktory nastavujeme tak, abychom primárně utíkali z ohrožení a teprve potom odcházeli do báze, pokud máme málo životů. Jelikož náš agent nepoužívá stavy, ve kterých by ohrožení jinak toleroval (např. pokud má nepřítel málo životů), tak stačí zvážit pouze ohrožení. Agent vždy utíká na nejbezpečnější pozici směrem k naši bázi.

Útočení

V teorii užitku je třeba rozhodovat o každé jednotce, které se nějaké rozhodnutí týká, zvlášť. Pro výběr jednotky, na kterou zaútočit, musíme vytvořit několik instancí rozhodnutí pro nepřátelské jednotky v našem okolí. V naši implementaci pro potřeby útočení generujeme rozhodnutí pro několik nejbližších jednotek a budov.

Pro výběr správné entity definujeme rozhodnutí, která můžete vidět v tabulce (viz Tabulka 6.2). Především vybíráme nejvýhodnější jednotku, hrdinu nebo budovu v našem okolí. Faktory definujeme tak, abychom po řadě cílili na udělování posledního úderu jednotce, útočení na hrdiny, budovy a útočení na jednotky.

Výběr dále závisí na vlastnostech jednotlivých entit. Například pokud bude mít nepřátelský hrdina málo životů, nejlepším rozhodnutím bude zaútočit na něj.

Faktor	Rozhodnutí	Úvahy	Funkce	Parametry
3	**AttackCreep**	Zdraví jednotky	Sinusoida	[1.9, 2, 5, 0.6]
		Ohrožení	Polynomiální	[-0.9, 2, -0.2, 1.2]
6	**LastHitCreep**	Zdraví jednotky	Binární	[0.5, -1, 0.6, 0.5]
		Ohrožení	Lineární	[0.3, 1, -0.7, 0.5]
		Zlato	Lineární	[0.3, 1, -0.7, 0.5]
		Rychlost projektilů	Lineární	[-0.2, 1, 5.1, 0]
6	**AttackHero**	Zdraví hrdiny	Lineární	[-0.2, 1, 0, 1]
		Ohrožení	Polynomiální	[-0.25, 2, 1.2, 1]
5	**AttackBuilding**	Zdraví budovy	Lineární	[-0.2, 1, 0, 1]
		Ohrožení	Polynomiální	[-3, 2, 1, 1]

Tabulka 6.2: Útočení na jednotky
Rozhodnutí použité při útočení na jednotky, příslušné úvahy s funkcemi užitku a faktory.

Ostatní rozhodnutí

Náš agent se pohybuje po své lince, útočí na nepřátelské jednotky a snaží se udělovat poslední údery. Mimo tyto akce definujeme ještě další rozhodnutí, které se starají o zbytek akcí.

Pro nakupování předmětů vytváříme rozhodnutí, které předmět koupí, pokud máme dostatek zlata a stojíme dostatečně blízko. Problém s tím, že předmět může být koupen v tajném obchodě řešíme void rozhodnutím, které nastaví tajný obchod jako cíl pro navigaci. Pokud jsme nedaleko hlavního obchodu a máme ve skladišti nějaké předměty, automaticky je bereme.

Dále definujeme rozhodnutí, které se starají o doplňování hrdinova zdraví a many. K tomu slouží lektvary, které může mít hrdina ve svém inventáři. Pokud má málo zdraví nebo many, dojde k vybrání rozhodnutí, které použije jeden z lektvarů. Také definujeme rozhodnutí, které hrdinu udrží v blízkosti fontány, pokud má málo zdraví. Toto rozhodnutí je důležité, pokud se hrdina vrátí do báze s málo životy.

Faktor	Rozhodnutí	Úvahy	Funkce	Parametry
10	**CastShrapnel**	Mana	Polynomiální	[-1.8, 2, 1, 0.9]
		Cooldown	Binární	[0.5, -1, 0.01, 0.5]
		Dostřel	Binární	[-1, 0.5, 0.9, 0.5]
		Uplynulý čas	Lineární	[2, 1, 0.6, 0]
		Počet cílů	Polynomiální	[-2.7, 2, 1, 1]
10	**CastAssasinate**	Mana	Polynomiální	[-1.8, 2, 1, 0.9]
		Cooldown	Binární	[0.5, -1, 0.01, 0.5]
		Dostřel	Binární	[-1, 0.5, 0.6, 0.5]
		Zdraví cíle	Polynomiální	[-2.7, 2, -0.3, 1.4]
		Vzdálenost	Lineární	[9.3, 1, 0.95, 1.1]

Tabulka 6.3: Kouzla hrdiny Snipera
Rozhodnutí pro použití kouzel používaných agentem, který ovládá hrdinu Snipera.

Poslední částí, kterou bylo třeba implementovat byly agentovy kouzla. Napří-

klad hrdina Sniper má dvě aktivní kouzla. Prvním je Shrapnel, který na omezený čas na vymezí oblast na herní mapě, která bude nepřátelským jednotkám udělovat poškození. Dalším je Assasinate, jehož pomocí může Sniper vystřelit na jednoho hrdinu a udělit mu velké poškození.

Vykouzlení Shrapnelu závisí především na počtu nepřátel v okolí. Pokud je nepřátel v agentově okolí hodně a rozhodne se kouzlo použít, pak kouzlo v rámci rozhodnutí vykouzlí do těžiště všech jednotek. Tím jich zasáhneme co nejvíce.

Při používání Assasinate se musíme chovat podobně jako při útočení — musíme zvažovat několik hrdinů v našem dostřelu. Assasiante použijeme tehdy, pokud bude mít nepřátelský hrdina málo zdraví (viz Tabulka 6.3).

Podobně definujeme kouzla také pro Linu.

6.4 Shrnutí

V této kapitole jsme popsali implementaci dvou agentů v našem rozhraní. Nejprve jsme popsali implementaci map užitku, které jsou používány pro navigaci, podporu farmení a vyhýbání se ohrožení. Poté jsme popsali implementaci teorie užitku a rozhodnutí, které používáme pro implementaci navigace, farmení, vyhýbání se ohrožení a dalších agentových dovedností.

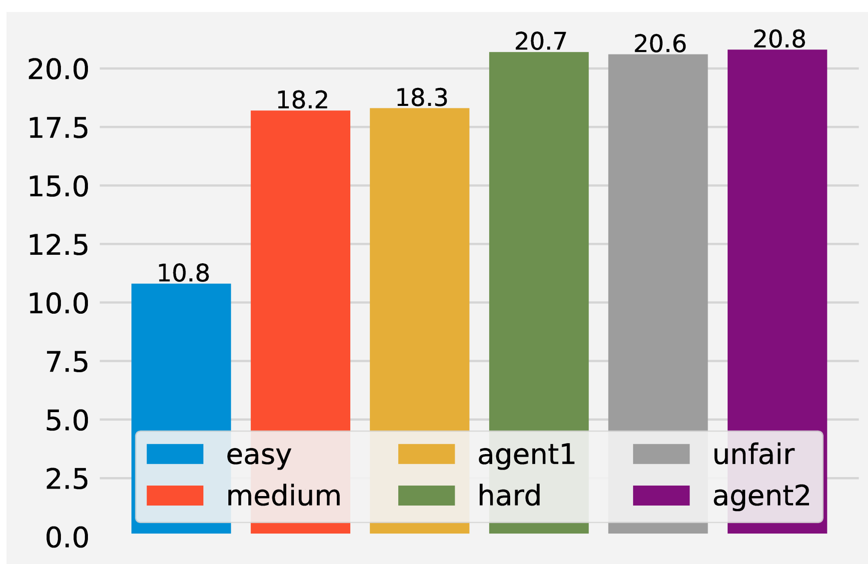
7. Experimenty a výsledky

V této kapitole se zabýváme především analýzou agentů, které jsme v rámci práce vytvořili. Analýzu dělíme do dvou sekcí, které obsahují analýzu farmení a analýzu schopností agentů vyhrát hru.

Agentovy dovednosti vždy analyzujeme v rámci hry, ve které hrají specifičtí hráči. Takto definovaným hrám říkáme scénáře. Součástí přílohy A je uživatelská dokumentace, ve které je vysvětleno, jak scénáře použít při nastavování hry v rozhraní.

7.1 Farmení

V předchozí kapitole jsme popsali, jak je farmení závislé na mapách vlivu a rozhodování. Při testování hraje náš agent sám na prostřední lince po dobu 5 minut (Scenario 3). Počítáme, kolik zvládne průměrně udělit posledních úderů za deset odehraných her. To porovnáváme s agenty, které implementuje Dota 2 (viz Obrázek 7.1). Všichni agenti ovládají hrdinu Snipera.



Obrázek 7.1: Farmení
Hodnoty představují průměrný počet jednotek zabitých hrdinou Sniperem za 5 minut hry. Každý hrdina hrál celkem 10 her.

Prostřední linku jsme zvolili protože se na ni jednotky na začátku střetávají přesně uprostřed, takže není pro jednu stranu výhodnější. Nehrajeme proti nepřátelským hrdinům, protože jejich přítomnost ovlivní přemýšlení našeho agenta, který upřednostňuje bezpečí před udělením posledního úderu. Hra je hrána krátkou dobu, aby hrdinové neměli čas nakoupit si předměty, které by jim farmení ulehčili.

Na obrázku 7.1 můžete vidět, že náš agent (agent1) udělil podobný počet posledních úderů, jako středně obtížný agent implementovaný v Dotě 2.

Agent hraje agresivně, protože útočí na jednotky i když mají hodně životů. Může dojít k tomu, že se dostane se svými jednotkami pod nepřátelskou věž, kde

není schopen jednotkám udělovat poslední údery. Proto jsme se rozhodli vytvořit také agenta (agent2), který uděluje pouze poslední údery (textbfLastHitCreep) a jinak neútočí. Tento agent byl schopen udělit podobný počet posledních úderů, jako nejobtížnější agenti implementovaní Dotou 2.

Při analýze jsme identifikovali další problém, který náš agent může mít při last-hitování. Uprostřed prostřední linky protéká řeka, skrze kterou linka prochází pomocí dvojice schodů. Pokud se hrdina postaví do řeky, má šanci, že nepřátelské jednotky, které stojí nad schody, netrefí a neudělí jim poslední úder (viz Kapitola 1). Proto se vyplatilo na jednotky aktivně neútočit (agent2), protože k této situaci dochází méně často. Problém by se dal vyřešit znevýhodněním pozic na mapě **FarmingLayer** podle jejich výšky.

7.2 Vyhrání hry

Abychom analyzovali agentovu schopnost vyhrát hru, nechali jsme agenta nejprve hrát samotného na prostřední lince, bez časového omezení, stejně jako v předchozí sekci (Scenario 3). Agent byl schopen opakovaně vyhrát hru bez jediné smrti po přibližně dvaceti minutách. Agent útočil na jednotky, budovy, používal kouzla a úspěšně se vyhýbal nebezpečí.

Na agentovi bylo vidět, že je schopen dobře udržovat svou polohu a odcházet z nebezpečí. Bylo ovšem zřejmé, že má občas problémy s udělováním posledního úderu ve chvíli, kdy na některou jednotku útočí mnoho jednotek. S tím mají problém i pokročilí hráči. Řešením by bylo při rozhodování uvažovat přímo počet jednotek, které na jednotku útočí, nikoliv ohrožení, jak to dělá rozhodnutí **LastHitCreep**.

Zároveň bylo vidět, že se agent nechová optimálně uvnitř nepřátelské báze. Především útočí na špatné budovy, rozhodnutí nerozlišují mezi typem budov. Takže místo na pevnost může útočit na jinou nedůležitou budovu. Dále mapy vlivu agentovi neumožňují přiblížit se ke dvěma věžím uprostřed báze. To ovšem nevadí, protože se agent snaží hrát co nejvíce bezpečně.

Poté jsme naše agenty nechali hrát proti agentům implementovaným Dotou 2 (tzv. boti). Naši agenti byli schopni boty překonat na začátku hry a zničit jejich první věže. Nepřátele měli na lince často málo životů. Za to mohly především kouzla, které naši agenti byli schopni správně použít. Agenti prokázali, že jsou schopni používat kouzla a udržet se na lince dostatečně dlouho, aby nepřátelům zničili první věže. Nebyli ovšem schopni nepřátelské hrdiny zabít, což ani nebylo cílem jejich implementace.

V pozdějších fázích hry začali vyhrávat boti, kteří dovedou správně řetězit kouzla a útočit tehdy, když mají výhodu. Také nakupují předměty, používají kurýra a hra je zvýhodňuje (mají více zlata a zkušeností). Také jsou schopni jisté spolupráce, na kterou naši agenti neumí správně reagovat. Ve výsledku nebyli naši agenti schopni hru proti botům vyhrát.

Závěr

V této práci jsme implementovali rozhraní, které umožňuje vývoj agentů pro Dotu 2. Rozhraní je založeno na existujícím Dota 2 Frameworku, jehož nedostatky jsme analyzovali a v rámci této práce překonali.

Nad implementovaným rozhraním jsme vytvořili dva agenty. Agenti úspěšně využili map vlivu a teorie užitku k zvládnutí základních herních dovedností. Agenti byli schopni vyhrát hru bez jediné smrti a prokázat podobné dovednosti při farmení jako hrdinové, které ovládá Dota 2.

Agenti byli schopni prokázat, že rozhraní může být smysluplně využito pro vývoj umělé inteligence pro Dotu 2.

Případné rozšíření

Rozhraní by mělo být lehce rozšířitelné. Rozšiřování může být využito pro dosažení některého z těchto cílů:

- **Úpravy rozhraní**: Pro další využití rozhraní bude třeba upravovat poskytnutou abstrakci, aby odpovídala požadavkům případných uživatelů.
- **Hra po síti**: Rozhraní umožňuje hru s více agenty na jednom počítači. Aby bylo rozhraní lépe použitelné je třeba prozkoumat, zda Dota 2 umožňuje hráčům vytvářet vlastní servery. Cílem by bylo umožnit hru více hráčům po síti proti agentům ovládaných serverem.
- **Analýza hry**: Hra našemu rozhraní posílá mnoho informací, které mohou být dále využity pro analýzu hry (analýzy, predikce).
- **Predikce výsledku hry**: Na základě stavu hry by bylo zajímavé pokusit se předvídat její výsledek.
- **Výuka umělé inteligence**: Rozhraní se dá použít pro výuku umělé inteligence.
- **Kurýr**: V rozhraní není implementován kurýr. Z dokumentace Dota 2 Scripting API není jasné, jak toho dosáhnout.

Rozhraní by nemělo stát v cestě použití žádného z přístupů, které jsme zmínili v kapitole 4. Ve stejné kapitole jsme také uváděli výzkum, který byl v žánru MOBA proveden. Většina zmíněných prací, by měla být proveditelná prostřednictvím našeho rozhraní.

Vedle rozhraní můžeme rozšiřovat také naše agenty. Napadají nás následující rozšíření:

- **Emoce a stavy**: Agentům by šlo přidat stavy, které by jim například umožnily hrát agresivněji, pokud by měl nepřítel málo životů. Se stavy je možné pracovat jako s emocemi a pokusit se imitovat chování lidských hráčů.

- **Útočení na nepřátele**: Cílem našich agentů nebylo zabíjet nepřátele. V dalším rozšíření je možné implementovat tuto dovednost.
- **Analýza nepřátel, kouzel a předmětů**: Existuje obrovské množství částí hry, které jsou pro agenta zajímavé a které může analyzovat.
- **Hra v týmu**: Pokusit se vytvořit tým agentů, kteří budou schopni spolupráce.
- **Výběr hrdinů**: Před začátkem hry bychom se mohli zabývat výběrem správného hrdiny.
- **Rozhraní pro teorii užitku**: Naši agenti používají teorii užitku, která nebyla zatím v žánru MOBA dostatečně použita. Pro vytvoření nástroje, který by umožnil její snadnější vývoj pro Dotu 2, by bylo třeba přesunout tvorbu rozhodnutí do konfiguračních souborů mimo rozhraní (změna != rekompilace) a vytvořit lepší testovací nástroje. Rozhraní by mohlo být využito k výuce teorie užitku.

Seznam použité literatury

[1] Eddie Makuch. Steam's Most-Played Games of 2016 Revealed. https://www.gamespot.com/articles/steams-most-played-games-of-2016-revealed-report/1100-6446818/, January 2017. [Online; accessed 2018-04-28].

[2] Siddhesh V Kolwankar. Evolutionary artificial intelligence for moba/action-rts games using genetic algorithms. *International Journal of Computer Applications (IJCA)(0975–8887)*, pages 29–31, 2012.

[3] Mateusz Wiśniewski and Artur Niewiadomski. Applying artificial intelligence algorithms in moba games. *Studia Informatica: systems and information technology*, 1(2 (20)):53–64, 2016.

[4] Victor do Nascimento Silva and Luiz Chaimowicz. On the development of intelligent agents for moba games. In *Computer Games and Digital Entertainment (SBGames), 2015 14th Brazilian Symposium on*, pages 142–151. IEEE, 2015.

[5] Valve. Dota 2 workshop tools. https://developer.valvesoftware.com/wiki/Dota_2_Workshop_Tools, 2017. [Online; accessed 2018-04-28].

[6] Mahlmann Tobias. The dota2 ai framework. https://github.com/lightbringer/dota2ai, 2017. [Online; accessed 2018-04-28].

[7] Gamepedia. Dota 2 wiki. https://dota2.gamepedia.com/Dota_2_Wiki, February 2017. [Online; accessed 2018-04-28].

[8] Wikipedia contributors. Defense of the ancients (dota). https://en.wikipedia.org/wiki/Defense_of_the_Ancients, 2017. [Online; accessed 2018-04-28].

[9] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (3rd Edition)*. Pearson, 2009.

[10] J Willich. Reinforcement learning for heroes of newerth. *Bachelor thesis, Technische Universitat Darmstadt*, 2015.

[11] Pu Yang, Brent E Harrison, and David L Roberts. Identifying patterns in combat that are predictive of success in moba games. In *FDG*, 2014.

[12] Michael Waltham and Deshen Moodley. An analysis of artificial intelligence techniques in multiplayer online battle arena game environments. In *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, page 45. ACM, 2016.

[13] Ian Millington and John Funge. *Artificial Intelligence for Games*. CRC Press, 2009.

[14] Tinnawat Nuangjumnonga and Hitoshi Mitomo. Leadership development through online gaming. In *19th ITS Biennial Conference 2012*. Bangkok: ITS, 2012.

[15] Nataliia Pobiedina, Julia Neidhardt, Maria del Carmen Calatrava Moreno, and Hannes Werthner. Ranking factors of team success. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1185–1194. ACM, 2013.

[16] A. Drachen, M. Yancey, J. Maguire, D. Chu, I. Y. Wang, T. Mahlmann, M. Schubert, and D. Klabajan. Skill-based differences in spatio-temporal team behaviour in defence of the ancients 2 (dota 2). In *2014 IEEE Games Media Entertainment*, pages 1–8, Oct 2014.

[17] Victor do Nascimento Silva and Luiz Chaimowicz. Moba: a new arena for game ai. *arXiv preprint arXiv:1705.10443*, 2017.

[18] Tom Batsford. Calculating optimal jungling routes in dota2 using neural networks and genetic algorithms. *Game Behaviour*, 1(1), 2014.

[19] Ruoyu Sun. Extremepush - dota 2 bot scripts. `https://github.com/insraq/dota2bots`, 2017. [Online; accessed 2018-04-28].

[20] OpenAI. Dota 2. `https://blog.openai.com/dota-2/`, 2017. [Online; accessed 2018-04-28].

[21] OpenAI. More on dota 2. `https://blog.openai.com/more-on-dota-2/`, 2017. [Online; accessed 2018-04-28].

[22] Mirna Paula Silva, Victor do Nascimento Silva, and Luiz Chaimowicz. Dynamic difficulty adjustment on moba games. *Entertainment Computing*, 18:103–123, 2017.

[23] Dave Mark. Modular tactical influence maps. *Game AI Pro*, 2:343, 2015.

[24] Johan Hagelbäck and Stefan J Johansson. Using multi-agent potential fields in real-time strategy games. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*, pages 631–638. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[25] Johan Hagelbäck and Stefan J Johansson. The rise of potential fields in real time strategy bots. In *Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*. Stanford University, 2008.

[26] Alberto Uriarte and Santiago Ontanón. Kiting in rts games using influence maps. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[27] Kevin Dill Dave Mark. Improving ai decision modeling through utility theory. `https://www.gdcvault.com/play/1012410/Improving-AI-Decision-Modeling-Through`, 2010. [Online; accessed 2018-04-28].

[28] Mike Lewis Dave Mark. Building a better centaur: Ai at massive scale. `https://www.gdcvault.com/play/1021848/Building-a-Better-Centaur-AI`, 2015. [Online; accessed 2018-04-28].

Seznam obrázků

1.1 Herní prostředí . . . 6
1.2 Viditelnost . . . 6
1.3 Minimapa s linkami . . . 8
1.4 Obrázek ze hry . . . 10

2.1 Části Dota 2 AI Frameworku . . . 15
2.2 Ukázka požadavku *update* . . . 15
2.3 Navržené požadavky . . . 18
2.4 Části rozhraní . . . 20
2.5 Stavy rozhraní . . . 21

3.1 Získaná reprezentace mapy . . . 23
3.2 GridSystem a GridBase . . . 26
3.3 Context, AgentContext a TeamContext . . . 26
3.4 WorldManager a World . . . 27
3.5 Objekty InterestsBase, Lanes a Jungle . . . 28
3.6 BaseEntity . . . 29
3.7 AgentController a BaseAgentController . . . 30
3.8 Ukázka GUI - nastavení hry . . . 31

5.1 Vliv dosahu . . . 41
5.2 Ukázkové rozhodnutí . . . 44
5.3 Základní funkce . . . 45

6.1 ExtendedAgentContext . . . 48
6.2 Používané mapy vlivu . . . 49
6.3 Ohrožení věží . . . 52
6.4 DecisionSet, Decision a Consideration . . . 53

7.1 Farmení . . . 58

Seznam tabulek

6.1 Navigační rozhodnutí 54
6.2 Útočení na jednotky 56
6.3 Kouzla hrdiny Snipera 56

Přílohy

Příloha A

Součástí elektronické verze práce je příloha, která obsahuje:

- **Dota2Framework**: Java Maven projekt, který obsahuje zdrojový kód implementovaného rozhraní.
- **ExampleAgents**: Java Maven projekt, který obsahuje zdrojový kód implementovaných agentů.
- **docs**: Obsahuje následující soubory:
  - **javadoc**: Složka, která obsahuje *javadoc* vygenerovaný pro oba výše uvedené projekty.
  - **UserDoc.pdf**: Uživatelská příručka.
  - **ProgrammerDoc.pdf**: Programátorská dokumentace rozhraní.
  - **ExampleAgentsDoc.pdf**: Programátorská dokumentace agentů.
- **release**: Složka obsahující zkompilované kódy a další data.
  - **addons**: Složka obsahující implementaci dota2ai addonů.
  - **configurations**: Obsahuje nastavení agentů, které jsou rozděleny do složek podle různých scénářů.
  - **data**: Obsahuje důležitá data jako nastavení rozhraní (config.cfg), exportovaný herní svět (grid.data) a předměty (items.data).
  - **lib**: Obsahuje dynamicky linkované knihovny nutné při spouštění.
  - ***.jar**: Archivy obsahující zkompilované projekty.
  - ***.bat**: Dávkové soubory sloužící pro spouštění rozhraní.

Struktura složky **release** je blíže popsána v uživatelské dokumentaci (viz **UserDoc.pdf**).