VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

TRÉNOVÁNÍ INTELIGENTNÍCH AGENTŮ V ENGINU UNITY

TRAINING INTELLIGENT AGENTS IN UNITY GAME ENGINE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JAN VACULÍK**
AUTHOR

VEDOUCÍ PRÁCE **MICHAL MATÝŠEK, Ing.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Vaculík Jan**
Program: Informační technologie
Název: **Trénování inteligentních agentů v enginu Unity**
Training Intelligent Agents in Unity Game Engine
Kategorie: Umělá inteligence

Zadání:

1. Seznamte se s problematikou trénování inteligentních agentů pomocí sady nástrojů Unity Machine Learning Agents.
2. Nastudujte relevantní metody strojového učení (reinforcement learning, imitation learning) pro trénování agentů v herním enginu Unity a seznamte se s obecnými postupy tvorby her v tomto enginu.
3. Navrhněte demonstrační aplikaci (hru nebo simulaci), která umožní trénování a testování agentů v různých prostředích/scénářích.
4. Implementujte demonstrační aplikaci a vyhodnoťte/porovnejte vlastnosti agentů a použitých metod.
5. Vytvořte video pro prezentaci projektu, zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu.

Literatura:

- Dle doporučení vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3, rozpracovaný bod 4.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Matýšek Michal, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 30. října 2020

Abstrakt

Cílem práce je navrhnout aplikace, které demonstrují sílu strojového učení pro tvorbu umělé inteligence ve videohrách. K řešení této problematiky je použita sada nástrojů ML Agents, která umožňuje tvorbu inteligentních agentů v enginu Unity. Jednotlivé demonstrační aplikace jsou zaměřeny na různé scénáře využití této sady. Pro trénování je použito zpětnovazební a imitační učení.

Abstract

The goal of this work is to design applications, which demonstrate the power of machine learning in video games. To achieve this goal, this work uses the ML-Agents toolkit, which allows the creation of intelligent agents in the Unity Game Engine. Furthermore, a series of experiments showing the properties and flexibility of intelligent agents in several real-time scenarios is presented. To train the agents, the toolkit uses reinforcement learning and imitation learning algorithms.

Klíčová slova

strojové učení, neuronová síť, posilované učení, učení napodobováním, Unity, ML Agents, Python, agenti

Keywords

Machine Learning, Neural Netowrk, Deep Reinforcement Learning, Imitation Learning, Unity, ML Agents, Python, Agents

Citace

VACULÍK, Jan. *Trénování inteligentních agentů v enginu Unity.* Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Michal Matýšek, Ing.

Trénování inteligentních agentů v enginu Unity

Prohlášení

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

.....
Jan Vaculík
6. května 2021

Poděkování

Tímto bych chtěl poděkovat mému vedoucímu, Ing. Michalovi Matýškovi, za pomoc a připomínky a Mgr. Evě Škrobálkové, za pravopisné rady.

Obsah

Kapitola 1

Úvod

Umělá inteligence neodmyslitelně patří k vývoji videoher. Používají se komplexní algoritmy, řešící problém tvorby inteligentních protivníků či spoluhráčů, které ovšem často nejsou schopné dostatečně dobře simulovat chování, jež by si tvůrce představoval, či jsou komplikované na navrhnutí. Pro tvorbu umělé inteligence se nicméně dá použít i v poslední době velmi protežované strojové učení. To je jako takové v posledních letech stále častěji probírané téma, ať už jde o autonomní vozidla, analýzu řeči nebo tvorbu neporazitelného šachového protivníka. Jeden nástroj, určený pro učení umělé inteligence, je podrobněji rozepsán a použit v této práci – tím je balíček nástrojů ML-Agents.

Cílem této práce je vytvořit sadu demonstračních aplikací, které budou prezentovat možnosti tvorby umělé inteligence při trénování inteligentních agentů pro účely vývoje počítačových her. Práce je zaměřena na scénáře, které nebudou jenom technickou ukázkou, ale reálnými návrhy, jež by mohly po dalším vývoji existovat i jako samostatné projekty.

V rámci první kapitoly bude nejdříve čtenář seznámen s obecnými koncepty a principy umělé inteligence a strojového učení, současnou situací v této sféře a jsou zde také představeny některé z metod Deep Reinforcement Learning, tedy zpětnovazebního učení a Imitation Learning – učení napodobováním. Dále bude obsah práce směřován k ML-Agents. Zde bude nejdříve popsáno samotné prostředí, ve kterém ML-Agents běží (Unity), jeho klíčové součásti a v jednotlivých kapitolách jsou blíže rozebrány možnosti této sady. Tímto budou uzavřeny teoretické kapitoly a následovat bude návrh a popis provedené implementace jednotlivých aplikací. Poté následuje průběh trénování agentů v části experimenty a na závěr zbývá zhodnocení každé aplikace a zda a jak se v nich podařilo vytrénovat neuronovou síť.

Kapitola 2

Umělá inteligence a strojové učení

V roce 1950 začal Allan Turing svou slavnou práci otázkou „Mohou stroje myslet?“ [33]. Tato otázka potom definuje druhou polovinu 20. století, kdy se čím dál více začíná pracovat na odvětví, které je dnes známé jako umělá inteligence (zkráceně AI). Je možné říct, že tento obor se v kontextu číslicových počítačů zabývá tvorbou programů plnících úkoly, které vyžadují lidskou inteligenci [7]. Podoblastí umělé inteligence je potom strojové učení, to je chápáno jako studium počítačových algoritmů, které se na základě zkušeností automaticky zlepšují [18]. V této kapitole bude stručně nastíněna historie umělé inteligence a strojového učení, jaké je dění na tomto poli v současnosti a rozebrány budou také některé koncepty související s problematikou této práce.

2.1 Historie umělé inteligence

Představy o inteligentních strojích napodobujících člověka provázejí lidstvo už od antiky. Byl to nicméně rok 1956 a John McCarthy, který společně s Marvinem Minskym v průběhu letního semináře v Dartmouth College založil obor umělé inteligence. Dalších zhruba 20 let převažoval optimismus. Během této doby vznikl projekt ELIZA, jeden z prvních programů zpracovávajících přirozenou řeč a vedoucí základní konverzaci. Ačkoliv byly vyhlídky do budoucnosti AI vysoké, ukázalo se že tehdejší výpočetní technika nenabízela dostatečný výkon a vývoj odvětví začal zpomalovat.

V 80. letech se vývoj, především v komerční sféře, točí okolo expertních systémů – programů, které se snaží simulovat rady experta v oblasti vyžadující expertní znalosti [42]. Příslib takovýchto programů se zalíbil i velkým firmám z Fortune 500 [25] a s tím přišlo zároveň zvýšení financovaní, kdy se průmysl od začátku 80. let a pár milionů dolarů vyšplhal v roce 1988 na 2 miliardy [26]. Dalším významným milníkem je určitě rok 1997 a superpočítač Deep Blue vyvíjený firmou IBM. Ten předešlý rok v jedné hře porazil tehdejšího světového šachového šampiona Garyho Kasparova, ale celé utkání nakonec nevyhrál. Bylo to až o rok později, kdy se upravené verzi tohoto počítače povedlo v odvetném zápase nad Kasparovem zvítězit. Aktuálnějším úspěchem na poli souboje AI a člověka je systém OpenAI Five, kterému se jako prvnímu podařilo pokořit esportový tým ve hře *Dota 2*. Protože je (například oproti šachům) Dota 2 hra s neúplnými informacemi, doufají autoři, že zkušenosti získané z tohoto projektu budou aplikovatelné obecněji [3, 2].

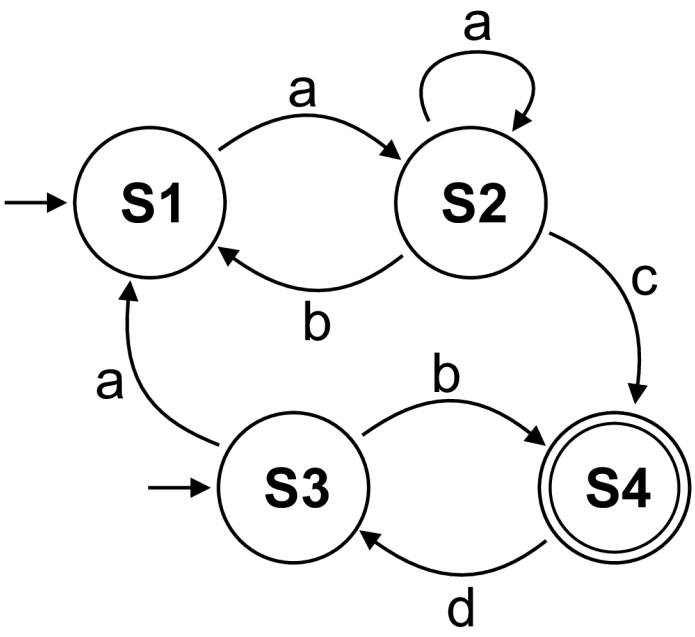
2.2 Umělá inteligence ve hrách

V současnosti je možno se s AI setkat takřka ve všech oborech [30, 12, 16]. Kde s umělou inteligencí do styku přijdeme téměř vždy jsou počítačové hry. V těch se chování AI v současnosti většinou programuje přímo, tedy tak, že si tvůrci sami vymezí konkrétní akce, a kdy k nim dochází, a k těm se vytvoří kód. Často používanými metodami k navržení takové umělé inteligence jsou konečné automaty a stromy chování.

2.2.1 Konečné automaty

Formálně je konečný automat popsán jakožto uspořádaná pětice $A = (Q, \Sigma, R, S, F)$, kde Q je konečná množina stavů, Σ je konečná vstupní abeceda, $R \subseteq Q \times \Sigma \to Q$ je konečná množina pravidel tvaru $pa \to q$, kde $p, q \in Q$ a $a \in \Sigma$, $S \in Q$ je počáteční stav a $F \subseteq Q$ je množina koncových stavů [17]. Na obrázku 2.1 je jednoduchý diagram konečného automatu, kde písmenem S v kruzích jsou označené stavy (S4 je koncový stav) a šipky s písmeny reprezentují přechody.

Pomocí konečných automatů lze definovat takovou umělou inteligenci, která má nějakou konečnou množinu stavů a akcí. Tato implementace je vhodná pro malé projekty a byla v minulosti hojně využívána, jak komplexita her rostla a projekty se zvětšovaly, tento přístup přinesl značné problémy. Grafy s velkým počtem stavů jsou složitě čitelné a údržba automatů s jejich velkým počtem je náročná, protože pokud se odstraní nebo přidá stav, musí se měnit často velké množství pravidel. Tento problém částečně řeší hierarchické konečné stavové automaty, které zapouzdřují uskupení stavů, tedy jiné konečné automaty, do jednoho stavu, vzniká tak tedy vyšší míra abstrakce [21]. Hierarchické stavové automaty využila v poslední době například hra *Doom* z roku 2016, nebo *Batman: Arkham Asylum* (2009) [31].



Obrázek 2.1: Diagram konečného automatu

2.2.2 Stromy chování

Možné řešení škálovatelnosti a rozšiřování chování počítačem řízených postav přináší stromy chování. Jedna z prvních her, která tuto metodu tvorby umělé inteligence použila a přinesla ji do širšího povědomí, je *Halo 2* [13]. Lepší škálovatelnost je oproti stavovým automatům zajištěna bezstavovostí stromů chování – velké množství stavů by v prvním případě vedlo také ke zvyšujícímu se množství přechodů mezi nimi (v nejhorším případě až n^2) a přidání jednoho stavu může znamenat změnu celého automatu. V druhém případě nové stavy, nebo celé podstromy, nijak neovlivňují už navrženou strukturu, což vede k lepší přehlednosti a modularitě chování [29].

Stromy chování mají jeden kořen a n potomků, vnitřní uzly se nazývají uzly řízení toku (*control flow nodes*) a listy potom uzly provádění (*execution nodes*). Strom začíná provádět svoje chování od kořene, kdy pošle potomkovi *tick*, ten ihned odpovídá jedním ze tří signálů: *běh* jestliže uzel (respektive potomci uzlu) dále provádí nějakou činnost, *úspěch* pokud byly úspěšně splněny podmínky uzlu a *neúspěch* v opačném případě. V klasických popisech stromů chování existují 4 typy uzlů řízení toku:

- sekvenční
- fallback
- paralelní
- dekorátorové

Sekvenční uzly postupně zleva všem potomkům posílají tick. Pokud některý z potomků signalizuje běh či neúspěch, propaguje tento signál dále svému rodičovskému uzlu a další z potomků v řadě už tick nedostane. Úspěch vrací pouze tehdy, když uspějí všichni potomci. Tyto uzly se dají chápat podobně jako logický AND. Fallback uzly jsou naopak analogií k logickému OR. Pokud některý z potomků zahlásí úspěch či běh, učiní tak i fallback uzel. Jakmile by selhali všichni potomci, tak signalizuje selhání svému rodičovskému uzlu.

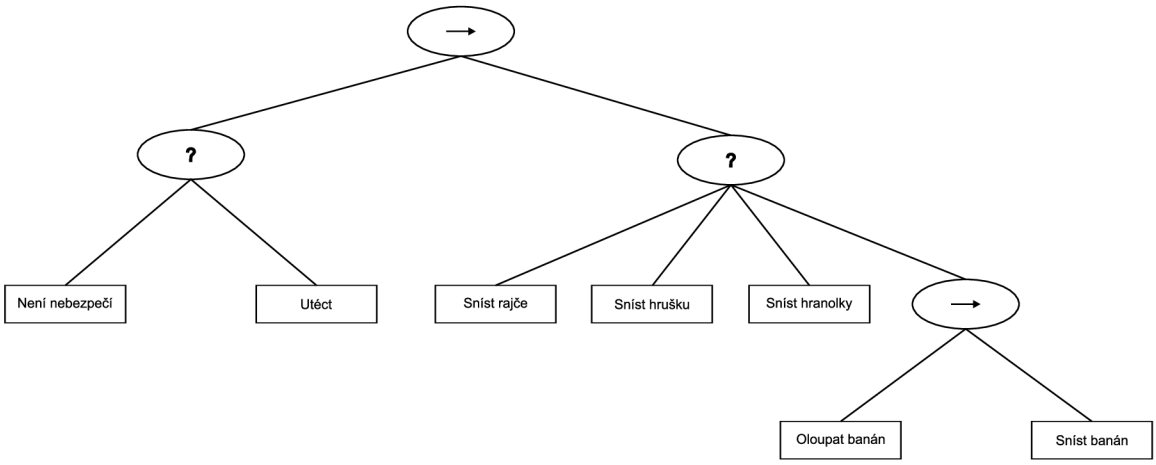
Paralelní uzel rozesílá tick všem potomkům najednou. Pokud M potomků vrací úspěch, signalizuje tak i paralelní uzel a neúspěch vrací v případě, že tak skončilo $N - M + 1$ potomků, kde N je počet potomků uzlu a $M \leq N$ je uživatelem zvolená hranice. Ve všech ostatních případech vrací status běhu.

Dekorátory se v případě stromů chování přidávají před právě jeden uzel, jehož chování nějakým způsobem mění. Mohou například invertovat úspěch/neúspěch potomka. Jednou z výhod dekorátorů je možnost použít již vytvořenou strukturu několika způsoby, aniž by se musela vyrobit znovu [6].

Ve zkratce se dá říct, že návrh umělé inteligence pomocí konečných automatů je rychlejší a snáze uchopitelný a bývá vhodný pro menší projekty. Pokud ovšem tvůrce očekává zvýšenou komplexitu chování, nebo chce větší modularitu, mohou být stromy chování tou správnou volbou.

2.3 Neuronové sítě

Umělou neuronovou síť lze popsat jako sbírku uzlů - *neuronů*, ty bývají typicky uspořádány do několika vrstev. Jednotlivé vrstvy jsou propojeny a neurony z první (vstupní) vrstvy posílají signál další vrstvě až k poslední (výstupní), která reprezentuje výsledek hledaného problému dané sítě. Jednotlivá spojení, nebo také *synapse*, mají přiřazené váhy. Tyto váhy



Obrázek 2.2: Diagram reprezentuje strom chování, kde kulaté uzly představují uzly řízení toku (otazník je potom fallback uzel a šipka uzel sekvenční) a hranaté uzly jsou uzly provádění.

určují, zda se při určitém vstupu neuron z následující vrstvy aktivuje. K tomuto výpočtu slouží tzv. *aktivační funkce*.

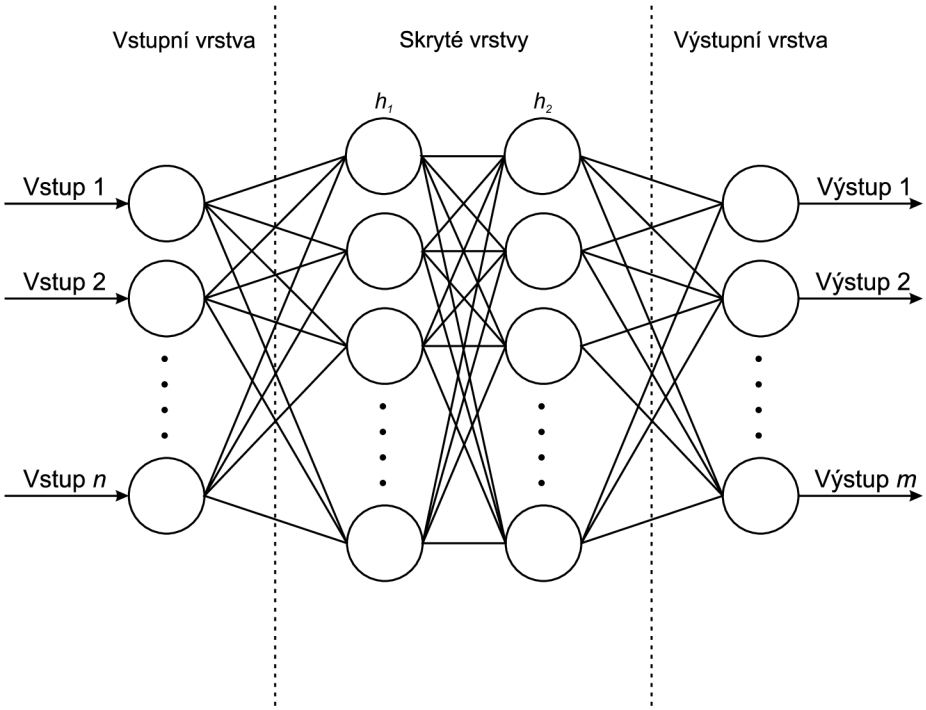
Výpočet hodnoty následující vrstvy může vypadat následovně: Mějme dvě plně propojené vrstvy x a y. První, v tomto případě vstupní vrstva, je popsána jako vektor o velikosti n_x (počet neuronů ve vrstvě). Dále je zapotřebí matice W o velikosti $n_x \times n_y$, která reprezentuje jednotlivé spojení mezi vrstvami. Poslední prvek použitý k určení je *bias* b – konstanta sloužící k vychýlení aktivační funkce A, která v podstatě určuje, jak snadno se neuron aktivuje. Hodnotu výstupní vrstvy je potom možné určit jako:

$$y = A(W \cdot x + b) \tag{2.1}$$

Mezi nejpoužívanější metody k optimalizaci parametrů neuronové sítě a dosažení kýženého výsledku patří metoda gradientního sestupu (*gradient descent*) za použití algoritmu zpětné propagace (*back-propagation*), která spočívá v nalezení lokálního minima. Pomocí této metody se zmenšuje cenová funkce sítě, určující správnost výsledku sítě – čím menší, tím lepší [8].

2.4 Inteligentní agenti

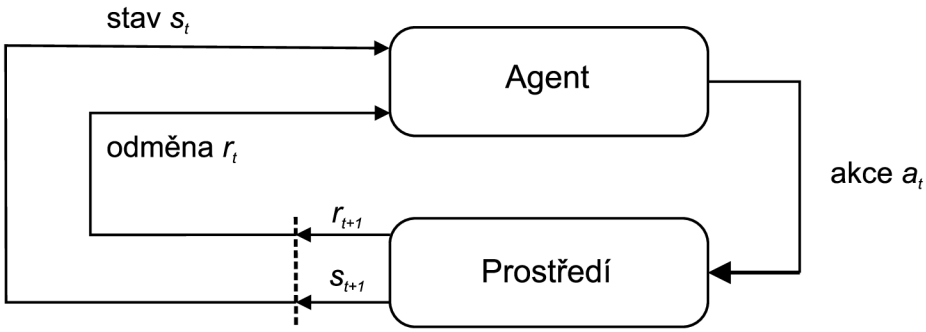
Co je to agent se může dle různých definic lišit [41, 5]. Nejčastěji se setkáváme s vysvětlením, že je to entita, která v rámci prostředí na základě vjemů nějak jedná – tedy že získává informace a provádí akce. Inteligentní agent je potom takový agent, jenž se může učit a dle znalostí se rozhodne provést akci co možná nejlepší k dosažení svého cíle [26]. Zatímco člověk má k dispozici smysly jako je vidění, sluch, nebo hmat, získávání vjemů pro agenty bývá jedním z prvních problémů, se kterým je možné se při jejich návrhu setkat. Pro autonomní vozidlo to mohou být různé kamery či senzory, pro program to potom jsou například proměnné měnící se v čase. Dalším rozhodnutím při návrhu je dosažení cíle, neboli jak agenta vytrénovat. Existuje mnoho metod jak agenta naučit jednat a liší se nejen způsobem učení (prozkoumávání prostředí, učení z demonstrací nebo datasetů, ...), ale také jakých výsledků mohou dosáhnout.



Obrázek 2.3: Schéma neuronové sítě s dvěma skrytými vrstvami

2.5 Zpětnovazební učení

Ve zpětnovazebním učení (je možné se setkat také s termínem posilované učení) se agenti snaží v daném prostředí maximalizovat odměnu. Není jim nicméně na začátku řečeno, jaké akce provádět – ty musejí odhalit sami. Za akce jsou odměňováni s cílem maximalizovat počet akcí s pozitivní odměnou a vyvarovat se těm negativním. Toto dodává agentovi prvotní impulz k jednání. Nerozhoduje se nicméně pouze dle okamžitého ohodnocení, ale snaží se maximalizovat kumulativní odměnu v dlouhodobém časovém horizontu. Na obrázku 2.4 je zobrazeno schéma interakce agenta a prostředí.



Obrázek 2.4: Schéma zpětnovazebního učení podle Richarda S. Suttona

Formálně je možné zpětnovazební učení popsat jako agenta a prostředí, kteří spolu interagují v časových krocích t a kdy v každém kroku získá agent nějaký stav prostředí $S_t \in \mathcal{S}$, kde $\mathcal{S}$ je množina možných stavů. Podle něj si zvolí akci $A_t \in \mathcal{A}(S_t)$, kde $\mathcal{A}(S_t)$ je množina akcí proveditelných ve stavu S_t. O krok později v S_{t+1} získá agent odměnu

$R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, kde $\mathcal{R}$ je množina možných odměn. Agent jednotlivé kroky v závislosti na stavu vybírá podle své rozhodovací taktiky (*policy*) π_t, kde $\pi_t(a|s)$ je pravděpodobnost $A_t = a$ za předpokladu že $S_t = s$. Pro epizodické, rozdělené do časově konečných oddělených částí, trénování agenta by stačila suma všech odměn, nicméně pro úkoly pokračující v čase, mající poslední krok $T = \infty$, se používá diskontní faktor γ, pro nějž platí $0 \leq \gamma \leq 1$. Celková odměna je potom získána jako:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.2}$$

Hluboké zpětnovazební učení je snoubení dekády starého zpětnovazebního učení a hlubokého učení, respektive hlubokých neuronových sítí. Důvod použití neuronových sítí v rámci mnohých metod zpětnovazebního učení je jejich schopnost zpracovávat a generalizovat velký stavový prostor, ve kterém se agent nachází [27].

2.5.1 Přístupy ve zpětnovazebním učení

Jednotlivé přístupy a algoritmy zpětnovazebního učení se dají rozdělit do tří kategorií:

- model-based
- value-based
- policy-based

Model-based algoritmy pracují s modelem prostředí, tento model je buď explicitně zadán (šachy, go), nebo se snaží vytvořit model samy. Výhodou tohoto přístupu je velká možnost plánování do budoucnosti. Model-based metody svoji strategii určují na základě naučeného modelu, tedy že agent předpovídá, jak bude prostředí reagovat. Na druhou stranu *model-free* algoritmy volí optimální strategii pomocí metody pokus omyl. Výhodou model-based přístupu je vysoká efektivita využití jednotlivých zkušeností, které algoritmus získá (jsou tzv. *sample efficient*), nevýhodou může být složitější implementace (nutnost získat model).

Value-based metody získávají hodnotu stavu, nebo dvojice akce a stavu. K tomuto se používají funkce hodnoty (*value function*) a snaží se tak najít optimální taktiku π nepřímo. Jinými slovy se učí, v jak výhodném stavu se nacházejí a jaká je nejlepší akce, kterou zvolit, a vycházejí čistě z hodnotových funkcí. Oproti tomu *policy-based* metody pracují s nějakou explicitní reprezentací taktiky s parametry, které upravují. Policy-based metody obecně lépe konvergují a jsou efektivnější při velkém množství možných akcí, protože jejich cílem je se učit skupinu zadaných parametrů. Nevýhodou bývá menší efektivita využití zkušeností. [20, 23]

2.6 Imitační učení

Zpětnovazební učení vycházelo z principu hledání optimální taktiky π agenta pomocí nahodilé interakce v prostředí. Ačkoliv je tento způsob učení agenta v dnešní době velice populární a bezesporu má své místo, jedno z velkých úskalí je délka učení. Pro dosažení nějakého výsledku agent musí projít velkým množstvím iterací pokus omyl. Toto může být ve skutečném světě náročné kvůli ceně učení. Zároveň se může zdát zbytečné nechat agenta v prostředí naslepo tápat, když existuje už předem definovaný postup, ve kterém se agenta snažíme vytrénovat. Tyto a další problémy se snaží vyřešit imitační učení [9].

Cílem imitačního učení je pro agenta přijít na optimální taktiku π, ale na rozdíl od zpětnovazebního učení zde agenti nedostávají za své akce odměnu, nýbrž mají místo toho k dispozici *demonstraci* vytvořenou expertem. Tato demonstrace je ve formě *trajektorie* $\tau = \{s_0, a_0, s_1, a_1, ..., s_n, a_n\}$, reprezentující množinu stavů a optimálních akcí. Je možno si představit například v závodní hře hráče, který projede s vozidlem okruh. Agenti se v metodách imitačního učení poté snaží z takto získaných demonstrací naučit π tak, aby jim co nejblíže odpovídala [32].

Imitační učení můžeme rozdělit na tři typy:

- Behavioral Cloning
- Direct Policy Learning
- Inverse Reinforcement Learning

Behavioral Cloning by se dal přeložit jako klonování chování a tento překlad blíže napoví, jak tento typ imitačního učení funguje. Expert poskytne demonstraci, která se uloží ve formě dvojic stavů a akcí jako trajektorie τ. Nad získaným datasetem se potom provádí vybraná metoda učení s učitelem [40]. Problém nastává, pokud není dostatek demonstračních vzorků a takto vytrénovaný agent se objeví ve stavu, který nebyl demonstrován. Toto by se dalo vyřešit, pokud by měl agent přístup k expertovi poskytujícímu demonstrace při trénování. Na tomto předpokladu stojí *Direct Policy Learning* – začne se získáním prvotní demonstrace od experta a provede se učení s učitelem, získaná taktiku π je použita v prostředí a od experta jsou získány nové informace jaké akce provést v nedefinovaných stavech. Tento postup se opakuje, dokud není dosaženo spokojenosti, respektive dokud trénování nekonverguje. V *Inverse Reinforcement Learning* se na rozdíl od předchozích dvou přístupů agent neučí π přímo, ale nejdříve se z demonstrací pokusí odhadnout funkci odměn $R(s, a)$, kterou potom použije na získání taktiky π pomocí nějakého zpětnovazebního algoritmu [9, 15]. Jedním z algoritmů stavícím na tomto principu je *GAIL*, který využívá právě i sada nástrojů ML-Agents.

Kapitola 3

Unity

Unity je multiplatformní herní engine vzniklý v roce 2004. V roce 2019 bylo v tomto enginu do světa vypuštěno více než 50% videoher napříč spektrem konzolí, mobilních telefonů a osobních počítačů [39]. Unity podporuje vývoj na Windows, iOS, Playstation a mnoho dalších.

Herní engine je virtuální prostředí určené primárně pro vývoj her a k tomu nabízí celou řadu nástrojů, jako například skriptovací API, fyzikální engine, nástroje pro tvorbu umělé inteligence a další prostředky běžně používané pro tvorbu her. Poskytuje tedy vyšší míru abstrakce nad jeho jednotlivými částmi a umožňuje jejich snadnější spojení [1]. Na trhu existuje celá řada jiných herních enginů, jako *Unreal Engine*, *CryEngine* či *GameMaker*, a častou praktikou herních studií bývá vývoj vlastních proprietárních herních enginů.

Unity ale neslouží pouze k vývoji videoher. Používá se také ve filmovém průmyslu pro tvorbu animovaných filmů, kde jeho velká síla tkví v renderování scény v reálném čase, nebo k realizaci architektonických návrhů. Dále je možné ho použít pro výukové účely, jako je třeba simulace práce s pacientem pro mediky, nebo manipulace s technikou ve strojírenství [34].

3.1 Prvky softwarové architektury

Základem tvorby aplikací v Unity jsou *scény*, ve kterých se nacházejí *herní objekty*. Celková aplikace je potom sbírka takovýchto scén, které aplikace na základě kontextu přepíná.

3.1.1 Herní objekt

Základním stavebním kamenem v Unity je *GameObject* [38] (dále herní objekt). Všechno, co se nachází v rámci Unity scény 3.1.3, je herním objektem. Ten z hlediska softwarové architektury reprezentuje pouze kontejner pro *komponenty* 3.1.2. Komponenty samotné potom definují chování a vlastnosti daného objektu. Herní objekt je možné vytvořit prázdný, obsahující pouze komponentu typu `Transform`, případně jsou k dispozici předdefinované herní objekty reprezentující kameru, světlo, primitivní tvary a podobně.

3.1.2 Komponenta

Komponenta [38] je vlastnost herního objektu, která definuje jeho funkcionalitu. Unity obsahuje řadu předpřipravených komponent, obstarávajících základní funkčnost, jako například `Rigidbody`, starající se o fyzikální simulaci objektu, různé typy kolizní geometrie, které se

starají o kolize s jinými objekty obsahujícími tyto komponenty. Každý herní objekt obsahuje komponentu `Transform`, pomocí níž je možné ve scéně objekt prostorově pozicovat.

3.1.3 Scéna

Scéna [38] by se dala pochopit jako prostor pro uspořádání herních objektů. Takovýchto scén aplikace většinou obsahuje více. Scéna může být třeba herní nabídka, či různé herní úrovně. Scénu uživatel vidí a spravuje v náhledu scény, neboli *Scene View*. Do této scény se přidávají jednotlivé herní objekty (kamery, terén, postavy). V rámci scény je možné vidět prostorové uspořádání těchto herních objektů a je možné s nimi manipulovat pomocí dostupných nástrojů.

3.2 Editor Unity

K tvorbě aplikací v enginu Unity se používá *editor*. Ten nabízí celou škálu nástrojů pro tvorbu prostředí. S výjimkou nabídky nástrojů, která se vždy nachází na horní straně editoru, jsou všechny součásti editoru upravitelné a přemístitelné.

3.2.1 Inspektor

Jednotlivé komponenty herních objektů je možné upravovat a sledovat v *inspektoru* [38]. Ten zobrazuje jeden herní objekt a všechny jeho komponenty. Inspektor je možné používat buď v normálním, nebo debugovacím režimu. V prvním z nich je možné upravovat hodnoty parametrů jednotlivých komponentů a to i v případě skriptů, kde je možné upravovat hodnoty jednotlivých proměnných (pokud byly nastaveny jako veřejné), v debug režimu je potom možné sledovat jejich změny za běhu aplikace. Umožňuje také komponenty přidávat, mazat, či měnit jejich pořadí.

3.2.2 Náhled projektu

Náhled projektu (*Project view)* [38] je část editoru, sloužící k procházení adresářové struktury projektu a jednoduchému přístupu ke všem souborům, které k projektu uživatel používá. Lze v něm jednoduše vyhledávat pomocí zabudovaného vyhledávacího pole.

3.2.3 Hierarchický náhled

V této části editoru má uživatel přístup k seznamu herních objektů ve scéně. Je možné mít v tomto okně otevřeno více scén, každou s vlastními přidruženými herními objekty. Tyto herní objekty je dále možné nořit pod další a tím vytvořit hierarchii herních objektů. Objekty, které takto dědí a jsou sdruženy pod otcovským objektem, z něj mohou dědit některé vlastnosti jako *tagy* nebo vrstvu (*layer*). Objekty také s otcovským objektem sdílejí hodnoty komponenty `Transform`. To znamená, že pokud se pohne otcovský objekt, stejně se přesunou i všechny objekty na něj navázané. Jejich vlastní hodnoty `Transform` potom určují prostorové umístění vůči otcovskému objektu [38].

3.2.4 Konzole

Editor také obsahuje konzoli [38], zobrazující výstražné či chybové zprávy, a to jak před spuštěním, tak za běhu aplikace. Dále je možné konzoli využít k vypsání ladících zpráv, jako například kumulativní odměnu agenta.

3.2.5 Další nástroje

V editoru se nad rámec výše zmíněných částí nachází nespočet dalších nabídek a jeho rozložení se dá upravit dle potřeb uživatele. Za zmínku stojí jistě *Game View*, neboli náhled hry, ve kterém se zobrazí běžící aplikace, pokud se ji uživatel rozhodne spustit pomocí tlačítka *Play*. Dále je možné v editoru spravovat balíčky pomocí rozhraní správce balíčků (*Package Manager*), kde je možné do projektu přidávat nebo z něj mazat různé balíčky přidávající do editoru (případně aplikace) další funkcionalitu. Příkladem takového balíčku je například i sada ML-Agents. V neposlední řadě se dá také v editoru připojit do Unity Asset Storu a z něj si do projektu vybrat z mnoha dostupných *assetů* (assetem je myšlen nějaký stavební díl, ze kterého se projekt skládá – modely, textury, ...).

Kapitola 4

ML-Agents

Následující kapitola se věnuje sadě nástrojů ML-Agents – platformě pro trénování inteligentních agentů. Tato sada je open-source projekt a funguje jako balíček pro herní engine Unity. Ten disponuje rozsáhlými možnostmi pro tvorbu prostředí, ve kterém se agenti nacházejí. ML-Agents používá k trénování agentů Python API a nabízí podporu algoritmů Soft Actor Critic [11] a Proximal Policy Optimization [28] (zpětnovazební učení 2.5). Dále je možné k trénování využít síly imitačního učení 2.6, a to Behavioral Cloning, nebo Generative Adversarial Imitation Learning (*GAIL*) [37]. Nejprve se kapitola zaměří na jednotlivé součásti ML-Agents, tedy co tento framework pohání a jak jsou na sobě jednotlivé součásti závislé. Kapitolu završuje popis, jak funguje a co umí agent v rámci této sady nástrojů a co je potřeba k jeho sestrojení.

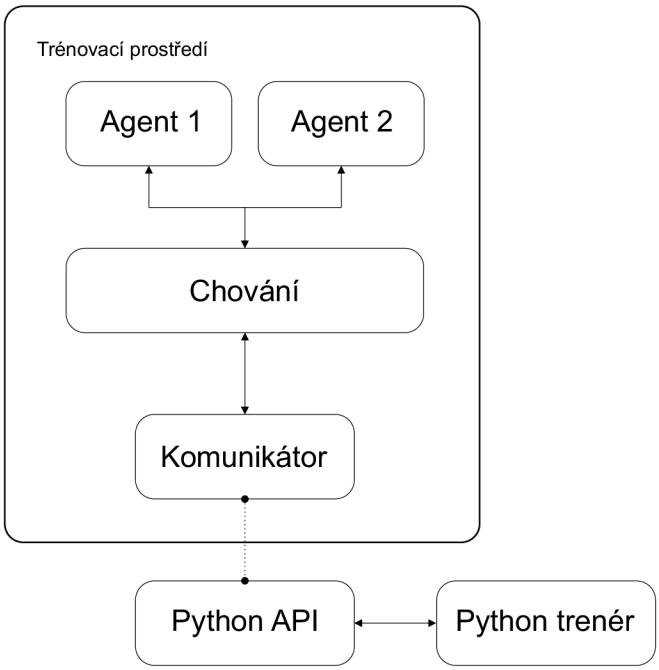
4.1 Klíčové součásti ML-Agents

Sada ML-Agents se skládá z několika klíčových součástí (viz obrázek 4.1). První částí je trénovací prostředí – to obsahuje veškeré komponenty v Unity scéně a agenty s nastavenými parametry. Každý agent musí mít definované chování. To určuje, jaké sbírá z prostředí vjemy, kolik mají k dispozici akcí a v jakém stavovém prostoru jednají (spojitý/diskrétní). Agent sbírá vjemy a odměny a na základě nich vyhodnocuje, jaké provede akce. Takto nastavené chování je připravené k trénování. Zároveň se ale k chování dá připojit i již vytrénovaný model neuronové sítě a ten potom v reálném čase tvoří rozhodování daného agenta.

Další důležitou částí celého procesu ML-Agents je Python Low-Level API. Toto API pracuje mimo samotné prostředí Unity a komunikuje s ním za použití systému *gRPC* (framework pro vzdálená volání procedur) [10]. Zajišťuje ovládání trénovacího prostředí a použití trénovacího algoritmu. Pokud by uživatel nechtěl použít nějaký z dříve zmíněných algoritmů připravených jako součást sady, je možné si definovat vlastní [37]. Zároveň je možné použít Gym API [22] a s ním kompatibilní algoritmy k trénování agentů v ML-Agents.

4.2 Agenti v ML-Agents

Agenti jsou ve frameworku implementováni jako třída *Agent*. Pro vytvoření agenta v rámci Unity scény je zapotřebí připojit k nějakému hernímu objektu skript, dědící z této třídy, který popisuje jeho chování. Agenti potom jednají na základě tohoto skriptu v epizodách.



Obrázek 4.1: Zjednodušený diagram ML-Agents

Třída agent implementuje několik metod, z nichž 3 jsou nutné pro správné fungování a jsou stěžejní součástí chování agenta. Jsou to metody:

- `OnEpisodeBegin()`
- `CollectObservations(VectorSensor)`
- `OnActionReceived(Single[])`

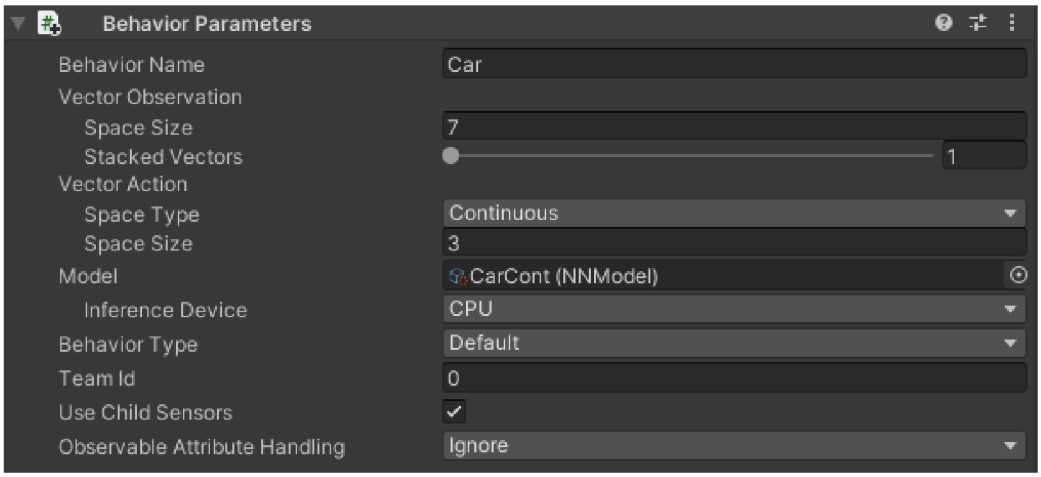
Jak již název napovídá, první z metod je volaná na začátku každé epizody a slouží k uvedení agenta (či prostředí) do počátečního stavu definovaného v těle metody.

V metodě `CollectObservations(VectorSensor)` rozhoduje uživatel o vjemech, které agent získává. Může se jednat například o jeho rotaci (`Transform.rotation`), výpočet vzdálenosti od nějakého objektu, nebo proměnnou obsahující pro agenta důležitou hodnotu. Vjemy, jež je možné agentovi předložit, jsou omezené datovým typem, který pojme objekt `VectorSensor`.

Poslední zmíněná metoda je možná metodou nejdůležitější, protože v ní uživatel definuje samotné chování agenta. Agent získává pole typu float nebo integer (v závislosti na tom, zda se jedná o spojitý nebo diskrétní akční prostor) o předem definované délce. Podle hodnot jednotlivých prvků v tomto poli uživatel definuje chování agenta.

Agentovo chování je tedy analogické k obecné definici agenta – obdrží vektor pozorování a na základě jeho hodnot a dříve získaných odměn volí neuronová síť z dostupných akcí co udělá, dokud nesplní nějakou ukončovací podmínku, nebo neuplyne stanovená doba trvání epizody.

Nad rámec třídy *Agent* je potřeba také nakonfigurovat parametry chování pomocí třídy *Behavior Parametres*. Na obrázku 4.2 je možné vidět nastavení skriptu po připojení k hernímu objektu v editoru Unity. Je možné zde nastavit jméno, na jakém zařízení se agent učí



Obrázek 4.2: Třída *Behavior Parameters*, jak je použita v aplikaci *Racer*

(CPU/GPU), či Team Id (využívané při Self-Play, viz sekce 5.1.1). Parametr *Vector Observation* určuje velikost pole `VectorSensor` neboli počet vjemů, které agent obdrží. Možnost *Stacked Vectors* umožňuje agentovi obdržet i vektory z předešlých kroků, což může být v některých případech užitečné, poněvadž to umožňuje agentovi sledovat jednotlivé prvky pole a jejich změny v čase.

Pomocí *Vector Action* uživatel rozhoduje o akčním prostoru a jeho parametrech - zda jedná agent v diskrétním nebo spojitém prostoru a kolik má k dispozici akcí. Pokud je zvolen *Behavior Type* typu *Inference*, je možné připojit a používat už vytrénovaný model, který se postará o ovládání agenta.

Chování může být trojího typu: *učení* – která slouží k trénování neuronové sítě, *heuristika* – kdy Agent jedná dle předem napsaných pravidel v kódu a *inference* – což je využití vytrénované neuronové sítě v režimu učení. Tuto funkcionalitu zajišťuje engine *Barracuda* [36]. K předepsání pravidel pro režim heuristiky slouží metoda `Heuristic(Single[])` a jejím častým využitím bývá přímé ovládání agenta uživatelem.

ML-Agents představuje několik režimů trénování agentů. Trénování agentů jednotlivě – klasický přístup jednoho agenta v prostředí. Umožňuje také simultánní trénování více agentů. Jednak v kooperaci, tedy že agenti sdílejí cíl, jednak je možné i kompetitivní trénování, kdy spolu agenti soupeří. Je také k dispozici trénování více agentů v rámci ekosystému, kde na sobě jednotliví agenti nezávisí a každý má v tomto prostředí své cíle.

Trénování jako takové neprobíhá v reálném čase, ale je možné ho zrychlit. Je ovládané mimo Unity za pomoci open source python knihovny *PyTorch* a lze ho sledovat během i po skončení pomocí vizualizačního nástroje *TensorBoard*, nebo přímo v editoru Unity.

Mimo vjemy, které agent získává z Vector Observation, je možné k jeho hernímu objektu připojit sadu raycastů (paprsky o předem definované délce, vycházející z, a mířící do, určeného bodu) ve formě komponenty *RayPerceptionSensor*. Proces vrhání paprsků lze nastavit tak, aby byla interakce mezi paprsky a objekty ve scéně omezena pouze na objekty ve specifických vrstvách, případně na objekty s určitými tagy. Jednotlivým paprskům je možné nastavit délku a směr. Je také možné určit jejich počet a maximální úhel, pod kterým jsou z objektu vysílány. Sada nabízí implementaci pro 2D i 3D prostředí.

V neposlední řadě je trénování možné upravit pomocí konfiguračního souboru obsahujícího jednotlivé hyperparametry, jako například diskontní faktor γ, kolik zkušeností potřebuje agent nasbírat pro aktualizaci modelu, nebo délku trénování (určenou počtem kroků

simulace). Rozhoduje se tu zároveň o použití modulů pro imitační učení. Pokud jej nicméně uživatel chce použít, je třeba pro agenta nejdříve nahrát demonstrace, k čemuž slouží třída *Demonstration Recorder*.

4.3 Signály odměn

Odměny jsou hlavní faktor, určující jakou akci si agent zvolí a ve výsledku jakým směrem se bude ubírat jeho trénování. V ML-Agents je možné agentům přidělit různé typy odměny, se kterými potom pracují. Těmto typům se říká signály. Ve většině případů je možné se setkat s odměnou z prostředí, neboli odměnou vnější (*extrinsic*). To je v případě ML-Agents odměna, kterou uživatel definuje a přiděluje přímo z kódu (většinou) za splnění různých podmínek. Vnitřní (*intrinsic*) odměny slouží k motivaci agenta chovat se určitým způsobem, který povede k maximalizaci vnější odměny. Jinými slovy pomáhá agentovi učit se z prostředí lépe. ML-Agents nabízí moduly pro 3 vnitřní signály odměn.

Prvním je modul `gail`, který využívá imitační učení a demonstrace. Agent se snaží co nejblíže odhadnout taktiku π z demonstrace (na rozdíl od klonování chování, které se snaží co nejblíže demonstrace napodobit). Funguje na principu GAN (Generative Adversarial Networks), z kterého má odvozené i jméno. Jsou přítomné dvě neuronové sítě. První se rozhoduje, jaké akce agent v daný moment agent učiní (generator). Druhá síť (discriminator) hodnotí, jak blízko byla daná dvojice akce/pozorování té z demonstrací a podle toho ohodnotí první síť odměnou. První síť se snaží tuto odměnu maximalizovat a vytvořit co nejlepší „podvrh“, druhá síť se naopak učí rozpoznávat, co vytvořila první síť a co je ze skutečné demonstrace. Tímto se postupně obě sítě zlepšují. Modul gail lze použít i bez vnější odměny, potom se agent učí pouze z demonstrací [37].

Modul `curiosity` je vhodný pro prostředí s malým výskytem odměn (sparse reward environment). Lépe řečeno pro prostředí, kde může být pro agenta složité udělat akci, za kterou bude odměněn. Snaží se v takovémto prostředí vyřešit klasický problém zpětnovazebního učení *exploration vs exploitation*. Jak již název napovídá, vychází z myšlenky zvídavosti. Vnitřní odměna agenta žene k prozkoumávání prostředí a s tím, jak bude objevovat více stavů, se zvyšuje pravděpodobnost, že najde stav, který pro něj bude výhodnější i z hlediska vnější odměny. Modul funguje na maximalizaci agentova „překvapení“. Jsou přítomné dvě neuronové sítě. První pracuje se současným a následujícím stavem, ve kterých se agent nachází. Ty zakóduje a vytváří odhad, jakou akci si agent mezi těmito stavy zvolil. Druhá neuronová síť vezme současný zakódovaný stav a akci a snaží se odhadnout následující agentův stav. Zakódování stavů se provádí z důvodů eliminace takových stavů, na které neměly agentovy akce vliv [24]. Rozdíl mezi odhadem a skutečným stavem potom určuje míru překvapení agenta. Čím větší rozdíl, tím dostává agent větší vnitřní odměnu. Tímto je odměňován za prozkoumávání prostředí a odkrývání nových stavů [14].

Další možností řešení problému prostředí s nízkým výskytem odměn je modul `rnd` vycházející z článku *Exploration by Random Network Distillation* [4].

Kapitola 5

Návrh

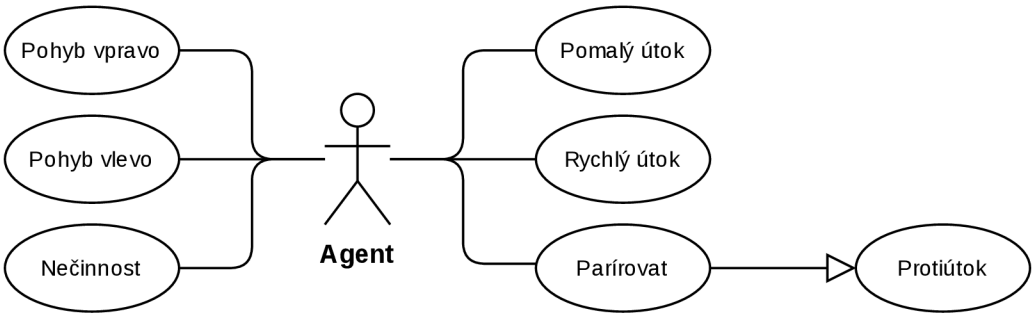
Tato kapitola se bude zaobírat návrhem demonstračních aplikací, v nichž byla použita sada ML-Agents – proč byly zvoleny právě tyto koncepty, jak se agenti v jednotlivých aplikacích chovají a jaké mají agenti cíle.

5.1 Volba aplikací

Prvotním problémem k obšírnější demonstraci možností sady ML-Agents v různých situacích je zvolit takové prostředí, které bude dostatečně komplexní pro zajímavou ukázku s možností porovnání jednotlivých scénářů, a trénování přitom bude konvergovat v rozumném čase. Volba nakonec padla na rozdělení projektu do několika aplikací, které budou zaměřené na řešení různých problémů.

5.1.1 Fighter

Jako první aplikace byla zvolena bojová hra inspirovaná klasikami, jako *Street Fighter* (na snímek ze hry je možné se podívat na obrázku 5.2), nebo *Mortal Kombat*, kde se střetávají na bojišti dvě postavy ovládané buď hráči nebo umělou inteligencí s cílem zneškodnit jako první soupeře a tím souboj vyhrát. Konceptem má nicméně aplikace blíž k méně známé hře *Footsies* z roku 2020 od studia *HiFight*, což je de facto typická bojová hra očesaná na absolutní minimum. Není zde klasický indikátoru zdraví, jehož hodnotu soupeři útoky postupně snižují, až dokud jednomu nepadne na nulu. Hráče zde vyřadí jedna dobře načasovaná rána a kolem toho se odvíjí celý zápas, je zde také omezená možnost blokování a používání speciálních útoků.



Obrázek 5.1: Diagram zobrazující dostupné akce v aplikaci Fighter

Jednoduchost této hry byla inspirací pro demonstrační aplikaci zvanou *Fighter*, ve které se nacházejí dva bojovníci v aréně, kterým je umožněn pohyb po horizontální ose, a každý má 3 životy. Dále mají k dispozici 2 útoky – pomalý, který ale pokrývá velkou vzdálenost, a rychlejší, jež má menší dosah. Poslední věc, kterou mohou využít, je parírování, kdy po použití tohoto pohybu vyblokují soupeřův pomalý útok a zasáhnou ho za 2 poškození. V opačném případě jsou nicméně nechráni na pospas protivníkovi, protože tento pohyb uživatele na značnou dobu znehybní.

Jelikož bojová hra je ve své podstatě soutěží dvou hráčů, je zde využita funkcionalita ML-Agents zvaná *Self-Play*, což může být volně přeloženo jako hraní sám se sebou. Jedná se o možnost trénování identických agentů (agentů majících stejné vjemy z prostředí a dostupné akce) proti sobě. Jelikož odměna agenta získaná v konkrétní epizodě závisí na schopnosti jeho protivníka, kdy slabší soupeř bude znamenat zpravidla vyšší odměnu a naopak, není kumulativní odměna dobrým indikátorem kvality agenta. Proto je pro implementaci Self-Play v ML-Agents použit systém Elo[1], který určuje úroveň schopnosti agenta v trénované aktivitě [37].

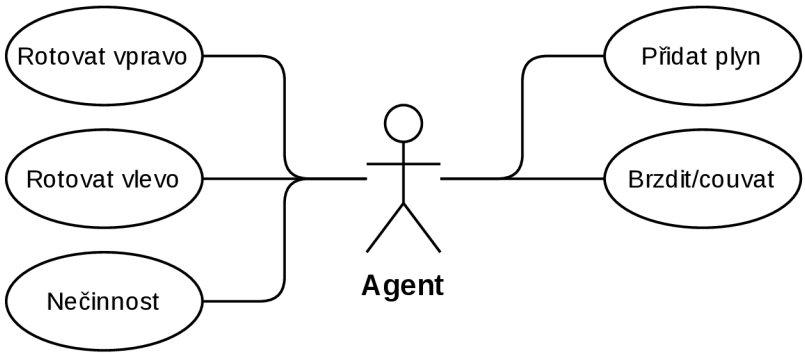
Obrázek 5.2: Obrázek ze hry *Street Fighter 2* (1991)

5.1.2 Racer

Jelikož prostředí první aplikace bylo dvoudimenzionální, bylo nasnadě vytvořit něco ve 3D prostoru. Prvotní myšlenky směřovaly k trénování automatického parkování vozidla, tedy nalezení volného parkovacího místa v prostoru a potom samotné provedení parkování. Tuto variantu jsem nicméně zavrhl z důvodu nesouladu s celkovým pojetím práce, kde jsem se chtěl držet ve sféře blížící se klasickým videohrám. Výsledná volba nicméně není tomuto konceptu příliš vzdálená, jelikož aplikace *Racer* demonstruje schopnosti vozidla naučit se v co možná nejlepším čase projet nachystanou trať.

[1]Elo je systém hodnotící úroveň schopnosti hráče, nebo družstva ve hrách s nulovým součtem (výhra jednoho hráče znamená prohru druhého) pro dvě soupeřící strany. Původně vytvořen Arpadem Elem pro hru šachy s cílem férově ohodnotit hráče na základě jeho výsledků ve velkém počtu her. Hráč získává hodnocení Elo (Elo rating) ve formě čísla. [19]

Aplikace je zaměřená na jednoho agenta s cílem co nejlépe projet trať samostatně, nikoliv na tvorbu agenta, který by v reálném čase soupeřil s jiným vozidlem. Agent má k dispozici možnost pohybu dopředu a dozadu a otáčení kolem své osy. Jízdní model je tedy zjednodušen na minimum.

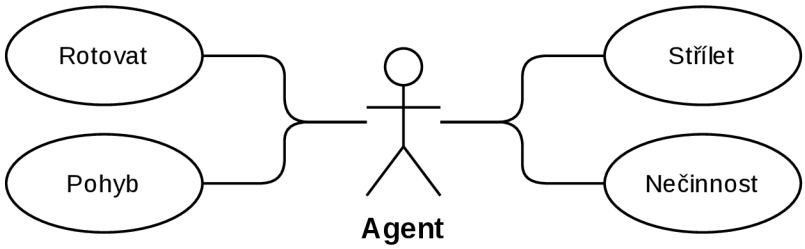


Obrázek 5.3: Diagram zobrazující dostupné akce v aplikaci Racer

Neboť je akce přidávání rychlosti a zatáčení proveditelná jak diskrétně (na segmenty) tak spojitě, tak tato demonstrační aplikace také slouží k porovnání použití těchto dvou akčních prostorů, respektive ke srovnání kvality výsledného řešení oproti náročnosti na trénování.

5.1.3 Nightmares

Tvorba aplikací, které jsou od základů budované s vědomím, že budou použity pro strojové učení, je věcí, kterou zkoumají první dvě demonstrační aplikace. Tato využívá projekt *Survival Shooter* [35], aby se pokusila odpovědět na otázku, jak náročné (či zda je to vůbec možné) je vycvičit umělou inteligenci v již vytvořené hře.



Obrázek 5.4: Diagram zobrazující dostupné akce v projektu Survival Shooter

Survival Shooter je hrou, ve které se hráč v uzavřené aréně snaží získat co nejvíce bodů střílením donekonečna se objevujících nepřátel, k čemuž mu slouží zbraň s neomezeným počtem nábojů. Zároveň ale musí dbát na to, aby si od nepřátel udržel odstup. Pokud se totiž nepřítel přiblíží na bezprostřední vzdálenost, tak začne hráče poškozovat. Hra končí v moment, kdy takto nepřátelé hráče zneškodní. Agent se může pohybovat po ose X a Z, otáčet se kolem vlastní osy a střílet.

Tento projekt byl zvolen kvůli relativní jednoduchosti originální implementace a přímočarosti stanovení cílů pro agenta. Agent by hypoteticky v tomto případě měl po vytrénování sloužit jako AI spoluhráč.

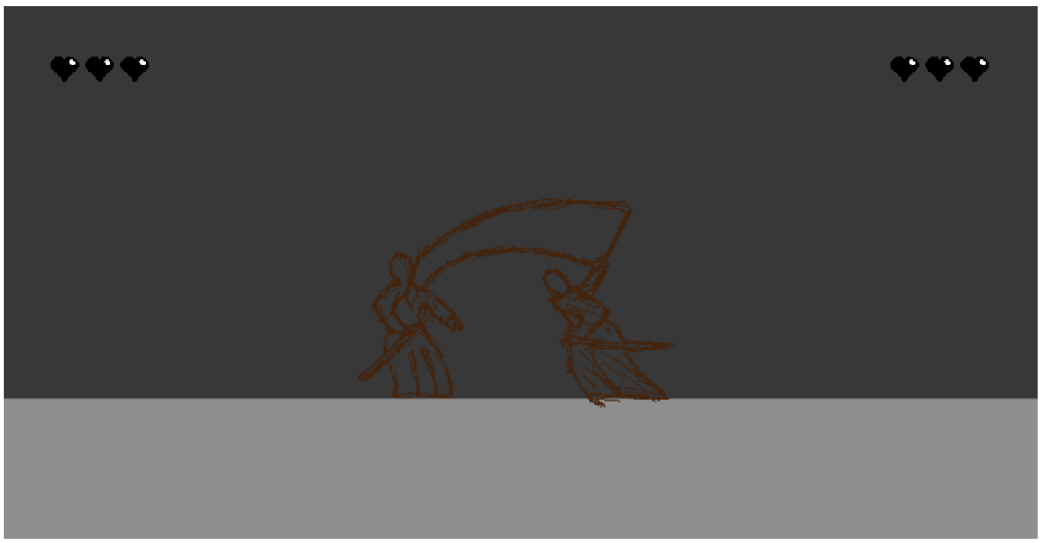
Kapitola 6

Implementace

Všechny aplikace jsou implementovány v herním enginu Unity a všechny skripty byly psány v jazyce C#.

6.1 Demonstrační aplikace Fighter

V této sekci budou popsány implementační detaily první aplikace. Obrázek 6.1 je ukázkový snímek z této aplikace a je na něm možné vidět, jak jeden bojovník zasahuje druhého.



Obrázek 6.1: Obrázek z demonstrační aplikace *Fighter*.

6.1.1 Prostředí

Agenti se v této demonstrační aplikaci pohybují po aréně o limitované délce, toto je standardní praktika, která zabraňuje nekonečnému ústupu jednoho z bojovníků. Podlaha arény je implementována obdélníkem o dané délce a konce arény jsou neviditelné stěny po jeho hranách. Prostředí také mění kamera, protože po jejích krajích jsou také neviditelné překážky zabraňující v pohybu. Důvodem je zabránit některému z hráčů vystoupit ze záběru kamery. Zajímavostí implementace této kamery je její rozšiřování v závislosti na vzájemné vzdálenosti protivníků. Kamera má maximální a minimální hodnotu, na kterou se může

od bitevního pole vzdálit, a tím omezuje odstup, na který se od sebe mohou hráči vzdálit. Tuto hodnotu získáme přičtením minimální velikosti kamery k podílu normalizované hodnoty vzdálenosti protivníků ku jejich nejvyššímu možnému vzdálení:

```
float ortho_size = (distance*(ortho_max-ortho_min))/dist_max+ortho_min;
```

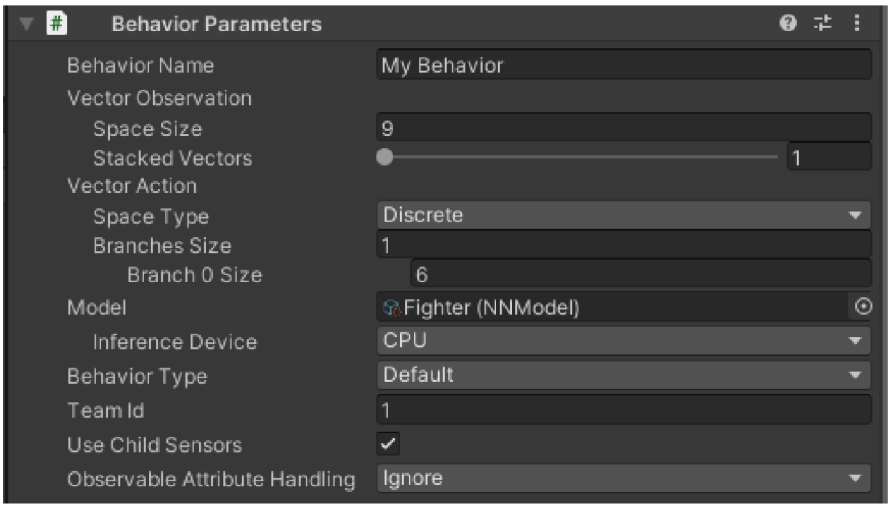
Na její okraje jsou poté umístěny překážky zabraňující pohybu. Kamera je implementována ve skriptu *CameraControl*.

6.1.2 Agent

Agenti jsou v této demonstrační aplikaci dva, a to spolu soupeřící bojovníci. Vizuálně jsou tvořeni animovanými 2D obrázky, neboli *2D sprity*, které zobrazuje komponenta *Sprite Renderer*. Použití této metody znamená, že veškeré animace se skládají z jednotlivých obrázků. Inspirací pro výtvarné ztvárnění byla originální verze hry *Samurai Shodown* – zde byla čerpána inspirace jednak východní estetikou (samurajové), jednak také použitím pixel artu. Obrázky byly vytvořeny v aplikaci Pixel Studio pro iPad. Herní objekt agenta zároveň musí obsahovat komponentu *Animator*, která v závislosti na okolnostech bojovníka animuje. Na toto je navázána hlavní herní mechanika – útoky.

Útoky

Jednotlivé útoky jsou zpracovány tak, že se v daný moment animace aktivuje jejich kolizní geometrie na určenou chvíli – pokud je detekována kolize se soupeřem, provádí se kód útoku. Jednotlivé útoky mají nicméně také periodu neaktivity, a to na začátku (bezprostředně po aktivaci) a ke konci animace, toto zamezuje bezcílnému používání útoků. Speciální případ útoku je parírování, které nedetekuje kolizi se soupeřem, nýbrž se soupeřovým útokem. V případě že dojde ke kolizi, aktivuje se speciální útok se zvýšeným poškozením.

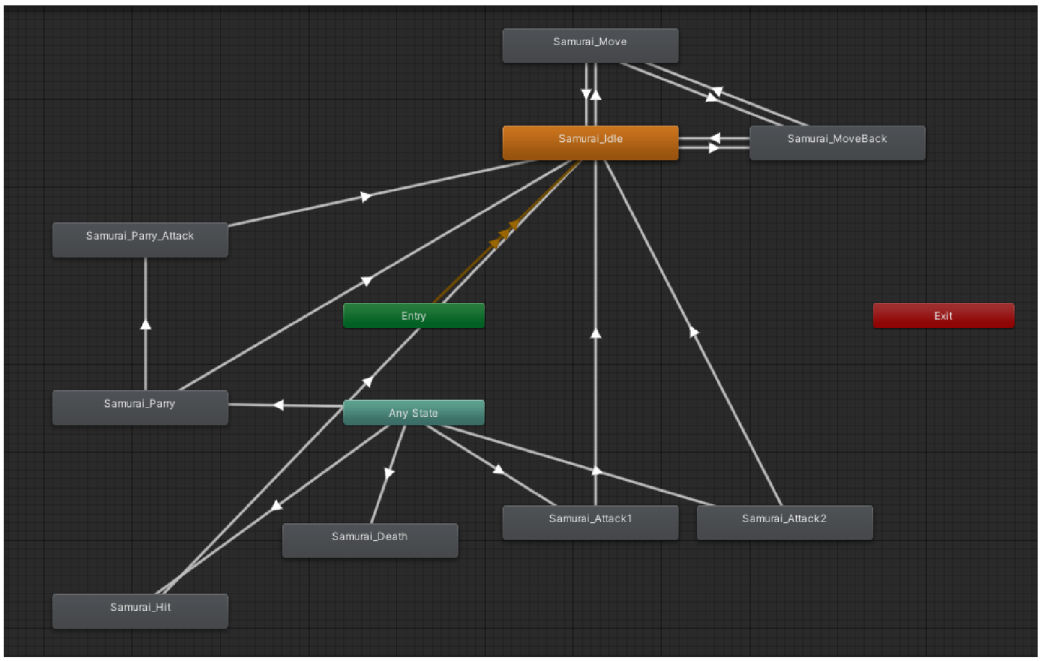


Obrázek 6.2: Nastavení třídy BehaviorParameters pro demonstrační aplikaci Fighter.

Vjemy

Na obrázku 6.2 je možné vidět, že má agent přidělený vektor pozorování o velikosti 9. Jednotlivé vjemy a další chování je popsané ve skriptu *PlayerMovement*, což je třída dědící ze

třídy *Agent*. V metodě `CollectObservations()` má do vektoru přiděleny údaje o své pozici a pozici oponenta. Zbývající tři pole vektoru jsou rezervována pro jednotlivé soupeřovy útoky, tímto agent pozná, zda a který útok soupeř používá.



Obrázek 6.3: *Animator Controller* z aplikace Fighter.

Akce

Jelikož agenti jednají v diskrétním akčním prostoru, je v nastavení parametrů chování možnost určit počet větví, tyto větve určují, kolik akcí může agent provádět současně. V tomto případě má agent dovoleno v daný moment konat pouze jednu z šesti akcí, které jsou definovány v metodě `OnActionReceived()`. Jsou to pohyb po ose X (1 a 2) a tři útoky (3, 4, 5), které má agent k dispozici. Poslední akcí je nedělat nic, tedy že agent zůstane stát na místě (0).

Jak už bylo zmíněno výše, kromě provedení samotné akce je nutné ji také animovat, toho je dosaženo nastavením jednotlivých parametrů v tzv. *Animator Controlleru*. Tyto parametry v závislosti na současném stavu kontroléru fungují jako impulz k přechodu do stavu jiného. Jinými slovy kontrolér funguje jako konečný automat, kde každý stav znamená jinou animaci bojovníka (pro ilustraci viz obrázek 6.3). Akce útoků jsou tedy implementovány nastavením korespondujících parametrů a akce pohybu navíc k tomu přesouvá herní objekt.

Další parametry

Na obrázku 6.2 je také možné vidět připojený model (už vytrénovaná neuronová síť). V případě, že je Behavior Type nastavený na inferenci (nebo default), tak agent jedná podle přiloženého modelu. Důležitý parametr je také *Team Id*, ten určuje, do kterého týmu (v tomto případě týmy jedinců) daný agent patří. Díky tomuto rozdělení je potom možné trénovat agenty v módu Self-Play, viz sekce 5.1.1.

Ovládání

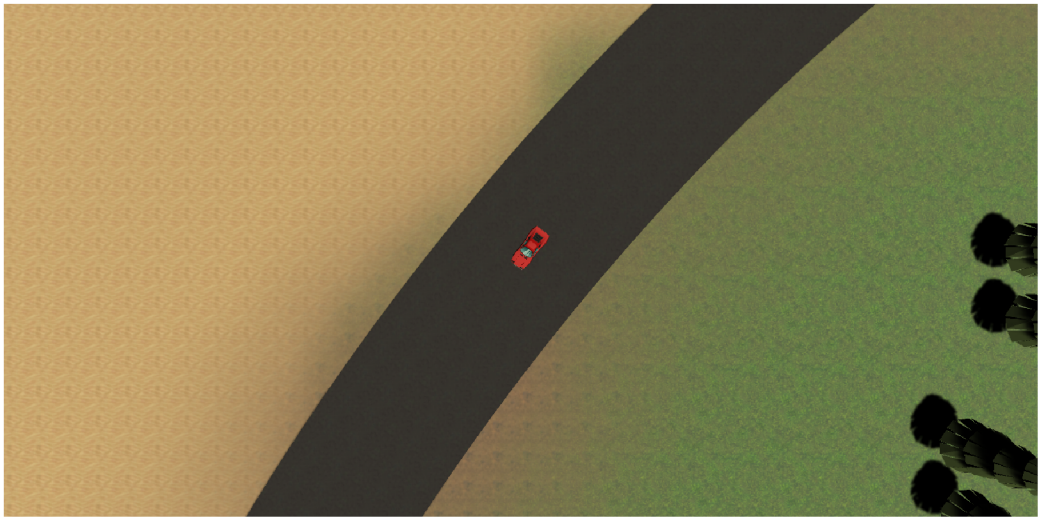
V metodě `Heuristic()` je implementováno přímé ovládání bojovníků, primárně pro ladící účely. Díky tomuto je umožněna i hra více hráčů na jedné klávesnici, byť to není primární účel aplikace v jejím současném stavu. Horizontální osy slouží k ovládání pohybu. Pro hráče na levé straně klávesy `A` a `D`, pro hráče na pravé straně šipky. Jednotlivé útoky v pořadí, v jakém byly popsány v sekci návrh, se spouštějí klávesami `J`, `K`, `D` a `Del`, `End`, `PageDown` pro hráče vlevo a vpravo v uvedeném pořadí. Metoda `OnEpisodeBegin()` slouží k přemístění bojovníků do jejich počátečních pozic.

6.1.3 Další třídy

Dále je v aplikaci přítomná třída *PlayerHealthManager*. Ta se stará o kontrolu kolizí agenta s útoky a o vyhodnocení zranění. Dále se zde za jednotlivé akce agentovi přidělují odměny (blíže v další kapitole). Současně se taky stará o animovaní ukazatele životů. Třída *Arena* zajišťuje resetování souboje, jakmile je jeden hráč vyřazen, nebo v případě že vyprší čas souboje – ten je nastaven na 30 vteřin. Jinými slovy ukončuje epizodu a nastavuje počáteční parametry pro následující epizodu.

6.2 Demonstrační aplikace Racer

Tato sekce kapitoly je zaměřena na implementační detaily demonstrační aplikace racer. Na obrázku 6.4 je možné vidět snímek ze hry.



Obrázek 6.4: Obrázek z demonstrační aplikace *Racer*.

6.2.1 Prostředí

Prostředím pro aplikaci Racer je závodní okruh. Inspirací pro tvar trati byl brněnský Masarykův okruh. Jeho model byl vytvořen v programu *Blender* pomocí Bézierových křivek a s využitím volně dostupných textur. Pro zlepšení prezentace byly také rozesety po mapě modely stromů. Dále jsou na vnějších stranách zatáček po vzoru skutečných okruhů štěr-

kové pásy, které v reálných podmínkách slouží ke zpomalení vozidla při opuštění okruhu, zde plní nicméně pouze estetický účel.

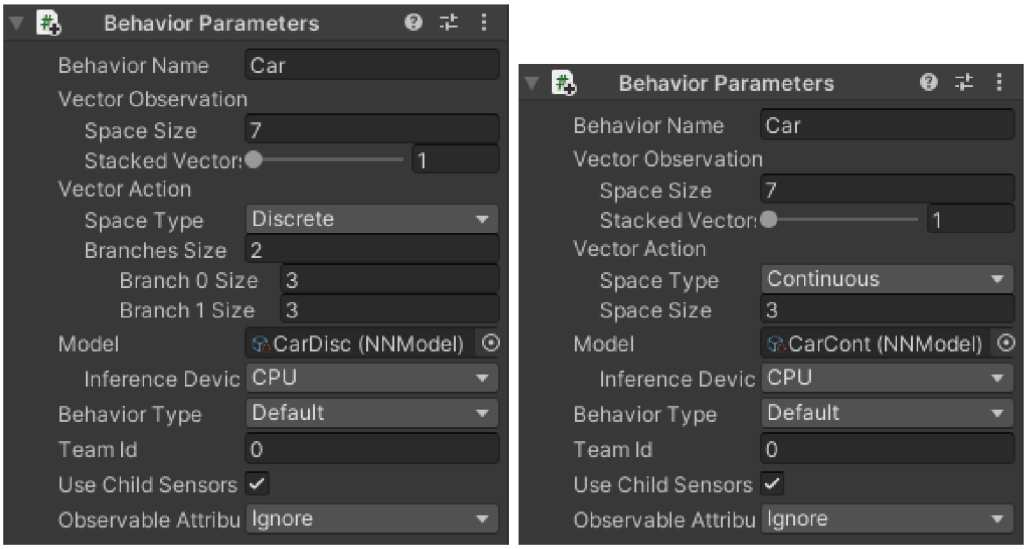
Pro potřeby trénování jsou podél celé trati rozesety neviditelné mantinely, které slouží pro agenta jako indikátor toho, že vyjel mimo trať. Další, pro agenta důležitou, částí trati jsou kontrolní body, které pomáhají agentovi v rozpoznání směru jízdy a tratí ho jistým způsobem vedou. Speciálním typem kontrolního bodu je potom blok *Start*, který pokud agent překročí a zároveň projel všechny ostatní kontrolní body, zdárně ukončuje trénovací epizodu.

6.2.2 Agent

V demonstrační aplikaci jsou vytvořeny dvě scény s totožným prostředím. Jedna scéna obsahuje agenta s diskrétním akčním prostorem a druhá se spojitým. Nastavení parametrů chování lze sledovat na obrázku 6.5.

Vjemy

Je zde vidět, že agent má v obou případech k dispozici 7 vjemů, respektive že *Vector Observation* je vektor o velikosti 7. Mimo výše zmíněné vjemy má agent také k dispozici dvojici komponent třídy *Ray Perception Sensor 3D*. Ty slouží při trénování k rozpoznávání mantinelů a kontrolních bodů a agent díky nim „vidí“ kolem sebe.



Obrázek 6.5: Nastavení třídy BehaviorParameters pro demonstrační aplikaci Racer pro diskrétní (vlevo) a spojitý akční prostor (vpravo).

Diskrétní akce

Neboť jsou rychlost a úhlová rychlost v Unity udávány strukturou *Vector3*, tak jsou obě složeny ze tří čísel typu float. Pro diskrétní akční prostor obsahuje vektor akcí dvě větve, obě o velikosti 3. To znamená, že agent dostává dva signály s hodnotami 0, 1 a 2 a může provádět dvě akce najednou. Dostupné akce jsou předepsané v metodě `OnActionReceived()`. První větev slouží k ovládání rychlosti, respektive udává směr, ve kterém se pohybuje. Druhá větev slouží k určení natočení.

Spojité akce

V případě spojitého prostoru má samozřejmě agent k dispozici daleko preciznější kontrolu jak rychlosti, tak natočení, neboť se nejedná pouze o 3 hodnoty, ale celý interval, v němž může vybírat. Jednotlivé hodnoty spojitého vektoru akcí vždy nabývají hodnot $< -1; 1 >$:

Pohyb

Pohyb a rotace agenta jsou zpracovány v metodě `FixedUpdate()`. Pohyb je řešen aplikací síly v určeném směru (pro kladné hodnoty dopředu, pro záporné naopak):

```
rb.AddRelativeForce(Vector3.forward * mag * speed * offTrack);
```

Pro rotaci je použita vestavěná metoda `Rotate()`, která otočí objekt o určitý úhel podél určené osy:

```
transform.Rotate(0, dir * rotation, 0);
```

Ovládání

Agenta je také možné ovládat přímo, a to buď pro testovací účely (kontrola funkčnosti agenta), nebo pokud chce uživatel nahrávat demonstrace. Ovládání je napsáno v metodě `Heuristic()`. Mezerníkem je možné vozidlu přidat rychlost dopředu, klávesou **S** se zrychluje v opačném směru. Šipkami se zatáčí. Alternativní volbou je ovládání na ovladači (uzpůsobené pro ovladač ke konzoli Xbox One), kde je nicméně možné jet pouze dopředu (páčka RT). Rychlost lze v případě spojitého prostoru korigovat, neboť ovladač snímá pozici zadní páčky a nenabývá pouze hodnoty 0 nebo 1 jako klávesa na klávesnici. Zatáčení je na levé analogové páčce.

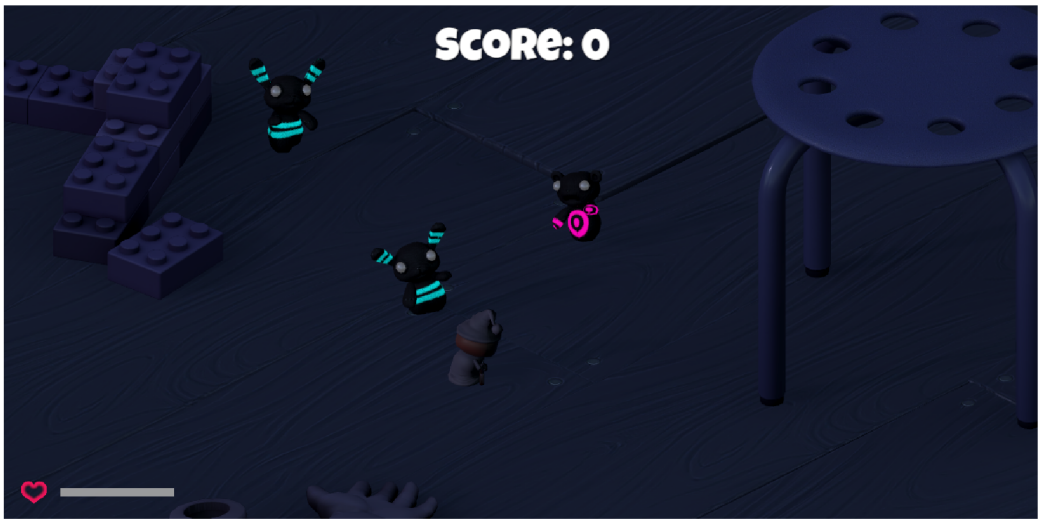
Pro pohodlné ovládání byla také vytvořena kamera, která sleduje vozidlo z ptačí perspektivy, a se zvyšující se rychlostí ho lehce „předbíhá“, aby uživatel viděl co největší plochu trati. Kamera je definovaná ve třídě *PlayerCamera*, kde se v metodě `FixedUpdate()` vypočítává její pozice dle pozice automobilu a jeho rychlosti. Kamera má současně maximální velikost, o kterou se může vzdálit (aby vozidlo stále zůstávalo na obrazovce). Pro hladký přechod kamery je při jejím nastavení použita lineární interpolace (metoda `Vector3.Lerp()`).

V metodě `OnEpisodeBegin()` se na začátku každé epizody resetují kontrolní body – nastavují se jako neprojeté, dále se vozidlo vrací do počáteční polohy a jeho rychlost je nastavena na nula.

6.3 Demonstrační aplikace Nightmares

Jak již bylo uvedeno v kapitole Návrh, demonstrační aplikace Nightmares je založena na experimentální hře Survival Shooter vytvořené společností Unity. Zpráva se tedy nebude věnovat implementačním detailům prostředí a hry samotné, ale pouze implementaci agenta a využití sady ML-Agents.

Na několika místech bylo třeba přesunout logiku z metody `Update()` do `FixedUpdate()`. Důvodem je zrychlení časového měřítka pro simulaci, metoda se potom nechovala dle očekávání, neboť implementace ML-Agents počítá právě s použitím druhé zmíněné metody. Dále bylo třeba zajistit zrušení všech instancí objektů nepřátel ve hře při skončení epizody – původní implementace po smrti hráče znovu spouštěla celou scénu, toto chování nebylo kvůli chybě v použité verzi ML-Agents (při restartu scény přestanou fungovat připojené senzory

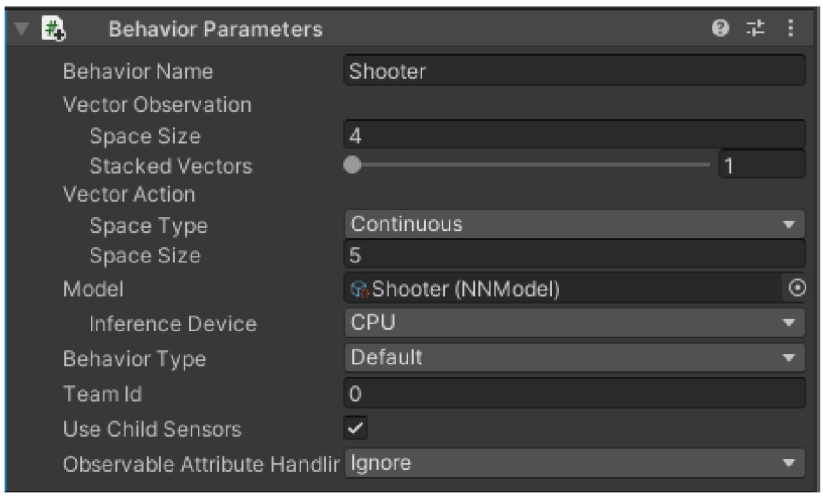


Obrázek 6.6: Obrázek z projektu Survival Shooter. Je možné vidět hráče a 3 nepřátele.

na agentovi) přípustné. Proto byla zvolena alternativa s odstraněním nepřátel a přemístěním agenta do počáteční polohy.

6.3.1 Agent

Aby bylo možné trénovat umělou inteligenci, bylo třeba do aplikace Survival Shooter [35] přidat třídu *Agent*. Pro toto byl zvolen skript *PlayerMovement*, který byl předělán, aby splňoval požadavky sady. Dále bylo nutné připojit k hráčské postavě všechny nutné skripty pro správné fungování agenta. Na obrázku 6.7 může čtenář sledovat nastavení třídy *BehaviorParameters*.



Obrázek 6.7: Nastavení třídy BehaviorParameters pro demonstrační aplikaci Nightmares.

Vjemy

Agentovy vjemy, přidělené v metodě `CollectObservations()`, jsou lokální rotace po ose Y (lokální v tomto případě znamená vůči otcovskému hernímu objektu), jeho X a Z souřadnice, určující dohromady jeho pozici v aréně a dále jeho současný počet životů. Jelikož jsou všechny tyto údaje určeny pouze jedním číslem, výsledný vektor má velikost 4. Akční prostor, ve kterém agent jedná, je v případě této aplikace spojitý. To je podobně jako u předchozího případu z důvodu preciznosti otáčení, neboť diskrétní prostor by zajistil pouze posun o fixní velikost. Navíc jsou připojeny dvě komponenty *RayPerceptionSensor3D*, díky kterým je agent schopen rozpoznat nepřátele a překážky. Důvodem pro dvě instance těchto komponent je možnost rozpoznávat nepřátele i za překážkou. Jakmile paprsek rozpozná kolizi s nějakým objektem, dále už neprochází. To znamená, že pokud je v prostředí překážka a za ní nepřítel, senzor by ho nerozpoznal. Pokud má ale dvě množiny senzorů, obě mohou rozpoznávat jiné herní objekty (podle tagů) a díky tomu je možné popsanému problému předejít.

Akce

Pohyb a otáčení byl v průběhu testování několikrát změněn (více v následující sekci). V originální implementaci aplikace je pohyb navázán na horizontální a vertikální osu (klávesy `WASD` nebo šipky) a otáčení postavy bylo realizováno myší – postava se otáčela za kurzorem. Ve finální implementaci má agent v rámci vektoru akcí možnosti pohybu, střelby a dvě položky vektoru jsou vyhrazeny pro určení rotace. Toto chování je implementováno v metodě `OnActionReceived()`.

První dvě hodnoty vektoru slouží pro nastavení hodnoty os. Další dvě slouží pro identifikaci natočení – pomocí nich je sestrojen vektor a z něj je určen, za použití metody `Quaternion.LookRotation()`, směr natočení. S tím je použita metoda `RotateTowards()`. Hodnoty pro sestrojení vektoru se pohybují v rozmezí $< -1; 1 >$. Takto byla realizace rotace oproti originálnímu řešení změněna, protože otáčení pomocí kurzoru je pro zpracování neuronovou sítí relativně komplexní řešení (myš se může vyskytovat kdekoliv na obrazovce a existuje mnoho hodnot, určujících jeho pozici). Poslední hodnota vektoru ustanovuje, zda oponent střílí, nebo ne.

6.3.2 Ovládání

Ovládání v metodě `Heuristic()` je v této demonstrační aplikaci uzpůsobeno pro herní ovladač, konkrétně byl testován model ke konzoli Xbox One. Levá analogová páčka slouží k pohybu, pravá určuje směr pohledu postavy a levý bumper (`LB`) spouští střelbu.

Kapitola 7

Experimenty a vyhodnocení

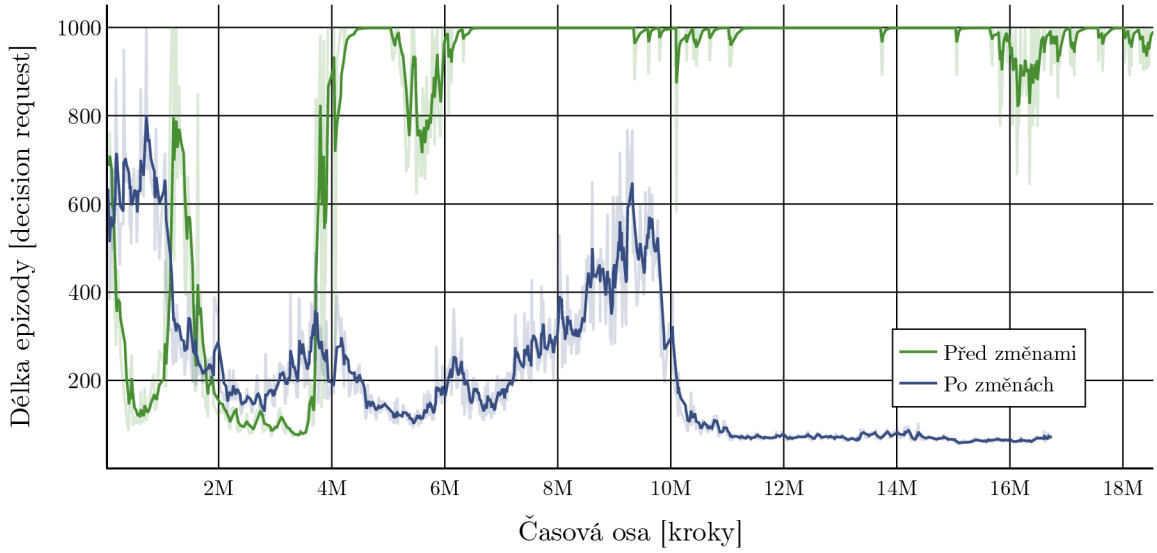
V této kapitole bude rozebráno samotné trénování agentů – jakých výsledků bylo dosaženo a jak se k nim došlo. Je zde také popsána zpětná vazba ve smyslu odměn, což je asi nejdůležitější faktor udávající, zda se agent naučí určenou úlohu. Kapitola se zaměří na nejzajímavější trénovací běhy a změny, ke kterým došlo. Na grafy bylo aplikované vyhlazení za použití klouzavého průměru. Originální hodnoty viditelné jako „stín“.

7.1 Demonstrační aplikace Fighter

V této demonstrační aplikaci bylo cílem vytrénovat bojovníka, který bude co možná nejefektivněji porážet soupeře. Jelikož byl zvolen modul Self-Play, znamená to dosažení co nejvyšší hodnoty Elo. Počínající hodnota pro hodnocení Elo byla zvolena 1200. Dále bylo nutné specifikovat signály odměny. Obecné doporučení pro distribuci odměny je udržovat jednotlivé signály kvůli stabilitě trénování v rozmezí $< -1; 1 >$. Není to nutné v případě, že je trénování stabilní i s jinými hodnotami, a i samotní autoři sady přidělují v některých ukázkách vyšší odměny. Jelikož je cílem agenta v boji zvítězit, největší odměna mu je přidělena ve chvíli, kdy tak učiní – tehdy je mu přidána (nad rámec odměn, které získal během epizody) odměna o hodnotě 5 a končí epizoda. Agent, který prohraje, o stejnou hodnotu přichází. Dále může v průběhu epizody získat agent odměnu za zasažení nepřítele útokem 1, nebo útokem 2. V obou případech agent, který zasáhl, získává odměnu 0,3 a protivník získává negativní odměnu o stejné hodnotě. Speciálním případem je potom parírování. Agent, v případě že úspěšně vyblokuje útok, aktivuje speciální útok, který když zasáhne, přiděluje odměnu 0,6 a -0,6.

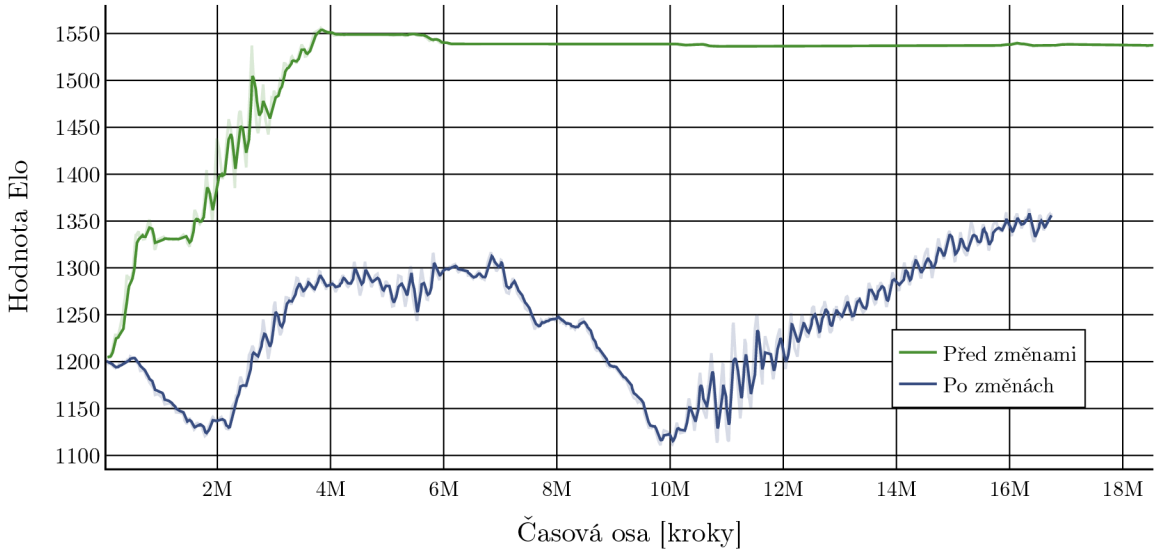
Při prvotních experimentech byly zavedeny postihy za používání útoků a v případě zranění získával agent vyšší negativní odměnu (zhruba o jednu třetinu). To mělo vést k většímu důrazu na přežití oproti vítězství a konzervativnějšímu používání útoků. Ačkoliv hodnota Elo na začátku trénování stoupala, při delším běhu simulace se začalo vyskytovat nežádoucí chování a hodnota stagnovala. Agenti se od sebe pouze vzdálili a nijak spolu neinteragovali, pouze ustupovali a případně čekali, než se soupeř přiblíží. Proto byla zkrácena maximální délka epizody, aby (v případě že se takové chování bude opakovat) mohlo trénování rychleji pokračovat. Byly také upraveny odměny za útok (tak jak jsou v současné podobě). Tím bylo pobídnuto agresivnější chování.

Dále byla pro agenta představena malá negativní odměna s každým krokem fyzikálního enginu. Po určitém počtu kroků (při časovém měřítku 1 tento počet koresponduje s 30 vteřinami) hra automaticky končí remízou. Toto nutí agenty jednat rychleji, neboť samozřejmě



Obrázek 7.1: Graf znázorňující dva trénovací běhy aplikace Fighter, ukazující změnu délky epizod v čase.

chtějí vyhrát a maximalizovat odměnu. Na obrázku 7.1 je možné vidět postupnou délku epizod u dvou běhů. Zeleně je znázorněn běh před implementací těchto změn. Je zde možné vidět, že se délka epizod postupně zvedá až k maximu, u kterého se drží. Po implementaci změn (modrý) graf ukazuje postupnou klesající tendenci. Jinými slovy je v tomto případě pro agenta výhodnější se pokusit zneškodnit soupeře co nejrychleji.



Obrázek 7.2: Graf znázorňující dva trénovací běhy aplikace Fighter, ukazující výši hodnoty Elo v průběhu trénování.

Další důležitou metrikou, na kterou je nutné se zaměřit, je hodnocení Elo. Zde nám zelený graf na obrázku 7.2 ukazuje, že v původním běhu už nebylo možné se po čase nijak zlepšit, neboť optimální strategií pro agenta bylo pouze čekat na konec epizody. Sekundární běh (modrý graf) ukazuje i po delším běhu simulace stoupající tendenci hodnoty Ela. Na

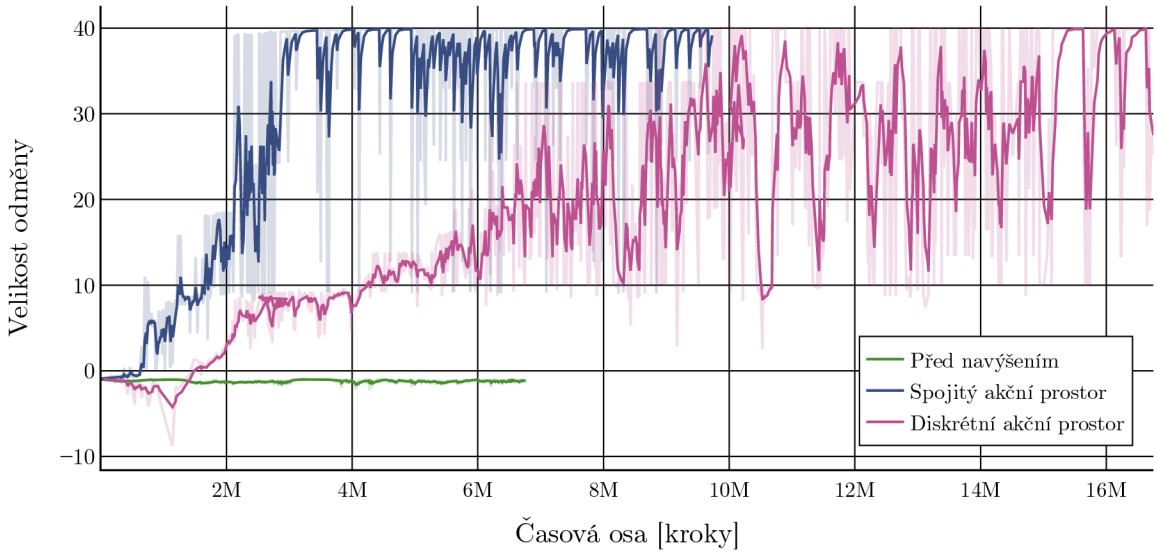
druhou stranu ale není dosaženo stejné efektivity, neboť je hodnota po stejném počtu kroků simulace menší.

Pro finální implementaci byla zvolena druhá varianta, která vedla k zajímavějšímu chování agenta v reálných podmínkách (v souboji člověka proti tomuto modelu). Pokud by si uživatel nicméně přál mít bojovníka s defenzivnějším chováním, šlo by použít variaci první zmíněné verze. Je tedy možné vidět, že relativně malé změny v prostředí a odměnách vedou k diametrálně odlišným výsledkům a typům chování.

7.2 Demonstrační aplikace Racer

Demonstrační aplikace se zaměřila na vytrénování vozidla schopného projet trať, aniž by ji opustila v co možná nejlepším čase. Byly vytvořeny dvě implementace – pro diskrétní a spojitý akční prostor. V obou případech jsou odměny přidělovány stejně a liší se jenom způsob, jakým agent provádí akce. Původním plánem bylo využít kolizní geometrii na rozpoznání, zda je vozidlo stále na ploše trati, nebo sjelo mimo. Tato metoda nebyla funkční, neboť se vozidlo o trať, která měla aktivovanou detekci kolizí „zasekávalo“. Jinými slovy trať nebyla sjízdná a bylo třeba najít jinou alternativu.

Proto jsou podél plochy okruhu rozesety malé kvádry – ty plní funkci mantinelů. Při trénování má agent k dispozici dvě sady senzorů (*RayPerceptionSensor3D*). Pomocí jedné je schopen tyto mantinely rozpoznávat. Ví tedy, jak jsou daleko a v jakém směru. Druhá sada rozpoznává kontrolní body. Ty slouží jednak jako navigační body pro agenta, aby věděl, kterým směrem se vydat, a zároveň jako podnět, aby jimi projížděl, neboť při průchodu takovýmto kontrolním bodem získává agent malou pozitivní odměnu. Důvodem pro dvě sady senzorů je možnost vidět mantinely i skrze kontrolní body, což je důležité hlavně v případě zatáček, kde by v případě jedné skupiny senzorů kontrolní bod zastínil informace o tvaru zatáčky za ním.



Obrázek 7.3: Graf znázorňující tři trénovací běhy aplikace Racer, zobrazující velikost kumulativní odměny v průběhu trénování.

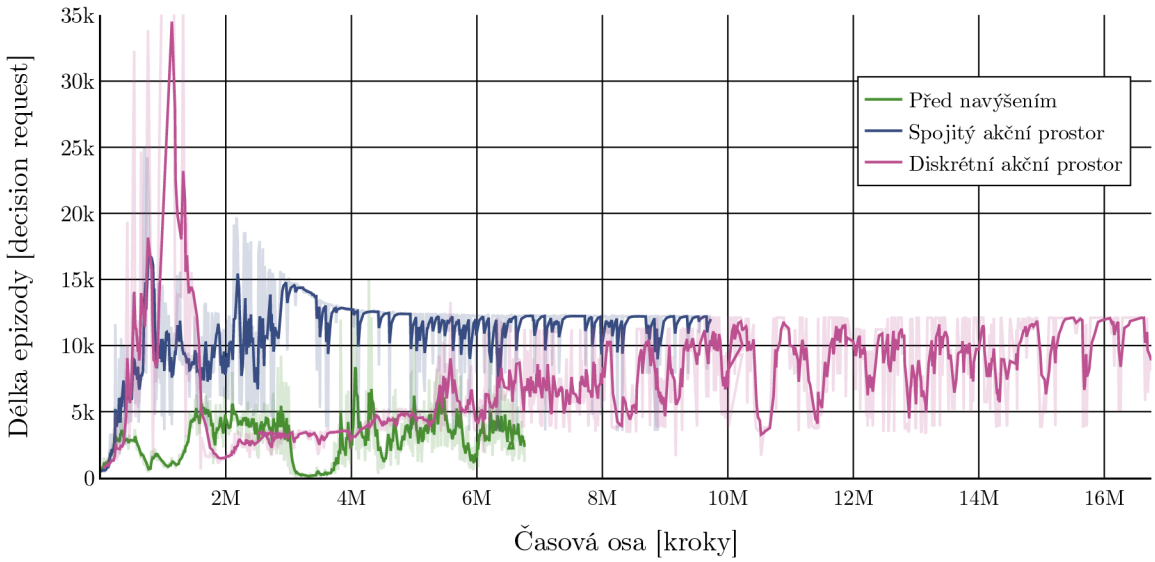
Nejdůležitější mechanismus, navádějící agenta k cíli, je nicméně negativní odměna za kontakt s mantinelem. V případě, že tak agent učiní, je trestán negativní odměnou a epizoda

končí. V neposlední řadě také získává, podobně jako v předchozí aplikaci, negativní odměnu s každým uplynulým krokem simulace. Toto ho má vést k co nejrychlejšímu jednání, neboli k co nejlepšímu času projetí okruhu.

Po prvotních testovacích bězích byla první velká změna učiněna z důvodu neschopnosti agenta ukázat zlepšení i po relativně velkém množství kroků. Řešením bylo navýšení počtu kontrolních bodů, ve výsledku zhruba o desetinásobek původních. To zmenšilo vzdálenost mezi jednotlivými body a pro agenta to znamenalo pravidelnější odměnu, což mu umožnilo jednodušší učení. Na obrázku 7.3 je zobrazena výše odměny získaná v epizodách pro tři trénovací běhy. Zeleně je běh původní před navýšením počtu kontrolních bodů. Graf nám napovídá, že ani po několika milionech kroků nedošlo k navyšování odměny a agent se tedy neblížil ke kýženému výsledku.

Fialový graf vyobrazuje běh agenta s diskrétním akčním prostorem. V porovnání s modrým grafem je vidět, že ačkoliv měl agent k dispozici pouze velice omezené možnosti, co se akcí týče, nebyl výsledek kvalitnější, ani ho nedosáhl v kratším čase. To mohlo být způsobeno buď lepšími náhodnými akcemi, které „modrý agent“ na počátku zvolil, nebo faktem, že v druhém případě vede větší preciznost v ovládání k lepším výsledkům i přes vyšší komplexitu. V obou případech byly po nasazení modely schopné trať projet, byť ne se stoprocentní úspěšností. Je také vidět, že odměna nepřesahuje hodnotu ≈ 40. Ta znázorňuje úspěšné projetí s dobrým časem a nárůst odměny už by byl na této hodnotě pomalý a relativně neznatelný.

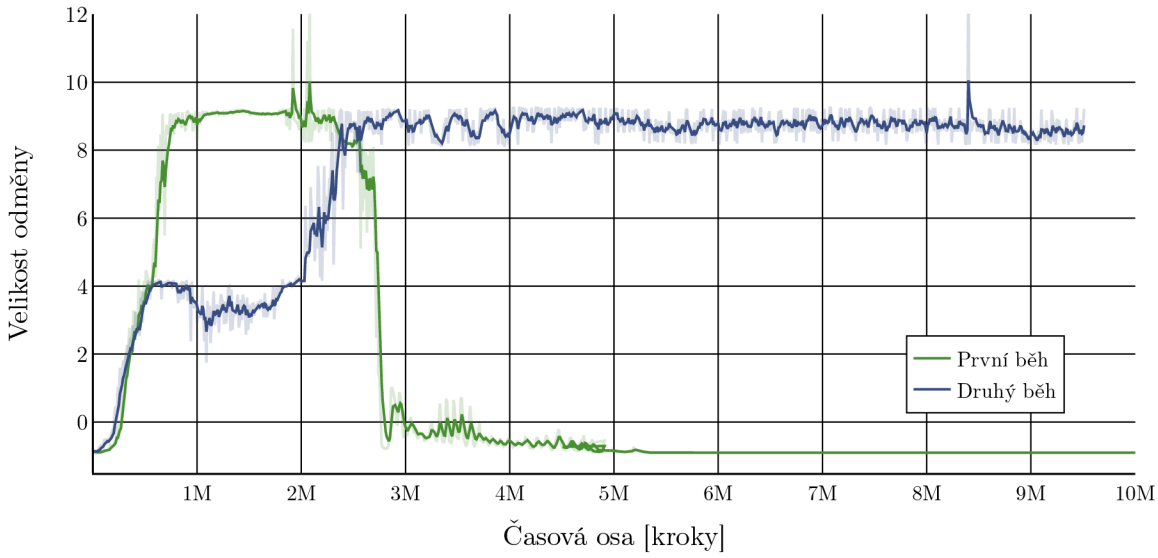
Obrázek 7.4 ilustruje délku odměny získané v epizodách. Zde je možné si všimnout, že jak pro diskrétní, tak pro spojitý běh křivky grafů od určitého momentu do značné míry kopírují předchozí graf. Což v praxi znamenalo, že získané odměny už převýšily časovou negativní odměnu i odměnu z kolizí. Agent pouze nebyl schopný úspěšně projet některou ze zatáček, ale už věděl, co má dělat.



Obrázek 7.4: Graf znázorňující dva trénovací běhy aplikace Racer, ukazující změnu délky epizod v čase.

Pro skutečné nasazení vytrénovaných modelů pomohlo například zrušit restartování epizody v případě kontaktu s mantinelem, protože sebemenší dotyk neznamená automatický

neúspěch. Toto by však mohlo také vést k nedefinovaným stavům a nepředvídatelnému chování.



Obrázek 7.5: Graf znázorňující 2 trénovací běhy aplikace Racer s použitím modulu `gail`, ilustrující velikost odměny v průběhu trénování.

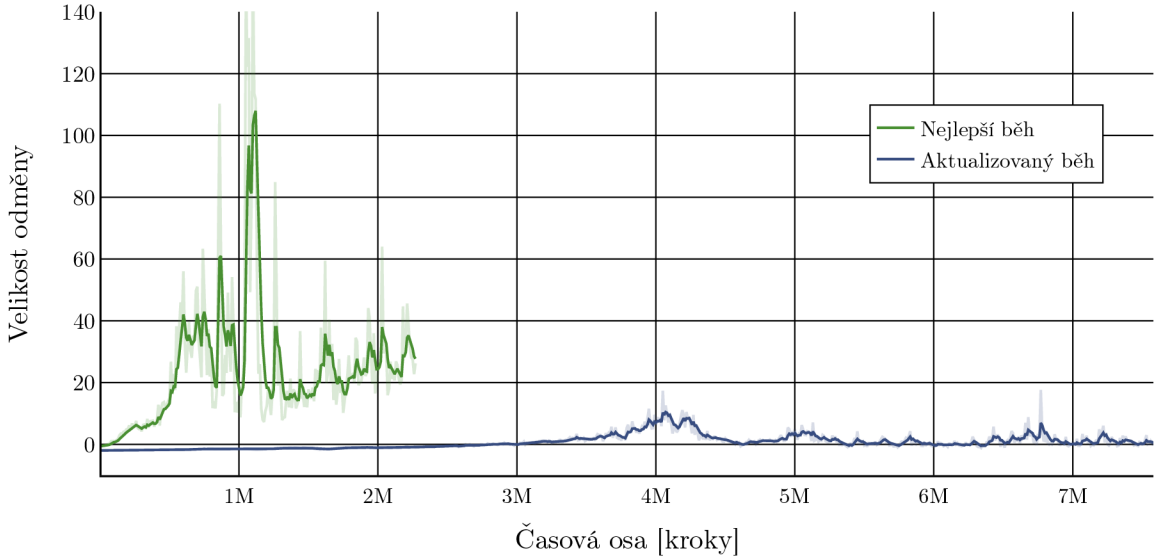
Jelikož má aplikace snadno definovatelný cíl (projet trať), byla aplikace vhodným kandidátem na vyzkoušení sil imitačního učení. Konkrétně byl použit modul GAIL a pro něj byla vytvořena demonstrace projetí jednoho kola okruhu. Zde nicméně výsledky neodpovídaly očekávání. Na obrázku 7.5 jsou vyobrazeny dva běhy s použitím tohoto modulu. Ačkoliv se byl agent schopen zlepšit velice rychle, nepovedlo se mu ani v jednom případě trať úspěšně projet. To mohlo být způsobeno buď nekvalitními demonstracemi, nebo přílišným důrazem na sílu tohoto signálu. V tomto kontextu síla určuje jak moc se agent řídí signálem `gail` oproti vnějším odměnám.

7.3 Demonstrační aplikace Nightmares

Cílem agenta v této aplikaci bylo přežít co nejdéle na bojišti a zlikvidovat u toho maximální možný počet nepřátel. Jelikož je toto jediná aplikace, která nebyla tvořena od základů, stala se také aplikací činící největší problémy. Agent získává odměnu dvojím způsobem – když nepřítele výstřelem trefí a když ho zabije. Jelikož nepřítel vydrží 5 zásahů, tak je pro agenta důležité, aby všechny své útoky nejdříve mířil na jednoho, než aby je rozděloval mezi více. V opačném případě by totiž byl obklopen a neměl už by kam utéci. Proto je za kompletní zneškodnění nepřítele vyšší odměna než pouze za zásah. Odměna za zásah byla přidána, neboť šance, že agent nepřítele trefí, je zpočátku podstatně vyšší, než že ho usmrtí – učí se tedy díky tomu rychleji. Negativní odměnu získává v okamžiku, kdy je sám zasažen, a jakmile agent umírá, epizoda končí.

První, relativně úspěšný, trénovací běh dosahoval po nasazení modelu skóre zhruba 1250 (125 zabitých nepřátel), což už by se dalo považovat za docela obstojný výsledek v porovnání s běžným člověkem. Ani zdaleka nicméně nepřekonává byť jen trochu lepšího hráče. K dosažení tohoto výsledku byl využit modul curiosity. Před jeho použitím agent ve většině běhů vyhodnotil, že je pro něj dobrou strategií schovat se v nejvzdálenějším

rohu arény – daleko od bodů, na kterých se periodicky nepřátelé objevovali. Ačkoliv byl takto schopen několik desítek nepřátel zneškodnit, jelikož již neměl dále kam ustupovat, nebyla to strategie optimální. S použitím modulu se zvýšil pohyb po mapě a dosáhl lepších výsledků. Zároveň byl v tomto běhu použit jednodušší způsob rotace kolem osy. Agent pouze určoval směr otočení, podobně jako v aplikaci *Racer*. To nicméně znamenalo, že se nemohl okamžitě otočit kýženým směrem. Proto bylo implementováno otáčení popsané v implementační části 6.3.1, které umožňuje téměř instantní otočení libovolným směrem. To mělo vést k rychlejším reakcím s velkým množstvím nepřátel. Zvýšená komplexita nicméně vedla v průběhu testování k horším výsledkům.



Obrázek 7.6: Graf znázorňující dva trénovací běhy aplikace Nightmares, vyobrazující velikost odměny za čas běhu.

Na obrázku 7.6 je možné vidět porovnání nejlepšího běhu (s curiosity) a běhu, který využívá imitační učení a aktualizovaný způsob ovládání. Z něj je patrné, že jednodušší ovládání vedlo k daleko lepším výsledkům, byť s velkou variancí. Aplikace byla problémová k vytrénování kvalitní neuronové sítě a i trénování samotné přineslo značná úskalí. Simulace s vyšším časovým měřítkem neodpovídala, kvůli původní implementaci, předpokladům (agent neměl možnost střílet tak často, jak by měl). Proto muselo trénování běžet s měřítkem menším a bylo kvůli tomu pomalejší než u ostatních aplikací. Pro zajímavost byl také vytvořen běh, ve kterém soupeři i agent umírají po jednom zásahu. Zde byl agent schopen získat velice vysoké skóre, ale jelikož k tomu nebyla aplikace původně uzpůsobena, bylo to pro něj v tomto případě podstatně jednodušší.

Kapitola 8

Závěr

Cílem této práce bylo vytvořit v herním enginu Unity, za pomoci sady ML-Agents, prostředí pro trénování agentů pomocí strojového učení. Zároveň si kladla práce za cíl zhodnotit otázku využitelnosti strojového učení, potažmo sady ML-Agents pro vývoj umělé inteligence ve skutečných projektech. K realizaci tohoto zadání byly navrhnuty a implementovány dvě demonstrační aplikace, s odlišným prostředím, ve kterých byly dále prováděny experimenty ve formě trénovacích běhů. U obou aplikací bylo možné vidět, že se agenti postupem času zlepšovali a výstupem byl vytrénovaný model neuronové sítě, který byl více či méně schopen plnit zadané úkoly.

V rámci třetí aplikace byl v práci zpracován již implementovaný projekt, do kterého byla přidána sada ML-Agents. Tato demonstrační aplikace měla sloužit jako příklad prostředí, které nebylo navržené s ohledem na aplikaci strojového učení. Použití sady se zde ukázalo být značně náročnější než v prvních dvou aplikacích. Jednak z důvodu vyšší komplexity prostředí, jednak kvůli způsobu implementace. Aby síť dosahovala skutečně výborných výsledků, vyžadovala by řádově delší trénovací běhy. I zde se nicméně podařilo dosáhnout alespoň částečně uspokojivého výsledku.

Nakonec chtěla práce odpovědět na otázku, zda by tento přístup mohl nahradit existující techniky návrhu umělé inteligence ve hrách. V případě úmyslu použití sady je nutné dbát již na návrh aplikace, kde musí vývojář počítat s jejím použitím a vytvořit ji tak, aby trénování bylo možné. Ačkoliv nejspíše v nejbližší době strojové učení zavedené techniky nenahradí, například z důvodu nepředvídatelnosti vytrénovaného modelu, kdy vývojář nemusí nutně vědět, jak přesně bude agent v daný okamžik jednat, udržuji si v tomto ohledu do budoucna optimismus. Využitelnost vidím především v simulování některých méně důležitých prvků ve videohrách. Těmi mohou být agenti ve formě fauny, či neinteraktivní nehráčské postavy, kde není kompletní znalost jejich chování nutností, a naopak jistá nepředvídatelnost v jednání může dodat na výsledném zážitku.

Literatura

[1] Ahlawat, A. *Game Engine and History of Game Development* [online]. [cit. 2021-01-17]. Dostupné z: https://www.studytonight.com/3d-game-engineering-with-unity/game-engine.

[2] Anyoha, R. The History of Artificial Intelligence. [online]. Srpen 2017, [cit. 2020-12-01]. Dostupné z: http://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/.

[3] Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P. et al. Dota 2 with Large Scale Deep Reinforcement Learning. *CoRR*. 2019, abs/1912.06680, [cit. 2020-12-04]. Dostupné z: http://arxiv.org/abs/1912.06680.

[4] Burda, Y., Edwards, H., Storkey, A. J. a Klimov, O. Exploration by Random Network Distillation. *CoRR*. 2018, abs/1810.12894, [cit. 2021-04-22]. Dostupné z: http://arxiv.org/abs/1810.12894.

[5] Coen, M. H. *SodaBot: A Software Agent Environment and Construction System*. Květen 1994 [cit. 2020-12-01].

[6] Colledanchise, M. a Ögren, P. Behavior Trees in Robotics and AI: An Introduction. *CoRR*. 2017, abs/1709.00084, [cit. 2020-12-16]. Dostupné z: http://arxiv.org/abs/1709.00084.

[7] Copeland, B. Artificial intelligence. [online]. Encyclopædia Britannica. Srpen 2020, [cit. 2020-12-01]. Dostupné z: https://www.britannica.com/technology/artificial-intelligence.

[8] François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G. a Pineau, J. An Introduction to Deep Reinforcement Learning. *CoRR*. 2018, abs/1811.12560, [cit. 2020-12-05]. Dostupné z: http://arxiv.org/abs/1811.12560.

[9] Galbraith, B. *Imitation Learning: Reinforcement Learning For The Real World - Dr. Byron Galbraith* [online]. [cit. 2021-01-05]. Dostupné z: https://www.youtube.com/watch?v=K_c2ko4AKgg&ab_channel=OpenDataScience.

[10] gRPC. *Introduction to gRPC* [online]. [cit. 2021-03-15]. Dostupné z: https://grpc.io/docs/what-is-grpc/introduction/.

[11] Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S. et al. Soft Actor-Critic Algorithms and Applications. *CoRR*. 2018, abs/1812.05905, [cit. 2021-01-20]. Dostupné z: http://arxiv.org/abs/1812.05905.

[12] Hofmann, M., Neukart, F. a Bäck, T. Artificial Intelligence and Data Science in the Automotive Industry. *CoRR*. 2017, abs/1709.01989, [cit. 2020-12-04]. Dostupné z: http://arxiv.org/abs/1709.01989.

[13] Isla, D. GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI. [online]. Gamasutra. 2005, [cit. 2020-12-15]. Dostupné z: https://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php.

[14] Juliani, A. Solving sparse-reward tasks with Curiosity. [online]. Unity Blog. 2018, [cit. 2021-04-22]. Dostupné z: https://blogs.unity3d.com/2018/06/26/solving-sparse-reward-tasks-with-curiosity/.

[15] Lőrincz, Z. A brief overview of Imitation Learning. [online]. Medium.com. 2019, [cit. 2021-01-06]. Dostupné z: https://medium.com/@SmartLabAI/a-brief-overview-of-imitation-learning-8a8a75c44a9.

[16] Marr, B. How Is AI Used In Healthcare - 5 Powerful Real-World Examples That Show The Latest Advances. [online]. Forbes. Červenec 2018, [cit. 2020-12-04]. Dostupné z: https://www.forbes.com/sites/bernardmarr/2018/07/27/how-is-ai-used-in-healthcare-5-powerful-real-world-examples-that-show-the-latest-advances/?sh=167940095dfb.

[17] Meduna, A. *Automata and Languages*. Leden 2000 [cit. 2020-12-08]. ISBN 978-1-85233-074-3.

[18] Mitchell, T. M. *Machine Learning*. 1. vyd. McGraw-Hill, Inc., 1997 [cit. 2020-12-01]. ISBN 0070428077.

[19] Mittal, R. What is an ELO Rating? [online]. Medium.com. 2020, [cit. 2021-04-30]. Dostupné z: https://medium.com/purple-theory/what-is-elo-rating-c4eb7a9061e0.

[20] Nachum, O., Norouzi, M., Xu, K. a Schuurmans, D. Bridging the Gap Between Value and Policy Based Reinforcement Learning. *CoRR*. 2017, abs/1702.08892, [cit. 2020-12-05]. Dostupné z: http://arxiv.org/abs/1702.08892.

[21] Nystrom, R. *Game Programming Patterns*. Genever Benning, 2014 [cit. 2020-12-08]. ISBN 9780990582908.

[22] OpenAI. *OpenAI Gym* [online]. [cit. 2021-01-19]. Dostupné z: https://github.com/openai/gym.

[23] Owen, L. Bird's-Eye View of Reinforcement Learning Algorithms Taxonomy. [online]. towards data science. 2020, [cit. 2020-12-06]. Dostupné z: https://towardsdatascience.com/birds-eye-view-of-reinforcement-learning-algorithms-landscape-2aba7840211c.

[24] Pathak, D., Agrawal, P., Efros, A. A. a Darrell, T. Curiosity-driven Exploration by Self-supervised Prediction. *CoRR*. 2017, abs/1705.05363, [cit. 2021-04-22]. Dostupné z: http://arxiv.org/abs/1705.05363.

[25] Press, G. 12 AI Milestones: 4. MYCIN, An Expert System For Infectious Disease Therapy. [online]. Forbes. Duben 2020, [cit. 2020-12-01]. Dostupné z: https://www.forbes.com/sites/gilpress/2020/04/27/12-ai-milestones-4-mycin-an-expert-system-for-infectious-disease-therapy/?sh=1e1551b176e5.

[26] Russell, S. J. a Norvig, P. *Artificial Intelligence: A Modern Approach*. 1. vyd. Prentice-Hall, Inc., 1995 [cit. 2020-12-01]. ISBN 0-13-103805-2.

[27] S. Sutton, R. a G. Barto, A. *Reinforcement Learning: An Introduction*. The MIT Press, 2015 [cit. 2020-12-05]. ISBN 9780262039246.

[28] Schulman, J., Wolski, F., Dhariwal, P., Radford, A. a Klimov, O. Proximal Policy Optimization Algorithms. *CoRR*. 2017, abs/1707.06347, [cit. 2021-01-20]. Dostupné z: http://arxiv.org/abs/1707.06347.

[29] Sekhavat, Y. Behavior Trees for Computer Games. *International Journal on Artificial Intelligence Tools*. Leden 2017, sv. 26, [cit. 2020-12-15]. DOI: 10.1142/S0218213017300010.

[30] Talaviya, T., Shah, D., Patel, N., Yagnik, H. a Shah, M. Implementation of artificial intelligence in agriculture for optimisation of irrigation and application of pesticides and herbicides. *Artificial Intelligence in Agriculture*. 2020, sv. 4, s. 58 – 73, [cit. 2020-12-04]. DOI: https://doi.org/10.1016/j.aiia.2020.04.002. ISSN 2589-7217. Dostupné z: http://www.sciencedirect.com/science/article/pii/S258972172030012X.

[31] Thompson, T. Cyber Demons | The Ai of DOOM (2016). [online]. Gamasutra. 2018, [cit. 2020-12-08]. Dostupné z: https://www.gamasutra.com/blogs/TommyThompson/20180806/323715/Cyber_Demons__The_AI_of_DOOM_2016.php.

[32] Torabi, F., Warnell, G. a Stone, P. *Recent Advances in Imitation Learning from Observation*. 2019 [cit. 2021-01-06]. Dostupné z: http://arxiv.org/abs/1905.13566.

[33] Turing, A. M. Computing Machinery and Intelligence. *Mind*. Oxford University Press on behalf of the Mind Association. 1950, sv. 59, č. 236, s. 433–460, [cit. 2020-12-01]. ISSN 00264423.

[34] Unity Technologies. *Solutions|Unity* [online]. [cit. 2021-01-16]. Dostupné z: https://unity.com/solutions.

[35] Unity Technologies. *Survival Shooter* [online]. [cit. 2021-03-28]. Dostupné z: https://learn.unity.com/project/survival-shooter-tutorial.

[36] Unity Technologies. *Unity Barracuda* [online]. [cit. 2021-02-04]. Dostupné z: https://github.com/Unity-Technologies/barracuda-release.

[37] Unity Technologies. *Unity ML-Agents Toolkit* [online]. [cit. 2021-01-14]. Dostupné z: https://github.com/Unity-Technologies/ml-agents.

[38] Unity Technologies. *Unity User Manual (2019.4 LTS)* [online]. [cit. 2021-04-02]. Dostupné z: https://docs.unity3d.com/2019.4/Documentation/Manual/UnityManual.html.

[39] Unity Technologies. *Welcome to Unity* [online]. [cit. 2021-01-16]. Dostupné z: https://unity.com/our-company.

[40] Wilson, A. A Brief Introduction to Supervised Learning. [online]. towards data science. 2019, [cit. 2021-01-06]. Dostupné z: https://towardsdatascience.com/a-brief-introduction-to-supervised-learning-54a3e3932590.

[41] Wooldridge, M. a Jennings, N. R. Intelligent agents: theory and practice. *The Knowledge Engineering Review*. Cambridge University Press. 1995, sv. 10, č. 2, s. 115–152, [cit. 2020-12-01]. DOI: 10.1017/S0269888900008122.

[42] Zwass, V. Expert system. [online]. Encyclopædia Britannica. Únor 2016, [cit. 2020-12-01]. Dostupné z: https://www.britannica.com/technology/expert-system.

Příloha A

Seznam příloh

- **latex-source/** – adresář obsahující zdrojový kód LaTeX
- **projects/** – adresář s demonstračními aplikacemi
- **promo.mp4** – kratší video s ukázkami z aplikací (bez komentáře)
- **README.md** – návod na spuštění aplikací
- **thesis-text.pdf** – toto PDF
- **video.mp4** – doprovodné video s komentářem a ukázkou aplikací