Vytvoření umělé inteligence pro mobilní hru pomocí Unity ML-Agents Toolkit

Jiří Burda

Bakalářská práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Jiří Burda**
Osobní číslo:	**A21001**
Studijní program:	**B0613A140020 Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Vytvoření umělé inteligence pro mobilní hru pomocí Unity ML-Agents Toolkit**
Téma práce anglicky:	**Artificial Intelligence for a Mobile Game Created Using the Unity ML-Agents Toolkit**

Zásady pro vypracování

1. Vypracujte literární rešerši na zadané téma.
2. Zaměřte se zejména na techniky strojového učení a knihovnu Unity ML-Agents Toolkit.
3. Navrhněte hru pro demonstraci použití dané knihovny.
4. Vytvořte hru dle návrhu.
5. Implementaci vhodně popište.
6. Otestujte a vyhodnoťte funkčnost umělé inteligence.

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. ENGELBRECHT, Dylan. *Introduction to Unity ML-Agents: Understand the Interplay of Neural Networks and Simulation Space Using the Unity ML-Agents Package*. Apress, 2023 ISBN 978-1484289976.
2. LANHAM, Micheal. *Learn Unity ML-Agents – Fundamentals of Unity Machine Learning: Incorporate new powerful ML algorithms such as Deep Reinforcement Learning for games*. Packt Publishing, 2018. ISBN 978-1789138139.
3. SHALEV-SHWARTZ, Shai; BEN-DAVID, Shai. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014. ISBN 978-1-107-05713-5. Dostupné z: https://www.cs.huji.ac.il/ shais/UnderstandingMachineLearning/understanding-machine-learning-theory-algorithms.pdf.
4. Unity Team. *UNITY MACHINE LEARNING AGENTS. v: unity.com* [online]. Dostupné z: https://unity.com/products/machine-learning-agents.
5. HALPERN, Jared. *Developing 2D Games with Unity: Independent Game Programming with C#*. Apress, 2018. ISBN: 978--1484237717.

Vedoucí bakalářské práce: **Ing. Tomáš Vogeltanz, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání bakalářské práce: **5. listopadu 2023**
Termín odevzdání bakalářské práce: **13. května 2024**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 5. ledna 2024

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl jsem seznámen s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval, v případě publikace výsledků budu uveden jako spoluautor;
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 13. 5. 2024 …….

podpis studenta

ABSTRAKT

Cílem bakalářské práce je vytvořit umělou inteligenci pro mobilní hru pomocí knihovny Unity ML-Agents Toolkit. Práce nejprve obsahuje teoretický úvod do umělé inteligence a strojového učení s důrazem na posilované učení, které je jednou z nejpoužívanějších metod v oblasti vývoje herního AI. V praktické části jsou aplikovány znalosti z teoretické části, je popsána implementace trénování AI pomocí knihovny ML-Agents a stručně nejpodstatnější kroky pro vytvoření hry. Výsledky práce ukazují, že natrénovaní agenti (herní AI) poskytují dynamické a adaptivní chování, které může značně zvýšit herní zážitek. Práce přináší nové možnosti pro vývojáře her a otevírá diskuzi o budoucích směrech využití strojového učení v herním průmyslu.

Klíčová slova: umělá inteligence, strojové učení, posilované učení, vývoj her, mobilní hry, trénování AI, agent

ABSTRACT

The aim of the bachelor's thesis is to create artificial intelligence for a mobile game using Unity ML-Agents Toolkit. The thesis firstly contents a theoretical introduction to artificial intelligence and machine learning, with an emphasis on reinforcement learning, which is one of the most commonly used methods in the field of game AI development. In the practical part, the knowledge from theoretical part is applied. Then in the thesis is described implementation of AI training using the ML-Agents library and briefly outlining the most crucial steps for creating a game. The results of the thesis show that the trained agents (game AI) provide dynamic and adaptive behavior that can significantly enhance the gaming experience. The work offers new possibilities for game developers and opens a discussion on future directions for the use of machine learning in the gaming industry.

Keywords: artificial intelligence, machine learning, reinforcement learning, game development, mobile games, training of AI, agent

Tímto bych rád velmi poděkoval hlavně svému vedoucímu práce panu Ing. Tomáši Vogel-tanzovi, Ph.D. za odborné vedení, trpělivost a skvělý přístup, za čas věnovaný mé práci, poskytnuté materiály a ochotu pomoci při vypracování bakalářské práce. Také děkuji za možnost, že jsem si mohl vybrat právě toto téma.

Dále bych rád poděkoval paní prof. Ing, Zuzaně Komínkové Oplatkové, Ph.D. za poskytnutí materiálů a správné nasměrování v oblasti strojového učení.

Chtěl bych také poděkovat všem akademikům, kteří se mnou v průběhu vývoje konzultovali některé části a pomohli mi přijít na lepší řešení.

Prohlašuji, že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

Umělá inteligence (AI) je dnes velmi probírané téma, protože s příchodem Chat GPT si všichni uvědomili, hlavně široká veřejnost, jak obrovský potenciál může AI mít. Oblast AI je tak obsáhlá, že se dělí na různé kategorie. Jednou z nich je strojové učení (angl. Machine Learning, dále jen ML). Tato práce se věnuje právě strojovému učení ve hrách, konkrétněji v herním enginu Unity za použití knihovny Unity ML-Agents Toolkit.

Herní průmysl, jako nejziskovější odvětví v zábavném průmyslu s dominancí mobilních her, je vhodné prostředí pro inovace v AI. Tato práce představuje návod na vytvoření mobilní hry se sofistikovaným oponentem. Popisuje, jak vytvořit prostředí pro trénování, metody trénování a různost hyperparametrů, které toto trénování ovlivňují. Předmětem práce je především software, tedy mobilní aplikace, kde lze otestovat funkčnost vytvořeného AI. V teoretické části je kromě analýzy technologií používaných k vytvoření herní AI také rozebrána právě knihovna Unity ML Agents Toolkit.

Cílem práce je pochopit terminologii ML, jak se ML dále rozděluje, kam zapadá posilovací učení (angl. Reinforcement Learning), a jak se dá využít pro vývoj her, specificky za pomocí herního enginu Unity.

I. TEORETICKÁ ČÁST

1 UMĚLÁ INTELIGENCE

V dnešní době je svět značně ovlivněn umělou inteligencí (AI) a strojovým učením (ML). Tyto obory otevřely cestu úplně novým možnostem. Dnešní AI/ML technologie zasahují do všech odvětví, včetně zdravotnictví, dopravy, bezpečnosti, zábavy atd. Tato kapitola slouží jako úvod do povědomí AI.

1.1 Dnešní AI a ML

Obrovský vliv mají AI a ML v doporučování obsahu. K analýze interakcí a preferencí uživatelů platformy jako YouTube Music, Spotify a TikTok je použito ML. To nejenže zlepšuje zákaznické zkušenosti, ale také podporuje loajalitu k platformám a růst příjmů prostřednictvím cílených reklam. [1]

Další průkopnickou aplikací ML jsou autonomní vozidla, ve kterých některé firmy (např. Tesla) využívají AI k vytvoření samořídících automobilů. Tato vozidla spoléhají na ML k zpracování senzorických dat, což jim umožňuje dělat rozhodnutí v reálném čase, která bezpečně navigují složitými prostředími. Ačkoliv pokroky jsou opravdu obrovské, autonomní vozidla stále nemáme. To se ale v budoucnosti změní. [1]

V oblasti zdravotnictví bylo strojové učení nasazeno jako prostředek pro dosažení pokroku v diagnostické přesnosti, personalizované medicíně a péči o pacienty. S extrémní přesností může AI diagnostikovat nemoci, např. rakovinu, na základě rentgenového snímku. Modely strojového učení jsou také používány pro predikci výsledků pacientů, optimalizaci plánů léčby a asistenci během operace, což znamená, že mohou zachránit životy a zlepšit poskytování zdravotní péče. [1]

Bezpečnost a dohled také těží z technologií AI a ML. Algoritmy ML mají schopnost identifikovat možné bezpečnostní hrozby, např. teroristy na základě analýzy videa a dalších dat. Tím se zvyšuje bezpečnost v celém světě. [1]

Americká neurotechnologická společnost Neuralink Corporation, jejímž spoluzakladatelem je Elon Musk, má fascinující výsledky ve vývoji AI a ML. Společnost se zaměřuje na vytváření moderních rozhraní mezi mozkem a počítači s možnou budoucností komunikace mezi lidmi a stroji. Léky na neurologické onemocnění a obnova senzorických funkcí jsou jen jedny z mnoha aspektů, které by Neuralink Corporation mohl nabídnout. Představa, že z AI může být lidský mozek, naznačuje nové horizonty pro interakce s technologiemi a porozuměním lidskému vědomí. [1]

AI má dlouhou historii ve hrách. Využívá se u starých her, jako např. šachy, ale také u moderních videoher, jako např. úspěšná Dota 2. Vědci dříve mysleli, že vrchol umělé inteligence nastane tehdy, až AI porazí velmistra v šachu. Věřilo se, že AI by musela být vědomá, aby mohla dostatečně dobře přemýšlet a hrát šachy. Tento názor se ukázal být daleko od pravdy. Počítač Deep Blue od IBM porazil v šachu Garryho Kasparova v roce 1997 [2], což byl velmi důležitý milník pro další vývoj umělé inteligence. Díky impozantnímu financování a vzestupu masivních vývojářských studií zaměřených na ML a AI se hodně pokročilo. V roce 2017 představil OpenAI svou OpenAI Five. Tento AI agent dokáže rozhodovat na základě pixelových dat a porazil nejlepší tým na světě v roce 2018 ve hře Dota 2. [1]

Ve hrách není vždy použito jen strojové učení. Většina AI ve hrách je založena na pevně zakódované logice, která je často úmyslně navržená tak, aby dělala chyby. AI, která je lepší než nejlepší lidští hráči, není dostatečně zábavná pro průměrného hráče. Hybridní přístupy k pevně zakódované AI a strojovému učení pro AI ve hrách jsou trendem ve vývoji her. [1]

Dnešní technologie AI a ML přinášejí spoustu inovací, nabízejí nepřekonatelné příležitosti k řešení složitých problémů. Rozsah použití je od personalizovaných doporučení obsahu a autonomních vozidel po pokroky ve zdravotnictví a bezpečnosti. Dopad AI a ML je všudypřítomný, přetváří průmysl a každodenní život každého z nás.

1.2 AI ve hrách

Použití umělé inteligence ve hrách (IBM Deep Blue, úspěch OpenAI v Dota 2) ukazuje, jak se zvyšují schopnosti AI v oblasti složitého rozhodování a strategického myšlení. Tento pokrok ilustruje postupně se rozvíjející integrovaný zdroj informací o možnostech umělé inteligence v konkrétních oblastech a představuje možnost transformace průmyslových odvětví pro zlepšení produktivity a řešení kritických problémů. [1]

Dějiny umělé inteligence jsou svědectvím lidského génia a snahy pochopit a napodobit mechanismy lidské inteligence. AI prošla neuvěřitelným vývojem. Významné jsou nejen teoretické práce (např. Alana Turinga), ale i praktické aplikace vyvinuté různými společnostmi (Google, Nvidia a OpenAI). Nyní stojíme na prahu budoucnosti. Práce jednotlivců a organizací v dané oblasti má za následek další objevy, možnosti uplatnění a etické rámce, které formují budoucí trajektorii AI. Cesta AI, od konceptuálního rámce po transformační technologie, zvyšuje důležitost mezidisciplinární spolupráce a etického přístupu v této oblasti. [1]

Vývojářům her hodně pomáhá ML v různých oblastech.

1.2.1 Generování map a levelů a PCG

Vývojáři her používají strojové učení k automatickému generování prostředí od dungeonů po realistický terén. Při správném použití ML lze vytvořit hry s nekonečným opětovným hraním, ale je to jedno z nejnáročnějších ML, jaké je možno vyvinout. Tato technika, kdy se generují objekty za běhu hry se nazývá PCG (Procedural Content Generation) neboli Procedurální generování obsahu. Příkladem takové hry je například „No Man's Sky", kde hráč může prozkoumávat celý automaticky generovaný vesmír a může objevit planety, na kterých ještě nebyl žádný hráč. [3] [4]

1.2.2 Generovaní textur a stínů

To je další oblast, kde se ML hodně používá. Zde bylo dosaženo značného pokroku díky GAN (Generative Adversrial Networks). GAN je framework, který slouží právě k vytvoření automaticky generovaných obrázků. Pomocí nich však vznikají podvrhy, které jsou označovány jako Deep Fake. [3]

1.2.3 Generování 3D modelů

Existuje několik projektů, které se uskutečňují v této oblasti a mohly by značně ulehčit konstruování 3D objektů pomocí pokročilého skenování nebo automatického generování. Stačí pouze textově popsat jednoduchý model a nechat ML, aby za nás ve hře nebo AR/VR/MR aplikaci 3D model postavilo. [3]

1.2.4 Generování zvuků

Možnost generovat audio se používá už delší dobu a není to pouze ve hrách. Je možno složit celý soundtrack pro hru jen pomocí strojového učení.

1.2.5 Umělí hráči

Zde je mnoho možností použití. Jednou z nich je, že samotní hráči používají ML, aby hrálo za ně (například „boti" ve World of Warcraft, kteří chodí a sbírají suroviny). Jinou možností jsou umělí hráči jako pokročilí testovací agenti nebo zapojení umělých hráčů během nízké aktivity. Umělé hráče lze použít pro testování různých levelů, herních vlastností, atd. [3]

1.2.6 NPC a herní AI

NPC je non-playable character, česky nehráčská postava. Zatím se hodně používají pro chování NPC a herního AI stromy chování (Behavorial Trees), avšak vývoj směřuje tím

směrem, že budou NPC a herní AI vytvářeny pomocí ML. Tím by se NPC mohly chovat více nepředvídatelně a současně zábavněji pro hráče. [3]

1.3 Budoucnost AI: Perspektivy a etické požadavky

Velká pozornost se věnuje možnostem změny AI v justičním systému. Lidstvo si může představit situaci, kde by právní služby byly poskytovány AI agenty a nestranné soudní procesy by byly založené na rozhodnutích bez zaujatosti. Poskytnutí levné právní ochrany a objektivního soudního řízení bez jakýchkoli forem předsudků by pomohlo vyřešit současné problémy dostupnosti právních služeb. Vedlo by to k spravedlivějšímu systému. [1]

Další oblastí s obrovským AI potenciálem je zdravotnictví. Integrace AI do systémů zdravotnictví slibuje snižování nákladů a zpřístupnění život zachraňujících operací chudým komunitám. Použití je možné od diagnostiky až po chirurgické zákroky prováděné s lidskou přesností. Například takové inovace, jako je AI Googlu AlphaFold se skládáním proteinů, ukazují, jak může AI urychlit lékařský výzkum a slibují léčbu nemocí, které byly dříve nemožné. [1]

V oblasti veřejné politiky, zejména daní, Dylan Engelbrecht [1] zdůrazňuje probíhající výzkum využití AI pro vývoj spravedlivé daňové politiky. Využitím strojového učení pro optimalizaci politiky existuje potenciál pro vytvoření daňového systému, který vyvažuje rovnost a produktivitu, dynamicky se přizpůsobuje ekonomické situaci. Tato aplikace AI by mohla vést k efektivnější správě a spravedlivějšímu rozdělení zdrojů. [1]

Hovoří se také o prodloužení života a o rozhraní mozek-počítač (BCI – Brain Computer Interface). V této oblasti by AI mohla předefinovat lidské schopnosti a dlouhověkost. Vývoj BCI, kterým se zabývá např. firma Neuralink, naznačuje budoucnost, ve které by lidé mohli rozšířit své kognitivní schopnosti, potenciálně chránící lidskou důležitost v pokročilé éře AI. Tato sekce také zvažuje etické důsledky takových technologií, včetně potenciálu AI interpretovat a rozšiřovat lidské myšlenky. [1]

V oblasti zábavy schopnost AI generovat umění a hudbu ukazuje, jak je AI nejen nástrojem pro efektivitu, ale také partnerem v kreativitě. Tato spolupráce mezi AI a lidskými umělci by mohla obohatit kreativní proces, poskytovat nové cesty pro umělecký projev. [1]

V diskusi je klíčová potřeba řešit a zmírnit předsudky v AI (bias), zajistit, aby systémy AI odrážely diverzitu globální populace. Je třeba, aby byl k dispozici dostatek rozmanitých datových sad jako například řešení předsudků ke kolektivnímu úsilí o budování AI, která

respektuje kulturní a individuální rozdíly. Je zdůrazněn význam etických úvah ve vývoji AI, usiluje se o předcházení potenciálním negativním výsledkům. Místo toho se podporuje budoucnost, kde AI zlepšuje společenské blaho. Budoucnost AI a ML tedy není jen o technologických úspěších, ale také o zajištění, že tyto pokroky přispívají k spravedlivější, bezpečnější společnosti. [1]

2 STROJOVÉ UČENÍ

Strojové učení (Machine Learning, ML) je to, že se stroje, mobily počítače učí z dostupných vstupů/dat, aby přizpůsoboval své chování podle daného prostředí bez explicitních intrukcí. Takže vstupem jsou trénovácí data neboli zkušenost a výstupem je dovednost. [1] [5]

2.1 Rozdělení strojového učení

Strojové učení, větev umělé inteligence se soustředí na učení vzorů z dat. Lze ho rozdělit do čtyř základních kategorií podle použitých metod a přístupů, a to takto: učení s učitelem (Supervised learning), učení bez učitele (Unsupervised learning), poloučené učení (Semi-supervised learning) a posilované učení (Reinforcment learning). Každá třída algoritmů se učí z rozdílných typů dat. [5] V následujícím textu jsou vysvětleny jednotlivé kategorie.

2.1.1 Učení s učitelem (Supervised Learning)

Učení s učitelem, jak termín naznačuje, je založen na dohledu. To znamená, že v tomto přístupu se trénují stroje za použití „označeného“ datového souboru (labeled dataset) a stroj předpovídá výstup na základě tréninku. Některé vstupy jsou už namapovány k výstupu, jak je naznačeno označenými daty. Konkrétněji lze popsat, že se nejdříve trénuje stroj se vstupem a výstupem, a potom je požadováno, aby byl předpovězen výstup na základě testovacího datového souboru. [6]

Algoritmy používané v tomto učení jsou: rozhodovací stromy (Decision Trees), náhodné lesy (Random Forrest), lineární regrese, logistická regrese, lasso regrese, K-Nearest Neighbours, SVM (Support Vector Machines) a Naive Bayes. Reálné využití: detekce podvodu, filtrování spamu, rozpoznání řeči. [6]

2.1.2 Učení bez učitele (Unsupervised Learning)

Učení bez učitele, jak naznačuje název, se liší od učení s učitelem v tom, že nepotřebuje dohled. V tomto učení, je počítač natrénován s „neoznačeným datovým souborem“ (unlabeled dataset) a předpovídá výsledek bez jakéhokoliv lidského zásahu. Hlavním cílem tohoto učení je roztřídit a kategorizovat nesetříděné informace do skupin nebo kategorií na základě podobností, vzorů a rozdílů. [6]

Používané algoritmy tohoto učení jsou: neuronové sítě (Neural Network / Deep Learning), Mean-Shift, algoritmy založené na centroidech a na hustotě, FP–Growth, pravděpodobnostní algoritmy (Probabilistic), metody pro redukci dimenzionality (Dimensionality Reduction) a

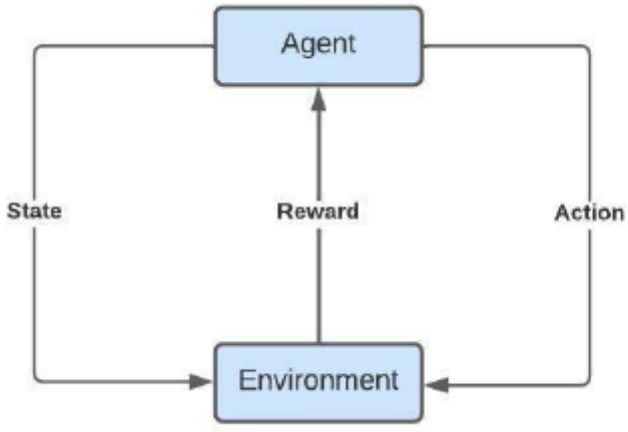
algoritmus učení asociačních pravidel (Association rule learning algorith). V praxi se toto učení používá pro doporučovací systémy, detekci anomálií, segmentace obrazu NLP (Natural Language Processing) a analýzu sociálních sítí. [6]

2.1.3 Poloučené učení (Semi-supervised Learning)

Poloučené učené je algoritmus strojového učení, který spadá přesně mezi učení s učitelem a bez učitele. Používá kombinaci označeného a neoznačeného datového souboru v průběhu trénovací fáze a představuje střední cestu mezi oběma zmíněnýma metodama. Příklad poloučeného učení je například analýza mluveného projevu. Označování zvukových nahrávek je časově náročné a drahé, potřebuje značné množství lidského úsilí. Pro tyto případy je zde toto učení. [6]

2.1.4 Zpětnovazební učení (Reinforcement Learning)

RL (také zpětnovazební nebo posilovací učení) je takový typ učení, který je založen na interakcích s prostředím a je zaměřen na určitý cíl. Agent, RL algoritmus se učí z prostředí jako opakující se metoda. Agent se učí z jeho zkušeností z prostředí, dokud neprozkoumá celý rozsah možných stavů, aby došel k cílovému stavu. Jednoduchá zpětná vazba v podobě odměny je pro agenta potřebná, aby se naučil své chování. Tento postup je znám jako „reinforcement signal“. Praktické užití tohoto učení jsou systémy těžby textu, robotika a videohry. [6]



Obrázek 1 Reinforcement Signal

V této práci bude hlavně věnována pozornost právě zpětnovazebnému učení, protože se používá ve hrách.

2.2 Posilované učení – Reinforcement Learning

V herním průmyslu už bylo použito mnoho technik strojového učení, kde učení s učitelem hrálo hlavní úlohu. I když tyto metody můžou vypadat inteligentně, jsou stále limitovány tím, že pracují s označenými nebo kategorizovanými daty. Tyto algoritmy však nemohou plánovat a vyvodit dobrý závěr v dlouhodobém rozhodování. AI systémy, které replikují plánování a interaktivní chování ve hrách jsou typicky dělány pomocí pevně zakódovaných (hardcoded) systémů stavových strojů jako jsou konečné automaty (FSM) nebo rozhodovací stromy. Možnost, že se agent naučí nejlepší pohyby, neboli akce v daném prostředí, je klíčová, a to nejen pro herní průmysl. [7]

Hlavními postavami Reinforcement Learning (RL) jsou agent a prostředí (environment). Prostředí je svět, ve kterém agent žije a interaguje s ním. V každém kroku iterace agent pozoruje stav prostředí a rozhodne se, jakou provede akci. Prostředí se mění podle toho, jak s ním agent interaguje, ale může se změnit i samo o sobě. [8]

Agent také vnímá reward signal z prostředí, to je číslo, které říká, jak dobrý nebo špatný je aktuální stav prostředí. Cílem agenta je maximalizovat kumulativní odměnu (cumulative reward), nazývanou vrácení (return). RL metody jsou způsoby, jak se agent naučí chování, které povede k dosažení jeho cíle. [8]

Dále bude vysvětlena následující terminologie pro pochopení RL:

- stavy a pozorování (states and observations),
- prostory akcí (action spaces),
- politika (policy),
- trajektorie,
- různé formulace návratnosti (different formulations of return),
- hodnotící funkce (value functions),
- Markovské rozhodovací procesy.

2.2.1 Stavy a pozorování (States and Observations)

Stav je kompletní popis stavu prostředí. Neměla by být žádná informace prostředí, která by byla skryta před stavem. Pozorování (observation) je o částečném popsání stavu, který může vynechávat informace. [8]

V hlubokém RL jsou pozorování (observations) skoro vždy reprezentovány vektory, maticemi reálných čísel a tenzory vyššího řádu. [8] Tenzor je mnohadimenzionální vektor

reálných čísel. Dvoudimenzionální tenzor je v podstatě matice. Například vizuální pozorování (visual observations) by mohlo být reprezentováno RGB maticí jeho hodnot pixelů a stav robota by mohl být reprezentován jeho úhly kloubů a rychlostmi. [8]

V případě, že je agent schopný pozorovat kompletní stav prostředí, říká se, že je prostředí plně odpozorováno (fully observed). Když je agent schopný vidět částečné pozorování (partial observation), říká se, že je prostředí částečně odpozorováno (partially observed). [8]

2.2.2 Prostory akcí (Action spaces)

Rozdílná prostředí umožňují rozdílné druhy akcí. Sada validních akcí v daném prostředí je nazývána prostor akcí (action space). Některé prostředí jako Atari a Go mají diskrétní prostory akcí, kde je dostupný agentovi pouze konečný počet pohybů. Jiné prostředí, kde například agent ovládá robota ve fyzickém světě mají kontinuální prostor akcí (continuous action space). V těchto prostředích jsou akce vektory reálných čísel. [8]

Toto rozdělení má docela velké následky pro metody v hlubokém posilovaném učení. Některé rodiny algoritmů mohou být přímo aplikovány v jednom případě, ale musí být podstatně přepracovány pro jiné případy. [8]

2.2.3 Politiky (Policies)

Politika (policy) je pravidlo použité agentem, které mu slouží k rozhodnutí, kterou akci podniknout. [8]

Může být deterministická, kdy je značená μ:

$$a_t = \mu(s_t)$$

a_t: akce v daný čas t

s_t: stav v čase t

$\mu(s_t)$: funkce μ, která na základě daného stavu vrátí akci s_t.

Nebo může být stochastická, kdy je značená π:

$$a_t \sim \pi(\cdot \mid s_t)$$

a_t: akce v daný čas t

~: symbol znamená úměrnost nebo ekvivalenci

$\pi(\cdot|s_t)$: akce daná neurčeným stavem s_t

Deterministická politika je tedy postup, podle kterého má agent na výběr z každého stavu přesně jednu akci. Zato u stochastické politiky jsou akce vybírány na základě pravděpodobnostního rozdělení, což znamená, že stejný stav může vést k různým akcím podle určité pravděpodobnosti. [8]

V hlubokém RL se pracuje s parametrizovanými politikami: politiky, jejichž výstupy jsou vypočitatelné funkce, které závisí na sadě parametrů (například váhy a zaujatosti (biases) neuronových sítí), které můžeme měnit, abychom změnili chování skrze optimalizační algoritmus. Takovéto parametry jsou často označovány jako θ (théta) nebo ϕ (fí) a zapisují se jako dolní index symbolu politiky:

$$a_t = \mu_\theta(s_t)$$

$$a_t \sim \pi_\theta(\cdot\,|s_t)$$

Dvě nejvíce běžné stochastické politiky v hlubokém RL jsou kategorické politiky (categorical poilicies) a diagonální Gaussovy politiky (diagonal Gaussian policies). Kategorické politiky mohou být použity v diskrétním prostoru akcí, zatímco diagonální Gaussovy politiky jsou použity v kontinuálních prostorech akcí. Kategorická politika je něco jako klasifikátor diskrétních akcí. [8]

2.2.4 Trajektorie

Trajektorie τ je sekvence stavů a akcí ve světě (prostředí). Stavové přechody (co se stane mezi jedním a následujícím stavem) jsou řízené přírodními zákony daného prostředí a závisí pouze na nejnovější akci. Mohou být opět deterministické nebo stochastické. U stochastické přicházejí akce od agenta podle jeho politiky. [8] Učení s politikou tedy používá stochastické metody.

Trajektorie se také nazývá epizoda. Je to v podstatě cyklus, ve kterém agent začíná a končí učení zakončeném určitou odměnou. Trajektorie může být například 5000 kroků dlouhá, ale ukončí se dřív, když agent dokončí cíl.

2.2.5 Odměna a navrácení (Reward and Return)

Funkce odměny R je kriticky důležitá v posilovaném učení. Závisí na aktuálním stavu prostředí, akci právě provedené a na následujícím stavu prostředí. Cílem agenta je maximalizovat kumulativní odměnu v průběhu trajektorie (epizody).

Na druhou stranu navrácení (return) je suma odměn získaných v pevném rámci kroků. Takto je definované „finite-horizon undiscounted return“, což je jeden způsob navrácení. Druhý způsob navrácení je „infinite-horizon discounted return“, což je suma všech získaných odměn agentem, ale je sražena tím, v jakém čase jsou získány. Často se používá, že se nastaví algoritmy k optimálnímu nesraženému navrácení, ale používají se srážející faktory v odhadování hodnotící funkce. [8]

2.2.6 Hodnotící funkce (Value Functions)

Hodnotící funkce měří kvalitu stavu nebo jak úspěšný je stav nebo akce na základě predikce budoucí odměny. Budoucí odměna, známá také jako zmíněné navrácení (return), je, jak už bylo uvedeno dříve, suma všech sražených odměn v čase. Toto navrácení (G_t) má vzorec:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

[9]

Srážející faktor $\gamma \in <0,1>$ určuje důležitost odměn v budoucnu, protože:

- budoucí odměny mohou mít větší neurčitost, např. akciový trh;
- budoucí odměny nepřinášejí okamžité výhody (např. my jako lidé preferujeme mít se dobře dnes, než až za 5 let);
- srážení poskytuje matematickou pohodlnost (nepotřebujeme sledovat budoucí kroky do nekonečna, abychom spočítali návratnost);
- nepotřebujeme se starat o nekonečné smyčky u grafu přechodových stavů. [9]

Stavová hodnota (state-value) je očekávané navrácení (return), když jsme v tomto stavu v čase t, $S_t = s$:

$$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

[9]

E – značí očekávání

Podobně akční hodnota (action-value), popisována také jako Q-value, je v akčně-stavovém páru (state-action pair):

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$ [9]

Navíc, protože následujeme cílovou politiku π, můžeme využít rozložení pravděpodobnosti přes možné akce a Q hodnoty, abychom získali hodnotu stavu:

$$V_\pi(s) = \sum_{a \in \mathcal{A}} Q_\pi(s,a)\pi(a|s)$$ [9]

Rozdíl mezi akční hodnotou a stavovou hodnotou je funkce akční výhody (action advantage function) neboli také A-value:

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$$ [9]

2.2.7 Markovské rozhodovací procesy

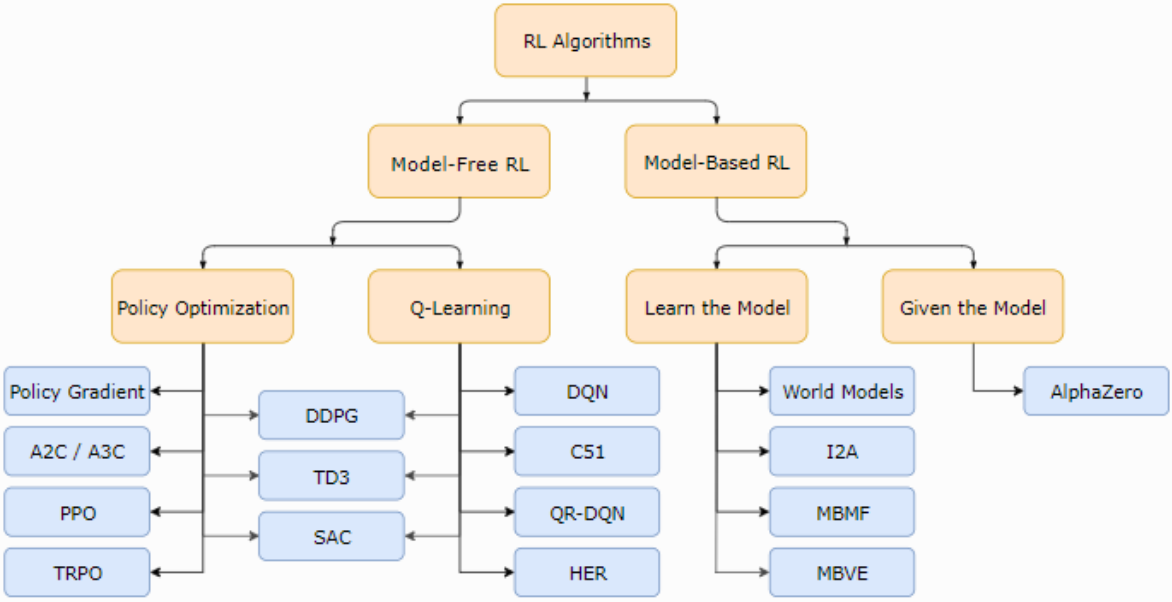
Dosud bylo diskutováno prostředí agenta neformálně. Standardním matematickým formalismem pro toto prostředí jsou Markovské rozhodovací procesy (MDPs). MDP je pětice ⟨S, A, R, P, ρ_0⟩, kde

- S je množina všech platných stavů;
- A je množina všech platných akcí;
- R: S × A × S → $\mathbb{R}$ je funkce odměny, kde $r_t = R(s_t, a_t, s_{t+1})$;
- P: S × A → $\mathcal{P}$(S) je funkce pravděpodobnosti přechodu, s P(s'|s,a) představující pravděpodobnost přechodu do stavu s', pokud se začne ve stavu s a provede se akce a;
- ρ_0 je distribuce počátečního stavu.

Název Markovské rozhodovací procesy odkazuje na skutečnost, že systém dodržuje Markovskou vlastnost - přechody závisí pouze na nejnovějším stavu a akci, nikoli na předchozí historii. [8]

2.3 Rozdělení posilovaného učení

Nyní po uvedení základů v terminologii strojového učení, je třeba uvést rozdělení moderních RL algoritmů. Podle Open AI jsou rozdělené následovně:



Obrázek 2 Taxonomie algoritmů posilovaného učení [10]

Model definuje funkci odměny a pravděpodobnosti přechodů. Podle toho, zda model je nebo není znám, rozlišujeme:

- **Model based RL**, kdy se používá plánování s perfektními informacemi. Prostředí je dokonale známé a optimální řešení se dělá pomocí dynamického programování.
- **Model free RL** je charakterizován učením s nekompletními informacemi. Algoritmus se snaží naučit model explicitně. [9]

Je velice obtížné sestrojit obrázek moderního rozdělení RL. V obrázku je vynecháno spoustu dalších zajímavých učení jako exploration, transfer learning, meta learning atd. Celkově pro účely práce postačí pouze tento obrázek, protože jsou zde znázorněny dva ze tří algoritmů z Unity Ml Agents Toolkit, a to jsou PPO a SAC. Poslední třetí algoritmus POCA patří do skupiny Policy Optimization.

Jelikož algoritmů, metod a pravidel od kterých se RL odvíjí je velké množství, budou dále uvedeny pouze metody, které jsou dostupné v Unity ML Agents Toolkit. Tyto metody spadají do skupiny „Policy Gradient“.

2.3.1 Policy Gradient metody

Jak název napovídá, tyto metody jsou policy-based. Použití metod s politikou je tam, kde je nekonečně mnoho stavů. V kontinuálním prostoru/prostředí je výpočetně mnohanásobně náročnější používat metody právě bez politiky. Je to metoda, která se zaměřuje na přímé učení politiky, tzn. model se učí pravděpodobnostní distribuci akcí, které by měl agent provádět v daných stavech. [11]

2.3.2 PPO

Proximal Policy Optimization je algoritmus RL, který patří právě do Policy Gradient metod. Byl vyvinut firmou OpenAI a je známý svou stabilitou a efektivitou. PPO zavedl mechanismus „clippingu“ pro omezení velikosti změn při verzování politiky. To pomáhá držet lépe stabilitu tréninku než jiné předchozí metody. Používá se pro to tzv. surrogátní cíl (surrogate objective), což je náhrada za původní cílovou funkci. [11]

PPO iterativně updatuje parametry théta dané politiky řešením lokálního optimalizačního problému:

$$\theta_{\text{new}} \leftarrow \arg\max_{\theta} \mathcal{L}(\theta; \hat{A}^{\pi_{\text{old}}})$$ [12]

Pro optimalizaci algoritmu se používá následující rovnice:

$$J^{\text{CLIP}}(\theta) = \mathbb{E}[\min(r(\theta)\hat{A}_{\theta_{\text{old}}}(s,a), \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_{\theta_{\text{old}}}(s,a))]$$ [11]

$J^{CLIP}(\theta)$ – cílová funkce v rámci PPO

E – značí očekávání

A – značí zde zmiňovanou výhodu

r(θ) – koeficient poměru (ratio factor)

Funkce clip(r(θ), 1 - ϵ , 1 + ϵ) sestřihne koeficient, aby nebyl víc než 1 + ϵ, ale ne míň než 1 - ϵ. Tato cílová funkce PPO vezme minimum z originální funkce akční výhody a její sestřihnuté (clipped) verze, proto není potřeba obrovských změn v politice, k dosažení lepších výsledků. [11]

Při aplikaci PPO na architekturu, která sdílí parametry pro funkce politiky (aktor) a hodnoty (kritik), je kromě klipované odměny cílová funkce rozšířena o chybu v odhadu hodnoty (vzorec červeně) a termín entropie (vzorec modře), k podpoření dostatečného průzkumu.

$$J^{\text{CLIP'}}(\theta) = \mathbb{E}[J^{\text{CLIP}}(\theta) - c_1(V_\theta(s) - V_{\text{target}})^2 + c_2 H(s, \pi_\theta(.))] \quad [11]$$

Kde c_1, c_2 jsou hyperparametry.

PPO má více návrhových vzorů, mezi než patří:

$$\begin{aligned}
\mathcal{L}^{\text{CLIP}}(\theta) &:= \mathop{\mathbb{E}}_{a,s\sim\pi_{\text{old}}}\left[\min\left(\frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)} A^{\pi_{\text{old}}}(a,s),\ \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\text{old}}}(a,s)\right)\right] \\
\mathcal{L}^{\text{KL,forward}}(\theta) &:= \mathop{\mathbb{E}}_{a,s\sim\pi_{\text{old}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)} A^{\pi_{\text{old}}}(a,s)\right] - \beta D_{\text{KL}}(\pi_{\text{old}} \| \pi_\theta) \\
\mathcal{L}^{\text{KL,reverse}}(\theta) &:= \mathop{\mathbb{E}}_{a,s\sim\pi_{\text{old}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)} A^{\pi_{\text{old}}}(a,s)\right] - \beta D_{\text{KL}}(\pi_\theta \| \pi_{\text{old}})
\end{aligned} \quad [12]$$

2.3.3 Soft Actor Critic (SAC)

SAC je založena na principu maximální entropie, který zvyšuje robustnost a schopnost průzkumu prostředí tím, že maximalizuje mimo odměny také entropii politiky. To vede k lepšímu průzkumu a stabilnějšímu učení. [13]

Algoritmus SAC využívá tzv. off-policy aktualizace. To umožňuje efektivnější využití učících dat oproti on-policy metodám, jako je PPO, protože tyto (on-policy) metody vyžadují nová data pro každý krok gradientu. Díky tomu je SAC méně citlivý na volbu hyperparametrů a na počáteční podmínky. To se projevuje konzistentnějšími výsledky. [13]

SAC demonstruje lepší výkon a vyšší rychlost učení oproti metodám jako DDPG a PPO na různých benchmarkových úlohách v kontinuálním prostředí. [13]

2.3.4 MA-POCA

Multi-Agent POsthumous Credit Assingment (MA-POCA) je vyvinut pro multiagentní prostředí. Stará se o agenty, kteří jsou vytvářeni nebo ničeni během epizody a sdílejí společnou funkci odměn. Pracuje se v rámci systému centralizovaného tréninku a decentralizovaného provádění. Je potřeba pouze umožnit kritikovi zvládat měnící se počet agentů v každém časovém kroku. MA-POCA překonává PPO ve scénářích, kde mohou agenti během epizody ukončit svou činnost nebo se objevit. MA-POCA je zvláště účinná v těchto dynamických prostředích, protože dokáže přesně přiřadit kredit agentům bez výpočetního zatížení udržování absorpčních stavů. [14]

3 KNIHOVNA UNITY ML-AGENTS

3.1 Základní charakteristika

Unity ML Agents Toolkit je open source projekt, který poskytuje hrám a simulacím prostředí pro trénování inteligentních agentů. Poskytuje implementace (založené na PyTroch) nejmodernějších algoritmů, které umožňují vývojářům her a nadšencům lehce trénovat inteligentní agenty ve 2D, 3D a VR/AR hrách. Výzkumníci mohou také použít lehce použitelné Python API k trénování agentů pomocí posilovacího učení, imitačního učení, neurorevoluce nebo jakékoliv jiné metody. Tito natrénovaní agenti mohou být užiti k mnoha účelům, včetně kontrolování NPC chování (v rozmanitém nastavení jako multi-agent a adversarial), například pro automatické testování herních buildů a hodnocení různých herních design rozhodnutí v předběžném vydání (hry, robota, simulace). ML-Agents Toolkit je vzájemně prospěšný jak pro vývojáře her, tak pro AI výzkumníky, jelikož poskytuje centrální platformu, kde pokroky v AI mohou být hodnoceny v bohatých Unity prostředích a jsou dostupné pro širší výzkum a komunitu herních vývojářů. [15]

3.2 Připravené funkce knihovny

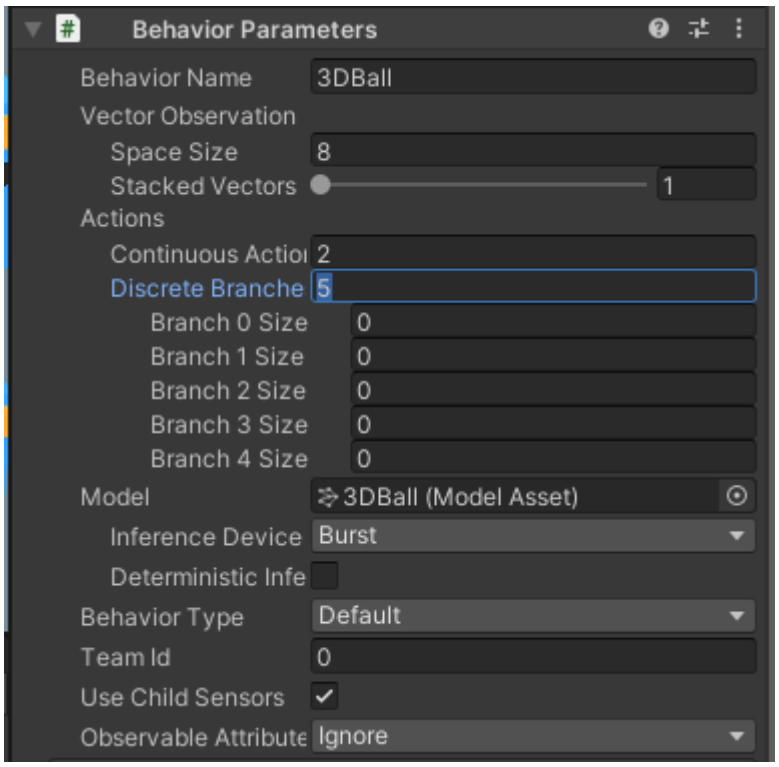
Unity ML-Agents je volně dostupný ke stažení. Obsahuje 17 příkladných učících prostředí, které nabízejí spoustu funkcí této knihovny. Tyto prostředí mohou také sloužit jako šablony pro nové prostředí nebo také slouží k testování nových ML algoritmů. Pro prostředí, které vyzdvihují určité funkce knihovny, poskytuje knihovna předtrénované soubory modelů a trénovací konfigurační soubor, který umožňuje uživateli trénovat scénu osobně. Prostředí, která jsou navržena jako výzva pro výzkumníky, neobsahují předtrénované modelové soubory, ani trénovací konfigurační soubory. [16]

3.3 Agent

Je potřeba definovat, co je vlastně agent. Agent je autonomní herec, který pozoruje a interaguje s prostředím. V kontextu s Unity, prostředí je scéna, který obsahuje jednoho nebo více agentů a samozřejmě další entity, se kterými agent interaguje. Agent má na starosti provádění akcí, sbírání pozorování (agnl. Collect Observations) a přidělování odměn. [17]

3.4 Behavior Parameters („Brain“)

Tato komponenta, která se v předchozích verzích jmenovala Brain, v podstatě určuje vlastnosti, které dále určují, jak ML-Agent poběží a konfiguruje nastavení parametrů mozku. Ve zkratce provádí rozhodnutí.



Obrázek 3 Parametry chování v knihovně ML-Agents

Tato komponenta se skládá z několika parametrů [1]:

3.4.1 Behavior Name

Jméno této komponenty slouží k identifikaci agenta, proto by mělo být unikátní, pokud agenti nesdílejí stejné funkcionality. [1]

3.4.2 Vector Observations

Tato komponenta říká mozku, kolik vstupů očekávat. Skládá se z počtu space size (velikosti prostoru) a stacked vectors (naskládaných vektorů). Hodnota space size se musí rovnat počtu vstupů, které kód předává ML-Agents. Stacked vectors na druhou stranu udává, s kolika kroky z minulosti bude pracovat. Např. Vector3 (poloha) obsahuje 3 Observations pro x, y i z a pokud by agent pracoval se třemi kroky pro lepší vyhodnocování, celkem by to bylo 9 Observations. Nutno podotknout, že čím vyšší hodnotu bude mít stacked vectors, tím by měl

být pro určité případy funkčnější trénink, avšak tím bude trénink i pomalejší. Pokud budeme mít space size 20 a stacked vectors 4, bude celkem 80 Observations, což může být pro nějaký hardware a rychlost tréninku docela náročné. [1]

3.4.3 Vector Actions

Toto pole nám umožňuje specifikovat, kolik různých spojitých akcí a diskrétních větví bude agent podporovat. [1], [18] To v praxi znamená, že pro spojité akce se vybere například zrychlení, které bude nabývat hodnot např. od 0 do 1. Pracuje se s reálnými čísly (float). Zkrátka tato akce slouží pro jemné změny výstupu, tedy spojitá akce. Zato diskrétní větve můžou být například dvě a určuje se u nich ještě velikost větví. Například pro jednoduchý pohyb můžou být dvě diskrétní větve, jedna pro pohyb nahoru a dolů (o velikosti větve dva) a druhá pro pohyb doleva a doprava (také o velikosti větve dva).

3.4.4 Model

Model je mozek agenta. Je to „snapshot“ formovaných neuronů a jejich vah. K modelu se váže „Inference device“ a Deterministic inferenece. Do vstupního pole pro model se vkládá nebo ukládá modelový soubor, který se dostane exportováním natrénovaného agenta. Tento model je to, co umožňuje našemu agentovi fungovat a může se serializovat, kompilovat do buildu pro používání za běhu. Jakmile se natrénuje agent a generuje jeho model, výkon agenta se značně zvýší. Inference device je fyzické zařízení, které ML-Agent používá pro odvozování závěrů nebo přehrávání závěrů, nikoliv pro trénování. [1] Volba zaškrtnutí Deterministic Inference napovídá, že agent by pro stejný vstup vyhodnotil pokaždé stejný výstup, bez možnosti náhody.

3.4.5 Behavior Type

Jsou celkem tři rozdílné typy chování: Default, HeuristicOnly, InferenceOnly. Tyto typy chování určují, kde agent vykoná rozhodnutí. Typ chování Default používá pro rozhodování agenta vzdálený přístup PyTroch. Pokud se tento vzdálený přístup nepovede, přepne se na Inference. A pokud Inference nebude správně fungovat, přepne se na Heuristic. HeuristicOnly je typ chování, který, jak název naznačuje, používá heuristiku. To znamená, že je potřeba neuronové síti poskytnout manuální akce, v podstatě dělat rozhodnutí námi samotnými, typicky skrze input uživatele za použití Unity Input System. Typ chování InferenceOnly vždy použije “inference” (závěry, odvození) na poskytnutém modelu. To

znamená, že agent provede rozhodnutí za použití již zmíněného “model by inference”. Nastavení Default se používá pro trénování agenta. [1]

3.4.6 Team Id

Tato hodnota specifikuje tým, do kterého by měl ML-Agent přispívat, což je vhodné pro týmově založené ML-Agenty. [1]

3.4.7 Use Child Sensors

Toto zaškrtávací políčko umožňuje říci agentovi, aby našel a použil jakékoliv sensory, které lze najít v jeho vlastní “child” hierarchii. Toto způsobuje malý výkonnostní dopad a mělo by být toto políčko nezaškrtnuté, pokud nejsou na agentovi dětské (child) sensory. [1] Zde lze použít jako child sensory komponentu Ray Perception Sensors, která podle nastavení snímá prostředí.

3.4.8 Observable Attributes

Zde je možné říci agentovi, jestli by měl použít „reflection“, jestli je použitý „[Observable]” atribut. Použití reflection způsobuje velké výkonnostní tresty, a pokud se nepoužívá takovýto atribut, pak by tato funkce měla být vypnutá. [1]

3.4.9 Decision Requester

Tento parametr určuje, jak často se má agent rozhodovat na základě aktuálních pozorování. Pokud je tento parametr nastaven na 5, agent každých 5 snímků provede rozhodnutí.

3.5 PyTorch a TensorBorad

Mnohé algoritmy poskytnuté knihovnou ML-Agent Toolkit využívají nějakou formu hlubokého učení. Implementace jsou postaveny na open-source knihovně PyTorch. V této sekci je potřeba zmínit lehký přehled této knihovny a také frameworku TensorBoard, které jsou využívány v knihovně ML-Agents Toolkit. [16]

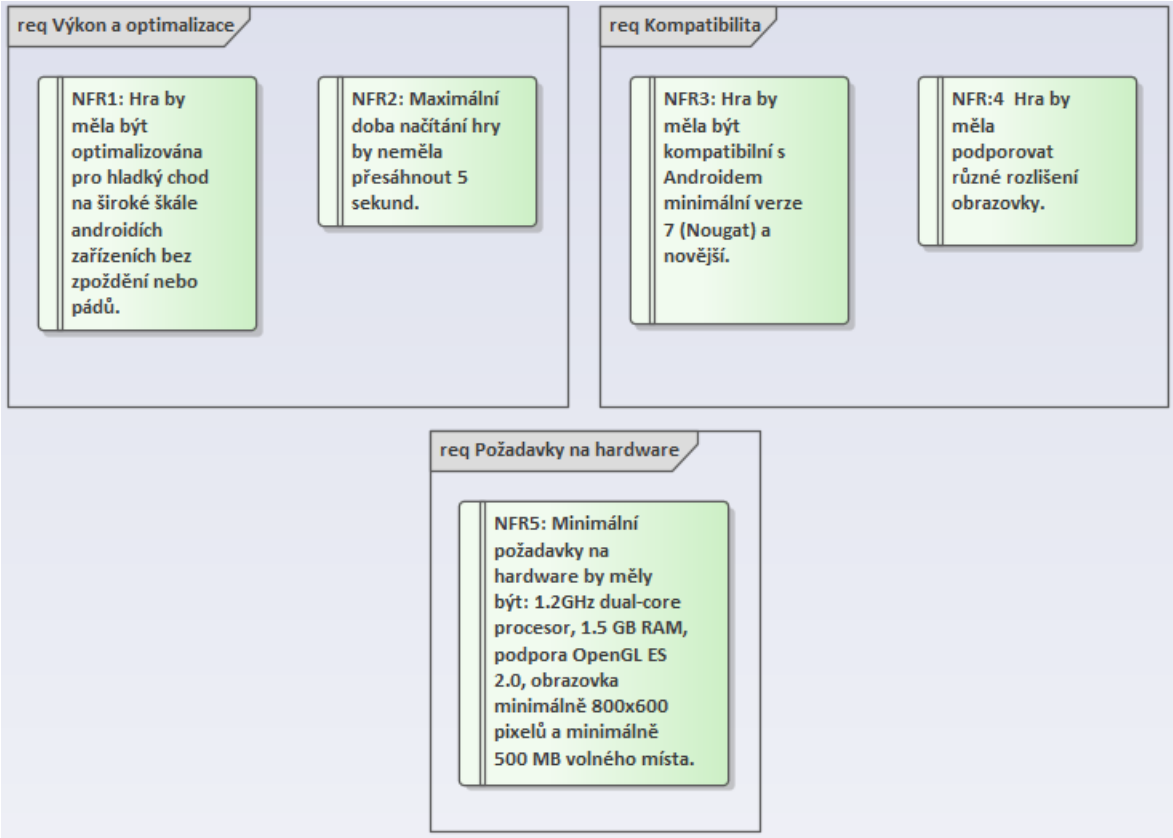
Pytorch je volně dostupná knihovna vykonávající výpočty pomocí tzv. “data flow graphs” neboli grafů datových toků, což je zásadní reprezentace modelů hlubokého učení. Umožňuje to trénink a vyvozování závěrů na CPU a GPU na stolním počítači, serveru nebo mobilním zařízení. V rámci ML-Agents Toolkit, chování agenta, tento výstup je soubor modelu (s koncovkou .onnx), který jsme schopni asociovat s agentem. Pokud neimplementujeme nový algoritmus, použití PyTorch je většinou v pozadí a my o něm nevíme. [16]

Zajímavá komponenta učení modelů pomocí PyTroch je nastavení si hodnot určitých vlastností modelu, nazývané hyperparameters. Najít správné hodnoty těchto parametrů může trvat několik iterací. Proto je doporučován vizualizační nástroj nazývaný TensorBoard. Umožňuje vizualizaci určitých vlastností agenta (např. odměna – reward) po celou dobu tréninku, což je užitečné pro obojí budování intuice pro různé hyperparametry i nastavení optimálních hodnot pro prostředí (environment). [16]

II. PRAKTICKÁ ČÁST

4 NÁVRH HRY

4.1 Nefunkční požadavky



Obrázek 4 Nefunkční požadavky

Obrázek 4 popisuje nefunkční požadavky pro vyhotovení mobilní hry. Hra je určena pro Android, a to pro verzi 7 (Nougat) nebo vyšší. Požadavky jsou navrženy v aplikaci Enterprise Architect.

4.2 Funkční požadavky

Funkční požadavky pro zvolenou hru jsou navrženy následovně:

4.2.1 Herní mechanismy

- **REQ1.** Ovládání letadla - hráč bude ovládat letadlo prostřednictvím joysticku na obrazovce. Joystick bude sloužit jen pro pohyb v ose y, protože letadlo poletí automaticky.
- **REQ2.** Hráč bude moci střílet pomocí dotykového tlačítka.
- **REQ3.** Hráč bude moci udělat s letadlem přetočení pomocí dotykového tlačítka pro smysluplné ovládání změny směru v ose x.

4.2.2 Skórování a odměny

- **REQ4.** Při sestřelení AI protivníka nebo naopak se přičte vítězi jeden bod a vítěz se znovu zrodí na jiném místě v prostředí. Kdo dřív nasbírá 10 bodů, vyhraje celou hru.

4.2.3 Prostředí

- **REQ5.** Prostředí bude ohraničené.
- **REQ6.** Při srážce hráče nebo protivníka se zemí, se danému aktérovi zničí a znovu zrodí letadlo, avšak prohraje dané kolo.
- **REQ7.** Prostředí bude koncipováno s tzv. "wrap-around" efektem na horizontálních okrajích (vlevo a vpravo). Tento efekt zajišťuje, že jak hráč, tak AI, po dosažení jedné strany obrazovky, okamžitě a plynule přejdou na opačný okraj.
- **REQ8.** Při nárazu letadla do horní hranice prostředí se letadlo odvrátí na druhou stranu, aby nedocházelo k zaseknutí.

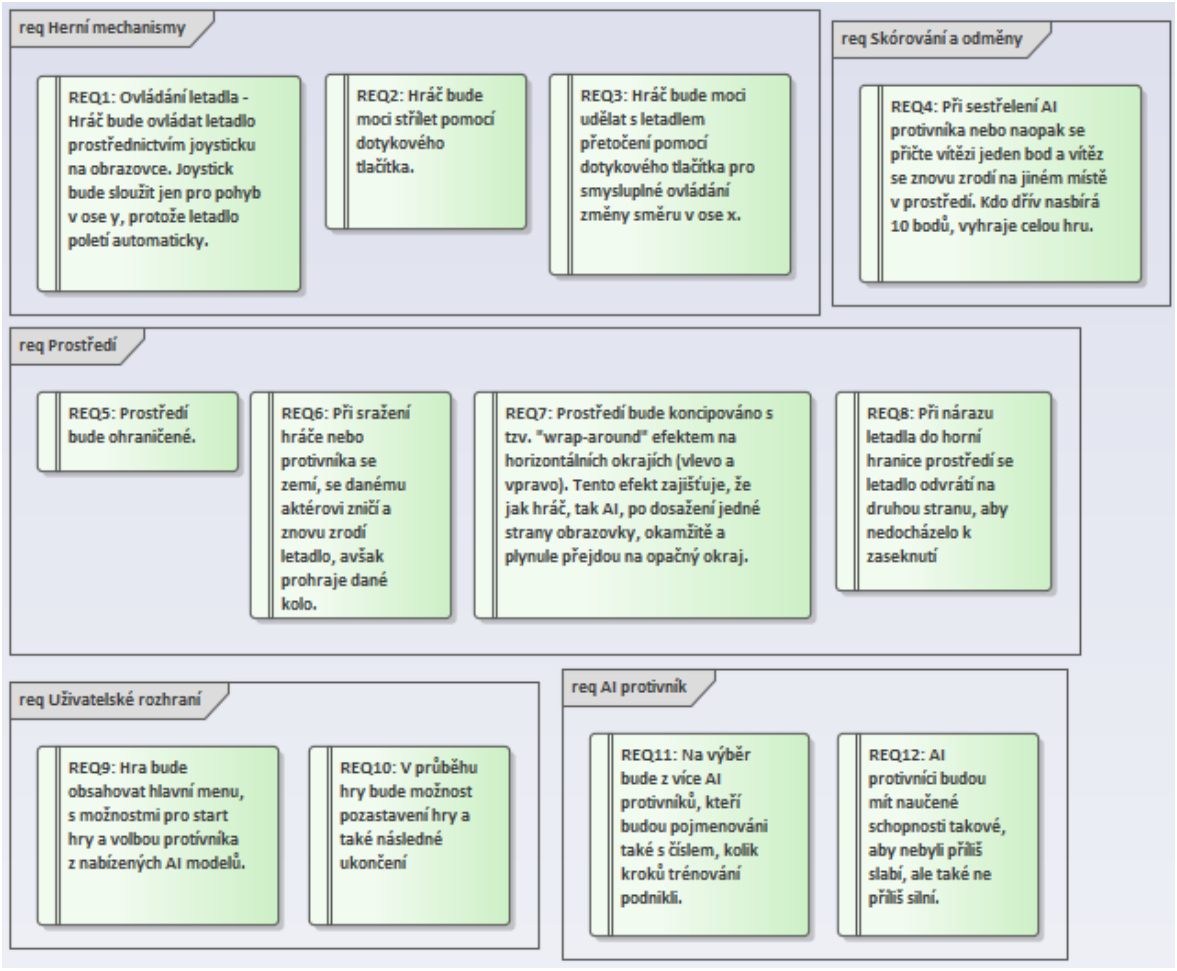
4.2.4 Uživatelské rozhraní

- **REQ9.** Hra bude obsahovat hlavní menu s možnostmi pro start hry a volbu protivníka z nabízených AI modelů.
- **REQ10.** V průběhu hry bude možnost pozastavení hry a také následné ukončení.

4.2.5 AI protivník

- **REQ11.** Na výběr bude z více AI protivníků, kteří budou pojmenováni také s číslem, kolik kroků trénování podnikli.
- **REQ12.** AI protivníci budou mít naučené schopnosti takové, aby nebyli příliš slabí, ale také ani příliš silní.

Pro názornost je zde přiložen obrázek návrhu funkčních požadavků, vytvořený také v Enterprise Architect.



Obrázek 5 Funkční požadavky

Jedná se o hru ve 2D prostoru, kde hráč bude ovládat své letadlo joystickem nahoru a dolů a k přetočení a ke střelbě budou sloužit speciální tlačítka. Cíl hry bude nasbírat 10 bodů, každý sestřelením protivníka.

Hra nebude dokonalá pro publikaci (zvukové efekty, vizualizace), hra bude pouze demonstrovat použití knihovny ML-Agents Toolkit.

5 VYHOTOVENÍ HRY

Tato část demonstruje implementaci a funkčnost vytvořené hry s využitím ML-Agents. Jsou zde popsány možnosti aplikace metod strojového učení na praktické ukázce. Tato metoda se nyní používá stále častěji a nahrazuje staré metody (pevně zakódované chování). Je to tím, že s flexibilitou dobře naučeného agenta a rychlostí jeho trénování, díky stále lepšímu hardwaru, se možnosti značně mění ve prospěch strojového učení.

Aby bylo možno začít pracovat na praktické části, musela být prostudována literatura specializovaná na vývoj 2D her v Unity [19], literatura o knihovně ML-Agents [1] [3] [7] [4], dále musely být prostudovány různé tutoriály [30] [20] [21] [22] [23] [24] [25] a také oficiální dokumentace Unity ML-Agents [15] [16] [26] [27] [28]. Dokumentace Unity ML-Agents však neposkytovaly aktuální a dostačující informace, proto bylo potřeba hodně experimentovat a zkoušet různé možnosti.

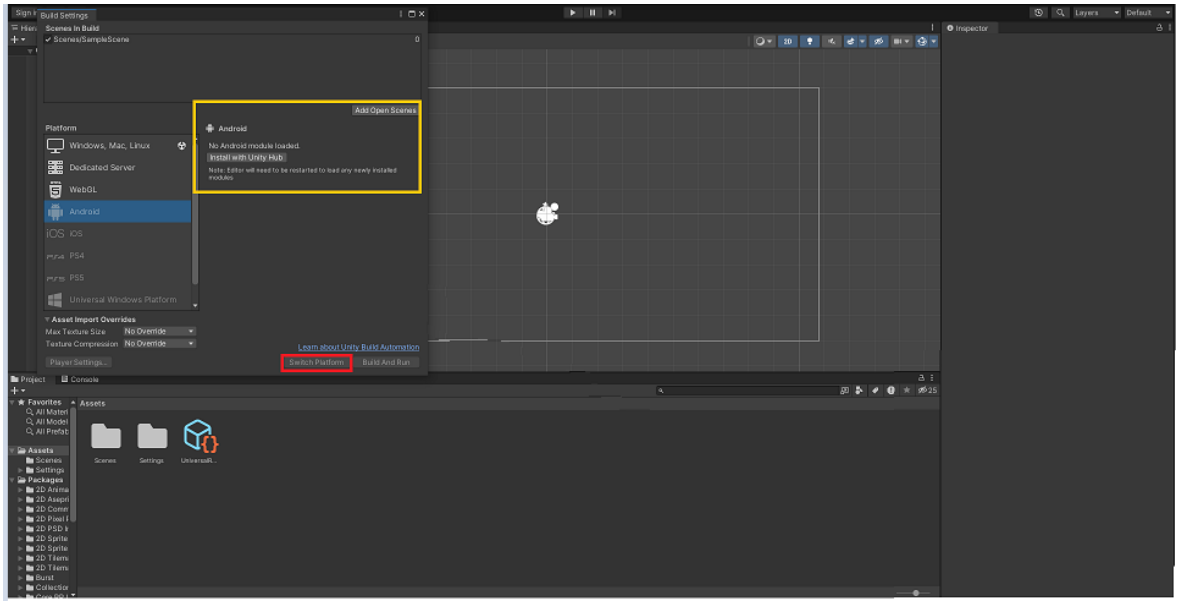
5.1 Instalace a konfigurace ML-Agents

Instalace všech potřebných nástrojů je prvním krokem pro zahájení vývoje hry, využívající zmíněné metody strojového učení. Unity je nejrozšířenější herní engine, který umožňuje vytvářet komplexní herní a simulované prostředí. Pro instalaci nejnovější verze se doporučuje použít Unity Hub, který je dostupný na oficiálních stránkách Unity. V Unity Hub už pak stačí vybrat požadovanou verzi Unity nebo více verzí a nainstalovat. Dále je potřeba stáhnout nebo klonovat repozitář ML-Agents z GitHubu, dostupný zde. [15]

Pro fungování Ml-Agents je potřeba Python 3.6 nebo novější. Doporučuje se vytvořit izolované virtuální prostředí pro zajištění kompatibility všech komponent bez ovlivnění globální instalace pythonu. Existují dvě hlavní metody a to buď přes nástroj Anaconda nebo venv. V této práci byl použit nástroj Anaconda. Obě metody se konfigurují přes příkazový řádek. V prostředí je potřeba nainstalovat všechny potřebné závislosti jako Pytorch, TensorFlow, GRPC, NumPy a další. A nakonec je potřeba nainstalovat samotnou knihovnu ML-Agents.

5.2 Nastavení prostředí a herní mechaniky

Při prvním spuštění Unity je potřeba nainstalovat Android SDK a NDK a nastavit vývojovou platformu na Android.



Obrázek 6 Vývoj pro Android

Pojmů pro vývoj ve 2D prostředí je mnoho. Tomu se tato práce nevěnuje, avšak pro tuto ukázku je důležité alespoň vědět, co je scéna a *GameObject*. *GameObject* je základní stavební prvek, který se vkládá do scény formou různých tvarů, jako jsou například čtverce, kruhy, trojúhelníky, různé 2D obrázky (sprity), atd. Tyto *GameObjecty* se mohou také inicializovat a ničit pomocí kódu. Scéna je celé plátno, na kterém se pracuje.

Pojmy *agent* a *prostředí* už byly vysvětleny. *Prostředí* je ohraničená oblast, ve které *agent* působí. Ve scéně může být mnoho prostředí, ale je to náročnější na hardware.

Do nově vytvořeného projektu bylo potřeba nejprve vytvořit prostředí a muselo se vytvořit několik herních objektů - hranice prostředí, pozadí prostředí, hráč, agent, projektil, trénovací cíl, joystick, tlačítko na střelbu, tlačítko na přetočení, kamera, aj. Tyto herní objekty také potřebují skripty, které definují jejich chování.

Nejdříve byl vytvořen skript pro pohyb letadla a jeho následování kamerou.

5.2.1 Skript pro pohyb letadla a jeho popis

Zde je skript, který byl napsán jako první pro hráče:

```
public class PlaneMovement : MonoBehaviour
{
    public Joystick joystick;
    public Button shootButton;
    public Button flipButton;
    private SpriteRenderer spriteRenderer;
    private bool isFlipped = false;
    public float speed;
    public float acceleration;
```

```
    private Rigidbody2D rb;
    public float rotationControl;
    float movY, movX = 1;

    public GameObject bulletPrefab;
    public Transform bulletSpawnPoint;
    public float bulletSpeed = 20f;
    public float shootingRate = 0.5f;
    private float shootCooldown;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        spriteRenderer = rb.GetComponent<SpriteRenderer>();
    }

    void Update()
    {

        shootCooldown -= Time.deltaTime;

        if (!isFlipped)
        {
            movY = joystick.Vertical;
        }
        else movY = - joystick.Vertical;


    }


        private void FixedUpdate()
    {
        Vector2 Vel = transform.right * (movX * acceleration);
        rb.AddForce(Vel);

        float Dir = Vector2.Dot(rb.velocity, rb.GetRelativeVector(Vector2.right));

        if (acceleration > 0 )
        {
            if(Dir>0)
            {
                rb.rotation += movY * rotationControl * (rb.velocity.magnitude / speed);
            }
            else
            {

                rb.rotation -= movY * rotationControl * (rb.velocity.magnitude / speed);
            }

        }

        float thrustForce = Vector2.Dot(rb.velocity, rb.GetRelativeVector(Vector2.down))
* 2.0f;

        Vector2 relForce = Vector2.up * thrustForce;

        rb.AddForce(rb.GetRelativeVector(relForce));

        if(rb.velocity.magnitude > speed )
        {
            rb.velocity = rb.velocity.normalized * speed;
        }
```

```
    }


    public void Shoot()
    {
        if (bulletPrefab != null && bulletSpawnPoint != null)
        {
            GameObject bullet = Instantiate(bulletPrefab, bulletSpawnPoint.position,
bulletSpawnPoint.rotation);
            bullet.GetComponent<Rigidbody2D>().velocity = bulletSpeed * bulletSpa-
wnPoint.up;
            Destroy(bullet, 2f);
            bullet.AddComponent<BulletBehavior>();

        }
    }

      public void FlipPlane()
    {
        transform.Rotate(180, 0, 0);

        if (isFlipped == false) { isFlipped = true; }
        else isFlipped = false;
    }


    public class BulletBehavior : MonoBehaviour
{
   void OnTriggerEnter2D(Collider2D other)
        {

            if (/*other.CompareTag("Enemy") ||*/ other.CompareTag("Target"))
            {
                Destroy(other.gameObject);
                Destroy(gameObject);
            }


        }
}

}
```

5.2.1.1 Deklarace proměnných

- *public Joystick joystick*: Přiřazení joysticku pro ovládání letadla
- *public Button shootButton, public Button flipButton*: Tlačítka pro střelbu a otáčení letadla.
- *private SpriteRenderer spriteRenderer*: Komponenta pro vykreslování spritů. Používá se k zobrazení vizuálního aspektu letadla.
- *private bool isFlipped = false*: Proměnná, která určuje, zda je letadlo otočeno.
- *public float speed, public float acceleration*: Proměnné pro rychlost a zrychlení letadla.
- *private Rigidbody2D rb*: Komponenta Rigidbody 2D určuje fyziku daného objektu.
- *public float rotationControl*: Proměnná pro ovládání rotace letadla.
- *float movY, movX = 1*: Defaultní nastavení hodnot pohybu na osách Y a X.

5.2.1.2 Inicializační metoda Start

- *Start()* je volána při spuštění skriptu a slouží k inicializaci. Zde se získává komponenta Rigidbody 2D a SpriteRenderer z aktuálního GameObjectu.

5.2.1.3 Metoda Update

- *Update()* se volá každý snímek a zpracovává vstupy od uživatele, aktualizuje stav cool-downu střelby, zpracovává vertikální pohyb joysticku a mění jeho input, podle toho, zda je letadlo otočené.

5.2.1.4 Metoda FixedUpdate

- *FixedUpdate()* metoda slouží v Unity pro aktualizace, které se týkají fyziky. Její frekvence volání se může lišit v závislosti na výkonu a zatížení systému. Volá se v pravidelných intervalech (např. každých 0.02 sekundy, což je 50 volání za sekundu). Tento interval lze nastavit v Unity Editoru pod Time Managerem. Používá se pro aplikaci sil v RigidBody (fyzika objektu). V této metodě je celá logika pohybu a rotace letadla na základě jeho zrychlení a aktuálních rychlostí.

5.2.1.5 Metoda Shoot

- *Shoot()* je metoda logiky střílení. Zde je instanciován projektil a nastavuje se zde doba, po jaké se projektil zničí.

5.2.1.6 Metoda FlipPlane

- *FlipPlane()* otočí letadlo o 180 stupňů. Slouží k převrácení letadla vzhůru nohama.

5.2.1.7 Vnitřní třída BulletBehavior

- *BulletBehavior* je komponenta přiřazená každému vystřelenému projektilu. Obsahuje metodu OnTriggerEnter2D, která detekuje kolize s jinými objekty.

5.2.2 Script pro kameru a popis

```
public class CameraFollow : MonoBehaviour
{
    public GameObject player;
    public Vector3 offset;

    void Start()
    {
```

```
    }

    void FixedUpdate()
    {
        if (player != null)
        {
            Vector3 camPos = transform.position;
            Vector3 desiredPos = player.transform.position + offset;

            Vector3 smoothedPos = Vector3.Lerp(camPos, desiredPos, 0.125f);

            float distanceToTarget = Vector2.Distance(player.transform.localPosition,
transform.localPosition);


         if (distanceToTarget > 85f){
            transform.position = player.transform.localPosition;
         }
         else
            transform.position = new Vector3(smoothedPos.x, smoothedPos.y, smoothed-
Pos.z);
        }
    }
}
```

Tento skript umožňuje kameře plynule sledovat hráče v daném prostředí.

5.2.2.1 Deklarace proměnných

- *public GameObject player*: Tato proměnná umožňuje v Inspectoru přiřadit herní objekt (GameObject), který má kamera sledovat. V tomto případě hráčovo letadlo.
- *public Vector3 offset*: Definuje odstup kamery od hráče.

5.2.2.2 Metoda Start

- Již zmíněná metoda, v tomto skriptu je prázdná, protože veškerá inicializace proměnných je řešena přímo v editoru Unity.

5.2.2.3 Metoda FixedUpdate

- Také již zmíněná metoda. V tomto případě je použita pro aktualizaci pozice kamery tak, aby odpovídala pozici hráče s určitým zpožděním a plynulostí.
- *Vector3 camPos = transform.position*: Ukládá aktuální pozici kamery do lokální proměnné.

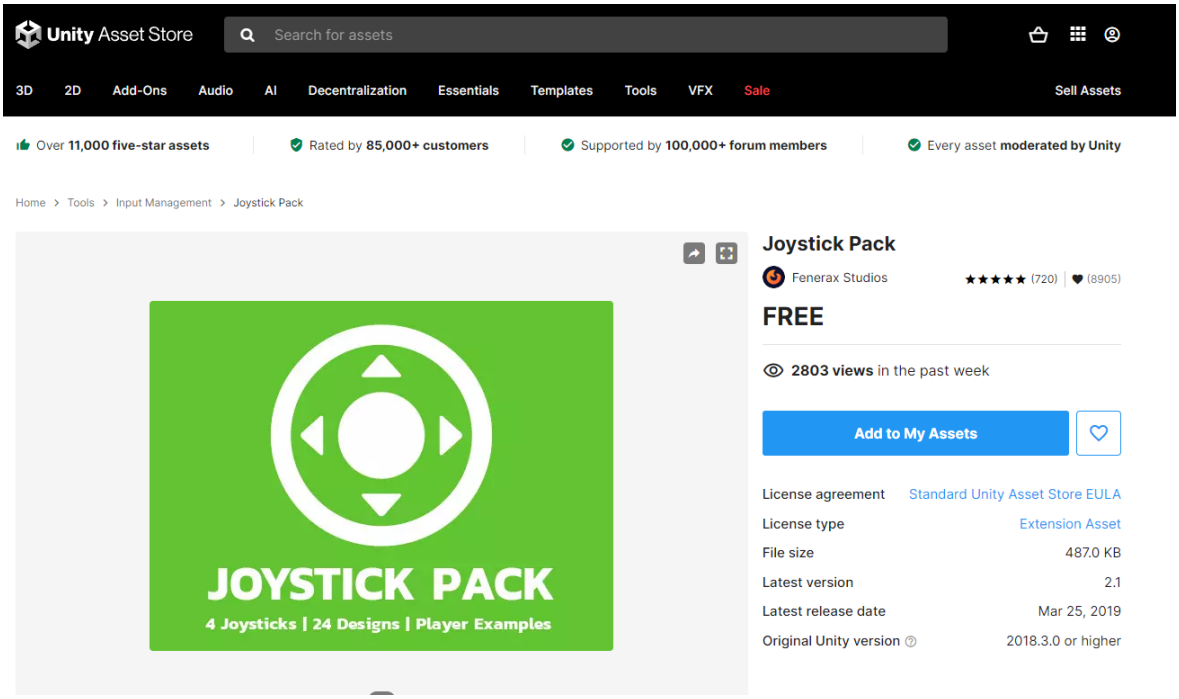
- *Vector3 desiredPos = player.transform.position + offset*: Vypočítává cílovou pozici kamery, tedy hráče s přidaným offsetem.
- *Vector3 smoothedPos = Vector3.Lerp(camPos, desiredPos, 0.125f)*: Je zde použita lineární interpolace (Lerp) pro plynulé přesouvání kamery. Číslo 0.125f ovlivňuje plynulost sledování.
- *(distanceToTarget > 85f) { transform.position = player.transform.localPosition; }*: Tato část kódu slouží k přesunutí kamery na hráče v případě, kdy hráč naletí do okraje a objeví se na druhé straně prostředí.

5.2.3 Uživatelské rozhraní

Prostředí nyní vypadalo následovně.

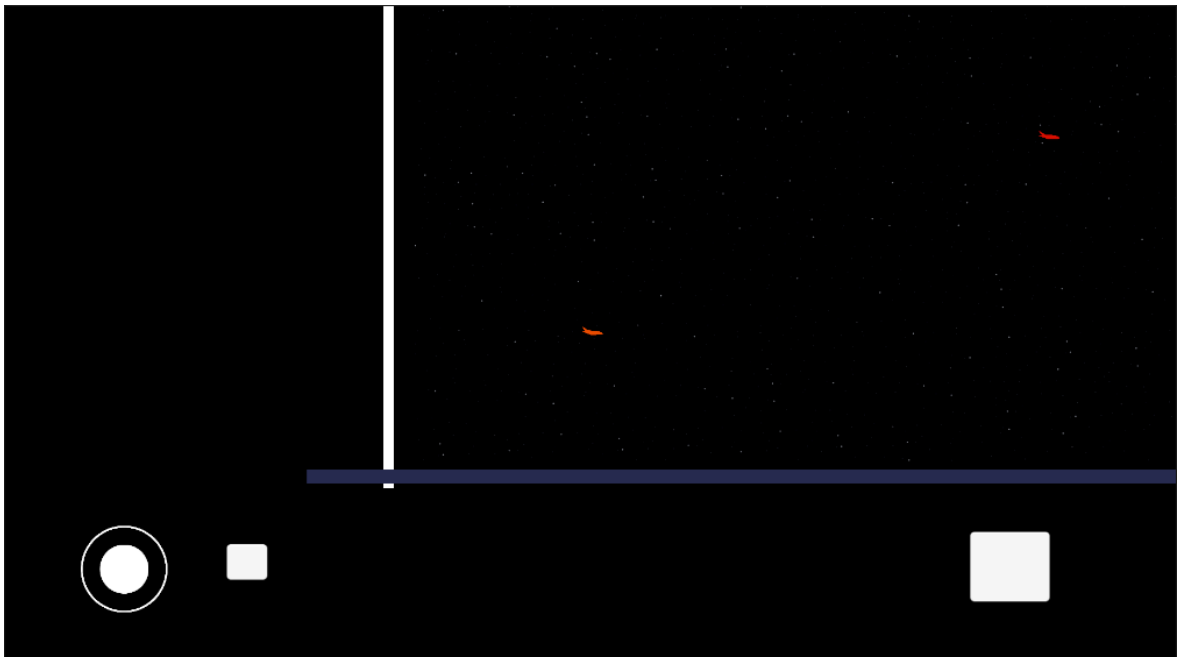
Obrázek 7 Vytvořené prostředí

Všechny hry a aplikace potřebují uživatelské rozhraní, tato hra není výjimkou. Nejprve bylo potřeba dodat joystick, aby se letadlo (říkejme hráč) dalo ovládat. K tomuto účelu se zde použil volně dostupný joystick z Unity Asset Storu. [29]



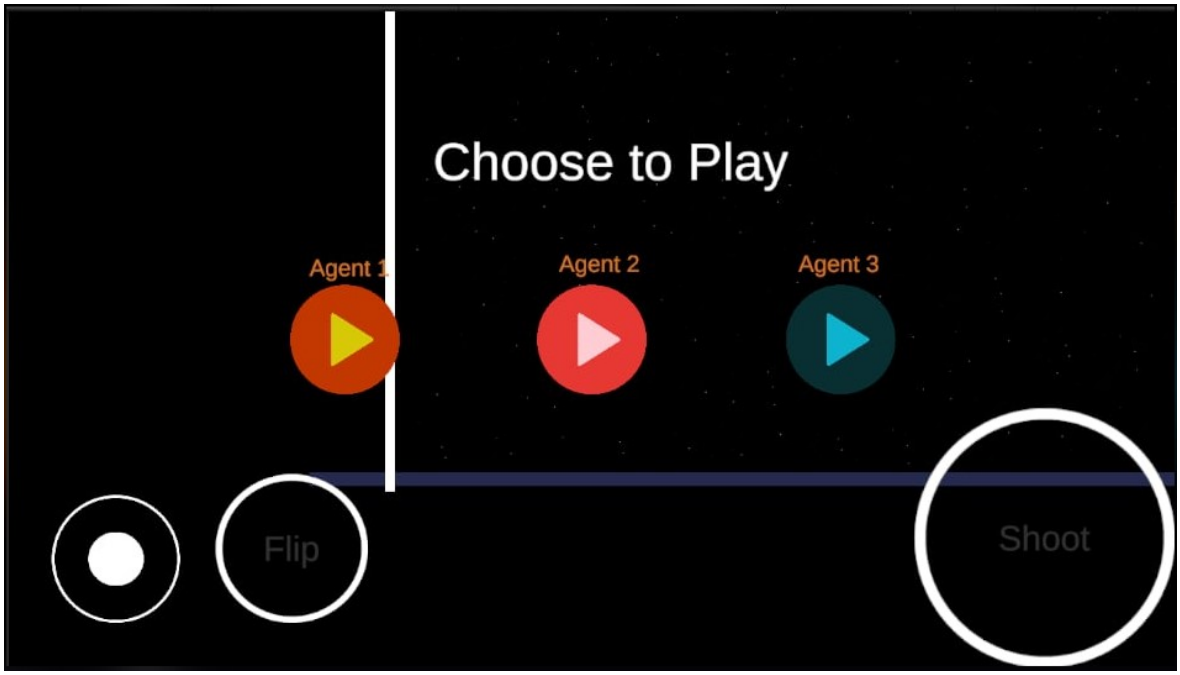
Obrázek 8 Free Joystick z Asset storu [29]

Pro ovládání letadla ve 2D prostoru postačí pohyb joystickem pouze vertikálně. Bylo ale potřeba přidat tlačítko pro otočení letadla, aby mohlo létat oběma směry. Dále se přidalo tlačítko na střelbu a základní logika byla hotová.



Obrázek 9 Joystick, tlačítka

Vytvoření hlavního menu bude lépe rozepsáno ke konci této práce. Pro souvislost postupu práce zde postačí pouze vzhled koncové verze použitého UI.



Obrázek 10 Konečná verze UI

5.3 Vytvoření AI modelu

Dokumentace k ML-Agents je poněkud zastaralá a fádní. V základních komponentách je uveden „Brain“ a ten je nyní přejmenován jako Behavior parameters. Mozek samostatný je nazýván jako NN network neboli NN network model. Proto bylo potřeba hodně experimentovat a zkoušet různé možnosti.

5.3.1 Skript pro agenta a jeho popis

Aby bylo prostředí plně definováno, bylo potřeba vytvořit agenta neboli AI, které by se trénovalo. Bylo nutné zkopírovat GameObject hráče a upravit jeho komponenty a script. Pro účely imitačního učení bylo ještě potřeba přidat komponentu Demonstration Recorder, která slouží k nahrávání demonstrací ukázek, podle kterých by se mohl agent učit. Konečná implementace agenta, jeho scriptu se správnými odměnami vypadala následovně:

```
public class AgentXXEnemy : Agent
{
    public RayPerceptionSensorComponent2D raysUp;
    public RayPerceptionSensorComponent2D rayDown;
    public RayPerceptionSensorComponent2D rayBack;
    public RayPerceptionSensorComponent2D raysForward;
    private float maxNearMissReward = 0f;
    private float maxHitReward = 0.05f;
    private float maxDistanceForReward = 3.0f;
    public GameObject agentPrefab;
    public AgentManager agentManager;
    public GameObject targetEnemy;
    private Transform targetEnemyTransform;
    private List<GameObject> bullets = new List<GameObject>();
    private GameObject newBullet;
    private bool targetHit;
    public float speed;
    public float acceleration;
    public float rotationControl;
    float movY = 0;
    float movX = 1;
    private bool isFlipped = false;
    public GameObject bulletPrefab;
    public Transform bulletSpawnPoint;
    private Rigidbody2D rb;
    private SpriteRenderer spriteRenderer;
    public float bulletSpeed = 20f;
    public float shootingRate = 0.5f;
    public float shootCooldown = 0.2f;
    float startTime;
```

```
 float maxTime = 30.0f;
float maxReward = 0.3f;
private GameObject currentTarget;
public Transform environmentTransform;
private int numberOfShots = 0;
public Vector2 newPosition;
private Vector2 updatedEnemyPosition;
private BulletBehavior bulletScript;

public override void Initialize()
{
    rb = GetComponent<Rigidbody2D>();
    spriteRenderer = GetComponent<SpriteRenderer>();
    bulletSpawnPoint = this.transform.Find("BulletSpawnPoint");
    if (bulletSpawnPoint == null)
    {
        Debug.LogError("BulletSpawnPoint not found.");
    }
    shootCooldown = Time.time;

    targetEnemyTransform = targetEnemy.GetComponent<Transform>();
}

public override void OnEpisodeBegin()
{
    targetHit = false;
    numberOfShots = 0;
     startTime = Time.time;
    float randomY = Random.Range(-19f, 24f);
    float randomX = Random.Range(-42f, 42f);
    this.transform.localPosition = new Vector2(randomX, randomY);
}

void FixedUpdate()
{
    updatedEnemyPosition = targetEnemyTransform.localPosition;
    foreach (var bullet in bullets.ToArray())
        {
            bulletScript = bullet.GetComponent<BulletBehavior>();
            if(!bulletScript.rewardAssessed)
            {
            bulletScript.AssessReward();
            }
        }
    Vector2 Vel = transform.right * (movX * acceleration);
    rb.AddForce(Vel);
    float Dir = Vector2.Dot(rb.velocity, rb.GetRelativeVector(Vector2.right));
```

```
        if (acceleration > 0)
        {
            if (Dir > 0)
            {
                rb.rotation += movY * rotationControl * (rb.velocity.magnitude / speed);
            }
            else
            {
                rb.rotation -= movY * rotationControl * (rb.velocity.magnitude / speed);
            }
        }

        float thrustForce = Vector2.Dot(rb.velocity, rb.GetRelativeVector(Vector2.down))
* 2.0f;
        Vector2 relForce = Vector2.up * thrustForce;
        rb.AddForce(rb.GetRelativeVector(relForce));
        if (rb.velocity.magnitude > speed)
        {
            rb.velocity = rb.velocity.normalized * speed;
        }
        else if (rb.velocity.x > 0.01f && isFlipped)
        {
            transform.Rotate(180, 0, 0);
            isFlipped = false;
        }
        else if (rb.velocity.x < -0.01f && !isFlipped)
        {
            transform.Rotate(180, 0, 0);
            isFlipped = true;
        }
    }


    public override void CollectObservations(VectorSensor sensor)
    {
        sensor.AddObservation(targetHit ? 1.0f : 0.0f);
        int observedBullets = 0;
        foreach (var bullet in bullets)
        {
            if (observedBullets >= 10) break;
            sensor.AddObservation(bullet.transform.localPosition.x);
            sensor.AddObservation(bullet.transform.localPosition.y);
            observedBullets++;
        }
        for (int i = observedBullets; i < 10; i++)
        {
            sensor.AddObservation(0f);
            sensor.AddObservation(0f);
```

```
        }
        sensor.AddObservation(transform.localPosition.x);
        sensor.AddObservation(transform.localPosition.y);
        if (updatedEnemyPosition != null)
        {
            sensor.AddObservation(updatedEnemyPosition.x);
            sensor.AddObservation(updatedEnemyPosition.y);
        }
        else
        {
            sensor.AddObservation(0f);
            sensor.AddObservation(0f);
        }

        sensor.AddObservation(rb.velocity.x);
        sensor.AddObservation(rb.velocity.y);

        Vector2 forwardDir = new Vector2(transform.up.x, transform.up.y).normalized;
        sensor.AddObservation(forwardDir.x);
        sensor.AddObservation(forwardDir.y);
    }


    public override void OnActionReceived(ActionBuffers actions)
    {
        movY = actions.ContinuousActions[0];
        bool shoot = actions.DiscreteActions[0] == 1;
        if (shoot && Time.time >= shootCooldown)
        {
            Shoot();
        }
    }

    public override void Heuristic(in ActionBuffers actionsOut)
    {
        var continuousActionsOut = actionsOut.ContinuousActions;
        var discreteActionsOut = actionsOut.DiscreteActions;
        continuousActionsOut[0] = Input.GetKey(KeyCode.UpArrow) ? 1.0f : Input.Ge-
tKey(KeyCode.DownArrow) ? -1.0f : 0.0f;
        discreteActionsOut[0] = Input.GetKey(KeyCode.Space) ? 1 : 0;
    }

    public void Shoot()
    {
        if (bulletPrefab != null && bulletSpawnPoint != null && Time.time >= shootCool-
down)
        {
```

```
            newBullet = Instantiate(bulletPrefab, bulletSpawnPoint.position, bulletSpa-
wnPoint.rotation);
            newBullet.GetComponent<Rigidbody2D>().velocity = bulletSpeed * bulletSpa-
wnPoint.up;
            BulletBehavior bulletScript = newBullet.AddComponent<BulletBehavior>();
            bulletScript.agentXX = this;
            bulletScript.rewardAssessed = false;
            bulletScript.targetEnemyTransform = targetEnemyTransform;
            bullets.Add(newBullet);
            Destroy(newBullet, 1.5f);
            float currentReward = GetCumulativeReward();
            if (currentReward >= -0.9f)
            {
                AddReward(-0.01f);
                numberOfShots++;
            }
            else if (currentReward <= -0.9f) { numberOfShots++; }
            shootCooldown = Time.time + shootingRate;
        }
    }

    public void RemoveBullet(GameObject bullet)
    {
        bullets.Remove(bullet);
    }

    public void CompleteTask()
    {
        float timeTaken = Time.time - startTime;
        float reward = (1 - Mathf.Clamp01(timeTaken / maxTime)) * maxReward;
        Debug.Log($"EA hit really early with a bonus reward of {reward} in time of {ti-
meTaken} seconds");
        AddReward(reward);
        EndEpisode();
    }

    public void FlipPlane()
    {
        transform.Rotate(180, 0, 0);
        if (isFlipped == false) { isFlipped = true; }
        else isFlipped = false;
    }


    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Ceiling"))
         {
```

```
            transform.Rotate(180, 180, 0);
        }
        else if (collision.gameObject.CompareTag("Ground"))
        {
            AddReward(-1f);
            EndEpisode();
        }
        else if (collision.gameObject.CompareTag("Bullet"))
        {
            SetReward(-1f);
            EndEpisode();
        }
    }

    public void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("RightWall"))
        {
            this.transform.localPosition = new Vector2(transform.localPosition.x - 110,
transform.localPosition.y);
        }
        else if (other.CompareTag("LeftWall"))
        {
            this.transform.localPosition = new Vector2(transform.localPosition.x + 110,
transform.localPosition.y);
        }
    }

    public void TargetHit()

    {
        targetHit = true; AddReward(1f);
        float currentReward = GetCumulativeReward();
        Debug.Log(string.Format("Enemy agent has shot the enemy with the {0}th bullet
with the final reward of {1}", numberOfShots, currentReward));
        CompleteTask();
        Debug.Log("Target hit by enemy Agent");
    }

    public class BulletBehavior : MonoBehaviour
    {
        public AgentXXEnemy agentXX;
        public Transform targetEnemyTransform;
        public bool rewardAssessed = false;
        private float maxDistanceForReward = 5.0f;
        private float maxHitReward = 0.05f;
        private float maxNearMissReward = 0f;
```

```
        public void OnDestroy()
        {
            if (!rewardAssessed)
            {
                AssessReward();
            }
            agentXX.RemoveBullet(gameObject);
        }

    public void AssessReward()
    {
        float currentReward = agentXX.GetCumulativeReward();
        float bulletDistance = Vector2.Distance(transform.localPosition, targetEne-
myTransform.localPosition);
        Vector2 directionToTarget = (targetEnemyTransform.localPosition - transform.lo-
calPosition).normalized;
        Vector2 bulletDirection = GetComponent<Rigidbody2D>().velocity.normalized;
        float directionDifference = Vector2.Angle(directionToTarget, bulletDirection);

        if (bulletDistance <= maxDistanceForReward)
        {
            float reward = Mathf.Lerp(maxHitReward, maxNearMissReward, bulletDistance /
maxDistanceForReward);
            agentXX.AddReward(reward);
            Debug.Log($"EA's {agentXX.numberOfShots}th bullet close for reward: {re-
ward}");
            rewardAssessed = true;
        }
       else if (directionDifference > 90 && (currentReward > -0.9f))
        {
            agentXX.AddReward(-0.015f);
           // Debug.Log($"EA shot with {agentXX.numberOfShots}th bullet significantly
off-course for -15 points");
            rewardAssessed = true;
        }
        //else if (directionDifference > 90 && (currentReward <= -900)){Debug.Log($"EA
shot with {agentXX.numberOfShots}th bullet out off-course and with no other pe-
nalty");  }
        }

        private void OnTriggerEnter2D(Collider2D other)
        {
            //Debug.Log($"Bullet collided with: {other.gameObject.name}");
            if (other.CompareTag("AgentEnemy"))
            {
                Destroy(gameObject); // Destroy the bullet on hit
                agentXX.TargetHit();
            }
```

```
        }
    }
}
```

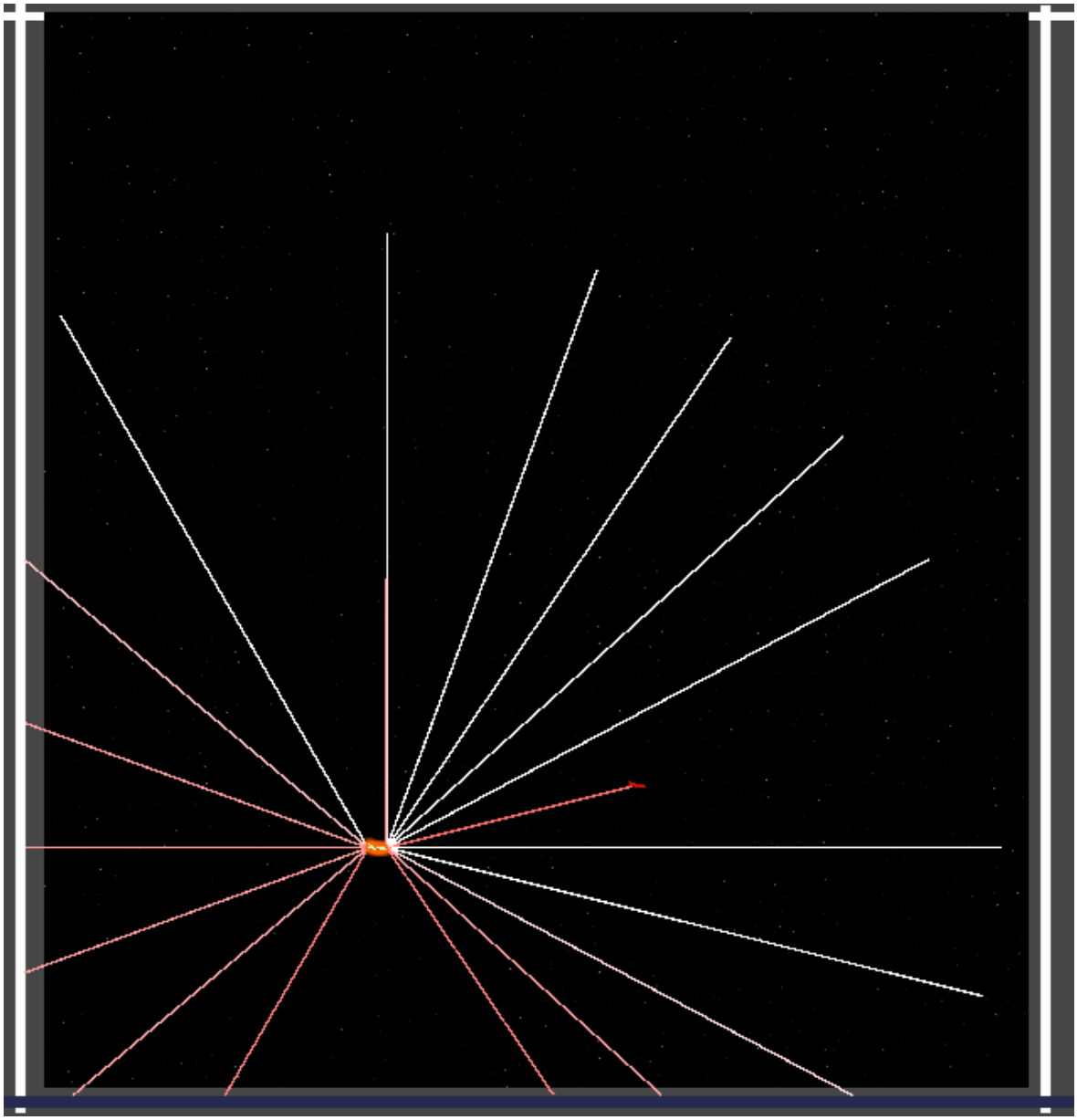
Pro ladění scriptu bylo hojně používáno vypisování do konsole pomocí *Debug.Log()*. Hlavní změnou je, že se agent převrací automaticky. Kdykoliv letí jedním nebo druhým směrem, překlopí se tak, aby letělo letadlo správně, a ne se spodní části letadla nahoru. Zajímavou funkcionalitou je snímání vystřelených střel, které slouží k lepšímu pochopení odměn.

Jak jde vidět, agentův skript se značně liší a to nejen tím, že neobsahuje metodu *Start()* a *Update()*, jako všechny *Monobehavior třídy*, ale naopak tyto: *Initialize(), OnEpisodeBegin(), CollectObservations(), OnActionsReceived a Heuristic()*.

5.3.1.1 Vysvětlení metod

- **Initialize** je iniciační metoda pro nastavení základních komponent, nahrazuje zde metodu Start.
- **OnEpisodeBegin** je funkce, ve které se definuje, co se stane při začátku každého cyklu. Trénování probíhá v určitých trajektoriích neboli v epizodách. Zde bylo potřeba nastavit, kde a jak se agent a cíl (target) zrodí, atd. Každá epizoda nabízí agentovi novou příležitost k interakci s prostředím a tím jeho iterativní zlepšování.
- **CollectObservations** je velice různorodá metoda pro všechny možné případy. Slouží k tomu, aby agent získal nějaké informace o prostředí, ve kterém se nachází. Zde byly použity metody pro zjištění polohy agenta, jeho cíle, rychlosti, směru a jiných různých akcí, jako je střelba a pozice jednotlivých nábojů. Tuto metodu lze vynechat nebo rozšířit pomocí komponenty RayPerception Sensor 2D/3D, kde se nastaví počet, rozpětí a délka paprsků, atd., které snímají prostředí. V této ukázce jsou použity obě metody, aby měl agent co nejvíce inputů a podle toho se mohl co nejlépe naučit požadovaný úkol. Zde se vycházelo z tutoriálu [30]. Zkrátka tato metoda i komponenta shromažďují data z prostředí, která jsou nezbytná pro rozhodování AI.
- **OnActionReceived** je metoda, kde se definuje, jak se vlastně agent může pohybovat (laicky řečeno, jaké tlačítka má mačkat). Pro tento případ je jedna spojitá akce, pohyb po ose y a druhá akce je diskrétní, tou je střelba. Agent zpočátku zkouší kombinaci všech akcí, aby získal, co největší odměnu.
- **Heuristic** je metoda, kde pouze definujeme, jak ovládat agenta manuálně. Toto slouží hlavně k testování prostředí, jestli všechno správně funguje, jestli se vše správně

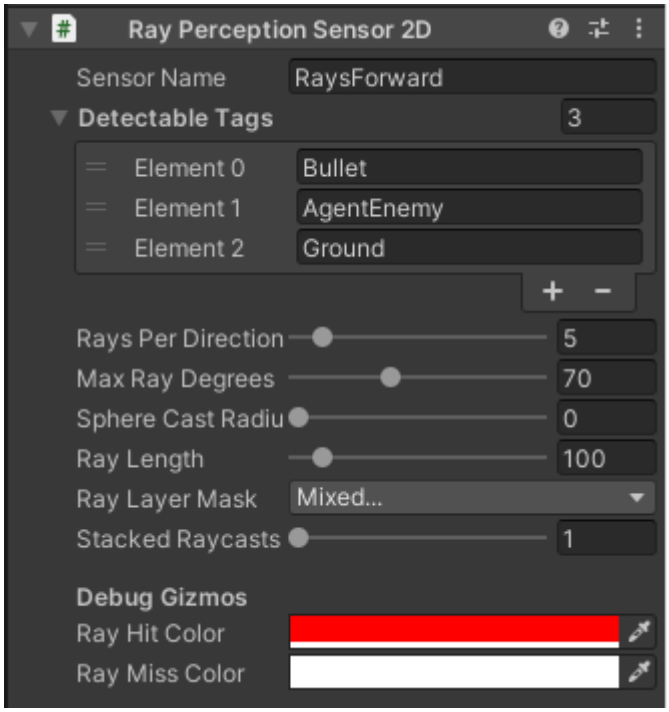
opětovně vytváří a při jakých akcích epizoda končí. Tato metoda slouží také k nahrávání demonstrací pro imitační učení, které je v této práci rovněž použito.



Obrázek 11 Využití Ray Perception Sensor 2D

Obrázek 11 ukazuje imlementaci komponenty Ray Perception Sensor 2D. Trénování je nastaveno tak, aby paprsky snímaly prostředí. Na obrázku označené letadlo (vlevo) snímá prostředí a je vidět, že jeho paprsky zachytily druhé letadlo.

Obrázek 12 ukazuje, jak se tyto paprsky nastavují.



Obrázek 12 Komponenta Ray Perception Sensor 2D

5.4 Průběh trénování

Připravení prostředí znamená připravit všechny skripty, game objekty na spravné pozice, přiřadit odměny pro naše posilovací učení a nastavit konfigurační soubor *name.yaml*.

Když je připravené prostředí, může se začít trénovat. Trénink se spouští pomocí příkazového řádku, kde je také možno sledovat průběžné učení. V příkazu je nutné zadat ID nového tréninku a cestu ke konfiguračnímu souboru s hyperparametry.

5.4.1 Hyperparametry pro trénování

Je doporučeno pro začátek přepsat základní konfigurační soubor z oficiální stránky ML-Agents a ten postupně upravovat. Zde je zmíněný konfigurační soubor.

```yaml
behaviors:
  RollerBall:
    trainer_type: ppo
    hyperparameters:
      batch_size: 10
      buffer_size: 100
      learning_rate: 3.0e-4
      beta: 5.0e-4
      epsilon: 0.2
      lambd: 0.99
      num_epoch: 3
      learning_rate_schedule: linear
      beta_schedule: constant
      epsilon_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 128
      num_layers: 2
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    max_steps: 500000
    time_horizon: 64
    summary_freq: 10000
```

Obrázek 13 Vzorový konfigurační soubor z oficiální dokumentace [28]

5.4.1.1 Vysvětlení všech základních hyperparametrů

- *trainer_type:* Zde je nutné vybrat, který algoritmus uživatel chce použít. Na výběr je z PPO, SAC a MA-POCA. Je doporučené pro začátek používat PPO.
- *batch_size*: Je počet zkušeností, použitých pro jednu iteraci aktualizace gradient descent. Tato hodnota by vždy měla být zlomkem buffer_size. V případě použití spojitého prostoru akcí by měla být tato hodnota velká (v řádu tisíců). V případě použití diskrétního prostoru akcí by měla být tato hodnota menší (v řádu desítek). Typický rozsah (spojité): 512 – 5120. Typický rozsah (diskrétní): 32 – 512. [27]
- *learning_rate:* Odpovídá síle každého kroku aktualizace gradient descent. Tato hodnota by měla být typicky snížena, pokud je trénink nestabilní a odměna neustále neroste. Typický rozsah: 1e-5 - 1e-3.
- *buffer size*: Odpovídá počtu zkušeností (pozorování agenta, akce a získané odměny), které se shromáždí před jakýmkoli učením nebo aktualizací modelu. Tato hodnota by

měla být násobkem batch_size. Obecně platí, že větší velikost bufferu vede k stabilnějším aktualizacím tréninku. Typický rozsah: 2048 – 409600. [28]

- *beta*: Je to míra náhodnosti, velice důležitý parametr. Čím je větší tato hodnota, tím větší je schopnost agenta objevovat.
- *epsilon*: Podobný parametr, který říká, jak velká je náhodnost chování v rozmezí <0, 1>. Při nule je vše plně deterministické.
- *labmd*: Regularizační parametr použitý při počítaní GAE. Znamená, do jaké míry se agent spoléhá na aktuální odhady hodnot. Typický rozsah: 0.9 - 0.95. [28]
- *num_epoch*: Počet projití skrze buffer za běhu Gradient Descent Optimization. Čím menší hodnota, tím stabilnější aktualizace, avšak za cenu rychlosti tréninku. Typický ozsah: 3 – 10. [27]
- *hidden_units*: Odpovídá počtu jednotek v každé plně propojené vrstvě neuronové sítě. U jednoduchých problémů, kde je správná akce přímočarou kombinací vstupních pozorování, by mělo být toto číslo malé. U problémů, kde je akce velmi komplexní interakcí mezi pozorovacími proměnnými, by mělo být toto číslo větší. Typický rozsah: 32 – 512. [28]
- *num_layers*: Odpovídá počtu skrytých vrstev přítomných po vstupu pozorování nebo po kódování CNN vizuálního pozorování. U jednoduchých problémů je pravděpodobnější, že méně vrstev se naučí rychleji a efektivněji. Pro složitější problémy řízení může být nutné použít více vrstev. Typický rozsah je 1 – 3. [28]
- *time_horizon:* Odpovídá počtu kroků zkušenosti, které se mají shromáždit na agenta před přidáním do bufferu zkušeností. Pokud je tento limit dosažen před koncem epizody, použije se odhad hodnoty k předpovědi celkové očekávané odměny ze současného stavu agenta. Typický rozsah je 32 – 2048. [30]
- *gamma*: Odpovídá diskontnímu faktoru pro budoucí odměny. Lze to chápat jako míru, jak daleko do budoucnosti by měl agent dbát na možné odměny. Pokud by měl agent jednat v současnosti, aby se připravil na odměny v daleké budoucnosti, měla by být tato hodnota vysoká. V případě, že jsou odměny bezprostřední, může být nižší. Typický rozsah je 0.8 – 0.995. [28]
- *normalize*: Odpovídá tomu, zda je aplikována normalizace na vstupní vektorová pozorování. Normalizace může být užitečná v případech s komplexními problémy spojitého řízení, ale může být škodlivá u jednodušších problémů diskrétního řízení. [28]
- *max_steps*: Odpovídá tomu, na jakém kroku se trénování zastaví.

5.4.2 Trénování PPO - agent sestřeluje nepohybující se cíl

Klíčový úkol v tento moment bylo vytvořit objekt, který by se hýbal a který by sloužil jako pohybující se cíl pro učení agenta. Tento pohyb se mohl napevno zakódovat, avšak v tomto případě stačilo naučit agenta se pohybovat a navigovat prostředím a vložit ho jako pohybující se cíl. Později sloužil jako protivník při „Self-play“, viz dále v textu.

Bylo potřeba nachystal nepohybující se cíl, který se náhodně „spawnoval“ po celém prostředí a agent měl za úkol tento cíl trefovat. Nejprve měl cíl větší rozměr, poté se postupně zmenšoval, aby se podmínky blížily koncové fázi nastavení prostředí.

Obrázek 14 Agent a cíl

Zde je skript pro opětovné vytvoření cíle:

```
public class TargetSpawning : MonoBehaviour
{
    public GameObject targetPrefab;
    private GameObject currentTarget;

    public void SpawnTarget()
    {
        if (currentTarget != null)
        {
            Destroy(currentTarget);
        }
        Vector2 spawnPos = new Vector2(Random.Range(-6, 80), Random.Range(-10, 34));
        currentTarget = Instantiate(targetPrefab, spawnPos, Quaternion.identity);
    }
}
```

V metodě SpawnTarget je logika opětovného vytvoření cíle

Nejprve bylo použito standardní RL, kde byly tyto odměny:

+1000 bodů za trefení cíle

-500 bodů při naražení do země

-500 bodů při naražení na kterýkoliv okraj

+0.1 za každý výstřel

Nutno dodat, že v této fázi vývoje nebyly okraje koncově nastavené, proto prostředí nemělo ideální podmínky. Nejprve našel agent semi-optimální řešení točit se pouze dokola a střílet všemi směry. Bylo tedy nutné přidat novou časovou odměnu, která měla maximální hodnotu 1000 a klesala na 0 po dobu 60 sekund. Po tomto zásahu se agent začal správně učit.

Byly použity následující hyperparametry:

```
trainer_type:   ppo
hyperparameters:
  batch_size:   1000
  buffer_size:  100000
  learning_rate:    0.0003
  beta: 0.0005
  epsilon:  0.1
  lambd:    0.95
  num_epoch:    3
  shared_critic:    False
  learning_rate_schedule:   linear
  beta_schedule:    constant
  epsilon_schedule: linear
network_settings:
  normalize:    False
  hidden_units: 128
  num_layers:   3
  vis_encode_type:  simple
  memory:   None
  goal_conditioning_type:   hyper
  deterministic:    False
reward_signals:
  extrinsic:
    gamma:  0.99
    strength:   1.0
    network_settings:
      normalize:    False
      hidden_units: 128
      num_layers:   2
      vis_encode_type:  simple
      memory:   None
      goal_conditioning_type:   hyper
      deterministic:    False
init_path:  None
keep_checkpoints:   5
checkpoint_interval:    500000
max_steps:  500000
time_horizon:   64
summary_freq:   10000
```

```
threaded:   False
self_play:  None
behavioral_cloning: None
```

Dále byla věnována pozornost RL ve spojení s imitačním učení. Toto učení prokázalo daleko lepší výsledky. Zde byly nastaveny tyto parametry:

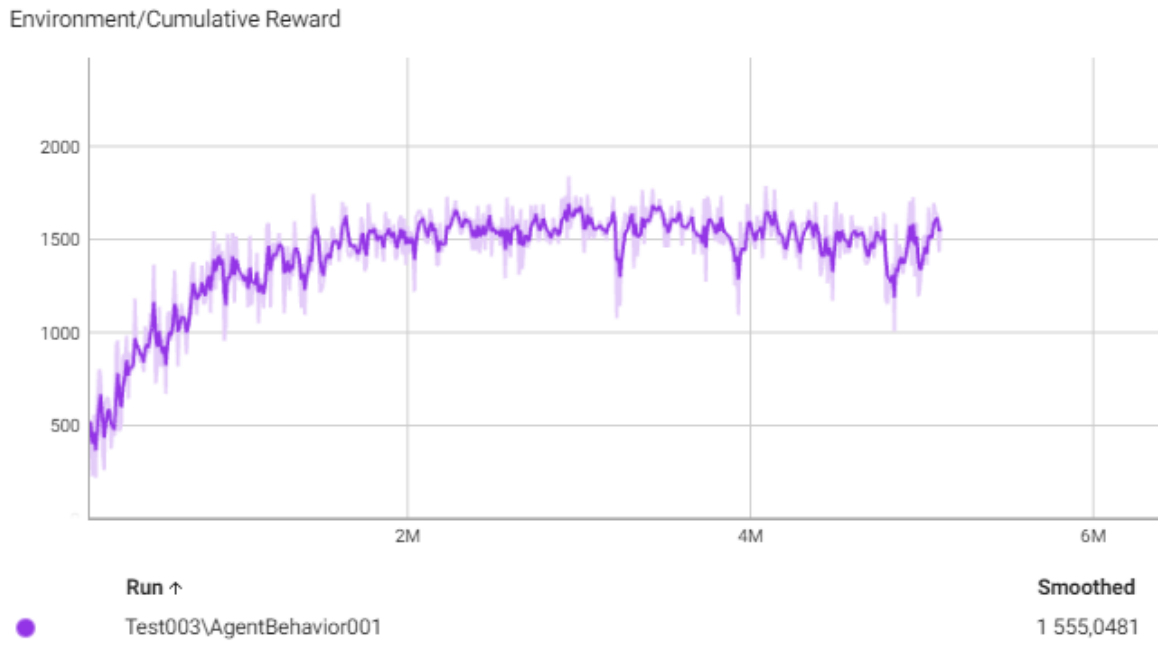
```
trainer_type:   ppo

hyperparameters:
  batch_size:   1024
  buffer_size:  10240
  learning_rate:    0.0003
  beta: 0.0005
  epsilon:  0.2
  lambd:    0.99
  num_epoch:    3
  shared_critic:    False
  learning_rate_schedule:   linear
  beta_schedule:    constant
  epsilon_schedule: linear
network_settings:
  normalize:    False
  hidden_units: 128
  num_layers:   2
  vis_encode_type:  simple
  memory:   None
  goal_conditioning_type:   hyper
  deterministic:    False
reward_signals:
  extrinsic:
    gamma:  0.9
    strength:   1.0
    network_settings:
      normalize:    False
      hidden_units: 128
      num_layers:   2
      vis_encode_type:  simple
      memory:   None
      goal_conditioning_type:   hyper
      deterministic:    False
  gail:
    gamma:  0.99
    strength:   0.9
    network_settings:
      normalize:    False
      hidden_units: 128
      num_layers:   2
      vis_encode_type:  simple
      memory:   None
      goal_conditioning_type:   hyper
      deterministic:    False
    learning_rate:  0.0003
    encoding_size:  None
    use_actions:    False
    use_vail:   False
```

```
demo_path:  A:\UnityProjects\Airplane001\Demo\Demonstration001.demo
```

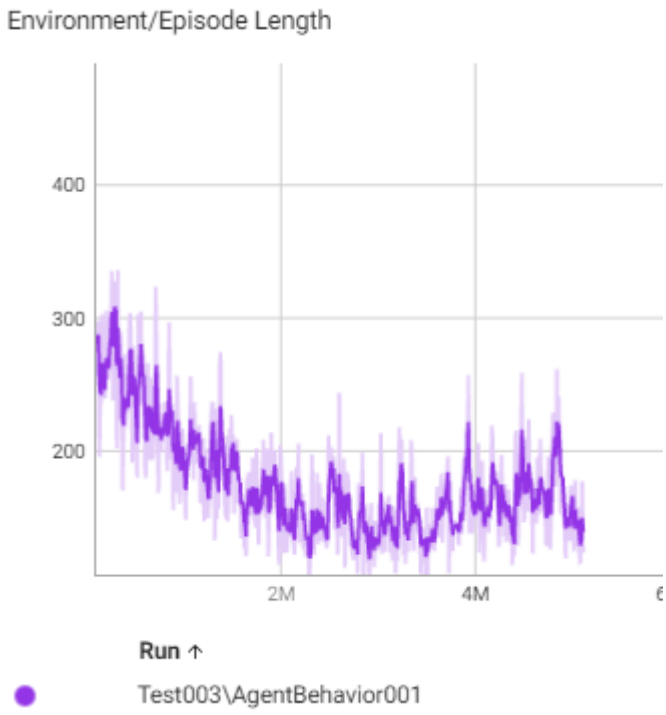
Zde bylo potřeba nahrát demonstrační ukázku, která se provedla v Unity editoru. Posloužila opět *metoda Heuristic*, která v sobě má manuální ovládání agenta pomocí kláves pro právě tyto účely nebo testování prostředí.

Na obrázcích je znázorněn průběh tohoto tréninku.



Obrázek 15 Průběh učení

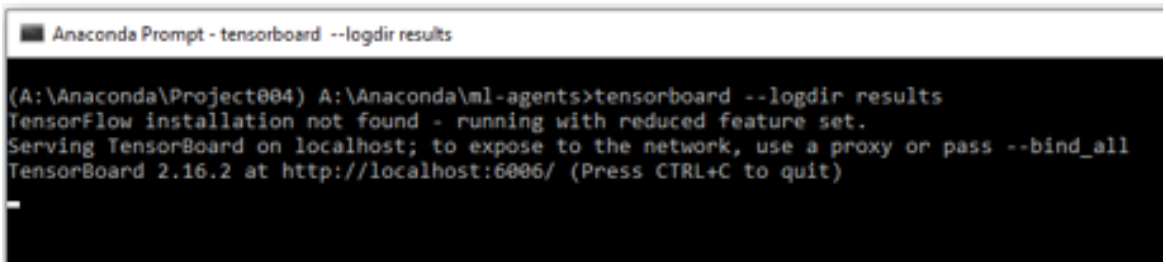
Obrázek ukazuje postupný zisk odměn u tohoto učení.



Obrázek 16 Doba epizody

Obrázek ukazuje, že se délka epizody postupně snižuje, protože agent potřebuje menší čas k tomu, aby splnil úkol.

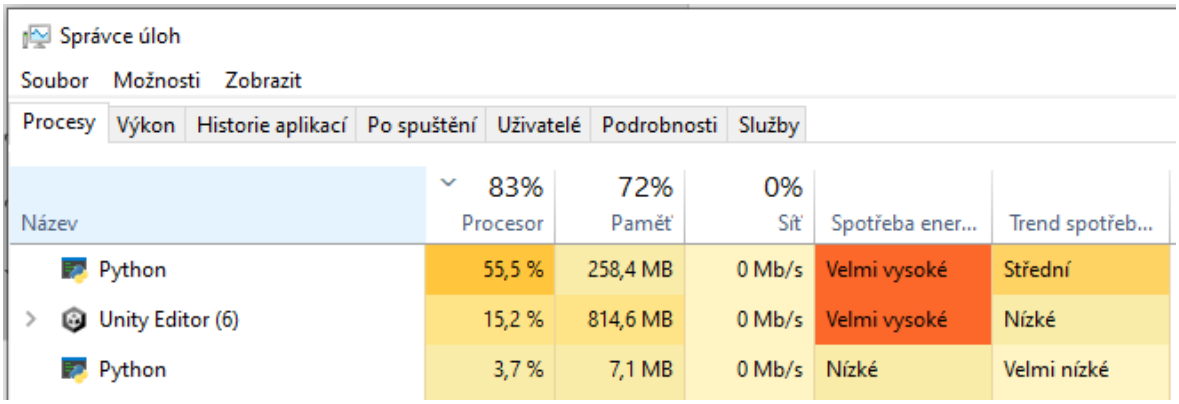
K obrázkům výsledků učení lze přistoupit přes příkaz v Anacondě nebo venv.



Obrázek 17 Spuštění TensorBoard

Nejprve byl k učení použit velký cíl a ten se poté zmenšil, aby se Agent připravil pro náročnější nadcházející tréninky.

Trénování samotné může být značně urychleno a to tím, že se zkopíruje dané prostředí a vloží několikrát do scény. Avšak tato možnost nebyla v tomto projektu možná, protože pro potřeby práce nebyl použit dostatečně výkonný hardware.



Obrázek 18 Výpočetní náročnost trénování jednoho 2D prostředí

5.4.3 Trénování PPO – agent proti agentovi (Self-play)

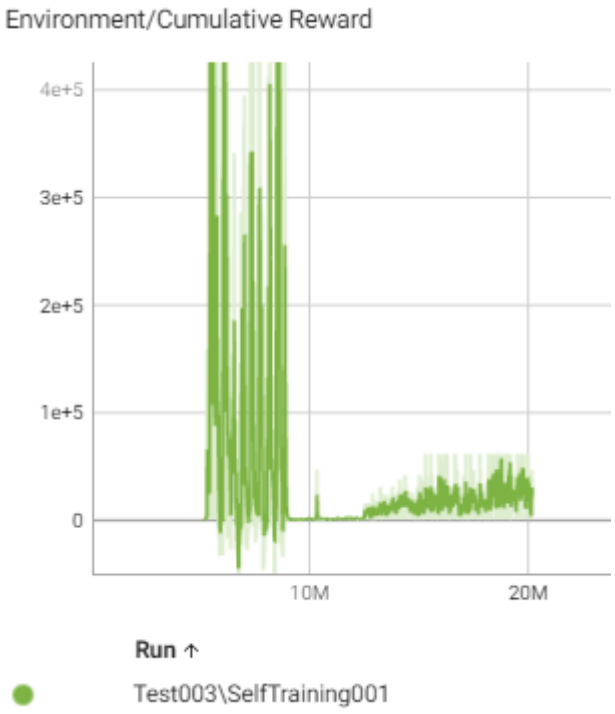
Zde se začalo s následujícími parametry:

```
trainer_type:   ppo
hyperparameters:
  batch_size:   2048
  buffer_size:  20480
  learning_rate:    0.0003
  beta: 0.0005
  epsilon:  0.2
  lambd:    0.99
  num_epoch:    3
  shared_critic:    False
  learning_rate_schedule:   linear
  beta_schedule:    constant
  epsilon_schedule: linear
network_settings:
  normalize:    False
  hidden_units: 128
  num_layers:   2
  vis_encode_type:  simple
  memory:   None
  goal_conditioning_type:   hyper
  deterministic:    False
reward_signals:
  extrinsic:
    gamma:  0.9
    strength:   1.0
    network_settings:
      normalize:    False
      hidden_units: 128
      num_layers:   2
      vis_encode_type:  simple
      memory:   None
      goal_conditioning_type:   hyper
      deterministic:    False
init_path:  None
keep_checkpoints:   5
checkpoint_interval:    500000
max_steps:  50000000000
```

```
time_horizon:   500
summary_freq:   10000
threaded:   False
self_play:
  save_steps:   50000
  team_change:  200000
  swap_steps:   2000
  window:   10
  play_against_latest_model_ratio:  0.5
  initial_elo:  1200.0
behavioral_cloning: None
```

Pozor na team ID u agenta. Pokud chceme, aby agenti soupeřili, nesmí mít ID totožné.

Zde je graf, jak bylo učení ve skutečnosti úspěšné.



Obrázek 19 Self-play

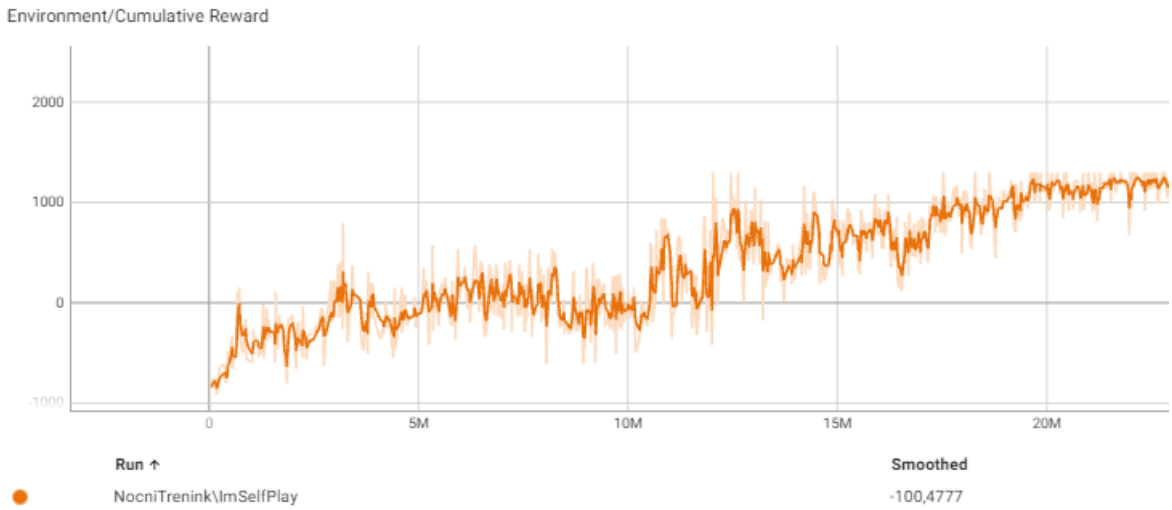
Toto učení bylo spuštěno z předchozího natrénovaného modelu z kroku cca 5 milionů. Je vidět, že nepřineslo výrazný úspěch, protože agent by se musel učit neskutečně dlouho, aby se dostavily očekávané výsledky.

5.4.4 Trénování PPO – Self-play, imitační učení

V této fázi vývoje už bylo přenastaveno prostředí pro lepší podmínky. Byla změněna logika pravého a levého okraje tak, aby se agent mohl tzv. teleportovat z jedné strany na druhou. Dále byly odstraněny odměny za střílení, a naopak přidány penalizace za každý výstřel a za každou střelu vystřelenou více jak 90 stupňů vedle od nepřítele.

Toto trénování vypadalo doposud nejlépe. Ukázalo se, že imitační učení značně zefektivnilo trénink. Agent konečně začal trefovat cíl alespoň trochu, jak by měl. Samozřejmě bylo potřeba ladit hyperparametry, protože odměny zde byly postupně přenastaveny a laděny.

Zde je zobrazen růst odměn u tohoto učení:



Obrázek 20 Self-play + imitační učení

Hyperparametry tohoto tréninku:

```
trainer_type:   ppo
hyperparameters:
  batch_size:   2048
  buffer_size:  20480
  learning_rate:    0.0003
  beta: 0.005
  epsilon:  0.2
  lambd:    0.99
  num_epoch:    5
  shared_critic:    False
  learning_rate_schedule:   linear
  beta_schedule:    constant
  epsilon_schedule: linear
network_settings:
  normalize:    True
  hidden_units: 256
  num_layers:   3
  vis_encode_type:  simple
  goal_conditioning_type:   hyper
  deterministic:    False
reward_signals:
  extrinsic:
    gamma:  0.9
    strength:   1.0
    network_settings:
      normalize:    False
      hidden_units: 128
      num_layers:   2
      vis_encode_type:  simple
```

```
      goal_conditioning_type:   hyper
      deterministic:    False
  gail:
    gamma:  0.99
    strength:   0.4
    network_settings:
      normalize:    False
      hidden_units: 128
      num_layers:   2
      vis_encode_type:  simple
      memory:   None
      goal_conditioning_type:   hyper
      deterministic:    False
    learning_rate:  0.0003
    encoding_size:  None
    use_actions:    False
    use_vail:   False
    demo_path:  A:\UnityProjects\Airplane001\Demos\DNT007.demo
init_path:  None
keep_checkpoints:   5
checkpoint_interval:    500000
max_steps:  50000000000
time_horizon:   1000
summary_freq:   10000
threaded:   False
self_play:
  save_steps:   50000
  team_change:  50000
  swap_steps:   5000
  window:   50
  play_against_latest_model_ratio:  0.5
  initial_elo:  1200.0
behavioral_cloning:
  demo_path:    A:\UnityProjects\Airplane001\Demos\DNT007.demo
  steps:    10000
  strength: 0.2
  samples_per_update:   256
  num_epoch:    3
  batch_size:   2048
```

Pozor, ne všechny hyperparametry lze v průběhu tréninku měnit. Například jednou při trénování bylo přenastaveno number of hidden units z 128 na 256 a celý trénink se zhroutil a nešel opravit. Dalo se začít od určitého kroku znovu, avšak přechod na 256 byl nutnou zkouškou trénování.

5.4.5 PPO vs MA-POCA

Protože při učení self-play hrál agent jen proti verzím sebe samého, byl tento trénink naopak velice dobrý pro generalizaci pohybů, agent se stal adaptivnějším. Ukázalo se, že mnohem úspěšnější bylo PPO, avšak MA-POCA je navržen pro multiagentní prostředí.

6 DOKONČENÍ HRY

Pro dokončení hry bylo nutné upravit skript pro hráče, vymazat všechna trénovací prostředí kromě jednoho, ve kterém by běžela celá hra. Existuje více možností pro aplikování funkcionality výběru hry s jedním z více modelů. V této práci to bylo implementováno tak, že se přidal skript pro aktivaci a deaktivaci jednotlivých game objektů v určitých situacích a na klikání jednotlivých tlačítek.

Zde je ukázka, jak byla tahle logika aplikována:

```
public class GameManager : MonoBehaviour
{
    public GameObject startButton1;
    public GameObject startButton2;
    public GameObject startButton3;
    public GameObject player;
    public GameObject agent1;
    public GameObject agent2;
    public GameObject agent3;


    void Start()
    {
        player.SetActive(false);
        agent1.SetActive(false);
        agent2.SetActive(false);
        agent3.SetActive(false);

    }

    void Update()
    {

    }

    public void GameStart1()
    {
        agent1.SetActive(true);
        player.SetActive(true);

        startButton1.SetActive(false);
        startButton2.SetActive(false);
        startButton3.SetActive(false);
    }
    public void GameStart2()
    {
        agent2.SetActive(true);
```

```
        player.SetActive(true);

        startButton1.SetActive(false);
        startButton2.SetActive(false);
        startButton3.SetActive(false);
    }

    public void GameStart3()
    {
        agent3.SetActive(true);
        player.SetActive(true);

        startButton1.SetActive(false);
        startButton2.SetActive(false);
        startButton3.SetActive(false);
    }

}
```

Poté bylo potřeba vyřešit skórování a to se naopak provedlo ve skriptu agenta. Bylo implementováno ještě tlačítko pro pozastavení hry, upraven design uživatelského rozhraní, atd. V poslední fázi byl doplněn herní panel konce hry s tlačítkem na repetici hry.

Jako poslední bylo potřeba hru sestavit (build). To lze v Unity provést snadno a to tak, že se vyberou parametry, jaké má hra mít. V tomto případě je nutné hlavně zvolit možnost, aby se hrálo na široké obrazovce bez možnosti přetočení.

7 VYHODNOCENÍ

Do hry se nakonec použily celkem tři modely, každý s trochu jiným tréninkem. Bylo provedeno testování funkčnosti a tyto modely jsou přijatelné pro herní požitek a ukázku, jak je strojové učení užitečné v mobilních i počítačových hrách nebo v různých simulacích.

Překvapující však bylo, jak moc dobrý hardware toto trénování potřebuje a jak dokonale musí být prostředí definováno. Hardware, na kterém byla tato ukázka prováděna, měl následující parametry:

- Procesor Intel core i7-7700 CPU 3.60 GHz
- Paměť RAM 16 GB
- Grafická karta NVIDIA GeForce GTX 1080

I tento, na dnešní poměry docela slušný hardware, nestačil k natrénování dokonalého AI (neporazitelný soupeř). K tomu by bylo potřeba upravit hyperparametry tak, aby se AI učilo pomaleji, ale stabilněji (zvýšit hodnoty u hidden units, batch size, buffer size atd.) a nechat ho trénovat několik desítek milionů kroků, nejlépe alespoň 200, což by pro použitý hardware trvalo pro jediné učení cca 40 dní čistého času. Pro každé prostředí a každé požadované chování a dostupnou délku učení je nutné ladit podmínky zvlášť.

Bylo provedeno testování všech tří algoritmů ML-Agents Toolkit (SAC, MA-POCA, PPO). Nejvíce se mi osvědčil algoritmus PPO. Bylo to možná také proto, že jsem se mu nejvíce věnoval.

Jak je vytvoření umělé inteligence ve hře nebo simulaci složité, popisuje i článek čtyř autorů Kaunaské technologické univerzity v Litvě [31], na který jsem narazil na konci své bakalářské práce. Autoři popisují podobnou problematiku, došli k podobným závěrům a měli podobné problémy s trénováním. Připadá mi, že se mi podařilo dosáhnout lepších výsledků, než popisují tito autoři ve své práci. I to dokazuje, jak těžké bylo dosáhnout stanovených cílů při vytváření dobrého agenta v komplikovaném prostředí.

ZÁVĚR

Tato práce se zabývá vytvořením umělé inteligence pro mobilní hru pomocí knihovny Unity ML-Agents. Cílem práce bylo nejen implementovat sofistikovaného AI protivníka, ale také prozkoumat demonstraci nejpoužívanějších technik v ML-Agents. V teoretické části byly rozebrány základy umělé inteligence a strojového učení s důrazem na reinforcement learning. Toto bylo klíčové pro pochopení metod užitých v praktické části.

V praktické části byla vytvořena hra v Unity, kde byly pomocí zmíněné knihovny vyvinuty a natrénovány herní modely, které byly následně použity pro hru.

Byla zdůrazněna důležitost správného nastavení hyperparametrů a odměn (tedy podmínky prostředí), protože to je to, co odlišuje dobrý model od špatného. Bylo také konstatováno, že tyto podmínky by se měly lišit podle možností hardwaru a tedy doby trénování.

Bakalářská práce přispívá k praktické aplikaci teoretických znalostí strojového učení a také otevírá dveře pro další rozvoj v oblasti vývoje her s využitím umělé inteligence. Rozšíření, zdokonalení a publikace aplikace by mohlo vést k širšímu uplatnění.

SEZNAM POUŽITÉ LITERATURY

[1] ENGELBRECHT, Dylan. *Introduction to Unity ML-Agents: Understand the Interplay of Neural Networks and Simulation Space Using the Unity ML-Agents Package*. Apress, 2023 ISBN 978-1484289976.

[2] GOODFELLOW, Ian; BENGIO, Yoshua a COURVILLE, Aaron. *Deep Learning*. Online. MIT Press, 2016. Dostupné z: www.deeplearningbook.org. [cit. 2024-04-16].

[3] LANHAM, Micheal. *Learn Unity ML-Agents – Fundamentals of Unity Machine Learning: Incorporate new powerful ML algorithms such as Deep Reinforcement Learning for games.* Packt Publishing, 2018. ISBN 978-1789138139.

[4] AVERSSA, Davide. *Unity Artificial Intelligence Programming*. 5th edition. Birmingham: Packt Publishing, 2022. ISBN 978-1-80323-853-1.

[5] HALEV-SHWARTZ, Shai; BEN-DAVID, Shai. *Understanding Machine Learning: From Theory to Algorithms.* Cambridge University Press, 2014. ISBN 978-1-107-05713-5. Dostupné z: https://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/understanding-machine-learning-theory-algorithms.pdf.

[6] GANESAN, Veeramani. Machine learning in mobile applications. Online. *International Journal of Computer Science and Mobile Computing*. 2022, roč. 11, č. 2, s. 110 - 118. ISSN 2320–088X. Dostupné z: https://doi.org/10.47760/ijcsmc.2022.v11i02.013. [cit. 2024-03-09].

[7] LANHAM, Micheal. *Hands-On Reinforcement Learning for Games*. PDF. 2020. Birmingham: Packt Publishing, 2020. ISBN ISBN 978-1-83921-493-6. Dostupné z: https://www.directtextbook.com/isbn/9781839214936. [cit. 2024-04-08].

[8] ACHIAM, Josh. *Part 1: Key Concepts in RL*. Online. Open AI Spinning up. 2018. Dostupné z: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html. [cit. 2024-04-10].

[9] WENG, Lilian. *A (Long) Peek into Reinforcement Learning*. Online. WENG, Lilian. Lil'log. 2018, 2020-09-03. Dostupné z: https://lilianweng.github.io/posts/2018-02-19-rl-overview/. [cit. 2024-04-10].

[10] ACHIAM, Josh. *Part 2: Kinds of RL Algorithms*. Online. OPEN AI. Open AI Spinning up. 2018. Dostupné z: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#id19. [cit. 2024-04-11].

[11] WENG, Lilian. *Policy Gradient Algorithms*. Online. Lil'log. 2018, 2020-10-15. Dostupné z: https://lilianweng.github.io/posts/2018-04-08-policy-gradient/. [cit. 2024-04-12].

[12] HSU, C. C.; MENDLER-DÜNNER, C. a HARDT, M. Revisiting Design Choices in Proximal Policy Optimization. Online. *ArXiv*. 2020, roč. 2009, č. 10897, s. 1 - 3. Dostupné z: https://doi.org/https://arxiv.org/abs/2009.10897. [cit. 2024-04-13].

[13] HAARNOJA, Tuomas; ZHOU, Aurick; ABBEEL, Pieter a LEVINE, Sergey. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. Online. *ArXiv*. 2018, roč. 1801, č. 01290. Dostupné z: https://doi.org/https://doi.org/10.48550/arXiv.1801.01290. [cit. 2024-04-15].

[14] COHEN, Andrew; TENG, Ervin; BERGES, Vincent-Pierre; DONG, Ruo-Ping; HENRY, Hunter et al. On the Use and Misuse of Absorbing States in Multi-agent Reinforcement Learning. Online. *ArXiv*. 2021, roč. 2111, č. 05992. Dostupné z: https://doi.org/https://doi.org/10.48550/arXiv.2111.05992. [cit. 2024-04-16].

[15] UNITY TEAM. *Unity ML-Agents Toolkit*. Online. Github. 2017, 2023-10-09. Dostupné z: https://github.com/Unity-Technologies/ml-agents. [cit. 2024-03-12].

[16] UNITY TEAM. *Unity ML-Agents Toolkit/docs*. Online. Github. 2017, 2023-10-09. Dostupné z: https://github.com/Unity-Technologies/ml-agents/blob/develop/docs. [cit. 2024-03-12].

[17] UNITY TEAM. *Agents*. Online. Github. 2018, 2018-03-14. Dostupné z: https://github.com/miyamotok0105/unity-ml-agents/blob/master/docs/Learning-Environment-Design-Agents.md. [cit. 2024-03-12].

[18] DUNG, Van Duc. Building machine learning bot with ML-Agents in Tank Battle. Online, bakalářská práce. Danang: FPT University, 2022. Dostupné z: http://ds.libol.fpt.edu.vn/bitstream/123456789/3327/2/Thesis%20-%20Building%20machine%20learning%20bot%20with%20Mlagents%20in%20Tank%20Battle.pdf. [cit. 2024-03-20].

[19] HALPERN, Jared. *Developing 2D games with Unity: indipendent game programming with C#*. New York: APress, 2024. ISBN 978-1484237717.

[20] CARDOSO*, Hugo. How to use Machine Learning AI in Unity! (ML-Agents). Online.* Youtube. 2020. Dostupné

z: https://www.youtube.com/watch?v=zPFU30tbyKs&list=PLzDRvYVwl53vehwiN_odYJkPBzcqFw110. [cit. 2024-02-11].

[21] CARDOSO, Hugo. *Teach your AI! Imitation Learning with Unity ML-Agents!*. Online. Youtube. 2020. Dostupné z: https://www.youtube.com/watch?v=supqT7kqpEI. [cit. 2024-02-12].

[22] SCHUCHMANN, Sebastian. *ML-Agents 1.0+ | Create your own A.I. | Full Walkthrough | Unity3D*. Online. Youtube. 2020. Dostupné z: https://www.youtube.com/watch?v=2Js4KiDwiyU&list=PLLNQAb-SFzRt9VJRTovXoQK_UEtACUjNB. [cit. 2024-02-12].

[23] How to build a mobile Airplane controller in Unity [2021]. Online. Youtube. 2021. Dostupné z: https://www.youtube.com/watch?v=DdH1BQRYgcU. [cit. 2024-02-14].

[24] SCHUCHMANN, Sebastian. *Unity ML-Agents 1.0+ - Self Play explained*. Online. Youtube. 2020. Dostupné z: https://www.youtube.com/watch?v=zAtcRbYdvuw&t=624s. [cit. 2024-21].

[25] DE BYL, Penny. *A Beginner's Guide To Machine Learning with Unity*. Online. Udemy. 2019. Dostupné z: https://www.udemy.com/course/machine-learning-with-unity/. [cit. 2024-03-01].

[26] UNITY TEAM. *Making a New Learning Environment.* Online. Github. 2017, 2023-10-09. Dostupné z: Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Create-New.md. [cit. 2024-03-20].

[27] UNITY TEAM. Training Configuration File. Online. Github. 2017, 2023-10-09. Dostupné z: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md#self-play. [cit. 2024-03-29].

[28] UNITY TECHNOLOGIES. *Training-PPO*. Online. Github. 2017. Dostupné z: https://github.com/gzrjzcx/ML-agents/blob/master/docs/Training-PPO.md. [cit. 2024-05-01].

[29] UNITY TECHNOLOGIES. *Joystick Pack*. Online. Unity Asset Store. 2019. Dostupné z: https://assetstore.unity.com/packages/tools/input-management/joystick-pack-107631. [cit. 2024-02-17].

[30] KELLY, Adam. *ML-Agents: Hummingbirds*. Online. UNITY TECHNOLOGIES. Https://learn.unity.com/. 2019, 2020-05-29. Dostupné z: https://learn.unity.com/project/scene-setup?uv=2019.3&courseId=5e470160edbc2a15578b13d7. [cit. 2024-04-29].

[31] SAVID, Yusef; MAHMOUDI, Reza; MASKELIŪNAS, Rytis a DAMAŠEVIČIUS, Robertas. Simulated Autonomous Driving Using Reinforcement Learning: A Comparative Study on Unity's ML-Agents Framework. Online. *Information*. 2023, roč. 14(5), č. 290, s. 1-22. Dostupné z: https://doi.org/https://doi.org/10.3390/info14050290. [cit. 2024-05-08].

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

ML Machine learning (strojové učení)

AI artificial intelligence (umělá inteligence)

RL reinforcement learning (posilované učení)

UI umělá inteligence

GAN Generative Adversrial Networks

PCG Procedural Content Generation

AR Augmented reality (rozšířená realita)

VR Virtual reality (virtuální realita)

MR Mixed reality (smíšená realita)

NPC non-playable character

BCI Brain Computer Interface

NLP Natural Language Processing

FSM Finite State Machines

MDP Markovský rozhodovací proces

NN Neural Network

SDK Software Development Kit

NDK Native Development Kit

PPO Proximal Policy Optimalization

SAC Soft Actor Critic

SEZNAM OBRÁZKŮ

SEZNAM PŘÍLOH

PŘÍLOHA P I - DVD S PRAKTICKOU ČÁSTÍ PRÁCE

Toto DVD obsahuje:

- Bakalářskou práci fulltext.pdf
- Build hry na mobil FF2D.apk
- Projekt v Unity projekt.zip