Analýza herní AI s implementací v herním enginu Unity

Martin Němeček

Bakalářská práce
2023

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

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

Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Martin Němeček**
Osobní číslo:	**A20444**
Studijní program:	**B0613A140020 Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Analýza herní AI s implementací v herním enginu Unity**
Téma práce anglicky:	**Game AI Analysis with Implementation in the Unity Game Engine**

Zásady pro vypracování

1. Vypracujte literární rešerši na dané téma.
2. Rámcově popište teoretické principy metod, které se používají pro herní AI.
3. Stručně představte několik algoritmů, které se pro herní AI využívají, přičemž dva vybrané, dále využité v praktické části práce, popište podrobněji.
4. Navrhněte a prakticky implementujte vhodné ukázky pro demonstraci funkce vybraných herních AI.
5. Pro tyto účely využijte herního enginu Unity.
6. V závěru kriticky zhodnoťte dosažené výsledky.

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

Seznam doporučené literatury:

1. BAYER, V. Umělá inteligence pro hraní her. Brno, 2017. Bakalářská práce, Fakulta informačních technologií, Vysoké učení technické v Brně.
2. RABIN, S. Game AI Pro 3: collected wisdom of game AI professionals. Boca Raton: Taylor & Francis, 2017. ISBN 978--1498742580.
3. CHURCHILL, D. Heuristic Search Techniques for Real-Time Strategy Games. Edmonton, 2016. Ph.D. thesis, Department of Computer Science, University of Alberta.
4. DAGRACA, M. Practical Game AI Programming. Birmingham: Packt Publishing, 2017. ISBN 978-1787122819.
5. SMITH, M., FERNS, S. Unity 2021 Cookbook. Birmingham: Packt Publishing, 2021. ISBN 978-1839217616.
6. Mark J. PRICE, M.J. C# 8.0 and .NET Core 3.0: Modern Cross-Platform Development. Birmingham: Packt Publishing, 2019. ISBN 978-1788478120.

Vedoucí bakalářské práce: **doc. Ing. František Gazdoš, Ph.D.**
Ústav řízení procesů

Datum zadání bakalářské práce: **2. prosince 2022**
Termín odevzdání bakalářské práce: **26. května 2023**

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 7. prosince 2022

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/a jsem seznámen/a 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

Martin Němeček v.r.
podpis studenta

ABSTRAKT

Předkládaná bakalářská práce „Analýza herní AI s implementací v herním enginu Unity“ se obecně zaměřuje na herní AI a dále na vzájemné porovnání konečného automatu a behaviorálního stromu pro tvorbu herní AI. Teoretická část představuje základní principy herní AI pokračuje uvedením nejčastěji používaných algoritmů a metod. Dále následuje kapitola s podrobnějším popisem konečného automatu a behaviorálního stromu. V další části je přehled využití herní AI ve vybraných populárních hrách. Poslední kapitola teoretické části pak stručně představuje Unity engine a programovací jazyk C# dále použité v praktické části této práce. Ta se zabývá vytvořením ukázkového prostředí, dále vlastním návrhem a implementací konečného automatu a behaviorálního stromu, které jsou v poslední části práce vzájemně porovnány.

Klíčová slova: herní AI, konečný automat, behaviorální strom, Unity engine

ABSTRACT

The present bachelor thesis "Game AI analysis with implementation in the Unity game engine" focuses on game AI in general and on the inter-comparison of a finite state machine and a behavioral tree algorithms for the creation of game AI. The theoretical part introduces the basic principles of game AI and goes on to list the most commonly used algorithms and methods. This is followed by a chapter with a more detailed description of the finite state machine and the behavioral tree. The next section provides an overview of the use of game AI in selected popular games. The last chapter of the theoretical part then briefly introduces the Unity engine and the C# programming language further used in the practical part of this thesis. Next section deals with the creation of a sample environment, then the actual design and implementation of the final state machine and behavioral tree, which are compared with each other in the last part of the thesis.

Keywords: game AI, finite state machine, behavioral tree, Unity engine

Tímto bych rád poděkoval vedoucímu této práce panu doc. Ing. Františkovi Gazdošovi, Ph.D. za odborné vedení, čas věnovaný mé práci a ochotu pomoci při vypracování bakalářské práce.

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

OBSAH

ÚVOD

Tato bakalářská práce se zabývá herní umělou inteligencí obecně a v praktické části pak porovnáním dvou často využívaných algoritmů. Nejdříve vymezím, co je herní umělá inteligence. Herní umělá inteligence (dále jen herní AI) se nezabývá řešením problémů v reálném životě, ale vytvořením iluze inteligence herních figur [2]. Tato iluze je úspěšná pouze v případě že postavy ovládané herní AI reagují na akce hráče nebo na herní prostor uvěřitelně a realisticky. Například tím, že jsou schopné navigovat okolo překážek mezi jejich původní pozicí a pozicí hráče, nebo tím, že reagují na akce hráče. Tímto herní AI pomáhají k vytvoření děje, který působí uvěřitelně.

Hlavním důvodem pro využívání herní AI ve video hrách je flexibilnější uskutečnění navrženého zážitku, kdy na rozdíl od jednoduchého skriptu bude herní AI schopna reagovat na chování hráče, které nebylo přímo předpokládané.

Nicméně herní AI neslouží jen k ovládání herních objektů. Také umožnuje vývojářům vytvářet náhodná herní prostředí, která budou generována herní AI při každém spuštění pomocí několika pravidel, a tím přispívá k opakovatelnému hraní dané hry. Dále se může využívat pro ovládání hudby jakou hráč momentálně slyší, anebo pro vytvoření počasí, které se bude moci měnit například mezi bouří a jasnou oblohou v závislosti na pravidlech AI. Herní AI může být použito pro vytvoření celého příběhu z již navržených částí.

Herní AI je obecně vytvořena z různých částí, které obstarávají jiné funkce. Například senzorové systémy obstarávající informace, které herní AI potřebuje, aby mohla bezproblémově fungovat. Nebo algoritmus pro hledání cest, který umožnuje, aby herní AI mohla reagovat na měnící se prostředí, aniž by bylo nutné každou cestu naprogramovat. Ale nejdůležitější částí herní AI je její algoritmus rozhodování, který umožňuje, aby herní AI mohlo reagovat na informace získané senzorovým systémem.

Předkládanou práci uvozuje literární rešerše následovaná kapitolou představující základní principy herní AI. Dále pak pokračuje představením nejčastěji používaných algoritmů a metod využívaných v tvorbě herní AI. V další časti blíže popisuje vybrané algoritmy rozhodování – konečný automat a behaviorální strom, kterými se zabývá i praktická část této práce. Další kapitola je zaměřena na využití herní AI v populárních videohrách v průběhu času a ilustruje vývoj herní AI. Kapitola 6 pak uzavírá teoretickou část práce stručným představením herního enginu a programovacího jazyku využitého dále v praktické části.

Praktická část se věnuje návrhu ukázkového prostředí využitého v obou praktických ukázkách, vlastním návrhem a implementací konečného automatu v první ukázce a návrhem a implementací behaviorálního stromu v druhé ukázce. Na závěr jsou tyto návrhy a implementace navzájem porovnány.

I. TEORETICKÁ ČÁST

1 LITERÁRNÍ REŠERŠE

Existuje velké množství zdrojů zaměřených na téma herní umělé inteligence a jejího využití ve video hrách. V zásadě můžeme rozdělit tvorbu ohledně herní AI na dva typy. První typ se zabývá obecným vytvoření herní AI nezávislé na tom, v jakém herním enginu je využitá [27]. Blíže o tomto postupu pojednává např. *Practical Game AI Programming* [4]. Druhý typ se zaměřuje přímo na vytvoření herního AI v konkrétním herním enginu za použití dostupných nástrojů [7].

Přestože je dnes většina herních AI vytvořena bez možnosti učení Bourg a Seemann [27] předpokládají, že budoucnost spočívá ve vytvoření herních AI, které budou mít schopnost se samy učit a měnit i po finálním vydání hry. Tím postupně AI může měnit své chování, a proto prodlužovat herní zážitek. Tento přístup byl v minulosti odmítán kvůli obtížnosti ladění vzniklého chování a výpočetní náročnosti učení. Ale tento sentiment se již díky několika úspěchům implementací ve vydaných hrách a díky novým vývojům v oblasti strojového učení mění [27]. V současné době se strojové učení využívá pro vývoj her i v odlišných oblastech, než je pouze herní AI například pro tvorbu animací postav nebo testování her [31].

Umělé inteligence lze využít i k zastoupení lidského hráče při hraní již hotových her – jak pro tréning, tak i pro kalibraci ovládající umělé inteligence. AI jako „umělí“ hráči se také používají v turnajích a různých soutěží, jako jsou například *Open RTS AI competition* nebo *the Google AI Challenge* [1], [3].

Ukázky herní AI v této bakalářské práci jsou implementovány v herním enginu Unity, který byl vybrán díky dostupnosti výukových materiálů, jež tvořily základ studijní literatury v oblasti využití enginu Unity. Zde bych zmínil zejména *Unity 2021 Cookbook* pojednávající o práci v herním enginu Unity s praktickými ukázkami projektů [5]. Dále *Unity Artificial Intelligence Programming* zabývající se představením a implementací herní AI v enginu Unity [7]. Pro další informace ohledně herní AI byly využity také *The Total Beginner's guide to Game AI* [8] a *Practical Game AI Programming* [4] . Výběr herního enginu také ovlivnila dostupnost nástrojů pro grafický návrh behaviorálních stromů přizpůsobených frameworku herního enginu Unity [28], [29].

Algoritmy konečného automatu a behaviorálního stromu využité v ukázkách pro rozhodování herní AI, jsou jedny z nejčastěji používaných algoritmů. Hlavním rozdílem mezi nimi

je složitost implementace i návrhu. Implementace konečného automatu je lehčí u jednodušších herních AI, ale u složitých herních AI je kód nepřehledný. Behaviorální strom je sice složitý na implementaci a návrh, ale u složitých herních AI zůstává kód díky stromové datové struktuře pochopitelný a přehledný [7], [22]. Oba algoritmy budou v této práci implementovány a složitost jejich návrhu náročnost implementace obou návrhů bude porovnána.

2 ZÁKLADNÍ TEORETICKÉ PRINCIPY HERNÍ AI

V této kapitole budou stručně popsané teoretické principy, které se často používají pro vytvoření herní AI. Tyto principy se dají rozdělit do několika hlavních sekcí: zjištění informace, přemýšlení a jednání. Pro zjištění informace se používá senzorový systém. Přemýšlení využívá metody rozhodování často kombinované s náhodou. A nakonec v jednání se vykoná akce zvolená v sekci přemýšlení. Další důležitou sekcí principů pro agenty herní AI je hledání cesty, ale tato sekce je podstatná jen, když zvolený agent má schopnost pohybu. Často herní AI mají jen krátký čas měřený v milisekundách pro splnění celé posloupnosti od zjištění informací až po vybrané jednání, které bylo zvoleno v sekci přemýšlení [7].

2.1 Senzorový systém

Senzorový systém umožnuje herní AI získávání informací pro rozhodování. Tyto informace se dají dělit na dvě časti: vnitřní a vnější. Vnitřní informace jsou například pozice charakteru ovládaného herní AI, jeho vybavení a stav postavy. Do vnějších informací náleží například aktuální pozice hráče, stav jeho zdrojů, zdali byl hráč spatřen či jestli byl další charakter ovládaný herní AI poražen. Dva často používané typy senzorového systému herní AI jsou *polling* a systém zasílání zpráv – více rozvedeno dále v následující kapitole [7].

2.2 Rozhodování

Rozhodování je jednou z nejdůležitějších částí sekce přemýšlení u herní AI. Stará se o zvolení akcí dle získaných informací a naprogramovaných možností. Pro jednoduchou herní AI můžeme také jednoduše napsat sérii *if–else* rozhodnutí, která se budou o vše starat a ovládat reakce hry. Ale pro složitější herní AI je často nutné, už jen z důvodu nepřehlednosti nebo složitosti uskutečnění požadovaného chování, využití sofistikovanějšího řešení problému jako je konečný automat, behaviorální strom nebo cílem řízené plánování akcí což je podrobněji popsáno dále [8].

2.3 Náhoda

Prvek „náhody“ se často využívá pro přidání nejistoty k chování herní AI. Tato nejistota přináší pocit realismu hráči tím, že nasimulované postavy herní AI budou dělat pochopitelné chyby a nevyberou vždy tu nejlepší možnost. Nicméně vytvoření pravé náhody je velice obtížné a pro hry neefektivní, a proto se často využívají algoritmy generující pseudonáhodné

číslice, které umožnují vyvolat pocit náhodnosti chování herní AI, která tyto algoritmy využívá [7].

2.4 Hledání cest

Schopnost hledání cesty je důležitá pro herní AI, která využívá agenty neovládané hráčem schopné pohybu. Pro pohyb je důležité mít definovanou cestu, pro kterou existuje možnost pohybu, aniž by se agent ovládaný herní AI střetl s překážkou. Tyto cesty můžou vývojáři pevně nastavit, ale takovéto řešení má problémy se škálováním implementace ve větších hrách a v dynamickém prostředí, kde herní AI musí reagovat na fyzikální simulace či na pohyb hráče v prostředí s překážkami. Z těchto důvodů se používají algoritmy hledání cesty pro vygenerování trasy od agenta k požadovanému cíli. Například často používaný algoritmus hledání cest v herním enginu Unity je A* nebo vestavěný nástroj Unity NavMesh [7].

Ne všechny výše uvedené principy jsou využité v každé herní AI, ale často se kombinují pro dosažení vývojářem specifikovaného zážitku. Například schopnost pohybu není nutná pro střeleckou věž ovládanou herní AI. Tato střelecká věž má za úkol zpozorovat hráčem ovládané postavy, upozornit další herní AI na zpozorování hráče a potom dále jen útočit na daného hráče. Na druhou stranu je tato schopnost nutná např. pro hráčem neovládaný charakter, který má za úkol dotknout se konkrétního hráče a tím ukončit hru.

3 PŘEHLED NEJČASTĚJI POUŽÍVANÝCH ALGORITMŮ A METOD

V této kapitole budou stručně popsány často používané metody a algoritmy využívané v herní AI.

3.1 Algoritmy rozhodování

3.1.1 Konečný automat

Konečný automat (anglicky „finite state machine“) je při použití u herní AI považován za behaviorálně modelovací algoritmus, který se standardně používá k návrhu a analýze automatizovaných procesů, jako jsou například algoritmy programu. Konečný automat se skládá ze dvou a více stavů a přechodů mezi těmito stavy. Přechody určují, za jakých podmínek konečný automat přejde z jednoho aktivního stavu do dalšího. Konečné automaty se využívají v herní AI na rozhodování implementací programové logiky „pokud se stane událost A tak vykonej akci B“ a rozdělením akcí na stavy. Toto rozdělení se využívá pro vytvoření dvou různých akcí v jiných stavech reagujících na totožné informace poskytnuté senzorovým systémem. Výhodou konečného automatu oproti ostatním způsobům rozhodování herní AI je jednoduchost grafického návrhu konečného automatu a dále implementace tohoto návrhu v kódu. Hlavní nevýhodou konečného automatu je nepřehlednost při tvoření složitějších herních AI [22], [18].

3.1.2 Behaviorální strom

Behaviorální strom (anglicky „behavioral tree“) je stromový datový model a také algoritmus procházení daného datového modelu využívaný pro návrh algoritmu rozhodování herní AI. V tomto datovém modelu většina uzlů reprezentuje rozhodnutí, jak má dále herní AI procházet strom rozhodnutí, až ke konečnému uzlu, jež má jen rodiče a nemá žádného potomka a obsahuje chování, které bude herní AI vykonávat. Behaviorální strom je pravidelně procházen při každém snímku od kořenového uzlu až po jeden finální uzel. Mezi výhody behaviorálního stromu patří jeho grafická podpora ve většině komerčně dostupných herních enginech, modularita jednotlivých uzlů, možnost postupného vývoje finální herní AI a jednoduchá přehlednost ve složitějších herních AI. Nevýhodou behaviorálního stromu je jeho složitá implementace a náročnost na výpočetní výkon [22].

3.1.3 Cílem řízené plánování akcí

Cílem řízené plánování akcí (anglicky „Goal-oriented action planning“) je herní plánovač využívaný pro vytvoření herní AI, která si dokáže udělat plán rozhodnutí založený na aktuálním stavu hry a na stanovených cílech. Při použití tohoto herního plánovače má herní AI přidělené možné akce a sadu cílů, které chce splnit. Pro vybrání a sestavení nejlepší sekvence akcí pro splnění požadovaných cílů a dosažení požadovaného stavu hry využívá herní AI plánovač. Zde má každá akce sadu požadavků, které musí být splněny před tím, než bude moci být vykonána, a sadu efektů, které upraví aktuální stav hry. Plánovač sestaví nejlepší postup akcí tím, že zvolí z těch akcí ty, které mají splněné požadavky a jejich efekt na aktuální stav hry vede ke splnění cíle a změně stavu hry na ten požadovaný. Cílem řízené plánování akcí je často používané ve hrách, kde je nutné, aby se herní AI přizpůsobovala neustále se měnící situaci. Výhody tohoto plánovače jsou v realismu naprogramované AI a její schopnosti reagovat na stav hry; jeho nevýhody spočívají v jeho složitosti [22].

3.1.4 Hierarchická síť úkolů

Hierarchická síť úkolů (anglicky „hierarchical task network“) je jako předešlý herní plánovač využívaná pro vytvoření herní AI, která si dokáže sama udělat plán rozhodnutí založený na aktuálním stavu hry a na stanovených cílech. Při využití hierarchické sítě úkolů má herní AI přidělené složité úkoly, které se postupně rozeberou na několik jednodušších úkol, kterých splněním se splní i tento složitý úkol. Jednodušší úkoly se vybírají na základě aktuálního stavu „světa“ při přidělení daného složitého úkolu. Každé herní AI využívající hierarchickou síť úkolů má přiděleno několik jednoduchých úkolů, které je schopno provést a několik složitých úkolů, které se mohou splnit různým složením přidělených jednoduchých úkolů. Při plánování a rozebírání složitých úkolů zjišťuje, zdali je možné pomocí jednoduchých úkolů provést tento složitý úkol. A jestli každý vybraný jednoduchý úkol splňuje část požadavků složitého úkolu. Pokud ano tak konkrétní jednoduchý úkol je vybrán a interní kopie stavu hry je pozměněna jako kdyby daný jednoduchý úkol byl už vykonán. Tato metoda rozkládání na hierarchii jednoduchých úkolů se využívá, dokud všechny požadavky složitého úkolu nejsou splněny a potom je tento zhotovený plán vykonán. Výhody tohoto plánovače jsou v realistickém chování herní AI a ve schopnosti uskutečňování odlišných plánů dle aktuálního stavu hry. Nevýhody spočívají v nutnosti naprogramovat všechny jednoduché a složité úkoly a nemožnosti uskutečnit plány, které nebyly navrženy vývojářem [22].

3.2 Algoritmy hledání cest

3.2.1 Algoritmus A*

Algoritmus A* je jeden z nejpopulárnějších algoritmů pro nalezení nejkratší cesty mezi body v grafu nebo mřížce. Díky své nenáročnosti na výpočetní výkon a jednoduchosti implementace je často používám v robotice, hledání cest na mapě, a i pro plánování cest ve hrách. A* využívá heuristickou metodu, která odhaduje cenu cesty na každou danou pozici. Tato metoda upřednostňuje prohledávání těch pozic, které jsou blízké pozicím s nízkou cenou. Dále si A* udržuje dva listy pozic. První list obsahuje všechny pozice a jejich ceny, které už byly vyhodnoceny a druhý list obsahuje všechny nevyhodnocené pozice. Když tento algoritmus vyhodnotí všechny pozice mezi startem a cílem, tak začne sestavovat cestu od cíle ke startu podle nejnižší ceny vyhodnocených pozic. Využití algoritmu A* oproti nastavení pevné cesty u agenta ovládaného herní AI je možnost reagování a změna cesty v případě následování hráče nebo také šetření času při vytváření více herních prostředí bez nutnosti vytvoření pevných cest pro každého agenta [4].

3.2.2 NavMesh

Unity NavMesh je vestavěný způsob hledání cesty v herním enginu Unity. Tento systém funguje tak, že vygeneruje navigační mřížku (anglicky „navigation mesh“) která představuje zjednodušený prostor herního prostředí, kde se agent ovládaný herní AI může pohybovat. Navigační mřížka se generuje pomocí analýzy geometrie herního prostředí, značek, kterými může vývojář označit objekty v herním prostředí a porovnání s nastavenou velikostí agenta, který se na vygenerované mřížce bude pohybovat. Tento systém podporuje dynamické hledání cesty pomocí znovu generace navigační mřížky nebo i více variant navigačních mřížek pro agentů o jiných velikostech. Tento systém využívá algoritmus A*. Tím získává jeho výhody v nenáročnosti na výpočetní výkon, ale obohacuje ho o automaticky generované mřížky [7].

3.2.3 Simulace pohybu hejna

Simulace pohybu hejna (anglicky „flocking“) je způsob pro jednoduchou simulaci např. hejna zvířat ve hře. Pro simulaci využívá několika pravidel, mezi které náleží separace, soudržnost a uspořádání. Separace ustanovuje, že každý člen skupiny by se měl pokoušet o udržení stanovené vzdálenosti od dalšího člena. Soudržnost znamená, že každý člen by se měl pohybovat směrem do centra skupiny. Uspořádání předpokládá, že každý člen by se měl

pohybovat stejným směrem jako jeho sousední člen. Tato metoda simulace se často využívá ve hrách pro nasimulování stád, hejn nebo i davů [7].

3.3 Senzorové systémy

3.3.1 Polling

Polling je jednou z metod senzorového systému, při které herní AI pravidelně získává informace o hodnotách herních proměnných, kterými jsou například pozice dalších charakterů a agentů, stavy objektů v herním prostoru, aktuální stav hráčem ovládaného charakteru nebo akce provedené hráčem. Nevýhodou pollingu je relativní výpočetní náročnost oproti ostatním metodám. Hlavně při využití více agentů herní AI. Ale tuto nevýhodu lze snížit zmenšením frekvence získávání informací, rozdělením agentů do skupin s jinými intervaly využití pollingu nebo i omezením prostoru, kde se informace získávají na okruh okolo daného agenta. Největší výhoda této metody spočívá v jednoduchosti naplánování jejího provedení a dále snadné implementaci [7].

3.3.2 Systém zasílání zpráv

Systém zasílání zpráv (anglicky „Messeging system“) je další z metod senzorového systému. Tato metoda využívá vysílání a naslouchání eventům k předávání důležitých informací o stavu herního prostředí. Často se metody pollingu a systému zasílání zpráv kombinují pro dosáhnutí nižší výpočetní náročnosti, než má samostatná metoda pollingu. Při využití této metody důležité události, jako je zpozorování postavy hráče nebo splnění specifického úkolu, vyšlou event, na který určití agenti herní AI naslouchají a čekají. Dále na tento event reagují vykonáním vybrané akce. Při kombinaci s metodou polling se systém zasílání zpráv spustí pro získání další informací. Metoda systému zasílaní zpráv má oproti dalším metodám výhodu v nízké výpočetní náročnosti při využití více agentů herní AI. Nevýhody této metody jsou ve složitější implementaci [7].

3.4 Strojové učení

Strojové učení (anglicky „Machine learning“) je způsobem využití algoritmů a modelů pro samostatné učení se z dat a dělání odhadů nebo rozhodnutí, aniž by bylo nutno tyto akce naprogramovat. Při využití strojového učení se používají algoritmy na analýzu velkého množství dat a z těchto dat se pak zjišťují postupy a zlepšují svůj výkon pro splnění speci-

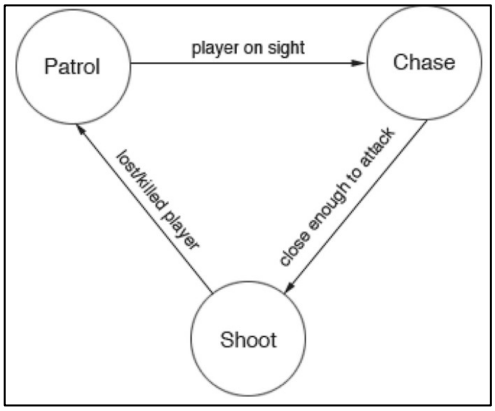
fických úkolů. Ve hrách se strojové učení může použít k vycvičení herní AI na předpřipravených modelech herního prostředí, které jsou upraveny o odměny a tresty za učení a tím se zredukují časové požadavky vývojářům na programování složité herní AI. Výhody strojového učení vůči konečnému automatu nebo behaviorálnímu stromu spočívají v jednoduchosti výsledné implementace a vyřešení složitých problémů. Nevýhody tohoto přístupu jsou obtížnost ladění problémů, obtížná změna chování vycvičeného herního AI, výpočetní a časová náročnost k vycvičení herní AI a nutnost vytvořit herní prostředí upravené pro cvičení [7].

4 PODROBNĚJŠÍ POPIS VYBRANÝCH ALGORITMŮ

V této kapitole budou ještě podrobněji popsané algoritmy dále využité a porovnané v praktické části této práce.

4.1 Konečný Automat

Konečný automat (anglicky „finite state machine“), jak už jsem dříve zmínil, je behaviorálně modelovací algoritmus, který se u herní AI využívá pro rozhodování pomocí výběru z vývojářem vytvořených rozhodnutí na základě získaných informací a aktuálního stavu pozice konečného automatu. Konečný automat je obecně vytvořen ze stavů a přechodů [22]. Na obrázku 1. je např. znázorněna logika chování postavy stráže složená ze tří stavů a tří přechodů.



Obrázek 1. Příklad jednoduchého konečného automatu simulujícího chování postavy [7].

U této metody může herní AI využívat pouze jediný stav v jednu chvíli. Přesunutí do jiného stavu nastane při splnění podmínky k přechodu. Například prodleni několik sekund nebo dokončení specifikované akce. Tím konečný automat umožnuje rozdělit chování na části oddělených stavů a využít vybraných přechodů jen za podmínky, že herní AI se bude nacházet v daném stavu. Tak je schopný tvořit herní AI, která hráči bude připadat inteligentnější. Ale existují i problémy, kdy je potřeba, aby herní AI byla ve více stavech najednou, nebo aby se několik stavů opakovalo. Tyto problémy se pak dají řešit pomocí hierarchického konečného automatu, při kterém je možné vnořit jeden nebo více konečných automatů do jednotlivých stavů celkového hierarchického konečného automatu. Tím se zjednodušuje struktura návrhu implementace a umožnují se chování nemožná prostou metodou konečného automatu. Stav může zahrnovat jak akce a chování herní AI, tak i jeden a více stavů, které se

používají čistě pro oddělení přechodů. Pokud stav neobsahuje přechod, tak se nazývá stavem konečným, který se ale tak často nepoužívá při programování herní AI [22].

Konečný automat dle implementace, například využitím jednoduchého vícecestného větvení může vyvolat akce po přechodu do nového stavu a potom tyto akce dále opakovat pokaždé, kdy herní AI vyhodnocuje, jakou akci má provést a stav zůstává stejný. Nebo pomocí více tříd, kdy každá třída bude obsahovat kompletní chování jednoho stavu rozdělené na tři metody. První metoda se nazývá začátek a vykoná se pouze při přechodu do stavu. Druhá metoda s názvem aktualizace se vykoná každý snímek za podmínky, že herní AI nezměnila stav. Poslední metoda s názvem konec se vykoná pouze při přechodu do jiného stavu [22].

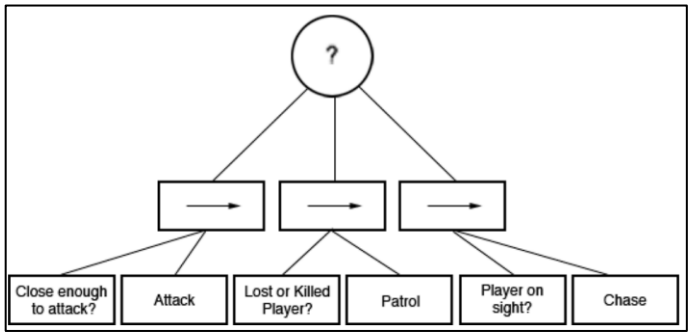
Tato metoda má několik výhod oproti dalším metodám rozhodování. První výhodou je organizace chování a akcí do stavů. Tímto umožnuje jednodušší úpravu nebo přidání chování podle pozdější potřeby. Další výhodou je jednoduchá struktura návrhu implementace, která umožnuje bezproblémové načrtnutí daného návrhu a vysvětlení chování agentů. Třetí výhodou je relativně nízká výpočetní náročnost oproti dalším metodám. A poslední výhodou je jednoduchost ladění implementace. Nicméně hlavní nevýhoda využití konečného automatu spočívá v nepřehlednosti při vytvoření větších a složitějších herních AI [22].

4.2 Behaviorální strom

Behaviorální strom (anglicky „behavior tree“), je datový model a algoritmus průchodu využívaný pro rozhodování herní AI. Jak už předem zmíněný konečný automat, tak tento algoritmus se používá u herní AI pro vybírání z vývojářem vytvořených rozhodnutí na základě získaných informací ze senzorového systému. Behaviorální strom je složen z různých druhů uzlů např. kořenu, přepínače, indikátoru řady, ozdobného indikátoru a listu [22].

Kořen stromu je často přepínač a chová se jako start, kde celý průchod algoritmem začíná. Přepínač vybírá ten přímo následující uzel, který jako první vrátí pozitivní informaci ohledně jeho vykonání. Při výskytu pozitivní informace následujícího uzlu, přepínač také vrátí pozitivní informaci ohledně jeho stavu předešlému uzlu. Když přepínač nedostane žádnou pozitivní informaci ohledně průběhu přímo následujících uzlů, tak přepínač vrátí negativní informaci ohledně jeho stavu předešlému uzlu. Indikátor řady vykonává všechny uzly pod tímto indikátorem postupně v řadě od uzlu nejvíce nalevo až po uzel nejvíce napravo. Vy-

konávání je zastaveno, pokud jakýkoliv uzel vrátí negativní informaci ohledně jeho vykonání. Při obdržení této negativní informace indikátor řady také vrátí negativní informaci ohledně jeho stavu předešlému uzlu. Ozdobný indikátor oznamuje upravené spuštění uzlů pod tímto indikátorem, například jejich opakováním. List je konečný bod celého stromu a obsahuje akci kterou herní AI provede nebo podmínku která s ohledem na výsledek vrátí pozitivní nebo negativní informaci ohledně stavu daného listu [22]. Na obrázku 2 níže. Je například znázorněna logika chování postavy stráže složená z jednoho kořenového přepínače, tří indikátorů řady a šesti listů obsahující jak podmínky, tak i akce vykonatelné postavou.



Obrázek 2. Příklad jednoduchého behaviorálního stromu chování postavy stráže [7].

Na rozdíl od konečného automatu, behaviorální strom neomezuje herní AI jen na jeden aktivní stav. V originální implementaci bez optimalizací návrhu algoritmu se prochází celý behaviorální strom od kořene až do listu, kde je vykonána akce uložená v daném listu, a tento průchod se opakuje při každém dotazu pro chování herní AI. Tento způsob průchodu má výhodu rychlé reakce na změnu informací získaných senzorovým systémem a na rozdíl od konečného automatu (který musí obsahovat kopie jedné podmínky na kterou vývojář navrhl herní AI, aby reagovala pokaždé, kdy se splnily ve více stavech) behaviorální strom může obsahovat pouze jednu danou podmínku bez nutnosti kopírování, která se splní i když herní AI aktuálně prochází jinou větev behaviorálního stromu. Díky tomu, že každý průchod behaviorálního stromu začíná od kořene, tak tyto podmínky je možné implementovat jen jako list zařazený pod přepínač nebo indikátor řady – tím snížíme počet opakovaného kódu v algoritmu. Hlavní nevýhoda začátku každého průchodu od kořene je vyšší výpočetní náročnost rozhodnutí. Ale tuto nevýhodu lze minimalizovat zapamatováním posledně aktivního listu – touto optimalizací snížíme počet průchodů od kořene po list behaviorálního stromu [22].

Populární optimalizací využití behaviorálního stromu pro vytvoření více herních AI je využití místa pro ukládání informací získaných senzorovým systémem, ze kterého můžou všechny herní AI získávat a ukládat do něj informace. Tato optimalizace umožnuje nižší využití senzorového systému pro získání informací o stavu herního světa [22].

Výhody behaviorálního stromu vůči konečnému automatu, jsou v grafické podpoře této metody ve většině herních enginů, jednodušší implementace podmínek, na které musí herní AI vytvořená konečným automatem reagovat ve více stavech, schopnost znovupoužití implementovaného chování jedné herní AI pro další odlišnou herní AI a vyšší přehlednost kódu ve větších a složitějších implementací herní AI. Ale i snadný převod návrhu z konečného automatu na návrh behaviorálního stromu. Nevýhody této metody spočívají ve vyšší složitosti implementace behaviorálního stromu oproti jednoduché implementace konečného automatu a vyšší nárok na výpočetní výkon neoptimalizovaného algoritmu [7].

5 VYUŽITÍ HERNÍ AI VE VYBRANÝCH HRÁCH

V této kapitole budou stručně popsány vybrané populární hry využívající některé algoritmy a metody zmíněné v předešlých kapitolách. Dané hry jsou seřazeny dle roku prvního vydání, pro naznačení vývoje herní AI.

5.1 Pac-Man

Herní AI je používaná v elektronických hrách téměř od samostatných počátků, kdy se objevila v arkádových automatech. Jednou z nejznámějších a nejstarších her využívajících herní AI – v tomto případě konečného automatu – byl Pac-Man (vydán roku 1980). V této hře bylo několik variací konečného automatu, kde každý byl přidělen jinak barevnému protivníkovi hráče a každý využíval lehce jiná pravidla. Tyto variace byly vytvořeny, aby se chování s protivníky lišilo a hráči měli různorodější a obtížnější hru [7].

5.2 Street Fighter 2

Bojová hra Street Fighter 2 (vydána v roce 1991) využívá herní AI pro simulaci protivníka v příběhovém nebo trénovacím režimu.

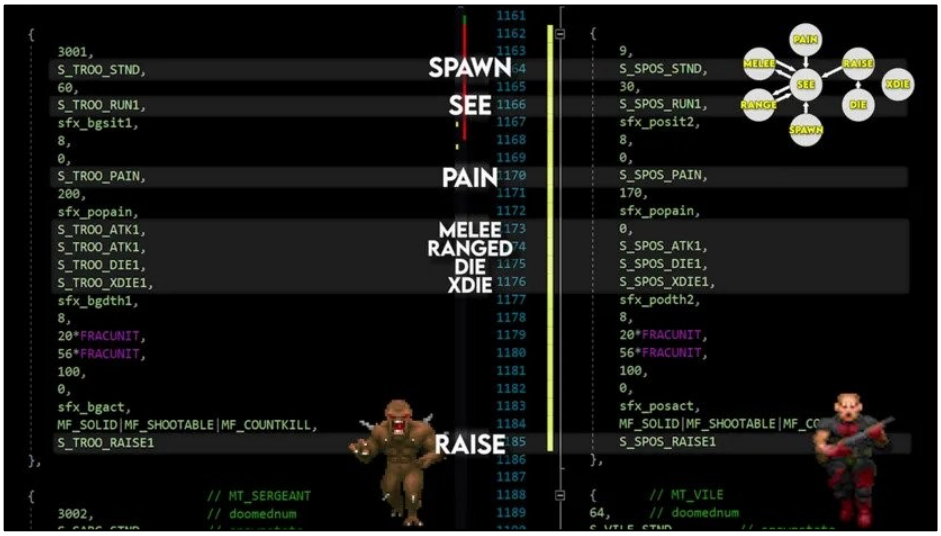
Obrázek 3. Street Fighter 2 [20].

Tato herní AI je sestavena ze sad skriptů, ze kterých jsou vybírány skripty na základě oponenta, proti kterému bude herní AI bojovat v souboji a také dle okolností tohoto souboje. Uvedené skripty se skládají z příkazů pro vyvolání útoku, pohybu na určitou pozici, nebo i jen jednoduchý skok. Dále tyto skripty mohou obsahovat jednoduché podmínky pro zjištění aktuálního stavu jak oponenta, tak i samostatného agenta ovládaného herní AI. Herní AI hlavně využívá typy chování. První typ je „čekání na útok“ při kterém se často volí pohyb

v náhodných směrech. Druhým typem je „Útočení“ při kterém se vybírají skripty s útoky. Poslední třetí typ je „Reagování“ při kterém se vybere skript s instrukcemi na negování útoků. Tato herní AI nedodržuje stejná pravidla jako hráč, tím že dokáže provést útoky z pozic, ze kterých by je hráč nemohl provést [20].

5.3 DOOM (1993)

DOOM, viz Obr. 4 níže, je akční hra, která svým stylem, využitím herní AI a popularitou inspirovala následující generace her.



Obrázek 4. Externí textový soubor definující chování agentů ve hře DOOM (1993) [17].

Tato hra využívá konečný automat sestavený z osmi stavů pro konkrétní rozhodování. Pro definování chování agentů se využívá externí textový soubor, ve kterém jsou určené hodnoty zdrojů, vlastnosti a akce pro dané stavy nepřátel. Tyto akce obsahují jak chování tak i animace nepřátel vybrané podle jejich orientace s ohledem na hráče. DOOM využil optimalizace rozdělením herní mapy na menší části, pro určení, jestli daný agent má možnost spatřit nebo zaslechnout hráče [12], [17].

5.4 Command & Conquer

Ve strategických hrách Command & Conquer (vydáno v roce 1996) a Command & Conquer Remaster (vydáno v roce 2020) se herní AI využívá u každé jednotky a pro celkovou simulaci protihráče. Většina jednotek má v kódu funkci pro AI, ve které je vše, co musí provádět automaticky – tedy animace, hledání cest a specifické chování. Dále má každá jednotka pří-

stup ke stavu herního světa, ale i několik interních stavů podle kterých vybírá správné chování. Chování oponenta ovládaného AI je zde rozděleno na mise a strategie. Mise se vztahují na chování jednotek, jenž by mohl reálný hráč přikázat. Ve hře je dvacet dva různých misí, ze kterých herní AI vybírá. Strategie se vztahují na celkové chování oponenta. Tyto strategie se skládají pouze z pěti stavů, podle kterých rozhoduje, zda postaví novou budovu nebo jaké příkazy přidělí jednotkám. Dále vykonává rozhodnutí ohledně budov, které musí postavit a na jaké cíle zaútočit, pomocí informací ohledně postavených budov dalších oponentů – jak oponentů ovládaných herní AI, tak i hráčů [16].

5.5 Half-Life

Další slavná akční hra Half-Life (vydána v roce 1999), popularizovala konečné automaty pro herní AI. Konečný automat zde využili pro oddělení velkého množství chování rozdělením tohoto chování do stavů. Half-life využil polymorfismus pro implementaci konečného automatu, kde každý agent ovládaný herní AI dědí ze třídy „Monster“ která obsahuje nejčastější a sdílené chování všech typů agentů. Jejich vlastní specifické chování nebo verze sdíleného chování upravená pro daný typ agenta je implementována přímo v dané třídě tohoto agenta. Half-life má implementovaný systém rozvrhů, kde rozvrh zahrnuje několik akcí, které se postupně provedou v pořadí. Tímto systémem bylo usnadněno vytvoření a využití vývojářem navržených sekvencí akcí, například sekvence začínající útěkem agenta od hráče, nalezením úkrytu, a nakonec útočení na hráče z úkrytu [10].

5.6 Halo 2

Ve slavné akční hře Halo 2 (vydáno v roce 2004), je popularizován behaviorální strom pro vytvoření komplexní herní AI. Dále je implementovaný systém důležitosti akcí do listů behaviorálního stromu, který upravuje rozhodování o výběr listů obsahující akce. Nejčastější využití systému důležitosti akcí je schopnost přerušit vykonávání listu s nižší hodnotou relevantnosti akcí s vyšší hodnotou. Tento systém ale zvyšuje výpočetní náročnost herní AI tím, že i v kontextu daného ovládaného agenta i akce, které jsou nepodstatné nebo nevykonatelné mají svou hodnotu důležitosti stále vypočítávanou. Tento problém byl ale vyřešen zákazem listů s akcemi, které v daném kontextu nejsou nutné a povolením těchto listů při změně kontextu. Například když agent herní AI řídí vozidlo, tak není nutné, aby mu byla přepočítávána hodnota důležitosti střelby na hráče z pozice za krytem. Tím se snížilo množství listů, u kterých bylo nutné přepočítat hodnotu důležitosti a tím i výpočetní náročnost [9].

5.7 F.E.A.R

Akční hororová hra F.E.A.R (vydáno v roce 2005) využívá konečný automat společně s cílem řízeným plánováním akcí pro herní AI která dokáže reagovat na změnu stavu herního světa a plánovat pro maximalizaci efektu svých akcí na zážitek hráče. Pro plánování akcí bylo vytvořeno 70 cílů s hodnotou priority která se přepočítává s ohledem na stav hry. Každá postava má přidělené určité akce, které může uskutečnit, a při přístupu k plánovači se generuje plán na základě těchto přidělených akcí. Tento způsob implementace herní AI ale vytvořil problém, kdy agenti herní AI nedokáží vzájemně spolupracovat, a tak všechny akce plánují jen s ohledem na aktuální stav hry a svůj vlastní stav [11].

5.8 DOOM (2016)

DOOM je restartování celé série DOOM. Obecně využívá hierarchický konečný automat pro herní AI navrženou pro akční styl hraní. Pro volení pozice viditelné hráčem využívá systém hledání krytu, pomocí kterého zjišťuje, jestli je agent ovládaný herní AI za něčím skryt. Dále si při střetnutí s hráčem na pozadí nepřátelští agenti zjišťují, kdy a jestli mohou zaútočit, a to systémem přidělování tokenů kterých je omezený počet, aby neútočilo až moc nepřátelských agentů na hráče [13].

5.9 Forza Motorsport 7

Závodní hra Forza Motorsport 7 (vydána v roce 2017) využívá herní AI s názvem „Drivatar“. Tato herní AI je jednou z prvních herních AI, které vyžívají strojové učení. Využitím strojového učení se dokáže naučit a dále simulovat chování a řízení reálných hráčů, čímž zjednodušuje navrhování a implementaci závodních okruhů. Pro každého hráče je vytvořen jeden „Drivatar“, který dále využívá data daného hráče pro adaptaci podle záznamů jízd. Tato data se generují pro každý závod, který hráč dohraje do konce a dělí se podle sekcí závodního okruhu. Vygenerovaná data jsou ošetřena proti špatným vstupům, kdyby se hráč například rozhodl celý závod odjet pozpátku. Dále tato data obsahují přesný čas vygenerování a starší data se pokládají za nepřesné kvůli předpokladu zlepšení řidičských schopností hráče. Takto vyučený „Drivatar“ se nadále využívá jako nepřátelský závodník při závodech ve světě dalšího hráče, kdy napodobuje předpokládané chování a řízení závodního automobilu na tratích, kde hráč, podle kterého je nasimulovaný „Drivatar“ nikdy nezávodil. Na tratích, kde simulovaný hráč závodil, se „Drivatar“ pokouší co nejblíže vystihnout originální chování a řízení daného simulovaného hráče [15].

5.10 Gran Turismo

Závodní hra Gran Turismo 7 (vydáno v roce 2022) se odlišuje od předchozích dílů série možností využití experimentální herní AI s názvem „Sophy“ která využívá hluboké učení pro simulování profesionálních závodníků.

Obrázek 5. Herní AI „Sophy“ závodící v Gran Turismo 7 [19].

První veřejný test této AI byl od 23. února 2023 až do 31.března 2023. Pro vývoj této herní AI byly navržené čtyři schopnosti, které se musí naučit, aby její simulace byla dostatečně blízká reálnému závodníkovi, konkrétně: kontrola vozidla, taktika řízení, strategie řízení a etiketa závodění. Herní AI „Sophy“ se na rozdíl od herní AI „Drivatar“ ve hře Forza neučila řídit podle dat řízení existujících hráčů dané série, ale bez předchozího modelu. Pro učení byl využit algoritmus QR-SAC (Quantile regression soft-actor critic), směs scénářů vytvořených vývojáři a pro učení bylo vybráno osm bonusů a postihů indikujících dobré a špatné řízení. Mezi tyto bonusy a postihy patří bonus za postup na závodní dráze nebo například postih za jízdu mimo závodní dráhu. Pro učení „Sophy“ byl využíván předešlý díl z herní série Gran Turismo, a to Gran Turismo Sport (vydáno v roce 2017), aby byla hotová a připravená do Gran Turismo 7. „Sophy“ funguje tak že, využívá několik vstupů, podle který se orientuje, ale ovládá jen dva výstupy brzda/plyn a zatáčení vpravo/vlevo. Před tím, než vybere, co a jestli něco změní na výstupu, tak nejdřív vypočítá několik možných rozhodnutí s pravděpodobností nejlepšího výsledku. Z těchto rozhodnutí potom zvolí to s nejvyšší pravděpodobností [14], [19].

6 VYUŽITÉ TECHNOLOGIE

V této kapitole bude stručně představen v praktické části využitý herní engine Unity a zvolený programovací jazyk.

6.1 Herní engine Unity

Herní engine je obecně softwarový framework nebo platforma, která je specificky navržená pro tvorbu video her. Vývojářům poskytuje nástroje a knihovny pro návrh a tvoření herních světů, prostředí, charakterů, animací, fyzických simulací, uživatelských rozhraní a dalších komponentů využívaných ve videohrách [23].

Herní enginy jsou navrženy, aby ulehčily a urychlily proces vývoje video her, tím že poskytují už vytvořené a otestované nástroje, které je možné využít ve hrách bez nutnosti jejich vytvoření pro danou hru. Tímto umožnují herním vývojářům, aby se mohli zcela soustředit na vývoj herních mechanik, grafické prvky hry nebo herní návrh [23].

Herní enginy jsou často sestaveny z více subsystémů, které spolu bezproblémově spolupracují. Například renderovací engine využitý pro vykreslování grafických prvků hry, fyzický engine simulující fyzické reakce herních objektů, audio engine využívaný pro zvukové efekty, síťový engine používaný pro hry, které komunikují přes internet nebo LAN síť, a další [23].

Herní enginy nejsou jen používány pro tvorbu video her, ale také se často používají pro vizualizaci architektonických návrhů, předvádění konceptů automobilů nebo i aplikace virtuální reality [23], [21].

Na aktuálním trhu se pohybuje několik herních enginů dostupných pro veřejnost nebo herní společnosti. Tato nabídka herních enginů obsahuje jednoduché enginy specializované na hry ve dvou dimenzionálním prostoru, složitější enginy specializované na hry ve třech dimenzích, a i herní enginy které dokážou vývoj jak dvoudimenzionálních, tak i trojdimenzionálních video her. Mezi nejpopulárnější herní enginy patří Unity, Unreal Engine, CryEngine a Godot. V této práci je dále využíván populární herní engine Unity. [32]

Unity je herní engine umožňující vývoj her nejenom pro osobní počítače, ale i mobilní telefony, herní konzole a virtuální realitu. Unity je vyvíjený společností Unity Technologies. Dále umožnuje vytvářet jak dvoudimenzionální, tak i trojdimenzionální video hry.

Mezi platformy, které jsou podporovány herním enginem Unity náleží operační systémy iOS, Android, Windows, macOS a Linux. Dále také herní konzole Xbox, PlayStation, Nintendo Switch a mnoho dalších. Svou širokou podporou se liší od dalších herních enginů, u kterých je obtížnější vytvořené hry publikovat na nepodporovaných platformách [25].

Dále Unity Technologies umožnuje bezplatné využívání herního enginu Unity pod personální licencí pro osoby nebo společnosti, které mají roční výdělek nižší než sto tisíc dolarů. Tato bezplatná verze neobsahuje některé nástroje a neumožnuje vývojářům odstranit úvodní obrazovku informující hráče, že daná hra byla vytvořena pomocí herního enginu Unity, ale je stále dostatečně robustní, aby v ní mohla být vytvořena plnohodnotná video hra [24].

Unity je robustní a flexibilní herní engine, který je často používaný v herních studiích. Díky jednoduchosti používání, velkému rozsahu poskytovaných nástrojů, podporovaných platforem a své bezplatné personální licenci, jsem si ho nakonec vybral jako ideální herní engine pro implementaci ukázek herní AI ve své bakalářské práci.

6.2 Programovací jazyk C#

Pro programování her a tvorbu herních AI se využívají různé programovací jazyky. V této kapitole jen velmi stručně uvedu populární jazyk C# podporovaný a využívaný herním enginem Unity

Jazyk C# je obecně objektově orientovaný programovací jazyk, který byl vytvořen v roce 2000 společností Microsoft. Byl navržen jako jednoduchý a efektivní programovací jazyk využívající platformu Microsoft .NET Framework [6].

Platforma .NET Framework obsahuje „Common Language Runtime“ a „Base Class Library“, kde první zmíněný je využívaný pro exekuci kódu a druhý obsahuje knihovny tříd s často využívanými metodami např. metody pro matematické výpočty nebo pro práci se vstupy a výstupy osobního počítače [6].

Herní engine Unity dále využívá implementaci .NET Frameworku s názvem Mono, která umožnuje využití naprogramovaného kódu i na jiných operačních systémech než Microsoft Windows. Specifikace implementace Mono jazyka C# byly schváleny soukromou neziskovou organizací ECMA, která se zaobírá normalizací informačních a komunikačních systémů. Dále Mono obsahuje kompilátor jazyka C#, Mono Runtime, knihovnu .NET Framework tříd a knihovnu Mono tříd [26].

II. PRAKTICKÁ ČÁST

7 CÍLE A STRUKTURA PRAKTICKÉ ČÁSTI

Cílem praktické části práce je návrh, implementace a porovnání dvou vybraných herních AI využívajících odlišné algoritmy pro rozhodování. První algoritmus využívá princip již dříve popsaného konečného automatu. Druhý je pak založený na využití behaviorálního stromu. Obě implementace využívají totožný ukázkový prostor a pokouší se vytvořit obdobné chování herních AI pro srovnatelnost obou vytvořených ukázek.

Osmá kapitola praktické části obsahuje popis tvorby ukázkového prostoru využitého v následujících dvou ukázkách. Tato sekce obsahuje dvě podkapitoly. První popisuje generaci navigační mřížky použité pro hledání cest a pohyb agentů herních AI. Druhá popisuje skript využitý pro vytváření instancí zlatých kostek.

Devátá kapitola podrobně popisuje návrh vlastního konečného automatu pro požadované chování. To se skládá ze tří stavů. První stav je počáteční a obsahuje přechody do dalších dvou stavů, kdy herní AI čeká na uběhnutí nastaveného času. Druhý stav obsahuje přechod do třetího stavu a chování, kdy herní AI náhodně volí body jako cíle, do kterých pohybuje agenta. Poslední stav obsahuje vratný přechod do předchozího, druhého stavu a chování, kdy agent herní AI dojde ke zlaté kostce a tu sebere. Kapitola dále popisuje implementaci vlastního konečného automatu. Nakonec je uvedena podkapitola ohledně měření času provedení vlastního konečného automatu.

Desátá kapitola potom obsahuje popis návrhu vlastního behaviorálního stromu. Chování ve stromu bylo rozděleno do tří větví. V první větvi byly využity tři listy obstarávající chování, kdy herní AI najde zlatou kostku, ke které agent dojde a tu pak sebere. Druhá větev obsahuje jen jeden list zabezpečující chování, kdy AI čeká na uběhnutí nastaveného času. Poslední větev obsahuje tři listy obstarávající chování, kdy AI volí náhodné body jako cíle, do kterých pohybuje agenta. Dále v této kapitole podrobně popisuje implementaci vlastního behaviorálního stromu. Nakonec je opět podkapitolou ohledně měření času provedení vlastního behaviorálního stromu.

Jedenáctá kapitola praktické části potom porovnává vytvořené implementace konečného automatu a behaviorálního stromu z několika hledisek.

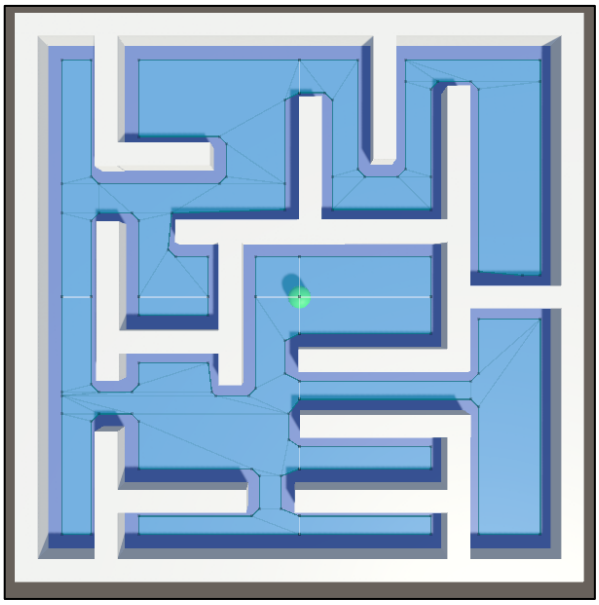
8 UKÁZKOVÝ PROSTOR

Návrh prostoru ukázky byl hlavně zaměřen na vytvoření prostředí a systému, na které herní AI může reagovat bez vnějšího vstupu hráče. Jako prostředí byl využit jednoduší typ labyrintu (viz Obr. 6 níže), aby bylo možné předvést automatickou navigaci agenta pomocí navigační mřížky.

Obrázek 6. Ukázkový prostor se zeleným agentem herní AI.

8.1 Navigační mřížka

Pro navigaci je využit vlastní nástroj enginu Unity pro generování navigační mřížky a následně dynamické hledání cest. Pro vytvoření navigační mřížky byly označeny všechny herní objekty, ze kterých se skládá prostor labyrintu, a bylo nastaveno, zda má u těchto objektů AI umožněn pohyb nebo nikoliv. Dále byla přidělena komponenta ***NavMeshAgent*** hernímu objektu s názvem ***Agent***, aby mohl hledat cestu a tím bylo umožněno herní AI využit tohoto agenta pro pohyb v ukázkovém prostoru. Nakonec byla vygenerována navigační mřížka pro tento labyrint, viz Obr. 7 níže.



Obrázek 7. Navigační mřížka labyrintu.

8.2 Vytváření instancí zlatých kostek

Pro vytvoření vstupu, na který může herní AI v ukázce reagovat, aniž by bylo potřeba vnějšího podnětu, jsem využil náhodně umísťované zlaté kostky. Kostky jsou vytvořeny pomocí skriptu s názvem ***ObjectSpawner*** umístěného v prázdném herním objektu s názvem ***GameManager***. Ten obsahuje tři proměnné pro ukládání dat – viz Obr. 8 níže.

```
//time set on start of the aplication. Periodicaly reset on cube spawn.
private double timeOnStart;
//field holding prefab used for spawning
public GameObject myPrefab;
//field holding time between spawning
public double waitTime = -5f;
```

*Obrázek 8. Zdrojový kód proměnných skriptu **ObjectSpawner.***

První privátní proměnná ***timeOnStart*** obsahuje čas, který je nastaven při spuštění ukázky a dále je aktualizována s vytvářením instancí zlatých kostek. Vůči tomuto času je kontrolován aktuální čas. Druhá veřejná proměnná ***myPrefab*** obsahuje herní objekt, ze kterého tento skript vytváří instance. Do této proměnné je vložený herní objekt zlaté kostky v editoru Unity. Třetí proměnná obsahuje čas mezi vytvářením jednotlivých instancí. Poslední dvě proměnné mají nastavenou přístupnost na veřejnou, aby mohly být nastaveny přímo z editoru Unity.

8.2.1 Metoda Start

Dále požívám metodu ***Start*** ze třídy ***MonoBehaviour*** viz Obr. 9 níže, která se provede pouze jednou před prvním vykonáním metody ***Update***.

```
// Start is called before the first frame update. Sets time.
Unity Message | 0 references
void Start()
{
    timeOnStart = Time.time;
}
```

*Obrázek 9. Zdrojový kód metody **Start** skriptu **ObjectSpawner.***

V této metodě se jen nastaví proměnná ***timeOnStart*** na aktuální čas v momentu spuštění ukázky.

8.2.2 Metoda Update

Druhá metoda ze třídy ***MonoBehaviour*** je ***Update***, zobrazená na obrázku 10 níže, která se vykonává každý snímek.

```
// Update is called once per frame. Calls the Spawn() method with RandomPosition() method as parameter.
Unity Message | 0 references
void Update()
{
    Spawn(RandomPosition());
}
```

*Obrázek 10. Zdrojový kód metody **Update** skriptu **ObjectSpawner.***

V této metodě jsou volány dvě originální metody, a to metoda ***Spawn*** pro kontrolu času mezi vytvořením jednotlivých instancí zlaté kostky a metoda ***RandomPosition***, která vrací náhodnou pozici v trojrozměrném vektorovém formátu.

8.2.3 Metoda Spawn

Metoda ***Spawn*** na obrázku 11 níže, se využívá pro vytváření instancí zlatých kostek.

```
//Checks if sufficient time has pased, if so then instantiates prefab on provided position.
1 reference
void Spawn(Vector3 spawnPositon)
{
    if (timeOnStart - Time.time <= waitTime)
    {
        Instantiate(myPrefab, spawnPositon, Quaternion.identity);
        timeOnStart = Time.time;
    }
}
```

*Obrázek 11. Zdrojový kód metody **Spawn** skriptu **ObjectSpawner.***

Tato metoda kontroluje, jestli rozdíl času uloženého v poli ***timeOnStart*** a aktuálního času je menší nebo roven času uloženému v poli ***waitTime***. V případě, že je rozdíl menší nebo roven poli ***waitTime***, tak metoda ***Spawn*** volá metodu ***Instantiate*** poskytovanou knihovnou ***UnityEngine*** pro vytvoření instance herního objektu na pozici, kterou má metoda ***Spawn*** předanou z vnějšku. Nakonec nastavuje proměnnou ***timeOnStart*** na aktuální čas.

8.2.4 Metoda RandomPosition

Poslední metoda obsažená v tomto skriptu je metoda ***RandomPosition*** zobrazena na obrázku 12 níže.

```
//Gets random Vector3 position, then checks if said position is on the NavMesh grid. Return Random positon.
1 reference
Vector3 RandomPosition()
{
    int distance = Random.Range(1, 50);
    Vector3 randomDirection = Random.insideUnitSphere * distance;
    NavMeshHit hit;
    NavMesh.SamplePosition(randomDirection, out hit, distance, -1);
    return hit.position + new Vector3(0,1,0);
}
```

*Obrázek 12. Zdrojový kód metody **RandomPosition** skriptu **ObjectSpawner.***

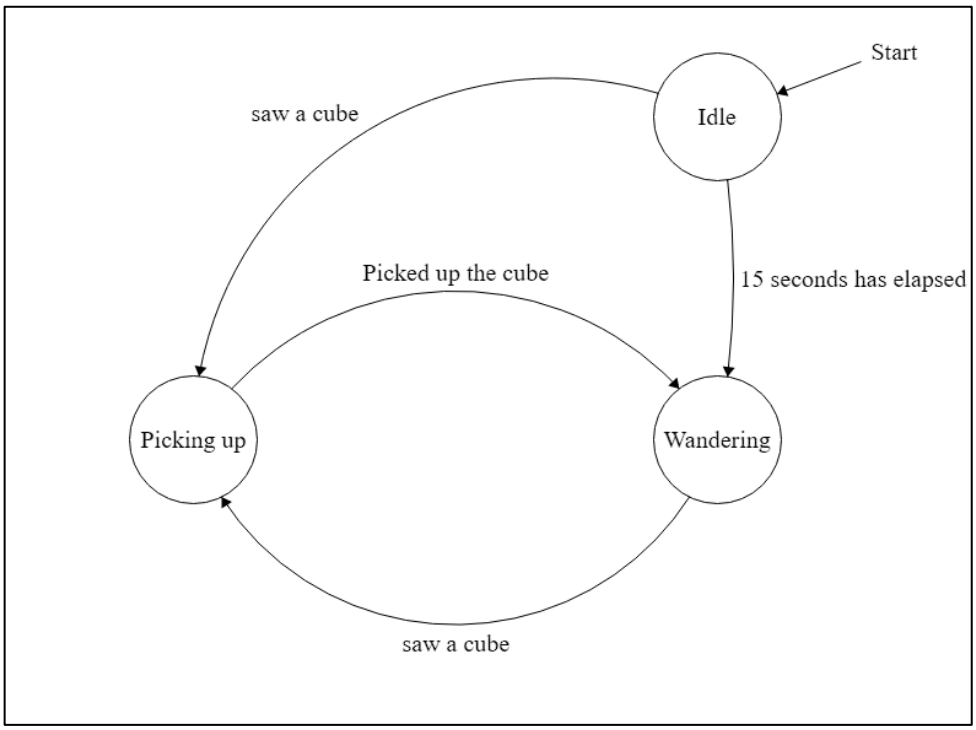
Tato metoda vrací náhodnou pozici. Začíná získáním náhodného celého čísla z rozsahu jedna až padesát. Dále získává náhodný trojrozměrný vektor zavoláním metody ***insideUnitSphere*** ze třídy ***Random***, který je potom násoben s již dříve získaným náhodným celým číslem. Pokračuje pak vytvořením proměnné ***hit***, do které je uložena pozice na navigační mřížce. Následně využívá metodu ***SamplePosition*** pro kontrolu, zdali je vygenerovaná náhodná pozice na navigační mřížce, v případě že ano tak je daná pozice předána do proměnné ***hit***. V případě že vygenerovaná náhodná pozice není na navigační mřížce tak metoda ***SamplePosition*** předá do proměnné ***hit*** nejbližší možnou pozici, která se na navigační mřížce nachází. Nakonec metoda vrátí ověřenou náhodnou pozici upravenou o ***Vector3***(0,1,0) pro změnu výšky.

9 VYUŽITÍ KONEČNÉHO AUTOMATU PRO VYTVOŘENÍ HERNÍ AI

V této kapitole bude navržena a implementována ukázka herní AI založena na principu konečného automatu.

9.1 Návrh konečného automatu

Pro ukázkové prostředí bylo vytvořené chování, kde agent náhodně bloudí v labyrintu do doby, dokud nezjistí, že zlatá kostka se od něho nachází ve stanovené vzdálenosti. Detekovanou kostku sebere a potom dále pokračuje v náhodném bloudění. Byl navržen konečný automat o třech stavech. Jeden počáteční a dva další, ve kterých se vykonává hlavní část chování. Tyto tři stavy využívané ukázkovou herní AI jsou ***Idle***, ***Wandering*** a ***Picking up***, viz Obr. 13 níže.



Obrázek 13. Grafický návrh konečného automatu.

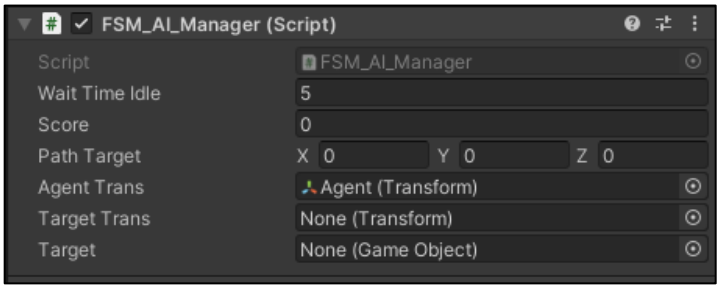
První a počáteční stav je ***Idle*** do kterého herní AI automaticky přejde při zapnutí ukázky. V tomto stavu bude herní AI kontrolovat daný čas od spuštění ukázky, až daný čas bude rovný nebo vyšší, než patnáct sekund, tak herní AI přejde do stavu ***Wandering***. Dále herní AI v tomto stavu kontroluje, jestli nebyla vytvořená instance zlaté kostky v určité vzdálenosti od agenta ovládaného herní AI. Když je kostka detekovaná tak místo přechodu do stavu ***Wandering***, přechází herní AI do stavu ***Picking up***.

Ve stavu ***Wandering*** bude herní AI náhodně volit body ve vytvořeném labyrintu do kterých agent herní AI dojde nejrychlejší cestu a kontrolovat, zda není zlatá kostka v určité vzdálenosti od agenta. V případě, že bude zlatá kostka v určité vzdálenosti od agenta, tak herní AI přejde do třetího stavu ***Picking up***. Jinak agent pokračuje v náhodné volbě cílů a přesunu do zvolených bodů.

Ve stavu ***Picking up*** nastaví herní AI agentovi trasu k dané zlaté kostce, přestane kontrolovat, zda jsou jiné zlaté kostky v určité vzdálenosti a začne kontrolovat, jestli je agent dostatečně blízko od pozice dané zlaté kostky. Až agent dorazí do dostatečně blízkosti, tak kostku odstraní a AI přejde zpět do stavu ***Wandering***.

9.2 Implementaci konečného automatu

Konečný automat využitý herní AI je implementován ve skriptu s názvem ***FSM_AI_Manager***, který je přidělen jako komponenta hernímu objektu ***Agent***.



*Obrázek 14. Komponent **FSM_AI_Manager** v herním objektu **Agent.***

Na obrázku 14 výše je vidět, jak vypadá komponenta ***FSM_AI_Manager*** v editoru enginu Unity a proměnné s veřejnou přístupností, které mohou být editovány přímo v editoru.

9.2.1 Vytvoření stavů

Pro vytvoření stavů a lepší přehlednost kódu byla využita enumerace s názvem ***FSMStates,*** viz Obr. 15 níže.

```
public enum FSMstates
{
    Idle = 1,
    Wandering,
    Picking_up
}
```

Obrázek 15. Zdrojový kód enumerace využité v konečném automatu.

Tato enumerace obsahuje tři stavy odpovídající návrhu v obrázku 15 výše. Prvnímu stavu ***Idle*** byla přidělena číslice jedna, aby celá enumerace nezačínala číslem nula a byla přehlednější.

9.2.2 Proměnné skriptu FSM_AI_Manager

Skript ***FSM_AI_Manager*** obsahuje proměnné pro uchovávání důležitých dat, se kterými herní AI dále pracuje, viz Obr. 16 níže.

```
private NavMeshAgent navMeshAgent;
private int currentState;
private int distance = 10;
private double timeOnStart;
private bool isThereATarget = false;
private bool thereIsNoPath = true;
private GameObject currentTarget;
private double timeMesureStart = 0;
private bool changedState = false;
private float timeMesure = 0;
private List<float> listOfTimeMesurments = new List<float>();
private int timesTimeMesured = 0;

public int waitTimeIdle = 15;
public int score = 0;
public Vector3 pathTarget;
public Transform agentTrans;
public Transform targetTrans;
public GameObject target;
```

*Obrázek 16. Zdrojový kód proměnných skriptu **FSM_AI_Manager.***

Prvních dvanáct proměnných má přístupnost nastavenou na privátní, aby nebylo možné je měnit z editoru Unity nebo z jiných skriptů. Dalších šest proměnných má přístupnost nastavenou na veřejnou, aby je bylo možné pozorovat a měnit v editoru Unity.

První proměnná ***navMeshAgent*** obsahuje komponentu ***NavMeshAgent*** využívanou pro nastavení cíle cesty a pohyb po vygenerované cestě do zvoleného cíle. Druhá proměnná ***currentState*** obsahuje aktuální stav konečného automatu v podobě celého čísla. Třetí proměnná ***distance*** obsahuje celé číslo využívané pro určení vzdálenosti možného náhodně vygenerovaného cíle od agenta herní AI. Čtvrtá proměnná ***isThereATarget*** obsahuje binární informaci, jestli herní AI detekuje zlatou kostku. Pátá proměnná ***thereIsNoPath*** obsahuje binární informaci, jestli agent našel cestu, kterou může použít ve stavu ***Wandering***. Šestá privátní proměnná ***currentTarget*** udržuje informaci o herním objektu, který je cílem ve stavu ***Picking_Up***. Následující dvě privátní proměnné obsahují data ohledně času mezi stavy a jestli byl stav změněn. Další proměnná s privátním přístupem je proměnná ***timeMesure***

využívaná na měření času vykonání konečného automatu. Následuje ji list ***listOfTimeMesurments*** uchovávající naměřené časy. Poslední proměnná s privátním přístupem je ***timesTimeMesured*** obsahující celkový počet všech měření.

Dalších šest proměnných s veřejným přístupem obsahují čas, jaký herní AI vyčká ve stavu ***Idle*** před tím, než přejde do stavu ***Wandering*** v proměnné ***waitTimeIdle***. Následující proměnná ***score*** udržuje hodnotu skóre, které získalo herní AI sebráním zlatých kostek. Další proměnná ***pathTarget*** udržuje pozici náhodně vygenerovaného cíle. Následující dvě proměnné obsahují transformační data agenta. Proměnné ***agentTrans*** udržuje informace ohledně jeho pozice, rotace a škály agenta. Následující proměnná ***targetTrans*** udržuje informace ohledně pozice, rotace a škály detekované zlaté kostky, která byla nastavena jako cíl herní AI. Poslední proměnná ***target*** obsahuje všechny informace herního objektu detekované zlaté kostky.

9.2.3 Metoda Awake

FSM_AI_Manager využívá několik metod ze třídy ***MonoBehaviour*** pro nastavení proměnných a periodické vykonávání metody ***FSM*** blíže popsané v kapitole **9.2.11**. První využitá metoda ***Awake*** je zobrazena na Obr. 17 níže.

```
// Awake is called when the script instance is being loaded
Unity Message | 0 references
private void Awake()
{
    navMeshAgent = GetComponent<NavMeshAgent>();
}
```

*Obrázek 17. Zdrojový kód metody **Awake** skriptu **FSM_AI_Manager.***

Tato metoda je volána při načtení skriptu, kdy získá komponentu ***NavMeshAgent*** využitou pro navigaci a pohyb agenta po navigační mřížce.

9.2.4 Metoda Start

Druhá metoda ze třídy ***MonoBehaviour*** je metoda ***Start,*** viz Obr. 18 níže.

```
// Start is called before the first frame update
Unity Message | 0 references
void Start()
{
    currentState = (int)FSM_states.Idle;
    timeOnStart = Time.time;
}
```

*Obrázek 18. Zdrojový kód metody **Start** skriptu **FSM_AI_Manager.***

Tato metoda je volána pouze před prvním zavoláním metody ***Update***. V této metodě je nastaven aktuální stav na ***Idle*** a také aktuální čas na začátku ukázky.

9.2.5 Metoda Update

Třetí metoda ze třídy ***MonoBehaviour*** je metoda ***Update,*** viz Obr. 19 níže.

```
void Update()
{
    FSM();
    timeMesure = Time.time;
}
```

*Obrázek 19. Zdrojový kód metody **Update** skriptu **FSM_AI_Manager.***

Metoda ***Update*** se volá každý snímek. V této metodě je volána metoda ***FSM,*** která obsahuje a vykonává navržený konečný automat pro popsané chování v kapitole **9.1 Návrh konečného automatu**. Také se nastavuje aktuální čas do proměnné ***timeMesure***.

9.2.6 Metoda OnGUI

Čtvrtá metoda ze třídy ***MonoBehavior*** je metoda ***OnGUI,*** viz Obr. 20 níže.

```
//Makes visible labe on the game screen
Unity Message | 0 references
private void OnGUI()
{
    var enumState = (FSM_states)currentState;
    string FSMData = enumState.ToString();
    GUILayout.Label($"<color='black'><size=50>{FSMData}</size></color>" + $"\n Score: {score}");
}
```

Obrázek 20. *Zdrojový kód metody **OnGUI** skriptu **FSM_AI_Manager.***

Tato metoda ovládá grafické uživatelské rozhraní přístupné v Unity. V tomto skriptu je metoda využita pro vytvoření dynamicky aktualizovaného textu v levém horním rohu ukázky konečného automatu, viz Obr. 21 níže.

Obrázek 21. Text zobrazující aktuální stav konečného automatu.

Tento vytvoření text informuje, v jakém stavu se konečný automat aktuálně nachází pomocí velkého černého textu a jaké skóre získalo herní AI pomocí menšího bílého textu.

9.2.7 Metoda OnApplicationQuit

Pátá metoda ze třídy ***MonoBehavior*** je metoda ***OnApplicationQuit***, která se volá jen při ukončení hry. Tato metoda obsahuje seřazení a vypsání všech naměřených časů do konzole editoru Unity.

9.2.8 Metoda SetRandomPath

Metoda ***SetRandomPath*** zobrazená na obrázku 22 níže, se používá pro nastavení náhodné cesty pro agenta ovládaného herní AI.

```
public void SetRandomPath()
{
    Vector3 randomDirection = (Random.insideUnitSphere * distance) + agentTrans.position;
    NavMeshHit hit;
    NavMesh.SamplePosition(randomDirection, out hit, distance, -1);
    thereIsNoPath = false;
    pathTarget = hit.position;
    navMeshAgent.SetDestination(hit.position);
    distance = Random.Range(10, 50);
}
```

*Obrázek 22. Zdrojový kód metody **SetRandomPath** skriptu **FSM_AI_Manager.***

Tato metoda je vytvořená úpravou metody ***RandomPosition*** použité pro nastavování místa instance zlaté kostky. ***SetRandomPath*** je odlišná tím, že náhodnou vzdálenost z rozsahu negeneruje na začátku, ale až na konci metody. Další rozdíl je, že mění proměnnou ***thereIsNoPath*** využívanou ve stavu ***Wandering*** pro kontrolu, zda existuje nějaká cesta. Poslední dva rozdíly jsou, že náhodný a ověřený bod, který se vyskytuje na navigační mřížce uloží do proměnné ***pathTarget*** a nastaví cestu agentovi.

9.2.9 Metoda OnTriggerStay

Metoda ***OnTriggerStay*** ze třídy ***MonoBehaviour*** zobrazena na obrázku 23, je využita pro detekci zlatých kostek v oblasti okolo agenta herní AI.

```
private void OnTriggerStay(Collider other)
{
    if (other.tag == "Gold Cube")
    {
        isThereATarget = true;
        targetTrans = other.transform;
        target = other.gameObject;
    }
}
```

*Obrázek 23. Zdrojový kód metody **OnTriggerStay** skriptu **FSM_AI_Manager**.*

Tato metoda se vykoná každý snímek pro každý objekt v okruhu okolo agenta. Metoda obsahuje podmínku kontrolující, jestli se jedná o zlatou kostku. V případě že ano, tak změní proměnnou ***isThereATarget*** na hodnotu ***true*** a tím informuje herní AI, že byla nalezena zlatá kostka v okruhu hledání. Nakonec uloží data ohledně daného objektu do dalších dvou proměnných.

9.2.10 Metoda DestroyCube

Metoda ***DestroyCube***, viz Obr. 24 níže, se používá pro mazání zlatých kostek a inkrementaci skóre.

```
//Method destroying cube and incrementing score
1 reference
public void DestroyCube(GameObject cube)
{
    Destroy(cube);
    score += 1;
}
```

*Obrázek 24. Zdrojový kód metody **DestroyCube** skriptu **FSM_AI_Manager.***

Tato metoda přebírá informace z argumentu ohledně herního objektu, který má smazat. Pro smazání předaného herního objektu využívá metodu ***Destroy*** poskytovanou enginem Unity pro mazání herních objektů. Nakonec inkrementuje skóre o jeden bod.

9.2.11 Metoda FSM

Metoda ***FSM*** obsahuje celou strukturu konečného automatu navrženého v kapitole **9.1**. Využívá příkazu ***switch*** pro rozhodování mezi chováním herní AI dle aktuálního stavu uchovávaného v proměnné ***currentState***.

```
case (int)FSMstates.Idle:

    if (isThereATarget == true)
    {
        // Debug.Log("Idle - found cube!!!!");
        navMeshAgent.SetDestination(targetTrans.position);
        currentTarget = target;
        timeMesureStart = Time.timeAsDouble;
        changedState = true;
        currentState = (int)FSMstates.Picking_up;
    }
    else if (Time.time - timeOnStart >= waitTimeIdle)
    {
        //Debug.Log("Idle - time has elapse, changing to wandering");
        changedState = true;
        timeMesureStart = Time.timeAsDouble;
        currentState = (int)FSMstates.Wandering;
    }
    Debug.Log($"Idle Time mesurring {Time.timeAsDouble - timeMesure}");
    break;
```

*Obrázek 25. Zdrojový kód stavu **Idle** skriptu **FSM_AI_Manager.***

První stav ***Idle***, viz Obr. 25 výše, je stav, ve kterém herní AI začíná při zapnutí ukázky. V tomto stavu AI čeká na detekci zlaté kostky, nebo než uplyne stanovený čas. Při detekci zlaté kostky je agentovi přidělena nejkratší trasa k detekované zlaté kostce, dále nastavena proměnná ***currentTarget*** na detekovanou kostku, je uložen aktuální čas před změnou stavu, také nastavena proměnná ***changedState*** na pozitivní hodnotu, a nakonec změněn stav na stav ***Picking_up***. V případě uplynutí stanoveného času je jen uložen čas před změnou, je nastavena proměnná ***changedState*** informující o změně stavu na pozitivní a stav je změněn na ***Wandering***.

```
case (int)FSMstates.Wandering:

    changedState = false;
    if (thereIsNoPath)
    {
        Debug.Log($"Wandering - Time from previous state: {Time.timeAsDouble - timeMesureStart}");
        SetRandomPath();

    }
    else if (Vector3.Distance(agentTrans.position, pathTarget) <= 2)
    {
        // Debug.Log("Wandering - set another path");
        SetRandomPath();
    }

    if (isThereATarget == true)
    {
        //Debug.Log("Wandering - found cube!!!!!!!");
        navMeshAgent.SetDestination(targetTrans.position);
        currentTarget = target;
        changedState = true;
        timeMesureStart = Time.timeAsDouble;
        currentState = (int)FSMstates.Picking_up;
    }
    Debug.Log($"Wandering Time mesurring {Time.timeAsDouble - timeMesure}");
    break;
```

*Obrázek 26. Zdrojový kód stavu **Wandering** skriptu **FSM_AI_Manager.***

Ve druhém stavu ***Wandering*** (Obr. 26 výše) herní AI zjišťuje, jestli agent už má nastavenou cestu a v případě že cestu ještě nemá (což je pravdou pouze při přechodu z jiného stavu), tak vypíše čas uplynulý přechodem mezi stavy do konzole a zavolá metodu ***SetRandomPath*** pro nastavení náhodné cesty. V případě, že cestu agent už má přidělenou, tak dále kontroluje vzdálenost agenta od cíle cesty, nebo jestli není zlatá kostka v určité vzdálenosti od agenta. V případě, že agent je blízko cíle nastavené cesty, tak znovu zavolá funkci ***SetRandomPath*** pro nastavení nové cesty. Nakonec v případě, že je detekována zlatá kostka v určité vzdálenosti, tak se vykonají stejné akce jako v předešlém stavu ***Idle*** při detekci zlaté kostky.

```
case (int)FSMstates.Picking_up:

    if (changedState == true)
    {
        Debug.Log($"Picking_up.  Time from previous state: {Time.timeAsDouble - timeMesureStart}");
    }

    changedState = false;
    if (Vector3.Distance(agentTrans.position, currentTarget.transform.position) <= 2)
    {
        //Debug.Log("Picking up - touched the cube, changing to wandering");
        DestroyCube(currentTarget);
        isThereATarget = false;
        thereIsNoPath = true;
        changedState = true;
        timeMesureStart = Time.timeAsDouble;
        currentState = (int)FSMstates.Wandering;
    }
    Debug.Log($"Picking up Time mesurring {Time.timeAsDouble - timeMesure}");
    break;
```

*Obrázek 27. Zdrojový kód stavu **Picking_up** skriptu **FSM_AI_Manager.***

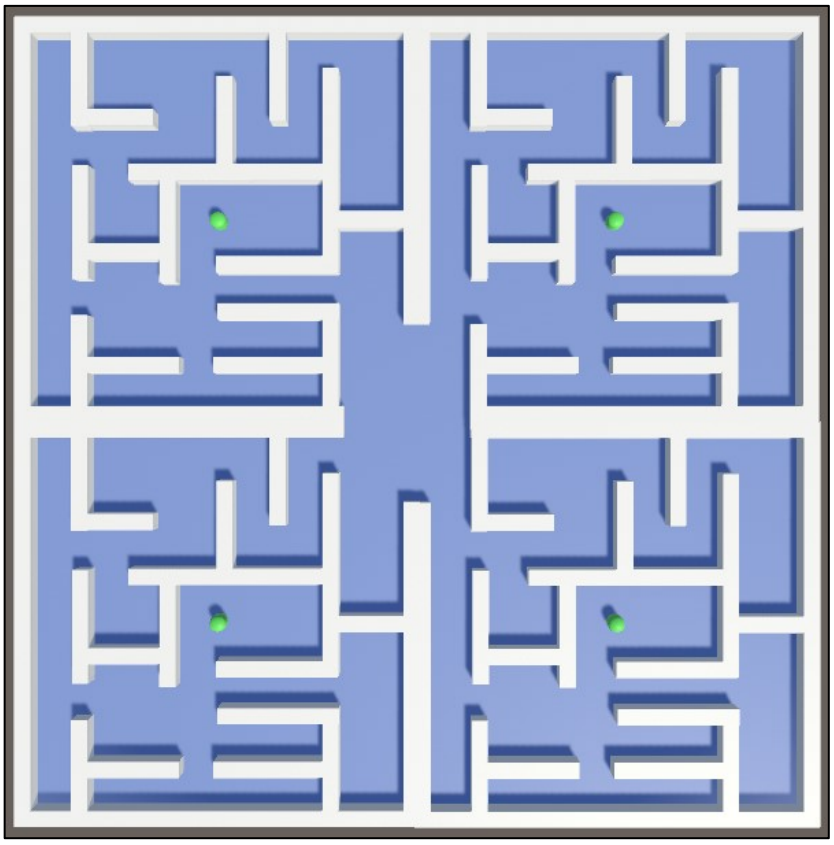
Třetí a poslední stav ***Picking_up***, viz Obr. 27 výše, obsahuje chování pro funkci sebrání detekované zlaté kostky. Na začátku je zkontrolováno, jestli před vykonáním této metody bylo herní AI v jiném stavu. V případě že ano, tak je zkontrolován čas přechodu mezi stavy. Dále se pokračuje kontrolou vzdálenosti agenta herní AI od cíle, na kterém se nachází zlatá kostka. V případě, že agent je dostatečně blízko k cíli své cesty, a tím pádem i vybrané zlaté kostce, tak danou zlatou kostku odstraní a změní proměnou využívanou pro určování, jestli existuje zlatá kostka v určitém rozsahu od agenta ***isThereATarget*** na negativní hodnotu. Dále nastaví proměnné ***thereIsNoPath*** oznamující, že agent aktuálně nemá cestu a ***changedState*** oznamující, že byl změněn stav na pozitivní hodnotu. Nakonec změní stav zpět na ***Wandering***.

Na konci celé metody ***FSM*** je měřen čas nutný k vykonání celé metody.

Ukázku použití konečného automatu ilustruje následující video: https://youtu.be/pIA-Rw9nPDI

9.3 Měření času provedení konečného automatu

Byla měřena doba nutná pro vykonání celé funkce konečného automatu v ukázkovém prostředí zobrazeném na obrázku 6 výše. Toto měření bylo opakováno deset tisíckrát. Výsledky měření jsou, že nejkratší čas nutný pro vykonání byl 0 sekund a nejvyšší je 0,3333, ale tyto časy nebyly přesné z důvodu načítání dalších podsystémů na začátku ukázky, pro získání přesnějších časů bylo prvních šest měření odstraněno. Po odstranění nový minimální naměřený čas je 0,0042 sekund a maximální naměřený čas je 0,0661. Průměrná hodnota všech naměřených časů je 0,0055 sekund a medián naměřených hodnot zase 0,0053 sekund.



Obrázek 28. Čtyři kopie ukázkového prostředí a agentů herních AI v ukázce konečného automatu.

Pro přesnější měření dopadu konečného automatu na výkon nutný pro vykonání celé ukázky byly vytvořeny čtyři kopie ukázkového prostředí a agentů herní AI v jednom projektu Unity, viz Obr. 28 výše. Výsledky po deseti tisících měření jsou že minimální naměřený čas provedení je 0,0044 sekund a maximální naměřený čas je 0,0507 sekund. Průměrná naměřená hodnota dosahuje 0,0064 sekund a medián hodnot pak představuje 0,0063 sekund.

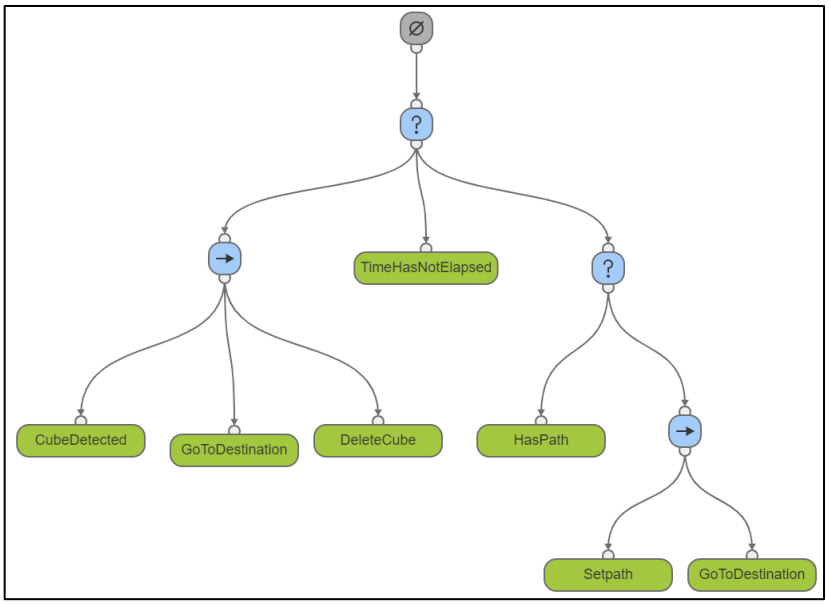
Dále byly ještě vytvořeny verze ukázky se šestnácti kopiemi ukázkového prostoru a verze se třiceti dvěma kopiemi pro porovnání růstu naměřeného času provedení algoritmu vůči behaviorálnímu stromu v kapitole 11. V ukázce se šestnácti kopiemi ukázkových prostorů a agentů herních AI je minimální naměřený čas 0,0052 sekund a maximální naměřené čas 0,0627 sekund, průměrný naměřený čas dosáhl 0,0072 sekund a medián hodnot naměřených časů zase 0,0071. V ukázce se třiceti dvěma kopiemi ukázkových prostorů a agentů herních AI je minimální naměřený čas 0,0063 sekund a maximální naměřený čas dosáhl 0,0558 sekund. Průměrný naměřený čas v této upravené ukázce ovšem představoval 0,0079 sekund a medián naměřených časů stoupl na 0,0077 sekund. Podrobné výsledky porovnání rychlostí uvedených algoritmů tabelární formou jsou pak uvedeny v kapitole 11 dále.

10 VYUŽITÍ BEHAVIORÁLNÍHO STROMU PRO VYTVOŘENÍ HERNÍ AI

V této kapitole bude navržena a implementována ukázka herní AI založena na principu behaviorálního stromu.

10.1 Návrh behaviorálního stromu

Pro vytvoření chování co nejvíce se podobajícímu chování z předešlé ukázky byl navržen behaviorální strom s chováním srovnatelným s chováním konečného automatu z předešlé ukázky. Toto chování na rozdíl od chování konečného automatu může být kdykoliv přerušeno, když bude detekována zlatá kostka v nastaveném rozsahu od agenta herní AI. Při přerušení detekcí zlaté kostky se potom hlavním cílem herní AI stane sebrání této kostky. Uvedené chování bylo pro druhou ukázku navrženo behaviorálním stromem o třech větvích a šesti originálních listech.



Obrázek 29. Návrh behaviorálního stromu.

Obrázek 29. výše ilustruje návrh behaviorálního stromu pro vytvoření totožného chování herní AI jako bylo ukázáno v předešlé ukázce za pomoci konečného automatu. Tento návrh behaviorálního stromu začíná přepínačem, který vybírá ze tří dále následujících uzlů.

Kořenový přepínač jako první vykoná indikátor řady, který je na uvedeném návrhu zobrazen jako uzel nejvíce vlevo obsahující šipku, pod kterým následují listy tvořící chování herní AI pro sebrání zlaté kostky srovnatelné se stavem ***Picking up*** konečného automatu v předešlé ukázce. Indikátor řady začne vykonáním prvního listu ***CubeDetected*** obsahující podmínku

dotazující se, jestli byla detekována zlatá kostka ve vývojářem dané vzdálenosti od agenta herní AI. V případě že ano, tak nastaví cíl na detekovanou zlatou kostku a vrátí pozitivní informaci umožňující indikátoru řady vykonání následujících listů ***GoToDestination*** a ***DeleteCube***. List ***GoToDestination*** nastavuje agentovi herní AI cestu k dané zlaté kostce. Poslední list ***DeleteCube*** odstraňuje danou zlatou kostku v případě, že agent je v dostatečné blízkosti od dané zlaté kostky. Když všechny listy vrátí pozitivní informaci ohledně stavu vykonání, tak indikátor také vrátí pozitivní informaci a kořenový přepínač přestane hledat pozitivní uzel.

V případě, že list ***CubeDetected*** vrátí negativní informaci ohledně stavu vykonání, tak celý indikátor řady vrátí negativní informaci ohledně stavu vykonání a kořenový přepínač vykoná list ***TimeHasNotElapsed*** srovnatelný se stavem ***Idle*** konečného automatu předešlé ukázky. List ***TimeHasNotElapsed*** vrací pozitivní informaci v případě, že ještě neuběhl vývojářem nastavený čas a kořenový přepínač přestává v hledání pozitivního uzlu. V případě, že daný čas uběhl, tak potom vrací negativní informaci ohledně vykonání a kořenový přepínač pokračuje v hledání pozitivního uzlu.

Poslední uzel je přepínač obstarávající chování „potulování“ herní AI srovnatelné se stavem ***Wandering*** předešlé ukázky. Tento přepínač vybere ze dvou uzlů. První list se vykoná v případě, že agent má už nastavenou cestu. Druhý uzel je indikátor řady vykonávající dva listy. První vykonávaný list ***SetPath*** nastaví náhodný bod jako cíl agenta. Druhý vykonávaný list je znovu využitý list ***GoToDestination*** nastavující cestu agentu herní AI.

10.2 Vytvoření uzlů behaviorálního stromu

Dále byli vytvořeny třídy obsahující chování listů, přepínačů a indikátorů řady. Třídy ***Node***, ***Selector*** a ***Sequence*** byly navrženy a vytvořeny podle video tutoriálu [30].

10.2.1 Třída Node

První třída převzatá z video tutoriálu [30] obsahuje abstraktní třídu ***Node***, nastavení možných stavů vykonání a vytvořenou abstraktní metoda ***Evaluate***. Z této třídy všechny ostatní třídy uzlů stromu dědí a díky tomu musí vytvořit vlastní implementaci abstraktní metody ***Evaluate***, která určuje chování daného uzlu.

10.2.2 Třída Selector

Druhá třída převzatá z video tutoriálu [30] obsahuje implementaci přepínače ve zděděné metodě ***Evaluate***. Využívá list obsahující následující uzly, ze kterých daný přepínač vybírá. Pro implementaci logiky přepínače se vyžívá příkaz ***switch*** reagující na informace ohledně stavu uzlů obsažených v listu uzlů. Při získání pozitivní informace ***Success*** nebo informace o stálém vykonávání ***Running***, vrací přepínač pozitivní informaci ***Success*** předchozímu uzlu a přestává ve volání metod ***Evaluate*** následujících uzlů. Na negativní informaci ***Failure*** přepínač nereaguje a pokračuje ve volání metod ***Evaluate*** následujících uzlů.

10.2.3 Třída Sequence

Třetí a poslední převzatá třída z video tutoriálu [30] obsahuje implementaci indikátoru řady ve zděděné metodě ***Evaluate***. Jako předešlá třída ***Selector*** také udržuje list následujících uzlů. Dále opět využívá příkaz ***switch*** pro implementaci logiky indikátoru řady. Indikátor řady na získání pozitivní informace ***Success*** nereaguje a pokračuje ve volání metod ***Evaluate*** následujících uzlů. Pří získaní negativní informace ***Failure,*** indikátor řady přestává ve volání metod ***Evaluate*** následujících uzlů a vrací negativní informaci ***Failure***. Při získání informace o stálém vykonávání nastaví proměnné ***isAnyNodeRunning*** pozitivní informaci a volá další metody ***Evaluate*** následujících metod.

10.2.4 Třída CubeDetected

První třída ***CubeDetected*** vytvořená pro vlastní ukázku – obsahuje podmínku kontrolující, zda se zlatá kostka nachází v určité vzdálenosti od agenta herní AI. V případě, že ano a herní AI nemá nastavenou cílovou kostku, tak nastaví cílovou kostku pomocí vlastnosti ***SelectedCubeTargetGameObject*** a také nastaví cílový bod cesty ke zlaté kostce. Další indikátory – jestli má herní AI už cíl nebo cestu potom nastaví na pozitivní informaci ***true***. Nakonec vrátí pozitivní informaci ***Success***. V případě, že zlatá kostka není detekovaná v určité vzdálenosti od agenta, tak vrací negativní informaci ***Failure***.

10.2.5 Třída DeleteCube

Druhá třída ***DeleteCube***, vytvořená pro vlastní ukázku – obsahuje podmínku kontrolující vzdálenost od cílové zlaté kostky. V případě že agent je dostatečně blízko od cílové kostky tak danou zlatou kostku smaže zavoláním funkce ***DestroyCube*** ze třídy ***AIController.*** Dále nastaví proměnné informující o stavu cesty, jestli agent má cíl a zda je detekovaná zlatá kostka, na negativní hodnotu ***false***. Nakonec vrátí pozitivní informaci ***Success***.

10.2.6 Třída GoToDestination

Třetí třída ***GoToDestination*** vytvořená pro vlastní ukázku zobrazena na obrázku 30 níže, se využívá pro nastavení pohybu do zvoleného cíle cesty.

```
public override NodeState Evaluate()
{
    Debug.Log($"Time to GoToDestination: {Time.timeAsDouble - _ai.timeMesure}");
    _navMeshAgent.SetDestination(_ai.PathTarget);
    return NodeState.Success;
}
```

*Obrázek 30. Metoda **Evaluate** třídy **GoToDestination.***

Tato metoda nastavuje cestu agentovi herní AI pomocí komponenty ***NavMeshAgent*** a voláním metody této komponenty ***SetDestination***. Nakonec vrací pozitivní informaci ***Success***.

10.2.7 Třída HasPath

Čtvrtá třída ***HasPath***, vytvořená pro vlastní ukázku – obsahuje podmínky kontrolující, zda agent herní AI nemá nastavenou cestu a jak je vzdálený od cíle své cesty. V případě že agent nemá nastavenou cestu, tak vrací negativní informaci ***Failure.*** Druhá podmínka kontroluje, jestli není agent až v moc velké blízkosti od cíle cesty. V případě, že je agent dostatečně vzdálený od cíle cesty, tak vrací pozitivní informaci ***Success***. V případě, že všechny předešlé podmínky byly vyhodnoceny negativně, tak nastavuje proměnnou informující, zdali má agent cestu pomocí vlastnosti ***AgentHasPath*** na negativní hodnotu ***false*** a vrací negativní informaci ***Failure***.

10.2.8 Třída SetPath

Pátá třída ***SetPath***, vytvořená pro vlastní ukázku, viz Obr. 31 níže, je využita pro přidělení náhodného cíle cesty.

```
public override NodeState Evaluate()
{
    //Debug.Log("Setting path");
    Debug.Log($"Time to SetPath: {Time.timeAsDouble - _ai.timeMesure}");
    _ai.BTDataString = "Wandering";
    _ai.AgentHasPath = true;
    int distance = Random.Range(_ai.randomDistanceLow, _ai.randomDistanceHigh);
    Vector3 randomDirection = (Random.insideUnitSphere * distance) + _ai.trans.position;
    NavMeshHit hit;
    NavMesh.SamplePosition(randomDirection, out hit, distance, -1);
    _ai.PathTarget = hit.position;

    _ai.timeMesureColective += Time.timeAsDouble - _ai.timeMesure;
    _ai.timesTimeMesured += 1;
    return NodeState.Success;
}
```

*Obrázek 31. Metoda **Evaluate** třídy **SetPath.***

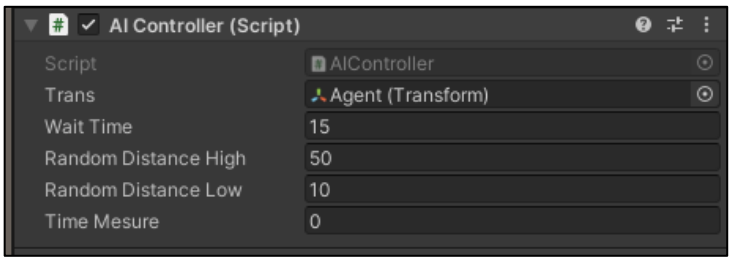
Tato třída nastavuje cíl cesty agenta na náhodný bod vytvořený podobnou metodou zvolení náhodného bodu využitou v kapitole **9.2.8.** Dále nastavuje text zobrazený v ukázce na textový řetězec „Wandering“. Nakonec vrací pozitivní informaci ***Success***.

10.2.9 Třída TimeHasNotElapsed

Šestá třída ***TimeHasNotElapsed***, vytvořená pro vlastní ukázku obsahuje celé chování srovnatelné se stavem ***Idle*** konečného automatu předešlé ukázky. Dané chování je vytvořeno podmínkou kontrolující, jestli uběhl vývojářem nastavený čas. V případě že ano tak vrací negativní informaci ***Failure***. V případě že ne, tak nastavuje text zobrazený v ukázce na textový řetězec „Idle“ a vrací pozitivní informaci ***Success***.

10.3 Implementace behaviorálního stromu

Behaviorální strom využívaný herní AI je sestaven a implementován ve skriptu s názvem ***AIController,*** který je přidělen jako komponenta hernímu objektu ***Agent,*** využívanému jako agent herní AI.



*Obrázek 32. Komponenta **AIController** v herním objektu **Agent.***

Na obrázku 32 výše je vidět, jak vypadá komponenta ***AIController*** v editoru herního enginu Unity a proměnné s veřejnou přístupností, které mohou být editovány přímo v tomto editoru.

10.3.1 Proměnné skriptu AIController

Skript ***AIController*** obsahuje proměnné zobrazené na obrázku 33 níže, prvních pět proměnných mají veřejný přístup, aby byly přístupné a měnitelné v editoru Unity.

```
public Transform trans;
public float waitTime = 15f;
public int randomDistanceHigh = 50;
public int randomDistanceLow = 10;
public float timeMesure = 0.0f;
private NavMeshAgent navMeshAgent;
private Node rootNode;
private float actualStartTime;
private bool isThereACube = false;
private bool agentHasATarget = false;
private bool agentHasPath = false;
private Vector3 pathTarget;
private GameObject cubeTargetGameObject;
private GameObject selectedCubeTargetGameObject;
private int score = 0;
private string BTData;

public List<float> listOfTimeMesurments = new List<float>();
private int timesTimeMesured = 0;
```

*Obrázek 33. Zdrojový kód proměnných skriptu **AIController.***

První proměnná ***trans*** obsahuje transformační data (pozici, rotaci a škálu) agenta herní AI. Druhá proměnná ***waitTime*** obsahuje čas,po který AI vyčká na začátku ukázky. Třetí a čtvrtá proměnná obsahují hranice náhodného generování číslic pro volbu náhodných cest. Pátá a poslední proměnná s veřejným přístupem ***timeMesure*** je využita pro měření času potřebného pro vlastní chod vytvořené herní AI.

První proměnná s privátním přístupem ***navMeshAgent*** obsahuje komponenta ***NavMeshAgent*** agenta herní AI, která se využívá pro nastavení cest. Následující proměnná ***rootNode*** obsahuje kořenový uzel, ze kterého začíná průchod behaviorálním stromem. Dále proměnná ***actualStartTime*** obsahuje čas nastavený na začátku ukázky, využívaný pro kontrolu, zdali uplynul čas nastavený v proměnné ***waitTime.*** Další tři proměnné ***isThereACube***, ***agentHasATarget*** a ***agentHasPath*** obsahují informace, zdali byla detekována zlatá kostka, jestli má agent herní AI nastavenou cílovou zlatou kostku a zdali má agent nastavenou cestu. Proměnná ***pathTarget*** obsahuje náhodně zvolený bod na navigační mřížce využitý jako cíl cesty agenta herní AI. Další dvě proměnné ***cubeTargetGameObject*** a ***selectedCubeTargetGameObject*** obsahují data ohledně detekovaných kostek. Proměnné ***cubeTargetGameObject*** je přidělená každá zlatá kostka pokaždé, když je detekována. Proměnné ***selectedCubeTargetGameObject*** je přidělená zlatá kostka jen když se ji herní AI rozhodne sebrat. Následující

dvě proměnné ***score*** a ***BTData*** obsahují data zobrazená na obrazovce ukázky. Dále je využit list ***listOfTimeMesurments*** pro uchování naměřených časů získaných měřením času vykonání vlastního behaviorálního stromu. Poslední proměnná ***timesTimeMesured*** obsahuje informaci o počtu měření.

10.3.2 Vlastnosti skriptu AIController

Skript ***AIController*** využívá osm vlastností umožňující přístup k proměnným s privátním přístupem ze tříd nacházejících se mimo skript ***AIController***, např. listům behaviorálního stromu.

```
public bool AgentHasATarget
{
    get { return agentHasATarget; }
    set { agentHasATarget = value; }
}
2 references
public bool IsThereACube
{
    get { return isThereACube; }
    set { isThereACube = value; }
}
```

*Obrázek 34. Vlastnosti **AgentHasATarget** a **IsThereACube** skriptu **AIController.***

Mezi tyto vlastnosti patří ***AgentHasATarget***, viz Obr. 34 výše, která umožnuje přístup k proměnné ***agentHasATarget*** udržující informaci, jestli agent herní AI má již zvolenou cílovou zlatou kostku. Nebo také vlastnost ***IsThereACube,*** viz Obr. 34 výše, umožňující přístup k proměnné ***isThereACube***, která udržuje informaci, jestli byla detekovaná zlatá kostka v okolí agenta. Vlastnost ***SelectedTargetGameObject*** umožnuje přístup k proměnné ***selectedTargetGameObject*** udržující informace ohledně zvolené zlaté kostky. Následující vlastnost ***CubeTargetGameObject*** umožnuje jen přečtení proměnné ***cubeTargetGameObject***, udržující informace ohledně detekované zlaté kostky. Následující dvě vlastnosti ***AgentHasAPath*** a ***PathTarget*** umožnují přístup k proměnným ***agentHasAPath*** a ***pathTarget*** udržujícím, jestli agent má již zvolenou cestu a daný cíl. Další vlastnost ***BTDataString*** umožnuje přístup k proměnné ***BTData*** udržující řetězec znaků zobrazovaný metodou ***OnGUI*** popsanou níže. Poslední vlastnost ***TimeMesured*** umžnuje přístup k proměnné ***timesTimeMesured*** udržující počet již provedených měření.

10.3.3 Metoda Awake

AIController využívá několik metod ze třídy ***MonoBehaviour*** pro nastavení několika proměnných a periodické vykonávání metody ***Evaluate*** kořenového uzlu. První využitá metoda ***Awake*** je zobrazena na Obr. 35 níže.

```
private void Awake()
{
    navMeshAgent = GetComponent<NavMeshAgent>();
}
```

*Obrázek 35. Zdrojový kód metody **Awake** skriptu **AIController.***

Tato metoda je volána při načtení skriptu a využívá se k získání komponentu ***NavMeshAgent*** využitou pro navigaci a pohyb agenta po navigační mřížce.

10.3.4 Metoda Start

Druhá metoda ze třídy ***MonoBehaviour*** je metoda ***Start,*** viz Obr. 36 níže.

```
void Start()
{
    actualStartTime = Time.time;
    CunstuctBT();
    timeMesure = Time.time;
}
```

*Obrázek 36. Zdrojový kód metody **Start** skriptu **AIController.***

Tato metoda se vykoná pouze před prvním vykonáním metody ***Update***. V této metodě je nastavený aktuální čas využitý při čekání herní AI ve třídě ***TimeHasNotElapsed*** a zavolaná metoda ***CunstructBT***, která sestaví behaviorální strom. Metoda končí nastavením aktuálního stavu do proměnné ***timeMesure*** využité na měření času vykonání behaviorálního stromu.

10.3.5 Metoda Update

Třetí metoda ze třídy ***MonoBehaviour*** je metoda ***Update***, viz Obr. 37 níže.

```
void Update()
{
    rootNode.Evaluate();
    timeMesure = Time.time;

}
```

*Obrázek 37. Zdrojový kód metody **Update** skriptu **AIController.***

Tato metoda se vykonává každý snímek. Zde je volaná metoda ***Evaluate*** kořenového uzlu a nastavená proměnná ***timeMesure*** využitá pro měření času vykonání behaviorálního stromu.

10.3.6 Metoda OnGUI

Čtvrtá metoda ze třídy ***MonoBehavior*** v tomto skriptu je metoda ***OnGUI*** zobrazená na obrázku 38 níže.

```
private void OnGUI()
{
    GUILayout.Label($"<color='black'><size=50>{BTData}</size></color>" + $"\n Score: {score}");
}
```

*Obrázek 38. Zdrojový kód metody **OnGUI** skriptu **AIController.***

Tato metoda ovládá grafické uživatelské rozhraní přístupné v Unity. V tomto skriptu je tato metoda využita pro vytvoření dynamicky aktualizovaného textu v levém horním rohu ukázky behaviorálního stromu.

10.3.7 Metoda OnApplicationQuit

Pátá metoda ze třídy ***MonoBehavior*** je metoda ***OnApplicationQuit***, která se volá jen při ukončení hry. Tato metoda obsahuje seřazení a vypsání všech naměřených časů do konzole editoru Unity.

10.3.8 Metoda ConstructBT

Navržení uvedené metody ***ConstructBT***, (viz Obr. 39 níže), bylo inspirováno video tutoriálem využitým při vytvoření tříd ***Node***, ***Selector*** a ***Sequence*** [30].

```
private void CunstuctBT()
{
    //creation of nodes
    TimeHasNotElapsed idle = new TimeHasNotElapsed(this, waitTime, actualStartTime);
    HasPath hasPath = new HasPath(this, trans);
    SetPath setPath = new SetPath(this);
    GoToDestination goToDestination = new GoToDestination(this, navMeshAgent);
    CubeDetected cubeDetected = new CubeDetected(this);
    DeleteCube deleteCube = new DeleteCube(this, trans);

    //creation of selectors and sequences
    Sequence doesntHavePath = new Sequence(this,new List<Node> {setPath,goToDestination });
    Selector wander = new Selector(this, true,new List<Node> {hasPath, doesntHavePath });
    Sequence getCube = new Sequence(this,new List<Node> {cubeDetected, goToDestination, deleteCube });

    //creating of the root node
    rootNode = new Selector(this,false,new List<Node> { getCube, idle, wander });
}
```

*Obrázek 39. Zdrojový kód metody **ConstuctBT** skriptu **AIController.***

V první části této metody jsou vytvořeny instance listů z již dříve zmíněných tříd v kapitole **10.2**. Dále, v druhé časti jsou vytvořeny instance přepínačů a indikátorů řad s listy všech uzlů, jež následují tyto přepínače a indikátory. V poslední části je nastaven kořenový uzel na instanci přepínače s listem obsahující počáteční uzly větví behaviorálního stromu.

10.3.9 Metoda OnStayTrigger

Metoda ***OnTriggerStay*** ze třídy ***MonoBehaviour***, viz Obr. 40 níže, je využita pro detekci zlatých kostek v oblasti okolo agenta herní AI.

```
private void OnTriggerStay(Collider other)
{
    if (other.tag == "Gold Cube")
    {
        isThereACube = true;
        cubeTargetGameObject = other.gameObject;
    }
}
```

*Obrázek 40. Zdrojový kód metody **OnStayTrigger** skriptu **AIController.***

Tato metoda se volá každý snímek pro každý objekt v okruhu kolem agenta. Metoda obsahuje podmínku kontrolující, jestli se jedná o zlatou kostku, v případě že ano, tak změní proměnnou ***isThereACube*** na hodnotu ***true*** a tím informuje herní AI, že byla nalezena zlatá kostka v okruhu hledání. Nakonec uloží data ohledně daného objektu do další proměnné ***cubeTargetGameObject***.

10.3.10Metoda DestroyCube

Metoda ***DestroyCube***, (viz Obr. 41 níže), se používá pro mazání zlatých kostek a inkrementaci skóre.

```
public void DestroyCube(GameObject cube)
{
    Destroy(cube);
    score += 1;
}
```

*Obrázek 41. Zdrojový kód metody **DestroyCube** skriptu **AIController.***

Tato metoda přebírá informace z argumentu ohledně herního objektu, který má smazat. Pro smazání předaného herního objektu využívá metodu ***Destroy*** poskytovanou enginem Unity pro mazání herních objektů. Nakonec inkrementuje skóre o jeden bod.

Ukázku použití behaviorálního stromu ilustruje následující video:
https://youtu.be/aGOHSb3GQaE

10.4 Měření času provedení behaviorálního stromu

Byla měřena doba nutná pro vykonání celé funkce behaviorálního stromu v ukázkovém prostředí zobrazeném na obrázku 6 výše. Toto měření bylo opakováno deset tisíckrát. Výsledky měření jsou, že nejkratší čas nutný pro vykonání je 0 sekund a nejvyšší je 0,333, ale tyto časy nebyly přesné z důvodu načítání dalších podsystémů na začátku ukázky, pro získání přesnějších časů bylo prvních šest měření odstraněno. Po odstranění nový minimální naměřený čas stoupl na 0,0038 sekund a maximální naměřený čas klesl na 0,0497. Průměrná hodnota všech naměřených časů dosahuje 0,0054 sekund a medián hodnot je 0,0052 sekund.

Pro přesnější měření dopadu behaviorálního stromu na výkon nutný pro kompletace celé ukázky byly opět vytvořeny čtyři kopie ukázkového prostředí a agentů herní AI v jednom projektu Unity, (viz Obr. 28 výše). Výsledky po deseti tisících měření jsou, že minimální naměřený čas je 0,0045 sekund a ten maximální naměřený čas dosáhl 0,0416 sekund. Průměrná naměřená hodnota stoupla na 0,0066 sekund a medián hodnot zase na 0,0061 sekund.

Dále byly stejně jako u konečného automatu vytvořeny verze ukázky se šestnácti kopiemi ukázkového prostoru a verze ukázky se třiceti dvěma kopiemi pro podrobnější porovnání rychlosti obou algoritmů v následující kapitole. V ukázce se šestnácti kopiemi ukázkových prostorů a agentů herních AI je minimální naměřený čas 0,0068 sekund a maximální naměřené čas 0,0512 sekund, průměrný naměřený čas je 0,0100 sekund a medián hodnot naměřených časů stoupl 0,0093. V ukázce se třiceti dvěma kopiemi ukázkových prostorů a agentů herních AI je minimální naměřený čas 0,0091 sekund a maximální naměřený čas je 0,0548 sekund. Průměrný naměřený čas v této upravené ukázce ovšemže představoval 0,0131 sekund a medián naměřených časů dosáhl 0,0122 sekund. Podrobné výsledky porovnání rychlosti uvedených algoritmů tabelární formou jsou pak uvedeny v následující části – kapitole 11 dále.

11 POROVNÁNÍ VYUŽITÝCH ALGORITMŮ

V této kapitole budou návrhy a implementace popsané v předchozích dvou kapitolách navzájem porovnány z několika hledisek.

11.1 Návrh algoritmů

Návrh konečného algoritmu byl jednodušší na vytvoření a vysvětlení chování než návrh behaviorálního stromu. V návrhu konečného algoritmu byly podmínky pro změnu stavů zřejmé a viditelné již na první pohled z popisu přechodů, přičemž podmínky zamezující či povolující vykonání určité větve behaviorálního stromu musely být vyčteny buď z popisku přepínače a indikátoru řady nebo z názvu listů. V návrhu behaviorálního stromu bylo možné znovu využívat již jednou navrženého chování pomocí opětovného využití hotového listu. V konečném automatu musel být každý přechod definován již při návrhu. V behaviorálním stromu bylo chování, které v konečném automatu muselo mít přechod z každého stavu, zařazeno zcela na levou stranu stromu, aby bylo vyhodnoceno již před ostatním chováním.

11.2 Implementace algoritmů

Implementace konečného automatu byla jednoduchá a bezproblémová, díky využití standardního příkazu ***switch*** jazyka C#. Implementace behaviorálního stromu byla naopak zdlouhavá, kvůli nutnosti vytvořit každý list a pak dále nutnosti správně sestavit navržený behaviorálního stromu z těchto listů. Ovšemže behaviorální strom na rozdíl od konečného automatu, zase umožnuje jednodušší rozšiřování chování pomocí jednoduchého přidání uzlu do již implementovaného algoritmu.

11.3 Rychlost algoritmů

Rychlost vykonání konečného automatu a behaviorálního stromu byla na provedených ukázkách s využitím pouze jedné instance herní AI srovnatelná (v řádu jednotek milisekund). Se zvyšováním složitosti ale rozdíl roste. Při využití čtyř, šestnácti a třiceti dvou kopií herní AI a ukázkového prostředí pak při vlastním měření byl nárůst času u behaviorálního stromu obecně vyšší než u konečného automatu. Ale i v těchto případech byly časy poměrně srovnatelné (v řádu jednotek až desítek). Ovšem můžeme předpokládat, že se zvyšováním složitosti by se rozdíl ještě zvětšoval. Rámcové srovnání provedených měření je uvedeno v tabulce 1 níže.

Pro veškerá provedená měření byl využit počítač s operačním systémem Windows 10, procesorem AMD Ryzen 7 3700U s Radeon Vega Mobile GFx 2.3 GHz a operační pamětí 8 GB.

	Konečný automat				Behaviorální strom			
Počet kopií	Min [s]	Průměrný čas [s]	Medián [s]	Max [s]	Min [s]	Průměrný čas [s]	Medián [s]	Max [s]
1	0,0042	0,0055	0,0053	0,0661	0,0038	0,0054	0,0052	0,0497
4	0,0044	0,0064	0,0063	0,0507	0,0045	0, 0066	0, 0061	0,0416
16	0,0052	0,0072	0,0071	0,0627	0,0068	0, 0100	0,0093	0,0512
32	0,0063	0,0079	0,0077	0,0558	0,0091	0, 0131	0,0122	0,0548

Tabulka 1. Naměřené časy nutné k vykonání vlastního konečného automatu a behaviorálního stromu.

Konečný automat je obecně jednodušší na návrh a implementaci než behaviorální strom. Dále konečný automat vykazuje nižší čas nutný na vykonání než behaviorální strom při využití více kopií herních AI. Behaviorální strom na rozdíl od konečného automatu je ale jednodušší na škálování, umožnuje znovu využití již jednou navrženého chování pomocí opětovného použití již hotového listu. Při měření času nutného k vykonání navržených AI se ukazuje, že při využití jen jednoho herního AI jsou konečný automat a behaviorální strom co se rychlosti týče srovnatelné. Při využití více herních AI je ale nárůst naměřeného času u behaviorálního stromu o něco vyšší.

ZÁVĚR

Bakalářská práce s názvem Analýza herní AI s implementací v herním enginu Unity se obecně zabývá základním přehledem v oblasti herní AI a dále vzájemným porovnáním dvou běžně používaných algoritmů pro rozhodování v jednoduché implementační ukázce.

Základní přehled herní AI byl vytvořen literární rešerší, popisem často používaných metod a algoritmů a rešerší použitých herních AI ve vybraných populárních hrách.

Implementované a porovnané algoritmy v praktické části byly behavioralně modelovací algoritmus konečného automatu a behaviorální strom. Pro vlastní porovnání byly vytvořeny dvě ukázky v herním enginu Unity, kdy obě tyto ukázky využívaly stejné prostředí a modelovaly srovnatelné chování u obou herních AI. Výsledné algoritmy byly porovnány z hlediska na složitosti návrhu, implementace a času potřebného na vykonání navrženého chování.

Z rešerše ohledně herní AI je jasné, že se do budoucna předpokládá vývoj herní AI, která se bude schopna učit i po konečném vydání hry, ale tento způsob zatím není až tak populární, jak někteří autoři očekávají v budoucnu. Je to z důvodu rizika nežádoucího chování herní AI, dále obtížného ladění vzniklého chování a také výpočetní náročnosti na učení herní AI. Proto je stále populární využívat konečné automaty a behaviorální stromy i pro současné herní AI.

Z porovnání konečného automatu a behaviorální stromu se ukazuje, že behaviorální strom je složitější jak na návrh, tak i na implementaci a je také ve výsledku o něco pomalejší než konečný automat. Behaviorální strom je ale přehlednější a jednodušší na implementaci při použití komplexních herních AI. Tyto výsledky se shodují s literárními zdroji.

Přínos předkládané práce tak spočívá jednak v analýze aktuálně používané herní AI, rámcovému popisu využívaných metod a algoritmů v této oblasti a také porovnání dvou nejpoužívanější algoritmů rozhodování herní AI s příklady konkrétních implementací vytvořených ve stejném ukázkovém prostředí.

Práce by dále mohla být rozšířena o složitější ukázky konečného automatu a behaviorálního stromu, také by mohla být vytvořena např. ukázka využívající strojové učení, což by bylo ale podstatněji náročnější.

SEZNAM POUŽITÝCH ZDROJŮ

[1] BAYER, V. Umělá inteligence pro hraní her. Brno, 2017. Bakalářská práce, Fakulta informačních technologií, Vysoké učení technické v Brně.

[2] RABIN, S. Game AI Pro 3: collected wisdom of game AI professionals. Boca Raton: Taylor & Francis, 2017. ISBN 978-1498742580.

[3] CHURCHILL, D. Heuristic Search Techniques for Real-Time Strategy Games. Edmonton, 2016. Ph.D. thesis, Department of Computer Science, University of Alberta.

[4] DAGRACA, M. Practical Game AI Programming. Birmingham: Packt Publishing, 2017. ISBN 978-1787122819.

[5] SMITH, M., FERNS, S. Unity 2021 Cookbook. Birmingham: Packt Publishing, 2021. ISBN 978-1839217616.

[6] PRICE, J, M, M.J. *C# 8.0 and .NET Core 3.0: Modern CrossPlatform Development*. Birmingham: Packt Publishing, 2019. ISBN 978-1788478120.

[7] AVERSA, D. *Unity Artificial Intelligence Programming*. Birmingham: Packt Publishing, 2022. ISBN 978-1803238531.

[8] SIZER, B. Kylotan. The Total Beginner's guide to Game AI. [online]. *gamedev.net*, 2018 [cit. 2023-01-19]. Dostupné z: online https://www.gamedev.net/tutorials/programming/artificial-intelligence/the-total-beginners-guide-to-game-ai-r4942/

[9] ISLA, D. GDC 2005 Proceeding: Gandling Complexity in the Halo 2 AI. [online]. *GameDeveloper* , 2005 [cit. 2023-03-12]. Dostupné z: online https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai

[10] AI and Games. The AI of Half-Life: Finite State Machines | AI 101. In: *YouTube* [online]. 29.5.2019 [cit. 2023-03-14]. Dostupné z: https://www.youtube.com/watch?v=JyF0oyarz4U&ab_channel=AIandGames

[11] AI and Games. Building the AI of F.E.A.R. with Goal Oriented Action Planning | AI 101. In: *YouTube* [online]. 6.5.2020 [cit. 2023-3-14]. Dostupné z: https://www.youtube.com/watch?v=PaOLBOuyswI&ab_channel=AIandGames

[12] AI and Games. The AI of DOOM (1993) | AI and Games. In: *YouTube* [online]. 28.4.2022 [cit. 2023-3-14]. Dostupné z: https://www.youtube.com/watch?v=owAxZPE9a0E&ab_channel=AIandGames

[13] AI and Games. The AI of DOOM (2016) | AI and Games. In: *YouTube* [online]. 5.8.2018 [cit. 2023-3-15]. Dostupné z: https://www.youtube.com/watch?v=RcOd-twioEfI&ab_channel=AIandGames

[14] AI and Games. How Gran Turismo's `Sophy` Actually Works | AI and Games. In: *YouTube* [online]. 1.8.2022 [cit. 2023-3-15]. Dostupné z: https://www.youtube.com/watch?v=dUnhVsU0evc&ab_channel=AIandGames

[15] AI and Games. How Forza's Drivatar Actually Works | AI and Games. In: *YouTube* [online]. 30.5.2021 [cit. 2023-3-16]. Dostupné z: https://www.youtube.com/watch?v=JeYP9eyIl4E&t=15s&ab_channel=AIandGames

[16] AI and Games. Exploring the AI of Command & Conquer | AI and Games. In: *YouTube* [online]. 29.7.2020 [cit. 2023-3-18]. Dostupné z: https://www.youtube.com/watch?v=Wb84Vi7XFRg&t=67s&ab_channel=AIandGames

[17] THOMPSON, T. The AI of DOOM (1993). In: *GameDeveloper* [online]. 2.5.2022 [cit. 2023-3-23]. Dostupné z: https://www.gamedeveloper.com/blogs/the-ai-of-doom-1993

[18] V. Adam. Umělá inteligence v počítačových hrách aneb Myslí ta hra skutečně sama? In: *ackee* [online]. 14.12.2016 [cit. 2023-4-14]. Dostupné z: https://www.ackee.cz/blog/umela-inteligence-v-pocitacovych-hrach-aneb-mysli-ta-hra-skutecne-sama

[19] HARRISON-LORD, T. Everything you need to know about Gran Turismo Sophy. In: *Traxion* [online]. 20.2.2023 [cit. 2023-5-9]. Dostupné z: https://traxion.gg/everything-you-need-to-know-about-gran-turismo-sophy/

[20] SF2PLATINUM. In: *WordPress* [online]. 20.1.2017 [cit. 2023-4-16]. Dostupné z: https://sf2platinum.wordpress.com/author/sf2platinum/

[21] QUYTECH. Unity: A Game Engine Also For a Non-Game App Development. In: *QUYTECH* [online]. 11.2.2020 [cit. 2023-4-16]. Dostupné z: https://www.quytech.com/blog/unity-a-game-engine-for-a-non-game-app/

[22] RABIN, S.. Game AI Pro: collected wisdom of game AI professionals. Boca Raton: Taylor & Francis, 2013. ISBN 9781466565968.

[23] The Cherno. What is a GAME ENGINE?. In: *Youtube* [online]. 14.10.2018 [cit. 2023-4-16]. Dostupné z: https://www.youtube.com/watch?v=vtWdg-tMo1T4&t=2s&ab_channel=TheCherno

[24] Unity. Plans and Pricing. In: *Unity* [online]. 2023 [cit. 2023-4-16]. Dostupné z: https://unity.com/pricing

[25] Jead. What platforms are supported by Unity?. In: *Unity* [online]. 27.6.2022 [cit. 2023-4-16]. Dostupné z: https://support.unity.com/hc/en-us/articles/206336795-What-platforms-are-supported-by-Unity-

[26] Mono Project. About Mono. In: *Mono-Project* [online]. 2023 [cit. 2023-4-18]. Dostupné z: https://www.mono-project.com/docs/about-mono/

[27] BOURG, M, D, Glenn SEEMANN. AI for Game Developers.California: O'Reilly Media. 2004. ISBN 9781491900109

[28] PadaOne Games. Behavior Bricks. In: *Unity* [online]. 3.4.2019 [cit. 2023-5-1]. Dostupné z: https://assetstore.unity.com/packages/tools/visual-scripting/behavior-bricks-74816#releases

[29] TheKiwiCoder. Behaviour Tree Editor. In: *TheKiwiCoder* [online]. 2023 [cit. 2023-5-1] Dostupné z: https://thekiwicoder.com/behaviour-tree-editor/

[30] GameDevChef. AI in Unity Tutorial. Behavior Trees.. In: *YouTube* [online]. 9.6.2020 [cit. 2023-5-9]. Dostupné z: https://www.youtube.com/watch?v=F-3nxJ2ANXg&t=624s&ab_channel=GameDevChef

[31] AI and Games. How Machine Learning is Transforming the Video Games Indistry | AI 101. In: *YouTube* [online]. 18.4.2023 [cit. 2023-5-12]. Dostupné z: https://www.youtube.com/watch?v=dm_yY-hddvE&t=644s&ab_channel=AIand-Games

[32] TOFTEDAHL, M. Which are the most commonly used Game Engines. [online]. *GameDeveloper*, 30.9.2019 [cit. 2023-03-12]. Dostupné z: https://www.gamedeveloper.com/production/which-are-the-most-commonly-used-game-engines-

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

AI	Artificial Intelligence
FSM	Finite State Machine
BT	Behavior Tree
RTS	Real Time Strategy
QR-SAC	Quantile Regression Soft-actor Critic
LAN	Local Area Network
GUI	Graphical User Interface

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

Příloha P1: CD-ROM obsahující projektové soubory ukázek, zdrojové kódy a spustitelné ukázky.

PŘÍLOHA P I: STRUKTURA PŘILOŽENÉHO CD-ROM

Přiložené CD-ROM obsahuje:

- Bakalářskou práci ve formátu .pdf:
  BP_NemacekMartin_2023.pdf
- Spustitelnou verzi ukázek ve formátu .zip:
  Spustitelne zerze.zip
- Oddělené zdrojové kódy ve formátu .zip:
  Oddelene kody.zip
- Ukázku konečného automatu ve formátu .unitypackage:
  FSM Projekt.unitypackage
- Ukázku behaviorálního stromu ve formátu .unitypackage:
  BT Projekt.unitypackage