Demonštrácia rozhodovacích herných algoritmov v Unity 3D

Martin Panáček

Bakalářská práce
2019/2020

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně

Fakulta aplikované informatiky

Ústav informatiky a umělé inteligence

Akademický rok: 2019/2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Martin Panáček**
Osobní číslo:	**A16082**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Demonstrace rozhodovacích herních algoritmů v Unity 3D**
Téma práce anglicky:	**The Demonstration of Decision-making Game Algorithms in Unity 3D**

Zásady pro vypracování

1. Nastudujte vývojový framework Unity 3D a v rámci teoretické části jej popište.
2. Popište způsoby využití rozhodovacích algoritmů a algoritmů pro vyhledání cesty v rámci nasazení v herních aplikacích.
3. Teoreticky definujte způsob funkce alespoň 2 rozhodovacích algoritmů a algoritmů pro vyhledání cesty.
4. Prakticky implementujte jednoduchou demo aplikaci vizualizující způsob práce vybraných algoritmů.
5. Dále implementujte rozšířené demo, kde bude ukázka praktického nasazení vybraných algoritmů v jednoduché herní aplikaci disponující herní scénou.

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

Seznam doporučené literatury:

1. DAGRACA, Micael. *Practical Game AI Programming*. Birmingham, B3 2PB, UK: Packt Publishing, 2017. ISBN 978-1-78712-281-9.
2. SMED, Jouni a Harri HAKONEN. *Algorithms and networking for computer games*. Second edition. Hoboken, NJ, USA, 2017. ISBN 978-111-9259-824.
3. MILLINGTON, Ian a John David FUNGE. *Artificial intelligence for games*. 2nd ed. Burlington, MA: Morgan Kaufmann/Elsevier, c2009. ISBN 978-0-12-374731-0.
4. GONZÁLEZ CALERO, Pedro A. a Marco Antonio GÓMEZ-MARTÍN. *Artificial Intelligence for Computer Games*. New York: Springer, [2011]. ISBN 978-1-4419-8187-5.
5. AVERSA, Davide, Clifford PETERS a Aung Sithu KYAW. *Unity Artificial Intelligence Programming*. Fourth Edition. Birmingham, B3 2PB, UK: Packt Publishing, 2018, 2017. ISBN 9781789533910.
6. KIRBY, Neil, Clifford PETERS a Aung Sithu KYAW. *Introduction to game AI*. Fourth Edition. Boston: Course Technology/Cengage Learning, c2011. ISBN 978-1-59863-998-8.
7. OKITA, Alex, Clifford PETERS a Aung Sithu KYAW. *Learning C# programming with Unity 3D*. Fourth Edition. Boca Raton: CRC Press, [2015]. ISBN 978-1-4665-8652-9.

Vedoucí bakalářské práce: **Ing. Radek Vala, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání bakalářské práce: **28. listopadu 2019**
Termín odevzdání bakalářské práce: **15. května 2020**

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

Ve Zlíně dne 9. prosince 2019

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 bakalářské 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 a jeden výtisk bude uložen u vedoucího práce;
- 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 Martin Panáček, v. r.

ABSTRAKT

Bakalárska práca sa zaoberá dvomi časťami hernej umelej inteligencie a tými sú hľadanie ciest a rozhodovanie. Pri hľadaní ciest sa bližšie pozrieme na algoritmy A* a Dijkstra. Rozoberieme si ich funkčnosť, využitie a pre každý vytvoríme demo poukazujúce na spôsob akým algoritmy fungujú. V ďalšej časti práce sa pozrieme na dva rozdielne prístupy pri tvorbe rozhodovania - konečné automaty a rozhodovacie stromy. Porovnáme tieto dva prístupy z hľadiska implementačnej náročnosti a škálovateľnosti. Toto dosiahneme napodobením iteratívneho vývojového cyklu, pri ktorom nadizajnujeme základné správanie NPC charakteru s drobnými nedostatkami a po otestovaní sa rozhodneme toto správanie rozšíriť. Na demonštráciu správania charakteru vytvoríme interaktívne demo, predstavujúce prototyp strielačky. V poslednej časti vytvoríme demo, v ktorom budeme demonštrovať správanie dvoch typov charakterov na seba reagujúcich a podnecovaných prostredím za použitia rozhodovacích stromov.

Kľúčové slová: Hľadanie ciest, rozhodovanie, umelá inteligencia, Dijkstra, A*, konečné automaty, rozhodovacie stromy, Unity 3D, ...

ABSTRACT

The bachelor thesis deals with the two parts of the game artificial intelligence, pathfinding and decision making. In the part about pathfinding, we will take a closer look at the A* and Dijkstra algorithms. We will discuss their functionality, usage, and we will demonstrate their approaches while searching for the shortest path. In the second part of the thesis, we will discuss two different approaches to decision making. These approaches are finite state machines and behavior trees. In this part, we will simulate the iterative development process where in the first stage we come up with some basic design of decision making for the character, with some unfinished parts. After testing this character, we will try to modify the mentioned designed behavior. For this demonstration, we will create an interactive demo, similar to a shooter prototype. The last part will demonstrate the behavior of the two types of characters reacting to each other and incited by the environment in which they are operating. All of this will be implemented with the usage of behavioral trees.

Keywords: Pathfinding, decision making, artificial intelligence, Dijkstra, A*, finite state machines, behavior trees, Unity 3D, ...

Rád by som poďakoval pánovi Ing. Radkovi Valovi, Ph.D, vedúcemu bakalárskej práce, za jeho ochotu, čas a cenné pripomienky poskytnuté pri spracovávaní bakalárskej práce. Ďalej by som chcel poďakovať svojej rodine za neúnavnú podporu, dôveru a trpezlivosť.

OBSAH

ÚVOD

Herná umelá inteligencia je v dnešnej dobe viditeľná takmer v každej hre na rôznych úrovniach zložitosti. Môže byť v podobe rôznych charakterov alebo dokonca aj systémov, o ktorých uživateľ vôbec nevie, že sa v hre nachádzajú. Umelá inteligencia charakterov sa skladá z viacerých úrovní, ktoré medzi sebou spolupracujú. Medzi základné úrovne môžeme radiť rozhodovanie a hľadanie ciest. Úroveň zameraná na hľadanie ciest je zodpovedná za nájdenie čo najlepšej cesty na cieľovú pozíciu z miesta, na ktorom sa charakter nachádza. Súčasne môže byť užitočná aj pri hľadaní cesty na viaceré miesta zároveň. Hľadanie ciest veľmi úzko spolupracuje s rozhodovaním. Je to jednoduché, keď sa charakter rozhodne niekam ísť, časť umelej inteligencie (AI) zodpovedná za nájdenie cesty ho tam dostane. V mnohých hrách sa nad tieto dve úrovne pridáva aj takzvaná strategická umelá inteligencia, ktorá dovoľuje spoluprácu viacerých charakterov zároveň. Keďže sa herné charaktery (NPC) nachádzajú v simulovanom svete, musí umelá inteligencia na rôznych úrovniach spolupracovať aj s ďalšími systémami alebo komponentami. Môže to byť napríklad komponent starajúci sa o spúšťanie a kontrolu animácií charakteru, alebo dokonca aj komponent zaobstarávajúci správnu fyziku v scéne.

V tejto bakalárskej práci si bližšie rozoberieme hľadanie ciest a rozhodovanie. Pri hľadaní ciest sa pozrieme na pomerne známe algoritmy A* a Dijkstra. Vysvetlíme si ich funkcionalitu, využitie a porovnáme ich výkonnosť pri hľadaní najkratšej cesty z bodu do bodu. Obidva algoritmy demonštrujeme v interaktívnom deme aj s vizualizáciou ich funkčnosti a spôsobu hľadania cesty.

V druhej časti práce rozoberieme a porovnáme konečné automaty a rozhodovacie stromy. Budeme porovnávať ich zložitosť implementácie a škálovateľnosť. Tohoto výsledku dosiahneme napodobením iteratívneho vývojového cyklu kedy si v prvej fáze navrhneme základný dizajn správania, ktorý po otestovaní budeme chcieť poupraviť a rozšíriť. Nadizajnované správanie bude demonštrované v interaktívnom deme predstavujúcom prototyp strieľačky.

Poslednú časť bakalárskej práce tvorí finálne demo, ktorého cieľom je demonštrovať rozhodovanie viacerých charakterov v scéne bez vstupu uživateľa.

I. TEORETICKÁ ČASŤ

1 Herný engine

Herný engine môžeme definovať ako súbor nástrojov a predprogramovaných funkcionalít alebo služieb, ktoré nám zjednodušia vývoj hry. V dnešnej dobe existuje pomerne veľa herných enginov s rôznymi špecializáciami. Každý engine by mal obsahovať aspoň tieto funkcionality:

- animačný framework,
- 2D alebo 3D grafické/dizajnové nástroje,
- asset manažment - schopnosť vytvárať a vkladať assety,
- podpora práce s audiom - vkladanie hudby, mixovanie hudby,
- cross-platform deployment - Možnosť vydať hru na rôzne platformy (Napríklad desktop, mobil, web, konzoly),
- grafické užívateľské rozhrania,
- podpora pre multiplayerové hry,
- fyzika,
- podpora scriptovania minimálne v jednom jazyku.

Herný engine by nám mal poskytnúť sadu nástrojov na zjednodušenie vývoja hry a jej komplexných systémov tak, aby sa vývojársky tím dokázal viac sústrediť na vytvorenie lepšieho zážitku pre svojich hráčov. V úplných začiatkoch herného vývoja bola každá hra vytváraná od začiatku. Časom, namiesto toho, aby sa rôzne žánre hier vytvárali od nuly, začali sa pomaly objavovať herné enginy, ktoré vývojárom zjednodušili prácu a poskytli predprogramované funkcionality.

Herné enginy nie sú softvérové balíčky, z ktorých môžete vytvoriť akúkoľvek hru. Tieto enginy sú špecializované, avšak aj veľmi flexibilné na vytváranie hier rôznych žánrov. [1] Napríklad GameMaker je vyvinutý primárne pre 2D hry. Ďalším príkladom môže byť engine Frostbite od firmy Dice, ktorý je primárne vyvinutý pre sériu hier Battlefield. Rovnako ako aj známa česká firma Warhorse studios používa upravenú verziu Unreal Engine pre stredoveké RPG. Dnes na internete nájdete mnoho herných enginov, ktoré sú zadarmo alebo open-source, niektoré z nich sú aj patentované.

Pred samotným vývojom hry je veľmi dôležité si uvedomiť aký druh hry ideme robiť a aké funkcionality bude mať daná hra. Následne na základe týchto informácií

urobiť rešerš herných enginov. Investovať čas do týchto rozhodnutí sa vyplatí, dokážu v budúcnosti ušetriť veľa práce pri implementácií rôznych funkcionalít.

V tejto bakalárskej práci sa pozrieme na herný engine Unity 3D, keďže sme si vybrali práve tento engine na demonštráciu už spomenutého hľadania ciest a rozhodovania.

2 Unity 3D

Unity je cross-platform herný engin vyvinutý spoločnosťou Unity Technologies. V engine môžete vytvárať 2D, 3D, virtual reality a augmented reality hry. Rovnako ako aj simulácie s rôznymi účelmi pre viac ako 25 platforiem. S tvorbou 2D hier, Unity ponúka import 2D spritov a pokročilého renderovania 2D sveta. Pri 3D hrách dovoľuje špecifikáciu textúrovej kompresie, mipmapy, nastaviteľné rozlíšenie pre každú platformu, bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), dynamické tiene používajúce shadow maps, render-to-texture a full-screen post-processing efekty. [9] Engine ponúka skriptovacie rozhranie (API) v programovacom jazyku C Sharp. V minulosti podporoval aj jazyk Boo, ktorý bol vymazaný s verziou Unity 5 [9] a JavaScript nazývaný UnityScript, ktorý prestal byť podporovaný po vydaní Unity 2017.1 [10]. V roku 2018 bolo Unity použité na tvorbu približne polovice mobilných hier a 60 percent augmented a virtual reality aplikácií. [7] Engine bol použitý aj na tvorbu momentálne jednej z najúspešnejších mobilných hier Call of Duty®: Mobile. V roku 2017 Unity predstavilo open-source softwér Unity Machine Learning agents prepojený s rôznymi machine learning programami vrátane TensorFlow od Google. Okrem herného vývoja sa tento softvér používa aj pri tvorbe robotov alebo autonómnych áut. [8]

2.1 Unity Asset Store

Tvorcovia môžu vytvoriť a následne predávať rôzne assety, predprogramované časti hier, frameworky, komponenty pomocou Unity Asset Store. V obchode môžete nájsť aj veľa assetov zadarmo, preto ho veľa tvorcov používa ako overený zdroj pri tvorbe aplikácií.

2.2 Licencia

Unity sa dá zadarmo používať pre štúdijné účely, alebo aj v prípade malej firmy generujúcej zisk menej ako 100 000$ ročne z hier vytvorených pomocou Unity. Ďalej sa dá vybrať z viacerých platených plánov ako Plus, Pro, Enterprise. Každý zo spomenutých plánov ponúka určité výhody ako napríklad technická podpora, diagnóza cloudu, analytiku a mnoho ďalších.

3 Umelá inteligencia

Žijúce organizmy ako zvieratá a ľudia majú prirodzene určitú úroveň inteligencie, ktorá im umožňuje robiť zmysluplné rozhodnutia počas ich každodenného života. Počítače sú zariadenia pracujúce s dátami, schopné robiť jednoduché matematické výpočty v obrovskej rýchlosti. Umelá inteligencia je oblasť, ktorá sa zaoberá učením počítačov schopnosti myslieť a robiť rozhodnutia ako žijúce organizmy. Je rozsiahla, lebo zahŕňa rôzne prístupy implementácie alebo učenia pri rôznych situáciách. Pozrime sa na zopár oblastí aplikácie umelej inteligencie:

- Počítačové videnie: Schopnosť počítača spracovať vizuálny vstup z videa alebo fotografií a na základe jeho analýzy vykonať určité operácie ako rozpoznávanie objektov, tváre, atď.
- Spracovanie prirodzeného jazyka (Natural language processing): Toto je schopnosť počítača čítať a rozumieť ľudskému jazyku. Problémom je, že ľudský jazyk je pre počítače zložitý. Je to aj z toho dôvodu dvojjazyčnosti. Ľudia dokážu poskladať dve rozdielne vety, ktoré znamenajú rovnakú vec. [2]

3.1 Umelá inteligencia v hrách

Umelá inteligencia je dôležitou súčasťou každej hry, kde sa hráč stretáva alebo spolupracuje s inými charaktermi, systémami, ktoré sa dokážu samostatne rozhodovať. Nie je to najdôležitejšia časť, ktorú hra obsahuje. Aby si niekto zahral hru, musí ho tá hra baviť. To už je z väčšej časti otázka kvality herného dizajnu. Ako sme si už spomenuli, umelá inteligencia dokáže byť dôležitou podporou pri zábavnosti hry. Uveďme si ako príklad ľubovoľnú strielačku. Keď hráč prechádza levelom a stretáva sa s rôznymi formami a počtami nepriateľov, očakáva, že boj s daným NPC bude predstavovať výzvu, ale zároveň nebude príliš náročný alebo veľmi jednoduchý. Disciplína hľadajúca ideálnu obtiažnosť pre hráča sa v hernom dizajne nazýva Flow - je to súčasť dizajnu, keď sa pre hráča snažíme vytvoriť výzvu, ktorá nebude náročná, ale ani príliš jednoduchá. K tomu nám môže v kompetitívnych hrách pomôcť umelá inteligencia. Napríklad vo forme botov, ktorí sa mnohokrát používajú v zápasoch, aj na miestach, kde si ľudia myslia, že hrajú len s ľudskými protihráčmi.

Táto technika sa používa z viacerých dôvodov. Prvým môže byť vydanie hry. Dnes je kúpa jednotlivých hráčov pomerne drahá vzhľadom na konkurenciu. Firmy sa uchyľujú k umelej inteligencii pri kompetitívnych hrách, aby aj zopár hráčom, ktorých majú, tesne po vydaní hry ponúkli dobrý herný zážitok a nenechali ich dlho čakať pri hľadaní zápasu. Druhým, veľmi častým dôvodom býva udržanie hráča v hre. Keď sa oprieme o hráčovu psychológiu, zistíme, že pre väčšinu hráčov je oveľa zábavnejšie hrať

proti ostatným hráčom ako proti botom. Pri hre proti reálnym ľuďom sa stretávame s oveľa náročnejšími výzvami. To automaticky znamená viac prehier pre hráča predtým ako naberie dostatočné skúsenosti. Keďže väčšina bežných hráčov sa nedostanú na úroveň "hardcore" a zároveň nechcú prehrávať, tak sa ako nepriatelia používajú boti. Samozrejme, hráči nevedia, že hrajú proti botom.

Keď sa vrátime k hodnoteniu kvality hry na základe umelej inteligencie, každá hra by mala pre hráča predstavovať určitý zážitok. Rôzni hráči vyhľadávajú rôzne zážitky. Teoreticky sa dá povedať, že čím silnejší špecifický zážitok vyhľadávaný špecifickou skupinou hráčov v hre dokážeme vytvoriť, tým bude naša hra úspešnejšia v danej skupine hráčov. K vytváraniu už spomínaného zážitku dokáže prispieť umelá inteligencia. Čím viac sa dokážu herné charaketry správať a rozhodovať ako ľudia, tým viac si my ľudia k danej hre vytvárame puto, pretože máme pocit, že jednáme s reálnymi ľuďmi. Dostávame sa k hlavnému rozdielu medzi hernou umelou inteligenciou a umelou inteligenciou používanou v reálnom svete.

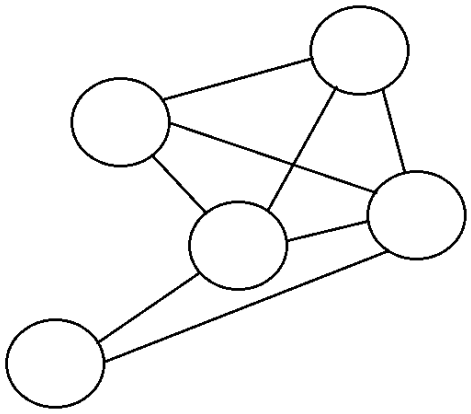
Úlohou umelej inteligencie v hrách je priniesť hráčovi zábavu poskytnutím súperov, proti ktorým je výzva hrať. Tiež poskytnutie zaujímavých charakterov, ktorí sa správajú realisticky vo vnútri herného sveta. Takže cieľom nie je napodobniť myšlienkový proces ľudí a zvierat tak ako tomu je pri normálnej umelej inteligencii. Cieľom je vytvoriť herné charaktery, ktoré sa správajú inteligentne a reagujú na meniace sa situácie v hernom svete spôsobom, ktorý dáva hráčovi zmysel. [2] Jedným z dôvodov prečo nechceme aby umelá inteligencia v hrách bola výpočtovo náročná je, že výpočtová sila potrebná pre kalkulácie umelej inteligencie je zdieľaná medzi mnoho ďalších operácii. Týmito operáciami môžu byť napríklad grafický rendering alebo fyzické simulácie. Všetky tieto kalkulácie sa počítajú v reálnom čase a pre každú hru je kritické dosiahnuť stabilný počet snímkov za sekundu (frame-rate) na čo najviac zariadeniach. Pre umožnenie vytvárania komplexnejšej umelej inteligencie bez straty na výkone sa Google pokúsil vytvoriť dedikovaný procesor pre kalkulácie umelej inteligencie (Google's Tensorflow). So zväčujúcim sa procesorovým výkonom sa vývojárom naskytujú príležitosti na pridelenie väčšej výpočtovej sily pre kalkulácie umelej inteligencie. No s ohľadom na ostatné operácie, ktoré potrebujú robiť výpočty v reálnom čase to stále zostáva výzvou. [2]

4 Hľadanie ciest

V dnešnej dobe v hrách nie je ničím výnimočným vidieť pohybujúci sa charakter po hernom prostredí smerujúci k určitému miestu a popri tom obchádzajúci rôzne prekážky, ktoré mu stoja v ceste. Niekedy je tento pohyb charakterov priamo ovládaný vývojármi, napríklad hliadkovanie, kde charakter slepo nasleduje svoju trasu. Zafixované cesty je jednoduché implementovať, no charakter nasledujúci túto cestu môže byť veľmi jednoducho pomýlený tým, že mu do cesty vložíme nejaký iný objekt, ktorý mu bude zavadzať. Charaktery s komplexnejšou umelou inteligenciou väčšinou nevedia dopredu kam pôjdu. Toto rozhodnutie za nich robí buď hráč alebo jednotlivé podnety z prostredia, na ktoré ich umelá inteligencia reaguje. Napríklad v strategických hrách môžete vybrať skupinu jednotiek a dať im príkaz na presun na určitú lokáciu. Vybrané jednotky potom musia nájsť cestu na danú lokáciu s tým, že budú obchádzať jednotlivé prekážky. Prekážky môžu byť rozdielne pre rôzne druhy jednotiek. Napríklad jednotky, ktoré môžu lietať sú schopné nadletieť budovy alebo kopce, pričom pozemné jednotky musia nájsť inú cestu. Hliadkujúci charakter v určitých hrách sa bude musieť presunúť k najbližšiemu alarmu, aby privolal posily ak zazrie hráča. V plošinovkách musia byť charaktery schopné prenasledovať hráča medzi rôznymi prekážkami. Pre každý jeden z týchto charakterov musí byť umelá inteligencia schopná kalkulovať vhodnú trasu z miesta, kde sa charakter nachádza, na miesto, ktoré sa stalo jeho cieľom. Pri výslednej trase chceme, aby dávala zmysel a zároveň, aby bola čo najkratšia. Toto správanie sa nazýva hľadanie ciest, niekedy plánovanie ciest. Môžeme ho nájsť ako súčasť takmer každej hernej umelej inteligencie. Niekedy sa hľadanie ciest používa len na kalkuláciu cesty z počiatočnej pozície do cieľa, kde o cieli rozhoduje iná časť umelej inteligencie. No hľadanie ciest môže byť tiež použité na rozhodovanie kam sa pohnúť a ako sa dostať do cieľovej pozície. Väčšina hier používa na hľadanie cesty algoritmus A*. Hoci je tento algoritmus veľmi ľahko implementovateľný a efektívny, nedokáže pracovať s dátami v bežnom hernom prostredí. Na to, aby bol schopný nájsť najkratšiu cestu z momentálnej pozície do cieľovej, potrebuje, aby bol herný level reprezentovaný v dátovej štruktúre, ktorá sa nazýva smerovaný nezáporný vážený graf (a directed non-negative weighted graph). Je dôležité, aby táto reprezentácia bola vytvorená správne. V inom prípade, by sa mohlo stať, že nám náš algoritmus vráti cestu, ktorá nie tak úplne dáva zmysel.

Graf v našom prípade, je matematická štruktúra reprezentovaná diagramaticky (Obr. 4.1). Nemá to nič spoločné s významom slova graf ako diagram reprezentovaný ako koláčový graf alebo histogram.

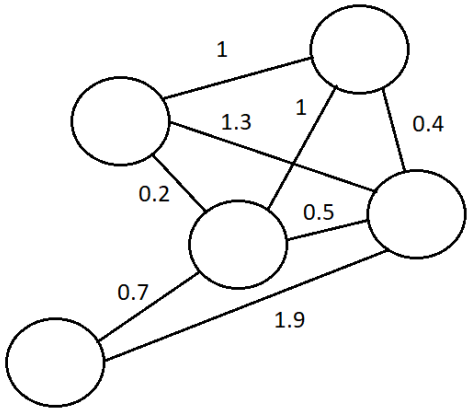
Náš graf pozostáva z dvoch druhov elementov, a to z uzlov a prepojení medzi uzlami. Pri hľadaní ciest uzol väčšinou reprezentuje nejakú časť levelu ako napríklad izbu, chodník alebo nejakú časť vonkajšieho priestoru. Prepojenie je neusporiadaný



Obr. 4.1 Základný graf.

pár uzlov (na oboch stranách sa nachádza uzol). Prepojenia nám definujú, ktoré lokácie sú prepojené. Napríklad, charakter dokáže vyjsť z izby na chodník, ak sú tieto dva uzly prepojené. Ak tieto dva uzly nie sú prepojené, charakter sa pokúsi nájsť cestu prostredníctvom iných uzlov. Cesta po grafe môže byť zložená z nula alebo viacerých prepojení. Napríklad, ak sa má charakter posunúť na pozícii, na ktorej sa už nachádza, nepotrebuje sa pohybovať po prepojeniach. Ak jeho štartovacia a cieľová poloha sú prepojené, tak je na pohyb potrebné len jedno prepojenie.

Tento graf je poskladaný z uzlov a prepojení medzi uzlami. Každému prepojeniu v grafe pridáme nejakú numerickú hodnotu, ktorú nazývame buď váha alebo cena (Obr. 4.2).



Obr. 4.2 Graf s vyznačenými váhami.

Jednotlivé ceny prepojení môžu predstavovať vzdialenosť, alebo čas potrebný pre charakter na zvládnutie danej cesty. Samozrejme, cenu prepojenia môže ovplyvňovať

veľa rôznych faktorov, ako napríklad náročnosť terénu, prístupnosť a podobne. Váha cesty grafom z uzlu, na ktorom začíname, do cieľového uzlu je vypočítaná jednoduchým sčítaním cien jednotlivých prepojení medzi uzlami. [3]

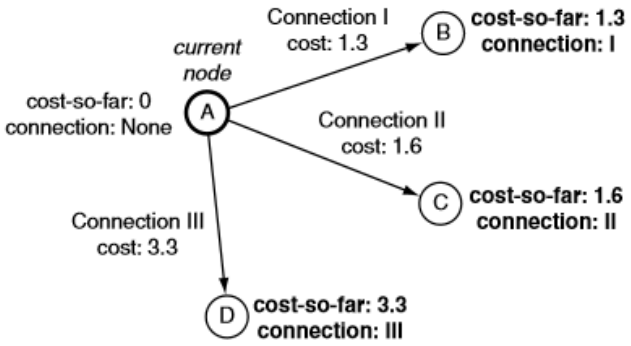
4.1 Dijkstra

Dijkstra algoritmus je pomenovaný po Edsgerovi Dijkstra. Slávnom matematikovi, ktorý sa preslávil svojim krátkym dvojstranovým článkom nazvaným "Goto statement considered harmful". [12] Dijkstra algoritmus nebol pôvodne vyvinutý na hľadanie ciest v hrách. Pôvodne slúžil na riešenie problému v teórii matematických grafov, ktorý sa nazýva "Najkratšia cesta". Hľadanie ciest v hrách má jeden začiatočný bod a jeden cieľový bod. Algoritmus pre hľadanie najkratšej cesty je nadizajnovaný na hľadanie najkratšej možnej cesty kamkoľvek zo štartovacej pozície. Dijkstrov algoritmus dokáže nájsť najkratšiu možnú cestu medzi počiatočným a cieľovým bodom. Jeho neefektivita pri hľadaní ciest spočíva v prechádzaní všetkých možných trás, nielen tých najkratších, po ktorých sa vieme dostať do cieľového bodu. Počas hľadania najkratšej cesty do cieľa, algoritmus nájde najkratšiu cestu do každého jedného uzlu v grafe. [13] Samozrejme, môže byť modifikovaný, aby vrátil len najkratšiu výslednú cestu, no stále nie úplne efektívnym spôsobom. Na základe týchto nedokonalostí sa Dijkstrov algoritmus v hrách väčšinou nepoužíva priamo na hľadanie ciest. Avšak je to dôležitý algoritmus, ktorý sa používa často na strategickú analýzu a má využitie aj v iných oblastiach hernej umelej inteligencie. [3] V tejto bakalárskej práci sa na algoritmus zameriame ako na jednoduchšiu a menej efektívnu verziu často používaného A* algoritmu.

Predstavme si graf a v ňom dva uzly: štart a cieľ. Medzi týmito uzlami by sme chceli nájsť cestu, ktorej váha zo štartu do cieľa by bola oproti ostatným cestám najnižšia. Môže existovať viacero možných ciest s rovnakou váhou. Ak taká situácia nastane, očakávame, že algoritmus nám vráti len jednu možnú cestu a nezáleží na tom ktorá to bude. Cesta, ktorú očakávame, že nám algoritmus vráti bude pozostávať z prepojení medzi uzlami. Každé prepojenie má určitú váhu. Preto musíme vedieť, ktoré prepojenia použiť. Zoznam uzlov v tom prípade stačiť nebude z dôvodu, že k jednému uzlu môže existovať viacero prepojení. Ak teda máme viacero prepojení k jednému uzlu, algoritmus by mal vždy automaticky vybrať to s nižšou váhou.

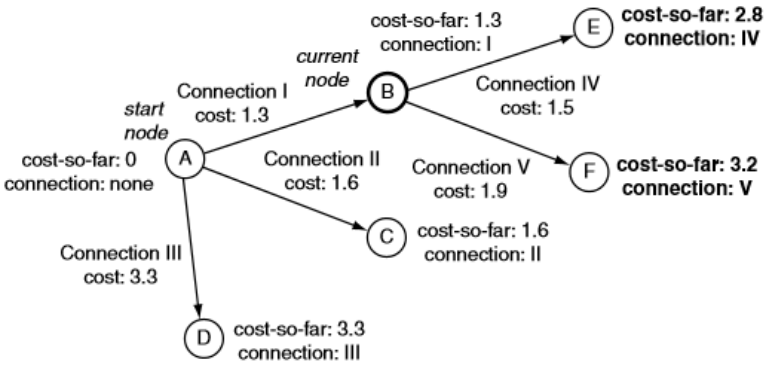
Dijkstra prechádza jednotlivé uzly zo štartovacieho uzlu po ich prepojeniach. Vždy si ukladá informáciu, odkiaľ prišiel, aby sa vedel pri nájdení cieľového uzlu vrátiť po čo najkratšej ceste naspäť do počiatočného uzlu. Dijkstra funguje v iteráciach. Pri každej iterácii pracuje s jedným uzlom a skontroluje všetky susedné uzly, ku ktorým existujú prepojenia z daného uzlu. Pre každé prepojenie nájde uzol, ku ktorému vedie a uloží celkovú cenu cesty do daného uzlu. Celková cena cesty je súčet všetkých váh jednot-

livých pripojení, po ktorých by charakter musel prejsť, aby sa do daného uzlu dostal. V prvej iterácii pracuje s uzlom, na ktorom sa nachádza náš charakter (Obr. 4.3). Celková cena presunu na jednotlivé susedné uzly je rovná cene jednotlivých prepojení. V nasledujúcich iteráciach si vyberá ďalšie uzly pripojené k štartovaciemu uzlu. [3]



Obr. 4.3 Prvá iterácia Dijkstrovho algoritmu s cenami jednotlivých prepojení. Graf prevzatý z [3].

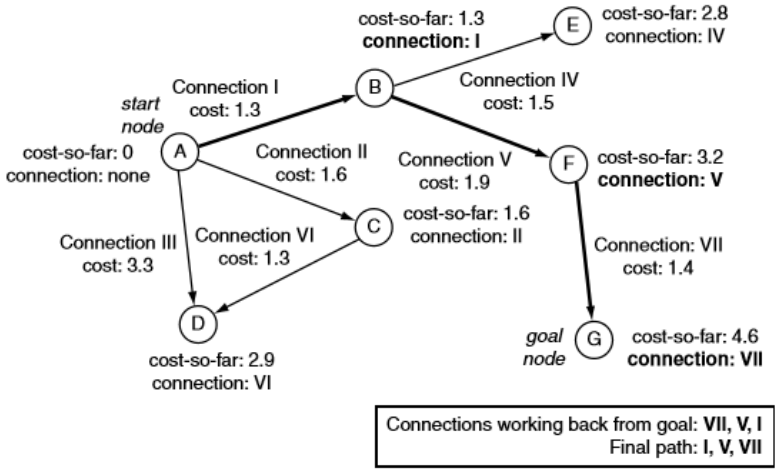
Algoritmus uchováva informáciu o všetkých uzloch, o ktorých vie, v dvoch listoch: open a closed. V open liste uchováva uzly, pre ktoré ešte nevykonával samostatnú iteráciu, ale vie, že existujú na základe iterácií vykonaných na ich susedných uzloch. V closed liste algoritmus uchováva všetky uzly, pre ktoré už vykonal samostatnú iteráciu. Ako sme už spomenuli, v prvej iterácii algoritmus vie iba o uzle, na ktorom sa nachádza charakter, tým pádom open list obsahuje jeden uzol a closed list je prázdny. V skratke - každý uzol môže byť pre algoritmus v jednej z troch kategórii: môže byť v open liste, closed liste, alebo sa s ním algoritmus ešte nestretol. V každej iterácii algoritmus vyberá z open listu uzol s najnižšou celkovou cenou (Obr. 4.4). Následne preskúma všetky jeho susedné uzly, ku ktorým vedie prepojenie, vymaže daný uzol z open listu a pridá ho do closed listu. Pri každej iterácii uzlu, na ktorom sa nachádza algoritmus predpokladáme, že sme ešte nenavštívili jeho susedné uzly. No môže nastať situácia keď v iterácii narazíme na susedný uzol, ktorý sa už nachádza v open alebo closed liste a už má vypočítanú celkovú sumu z predchádzajúcich iterácii. V tomto prípade skontrolujeme, či je cesta k uzlu, ktorú sme momentálne našli, lacnejšia na celkovú sumu ako bola predchádzajúca cesta. Ak je aktuálna cena cesty väčšia ako bola predchádzajúca cena, tak algoritmus nechá daný uzol bez zmeny. V opačnom prípade, aktualizujeme celkovú cenu uzlu, prepíšeme rodiča uzlu na uzol, ktorý momentálne iterujeme a presunieme ho do open listu. Ak sa daný uzol predtým nachádzal v closed liste, vymažeme ho.



Obr. 4.4 Druhá iterácia Dijkstrovho algoritmu. Graf prevzatý z [3].

Dijkstra algoritmus sa ukončí v prípade, že open list je prázdny, to znamená, že algoritmus preiteroval všetky uzly v grafe, na ktoré sa vieme presunúť zo štartovacieho uzlu a všetky sa nachádzajú v closed liste. Pri hľadaní ciest nás ale primárne zaujíma situácia, keď algoritmus dorazí do cieľového uzlu, čiže ho môžeme zastaviť skôr. Algoritmus by sa mal pri hľadaní cesty zastaviť, keď má cieľový uzol najnižšiu cenu v open liste. To znamená, že algoritmus už dosiahol cieľový uzol v predchádzajúcej iterácii, kde ho presunul do open listu. Prečo sme teda algoritmus neukončili keď už narazil na cieľový uzol? Pretože, keď algoritmus narazí na cieľový uzol v iteráciách pre iné uzly, neznamená to, že sme našli najkratšiu cestu ale iba to, že sme našli cieľový uzol. V praxi je toto pravidlo často porušené, pretože prvá cesta, pri ktorej nájdeme cieľový uzol býva vo väčšine prípadov zároveň aj najkratšou cestou. Ak by aj existovala kratšia cesta, tak to bude len o malý kúsok. Z tohto dôvodu veľa programátorov implementuje algoritmus spôsobom, ktorý ho ukončí hneď ako narazí na cieľový uzol.

Finálnym štádiom algoritmu je získanie konečnej cesty. Získavanie cesty začneme v cieľovom uzle a následne na základe prepojení, ktorými sme sa k danému uzlu dostali sa dostaneme do štartovacieho uzlu (Obr. 4.5). Uzly jednotlivých prepojení ukladáme do finalPath listu. Po dosiahnutí štartovacieho uzlu získame finálnu cestu, no v zlom poradí. Pred vrátením finálnej cesty musíme prevrátiť list, aby sme získali správne poradie. [3]



Obr. 4.5 Finálna cesta z uzla A do uzla G. Graf prevzatý z [3].

4.2 A*

A* je algoritmus slúžiaci na hľadanie cesty veľmi často používaný v hrách primárne kvôli jeho výkonnosti, presnosti a jednoduchosti implementácie. Algoritmus patrí medzi najpoužívanejšie algoritmy na hľadanie cesty v dnešnom hernom vývoji. [14] Algoritmus sa používa aj mimo vyhľadávania ciest, napríklad na plánovanie komplexných akcií pre charaktery. [3] Na rozdiel od Dijkstru, A* je nadizajnovaný na vyhľadávanie ciest typu "Point to point"(z jedného bodu do druhého), a používa sa na hľadanie najkratšej cesty v teórii grafu. Problém, ktorý algoritmus rieši je identický problému riešenému algoritmom Dijkstra. Rovnako, ako pri predchádzajúcom algoritme, máme vážený graf obsahujúci uzly a prepojenia medzi uzlami, ktoré majú rôzne váhy. Sú tu dva uzly, ktoré nás zaujímajú. Jeden je uzol, na ktorom sa nachádzame a druhý, kam sa chceme dostať. Úlohou algoritmu je nájsť čo najkratšiu cestu pozostávajúcu z prepojení medzi týmito dvoma uzlami. [3]

A* funguje na rovnakom princípe ako Dijkstra. Namiesto prehodnocovania uzla s najnižšou celkovou cenou si algoritmus vyberie uzol, pri ktorom má najväčšiu pravdepodobnosť, že vedie najkratšou cestou k cieľovému uzlu. Výber tohoto uzla je založený na heurisickom spôsobe. Ak je heuristika presná, algoritmus bude efektívny. Naopak, ak je heuristika nepresná, algoritmus môže byť menej efektívny ako Dijkstra.

A* rovnako ako Dijkstra pracuje v iteráciach. V každej iterácii vyhodnocuje jeden uzol z grafu a nasleduje jeho prepojenia so susednými uzlami. Pre každý susedný uzol ukladá celkovú odhadovanú cenu cesty a prepojenie, ktorým sa do daného uzlu dostal, presne ako v Dijkstra. Na rozdiel od Dijkstru, tento algoritmus počíta celkovú odhadovanú cenu pre cestu zo štartovacieho uzlu, cez iterovaný uzol až do cieľového uzlu.

Dijkstra v tomto prípade počítal len cenu cesty zo štartovacieho uzlu do uzlu ktorý prehodnocuje.

Celkový odhad ceny cesty sa počíta sčítaním dvoch hodnôt: celkovej ceny presunu na tento uzol, ktorú si pre lepšiu zrozumiteľnosť označíme G a vzdialenosti od tohoto uzlu do cieľa. Túto vzdialenosť označíme písmenom H. Ako sme už spomenuli, sčítaním týchto dvoch hodnôt dostaneme celkovú odhadovanú vzdialenosť, ktorú si označíme písmenom F (F = G + H). [15]

Rovnako ako Dijkstra, A* pracuje s open listom, kde uchováva všetky uzly o ktorých vie, no ešte nemali vlastnú iteriáciu (boli nájdené na konci prepojení momentálne iterovaných uzlov) a closed listom kde ukladá všetky uzly ktoré už mali vlastnú iteráciu. Na rozdiel od Dijkstru, každú iteráciu vyberá z listu uzol s najnižšou hodnotou F. Výber tohoto uzlu je takmer vždy rozdielny od výberu uzlu na základe najnižšej celkovej ceny ako tomu bolo v prípade Dijkstru. Primárne z dôvodu, že na základe spôsobu počítania ceny F dokáže algoritmus najskôr vybrať uzly, ktoré s najvyššou pravdepodobnosťou, budú viesť najkratšou cestou do cieľového uzlu.

Podobne ako pri predchádzajúcom algoritme môžeme naraziť počas iterovania medzi jednotlivými uzlami na uzol, ktorý už patrí buď do closed alebo open listu. V takejto situácii porovnáme celkovú cenu uzlu G z danej iterácie s jeho uloženou cenou. V prípade, že nová cena G je menšia, aktualizujeme túto hodnotu. Na rozdiel od Dijkstru, A* dokáže nájsť lepšiu cestu pre uzly, ktoré sa už nachádzajú v closed liste. Ak predchádzajúci odhad bol príliš optimistický, môže sa stať, že algoritmus vyberie nevhodný uzol na iteráciu. Ak bol tento uzol v closed liste, znamená to, že algoritmus vypočítal odhadovanú cenu F pre všetkých jeho susedov. To znamená, že tieto hodnoty boli vypočítané na základe zlého uzlu. V takomto prípade, aktualizácia hodnoty F pre daný uzol nie je dostačujúca. Algoritmus musí prejsť aj všetky jeho susedné uzly a napraviť ich ceny F. Na opravu tohoto problému existuje pomerne jednoduché riešenie. Čo musíme urobiť je, že daný uzol odstránime z closed listu a vložíme ho naspäť do open listu. Následne počkáme pokiaľ sa daný uzol dostane do iterácie a algoritmus aktualizuje ceny F všetkých jeho susedných uzlov.

V mnohých implementáciach sa zvykne hľadanie cesty algoritmom ukončovať, keď sa v open liste nachádza cieľový uzol s najnižšou hodnotou F spomedzi ostatných uzlov v liste. Ako sme si mohli všimnúť, to že má momentálne uzol najnižšiu hodnotu F v open liste neznamená, že máme najkratšiu cestu. Pretože táto hodnota F daného uzla môže byť ešte zmenená počas rôznych iterácií, algoritmus sa bežne necháva pracovať o chvíľu dlhšie. Toto môžeme dosiahnuť pridaním podmienky, že sa algoritmus zastaví len vtedy, ak bude mať uzol s najnižšou celkovou cenou G v open liste túto cenu vyššiu ako cenu cesty F, ktorú sme našli do cieľa.

Finálnu cestu dostávame rovnakým spôsobom ako pri Dijkstra algoritme. Začneme na cieľovom uzle a na základe prepojení sa dostaneme k začiatočnému uzlu. Následne list, do ktorého ukladáme jednolivé uzly obrátime naopak, aby sme dostali ich správne poradie.

Pri porovnaní A* s Dijkstrom, zistíme, že sú takmer identické. V A* pridávame naviac kontrolu, či uzol v closed liste potrebuje aktualizovať svoju cenu F. Taktiež pridávame zopár riadkov na kalkuláciu celkovej odhadovanej ceny F pomocou heuristickej funkcie a do triedy Node pridáme vlastnosť na ukladanie tejto informácie. [3]

5 Rozhodovanie

Rozhodovanie v hrách je založené na schopnosti NPC rozhodovať sa, čo urobia ďalej na základe vonkajších vplyvov. Tieto charaktery môžu mať niekoľko rôznych správaní, ako napríklad útočenie, státie, hliadkovanie, skrývanie sa, atď. Úlohou rozhodovacieho systému je rozhodnúť, ktoré z možných správaní je najvhodnejšie pre rôzne situácie v hre. Po výbere vhodného správania môže byť toto správanie vykonané rôznymi systémami umelej inteligencie, ktoré spolupracujú. [3] Napríklad, pomocou umelej inteligencie, ktorá sa stará o hľadanie ciest alebo o spúšťanie vhodných animácii. Charakter môže mať veľmi jednoduché pravidlá slúžiace na výber vhodného správania. Ako príklad si môžeme uviesť jednoduché správanie zvierat v niektorých hrách. Tieto zvieratá sa môžu rozhodovať na základe vzdialenosti od hráča. Pokiaľ hráč neprekročí určitú vzdialenosť, zvieratá sa kľudne pasú, v opačnom prípade urobia pár krokov, aby sa vzdialili. Príkladom komplikovanejšieho rozhodovania môžu byť nepriatelia v hre Call of Duty, ktorí používajú rôzne stratégie ako zneškodniť hráča. Napríklad, zreťazenie niekoľkých akcií ako krytie sa pred streľbou hráča a zároveň hádzanie granátov. Niektoré rozhodnutia môžu vyžadovať hľadanie cesty na dokončenie. Útok na blízko vyžaduje, aby sa NPC priblížil k svojmu cieľu na určitú vzdialenosť a následne ho napadol. Iné môžu byť vykonané len pustením animácie ako napríklad otváranie dverí. Na vytvorenie funkčného rozhodovacieho systému existuje mnoho techník.

V tejto bakalárskej práci sa pozrieme na konečné automaty a rozhodovacie stromy. Oba z týchto prístupov majú svoje výhody aj nevýhody. Výhodou konečných automatov je veľmi jednoduchá implementácia, ich nevýhodou je slabá škálovateľnosť. Túto slabinu majú z dôvodu mnohých závislostí pri prechode medzi jednotlivými stavmi. Škálovateľnosť a znovupoužiteľnosť jednotlivých úloh patrí medzi silné stránky rozhodovacích stromov. Medzi slabšie stránky rozhodovacích stromov môžeme zaradiť ich implementáciu. Nie je to z dôvodu, že by boli náročné logikou, no z dôvodu, že predtým, ako sa môže pustiť programátor do implemenácie, potrebuje sa zoznámiť s frameworkom pre tvorbu rozhodovacích stromov alebo si vytvoriť svoj vlastný.

5.1 Konečné automaty

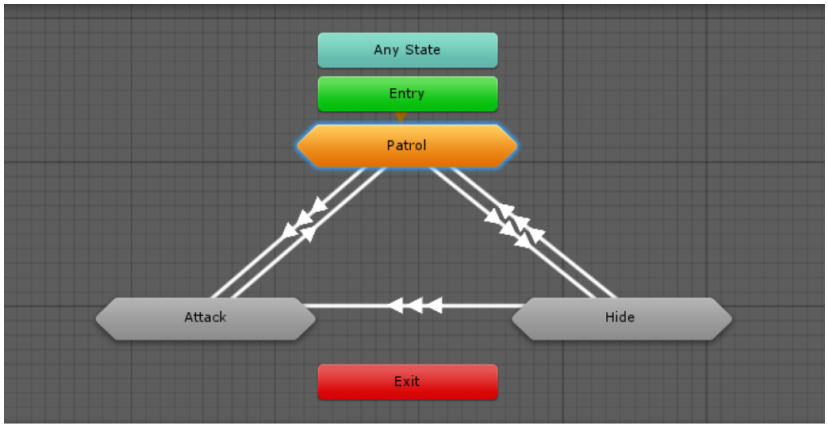
Konečné automaty patria k jednému z prvých rozhodovacích systémov používaných v hrách. Tento prístup si stále nachádza svoje miesto aj v dnešných moderných hrách v rôznych systémoch AI. Je to aj z dôvodu jednoduchej implementácie, pochopenia a debugovania. [14] Myšlienka konečných automatov spočíva v konečnom počte stavov, do ktorých sa konečný automat dokáže dostať. Tieto stavy sú prepojené do grafu jednotlivými prechodmi. Každý konečný automat má počiatočný stav, v ktorom začína svoje rozhodovanie. Jednotlivé rozhodnutia, do ktorého stavu sa posunie ďalej, koná

na základe podmienok, ktoré si stanovíme pre jednotlivé prechody. Konečný automat sa v jednom čase môže nachádzať len v jednom stave. [2]

Jednoduchý konečný automat obsahuje štyri komponenty:

- **Stavy:** Množina stavov z ktorých si herná entita môže vyberať.
- **Prechody:** Komponent definujúci vzťahy medzi jednotlivými stavmi.
- **Pravidlá:** Pravidlá sa používajú na spustenie prechodu medzi stavmi.
- **Udalosti:** Jednotlivé udalosti, ktoré kontrolujú pravidlá. Napríklad kontrola, či je hráč dostatočne blízko na útok, alebo na prenasledovanie. [2]

Ako príklad si môžeme uviesť použitie pre rozhodovanie klasického NPC herného charakteru v strielačke. Jeho jednotlivé stavy môžu vyzerať aj ako na obrázku (Obr. 5.1).

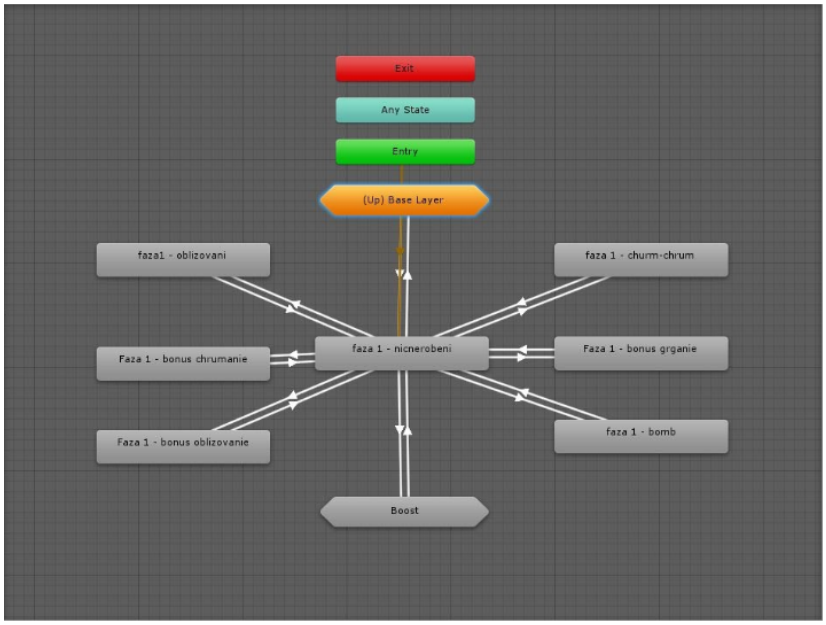


Obr. 5.1 Príklad konečného automatu.

Na obrázku si môžeme všimnúť tri základné stavy: **Patrol (hliadka)**, **Attack (útok)**, **Hide (skrývanie sa)**. Jednotlivé stavy sú prepojené prechodmi, ktoré definujú vzťahy medzi danými stavmi. Prechod medzi stavmi môže byť definovaný jednou alebo viacerými podmienkami. Napríklad, prechod zo stavu Patrol do stavu Attack nastane, ak je hodnota vzdialenosti medzi NPC, ktorý používa tento konečný automat, a iného objektu v hre menšia ako 20. Ako príklad prechodu s použitím viacero podmienok si môžeme uviesť prechod zo stavu Patrol do stavu Hide, ktorý sa uskutoční ak bude vzdialenosť medzi NPC a iného objektu viac ako 20 a zároveň momentálna životná sila NPC bude nižšia ako maximálna životná sila.

Okrem rozhodovania sa konečné automaty v hernej umelej inteligencii používajú aj na kontrolu jednotlivých animácií. Ako príklad si môžeme uviesť hru Hangry, ktorej

cieľom bolo nakŕmiť Suma. Na základe jedla, ktorým ste ho kŕmili, Sumo prehrával rôzne animácie. Ak ste mu dali niečo chutné, začal sa zalizovať, naopak, ak dostal niečo čo mu nechutilo, zatváril sa kyslo. V prípade, že bol dostatočne najedený, konečný automat sa presunul do pod-automatu. S implementáciou v animation controller komponente, môže takýto automat vyzerať aj ako na obrázku (Obr. 5.2).



Obr. 5.2 Konečný automat na riadenie animácií.

5.2 Rozhodovacie stromy

Rozhodovacie stromy patria medzi ďalšie techniky používané na reprezentáciu a kontrolu NPC. Stali sa populárne vďaka využitiu v AAA hrách ako Halo a Tom Clancy's The Division. Podobajú sa konečným automatom, avšak konečné automaty používajú priamočiaru cestu definujúcu logiku NPC charakteru, postavenú na rôznych stavoch a prechodoch medzi nimi. Konečné automaty, ako sme si už spomenuli sú náročné na škálovanie a opätovné použitie. Pri rozširovaní už vytvoreného rozhodovania by sme narazili na mnoho problémov týkajúcich sa závislostí jednotlivých stavov. Rovnako by sme mali tažkosti aj pri vytváraní komplexnejších rozhodovaní, keďže konečné automaty sa v prípade viacerých stavov stávajú nečitateľnými. Pri tvorbe komplexnejších rozhodovacích systémov potrebujeme použiť škálovateľnejší prístup. Aj preto rozhodovacie stromy predstavujú lepšie riešenie ako implementovať rozhodovací systém, ktorý v budúcnosti môže rásť na svojej komplexnosti.

Základným elementom rozhodovacích stromov sú úlohy (pri konečných automatoch sú to stavy). Úlohy sú spolu prepojené riadiacimi uzlami v stromovej štruktúre. Po-

známe veľa druhov často používaných uzlov, ako sú napríklad sequenčné, selektory, paralelný dekoratér. Úlohy sú listami stromu. Jednoduché správanie rozhodovacích stromov pozostáva z troch druhov úloh: **Podmienky**, **Akcie**, **Kompozity**.

Podmienky môžu kontrolovať ako blízko sa nachádza nepriateľ, v akom stave je hráčov charakter (koľko má života, munície) a podobne. Každá z týchto kontrôl musí byť implementovaná ako samostatná úloha, zvyčajne s nejakou parametrizáciou, takže môže byť použitá viackrát. Každá podmienka vráti "úspech", ak je jej podmienka splnená, a "neúspech", ak nie je splnená.

Akcie môžu existovať pre animácie, pohyb charakteru, na zmenu parametrov charakteru, na hľadanie cesty. Rovnako ako pri podmienkach, každá akcia potrebuje svoju vlastnú implementáciu. Nie každá akcia môže uspieť. Z toho dôvodu sa odporúča pred spustením samotnej akcie skontrolovať vhodnosť situácie podmienkou. Na druhej strane je možné napísať akciu, ktorá zlyhá ak nie je vykonaná/dokončená.

Kľúčový rozdiel medzi rozhodovacími stromami a konečným automatom je, že rozhodovacie stromy používajú jediné spoločné rozhranie pre všetky úlohy. To znamená, že všetky podmienky a akcie môžu byť kombinované do skupín a pri tom nepotrebujú vedieť, aké iné podmienky a akcie sa nachádzajú v rozhodovacom strome. Podmienky aj akcie sú umiestnené v listových uzloch stromu. Väčšina vetiev sa skladá z kompozitných uzlov. Tieto sa starajú o kolekcie úloh (podmienky, akcie, ďalšie kompozity) a správanie je postavené na správaní ich potomkov. Na rozdiel od akcií a podmienok nájdeme len hŕstku kompozitných úloh v rozhodovacom strome. Je to aj z dôvodu, že už len s malým množstvom kompozít môžeme vytvoriť veľmi sofistikované správanie.

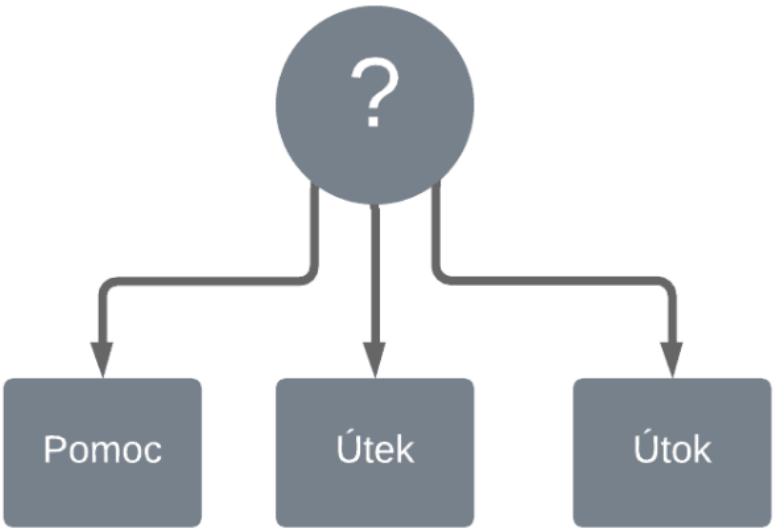
V praxi poznáme dva najčastejšie typy kompozitných úloh a to sú selektory a sequencie. Obe úlohy majú vlastných potomkov, po ktorých dokončení vracajú stavový kód na základe ktorého sa kompozita rozhodne, či bude pokračovať s daným potomkom alebo vráti hodnotu.

Selektor spúšťa potomkov z ľava do prava a hľadá aspoň jedného, ktorý mu vráti "úspech". To znamená, že pokiaľ selektoru potomok vráti že "zlyhal", tak sa selektor presunie na jeho rovesníka po pravej strane. Ak mu tento potomok vráti "úspech", selektor sa ukončí so stavovým kódom "úspech". Ak všetci potomkovia zlyhajú, selektor vráti stavový kód "neúspech".

Sequencia rovnako spúšťa potomkov z ľavej strany do pravej. Podmienkou sequencie, aby vrátila "úspech"je, aby všetci potomkovia skončili "úspechom". Naopak celá sequencia vráti "zlyhanie" hneď ako jeden z jej potomkov vráti hodnotu "zlyhanie". Ak sa vykonajú všetci potomkovia sequencie s návratovou hodnotou "úspech", tak celá sequencia vracia "úspech". [3]

Selektory sa používajú na výber jednej z možných akcií, ktorá je úspešná. Selektor

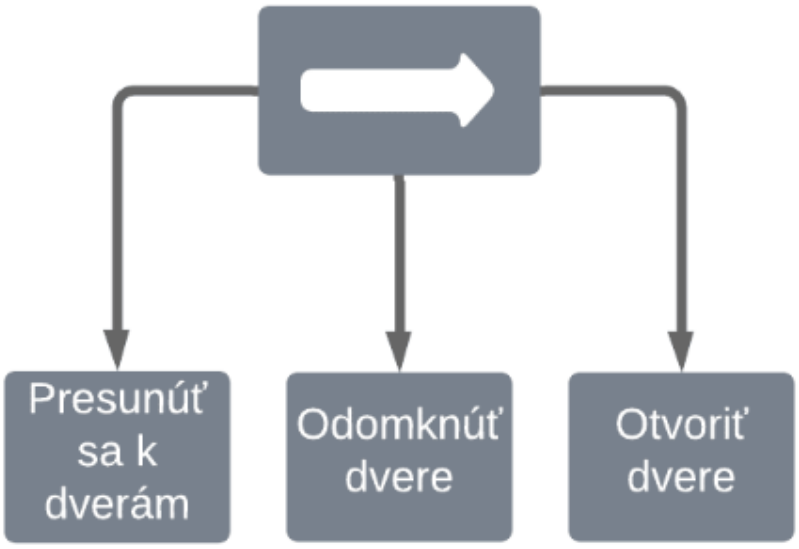
môže reprezentovať NPC, ktorý stráži nejaký objekt. Správanie takéhoto NPC môže pozostávať z úloh ako prenasledovanie, hliadkovanie a strieľanie. Selektor najskôr vyskúša akciu prenasledovanie. Ak táto akcia zlyhá (nemá koho prenasledovať), presunie sa na akciu hliadkovanie. Ak táto akcia uspeje, selektor skončí. Ak vyčerpáme všetky možnosti bez úspechu, potom selektor zlyhá.



Obr. 5.3 Príklad fungovania selektoru.

Pozrime sa na príklad fungovania selektoru na obrázku (Obr. 5.3). Povedzme, že NPC zbadalo hráča a tým sme sa dostali do vetvy rozhodovania selektoru. Na obrázku môžeme vidieť, že selektor najskôr vyskúša privolať pomoc, ak uspeje, selektor sa ukončí. Ak nie, presunieme sa na úlohu útek. Ak ani útek neuspeje, tak vyskúša zaútočiť. Ak aj posledná akcia zlyhá, tak celý selektor zlyháva.
Sequencie predstavujú sériu úloh, ktoré musia byť vykonané, aby sequencia vrátila návratovú hodnotu "úspech". Príkladom na sequenciu môže byť hliadkovanie. Ak sa chceme niekam presunúť, potrebujeme nájsť miesto, kam chceme ísť a presunúť sa na dané miesto. Ak sa NPC nedokáže presunúť na hliadkovacie miesto, celá sequencia zlyháva. Jedine v prípade, ak všetky úlohy v sequencii sú úspešné, tak sa sequencia považuje za úspešnú.

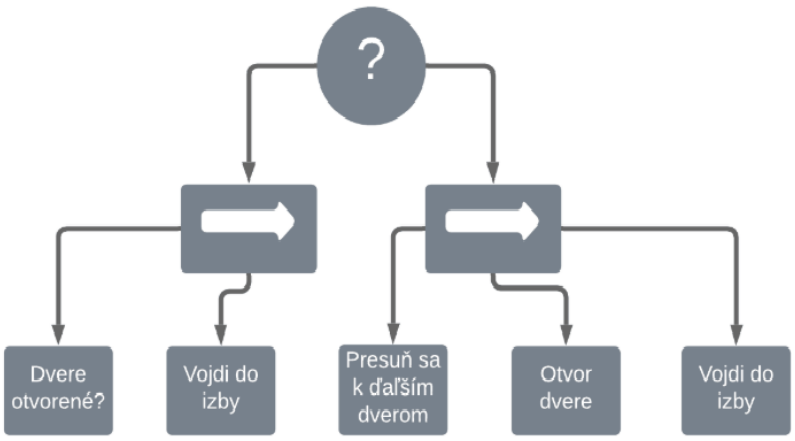
Obrázok (Obr. 5.4) ukazuje jednoduchý príklad použitia sequenčného uzla v prípade, keď budeme chcieť, aby náš charakter odomkol dvere. V tomto strome prvá potomkovská úloha presunie NPC k správnym dverám. Následne, ak táto úloha uspeje a NPC sa nachádza pri dverách, tak sa v sequencii posunieme na úlohu, ktorá sa postará o



Obr. 5.4 Príklad fungovania sequencie.

odomknutie dverí. Ak sa aj odomykanie vykoná bez ťažkostí, v sequencii sa presunieme na poslednú úlohu, otváranie dverí. V prípade, že aj posledná úloha skončí úspechom, tak celá sequencia končí úspešne.

V ďalšom strome na obrázku (Obr. 5.5) môžeme vidieť kombináciu použitia selektoru a sequencií. Už pri malom počte úloh dokážeme v rozhodovacích stromoch vytvoriť zaujímavé správanie. V tomto príklade rozhodovací strom bude reprezentovať nepriateľa, ktorý prenasleduje hráča. [3] Na začiatku môžete vidieť rodičovský selektor, ktorý má ako potomkov dve sequencie. Tento prístup je výbornou demonštráciou rozhodovania sa medzi viacerými činnosťami pozostávajúcimi z viacerých akcií.

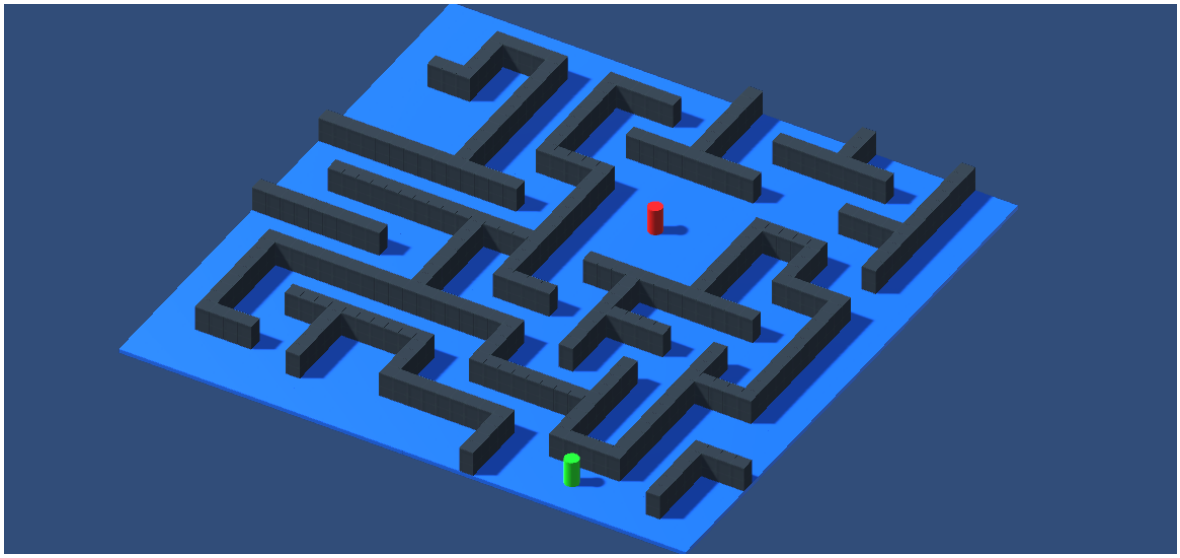


Obr. 5.5 Kombinácia selektoru a sequencií.

II. PRAKTICKÁ ČASŤ

6 Hľadanie ciest

Na demonštráciu fungovania algoritmov na hľadanie ciest sme v praktickej ukážke vytvorili jednoduchú scénu zloženú z vygenerovaného grafu, na ktorom vizuálne demonštrujeme hľadanie cesty v každej iterácii bludiska zo štartovacej do cieľovej pozície (Obr. 6.1).



Obr. 6.1 Scéna

6.1 Graf

Graf je generovaný z jednotlivých kociek, ktoré predstavujú uzly. O jeho generovanie sa nám starajú dva scripty. Prvý, WGridSetup.cs dedí od triedy MonoBehavior a teda ho môžeme pripnúť v scéne na akýkoľvek herný objekt. V tomto scripte sa nachádzajú deklarácie premenných, ktoré budeme potrebovať na vygenerovanie grafu. Obsahuje aj funkciu InstantiateNode, ktorá nám naklonuje počet uzlov potrebný na vygenerovanie mriežky podľa jej x a y veľkosti.

```
public class WGridSetup : MonoBehaviour
{
    public Transform StartPosition;
    public LayerMask WallMask;
    public Vector2 GridWorldSize;
    public GameObject Node;
    public List<GameObject> InstantiateNode(int gridSizeX, int
    ↪ gridSizeY)
    {
        int amount = Mathf.RoundToInt(gridSizeX * gridSizeY);
        List<GameObject> clonedObjects = new List<GameObject>();
        for (int i = 0; i < amount; i++)
        {
            GameObject @object = Instantiate(Node);
            @object.SetActive(false);
            clonedObjects.Add(@object);
```

```
        }
        return clonedObjects;
    }
}
```

Druhý script WGrid.cs obsahuje všetky dôležité funkcie pre generáciu nášho grafu a aj prácu s ním. Je to generická trieda, ktorá ako svoj generický parameter berie typ Node (uzol), ktorý je rodičovský typ pre uzol A* a Dijkstra. Nastavenia o veľkosti mriežky a typu uzlu preberá v konštruktore od triedy WGridSetup. Veľmi dôležitá funkcia v tomto scripte je CreateGrid(Transform gridObj), ktorá si ako parameter berie pozíciu, kde má byť vygenerovaná mriežka. V tejto funkcii preberáme jednotlivé uzly pomocou funkcie getFreeObjectNode(), ktoré ešte neboli použité. Následne im nastavujeme pozíciu a či predstavujú pre algoritmus prekážku alebo nie.

```
  private void CreateGrid(Transform gridObj)
    {
        NodeArray = new T[iGridSizeX, iGridSizeY];
        Vector3 bottomLeft = gridObj.position - Vector3.right *
            GridWorldSize.x / 2 - Vector3.forward * GridWorldSize.y /
 ↪ 2;
        for (int x = 0; x < iGridSizeX; x++)
        {
            for (int y = 0; y < iGridSizeY; y++)
            {
                Vector3 worldPoint = bottomLeft + Vector3.right *
                    (x * fNodeDiameter + nodeRadius) + Vector3.forward
 ↪  * (y * fNodeDiameter + nodeRadius);
                bool Wall = Physics.CheckSphere(worldPoint, nodeRadius
 ↪ , WallMask);
                T node = getFreeObjectNode();
                node.IsWall = Wall;
                node.Position = worldPoint;
                node.gridX = x;
                node.gridY = y;
                NodeArray[x, y] = node;
                node.gameObject.transform.localPosition = worldPoint;
            }
        }
    }

 private T getFreeObjectNode()
    {
        foreach (var obj in ClonedNodes)
        {
            if (!obj.activeInHierarchy)
            {
                obj.SetActive(true);
                return obj.GetComponent<T>();
            }
        }
        return null;
    }
```

Ďalšou dôležitou funkciou je NodeFromWorldPoint(Vector3 worldPos), ktorá vracia pozíciu uzlu v grafe na základe worldPos, ktorá predstavuje pozíciu v scéne. Túto funkciu budeme používať priamo vo funkciách algoritmov A* a Dijkstra, aby sme získali uzly, na ktorých sa nachádzajú objekty, ktoré predstavujú štart a cieľ.

```
  public T NodeFromWorldPoint(Vector3 worldPos)
```

```
    {
        float xPos = ((worldPos.x + GridWorldSize.x / 2) /
↪ GridWorldSize.x);
        float yPos = ((worldPos.z + GridWorldSize.y / 2) /
↪ GridWorldSize.y);
        xPos = Mathf.Clamp01(xPos);
        yPos = Mathf.Clamp01(yPos);
        int x = Mathf.RoundToInt((iGridSizeX - 1) * xPos);
        int y = Mathf.RoundToInt((iGridSizeY - 1) * yPos);
        return NodeArray[x, y];
    }
```

Posledná z dôležitých funkcií je GetNeighboringNodes(T currrentNode), ktorá nám vráti list susedných uzlov k uzlu, pre ktorý momentálne vykonávame iteráciu.

```
public List<T> GetNeighborNodes(T currrentNode)
    {
        List<T> NeighborList = new List<T>();
        int checkX;
        int checkY;
        checkX = currrentNode.gridX + 1;
        checkY = currrentNode.gridY;
        if (checkX >= 0  checkX < iGridSizeX)
        {
            if (checkY >= 0  checkY < iGridSizeY)
            {
                NeighborList.Add(NodeArray[checkX, checkY]);
            }
        }
        checkX = currrentNode.gridX - 1;
        checkY = currrentNode.gridY;
        if (checkX >= 0  checkX < iGridSizeX)
        {
            if (checkY >= 0  checkY < iGridSizeY)
            {
                NeighborList.Add(NodeArray[checkX, checkY]);
            }
        }
        checkX = currrentNode.gridX;
        checkY = currrentNode.gridY + 1;
        if (checkX >= 0  checkX < iGridSizeX)
        {
            if (checkY >= 0  checkY < iGridSizeY)
            {
                NeighborList.Add(NodeArray[checkX, checkY]);
            }
        }
        checkX = currrentNode.gridX;
        checkY = currrentNode.gridY - 1;
        if (checkX >= 0  checkX < iGridSizeX)
        {
            if (checkY >= 0  checkY < iGridSizeY)
            {
                NeighborList.Add(NodeArray[checkX, checkY]);
            }
        }
        return NeighborList;
    }
```

Ostatné doplnkové funkcie v triede sa starajú o správne zafarbenie jednotlivých uzlov. Napríklad funkcia ChangeColor(T node, Color color) nám zmení farbu daného uzlu na nami definovanú. Túto funkciu používame primárne na zvýraznenie uzlov, cez ktoré prechádza algoritmus počas hľadania. Na zobrazenie konečnej cesty používame funkciu ShowFinalPath(), ktorú voláme až po získaní najkratšej cesty. Táto funkcia zafarbí uzly, ktoré patria do finálnej cesty na červeno.

```
  public void ShowFinalPath()
    {
        foreach (var node in NodeArray)
        {
            if (FinalPath != null)
            {
                if (FinalPath.Contains(node))
                    ChangeColor(node, Color.red);
            }
        }
    }
```

Poslednou funkciou je funkcia CheckWalls(), ktorá po zavolaní skotroluje všetky uzly, či sa na nich nachádzajú prekážky alebo nie.

Každý uzol v grafe má na sebe pripnutý script, dediaci od triedy Node.cs. Táto rodičovská trieda predstavuje základné rozhranie pre uzly, ktoré používame pri Dijkstra alebo A* algoritme. Nájdeme v nej premenné ukladajúce pozíciu v grafe, v hernej scéne, alebo informáciu, či daný uzol predstavuje prekážku alebo nie. Potomkovské triedy ANode.cs (používaná pri A*) a DNode.cs (používaná pri Dijkstra) sú si veľmi podobné. Každá obsahuje informáciu o rodičovi (v algoritme predchádzajúci uzol ktorým sme sa na momentálny uzol dostali). Uzol je rovnakého typu ako trieda. Ďalej obsahujú informáciu o cene cesty presunu na pozíciu uzlu, na ktorom sú pripnuté. Napríklad pri ANode máme premenné gCost (cena G), hCost (cena H) a FCost (cena F) ktorá vždy pri volaní vypočíta celkovú odhadovanú cenu uzlu sčítaním gCost a hCost. V DNode na uchovávanie ceny cesty máme premennú CostSoFar.

```
public class Node : MonoBehaviour
{
    public int gridY;
    public int gridX;
    public bool IsWall;
    public Vector3 Position;
}
```

6.2 Algoritmy

Rodičovskú triedu pre algoritmy predstavuje trieda PathfindingBase.cs, ktorá slúži ako základné rozhranie pre A* a Dijkstru. V tejto triede nájdeme základné premenné, ktoré používajú oba algoritmy, funkcie na obsluhu grafu a taktiež kontrolu ovládania cieľovej pozície. V každej scéne môžeme presúvať cieľovú pozíciu ľavým tlačidlom myši. Túto funkcionalitu máme pokrytú vo funkcii MoveTarget(). Po presunutí cieľovej pozície sa nám automaticky spustí algoritmus a začne hľadať cestu k danej pozícii. Táto kontrola je pokrytá vo funkcii Update(), ktorá sa automaticky volá a kontroluje stisk tlačidla každý snímok (frame).

```
 protected void MoveTarget()
    {
```

```
        Vector3 mouse = Input.mousePosition;
        Ray ray = Camera.main.ScreenPointToRay(mouse);
        RaycastHit hit;
        if (Physics.Raycast(ray, out hit, Mathf.Infinity,
↪ TargetPositionMoveMask))
        {
            Node node = hit.collider.gameObject.GetComponent<Node>();
            if (!node.IsWall)
            {
                Transform nodeTransform = node.gameObject.transform;
                TargetPosition.position = new Vector3(nodeTransform.
↪ position.x,
                    TargetPosition.position.y, nodeTransform.position.
↪ z);
            }
        }
    }
```

Ďalšia dôležitá funkcia, ktorá stojí za zmienku je GetManhattanDistance(Node currentNode, Node neighborNode). Túto funkciu používame na počítanie ceny cesty medzi aktuálnym a susedným uzlom metódou Manhattan distance. Manhattan disctance je vzdialenosť medzi dvomi bodmi meraná pozdĺž osi v pravom uhle.

```
protected int GetManhattanDistance(Node currentNode, Node
 ↪ neighborNode)
  {
      int x = Mathf.Abs(currentNode.gridX - neighborNode.gridX);
      int y = Mathf.Abs(currentNode.gridY - neighborNode.gridY);
      return x + y;
  }
```

6.2.1 Dijkstra

Dijkstra algoritmus je zrealizovaný v triede DijkstraPathfinding.cs vo funkcii FindPath, ktorá má návratovú hodnotu typu IEnumerator. Návratovú hodnotu sme zvolili z dôvodu, že chceme s funkciou pracovať ako s coroutinou. Courotine je funkcia, ktorá má schopnosť pozastaviť vykonávanie funkcie v určitom bode, vrátiť kontrolu naspäť Unity, počkať určitú dobu a následne pokračovať tam, kde bolo vykonávanie prerušené na nasledujúcom snímku (frame). Courotiny sa môžu používať na rozloženie rôznych úloh na časové úseky. Rovnako si vedia nájsť aj svoje miesto pri optimalizácii. [11] V našom prípade používame tento typ funkcie na zvizualizovanie funkčnosti algoritmov, kde pri každom posune na nový uzol pozastavíme cyklus na jednu desatinu sekundy. Robíme to z dôvodu, aby bolo možné vizuálne zdemonštrvať uzly, cez ktoré už algoritmus preiteroval.

```
protected override IEnumerator FindPath()
   {
       DNode startNode = grid.NodeFromWorldPoint(StartPosition.
 ↪ position);
       DNode targetNode = grid.NodeFromWorldPoint(TargetPosition.
 ↪ position);
       List<DNode> openList = new List<DNode>();
       List<DNode> closedList = new List<DNode>();
```

```
        openList.Add(startNode);
        while(openList.Count > 0)
        {
            DNode currentNode = openList[0];
            for (int i = 0; i < openList.Count; i++)
            {
                if (openList[i].CostSoFar < currentNode.CostSoFar)
                    currentNode = openList[i];
            }
            closedList.Add(currentNode);
            openList.Remove(currentNode);
            if (currentNode == targetNode)
            {
                GetFinalPath(startNode, targetNode);
                break;
            }
            foreach (DNode neighborNode in grid.GetNeighborNodes(
↪ currentNode))
            {
                ChangeColor(neighborNode, Color.blue);
                if (neighborNode.IsWall || closedList.Contains(
↪ neighborNode))
                    continue;
                int moveCost = currentNode.CostSoFar +
↪ GetManhattanDistance(currentNode, neighborNode);
                if (moveCost < neighborNode.CostSoFar || !openList.
↪ Contains(neighborNode))
                {
                    neighborNode.CostSoFar = moveCost;
                    neighborNode.Parent = currentNode;
                    if (!openList.Contains(neighborNode))
                    {
                        openList.Add(neighborNode);
                    }
                }
            }
            yield return new WaitForSeconds(.1f);
        }
        StopCoroutine("FindPath");
    }
```

6.2.2 A*

A* algoritmus je zrealizovaný v triede AStarPathfinding.cs v coroutine FindPath má návratovú hodnotu typu IEnumerator.

```
protected override IEnumerator FindPath()
    {
        ANode startNode = grid.NodeFromWorldPoint(StartPosition.
↪ position);
        ANode targetNode = grid.NodeFromWorldPoint(TargetPosition.
↪ position);
        List<ANode> OpenList = new List<ANode>();
        HashSet<ANode> ClosedList = new HashSet<ANode>();
        OpenList.Add(startNode);
        while (OpenList.Count > 0)
        {
            ANode currentNode = OpenList[0];
            for (int i = 0; i < OpenList.Count; i++)
            {
                if (OpenList[i].FCost < currentNode.FCost
                    || OpenList[i].FCost == currentNode.FCost
↪ OpenList[i].hCost < currentNode.hCost)
```

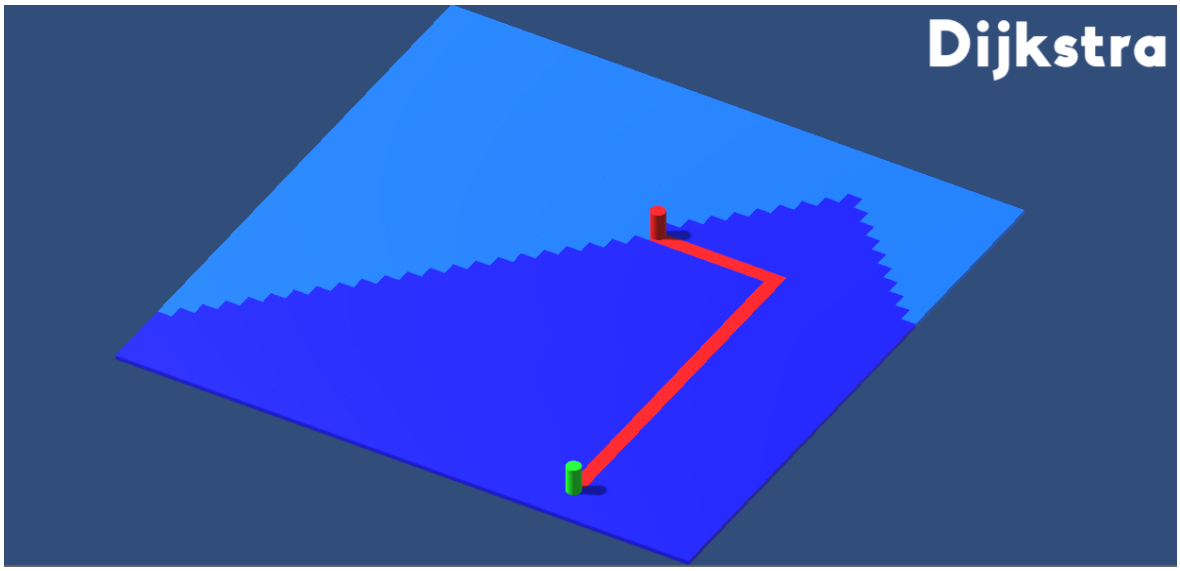
```
                {
                    currentNode = (OpenList[i]);
                }
            }
            ClosedList.Add(currentNode);
            OpenList.Remove(currentNode);
            if (currentNode == targetNode)
            {
                GetFinalPath(startNode, targetNode);
                break;
            }
            foreach (ANode neighborNode in grid.GetNeighborNodes(
↪ currentNode))
            {
                ChangeColor(neighborNode, Color.blue);
                if (neighborNode.IsWall || ClosedList.Contains(
↪ neighborNode))
                    continue;
                int MoveCost = currentNode.gCost +
↪ GetManhattanDistance(currentNode, neighborNode);
                if(MoveCost < neighborNode.gCost || !OpenList.Contains
↪ (neighborNode))
                {
                    neighborNode.gCost = MoveCost;
                    neighborNode.hCost = GetManhattanDistance(
↪ neighborNode, targetNode);
                    neighborNode.Parent = currentNode;
                    if(!OpenList.Contains(neighborNode))
                    {
                        OpenList.Add(neighborNode);
                    }
                }
            }
            yield return new WaitForSeconds(.1f);
        }
        StopCoroutine("FindPath");
    }
```

Už na prvý pohľad si môžeme všimnúť veľkú podobnosť týchto dvoch algoritmov, ako sme spomínuli v teoretickej časti. Oba algoritmy fungujú na rovnakom princípe. Pracujú s openListom a closedListom, z openListu v cykle while vyberú uzol s najnižšou váhou, odstránia ho z openListu a pridajú do closedListu. Ak daný uzol nepredstavuje cieľ, prezerajú jeho susedné uzly. Pri hľadaní susedných uzlov kontrolujú, či náhodou nepredstavujú prekážky alebo či už niesú obsiahnuté v closedListe. Následne počítajú cenu presunu na daný susedný uzol, aktualizujú jeho rediča a pridajú ho do openListu v prípade, že sa tam už nenachádza.

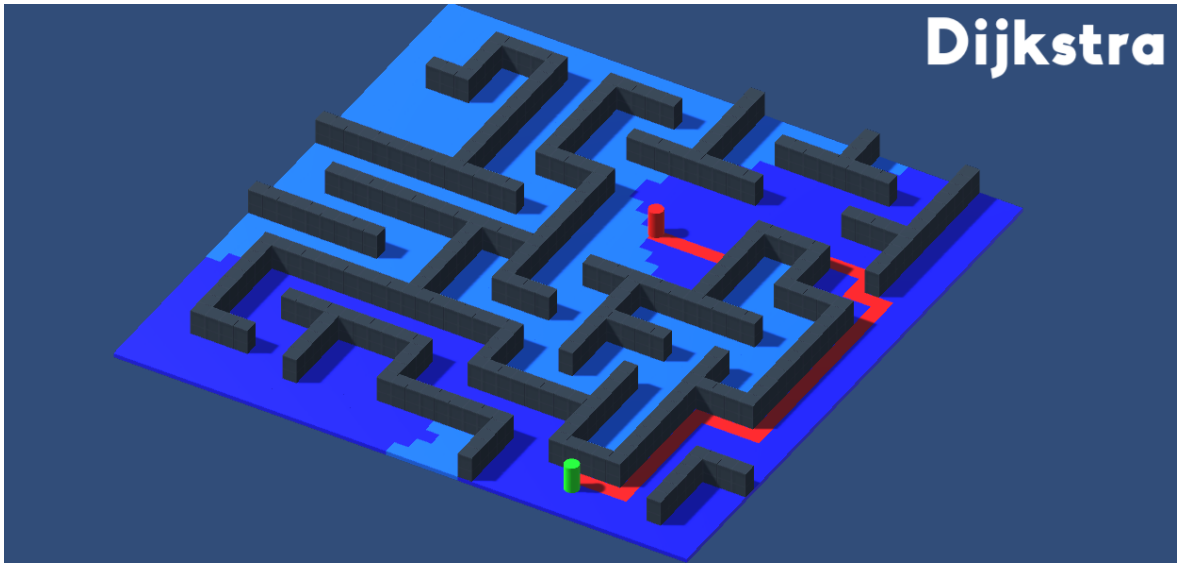
Rozdiel nastáva pri počítaní cien presunu na jednotlivé uzly. Ako sme si už spomenuli, A* počíta cenu presunu na daný uzol sčítaním ceny presunu z iterovaného uzla a ceny presunu z uzla na ktorý sa chceme presunúť do cieľa. Naopak, Dijkstra používa pri počítaní ceny uzlu len jednu hondotu. Tou je cena presunu na uzol, ktorú sčítame s celkovou cenou uzlu, na ktorom sa nachádzame. Už len tento drobný rozdiel v počítaní váh dokáže spraviť A* algoritmus oveľa efektívnejším pri hľadaní najkratšej cesty z miesta na miesto.

6.2.3 Benchmark

Na demonštráciu tejto efektivity sme zmerali a porovnali čas a počet iterácií potrebných na nájdenie cesty. Algoritmy sme otestovali v dvoch prostrediach. V jednom bez prekážok na voľnej ploche a v druhom v bludisku (Obr. 6.2) (Obr. 6.3). Na obrázkoch môžeme vidieť zelený a červený valec. Zelený valec predstavuje štartovaciu pozíciu, červený cieľovú. Tmavomodré štvorčeky predstavujú uzly, cez ktoré algoritmus preiteroval pri hľadaní cesty. Bledomodré štvorčeky predstavujú uzly, ktoré algoritmus ešte nepozná. Červenou farbou je zvýraznená najkratšia cesta. Oba algoritmy hľadajú cestu na rovnakú pozíciu. Obrázky boli tvorené pri použití Dijkstra algoritmu.



Obr. 6.2 Benchmark bez prekážok



Obr. 6.3 Benchmark s prekážkami

Keď sa pozrieme na dáta z benchmarku pri teste bez prekážok je jasné, že A* algoritmus bol výkonejší. Na nájdenie cesty do cieľovej pozície potreboval len 2 milisekundy a 33 iterácií kdežto pri Dijkstrovi to bolo omnoho viac (Tab. 6.1).

Tab. 6.1 Hodnotenie algoritmu v prostredí bez prekážok

Algoritmus	Čas [ms]	Počet iterácii
Dijkstra	127	867
*A**	2	33

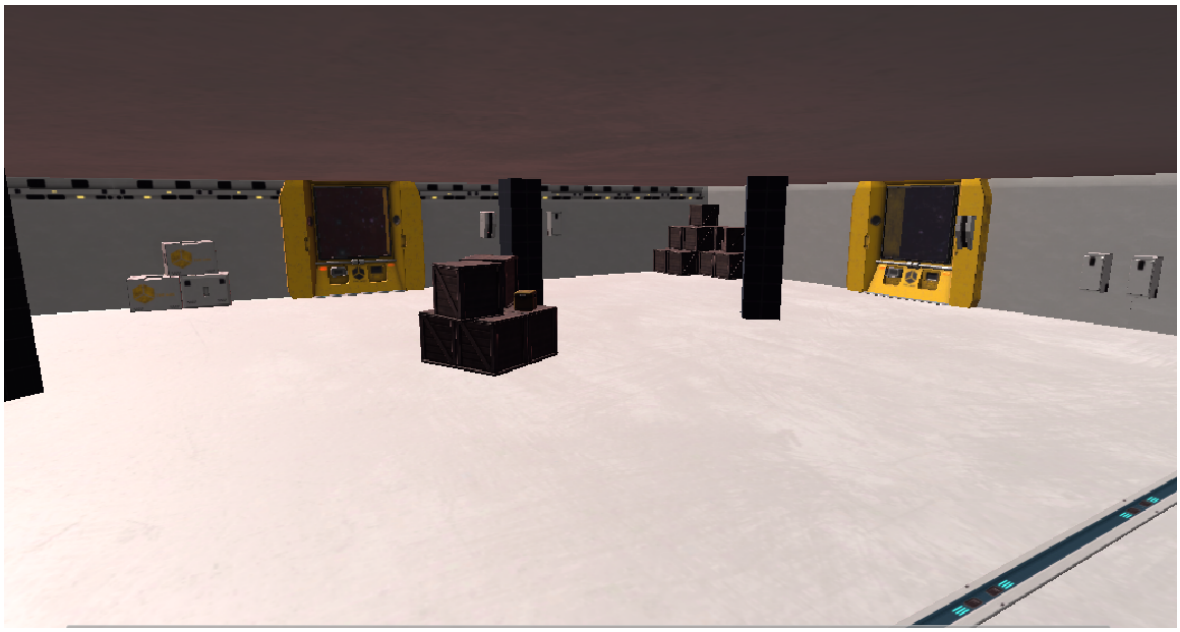
V teste s prekážkmi bol A* algoritmus znovu výkonejší. No ako môžeme vidieť z dát, pomer rozdielu dát medzi algoritmami už nieje taký veľký ako v prípade bez prekážok (Tab. 6.2).

Tab. 6.2 Hodnotenie algoritmu v prostredí s prekážkami

Algoritmus	Čas [ms]	Počet iterácii
Dijkstra	59	571
*A**	8	241

7 Rozhodovanie

Na porovnanie implementácie konečného automatu a rozhodovacích stromov sme si vytvorili jednoduché demo, v ktorom budeme demonštrovať funkcionalitu a implementáciu na NPC (Obr. 7.1). Pri navrhovaní rozhodovania pre NPC sme sa inšpirovali správaním a rozhodovaním klasického charakteru zo strielačky. Ak chceme, aby sa náš charakter začal rozhodovať alebo striedať rôzne správania, potrebujeme ho vložiť do prostredia, ktoré ho podnieti k daným rozhodnutiam. V našom prípade sme si na podnety zvolili uživateľa, ktorý svojimi akciami bude vplývať na rozhodovanie nášho charakteru. Pri tvorbe a návrhu sme prešli rôznymi iteráciami, pri ktorých sme menili a rozširovali navrhnuté správanie. Na nasledujúcich riadkoch si prejdeme cez jednotlivé iterácie a následne porovnáme zložitosť údržby/zmien pri konečných automatoch a rozhodovacích stromoch.



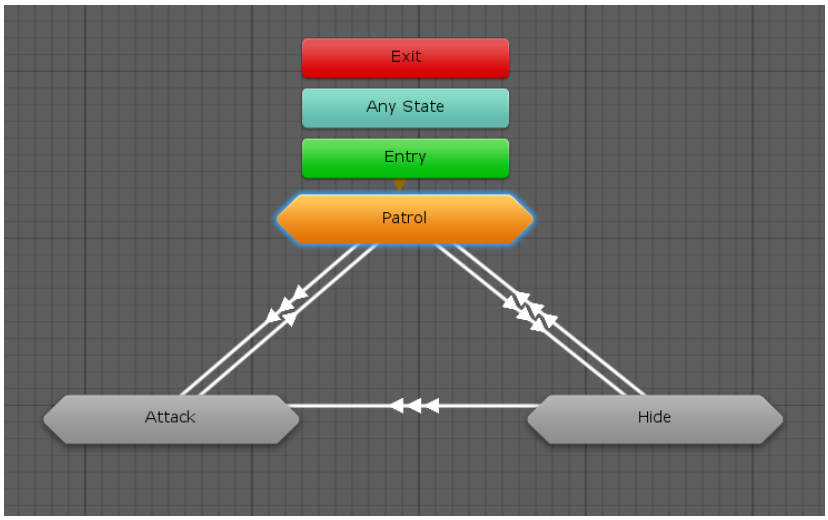
Obr. 7.1 Scéna

Predtým ako sa pustíme do samotných implementácií rozhodovania, pozrime sa najskôr na triedy a funkcie, ktoré používame ako hlavný motor na vykonávanie rozhodnutí naším NPC. Na pohyb NPC po scéne používame NavMeshAgent komponent, ktorý je pripnutý na jednotlivých agentoch. Graf, na ktorom sa môžu charaktery pohybovať, je vytvorený pomocou NavMesh. Na prístup a prácu s NavMeshAgentom sme si vytvorili Motor.cs triedu, ktorá zároveň slúži aj ako rodičovská trieda všetkým NPC charakterom, ktoré používame v tejto bakalárskej práci. Táto trieda obsahuje funkcie, vďaka ktorým vieme posielať charakter na rôzne pozície pomocou funkcie SendCharacterToDestination(Transform destination). Dokážeme zistiť momentálnu pozíciu charakteru funkciou GetCharacterDestination() a aj natočiť charakter na svoj cieľ funkciou Face-

Target(Transform target). V triede nájdeme aj funkciu, vďaka ktorej nastavíme na aký cieľ sa má NPC zamerať a podobne. Trieda má dvoch potomkov, v tejto kapitole si rozoberieme len jedného z nich a tým je EnemyMotor.cs. EnemyMotor.cs rozširuje triedu Motor.cs o funkcie a nastavenia spôsobu pohybu NPC. Napríklad funkcia ChangeStoppingDistance(AISettings_StoppingDistance aISettings_StoppingDistance), vďaka ktorej dokážeme zmeniť vzdialenosť od cieľa, pri ktorom sa charakter zastaví. Ďalej tu máme funkciu ChangeSpeed(AISettings_Speed aISpecifications_Speed), ktorá nám umožňuje zmeniť maximálnu rýchlosť pohybu NPC na základe rôznych situácií. V triede nájdeme aj dátovú štruktúru List (List<Point> PatrolPoints), ktorá nám uchováva zoznam miest, na ktorých môže NPC hliadkovať. Na obsluhu získavania vhodného miesta na hliadkovanie používame funkciu GetPatrolPoint(), ktorá vráti typ Point, čiže miesto na ktorom ešte NPC nehliadkoval. Potomkom tejto triedy, ktorú priamo používame v deme o rozhodovaní je trieda EMotor_BTDemo.cs, ktorá rozširuje EnemyMotor.cs o možnosť nájsť najvhodnejšie miesto na skrytie funkciou GetClosestHidePoint(). Ďalšou veľmi dôležitou triedou slúžiacou na ovládanie animácií a prístup k iným dôležitým scriptom ako HealthControl.cs alebo EnemyGun.cs je EnemyController.cs. Okrem spomenutých vecí, trieda nám ponúka aj funkciu getSpeed(EnemyMotor motor), ktorá nám vráti momentálnu rýchlosť charakteru, na základe ktorej vieme prispôsobiť animáciu k jeho pohybu. Potomkovskou triedou je EnemyController_BTDemo.cs, ktorá obsahuje referenciu na EMotor_BTDemo.cs script a vo funkcii Update() aktualizuje momentálnu rýchlosť NPC, pre správny výber animácií. Keďže sme sa pri návrhu správania inšpirovali strieľačkami, potrebujeme pridať možnosť strieľať a špecifikovať vlastnosti zbrane či už pre hráča alebo NPC. Táto funkcionalita je pokrytá v triede Gun.cs. Trieda obsahuje parametre špecifikujúce správanie zbrane, na ktorú daný script pripneme. Môžeme tam nájsť parameter charakterizujúci aké poškodenie spôsobí zbraň nepriateľovi, aký má dostrel, ako rýchlo dokáže strieľať, alebo aký efekt použije pri strieľaní. Rovnako v triede nájdeme referencie na Animator komponent, ktorý obsluhuje animácie zbrane pri strieľaní, ShootOrigin typu GameObject špecifikujúci pozíciu, z ktorej bude vychádzať lúč (ray) predstavujúci letiacu guľku. Okrem iného tu nájdeme aj funkciu Shoot(), ktorou vysielame lúče zo ShootOriginu smerujúce dopredu a kontrolujeme, či sme trafili nejaký fyzický objekt pred nami. Ak sme taký objekt trafili, skontrolujeme či má na sebe pripnutý HealthControl.cs script, ktorý obsahuje funkcie pracujúce so životnou energiou NPC. V prípade, že zasiahnutý objekt obsahuje spomenutý script, zavoláme funkciu TakeDamage(float damage), ktorá odčíta trafenému objektu poškodenie spôsobené našou zbraňou zo životnej energie. Od triedy Gun.cs dedia ďalšie dve triedy EnemyGun.cs a PlayerGun.cs slúžiace na špecifické prispôsobenie zbrane a streľby potrebám NPC a hráča.

7.1 Konečné automaty

Pre implementáciu konečných automatov a ich lepšiemu vizuálnemu zobrazeniu sme použili nástroj AnimationController, ktorý je zároveň aj súčasťou Unity3D. Ako to už z názvu vyplýva, AnimationController sa používa primárne na implementáciu a kontrolu animácií, no keďže animačné schémy fungujú aj na princípe konečného automatu, je to výborný nástroj pre demonštráciu nášho problému. Prvá verzia návrhu automatu obsahovala 4 základné stavy správania (Obr. 7.2), ktoré sa skladali z jednotlivých drobnejších akcií. Ako môžete vidieť na obrázku, tieto stavy sú Patrol (Hliadkovanie), Attack (Útok), Hide (Skrývanie sa). Pri implementácií jednotlivých stavov sme použili knižnicu z rodičovskej triedy StateMachineBehaviour, ktorá nám poskytla rozhranie na prácu s jednotlivými stavmi aj animáciami. V každom stave voláme triedu FSM_TransitionController, ktorá nám slúži ako rozhranie na komunikáciu medzi stavmi a volanie rôznych funkcií z EnemyController_BTDemo. Trieda tiež obsahuje funkciu Update(), v ktorej aktualizujeme premennú Distance a kontrolujeme životnú energiu NPC, na ktorom je pripnutá. Ak je táto energia menšia alebo rovná nule, aktivujeme trigger Dead, ktorý presunie NPC do stavu Die a spustí animáciu umierania.

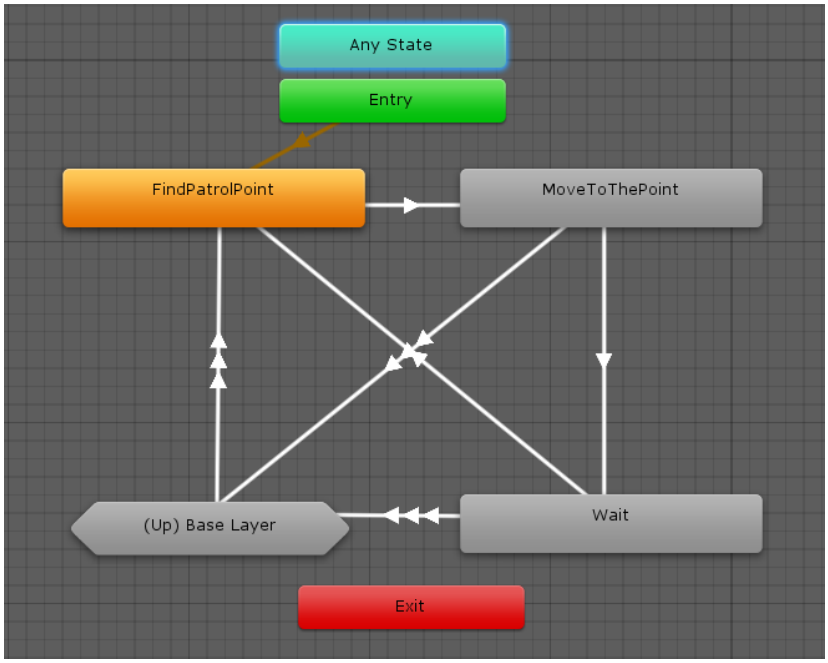


Obr. 7.2 Základný konečný automat

7.1.1 Patrol

FindPatrolPoint

Tento stav sa zaoberá hľadaním nasledujúcej pozície na hliadkovanie. Na začiatku vykonávania danej akcie, na ktorú je určený, si nastavíme premenú StoppingDistance, čo je vzdialenosť pri ktorej sa charakter zastaví od cieľového miesta, na Move. Následne zavoláme funkciu GetPatrolPoint, ktorá nám vráti miesto, na ktorom charakter ešte



Obr. 7.3 Patrol podautomat

nehliadkoval a ak také miesto existuje, aktivujeme trigger continue a presunieme sa do stavu MoveToThePoint.

```
public class FSM_FindPatrolPoint : StateMachineBehaviour
{
    private FSM_TransitionController fsm_Controller;
    override public void OnStateEnter(Animator animator,
             AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (fsm_Controller == null)
            fsm_Controller = animator.gameObject
            .GetComponent<FSM_TransitionController>();
        fsm_Controller.enemyController.motor
        .ChangeStoppingDistance(AISettings_StoppingDistance
        .Move);
        Point nextPoint = fsm_Controller.
             enemyController.motor.GetPatrolPoint();
        if (nextPoint != null)
        {
            fsm_Controller.nextPatrolPoint = nextPoint;
            fsm_Controller.nextPatrolPoint.Controlling = true;
            animator.SetTrigger("Continue");
        }
    }
}
```

MoveToThePoint

V tomto stave sa náš NPC začne presúvať na pozíciu vybranú predchádzajúcim stavom. Okrem samotného presunu na začiatku stavu meníme aj rýchlosť, ktorou sa bude NPC presúvať. Počas vykonávania stavu neustále aktualizujeme hodnotu vzdialenosti od hráča a kontrolujeme životnú hodnotu NPC (či na nás náhodou niekto neútočí a zá-

roveň nie je v dosahu). Ak zistíme, že momentálna hodnota životnej energie sa nerovná prednastavenej hodnote životnej energie, nastavíme IsSomeoneHitMe na true a tým sa automaticky presunieme do stavu GetClosestHidePoint, ktorý si rozoberieme neskôr. Keď sa NPC dostaví na zadaný bod, aktivujeme trigger continue ktorý nás presunie do stavu Wait.

```
public class FSM_MoveToThePoint : StateMachineBehaviour
{
    private FSM_TransitionController fsm_Controller;
    override public void OnStateEnter(Animator animator,
     AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (fsm_Controller == null)
            fsm_Controller = animator.gameObject.GetComponent<
   ↪ FSM_TransitionController>();
        fsm_Controller.enemyController.motor.
        ChangeSpeed(AISettings_Speed.Walk);
        fsm_Controller.enemyController.motor.
        SendCharacterToDestination(fsm_Controller.
        nextPatrolPoint.gameObject.transform);
    }
    override public void OnStateUpdate(Animator animator,
    AnimatorStateInfo stateInfo, int layerIndex)
    {
        animator.SetFloat("Distance", fsm_Controller.Distance);
        if (fsm_Controller.enemyController.health.CurrentHealth <
   ↪ fsm_Controller.enemyController.health.MaxHealth)
            animator.SetBool("IsSomeoneHitMe", true);
        else animator.SetBool("IsSomeoneHitMe", false);
        if (!fsm_Controller.enemyController.motor.
        navMeshAgent.pathPending
            fsm_Controller.enemyController.motor.
            navMeshAgent.remainingDistance
           <= fsm_Controller.enemyController.motor.navMeshAgent.
   ↪ stoppingDistance)
            animator.SetTrigger("Continue");
    }
}
```

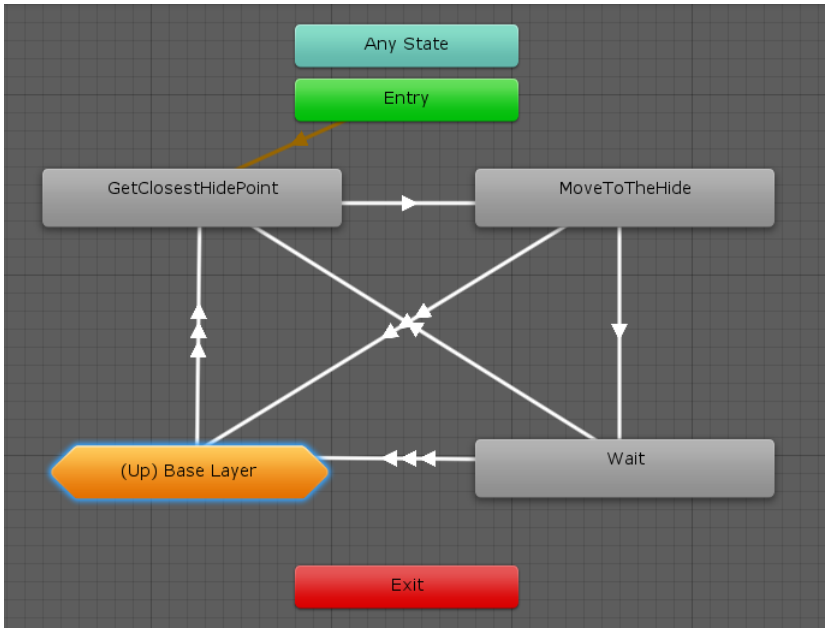
Wait

Stav Wait nám zastaví charakter na počet sekúnd ktorý si nastavíme v premennej TimeSet. Používame ho primárne kvôli realistickejšiemu pocitu zo správania NPC. V tomto prípade sa vždy zastaví, keď dojde na hliadkovacie miesto a až po chvíli pokračuje v rozhodovaní. Ak by sme takuto pauzu nepoužili, charakter by sa ani nestihol zastaviť a uz by pokračoval na ďalšie miesto v hliadkovaní. Z implementačného hľadiska nie je tento stav ničím zložitý. Jednoducho pri každom snímku (frame) odčítame Time.deltaTime od time a keď hodnota premennej time bude menšia alebo rovná nule, skontrolujeme vzdialenosť od pozície hráča a ak nie je v dosahu NPC, presunieme sa naspäť do stavu FindPatrolPoint. Samozrejme, neustále kontrolujeme životnú hodnotu NPC. Tento stav používame aj v ďalšom podstrome v stave Hide.

```
public class FSM_Wait : StateMachineBehaviour
{
    public float TimeSet;
```

```
    private float time;
    private FSM_TransitionController fsm_Controller;
    override public void OnStateEnter(Animator animator,
    AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (fsm_Controller == null)
            fsm_Controller = animator.gameObject.GetComponent<
    ↪ FSM_TransitionController>();
        time = TimeSet;
    }
    override public void OnStateUpdate(Animator animator,
    AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (fsm_Controller.enemyController.health.CurrentHealth <
    ↪ fsm_Controller.enemyController.health.MaxHealth)
            animator.SetBool("IsSomeoneHitMe", true);
        else animator.SetBool("IsSomeoneHitMe", false);
        if (time <= 0)
        {
            time = TimeSet;
            animator.SetFloat("Distance", fsm_Controller.Distance);
        }
        else
        {
            time -= Time.deltaTime;
        }
    }
}
```

7.1.2 Hide



Obr. 7.4 Hide podautomat

Do tejto časti automatu sa dostávame v prípade, keď na NPC začne hráč útočiť, no zároveň nie je vo vzdialenosti, z ktorej by ho NPC videl.

GetClosestHidePoint

GetClosestHidePoint je veľmi podobný stavu FindPatrolPoint, akurát, že na miesto

funkcie GetPatrolPoint() voláme funkciu GetClosestHidePoint(). Funkcia vráti NPC najbližšiu skrýšu, kde by sme sa mohli schovať pred streľbou. Tento stav sa opakuje, pokiaľ sa náš charakter nepresunie cez všetky dostupné skrýše. Toto správanie sme implementovali primárne z dôvodu, aby presunom do jednotlivých skrýš dostal NPC príležitosť zazrieť hráča a oplatiť mu úder. Ak náš charakter prejde cez všetky dostupné skrýše, vráti sa naspäť k hliadkovaniu.

```
public class FSM_GetClosestHidePoint : StateMachineBehaviour
{
    private FSM_TransitionController fsm_Controller;
    override public void OnStateEnter(Animator animator,
    AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (fsm_Controller == null)
            fsm_Controller = animator.gameObject.GetComponent<
    ↪ FSM_TransitionController>();
        fsm_Controller.enemyController.motor.
        ChangeStoppingDistance(AISettings_StoppingDistance.Move);
        Point point = fsm_Controller.enemyController.motor.
    ↪ GetClosestHidePoint();
        if (point != null)
        {
            fsm_Controller.nextHidePoint = point;
            fsm_Controller.nextHidePoint.Controlling = true;
            animator.SetTrigger("Continue");
        }
        else
            fsm_Controller.enemyController.health.MaxHealth =
    ↪ fsm_Controller.enemyController.health.CurrentHealth;
    }
}
```

MoveToTheHide

V stave MoveToTheHide presúvame charakter do najbližšej skrýše. Na začiatku si nastavíme rýchlosť pohybu vo funkcii ChangeSpeed na Run, čo znamená, že charakter sa do skrýše rozbehne. NPC zrýchľujeme, aby vyzeral uveriteľnejšie. Keďže po ňom niekto strieľa, tak sa pravdepodobne poponáhľa do skrýše. Následne, počas celého stavu neustále kontrolujeme vzdialenosť od hráča. Akonáhle sa NPC dostane do skrýše, aktivujeme trigger continue.

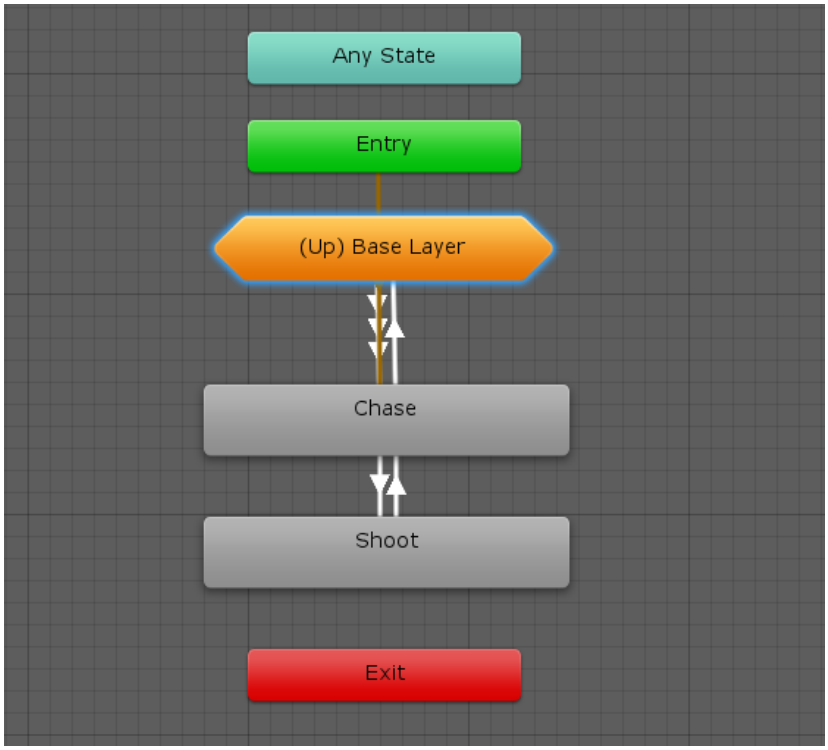
```
public class FSM_MoveToTheHide : StateMachineBehaviour
{
    private FSM_TransitionController fsm_Controller;
    override public void OnStateEnter(Animator animator,
    AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (fsm_Controller == null)
            fsm_Controller = animator.gameObject.GetComponent<
    ↪ FSM_TransitionController>();
        fsm_Controller.enemyController.motor.
        ChangeSpeed(AISettings_Speed.Run);
        fsm_Controller.enemyController.motor.
        SendCharacterToDestination(fsm_Controller.nextHidePoint.
    ↪ gameObject.transform);
    }
    override public void OnStateUpdate(Animator animator,
    AnimatorStateInfo stateInfo, int layerIndex)
```

```
    {
        animator.SetFloat("Distance", fsm_Controller.Distance);
        if (!fsm_Controller.enemyController.motor.
        navMeshAgent.pathPending
            fsm_Controller.enemyController.motor.
            navMeshAgent.remainingDistance
           <= fsm_Controller.enemyController.motor.
           navMeshAgent.stoppingDistance)
            animator.SetTrigger("Continue");
    }
}
```

Wait

Používame rovnaký stav ako v Patrol state.

7.1.3 Attack



Obr. 7.5 Attack podautomat

Do tejto časti automatu sa NPC dostáva v prípade, že sa hráč dostal do vzdialenosti na ktorú dokáže dovidieť.

Chase

V Chase stave sa NPC snaží priblížiť k hráčovi na vzdialenosť, z ktorej dokáže strieľať. Na začiatku tohoto stavu zvýšime rýchlosť na Run, aby sa NPC začal rýchlejšie pohybovať. Ďalej nastavíme StoppingDistance na Shoot, čo nám poskytne potrebný odstup od hráča na vzdialenosť, z ktorej dokáže NPC strieľať. Pomocou funkcie

SendCharacterToDestination() presunieme charakter na pozíciu hráča. Následne kontrolujeme vzdialenosť od hráča, ktorá ak dosiahne určitej hodnoty NPC prejde do stavu Shoot.

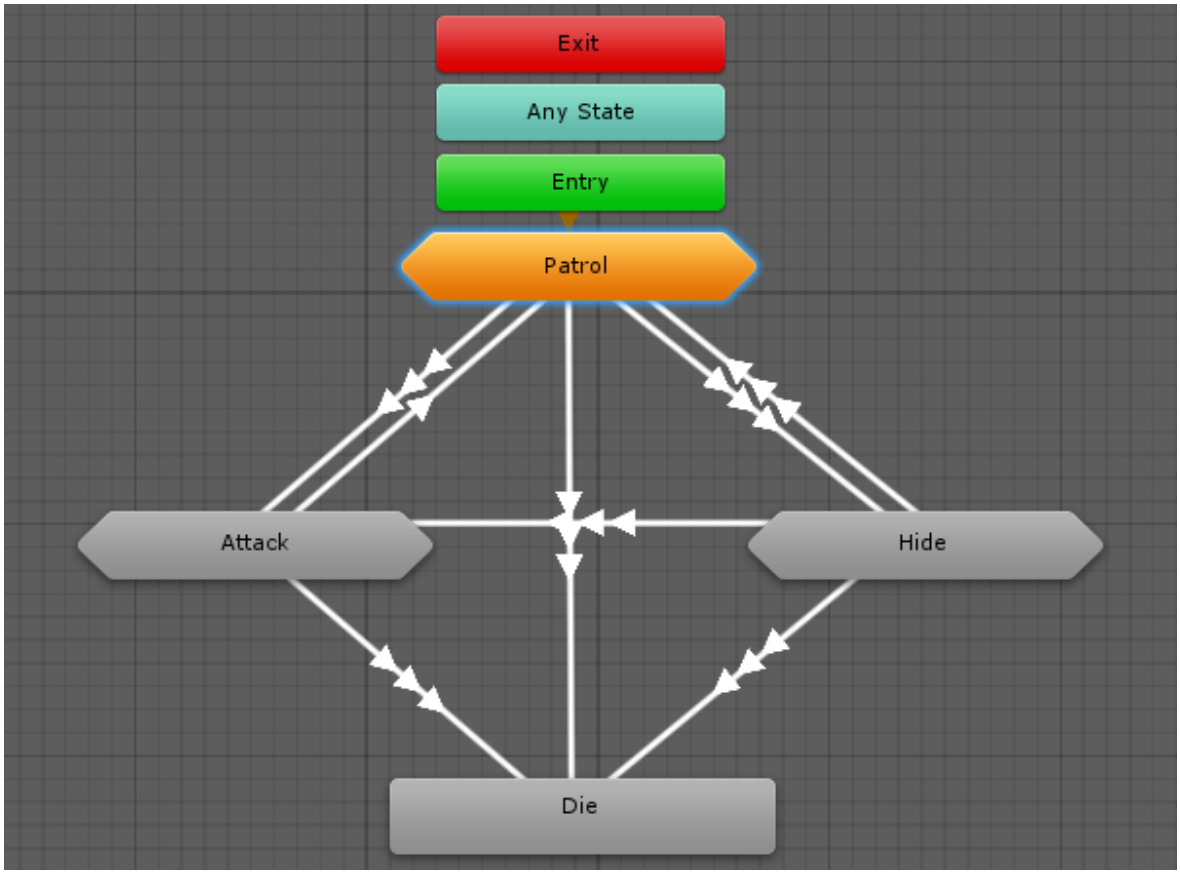
```
public class FSM_Chase : StateMachineBehaviour
{
    private FSM_TransitionController fsm_Controller;
    override public void OnStateEnter(Animator animator,
    AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (fsm_Controller == null)
            fsm_Controller = animator.gameObject.GetComponent<
   ↪ FSM_TransitionController>();
        fsm_Controller.enemyController.motor.
        ChangeSpeed(AISettings_Speed.Run);
        fsm_Controller.enemyController.motor.
        ChangeStoppingDistance(AISettings_StoppingDistance.Shoot);
        fsm_Controller.enemyController.motor.
        SendCharacterToDestination(fsm_Controller.Player.transform);
    }
    override public void OnStateUpdate(Animator animator,
    AnimatorStateInfo stateInfo, int layerIndex)
    {
        animator.SetFloat("Distance", fsm_Controller.Distance);
    }
}
```

Shoot

NPC v shoot stave začne strieľať po hráčovi. Máme tu premennú TimeSet, ktorou si možeme nastaviť pauzu medzi jednotlivými výstrelmi. Pri každom výstrele, spúšťame efekt streľby (MuzzleFlash) a voláme funkciu Shoot(), ktorú sme si spomenuli v úvode.

```
public class FSM_Shoot : StateMachineBehaviour
{
    public float TimeSet;
    private float time;
    private FSM_TransitionController fsm_Controller;
    override public void OnStateEnter(Animator animator,
    AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (fsm_Controller == null)
            fsm_Controller = animator.gameObject.GetComponent<
   ↪ FSM_TransitionController>();
        time = TimeSet;
    }
    override public void OnStateUpdate(Animator animator,
    AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (time <= 0)
        {
            time = TimeSet;
            fsm_Controller.enemyController.
            gun.muzzleFlash.Play();
            fsm_Controller.enemyController.
            gun.Shoot();
            animator.SetFloat("Distance",
             fsm_Controller.Distance);
        }
        else time -= Time.deltaTime;
    }
}
```

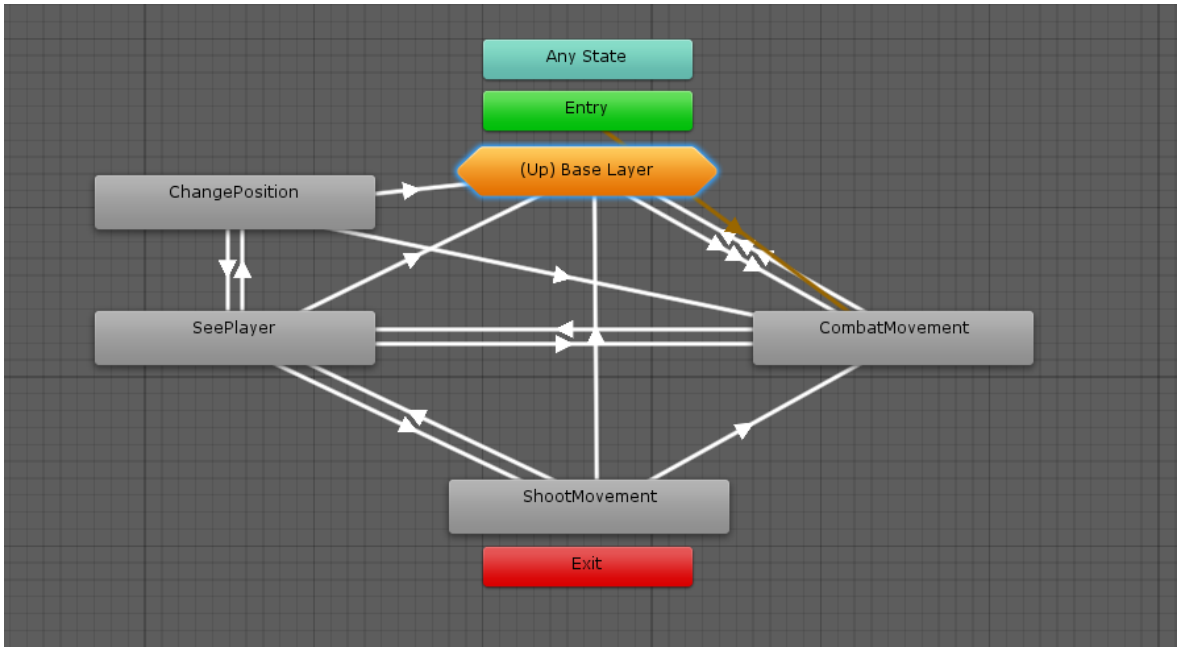
Toto bola prvá iterácia návrhu konečného automatu. Na prvý pohľad sa zdá, že správanie, ktoré sme navrhli je dostačujúce a charakter s rozhodnutiami predstavujúcimi jednotlivé stavy je dostatočne inteligentný na to, aby vyzeral v hernej scéne dôveryhodne. No po krátkom testovaní sme si uvedomili, že charakter je pomerne nemotorný čo sa týka prenasledovania hráča a strieľania po ňom. Nemotornosť sa prejavila v prípade, keď sa hráč schoval za objekt a NPC si stále myslel, že medzi ním a hráčom žiaden objekt nieje. Tým pádom namiesto toho, aby pri streľbe triafal hráča, triafal objekt ktorý mu v tom prekážal. Taktiež, keďže sme si vybrali ako príklad rozhodovanie zo strielačky, ku ktorému NPC podnecujeme akciami ako streľba alebo pohyb okolo neho, nám chýba stav pri ktorom NPC porazíme a on umrie. Na základe týchto faktov sme sa pustili do prerábky návrhu konečného automatu s tým, že sme museli pridať stav Die do hlavnej časti automatu (Obr. 7.6) a niekoľko stavov do podautomatu Attack (Obr. 7.7).



Obr. 7.6 Modifikovaný konečný automat

Die

Na stav Die nemáme pripnutý žiaden script. Do tohoto stavu sa dokážeme dostať z ktoréhokoľvek iného stavu v automate aktivovaním triggeru Death. Je to konečný



Obr. 7.7 Podautomat Attack

stav, do ktorého sa NPC dostane v prípade, že už nemá žiadnu životnú energiu. Ak sa tak stane, prehrá sa animácia umierania a NPC zostane ležať na zemi.

Podľa obrázku si môžeme všimnúť, že medzi stavy CombatMovement a ShootMovement sme pridali stav SeePlayer a ChangePosition.
Na kontrolu jednotlivých prechodov medzi stavmi sme pridali parameter SeePlayer typu bool.

SeePlayer

Do tohoto stavu sa dostaneme v prípade, ak vzdialenosť od hráča je na úrovni vzdialenosti potrebnej na strieľanie. V stave následne skontrolujeme, či daný hráč stojí pred nami vyslaním lúča, ktorý ak narazí na objekt s tagom "Player"podmienka SeePlayer nám vráti true a presunieme sa do stavu ShootMovement.

```
public class FSM_SeePlayer : StateMachineBehaviour
{
    private FSM_TransitionController fsm_Controller;
    override public void OnStateEnter(Animator animator,
    AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (fsm_Controller == null)
            fsm_Controller = animator.gameObject.
            GetComponent<FSM_TransitionController>();
        RaycastHit hit;
        if (Physics.Raycast(fsm_Controller.enemyController.
        gun.ShootOrigin.transform.position, fsm_Controller.
↪ enemyController.gun.
        ShootOrigin.transform.forward,
            out hit, fsm_Controller.enemyController.gun.Range))
        {
            Debug.Log(hit.transform);
```

```
            if (hit.transform.gameObject.tag == "Player")
            {
                animator.SetBool("SeePlayer", true);
            }
            else animator.SetBool("SeePlayer", false);
        }
    }
}
```

ChangePositon

Do tohoto stavu sa dostaneme zo stavu SeePlayer v prípade, ak vyslaný lúč na kontrolu viditeľnosti hráča narazí na iný objekt, ako na objekt samotného hráča. Následne vypočítame novú pozíciu pre NPC. Funkcia vráti pozíciu kolmú na hráčovu pozíciu, takže NPC sa vždy dokáže dostať na pozíciu, z ktorej uvidí hráča, aj keď je hráč schovaný za nejakým objektom.

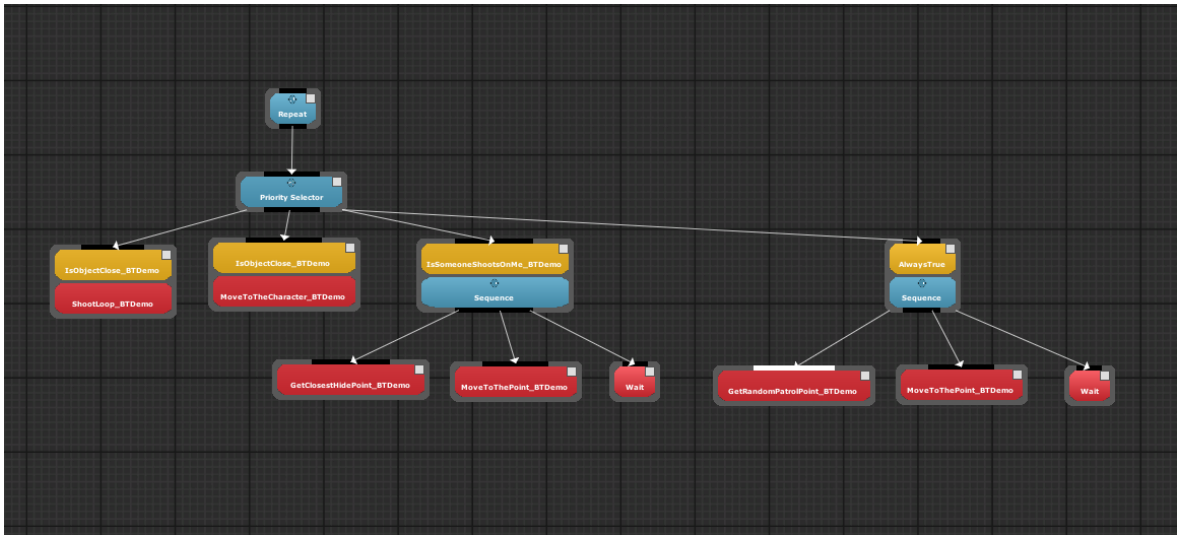
```
public class FSM_ChangePosition : StateMachineBehaviour
{
    private FSM_TransitionController fsm_Controller;
    override public void OnStateEnter(Animator animator,
    AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (fsm_Controller == null)
            fsm_Controller = animator.gameObject.
            GetComponent<FSM_TransitionController>();
        Vector3 perpendicular = new Vector3(-fsm_Controller.
        Player.transform.position.z, 0, fsm_Controller.Player.
    ↪ transform.position.x);
        float koeficient = fsm_Controller.enemyController.motor.
    ↪ navMeshAgent.
        stoppingDistance / perpendicular.magnitude;
        Vector3 vecFromPlayerToGoal = new Vector3(koeficient *
    ↪ perpendicular.x, perpendicular.y, koeficient * perpendicular.z);
        fsm_Controller.MoveObject.position = new Vector3(
    ↪ fsm_Controller.Player.transform.position.x - vecFromPlayerToGoal
    ↪ .x, 0, fsm_Controller.Player.transform.position.z -
    ↪ vecFromPlayerToGoal.z);
        fsm_Controller.enemyController.motor.
        ChangeStoppingDistance(AISettings_StoppingDistance.Move);
        fsm_Controller.enemyController.motor.
        navMeshAgent.SetDestination(fsm_Controller.MoveObject.position
    ↪ );
    }
    public override void OnStateExit(Animator animator,
    AnimatorStateInfo stateInfo, int layerIndex)
    {
        fsm_Controller.enemyController.motor.
        ChangeStoppingDistance(AISettings_StoppingDistance.Shoot);
        base.OnStateExit(animator, stateInfo, layerIndex);
    }
}
```

Ako môžeme vidieť na obrázku (Obr. 7.7) touto zdanlivo jednoduchou zmenou sme museli pozmeniť celý automat. Stavy, ktoré sme pridali kódom nezasahujú do už funkčných častí automatu. Kameňom úrazu v tomto prípade sú prechody medzi jednotlivými stavmi. Každý jeden stav je zavislý na ostatných stavoch, s ktorými je prepojený prechodmi. Z tohoto dôvodu môže byť pri komplexnejších automatoch aj drobná zmena pomerne dosť nebezpečná a náročná, keďže treba premýšlať nad každým stavom samo-

statne, či nebude treba aktualizovať prechody s ostatnými stavmi. Aj preto sa automaty používajú pri jednoduchších implementáciach, ako napríklad už spomenuté animácie alebo pri implementáciach, pri ktorých sa nepočíta so škálovaním.

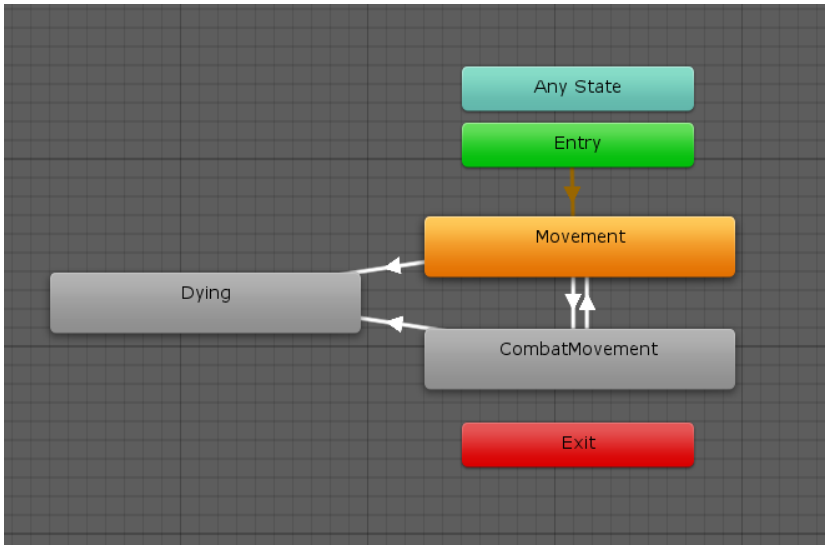
7.2 Rozhodovacie stromy

Na implementáciu rozhodovacích stromov sme použili Behavior Bricks plugin, používajúci sa na tvorbu správania pomocou rozhodovacích stromov. [4] Rovnako ako pri konečných automatoch sme použili iteratívny prístup na návrh rozhodovania s cieľom lepšie demonštrovať škálovanie rozhodovacích stromov oproti konečným automatom. Na nasledujúcom obrázku môžete vidieť základný návrh stromu s rozhodovacími možnosťami prvej iterácie spracovanej pri konečných automatoch (Obr. 7.8).



Obr. 7.8 Základný rozhodovací strom

Okrem samotného stromu sme vytvorili aj konečný automat s použitím Animátoru slúžiaci na ovládanie jednotlivých animácií (Obr. 7.9). Na obrázku môžete vidieť tri základné stavy Movement, CombatMovement a Dying. Movement a CombatMovement predstavujú BlendTree. BlendTrees sa používajú na miešanie viacerých animácií, ich začlenením do viacerých stupňov. Hodnota akou jednotlivé animácie prispievajú k finálnemu efektu je kontrolovaná nami zvoleným parametrom. Samozrejme, aby mali tieto zmiešané animácie nejaký zmysel, musia byť podobnej povahy. Ako príklad môžeme uviesť použitie BlendTrees na pohyb NPC po scéne, kde môžeme zmiešať animácie chôdze a behu podľa rýchlosti postavy. [5] Stav Dying predstavuje stav, v ktorom sa spustí animácia umierania.
V tomto automate používame na prechody medzi stavmi štyri parametre.



Obr. 7.9 Animačný automat

Speed

Parameter typu float, ktorý používame priamo v stavoch Movement a CombatMovement vytvorených z BlendTree. Na parametri je pripnutých niekoľko animácií. Tieto animácie sa striedajú na základe rýchlosti ako sa náš charakter pohybuje po scéne. Túto rýchlosť získavame už spomenutou funkciou getSpeed(EnemyMotor motor).

SeePlayer

Parameter SeePlayer je typu bool a slúži na prechod medzi stavmi Movement a CombatMovement. Vždy keď NPC uvidí hráča presunie sa z Movement do CombatMovement a naopak.

Die

Je parameter typu bool, ktorý ak aktivujeme, dokážeme sa presunúť z akéhokoľvek stavu do stavu Dying.

V strome môžeme vidieť základné uzly typu priority selektor a sequencie. V teoretickej časti sme si obidva druhy uzlov popísali. Priority selektor, ktorý môžeme vidieť na obrázku sa v našom prípade odlišuje od selektoru možnosťou pridať podmienku pred samotnú úlohu. Ak je daná podmienka splnená, selektor spustí úlohu nasledujúcu po podmienke. Ak podmienka naopak splnená nie je, pokračuje nasledujúcou podmienkou vpravo. Pre zopakovanie, selektor spúšťa jednotlivé úlohy podľa priority z ľavej strany do pravej, s tým, že hľadá aspoň jednu, ktorá uspeje. Samozrejme, za podmienku môžeme okrem úloh vkladať aj ďalšie druhy uzlov, ako sú v našom prípade sequenčné uzly. Sequenčné uzly spúšťaju jednotlivé úlohy zľava doprava, s tým, že každá jedna

úloha musí uspieť, inak celá sequencia vracia neúspech. Uzol typu repeat spúšťa celý strom stále dookola.

7.2.1 Rozhodovací strom

Všetky úlohy používané v tomto strome dedia od úlohy EnemyAIBase_BTDemo, ktorá slúži ako rozhranie na prístup k funkciám a parametrom triedy EnemyController_BTDemo a hernému objektu predstavujúcemu hráča.

```
public class EnemyAIBase_BTDemo : GOAction
{
    [InParam("EnemyController")]
    public EnemyController_BTDemo enemyController;
    [InParam("Player")]
    public GameObject Player;
}

public class EnemyController_BTDemo : EnemyController
{
    public EMotor_BTDemo motor;
    private void Update()
    {
        m_anim.SetFloat("Speed", getSpeed(motor));
    }
}
```

V prvej úrovni stromu, pod Priority Selektorom začíname podmienkou HealthCheck_BTDemo.

HealthCheck_BTDemo

Podmienka kontroluje, či životná energia NPC nie je menšia alebo rovná nule. Ak je táto podmienka splnená, prechádzame do úlohy Die_BTDemo.

```
public class HealthCheck_BTDemo : GOCondition
{
    [InParam("EnemyController")]
    public EnemyController_BTDemo enemyController;
    public override bool Check()
    {
        if (enemyController.health.CurrentHealth <= 0)
            return true;
        else return false;
    }
}
```

Ak má NPC dostatok životnej energie, presunieme sa na ďalšiu podmienku a tou je IsObjectClose_BTDemo.

IsObjectClose_BTDemo

V podmienke kontrolujeme, či je vzdialenosť od hráča vhodná na strieľanie. Táto podmienka vypočíta vzdialenosť cesty k hráčovi za pomoci NavMeshAgent a NavMeshPath. Najskôr skúsime získať cestu k hráčovi funkciou CalculatePath, ktorá nám vráti

True, ak sa nám to podarí. Následne si vytvoríme pole allCorners typu Vector3, do ktorého budeme ukladať jednotlivé rohy/záhyby vyskytujúce sa na získanej ceste. Ďalej prejdeme cyklom for cez všetky rohy a získame jednotlivé vzdialenosti medzi nimi funkciou Vector3.Distance a všetky vzdialenosti sčítame do premenej pathLength. Na konci kontrolujeme či vzdialenosť od hráča je menšia alebo rovná predvolenej vzdialenosti uloženej v premenej distance. Ak tomu tak je, zavoláme funkciu Animation_SeePlayer(), ktorá nám zmení animáciu na prenasledovanie a podmienka vráti True. V opačnom prípade, zavoláme funkciu Animation_DoNotSeePlayer() a podmienka vráti false.

```
[Condition("MyConditions/IsObjectClose_BTDemo")]
[Help("Checks whether a target is close depending on a given distance"
   ↪ )]
public class IsObjectClose_BTDemo : GOCondition
{
    [InParam("Object")]
    public GameObject Object;
    [InParam("Distance")]
    [Help("The maximun distance to consider that the target is close")
   ↪ ]
    public float distance;
    [InParam("EnemyController")]
    public EnemyController_BTDemo enemyController;
    private NavMeshAgent m_agent;
    public override bool Check()
    {
        NavMeshPath path = new NavMeshPath();
        if (m_agent == null)
            m_agent = gameObject.GetComponent<NavMeshAgent>();
        bool calculated = m_agent.CalculatePath(Object.transform.
   ↪ position, path);
        if (!calculated)
        {
            Debug.LogError("Path for " + gameObject.name + " was not
   ↪ calculated.");
            return false;
        }
        Vector3[] allCorners = new Vector3[path.corners.Length + 2];
        allCorners[0] = gameObject.transform.position;
        allCorners[allCorners.Length - 1] = Object.transform.position;
        for (int i = 0; i < path.corners.Length; i++)
        {
            allCorners[i + 1] = path.corners[i];
        }
        float pathLength = 0f;
        for (int i = 0; i < allCorners.Length - 1; i++)
        {
            pathLength += Vector3.Distance(allCorners[i], allCorners[i
   ↪  + 1]);
        }
        if (pathLength <= distance)
        {
            enemyController.Animation_SeePlayer();
            return true;
        }
        else
        {
            enemyController.Animation_DoNotSeePlayer();
            return false;
        }
    }
}
```

Ak sa teda NPC nachádza vo vzdialenosti z ktorej dokáže trafiť hráča, presunieme

sa do úlohy ShootLoop_BTDemo.

ShootLoop_BTDemo

Táto úloha dedí od úlohy ShootForOnce_BTDemo, v ktorej nastavujeme parameter StoppingDistance na Shoot, aby sa NPC zastavil od hráča vo vzdialenosti, z ktorej dokáže strieľať. Následne voláme už spomenutú funkciu Shoot() z Gun.cs scriptu. V samotnej úlohe ShootLoop_BTDemo nastavujeme pauzu medzi jednotlivými výstrelmi pomocou premenej delay.

```
[Action("ShootForOnce_BTDemo")]
public class ShootForOnce_BTDemo : EnemyAIBase_BTDemo
{
    [InParam("ShootOrigin")]
    public GameObject ShootOrigin;
    public override void OnStart()
    {
        base.OnStart();
        if (enemyController.motor.Target != Player.transform)
            enemyController.motor.Target = Player.transform;
        enemyController.motor.ChangeStoppingDistance(
   ↪ AISettings_StoppingDistance.Shoot);
    }
    public override TaskStatus OnUpdate()
    {
        if (ShootOrigin == null)
            return TaskStatus.FAILED;
        enemyController.gun.muzzleFlash.Play();
        enemyController.gun.Shoot();
        return TaskStatus.COMPLETED;
    }
}

[Action("ShootLoop_BTDemo")]
public class ShootLoop_BTDemo : ShootForOnce_BTDemo
{
    [InParam("delay", DefaultValue = 30)]
    public int delay;
    private int elapsed = 0;
    public override TaskStatus OnUpdate()
    {
        if(delay > 0)
        {
            ++elapsed;
            elapsed %= delay;
            if (elapsed != 0)
                return TaskStatus.RUNNING;
        }
        base.OnUpdate();
        return TaskStatus.RUNNING;
    }
}
```

V prípade, že podmienka IsObjectClose_BTDemo, s ktorou sme kontrolovali, či je NPC vo vzdialenosti, z ktorej dokáže strieľať zlyhá, presunieme sa na kontrolu, či je hráč vo vzdialenosti, z ktorej by ho mohol NPC začať prenasledovať. Na túto kontrolu použijeme rovnakú podmienku, akurát zmeníme jej vstupný parameter distance, ktorý nastavíme na vyššiu hodnotu. Ak sa teda hráč nachádza od NPC vo vzdialenosti menšej ako sme si nastavili na vstupe, presunieme sa na úlohu MoveToTheCharac-

ter_BTDemo.

MoveToTheCharacter_BTDemo

V tejto úlohe sa NPC začne presúvať na pozíciu hráča. Pri spúšťaní danej úlohy nastavíme cieľovú pozíciu charakteru na pozíciu hráča a rýchlosť na Run, na základe ktorej sa prepne animácia v už spomenutom BlendTree. Vo funkcii OnUpdate kontrolujeme, či už NPC dorazil na pozíciu. Ak dorazil, úloha vráti stav Completed, ak sa ešte presúva, úloha vracia stav Running.

```
[Action("MoveToTheCharacter_BTDemo")]
[Help("Chasing character.")]
public class MoveToTheCharacter_BTDemo : EnemyAIBase_BTDemo
{
    public override void OnStart()
    {
        enemyController.motor.SendCharacterToDestination(Player.
    ↪ transform);
        enemyController.motor.ChangeSpeed(AISettings_Speed.Run);
    }
    public override TaskStatus OnUpdate()
    {
        if (!enemyController.motor.navMeshAgent.pathPending
    ↪ enemyController.motor.navMeshAgent.remainingDistance
            <= enemyController.motor.navMeshAgent.stoppingDistance)
            return TaskStatus.COMPLETED;
        return TaskStatus.RUNNING;
    }
}
```

V prípade, že sa hráč nenachádza v dosahu NPC, priority selector vyskúša spustiť podmienku **IsSomeoneShootsOnMe_BTDemo**, v ktorej kontrolujeme momentálnu hodnotu životnej energie. Ak je životná energia nižšia ako maximálna životná energia, spustíme sequenciu po ktorej splnení by sa mal NPC dostať bezpečne do skrýše.

```
[Condition("MyConditions/IsSomeoneShootsOnMe_BTDemo")]
public class IsSomeoneShootsOnMe_BTDemo : GOCondition
{
    [InParam("TargetScript")]
    public HealthControl targetScript;
    public override bool Check()
    {
        if (targetScript.CurrentHealth < targetScript.MaxHealth)
            return true;
        else return false;
    }
}
```

GetClosestHidePoint_BTDemo

Prvou úlohou v sequencii je GetClosestHidePoint_BTDemo. Táto úloha obsahuje parameter HidePoint typu Transform označený ako výstupný. Výstupné parametre sa v sequencii predávajú do ďalších úloh ako vstupné parametre. Na začiatku úlohy meníme StoppingDistance na Move, čo znamená, že NPC zastaví priamo na mieste kam

smeruje bez odstupu. Následne, meníme rýchlosť pohybu na Run. Vo funkcii OnUpdate() voláme funkciu GetClosestHidePoint(), ktorá nám vráti najbližšie miesto na skrytie. Ďalej vložíme do premennej HidePoint referenciu na Transform komponent vráteného miesta a označíme dané miesto, ako skontrolované premenou Controlling. Úlohu označíme za splnenú. V prípade, že nám funkcia GetClosesthidePoint() vráti null (to znamená, že už sme vyčerpali všetky skrýše), aktualizujeme hodnotu maximálnej životnej energie v scripte HealthControl.cs hodnotou momentálnej životnej energie a úloha vráti status failed.

```
[Action("GetClosestHidePoint_BTDemo")]
public class GetClosestHidePoint_BTDemo : EnemyAIBase_BTDemo
{
    [OutParam("HidePoint")]
    public Transform HidePoint;
    public override void OnStart()
    {
        enemyController.motor.ChangeStoppingDistance(
↪ AISettings_StoppingDistance.Move);
        enemyController.motor.ChangeSpeed(AISettings_Speed.Run);
    }
    public override TaskStatus OnUpdate()
    {
        Point point = enemyController.motor.GetClosestHidePoint();
        if (point != null)
        {
            HidePoint = point.transform;
            point.Controlling = true;
            return TaskStatus.COMPLETED;
        }
        else
            enemyController.health.MaxHealth = enemyController.health.
↪ CurrentHealth;
        return TaskStatus.FAILED;
    }
}
```

Ak je táto úloha splnená, presunieme sa v sequencii do úlohy MoveToThePoint_BTDemo.

MoveToThePoint_BTDemo

Táto úloha presúva NPC z jedného miesta na druhé. Na začiatku máme jeden vstupný parameter Point typu Transform ktorý získava svoju hodnotu z predchádzajúceho stavu. Logika úlohy je veľmi podobná ako v MoveToTheCharacter_BTDemo, na začiatku posielame NPC na dané miesto a následne vo funkcii OnUpdate() kontrolujeme či sa tam dostavil alebo nie.

```
[Action("MoveToThePoint_BTDemo")]
[Help("Moving to the point")]
public class MoveToThePoint_BTDemo : EnemyAIBase_BTDemo
{
    [InParam("NextDestination")]
    Transform Point;
    public override void OnStart()
    {
        enemyController.motor.SendCharacterToDestination(Point);
    }
    public override TaskStatus OnUpdate()
```

```
    {
        if (!enemyController.motor.navMeshAgent.pathPending
    ↪ enemyController.motor.navMeshAgent.remainingDistance
            <= enemyController.motor.navMeshAgent.stoppingDistance)
            return TaskStatus.COMPLETED;
        return TaskStatus.RUNNING;
    }
}
```

Wait

Poslednou úlohou v sequencii je Wait. Cieľom úlohy je zastaviť charakter na cieľovom mieste, aby jeho správanie vyzeralo uveriteľnejšie. Wait obsahuje jeden vstupný parameter TimeSet, ktorý slúži na nastavenie hodnoty času v sekundách, ktoré bude NPC čakať a jeden privátny parameter time. Vo funkcii OnUpdate() kontrolujeme hodnotu parametru time, ktorá ak je väčšia ako nula, úloha nám vráti status Running. Naopak, ak hodnota premenej klesne pod nulu, úloha vráti status Completed.

```
[Action("Wait")]
[Help("Wait till enemy is far enough")]
public class Wait : GOAction
{
    [InParam("TimeToWait")]
    public float TimeSet;
    private float time;
    public override void OnStart()
    {
        base.OnStart();
        time = TimeSet;
    }
    public override TaskStatus OnUpdate()
    {
        if (time <= 0)
        {
            time = TimeSet;
            return TaskStatus.COMPLETED;
        }
        else
        {
            time -= Time.deltaTime;
            return TaskStatus.RUNNING;
        }
    }
}
```

V prípade, že ani jedna z predchádzajúcich podmienok alebo úloh nebola splnená, Priority selektor skontroluje poslednú podmienku AlwaysTrue, ktorá už podľa názvu vracia vždy úspech. Tento typ podmienky sme použili z dôvodu, aby mal NPC nejaký základný stav, v ktorom bude pretrvávať bez hráčovho ovplyvnenia. Nemať také správanie, charakter by nehybne stál na mieste a čakal, pokiaľ jedna z predchádzajúcich podmienok bude splnená. Ak sa priority selektor dostane k tejto podmienke, presunieme sa do sequencie, ktorej cieľom je podnietiť charakter k hliadkovaniu.

GetRandomPatrolPoint_BTDemo

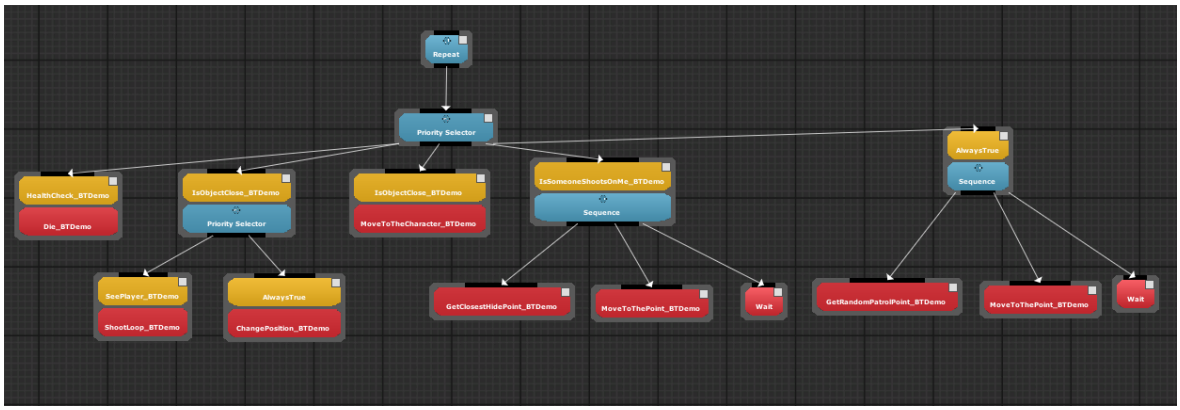
Cieľom prvej úlohy v sequencii je získať pozíciu miesta na hliadkovanie. Úloha je

veľmi podobná GetClosestHidePoint_BTDemo, akurát, že vo funkcii OnUpdate() voláme funkciu GetPatrolPoint(), ktorá nám vráti náhodné miesto na hliadkovanie.

```
[Action("GetRandomPatrolPoint_BTDemo")]
[Help("Looking for patrol point")]
public class GetRandomPatrolPoint_BTDemo : EnemyAIBase_BTDemo
{
    [OutParam("NextPatrolpoint")]
    public Transform PatrolPoint;
    public override void OnStart()
    {
        enemyController.motor.ChangeStoppingDistance(
    ↪ AISettings_StoppingDistance.Move);
        enemyController.motor.ChangeSpeed(AISettings_Speed.Walk);
    }
    public override TaskStatus OnUpdate()
    {
        Point nextPoint = enemyController.motor.GetPatrolPoint();
        if (nextPoint != null)
        {
            PatrolPoint = nextPoint.transform;
            nextPoint.Controlling = true;
            return TaskStatus.COMPLETED;
        }
        return TaskStatus.FAILED;
    }
}
```

Ak táto úloha vráti status completed, sequencia sa presunie na už spomenutú úlohu MoveToThePoint_BTDemo a následne na Wait.

Rovnako ako v návrhu pri konečných automatoch, po testovaní sme prišli na to, že NPC je pomerne nemotorný pri nasledovaní hráča a strieľaní, primárne v častiach scény s viacerými prekážkami. Taktiež musíme pridať možnosť poraziť NPC. Na základe týchto zistení sme strom pozmenili(Obr. 7.10).



Obr. 7.10 Modifikovaný strom Enemy

Keďže sa daná zmena týkala prenasledovania hráča a útoku na neho, stačilo pozmeniť len danú časť stromu, ktorá sa zaoberala týmto správaním. Ako môžete vidieť na obrázku, do časti, kde v predchádzajúcom strome NPC začal strielať po hráčovi, sme pridali priority selektor, zopár podmienok a jednu úlohu. Keďže priority selektor prioritizuje úlohy z ľavej strany, začnime aj my vľavo. Máme tu novú podmienku,

SeePlayer_BTDemo s jedným vstupným parametrom typu EnemyGun. Vždy keď sa strom dostane do tejto podmienky, vyšle lúč a skontroluje na aký objekt daný lúč narazil. Ak lúč narazil na hráča, podmienka vráti True a strom vykoná úlohu ShootLoop_BTDemo, ak nie, podmienka vráti false a selektor pokračuje ďalej do podmienky AlwaysTrue.

```
[Condition("SeePlayer_BTDemo")]
public class SeePlayer_BTDemo : GOCondition
{
    [InParam("EnemyGun")]
    public EnemyGun gun;
    public override bool Check()
    {
        RaycastHit hit;
        if (Physics.Raycast(gun.ShootOrigin.transform.position, gun.
   ↪ ShootOrigin.transform.forward, out hit, gun.Range))
        {
            Debug.Log(hit.transform);
            if (hit.transform.gameObject.tag == "Player")
                return true;
        }
        return false;
    }
}
```

ChangePosition_BTDemo

V tejto úlohe počítame novú pozíciu pre NPC pomocou pravouhlého trojuholníka ako v prípade konečných automatov. Funkcia vráti pozíciu kolmú na hráčovu pozíciu, takže NPC sa vždy dokáže dostať na pozíciu, z ktorej uvidí hráča, aj keď je hráč schovaný za nejakým objektom.

```
[Action("ChangePosition_BTDemo")]
public class ChangePosition_BTDemo : EnemyAIBase_BTDemo
{
    [InParam("NewDestination")]
    private Transform newDestination;
    public override void OnStart()
    {
        Vector3 perpendicular = new Vector3(-Player.transform.position
   ↪ .z, 0, Player.transform.position.x);
        float koeficient = enemyController.motor.navMeshAgent.
   ↪ stoppingDistance / perpendicular.magnitude;
        Vector3 vecFromPlayerToGoal = new Vector3(koeficient *
   ↪ perpendicular.x, perpendicular.y, koeficient * perpendicular.z);
        newDestination.position = new Vector3(Player.transform.
   ↪ position.x - vecFromPlayerToGoal.x, 0, Player.transform.position
   ↪ .z  - vecFromPlayerToGoal.z);
        enemyController.motor.ChangeStoppingDistance(
   ↪ AISettings_StoppingDistance.Move);
        enemyController.motor.ChangeSpeed(AISettings_Speed.Run);
        enemyController.motor.navMeshAgent.SetDestination(
   ↪ newDestination.position);
    }
}
```

Ako najprioritnejšiu úlohu sme pridali **Die_BTDemo**. V úlohe voláme funkciu Animation_Die() slúžiacu na spustenie animácie umierania. Dôvodom, prečo táto úloha

má najvyššiu prioritu (jej pozícia je úplne v ľavo) je, že najskôr pred vykonaním akejkoľvek inej akcie si musí NPC skontrolovať, či ho už náhodou hráč neporaził. Ak tomu tak je, s nulovou životnou energiou nie je schopný vykonávať ďalšie akcie.

```
[Action("Die_BTDemo")]
public class Die_BTDemo : EnemyAIBase_BTDemo
{
    public override void OnStart()
    {
        enemyController.Animation_Die();
    }
}
```

Zdroje grafických assetov

Pri tvorbe jednotlivých častí dema o rozhodovaní sme použili nasledujúce assety:

- skybox SpaceSkies Free https://assetstore.unity.com/packages/2d/textures-materials/sky/spaceskies-free-80503,
- prostredie Modular Sci-Fi Corridor https://assetstore.unity.com/packages/3d/environments/sci-fi/modular-sci-fi-corridor-142811,
- UFPS : Ultimate FPS https://assetstore.unity.com/packages/templates/systems/ufps-ultimate-fps-2943,
- modely charakterov aj s animáciami https://www.mixamo.com/,
- modely zbraní https://devassets.com/assets/modern-weapons/.

8 Finálne Demo

Cieľom finálneho dema je demonštrovať rozhodovanie viacerých charakterov v scéne bez vstupu uživateľa. Aby sme umožnili jednotlivým NPC nezávislé rozhodovanie, potrebovali sme vytvoriť prostredie a kontext, v ktorom by sa mohli rozhodovať. Potrebovali sme určiť ciele jednotlivých druhov charakterov a ich rozmedzie možných vykonávaných akcií. V tomto deme preto máme dva druhy charakterov s rozdielnymi cieľmi, Player a Enemy. Player charakter alebo v scéne pomenovaný aj "MainCharacter" je NPC, ktorý sa snaží prejsť bludisko a popri tom otvára jednotlivé dvere stojace mu v ceste. Otváranie dverí nie je jednoduchá záležitosť, pretože každé dvere sa otvárajú špeciálnym kľučom vytvoreným priamo pre dané dvere. To znamená, že keď Player narazí na nové dvere, ktoré potrebuje odomknúť, musí najskôr pri interakcii zistiť, aký kľúč potrebuje. Potom sa vydá hľadať kľúč po scéne v rôznych objektoch, s ktorými môže interagovať. Ak sa mu podarí odomknúť všetky dvere, dostane sa z bludiska. Potom tu máme druhý typ NPC nazvaný Enemy. Na scéne môžeme nájsť niekoľko NPC tohoto typu. Každý Enemy má nastavenú určitú časť bludiska na hliadkovanie. V prípade, že Enemy zazrie Playera potulujúceho sa po bludisku, začne ho prenasledovať. Prenasleduje ho len do bodu, kedy je ho schopný vidieť. Akonáhle mu Player zmizne zo zorného uhla, zastaví sa a zavolá si posily, na ktoré počká pokiaľ sa k nemu nedostanú. Následne sa spolu vydajú na miesto, kde videli Playera naposledy. Ak sa dostanú k Playerovi do vzdialenosti na strieľanie, tak ho zastrelia.

Skôr ako sa pustíme do podrobného systému ako fungujú stromy našich NPC, poďme sa pozrieť na kód, ktorý pretvára obyčajné objekty v scéne na objekty, s ktorými sa dá interagovať. V scéne máme dva druhy takýchto objektov: dvere a nábytok. Každý z týchto objektov má špeciálne vlastnosti, no oba dedia od rodičovskej triedy Interactable, ktorá sa stará o základné vlastnosti interakčných objektov. V tejto triede nájdeme funkcie OnTriggerEnter a OnTriggerExit, ktoré sa volajú vždy keď nejaký cudzí collider naruší collider objektu používajúci daný script.

```
private void OnTriggerEnter(Collider other)
{
    if (other.gameObject.tag == "Player")
    {
        player = other.GetComponent<PlayerController>();
    }
}

private void OnTriggerExit(Collider other)
{
    player = null;
}
```

Ďalej v triede môžeme nájsť funkciu MoveAllItems(Inventory otherInventory), ktorá sa volá pri kontakte hráča s daným objektom a presunie všetky predmety z inventáru

objektu do hráčovho inventáru.

```
 public void MoveAllItems(Inventory otherInventory)
    {
        if (Inventory.InventoryItems.Count > 0)
        {
            foreach (var item in Inventory.InventoryItems)
            {
                player.CheckIfKeyFound(item);
                otherInventory.AddItem(item);
            }
            Inventory.InventoryItems = new List<InventoryItem>();
        }
        else Debug.Log("Nothing in " + gameObject.name + " inventory")
   ↪ ;
    }
```

O samotný inventár sa nám stará trieda Inventory, v ktorej nájdeme List<InventoryItem> InventoryItems a metódy na obsluhu inventáru (SearchItem, RemoveItem, AddItem). Jednotlivé predmety (v kóde InventoryItems) sú vytvorené pomocou ScriptableObjects. ScriptableObjects sú dátové kontajnery, ktoré dokážu ukladať rôzne dáta nezávistle od inštancií tried. [6] V našom kontajneri ukladáme informácie typu meno, typ objektu a farbu daného predmetu. Týmto spôsobom sú v projekte vytvorené všetky kľúče.

```
public class Inventory : MonoBehaviour
{
    public List<InventoryItem> InventoryItems;
    public int InventorySize = 1;
    public void AddItem(InventoryItem item)
    {
        if(InventoryItems.Count < InventorySize)
        {
            InventoryItems.Add(item);
            Debug.Log("Item " + item.name + " added to " + gameObject.
   ↪ name + " inventory");
        }
        else Debug.LogError("Item " + item.name + " note added to
   ↪ inventory. Inventory full.");
    }
    public void RemoveItem(InventoryItem item)
    {
        InventoryItems.Remove(item);
        Debug.Log("Item " + item.name + " removed from " + gameObject.
   ↪ name + " inventory");
    }
    public InventoryItem SearchItem(string itemName)
    {
        foreach (var item in InventoryItems)
        {
            if (item.Name == itemName)
                return item;
        }
        Debug.LogError("Item " + itemName + " not found");
        return null;
    }
}

public enum ItemTypes { Key, Item }
[CreateAssetMenu(fileName = "Item", menuName = "Inventory/Item", order
   ↪  = 1)]
public class InventoryItem : ScriptableObject
{
    public Sprite Icon;
    public string Name;
```

```
    public ItemTypes itemType;
    public Color itemColor;
}
```

Na koordináciu hráča v scéne (aby vedel, ku ktorým dverom má smerovať) používame triedu QuestManager. Trieda obsahuje List<Quest> QuestsToComplete, ktorý ukladá informácie o jednotlivých úlohách prostredníctvom triedy Quest a funkciu GetActiveQuest(), ktorá nám vráti Quest práve plnený Playerom. V triede používame návrhový vzor Singleton.

```
public Quest GetActiveQuest()
 {
     foreach (var quest in QuestsToCopmlete)
     {
         if (!quest.isFinished)
             return quest;
     }
     return null;
 }
```

Ďalšou dôležitou triedou, od ktorej dedí trieda EnemyController_FinalDemo je trieda AgentControl. AgentControl dedí vlastnosti od základnej triedy EnemyController, ktorú sme si už spomenuli. Dôvodom vloženia tejto triedy do hierarchie dedenia pre základnú kontrolu Enemy NPC bola schopnosť pracovať v skupinách s inými NPC. Trieda nám poskytuje krátku knižnicu s funkciami, ktoré nám umožňujú vytvoriť skupinu, presunuť skupinu jednotiek, získavať pozíciu skupiny, a ďalšie. V tejto časti sa pozrieme len na tie najdôležitejšie z nich. Začnime samotným vytvorením skupiny jendotiek vo funkcii CreateSquad(). Táto funkcia je volaná v prípade, že jeden z NPC zazrie hráča a počas jeho prenasledovania sa mu stratí z dohľadu. Takže NPC vytvorí skupinu, pridá sa do nej a začne hľadať najbližšieho NPC v okolí, ktorý by sa k nemu mohol pridať. Po jeho nájdení, mu aktualizuje nasledujúce miesto na presun a zavolá ho.

```
public void CreateSquad()
  {
      AgentBase.Instance.AvailableAgentsInTheScene.Remove((
↪ EnemyController_FinalDemo)this);
      agentsInSquad.AddAgentIntoSquad((EnemyController_FinalDemo)
↪ this, SquadPermissions.Lead);
      Dictionary<float, EnemyController_FinalDemo> agentNDistance
          = GetAgentsWithDisctances(AgentBase.Instance.
↪ AvailableAgentsInTheScene);
      var distance = AgentBase.Instance.AvailableAgentsInTheScene
↪ [0].motor.
          CaclculatePathDistance(gameObject);
      EnemyController_FinalDemo npcToSquad = null;
      foreach (var agent in AgentBase.Instance.
↪ AvailableAgentsInTheScene)
      {
          if (agent.motor.CaclculatePathDistance(gameObject) <=
↪ distance)
              npcToSquad = agent;
      }
      agentsInSquad.AddAgentIntoSquad(npcToSquad, SquadPermissions.
↪ Member);
```

```
        npcToSquad.motor.ChangeSpeed(AISettings_Speed.Run);
        MoveSquad(gameObject.transform.position);
        AgentBase.Instance.Squads.Add(agentsInSquad);
    }
```

Ďalšou dôležitou funkciou je MoveSquad(Vector3 targetPosition), v ktorej presunieme skupinu pozostávajúcu z NPC na určenú pozíciu. Túto pozíciu predávame ako parameter funkcie. Vo funkcii používame GetPositionsForSquad(Vector3 startPosition, float distance, int positionCount), ktorá nám vráti List pozícií pre jednotlivých členov skupiny tak, aby do seba nenarazili.

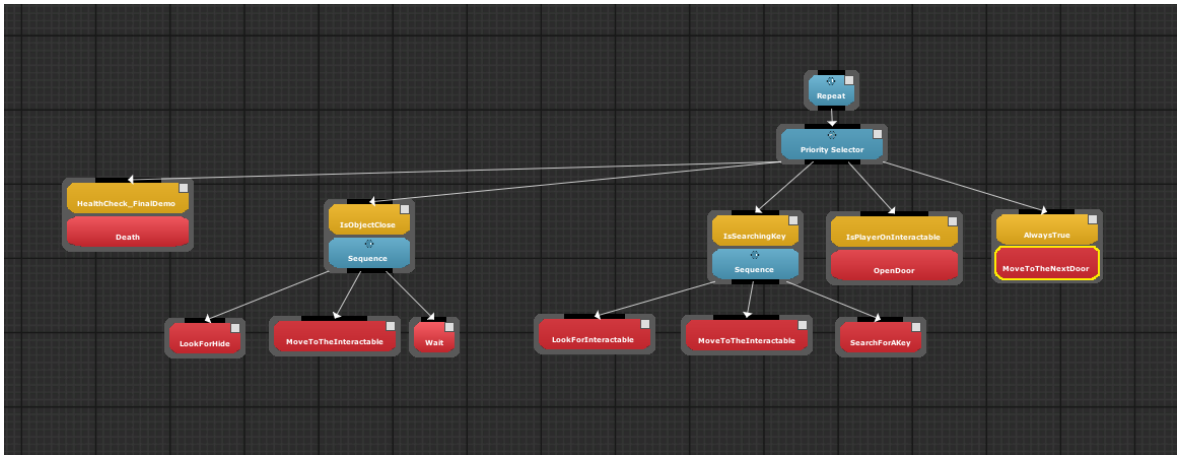
```
public void MoveSquad(Vector3 targetPosition)
    {
        float distance = 3f;
        int positionCount = agentsInSquad.agents.Count;
        List<Vector3> positionList = GetPositionsForSquad(
  ↪ targetPosition, distance, positionCount);
        for (int i = 0; i < positionCount; i++)
        {
            agentsInSquad.agents[i].MovePoint = positionList[i];
        }
    }
```

Základnou triedou na udržiavanie informácií o skupinách v scéne je AgentBase. V celom projekte sa používa len jedna inštancia triedy, pomocou návrhového vzoru Singleton. Trieda obsahuje informácie, ktoré medzi sebou zdieľajú jednotlivé NPC typu Enemy. Nájdeme v nej list všetkých agentov nachádzájúcich sa v scéne, list všetkých skupín v scéne. Uchováva aj informáciu o pozícii, kde agenti naposledy videli hráča, či je hráč mŕtvy atď.

```
public class AgentBase : MonoBehaviour
{
    #region Singleton
    private static AgentBase instance;
    public static AgentBase Instance
    {
        get
        {
            return instance;
        }
    }
    private void Awake()
    {
        if (instance == null)
            instance = this;
    }
    #endregion
    public List<EnemyController_FinalDemo> AvailableAgentsInTheScene;
    public Transform PlayerLastPosition;
    public bool IsPlayerDead;
    public EnemyController_FinalDemo AgentWhoSawPlayer;
    public List<SquadControl> Squads;
}
```

8.0.1 Strom Player

Rovnako ako v predošlom prípade začíname Priority Selektorom, ktorý spúšťa jednotlivé úlohy zľava doprava (Obr. 8.1). Keďže sa jedná o NPC, ktorý môže počas rozhodovania umrieť, ako najprioritnejšiu podmienku pri návrhu stromu sme zvolili podmienku HealthCheck_FinalDemo.



Obr. 8.1 Strom Player

HealthCheck_FinalDemo

Podmienka funguje úplne rovnako ako HealthCheck_BTDemo v predchádzajúcej kapitole. S rozdielom, že meníme typ controller scriptu na PlayerController. V prípade, že je životná hodnota menšia alebo rovná nule, vykoná sa akcia Death ktorá spustí animáciu umierania.

Následne ako druhú najprioritnejšiu podmienku NPC kontroluje, či sa k nemu náhodou neblíži druhý NPC (Enemy) pomocou počítania cesty za pomoci NavMeshPath. Ak sa Enemy nachádza v blízkosti, začne sa vykonávať sekvencia pozostavájúca z troch akcií. Prvou akciou je LookForHide.

LookForHide

V tejto akcii si NPC nájde najbližšie miesto, kde sa môže schovať. Keď skrýšu nájde, presunieme sa do akcie MoveToTheInteractable, ktorá je veľmi podobná akcii z predchádzajucej kapitoly MoveToThePoint. Ak sa chrakter dopraví do skrýše, počká tam určitý čas v už spomenutej akcii Wait.

```
[Action("LookForHide")]
[Help("Looking for hide point")]
public class LookForHide : BaseAction
{
```

```
    [InParam("Enemy")]
    public GameObject Enemy;
    [OutParam("Hide")]
    public Interactable Hide;
    public override void OnStart()
    {
        base.OnStart();
 MainCharacter pozeral
        motor.Target = Enemy.transform;
    }
    public override TaskStatus OnUpdate()
    {
        float distance = Vector3.Distance(gameObject.transform.
   ↪ position, activeQuest.PlacesToHide[0].InteractPosition.position)
   ↪ ;
        int x = 0;
        for (int i = 1; i < activeQuest.PlacesToHide.Count; i++)
        {
            float distanceToPoint = Vector3.Distance(gameObject.
   ↪ transform.position, activeQuest.PlacesToHide[i].InteractPosition
   ↪ .position);
            if (distance > distanceToPoint)
            {
                distance = distanceToPoint;
                x = i;
            }
        }
        Hide = activeQuest.PlacesToHide[x];
        return TaskStatus.COMPLETED;
    }
}

[Action("MoveToTheInteractable")]
[Help("Moving to the interactable object.")]
public class MoveToTheInteractable : BaseAction
{
    [InParam("Interactable")]
    Interactable nextObject;
    public override void OnStart()
    {
        base.OnStart();
        motor.SendCharacterToDestination(nextObject.InteractPosition);
    }
    public override TaskStatus OnUpdate()
    {
        if (!motor.navMeshAgent.pathPending  motor.navMeshAgent.
   ↪ remainingDistance
            <= motor.navMeshAgent.stoppingDistance)
            return TaskStatus.COMPLETED;
        return TaskStatus.RUNNING;
    }
}
```

Ak je hráč stále živý a v okolí sa nenachádza žiaden Enemy, presunieme sa na podmienku IsSearchingKey.

IsSearchingKey

V tejto podmienke kontrolujeme, či hráč vie aký kľúč hľadať a či už náhodou správny kľúč nenašiel. Ak vie aký kľúč má hľadať, presunieme sa do sequencie, ktorej cieľom je prehľadávať jednotlivé objekty a pokúsiť sa nájsť kľúč.

```
[Condition("Action/IsSearchingKey")]
[Help("Checks if player know what key to look for and if the key was
   ↪ not found")]
public class IsSearchingKey : GOCondition
```

```
{
    [InParam("PlayerController")]
    public PlayerController player;
    public override bool Check()
    {
        if (player.KeyToFind != ""  player.keyFound == false)
            return true;
        else return false;
    }
}
```

LookForInteractable

Začneme akciou LookForInteractable, v ktorej si z aktívneho questu zistíme pozíciu objektu, v ktorého inventári sa môže nachádzať správny kľúč. Akonáhle získame kandidáta, začneme sa presúvať pomocou akcie MoveToTheInteractable. Keď dôjdeme k objektu, strom sa presunie do akcie SearchForAKey.

SearchForAKey

V akcii zavoláme už spomenutú funkciu MoveAllItems(Inventory otherInventory) a označíme objekt za skontrolovaný.

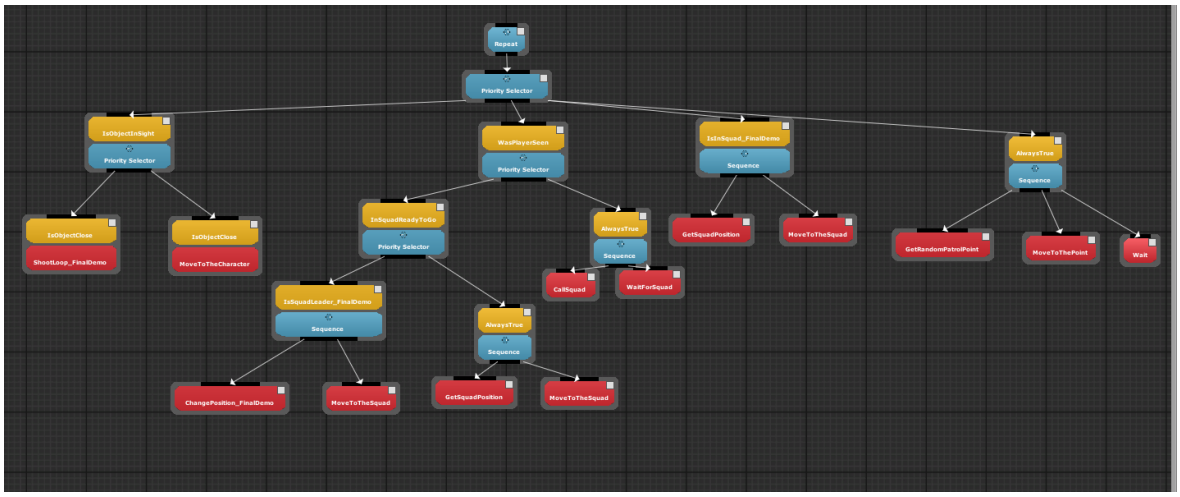
```
[Action("SearchForAKey")]
[Help("Searching for a key.")]
public class SearchForAKey : BaseAction
{
    public override void OnStart()
    {
        base.OnStart();
        player.m_anim.SetTrigger("Interaction");
    }
    public override TaskStatus OnUpdate()
    {
        if (player.InteractingWith != null)
        {
            player.InteractingWith.MoveAllItems(player.Inventory);
            player.InteractingWith.isChecked = true;
            return TaskStatus.COMPLETED;
        }
        return TaskStatus.RUNNING;
    }
}
```

Ak hráč nevie aký kľúč potrebuje, presunieme sa na podmienku **IsPlayerOnInteractable**. V podmienke kontrolujeme, či hráč interaguje s dverami. Ak áno, spustíme akciu **OpenDoor**, v ktorej sa pokúsime otvoriť dvere. Ak nie, presunieme sa na poslednú časť stromu s podmienkou typu AlwaysTrue a spustíme akciu **MoveToTheNextDoor**,v ktorej na základe aktívneho questu pošleme NPC k ďalším dverom, ktoré bude odomykať. Akcia funguje na podobnom princípe ako už spomínaná akcia MoveToThePoint.

```
[Action("OpenDoor")]
[Help("Try to open the door")]
public class OpenDoor : BaseAction
```

```
{
    public override TaskStatus OnUpdate()
    {
        if (activeQuest.door.isLocked)
        {
            if (player.KeyToFind == "")
                player.KeyToFind = activeQuest.door.keyToUnlock;
            else
            {
                if(player.keyFound)
                {
                    activeQuest.door.DoorUnlock();
                    player.KeyToFind = "";
                    player.keyFound = false;
                    activeQuest.door.Interacted = true;
                    player.InteractingWith = null;
                    activeQuest.isFinished = true;
                    return TaskStatus.COMPLETED;
                }
            }
        }
        return TaskStatus.FAILED;
    }
}
```

8.0.2 Strom Enemy



Obr. 8.2 Strom Enemy

IsObjectInSight

V tejto akcii kontrolujeme, či NPC vidí hráča. Videnie je implementované podobným spôsobom ako strieľanie. Rovnako ako pri strieľaní vysielame lúč a kontrolujeme, či narazil na nejaký objekt. Ak narazil na obejkt, ktorý predstavuje hráča, znamená to, že hráč sa nachádza v zornom poli NPC. Rozdiel oproti strieľaniu je, že pri strieľaní vysielame jeden lúč smerom vpred, avšak tu vysielame lúče v zornom poli NPC, čo v našej implementácii predstavuje 130 stupňov. Ak teda NPC zahliadne hráča, zmeníme animáciu, aktualizujeme hráčovu pozíciu a NPC, ktorý uvidel hráča v triede AgentBase. Ak NPC vidí hráča, no hráč je už mŕtvy rozpustí skupinu ak v nejakej bol.

```
[Condition("MyConditions/IsObjectInSight")]
[Help("Checks whether a target is close depending on a given distance"
    ↪ )]
public class IsObjectInSight : GOCondition
{
    [InParam("EnemyController")]
    public EnemyController_FinalDemo enemyController;
    [InParam("Target")]
    public GameObject Target;
    [InParam("LookOrigin")]
    public GameObject LookOrigin;
    private float fieldOfViewAngle = 130f;
    [InParam("SeeDistance")]
    [Help("The maximun distance to consider that the target is close")
   ↪ ]
    public float seeDistance;
    public override bool Check()
    {
   ect.transform.position;
        float angle = Vector3.Angle(direction, gameObject.transform.
   ↪ forward);
        if (angle < fieldOfViewAngle * 0.5f)
        {
            RaycastHit hit;
            if (Physics.Raycast(LookOrigin.transform.position,
            direction.normalized, out hit, seeDistance))
            {
                Debug.DrawRay(LookOrigin.transform.position, direction
   ↪ .normalized);
                if (hit.collider.gameObject == Target)
                {
                    if (hit.collider.gameObject.GetComponent<
   ↪ HealthControl>() != null)
                     if (hit.collider.gameObject.GetComponent<
   ↪ HealthControl>().CurrentHealth <= 0)
                     {
                         AgentBase.Instance.IsPlayerDead = true;
                         foreach (var squad in AgentBase.Instance.
   ↪ Squads)
                         {
                             foreach (var agent in squad.agents)
                             {
                                 agent.InSquad = false;
                                 agent.SquadLead = false;
                                 agent.WasPlayerSeen = false;
                             }
                         }
                         return false;
                     }
                    enemyController.motor.ChangeStoppingDistance(
   ↪ AISettings_StoppingDistance.Shoot);
                    AgentBase.Instance.PlayerLastPosition = Target.
   ↪ transform;
                    AgentBase.Instance.AgentWhoSawPlayer =
   ↪ enemyController;
                    enemyController.WasPlayerSeen = true;
                    enemyController.Animation_SeePlayer();
                    return true;
                }
            }
        }
        enemyController.Animation_DoNotSeePlayer();
        return false;
    }
}
```

Ak je táto podmienka splnená, teda Enemy vidí Playera, presunieme sa o level nižšie v strome do ďalšieho Priority Selektoru, v ktorom máme dve podmienky rovnakého typu **IsObjectClose**. V tomto prípade podmienka funguje na rovnakom princípe ako

v predchádzajúcej kapitole podmienka IsObjectClose_BTDemo. Začneme teda kontrolovať podmienku s vyššou prioritou (na ľavej strane), po ktorej splnení sa charakter prepne do akcie ShootLoop_FinalDemo, ktorá taktiež funguje na rovnakom princípe ako v predchádzajúcom prípade. Ak nie je Player dostatočne blízko, aby bol po ňom schopný Enemy strieľať, skontroluje, či je aspoň dostatočne blízko na prenasledovanie. Ak áno, spustíme úlohu MoveToTheCharacter, ktorá funguje na rovnakom princípe ako úloha MoveToThePoint.
V prípade, že enemy nevidí hráča, presunieme sa na podmienku WasPlayerSeen.

WasPlayerSeen

Táto podmienka je jednoduchá. Jediné čo urobíme je, že sa pozrieme do EnemyController skriptu na premenú typu bool WasPlayerSeen. Ak sa rovná true zmeníme stoppingDistance NPC a aktualizujeme premenú PlayerLastSeenPosition poslednou pozíciou, na ktorej bol videný hráč. Následne hodnotu tejto premenej predávame ďalej do podstromu. Ak je podmienka splnená, prechádzame o úroveň nižšie do ďalšieho priority selektoru, v ktorom začíname podmienkou InSquadReadyToGo.

```
[Condition("MyConditions/WasPlayerSeen")]
public class WasPlayerSeen : GOCondition
{
    [InParam("EnemyController")]
    public EnemyController_FinalDemo enemyController;
    [OutParam("PlayerLastSeenPosition")]
    public Transform PlayerLastSeenPosition;
    public override bool Check()
    {
        if (enemyController.WasPlayerSeen == true)
        {
            enemyController.motor.ChangeStoppingDistance(
↪ AISettings_StoppingDistance.Squad);
            PlayerLastSeenPosition = AgentBase.Instance.
↪ PlayerLastPosition;
            return true;
        }
        else return false;
    }
}
```

InSquadReadyToGo

V tejto podmienke kontrolujeme, či je NPC v skupine a zároveň či sa celá skupina nachádza na jednom mieste. Ak vyhovujú spomenuté podmienky, podmienka vráti pravdu. V prípade, že podmienka zlyhala presunieme sa na pravú stranu selektoru do podmienky typu Always true a začneme vykonávať úlohu CallSquad v sequencii.

```
[Condition("MyConditions/InSquadReadyToGo")]
public class InSquadReadyToGo : GOCondition
{
    [InParam("EnemyController")]
    public EnemyController_FinalDemo enemyController;
```

```
    public override bool Check()
    {
        if (enemyController.InSquad  enemyController.SquadArrived)
        {
            return true;
        }
        else return false;
    }
}
```

CallSquad

Na začiatku úlohy CallSquad kontrolujeme, či NPC v minulosti vytvoril nejakú skupinu ktorá je momentálne aktívna. Ak tomu tak nie je, vytvoríme skupinu zavolaním funkcie CreateSquad() a zastavíme charakter. V prípade že úloha uspela, presunieme sa do úlohy WaitForSquad.

```
[Action("MyActions/CallSquad")]
public class CallSquad : EnemyAIBase
{
    public override TaskStatus OnUpdate()
    {
        if (!enemyController.SquadCreated)
        {
            enemyController.SquadCreated = true;
            enemyController.CreateSquad();
            enemyController.motor.SendCharacterToDestination(
↪ enemyController.transform);
            return TaskStatus.COMPLETED;
        }
        else return TaskStatus.FAILED;
    }
}
```

WaitForSquad

V tejto akcii NPC čaká na pozícii, kým neprídu ďalší členovia skupiny. Čakanie sa uskutočňuje na základe vzdialenosti jednotlivých NPC. V prípade, že sa členovia skupiny dostavia, zmeníme im premenné SquadArrived a WasPlayerSeen na true. Meníme ich primárne z dôvodu, aby sa v rozhodovacom strome presunuli do častí, v ktorých môžu jednať ako skupina.

```
[Action("MyActions/WaitForSquad")]
public class WaitForSquad : EnemyAIBase
{
    public override TaskStatus OnUpdate()
    {
        if (enemyController.agentsInSquad.agents[1].motor.
        CaclculatePathDistance(enemyController.gameObject) > 3
             enemyController.agentsInSquad.agents[2].motor.
            CaclculatePathDistance(enemyController.gameObject) > 3)
        {
            return TaskStatus.RUNNING;
        }
        foreach (var agent in enemyController.agentsInSquad.agents)
        {
            agent.SquadArrived = true;
            agent.WasPlayerSeen = true;
        }
```

```
        return TaskStatus.COMPLETED;
    }
}
```

Keď sa vrátime k podmienke InSquadReadyToGo a predpokladáme, že bola splnená presunieme sa do ďalšieho priority selektoru o úroveň ňižšie. Začneme znovu na ľavej strane v podmienke IsSquadLeader_FinalDemo.

IsSquadLeader_FinalDemo

V tejto podmienke kontrolujeme, či je NPC líder skupiny. Lídrom skupiny sa NPC stáva pri tvorbe skupiny. Ostatní NPC sú len členovia. V návrhu stromu sme oddelili rozhodovanie lídra a členov primárne z dôvodu, aby sa nerozhodoval každý NPC v skupine samostatne. Líder skupiny dokáže robiť rozhodnutia, kam sa celá skupina presunie, členovia ho nasledujú. Ak podmienka uspeje, presunieme sa do sequencie na úlohu ChangePosition_FinalDemo.

```
[Condition("MyConditions/IsSquadLeader_FinalDemo")]
public class IsSquadLeader : GOCondition
{
    [InParam("EnemyController")]
    public EnemyController_FinalDemo enemyController;
    public override bool Check()
    {
        if (enemyController.SquadLead)
        {
            return true;
        }
        else return false;
    }
}
```

ChangePosition_FinalDemo

V tejto úlohe zavoláme funkciu MoveSquad, do ktorej vložíme pozíciu, kde sme naposledy videli hráča a aktualizujeme premenú SquadDestination rovnakou pozíciou. Hodnotu tejto premenej ďalej predávame do úlohy **MoveToTheSquad**, ktorá funguje na rovnakom princípe ako MoveToThePoint. Ibaže berie vstupný parameter typu Vector3. V prípade, že NPC nie je líder skupiny, presunieme sa do podmienky typu AlwaysTrue a následne do sequencie.

```
[Action("ChangePosition_FinalDemo")]
public class ChangePosition_FinalDemo : EnemyAIBase
{
    [InParam("Player")]
    public GameObject Player;
    [OutParam("SquadDestination")]
    public Vector3 SquadDestination;
    public override TaskStatus OnUpdate()
    {
        enemyController.MoveSquad(AgentBase.Instance.
↪ PlayerLastPosition.position);
        SquadDestination = enemyController.MovePoint;
```

```
            return TaskStatus.COMPLETED;
        }
}
```

GetSquadPosition

V úlohe aktualizujeme premenú SquadDestination hodnotou, ktorú predal líder skupiny v akcii ChangePosition_FinalDemo. Akonáhle NPC získa pozíciu, presunieme sa do už spomenutej akcie **MoveToTheSquad**.

```
[Action("GetSquadPosition")]
public class GetSquadPosition : EnemyAIBase
{
    [OutParam("SquadDestination")]
    public Vector3 SquadDestination;
    public override void OnStart()
    {
        SquadDestination = enemyController.MovePoint;
    }
}
```

Keď sa vrátime o kúsok naspäť do Priority Selektoru z podmienky WasPlayerSeen, presunieme sa do nasledujúcej časti stromu, do podmienky IsInSquad_FinalDemo.

IsInSquad_FinalDemo

V tejto podmienke kontrolujeme, či bol daný NPC pridaný do skupiny. Podmienka má nižšiu prioritu ako podmienka WasPlayerSeen z dôvodu oddelenia správania bežného NPC od lídra skupiny. Ak podmienka vráti úspech, presunieme sa do akcie MoveToTheSquad.

Ako poslednú podmienku v Priority Selektore máme podmienku typu AlwaysTrue, po ktorej splnení nasleduje sequencia obsahujúca úlohy spojené s hliadkovaním po scéne.

GetRandomPatrolPoint

Akcia funguje na rovnakom princípe ako akcia GetRandomPatrolPoint_BTDemo z predchádzajúceho dema. Následne, ako dostaneme miesto na hliadkovanie, sequencia sa presunie do už spomínanej úlohy MoveToThePoint a po príchode NPC na miesto počkáme zadaný čas v úlohe Wait.

Zdroje grafických assetov

Pri tvorbe jednotlivých častí finálneho dema sme použili nasledujúce assety:

- skybox SpaceSkies Free https://assetstore.unity.com/packages/2d/textures-materials/sky/spaceskies-free-80503,

- prostredie Modular Sci-Fi Corridor https://assetstore.unity.com/packages/3d/environments/sci-fi/modular-sci-fi-corridor-142811,
- modely charakterov aj s animáciami https://www.mixamo.com/,
- modely zbraní https://devassets.com/assets/modern-weapons/.

ZÁVER

Cieľom tejto bakalárskej práce bolo demonštrovať a priblížiť rôzne prístupy a spôsoby implementácie hľadania ciest a rozhodovania v herných aplikáciach. Pri rozoberaní hľadania ciest sme sa bližšie pozreli na algoritmy A* a Dijkstra. Pri každom sme rozobrali funkcionalitu, algoritmus a využitie. Oba algoritmy sme vložili do hernej scény a vizualizovali ich rozdielny prístup na hľadanie najkratšej cesty z bodu A do bodu B.

Obidva algoritmy fungujú na podobnom princípe, avšak Dijkstra na určenie váhy cesty používa cenu prechodu do uzla, ktorá sa skladá zo všetkých cien prechodov potrebných na prejdenie, aby sa charakter dostal do cieľového uzla. V praxi to vyzerá tak, že algoritmus hľadá najkratšiu cestu do jednotlivých susedných uzlov až pokiaľ nenarazí na cieľový uzol. Keďže v tejto bakalárskej práci rozoberáme hľadanie cesty z jedného bodu do druhého (point-to-point) je tento prístup neoptimálny. Keďže Dijkstra pracuje len s jednou hodnotou (celkovou cenou cesty) nedokáže odhadnúť, ktorý susedný uzol bude pravdepodobne ten, ktorý bude viesť najkratšou cestou do cieľa. Preto prehľadáva všetky uzly. Aj z tohoto dôvodu sa Dijkstra používa na hľadanie ciest z bodu do bodu len v jednoduchších hrách. V tých zložitejších si môže nájsť svoje využitie v strategickej časti umelej inteligencie, ako napríklad hľadanie najkratšej možnej cesty na viaceré miesta zároveň.

A* algoritmus je nadizajnovaný priamo na hľadanie cesty z bodu do bodu. Funguje veľmi podobne ako Dijkstra, ibaže namiesto prehodnocovania uzlu s najnižšou celkovou cenou si algoritmus vyberie uzol, pri ktorom má najväčšiu pravdepodobnosť, že vedie najkratšou cestou k cieľovému uzlu. Aby dokázal algoritmus odhadnúť správny uzol používa heuristický spôsob, pri ktorom počíta celkovú odhadovanú cenu uzlu. Tento odhad ceny sa počíta sčítaním celkovej ceny presunu na uzol, ktorý prehodnocuje a ceny vzdialenosti tohoto uzla od cieľového uzla. Následne sa rozhoduje pre uzol s najnižšou získanou hodnotou po tomto sčítaní. Algoritmus sa teda rozhoduje na základe sčítania dvoch hodnôt a nie na pravdepodobnosti ako sme si uviedli pár viet dozadu. Uzol s najnižšou odhadovanou cenou sme nazvali najpravdepodobnejší, pretože algoritmus sa môže aj mýliť. V takom prípade dokáže byť menej efektívny ako Dijkstra. Pri našom testovaní, kde sme porovnávali počet iterácii a čas potrebný na nájdenie najkratšej cesty v prípade bez prekážok a s prekážkami, nám jednoznačne vyšlo v oboch prípadoch, že A* je výkonejší pre hľadanie cesty z bodu do bodu.

V ďalšej časti sme sa pozreli na dva prístupy k vytvoreniu rozhodovania v herných aplikáciách - konečné automaty a rozhodovacie stromy. Cieľom tejto časti bolo porovnať dva prístupy z hľadiska implementačnej náročnosti a škálovateľnosti. Na účelné porovnanie týchto dvoch metód sme zvolili iteratívny prístup, ktorý sa využíva v reál-

nom vývoji. V prvom kroku sme vytvorili základný návrh správania pre obe metódy, následne sme daný návrh otestovali, analyzovali nedostatky a urobili ďalšiu iteráciu s cieľom vylepšiť už funkčný návrh. Zvolením tohoto prístupu sme dokázali demonštrovať výhody a nevýhody jednotlivých prístupov a taktiež otestovať ich škálovateľnosť. V prípade konečných automatov je výhodou jednoduchá implementácia, keďže tento koncept je pomerne známy. Tento prístup je ideálna voľba pre jednoduché rozhodovanie, kde neočakávame, že dané rozhodovanie bude naberať na komplexnosti. Tým sa dostávame k škálovateľnosti. Ako už vieme, konečný automat sa do jednotlivých stavov dostáva pomocou prechodov, ktoré obsahujú pravidlá rozhodujúce o tom, či bude daný prechod uskutočnený. Keď potrebujeme niečo zmeniť, vždy musíme myslieť na jednotlivé prechody medzi stavmi a väčšinou brať do úvahy každý jeden stav. To znamená, že pri pridaní nového stavu, alebo zmeny len určitej časti automatu, musíme aktualizovať jednotlivé prechody a podmienky daných prechodov, aby s novým stavom začali počítať. V jednoduchších rozhodovaniach to nie je až taký problém, no pri zložitejších to môže byť veľmi náročné. Aj z toho dôvodu sa na zložitejšie prípady používajú rozhodovacie stromy, ktoré používajú rôzne podmienky a kompozity, na základe ktorých púšťajú akcie. Vďaka týmto rôznym druhom "uzlov" v strome si vieme nadizajnovať rozhodovanie, ktorého jednotlivé časti sú od seba takmer nezávistlé. Keď chceme vylepšiť rozhodovanie NPC pri strieľaní, zameriame sa práve na tú časť stromu a ostatné časti zostanú bez zmeny. Ďalšou výhodou tohoto prístupu je udržovateľnosť, keďže sú jednotlivé akcie od seba takmer nezávislé. Z toho vyplýva, že rozhodovacie stromy sú dobrou voľbou pre väčšie projekty, v ktorých sa očakáva komplexnejšie rozhodovanie. Tento prístup nie je zložitý na naučenie, no člen vývojárskeho týmu, ktorý sa ním bude zaoberať, strávi viac času učením sa ako keby sa zaoberal konečnými automatmi. Je to aj z toho dôvodu, že väčšinou sa na tvorbu rozhodovacích stromov používajú rôzne frameworky, s ktorými sa musí vývojár zoznámiť. Samozrejme, takýto framework sa dá vytvoriť, no zaberie to nejaký čas.

Poslednú časť bakalárskej práce tvorí finálne demo, v ktorom sme vytvorili interaktívne bludisko pre dva druhy NPC. Cieľom tejto časti bolo prakticky demonštrovať možnosti správania a kolaborácie jednotlivých charakterov za použitia rozhodovacích stromov.

ZOZNAM POUŽITEJ LITERATÚRY

[1] *LAVIERI, Dr. Edward. Getting Started with Unity 2018: A Beginner's Guide to 2D and 3D game development. 3rd edition.* Birmingham: Packt Publishing, 2018. ISBN 978-1-78883-010-2.

[2] *AVERSA, Dr. Davide, Aung Sithu KYAW a Clifford PETERS. Unity Artificial Intelligence Programming: Add powerful, believable, and fun AI entities in your game with the power of Unity 2018!. Fourth Edition.* Birmingham: Packt Publishing, 2018. ISBN 978-1-78953-391-0.

[3] *MILLINGTON, Ian a John David FUNGE. Artificial intelligence for games. 2nd ed.* Burlington, MA: Morgan Kaufmann/Elsevier, c2009. ISBN 01-237-4731-7.

[4] *Quick Start Guide. In: Behavior Bricks [online]. Madrid: PADAONE GAMES S.L. [cit. 2020-06-16].* Dostupné z: http://bb.padaonegames.com/doku.php?id=quick:design

[5] *Blend Trees. In: Unity User Manual (2019.4 LTS) [online]. Unity Technologies, [2020 cit. 2020-06-16].* Dostupné z: https://docs.unity3d.com/Manual/class-BlendTree.html

[6] *ScriptableObject. In: Unity Documentation [online]. Unity Technologies, 2020 [cit. 2020-06-18].* Dostupné z: https://docs.unity3d.com/Manual/class-ScriptableObject.html

[7] *Bonfiglio, Nahila (October 1, 2018). "DeepMind partners with gaming company for AI research".* The Daily Dot. Archived from the original on October 2, 2018. Retrieved October 17, 2018.

[8] *Captain, Sean (September 19, 2017). "Machine Learning Is Making Video Game Characters Smarter And Robots More Competent". Fast Company. Archived from the original on November 21, 2018.* Retrieved November 26, 2018.

[9] *"Using DirectX11 in Unity 4". Unity Technologies. Archived from the original on March 12, 2013.* Retrieved February 19, 2013.

[10] *Fine, Richard (August 11, 2017). "UnityScript's long ride off into the sunset". Unity Technologies Blog. Archived from the original on October 17, 2017.* Retrieved September 18, 2017.

[11] *Unity Manual: Coroutines. In: Unity Documentation [online].* Unity Technologies, 2020 [cit. 2020-07-16]. Dostupné z: https://docs.unity3d.com/Manual/Coroutines.html

[12] Edsger W. Dijkstra - Structured Programming. BROY, Manfred a Ernst DENERT. Software Pioneers: Contributions to Software Engineering. Berlin: Springer Science & Business Media, 2002, s. 19. ISBN 9783540430810.

[13] Dijkstra's Algorithm. SKIENA, Steven S. The Algorithm Design Manual. 2nd. London: Springer Science & Business Media, 2009, s. 206. ISBN 978-1-84800-069-8.

[14] A* Pathfinding. BOURG, David M. a Glenn SEEMANN. AI for Game Developers. 1005 Gravenstein Highway North, Sebastopol: O'Reilly Media, 2004, s. 126. ISBN 9780596005559.

[15] A_Star (A*) Algorithm. Proceedings of the International Conference on Advanced Intelligent Systems and Informatics 2017. New York, NY: Springer Berlin Heidelberg, 2017, s. 74. ISBN 9783319648606.

ZOZNAM POUŽITÝCH SYMBOLOV A SKRATIEK

AAA	Označenie pre videohru vyvíjanú s vysokým rozpočtom a veľkým vývojárskym tímom
NPC	Označenie pre herný charakter ktorý nieje ovládaný hračom (Non playable character)
RPG	Hra na hrdinov (Role-playing game)
AI	Skratka pre Artificial Intelligence (Umelá Inteligencia)

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ZOZNAM PRÍLOH

P I. prilohy.zip

PRÍLOHA P I. PRILOHY.ZIP

Príloha prilohy.zip obsahuje dve zložky: Verzia_1.0.0 a BakalarskaPraca. V zložke Verzia_1.0.0 nájdete spustiteľný build projektu. V zložke BakalarskaPraca nájdete Unity projekt, kompatibilný s verziou Unity 2018.4.12f1.