VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

UMELÁ INTELIGENCIA PRE STRATEGICKÉ HRY

ARTIFICIAL INTELLIGENCE IN STRATEGIC GAMES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ĽUBOŠ ŠČEVIK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. MICHAL MATÝŠEK**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Ščevik Ľuboš**
Program: Informační technologie
Název: **Umělá inteligence pro strategické hry**
Artificial Intelligence for Strategy Games
Kategorie: Umělá inteligence

Zadání:

1. Prostudujte metody umělé inteligence využitelné při vývoji strategických počítačových her.
2. Seznamte se s obecnými postupy tvorby her v enginu Unity. Prostudujte dostupné/podporované nástroje a knihovny umožňující vývoj umělé inteligence pro strategické hry.
3. Navrhněte strategickou hru a vyberte konkrétní metodu umělé inteligence vhodnou pro tuto hru.
4. Implementujte navrženou hru a vybranou metodu.
5. Proveďte experimenty prezentující věrohodnost umělé inteligence a její adaptaci na změny prostředí a stav hry.
6. Zhodnoťte dosažené výsledky, vytvořte krátké prezentační video a diskutujte možnosti budoucího vývoje.

Literatura:

- Gregory, Jason. *Game engine architecture*. crc Press, 2018. ISBN 1351974289, 9781351974288
- Adams, Ernest, and Joris Dormans. Game mechanics: advanced game design. New Riders, 2012. ISBN 0321820274, 9780321820273

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

Vedoucí práce: **Matýšek Michal, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 1. listopadu 2021

Abstrakt

Táto bakalárska práca sa zaoberá umelou inteligenciou pre strategické hry v reálnom čase. Pri tvorbe umelej inteligencie sa využila metóda Monte Carlo. Hra, spolu s umelou inteligenciou, bola vytvorená v hernom engine Unity.

Abstract

This bachelor thesis deals with artificial intelligence in real-time strategy games. The Monte Carlo method was used in the creation of artificial intelligence. The game, along with artificial intelligence, was made using the Unity game engine.

Kľúčové slová

Strategická hra v reálnom čase, Umelá inteligencia, Unity, Monte Carlo, Minmax, Lanchester, Lanchesterovo mocninové pravidlo

Keywords

Real-Time strategic game, Artificial intelligence, Unity, Monte Carlo, Minmax, Lanchester, Lanchester Square Law

Citácia

ŠČEVIK, Ľuboš. *Umelá inteligencia pre strategické hry.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Michal Matýšek

Umelá inteligencia pre strategické hry

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Michala Matýška. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Ľuboš Ščevik
10. mája 2022

Poďakovanie

Chcem sa poďakovať vedúcemu mojej práce pánovi Ing. Michalovi Matýškovi.

Obsah

1 Úvod **2**

2 Analýza vývoja umelej inteligencie a strategických hier **3**
2.1 Vývoj RTS hier v 21. storočí 4
2.2 Analýza existujúcich RTS hier a ich umelej inteligencie 4
2.3 AlphaGo 6
2.4 AlphaStar AI pre StarCraft II 7
2.5 Pravidlová umelá inteligencia 7
2.6 Stromové vyhľadávacie algoritmy 8
2.7 Lanchesterové pravidlá 11

3 Návrh strategickej hry v Unity **13**
3.1 Pravidlá hry 13
3.2 Umelá inteligencia 15

4 Implementácia **17**
4.1 Herný engine 17
4.2 Mapa 18
4.3 Jednotky 18
4.4 Umelá inteligencia 19
4.5 Používateľské rozhranie 26

5 Vyhodnotenie **27**
5.1 Úvod do testovania 27
5.2 Modrá umelá inteligencia verzus červená umelá inteligencia 28
5.3 Zhrnutie testov 29

6 Záver **30**

Literatúra **31**

Kapitola 1

Úvod

RTS žáner v poslednej dobe nadobúda druhý dych a vracia sa späť na výslnie. Každým dňom uvidí svetlo sveta množstvo nových RTS hier, avšak veľká časť z nich je zamerané najmä alebo výhradne na multiplayer (hru pre viacerých hráčov). Jedným z dôvodov je aj náročnosť vytvorenia kvalitnej umelej inteligencie.

Cieľom mojej bakalárskej práce je návrh a implementácia umelej inteligencie, ktorá by ponúkla riešenie pre hlavné problémy terajších umelých inteligencií používaných pri strategických hrách v reálnom čase. Medzi tieto problémy patrí minimálna flexibilita, časté podvádzanie vo forme vyššieho množstva zdrojov a informácií ku ktorým by normálny hráč nemal prístup, ako napríklad poloha jednotiek schovaných vo Fog of War, vojnovej hmle. Budem sa zaoberať potenciálnym riešením v podobe umelej inteligencie využívajúcej Monte-Carlo metódu.

V druhej kapitole sa venujem rôznym druhom umelej inteligencie a niektorým existujúcim riešeniam. V tretej kapitole je rozobraný návrh strategickej hry a jej umelej inteligencie. V štvrtej kapitole je vysvetlená implementácia môjho riešenia. Výsledok testovania je zanalyzovaný v piatej kapitole. V šiestej kapitole sú vyhodnotené dosiahnuté výsledky a rozobraté potenciálne možnosti budúceho vývoja.

Kapitola 2

Analýza vývoja umelej inteligencie a strategických hier

V Európe je najznámejším predstaviteľom strategických hier šach. Pôvodom z Indie, šach sa dostal do Európy niekedy počas 11. storočia. Obľube širokej verejnosti sa dočkal až počas romantizmu. Šach tej doby bol charakteristický najmä rýchlymi taktickými manévrami a ofenzívne orientovaným štýlom hry. Hráči k hre pristupovali skôr ako k umeniu, než k niečomu čo otestuje ich logické myslenie a predvídavosť. Avšak to sa rýchlo zmenilo s príchodom matematikov ako Wilhelm Steinitz [14], ktorý sa stal prvým svetovým šampiónom v šachu, keď zdolal nemeckého majstra Johannesa Zukertora, či Emanuel Lasker [8], ktorý zdolal Steinitza a udržal si titul svetového šampióna nasledujúcich 27 rokov. K hre pristupovali vedecky a so svojimi stratégiami navždy zmenili spôsob, akým sa ľudia pozerajú na šach.

V 20. storočí sa strategické hry dostali zo stolov aj na počítače. To umožnilo tento žáner inovovať do podoby strategických hier v reálnom čase ďalej len RTS (real-time strategy). RTS hry sú v podstate zjednodušené simulátory vojenských konfliktov. Medzi ich predstaviteľov patria napríklad Age of Empires, Command & Conquer, Warcraft, Wargame: Red Dragon, Warhammer 40,000: Dawn of War. Sú charakteristické tým, že hráčom zo začiatku dajú malý počet jednotiek a ten ich musí využiť k zbieraniu surovín, rozširovaniu základne a stavaniu ďalších jednotiek, ktoré neskôr využije k zničeniu základne nepriateľa.

Avšak ešte väčším pokrokom bola možnosť hrať proti počítaču. Vývoj umelej inteligencii pomerne rýchlo napredoval, už v roku 1965 sa uskutočnila prvá partia šachu, v ktorej sa streli dve umelé inteligencie, prvá pochádzala zo Stanfordskej univerzity a druhá z Inštitútu pre teoretickú a experimentálnu fyziku v Moskve [13, 21]. V roku 1996 umelá inteligencia Deep Blue, vyvíjaná od roku 1985, prvýkrát porazila úradujúceho majstra sveta v šachu Kasparova [7].

Avšak vývoj umelej inteligencie pre RTS hry napredoval pomalším tempom. Hlavným dôvodom je neporovnateľne väčší počet možných stavov. V porovnaní so šachom sú mapy rozsiahlejšie, počet realizovateľných ťahov sa tiež zvýšil. Pridala sa aj neistota ohľadom splnenia príkazov. Pri šachu viete, že ak posuniete vežu o dve políčka dopredu, tak sa tam figúrka dostane predtým, než oponent zahrá jeho vlastný ťah. Pri RTS hrách to neviete zaručiť. Vzhľadom na vysokú náročnosť implementácie kvalitnej umelej inteligencie sa mnoho vývojárov vydalo ľahšou cestou a umelej inteligencii dali oproti normálnym hráčom výhody. Napríklad dáta o jednotkách, ktoré boli schované vo vojnovej hmle, prípadne vyšší počet surovín, jednotiek a tak ďalej. Takéto výhody sú v dnešnej dobe nežiaduce.

2.1 Vývoj RTS hier v 21. storočí

Podobne rýchlym tempom ako vývoj umelej inteligencie napredovala aj efektivita vývoja samotných hier. Štúdia ako Ensemble Studios, stojace za Age of Empires sériou, či Westwood Studios, neskôr EA Los Angeles, ktoré pracovalo na Command & Conquer sérii, postupne zdokonaľovali aj ich in-house herné enginy, na ktorých dané hry stáli. Tie umožnili dizajnérom rýchlo navrhnúť a naprogramovať hru bez toho, aby ich museli stavať úplne od základov.

Aj keď väčšina herných štúdií stále používa ich vlastné in-house enginy, v poslednej dobe je na trhu mnoho úspešných strategických hier postavených vo voľne dostupných herných enginoch, napríklad hra Shadow Tactics: Blades of the Shogun, ktorá bola spravená v Unity, či herná séria XCOM, reprezentujúca Unreal Engine.

2.2 Analýza existujúcich RTS hier a ich umelej inteligencie

V kontexte umelej inteligencie pre strategické hry prakticky existujú dva rôzne smery vývoja. Prvým je takzvaná reakčná umelá inteligencia [17], ktorá len reaguje na základe vstupných dát. Hovorí sa jej aj pravidlová umelá inteligencia, viz. kapitola 2.5. Druhou formou je umelá inteligencia s limitovanou pamäťou, viz. kapitola 2.4. Táto AI si ukladá predchádzajúce interakcie a na jej základe sa snaží zefektívniť jej budúce interakcie.

Vzhľadom na fakt, že cieľom väčšiny strategických hier je zabaviť hráča, dá sa povedať, že samotná efektivita umelej inteligencie nie je až tak dôležitá. Dokonca by sa dalo povedať, že príliš vysoká efektivita nie je žiaduca. Preto je pri strategických hrách obľúbeným riešením práve reakčná umelá inteligencia.

V praxi sa RTS hry delia na tie, v ktorých je hlavnou náplňou budovanie základne a manažovanie zdrojov, zatiaľ čo boj je len vedľajšou záležitosťou, a na tie, ktoré sa točia len okolo samotného boja. Typickým zástupcom prvého typu je hra Age of Empires II. Druhý typ dobre zastupuje menej známa hra Wargame: Red Dragon.

Age of Empires II: Definite Edition [11]

Tretia iterácia hry Age of Empires II prešla rozsiahlymi zmenami, okrem lepšej grafiky sa dočkala úplne novej umelej inteligencie. V tejto hre ide hlavne o:

- budovanie základne,
- zbieranie a riadenie zdrojov,
- vojnová hmla,
- boj na spôsob kameň-papier-nožnice.

Obr. 2.1: Age Of Empires 2 Definite Edition

Zatiaľ čo pôvodná umelá inteligencia bola známa najmä svojim podvádzaním, hlavne ignorovaním vojnovej hmly alebo získavaním extra zdrojov, nová je známa skvelým ovládaním jednotiek, správnym spravovaním surovín či širokou škálou stratégií, ktoré používa.

Tento veľký posun vpred bol možný aj vďaka rozsiahlej analýze tisícok hier hráčov rôznych úrovní, od začiatočníkov až po profesionálov. To autorom umožnilo implementovať nespočetné množstvo stratégií a taktík. Preto má Age of Empires II: DE jednu z najkvalitnejších pravidlových umelých inteligencií tejto doby [2].

Wargame: Red Dragon [10]

Eugen Systems, jeden z nováčikov na scéne, sa postaral o pomerne dobrú strategickú hru. V hre máte k dispozícii určitý počet bodov, za ktorý si nakupujete jednotky. Ďalšie body získavate ak máte pod vašou kontrolou určité strategické zóny. Táto hra je len o ovládaní jednotiek a vôbec v nej nefiguruje budovanie základne. Medzi jej hlavné črty patria:

- budovanie balíčku jednotiek,
- riadenie zdrojov,
- vojnová hmla,
- boj spôsobom kameň-papier-nožnice.

Obr. 2.2: Wargame Red Dragon

Umelú inteligenciu tejto hry by sme vedeli rozdeliť na 2 časti:

1. *Makro* - zadávanie rozkazov jednotkám, riadenie zdrojov vo forme nákupu ďalších jednotiek a tak ďalej,

2. *Mikro* - riadenie jednotlivých jednotiek po prijatí rozkazu.

Mikro funguje neustále, je jedno či príkazy prídu od umelej inteligencie alebo normálneho hráča. Slúži na automatické otáčanie jednotiek čelom k nepriateľovi, schovanie jednotiek v prípade, že sa v blízkosti nachádza krytie alebo na určovanie prioritných cieľov, napríklad tank uprednostní zasiahnuť iný tank aj keď sa v blízkosti nachádza nepriateľská pechota.

Pri makre to už je komplikovanejšie, napríklad umelá inteligencia pre odkrývanie vojnovej hmly nepoužíva prieskumné jednotky, keďže vojnovú hmlu úplne ignoruje a vidí aj jednotky, ktoré sú schované. Na každej mape je päť až osem zón, veľkosť zdrojov, ktoré sa z nich dajú získať sa z pravidla rovnajú náročnosti ich obsadenia. Samotné obsadenie je podmienené umiestnením veliteľského vozidla do danej zóny. V prípade, že sa v zóne nachádzajú veliteľské vozidlá obidvoch strán, sa zóna stáva neutrálnou. Z tohto je zrejmé, že ignorovanie vojnovej hmly je veľký nedostatok, pretože umelá inteligencia môže zneškodňovať dôležité veliteľské vozidlá bez toho, aby ich reálne videla. Ďalšou umelou výhodou, ktorou umelá inteligencia disponuje, je prístup k extra zdrojom. Ich objem sa určuje nastavením obťažnosti umelej inteligencie. Z toho je zrejmé, že umelá inteligencia podvádza, vzhľadom na fakt, že má prístup k dátam a surovinám, ku ktorým normálny hráč prístup nemá.

2.3 AlphaGo

Hra Go je známa ako najnáročnejšia klasická hra pre umelú inteligenciu. Napriek desaťročiam práce ani najvýkonnejšie Go programy nedosahovali výrazne lepšie výsledky ako prie-

merní kluboví hráči. Štandardné metódy umelej inteligencie, ktoré testujú všetky možné pohyby a pozície pomocou vyhľadávacieho stromu, nedokážu zvládnuť veľké množstvo možných ťahov ani vyhodnotiť silu každého možného stavu hracej plochy. Dvaja hráči, ktorí používajú biele alebo čierne kamene, striedavo umiestňujú svoje kamene na hraciu plochu. Cieľom je obkľúčiť a zajať súperove kamene alebo strategicky vytvárať územia. Po odohratí všetkých možných ťahov sa spočítajú kamene na doske a aj prázdne body. Vyhráva hráč s najvyšším počtom bodov.

Aj keď sa pravidlá môžu zdať jednoduché, Go je veľmi zložitá hra. Existuje úžasných 10^{170} možných konfigurácií dosiek - viac ako počet atómov v známom vesmíre. Vďaka tomu je hra Go 10^{100} krát zložitejšia ako šach. AlphaGo je počítačový program, ktorý kombinuje pokročilé vyhľadávacie stromy s hlbokými neurónovými sieťami. Tieto neurónové siete berú ako vstup popis hracej dosky Go a spracovávajú ho cez množstvo rôznych sieťových vrstiev obsahujúcich milióny spojení, ktoré sa podobajú neurónom.

Jedna neurónová sieť, ktorá sa označuje aj ako taktická sieť, vyberá ďalší ťah na hranie. Druhá neurónová sieť, označovaná ako hodnotová sieť, predpovedá víťaza hry. AlphaGo sa pomocou dát z obrovského počtu amatérskych hier naučilo ľudsky štýl hrania hry Go. Následne hralo proti iným verziám AlphaGo, čím sa postupne zlepšovalo.

Postupom času sa AlphaGo stalo čoraz silnejším a lepším v učení a rozhodovaní. Tento proces je známy ako spätnoväzbové učenie. V októbri 2015 odohralo AlphaGo svoj prvý zápas proti úradujúcemu trojnásobnému majstrovi Európy, pánovi Fan Hui. AlphaGo vyhralo vôbec prvú hru proti profesionálovi Go so skóre 5:0.

AlphaGo potom súperilo s legendárnym hráčom Go, pánom Lee Sedolom, víťazom 18 titulov majstra sveta, ktorý je všeobecne považovaný za najlepšieho hráča posledného desaťročia. Víťazstvo AlphaGo 4:1 v Soule v Južnej Kórei v marci 2016 sledovalo viac ako 200 miliónov ľudí na celom svete.

AlphaGo pokračovalo v porážke majstrov sveta Go v rôznych globálnych arénach a stalo sa pravdepodobne najlepším hráčom Go všetkých čias [6].

2.4 AlphaStar AI pre StarCraft II

StarCraft sa už dlhšie používa ako platforma pre vývoj umelých inteligencií. V roku 2010 prebehol prvý StarCraft turnaj, ktorého sa zúčastnili výlučne agenti umelej inteligencie. Agenti sú kontrolovaní pomocou BWAPI, Brood War Application Programming Interface, rozhranie ktoré bolo vyvinuté v roku 2009 pre hru StarCraft: Brood War, ktorého funkciou bolo dať agentom plnú kontrolu nad všetkou funkcionalitou tejto hry. [16]

AlphaStar je v tomto obore unikát, pre maximálnu efektivitu využíva hlbokú neurónovú sieť, ktorá je trénovaná z herných dát pomocou takzvaného učenia pod dohľadom a spätnoväzobného učenia. V roku 2019 sa utkala v súboji s hráčom Grzegorz "MaNa"Komincz, ktorý je považovaný za jedného z najlepších na svete. Stretnutie sa konalo v rámci profesionálnych súťažných podmienok bez akýchkoľvek obmedzení v prospech umelej inteligencie. AlphaStar vyhrala 5:0 [24], čo je pre umelú inteligenciu podobný míľnik, ako ked keď umelá inteligencia DeepBlue porazila Kasparova v roku 1987 [7].

2.5 Pravidlová umelá inteligencia

Môžeme povedať, že pravidlový systém pozostáva z troch základných komponentov: súboru pravidiel, databázy a interpreta pravidiel. V najzákladnejšej forme je pravidlo usporiadaná

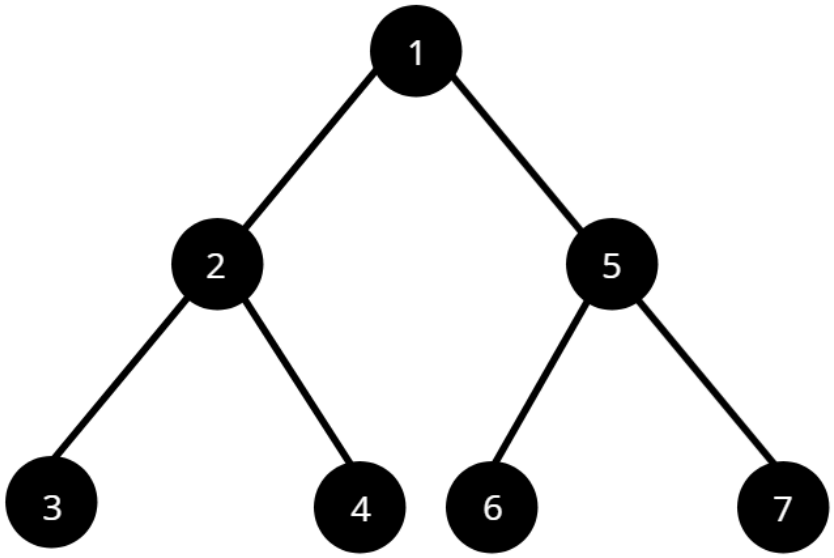
dvojica reťazcov symbolov, kde jeden je na ľavej (LHS) a druhý na pravej (RHS) strane. Sada pravidiel je lineárne usporiadaná množina a databáza je jednoducho súborom symbolov. Interpret v tomto jednoduchom dizajne skenuje ľavú stranu každého pravidla, dokým sa nenájde to, ktoré je zhodné s tým v databáze. V tomto momente sú zhodné symboly nahradené tými, ktoré sa nachádzajú na pravej strane pravidla a skenovanie buď pokračuje s nasledujúcim pravidlom alebo sa začne od znova [17].

2.6 Stromové vyhľadávacie algoritmy

Základom je stromová dátová štruktúra, ktorej koreňový uzol reprezentuje počiatočný stav hry. Vetvy reprezentujú akcie, ktoré agent spraví, aby sa dostal z jedného stavu hry do druhého a uzle reprezentujú samotné stavy hry. Strom sa rozvetvuje, pretože zvyčajne sa z jedného stavu hry dá prejsť do viacerých rozličných stavov. Stromové vyhľadávacie algoritmy sa líšia v závislosti od spôsobu prehľadávania vetví stromu.

Prehľadávanie do hĺbky - Depth First Search

Je to vyhľadávací algoritmus, ktorý začína v koreňovom uzle stromu. Postupne expanduje prvého nasledovníka, až dokým nenarazí na konečný uzol. Následne sa vráti k poslednému uzlu, ktorého vetva nebola preskúmaná a preskúma ju. Algoritmus to opakuje, až dokým sa nepreskúma celý strom[3].

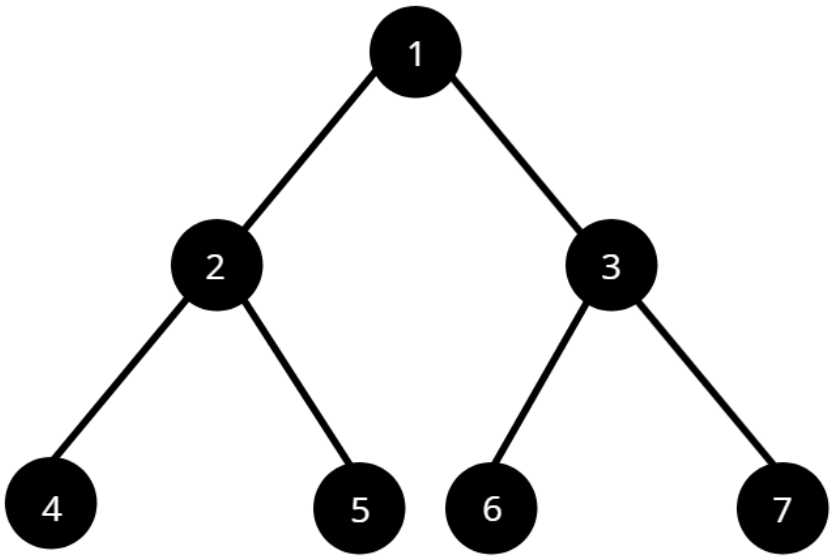


Obr. 2.3: Algoritmus Depth First Search

Hodnota každého uzlu stromu na obrázku 2.3 predstavuje poradie, v ktorom by sa expandovali, ak by sa použil algoritmus Depth First Search.

Prehľadávanie do šírky - Breadth First Search

Najskôr skontroluje všetkých susedov koreňového vrcholu, potom ich postupne expanduje a kontroluje susedné uzly susedných uzlov, tie potom postupne expanduje. Takto to pokračuje, až dokým sa nenájde hľadaný uzol alebo sa neprezrie celý vyhľadávací strom[5].

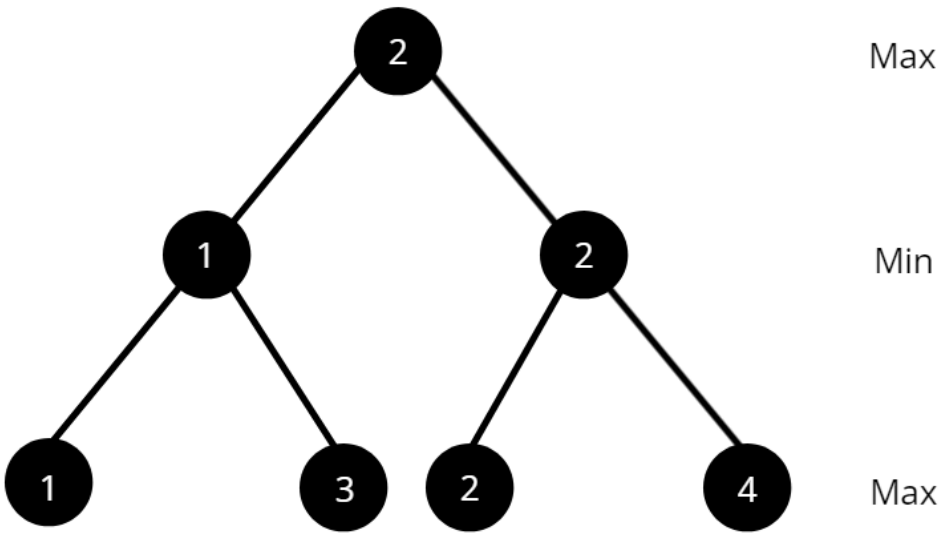


Obr. 2.4: Algoritmus Breadth First Search

Hodnota každého uzlu stromu na obrázku 2.4 predstavuje poradie, v ktorom by sa expandovali, ak by sa použil algoritmus Breadth First Search.

Minimax

Minimax je algoritmus, ktorý sa môže používať pri hrách pre dvoch hráčov. Stav hry sa znázorňuje číslom. Čím je dané číslo väčšie, tým je hráč 1 bližšie k výhre. Dá sa povedať, že hráč 1 sa snaží čo najviac zvýšiť skóre, zatiaľ čo hráč 2 znížiť. Dej hry sa dá modelovať vyhľadávacím stromom v prípade, že sa hráči po ťahu striedajú.



Obr. 2.5: Ilustrácia algoritmu minimax

Fungovanie algoritmu vysvetlím použitím obrázka 2.5. Každý uzol tohto stromu, okrem konečných uzlov, reprezentuje hodnotu ťahu, ktorý sa dá spraviť v danom momente. V tomto príklade sa dajú spraviť len dva ťahy, avšak normálne ich môže byť ďaleko viac. Koreňový uzol reprezentuje momentálny stav hry, v ktorom sa rozhoduje o nasledujúcom ťahu. Začína sa funkciou Max, keďže práve hráč, ktorý používa Minimax algoritmus chce získať čo najväčšie skóre. Namiesto toho, aby sa rozhodol len na základe nasledujúcich stavov hry, sa zoberú do úvahy aj potenciálne ťahy protivníka. Zavolá sa Min, a zistí sa aký ťah by zvolil protivník v prípade, že by hráč zvolil ľavý ťah a následne v prípade, že by zvolil pravý ťah, a to tak, že sa zavolá Max na ďalšie ťahy hráča. Z nich si Min vyberie možnosť s najmenšou hodnotou. Tým pádom sa Max a Min volajú navzájom, dokým sa nedostanú do koncového stavu. Ten by v ideálnom prípade reprezentoval stav hry, v ktorom sa hra skončí. Avšak kvôli obmedzeným zdrojom sa zvyčajne nastaví maximálna hĺbka, v tomto príklade sa nastavila na dvojku. Keď sa narazí na konečné uzly, tak sa už nedá použiť stratégia, ktorá sa použila doteraz. Namiesto toho sa musí priamo vypočítať hodnota, ktorá reprezentuje daný stav hry [20, 4].

Monte-Carlo metóda

Je to trieda algoritmov, ktorá sa používa na odhadnutie možných výsledkov nejakej neistej udalosti. Bola vyvinutá počas Druhej svetovej vojny za účelom zlepšenia rozhodovania pri neistých situáciách. Keďže sa v tejto metóde do veľkej miery používajú náhodné čísla, tak získala meno podľa známeho kasína, ktoré sa nachádza v Monaku. Monte Carlo získa množinu výsledkov na základe rozsahu vstupných hodnôt. Inými slovami, simulácia Monte Carlo vytvára model možných výsledkov využívaním rozdelenia pravdepodobnosti, akými sú napríklad rovnomerné alebo normálne rozdelenie pravdepodobnosti, pre každú premennú,

ktorej hodnota je otázna. Potom sa opakovane prepočítavajú výsledky, zakaždým s použitím inej sady náhodných čísel medzi minimálnou a maximálnou hodnotou. Pri typickom Monte Carlo experimente sa simulácia opakuje aj niekoľko tisíc krát, čím sa vyprodukuje veľké množstvo pravdepodobných výsledkov. Postup pri Monte Carlo metóde:

1. Vytvorenie prediktívneho modelu, identifikácia závislých premenných, ktoré sa predpovedajú, ale aj nezávislých premenných, ktoré sú známe aj ako vstupné, rizikové alebo prediktorové premenné, ktoré budú riadiť predikciu.

2. Špecifikácia rozdelenia pravdepodobnosti nezávislých premenných. Používajú sa historické údaje alebo subjektívny úsudok analytika na definovanie rozsahu hodnôt a samotné rozdelenie pravdepodobnosti.

3. Opakované spúšťanie simulácií a generovanie náhodných hodnôt nezávislých premenných. To sa robí dovtedy, kým sa nezíska dostatok výsledkov na vytvorenie reprezentatívnej vzorky takmer nekonečného počtu možných kombinácií [9].

2.7 Lanchesterové pravidlá

V roku 1916 britský matematik a inžinier Frederick William Lanchester navrhol sériu diferenciálnych rovníc, pomocou ktorých sa dá demonštrovať dynamika strát medzi dvoma bojujícimi stranami. Medzi najznámejšie patrí takzvané Lanchesterové lineárne pravidlo. Toto pravidlo sa dá uplatniť pri simulácií starovekých bitiev, ktoré sa odohrali v dobe, keď sa bojovalo s mečmi a lukmi. Druhým známym pravidlom je Lanchester Square Law, ktoré sa dá použiť aj pri modelovaní moderných bitiev, v ktorých hrajú rolu strelné zbrane.

Obr. 2.6: Falanx z hry Total War: Rome 2

2.7.1 Lanchesterovo lineárne pravidlo - Lanchester linear law

Dá sa povedať, že v starovekých bitvách, v dobe, keď medzi sebou bojovali falanxy vojakov, ktorí boli vyzbrojení kopiami, mohol jeden vojak v danom momente bojovať iba s jedným iným vojakom. V prípade, že by si všetci vojaci boli rovní a teda každý vojak by zabil jedného iného vojaka a následne bol zabitý nasledujúcim vojakom, tak zostavajúci počet vojakov na konci bitvy je jednoducho rozdiel medzi početnejšou a menej početnejšou bojujúcou stranou [15]. Vzorec pre Lanchesterové lineárne pravidlo je:

$$Am = b * N \qquad Bm = a * N$$

`Am` a `Bm` sú straty jednotlivých strán, `a` je bojová efektivita strany A, `b` je bojová efektivita strany B a `N` je počet vojakov, ktorí bojujú v prvej línii. Napríklad, ak by sa bojovalo na fronte, na ktorú sa zmestí falanx, ktorý má šírku 10 vojakov a každý vojak by za 1 jednotku času dokázal zneškodniť jedného vojaka, tak by straty za jednu jednotku času predstavovali 10 vojakov.

2.7.2 Lanchesterové mocninové pravidlo - Lanchester square law

Moderné strelné zbrane a delostrelectvo umožňujú útočiť z diaľky na viacero cieľov naraz. Podľa Lanchestera bojová sila takejto armády nie je úmerná počtu jednotiek, ale druhej mocnine počtu jednotiek. Vzorec pre Lanchesterové mocninové pravidlo je:

$$Bf = \sqrt{B0^2 - \left(\frac{a}{b}\right) * A0} \qquad Af = \sqrt{A0^2 - \left(\frac{b}{a}\right) * B0}$$

`Af` a `Bf` je výsledný počet vojakov, `A0` a `B0` je počiatočný počet vojakov, `a` je bojová efektivita strany A a `b` je bojová efektivita strany B [22]. Pomocou tohto pravidla sa úspešne modelovalo množstvo známych bitiev, napríklad bitva o Iwodžimu [18].

Kapitola 3

Návrh strategickej hry v Unity

Pri návrhu hry som sa inšpiroval vojenskými strategickými hrami, akými sú napríklad Wargame alebo Combat Mission. Hlavnou náplňou je ovládanie vojenských jednotiek, ktoré boli hráčovi zverené na začiatku hry a využívanie ich k obsadzovaniu územia a k ničeniu jednotiek nepriateľa. V hre sa preto nenachádza takzvané budovanie základne, ani budovanie jednotiek. Ako oponent slúži umelá inteligencia, ktorá využíva metódu Monte Carlo. Umelá inteligencia musí hrať férovo, to znamená, že nemôže podvádzať napríklad tým, že by videla cez vojnovú hmlu alebo tým, že by mala väčšie množstvo jednotiek. Dôležitou súčasťou hry sú aj objekty, ktoré zaručujú výhru strane, ktorá ich bude ovládať. Najväčšou výzvou bolo kvalitne navrhnúť Monte Carlo algoritmus tak, aby vyberal vhodnú stratégiu.

3.1 Pravidlá hry

Na začiatku majú obe strany pod svojou kontrolou rovnaký počet jednotiek. Body sa získavajú za obsadenie strategických objektov, akými sú napríklad budovy. Body sa tiež získavaju zneškodnením jednotiek nepriateľa.

3.1.1 Herné pole

Pre vhodné otestovanie umelej inteligencie by malo herné pole čo najviac pripomínať reálnu bojovú zónu. Preto pozostáva z členitého terénu, rôznej vegetácie, budov a telekomunikačnej sieti.

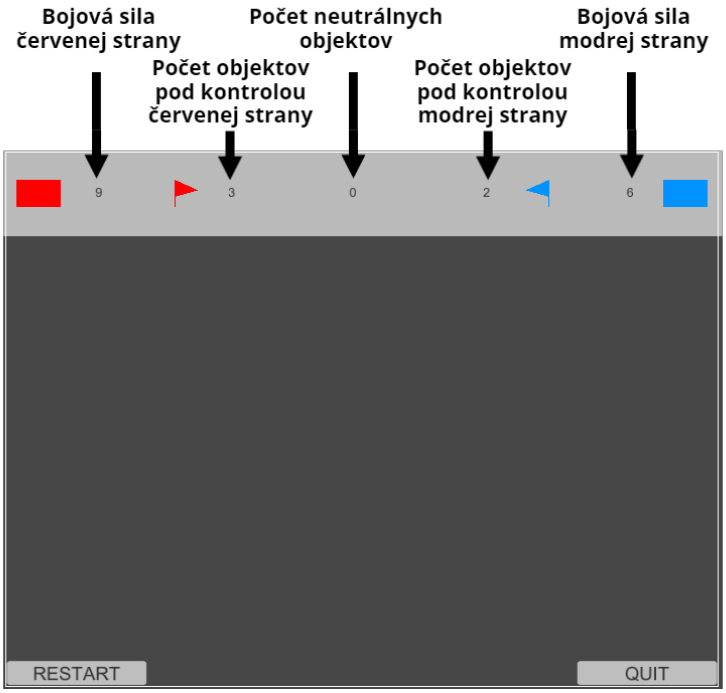
Kopce a iné vyvýšené miesta majú priaznivý vplyv na presnosť a zrak jednotiek, ktoré sa na nich nachádzajú. Preto sú ideálnym miestom pre obranné postavenia. Lesy a budovy vedia ukryť jednotky pred zrakom nepriateľa, znižujú ich rýchlosť a zvyšujú ich schopnosť prežitia, v prípade budov zásadným spôsobom.

3.1.2 Vojenské jednotky

Vojenské jednotky sú rozdelené na dva druhy: pešiaci a tanky. Pešiak je vybavený útočnou puškou. Tank má tankový kanón. Každá jednotka má zdravie, morálku, víziu a rozmer. Strelec má maximálne 100 zdravia, tank 500. Zdravie sa znižuje v prípade, ak je jednotka zasiahnutá projektilom. V prípade, že zdravie klesne na 0, tak je jednotka zneškodnená.
Morálka má maximálnu hodnotu 100 a minimálnu 0. Avšak znižuje sa, ak popri jednotke preletí nepriateľský projektil, alebo v prípade vysokých strát na jeho strane. Ak sa morálka zníži na 0, tak po dobu 5 až 10 sekúnd je jednotka v takzvanej panike. Efekt tohto stavu

závisí od prostredia. Ak je jednotka v kryte, napríklad v budove, tak zostane na svojej pozícii, ale prestane bojovať. V prípade, že nie je v kryte, tak jednotka prestane bojovať a zároveň začne svojvoľne ustupovať. Vízia určuje do akej vzdialenosti jednotka spozoruje nepriateľa. Rozmer znázorňuje časť jednotky, ktorá môže byť zasiahnutá nepriateľskou paľbou. Štandardne sa tento atribút rovná 1, znižuje sa v prípade, ak sa jednotka nachádza v alebo pri nejakom kryte, akými sú napríklad stromy alebo budovy. Rozmer má efekt na percentuálnu úspešnosť zásahu (presnosť × rozmer).

3.1.3 Používateľské rozhranie



Obr. 3.1: Návrh používatelského rozhrania

Na obrázku 3.1 je nakreslený návrh používateľského rozhrania. Na začiatku sú všetky objekty neutrálne, a teda nie sú pod kontrolou žiadnej strany. Potom, ako do daného strategického objektu vkročia jednotky jednej strany, tak sa objekt považuje za obsadený danou stranou až dokým do neho nevstúpia jednotky druhej strany. Bojová sila predstavuje celkovú bojovú silu danej strany. Každý strelec zvyšuje bojovú silu o 1, tank zvyšuje bojovú silu o 5 bodov. Po zneškodnení jednotky sa z celkovej bojovej sily odpočíta hodnota danej jednotky.

3.2 Umelá inteligencia

Umelá inteligencia sa skladá z 3 častí:

- ovládača každej jednotky 3.2.1,
- hlavného ovládača 3.2.2, ktorý má na starosť celú vojnovú frakciu,
- a Monte Carlo plánovača 3.2.3, ktorého úlohou je tvorba bojových plánov.

3.2.1 Ovládač jednotky

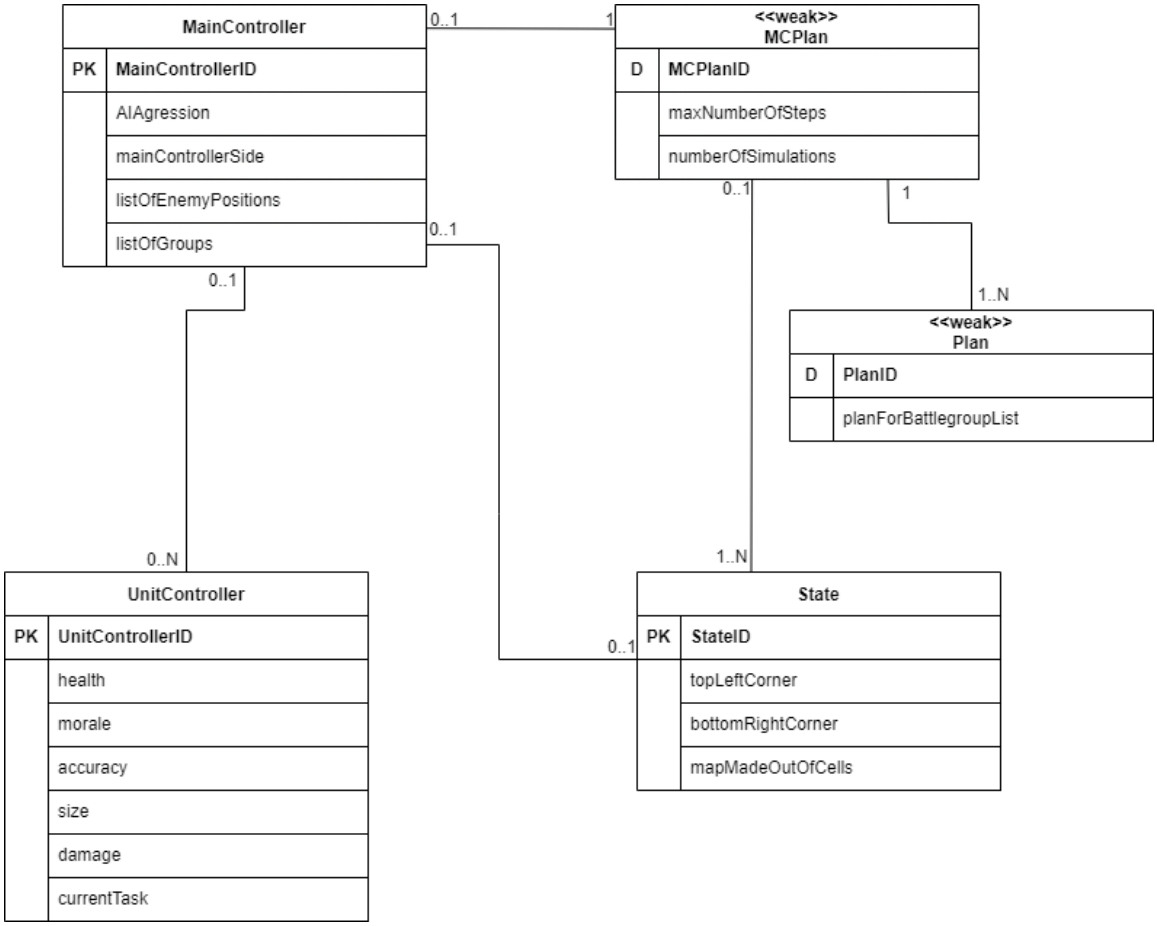
Má na starosť jednotlivé jednotky. Jeho úlohou je implementácia príkazov, ktoré dostane od hlavného ovládača. Konkrétne sa jedná o pohyb jednotky, o nahlasovanie pozícií nepriateľa, o streľbu, kontrolu zdravia, vyhodnocovanie krytia a samotnú víziu.

3.2.2 Hlavný ovládač

Jeho úlohou je vyhodnocovanie situácie na bojisku z dát, ktoré mu poskytnú jednotky. Napríklad uchováva údaje o zistených pozícií nepriateľa, status dôležitých objektov a tak ďalej. Taktiež organizuje svoje jednotky do rôzne veľkých bojových skupín, ktorým bude dávať bojové úlohy na základe aktuálneho bojového plánu. Ten získa pomocou triedy `MCPlan`, v ktorej bude implementované generovanie, testovanie a výber bojového plánu pomocou Monte Carlo algoritmu.

3.2.3 Monte Carlo plánovač

Pri návrhu Monte Carlo plánovača som využil poznatky z dokumentu: Monte Carlo Planning in RTS Games od autorov Michaela Chunga, Michaela Buroa, a Jonathana Schaeffera [23]. Autori prišli na to, že je dôležité vytvoriť abstraktnú verziu hry, ktorá sa využije pri simuláciách, vďaka čomu sa celý proces urýchli a tým sa môže zvýšiť počet simulácií. Podobne to bude aj v mojom Monte Carlo plánovači, ktorý získa dáta o hre priamo od hlavného ovládača a na ich základe vytvorí abstraktnú verziu hry, ktorá sa podobá šachovnici. Šachovnica sa bude skladať z buniek, ktoré uchovávajú všetky dôležité dáta pre daný región mapy. Následne sa vytvorí plán pre hlavný ovládač a ten sa otestuje v sérii simulovaných bojov proti rôznym náhodným plánom protivníka. Po vygenerovaní a odsimulovaní dostatočného množstva plánov sa vyberie ten najefektívnejší a ten sa pošle hlavnému ovládaču, ktorý ho bude implementovať až dokým nepožiada o nový aktuálny plán.



Obr. 3.2: ER diagram umelej inteligencie

Pri ER diagrame na obrázku 3.2, `MainController` znázorňuje hlavný ovládač, ktorého skutočný názov závisí od strany, ktorú ovláda (`RedController` pre červenú stranu, `BlueController` pre modrú).

Kapitola 4

Implementácia

Pre zníženie grafickej náročnosti a možnosti simulácie veľkého množstva jednotiek som sa rozhodol pre vytvorenie 2D hry. Kamera je otočená smerom zhora dole.

4.1 Herný engine

Unity, tiež známe aj ako Unity3D, je jeden z najpopulárnejších enginov na svete. Viac ako 50% mobilných, počítačových a konzolových hier bolo vytvorených v Unity. 2.5 miliardy ludí každý mesiac využíva služby alebo konzumuje obsah vytvorený pomocou Unity [1]. Zatiaľ čo prvé verzie podporovali len OSX, postupne sa ich podpora rožšírila na prakticky všetky platformy, od rôznych herných konzol, cez mobilné telefóny až po počítače. Okrem vývoja hier sa používa najmä vo filmovom priemysle. Tento engine je obľúbený kvôli veľmi dobrej dostupnosti, flexibilite a skvelej funkcionalite. Unity slúži aj ako integrované vývojové prostredie. Aj keď samotné Unity beží pomocou C++ [19], tak pri programovaní skriptov sa používa C#.

Životný cyklus skriptu v Unity

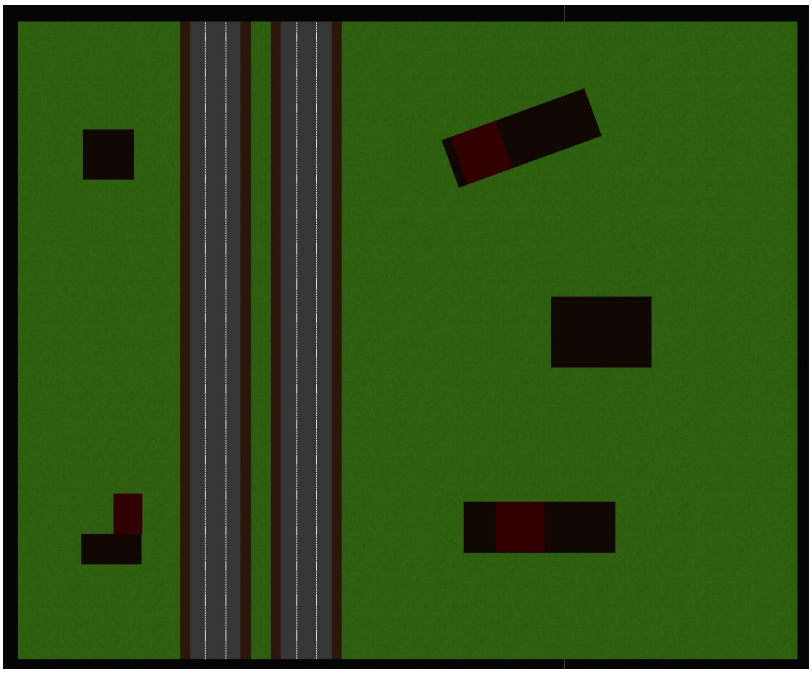
Pri upravovaní stavu hry v nejakom pravidelnom intervale sa najbežnejšie používa funkcia `Update()` alebo nejaká jej alternatíva.

- `Awake()`: Táto funkcia sa volá pred akoukoľvek funkciou `Start()`, prípadne hneď po inicializácii objektu typu `Prefab`.
- `Start()`: Volá sa pred prvým volaním funkcie `Update()`.
- `Update()`: Volá sa raz za snímok.
- `FixedUpdate()`: Zvyčajne sa volá častejšie než `Update()` a `LateUpdate()`. Frekvencia volania je nezávislá od počtu snímkou za sekundu, počas jedného snímku sa môže zavolať aj niekoľko krát. Všetky fyzikálne výpočty prebiehajú hneď po `FixedUpdate()`.
- `LateUpdate()`: Vola sa hneď po dokončení funkcie `Update()`. Všetky výpočty vykonané v `Update()` budú dokončené, v momente, keď sa spustí `LateUpdate()`. Využíva sa napríklad pri nastavení pozície kamery, ktorá sleduje nejaký pohyblivý herný objekt.
- `OnTriggerEnter()`: Keď sa herný objekt zrazí s iným herným objektom a oba objekty majú kolíznu geometriu, Unity zavolá túto funkciu.

Pri načítaní scény sa pri každom objekte, ktorý sa nachádza v scéne a je potomkom základnej Unity triedy MonoBehavior, volá funkcia `Awake()`. Neskôr sa pred prvým volaním funkcie `Update()` volá funkcia `Start()` [12].

4.2 Mapa

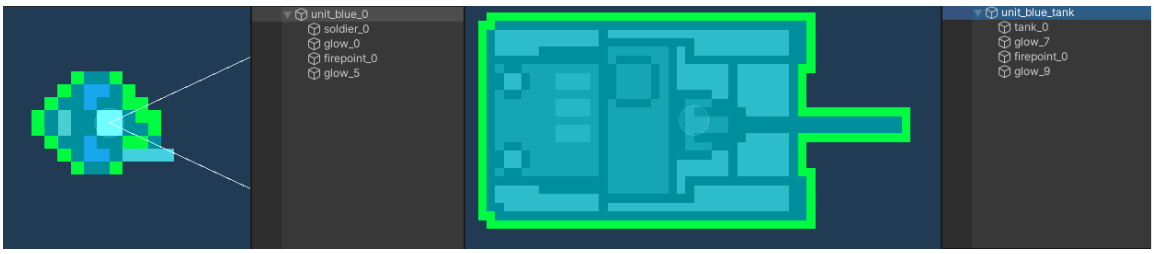
Mapa je vidieť na obrázku 4.1. Skladá sa z trávnatej plochy, diaľnice a piatich strategických objektov, ktoré sa snažia obe strany obsadiť. Každý strategický objekt pozostáva z jedného alebo viacerých dvojrozmerných obrázkov a kolíznej geometrie, vďaka ktorej sa s ním môže okolie interagovať.



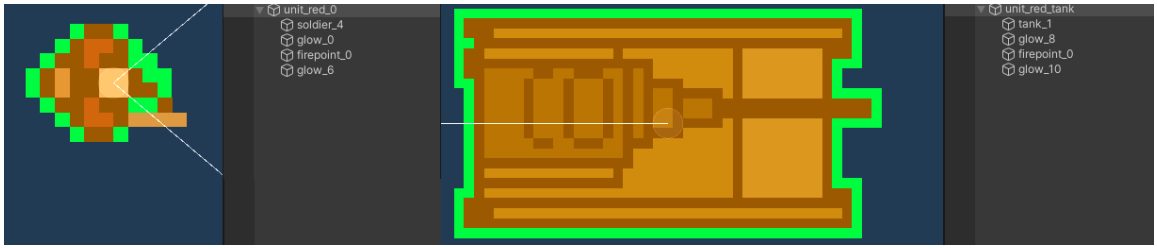
Obr. 4.1: Bojové pole, na ktorom jednotky vedú bojovú činnosť

4.3 Jednotky

Obe frakcie majú 2 druhy jednotiek. Strelec je vybavený útočnou puškou a je najpočetnejšou jednotkou na bojisku. Tank je vybavený tankovým kanónom, má väčší dostrel a poškodenie, než normálny strelec.



Obr. 4.2: Modré prefabrikáty



Obr. 4.3: Červené prefabrikáty

Každý prefabrikát jednotky (viď obrázky 4.2 a 4.3) sa okrem skriptov skladá aj zo 4 objektov. Prvý objekt uchováva obrázok jednotky, druhý uchováva bledo modrý/oranžový filter, ktorý sa aktivuje v prípade, že je jednotka v kryte. Tretí objekt, je `Firepoint`, bod, z ktorého budú vychádzať projektily. Posledným objektom je bledo zelený obrys jednotky, ktorý sa aktivuje v prípade, že je jednotka označená myšou.

4.4 Umelá inteligencia

Umelá inteligencia sa skladá z množstva jednoduchých akcií akými sú napríklad pohyb na určité miesto či streľba na inú jednotku. Nad nimi je hlavný ovládač, ktorý pomocou metódy Monte Carlo určuje, aké akcie sa majú v akom poradí vykonať.

4.4.1 Vízia

Víziu jednotlivých jednotiek má na starosť skript `VisionController.cs`. Tento skript má niekoľko premenných:

- `public LayerMask targetMask`: Maska, ktorá je priradená jednotkám nepriateľa.
- `public LayerMask obstacleMask`: Maska, ktorá je priradená prekážkam, cez ktoré jednotka nevidí. Napríklad steny alebo skaly.
- `public float viewRadius`: Určuje maximálnu vzdialenosť na ktorú jednotka môže spozorovať iné objekty.
- `public float viewAngle`: Znázorňuje šírku zorného poľa jednotky.
- `public List<Transform> visibleTargets`: Zoznam nepriateľských jednotiek, ktoré sú na dohľad.
- `public float visionResolution`: Udáva hustotu vysielaných lúčov (`Raycast2D`), ktoré hľadajú a interaktujú s objektami.
- `public List<Transform> globalHitList`: Zoznam objektov, do ktorých narazili vysielané lúče.

`Start()` volá funkciu `StartCoroutine("SearchForTargets", .2f)` . Úlohou tejto vstavanej funkcie je vytvoriť korutinu, ktorá bude fungovať naprieč viacerými snímkami. Jej úlohou je každých 200 milisekúnd volať funkciu `FindVisibleTargets()`. Na začiatku tejto funkcie sa pomocou `.Clear()` vyčistí zoznam `visibleTargets`. Následne sa do lokálnej premennej `targetsInViewRadius` načítajú pomocou funkcie

`Physics2D.OverlapCircleAll()` všetky objekty, ktoré majú potrebnú masku a sú dostatočne blízko jednotky, bez ohľadu na jej zorné pole. Nasleduje cyklus v ktorom sa kontrolujú nájdené objekty. Vypočíta sa akým smerom je nájdený objekt a to tak, že od pozície nájdeného objektu sa odpočíta pozícia jednotky a následne sa tento vektor normalizuje. Pozícia jednotky a smer k nájdeného objektu sa vložia ako argumenty do funkcie `Vector3.Angle()` a zistí sa, či je výsledný uhol vo vnútri zorného poľa jednotky. Ak áno, tak sa cez `Vector3.Distance()` zistí vzdialenosť medzi jednotkou a objektom. Následne sa zavolá `Physics2D.Raycast()`, ako argumenty sa použijú: pozícia jednotky, smer k nájdenému objektu, vzdialenosť k nájdenému objektu a `obstacleMask`. Cieľom je zistiť, či sa v ceste nenachádza prekážka. Ak nie tak sa objekt pridá do zoznamu `visibleTargets`.

V druhej časti skriptu `VisionController.cs` sa pracuje s nájdenými prekážkami. Začína sa v `Update()`, kde sa volá funkcia `DrawVision()`. V nej sa najskôr vytvorí nový zoznam `localHitList`, ktorý uchováva nájdené objekty. Vynásobením `viewAngle` a `visionResolution` sa získa počet vysielaných lúčov, `rayCount`, a potom veľkosť kroku v úhloch, `stepAngleSize` tým, že sa `viewAngle` vydelí `rayCount`-om. Následne sa cez cyklus vykresľujú jednotlivé lúče. Ako prvé sa musí získať uhol ktorým bude lúč smerovať: `angle = transform.eulerAngles.z - viewAngle / 2 + stepAngleSize * i - 90;` `transform.eulerAngles.z` je smer, ktorým je jednotka otočená, `i` je získané z `for` cyklu a určuje, koľký lúč v poradí sa vykresľuje. Následne sa získa cieľová destinácia lúču tak, že k pozícii ciela sa pripočíta smer k cieľu, ktorý sa vypočítal pomocou `angle`, to celé sa vynásobí hodnotou `viewRadius` a na záver sa odpočíta pozícia jednotky. Zavolá sa funkcia `Physics2D.Raycast()`. Raycast je prakticky laserový lúč, ktorý je vyžarovaní určitým smerom z bodu v 2D priestore. Každý objekt prichádzajúci do kontaktu s lúčom môže byť detekovaný a nahlásený. Táto funkcia vracia objekt `RaycastHit` s odkazom na zasiahnutú kolíznu geometriu, ktorá je zasiahnutá lúčom. Vďaka použitiu argumentu `LayerMask` sa vracajú len objekty, ktoré majú danú vrstvu.

4.4.2 Pohyb

Jednotka sa da do pohybu volaním funkcie `MoveTo`, ktorá sa nachádza v ovládači danej jednotky. Táto funkcia berie ako argument 3D vektor, ktorý znázorňuje pozíciu na mape, kam sa má jednotka premiestniť. Funkcia `MoveTo()` danú pozíciu zapíše do globálnej premennej `movePosition`. V `Update()` sa počíta smer pohybu a to tak, že sa od `movePosition` odpočíta pozícia jednotky a výsledok sa normalizuje. V prípade, že sa jednotka presunie na danú pozíciu, tak sa `moveDir` zmení na `Vector3(0, 0, 0)`. Samotný pohyb sa uskutočňuje až vo funkcii `FixedUpdate()` pomocou komponentu `Rigidbody2D`, ktorý je súčasťou každej jednotky a to tak, že sa smer pohybu vynásobí rýchlosťou jednotky a výsledok sa uloží do `rigidbody2D.velocity`. Tým padom ak je v `moveDir` uložení `Vector3(0, 0, 0)` tak jednotka stojí na mieste.

4.4.3 Streľba

Táto akcia sa volá funkciou `Shoot()`. V nej sa najskôr pomocou `Instantiate(bulletPrefab, firePoint.position, firePoint.rotation)` vytvorí nový objekt. `BulletPrefab` je prefabrikát tvorený z grafiky reprezentujúcej náboj, komponentov RigidBody2D, BoxCollider2D, skriptu `BulletController` a objektu `BulletMoraleController`, ktorý má na starosť znižovanie morálky nepriateľských jednotiek, okolo ktorých preletí náboj. Náboj bude interagovať s inými objektami pomocou funkcie `OnTriggerEnter2D()`. Keďže môže väčšina jednotiek strieľať napríklad z budov alebo iných krytov, tak je po-

trebné ošetriť aby náboj nebol ovplyvnený objektami nad ktorými vznikol. To sa docieli získaním zoznamu objektov v tesnej blízkosti náboja pomocou funkcie `Physics2D.OverlapCircleAll()`. Pomocou cyklu sa zruší kolízia medzí každým nájdeným objektom a nábojom použitím funkcie `Physics2D.IgnoreCollision()`. Po ošetrení kolízii sa funkciami `SetHitChance()`, `SetDamage()` a `SetSide()` nastaví presnosť, poškodenie a strana náboja, čím sa zaručí, že náboj nepoškodí priateľské jednotky. Nakoniec sa náboj dá do pohybu aplikovaním funkcie `AddForce()` na `RigidBody2D`, ktorý je komponentom náboja.

V skripte `BulletController.cs` sa pri vzniku náboja volá funkcia `Awake()`, v ktorej sa nastaví samo deštrukčný mechanizmus `Destroy(gameObject)` na 4 sekundy a to z preventívneho dôvodu, aby sa náboj sám zničil v prípade, že sa netrafí do akékoľvek objektu. Druhou dôležitou funkciou je `OnTriggerEnter2D`, ktorej prvý parameter je objekt typu `Collider2D`, ktorý patrí objektu do ktorého náboj narazil. V tejto funkcii sa najskôr zistí typ objektu do ktorého náboj narazil. Ak sa nejedná o jednotku, tak sa náboj zničí pomocou funkcie `Destroy()`. V opačnom prípade sa vygeneruje náhodné číslo od 0 do 100. Toto číslo sa vynásobí veľkosťou cieľa, `size`, ktorá znázorňuje plochu jednotky, ktorá sa dá zasiahnuť. Napríklad ak je jednotka schovaná v budove, tak je táto plocha menšia než keď sa jednotka pohybuje cez otvorené priestranstvá. Ak je výsledné číslo menšie než presnosť jednotky, ktorá strieľala, tak sa cez skript prepojení so zasiahnutým objektom volá funkcia `AddDamage()`, ktorá zníži život a morálku zasiahnutej jednotky a na záver sa náboj zničí cez `Destroy()`.

`Shoot()` sa používa najmä vo funkcii `EngageHostilies()`, ktorej cieľom je bojovať s jednotkami protivníka, ktoré sú viditeľné samotnou jednotkou. V nej sa najskôr získa zoznam nepriateľských jednotiek, na ktoré môže jednotka zaútočiť. Následne sa náhodne vyberie 1 jednotka, ktorá sa uloží do premennej `currentTarget`. Získa sa smer k nepriateľskej jednotke a následne sa vypočíta uhol, ktorý sa uloží do `rb.rotation` (rb, Rigidbody, je komponent, ktorý ma na starosť interakciu objektu s fyzikou Unity enginu). Uložením daného uhla do `rb.rotation` sa naša jednotka otočí smerom k nepriateľskej jednotke. Následne sa vytvorí korutina, v ktorej sa pravidelne volá funkcia `Shoot()`. Doba medzi výstrelmi závisí od rýchlosti paľby. Aby sa strielalo v dávkach, ukladá sa počet výstrelov do premennej `currentBurstCounter`, zatial čo počet nábojov v aktuálnom zásobníku sa ukladá do `currentMagazineCounter`. V prípade, že sa `currentBurstCounter` rovná nastavenej dávke, tak sa korutina, ktorá volá `Shoot()`, zruší, zavolá sa korutina, ktorá chvílu čaká, čo má reprezentovať čas medzi jednotlivými dávkami a následne sa resetuje obsah premennej `currentBurstCounter` a vytvorí korutinu pre funkcia `Shoot()`. Podobne to funguje so zásobníkom.

4.4.4 Krytie

`UnitController` (konkrétny názov skriptu závisí od typu jednotky a na tom, na akej je strane) priebežne kontroluje stav jednotky, jej prostredie a kde sa nachádza. Jednou z úloh tohto skriptu je aj kontrolovať, či sa jednotka kryje v budove. To sa dosahuje tak, že sa vo `FixedUpdate()` kontrolujú objekty v tesnej blízkosti jednotky pomocou funkcie `Physics2D.OverlapCircleAll()`. Ak sa jedná o druh krytia, tak sa zavolá funkcia `SetInCover()`. Ak sa žiadny taký objekt nenájde, tak sa volá funkcia `SetOutOfCover()`. `SetInCover()` volá funkciu `SetCoverVisible()`, ktorá upraví vzhľad jednotky pridaním modrej alebo červenej farby za účelom rozlíšenia jednotiek, ktoré su v kryte od tých ktoré nie sú. Následne sa v `SetInCover()` upravia vlastnosti jednotky, aby bola efektívnejšia

v boji. Vo funkcii `SetOutOfCover()` sa všetky tieto premenné vrátia do pôvodnej hodnoty a taktiež sa odstráni farebný filter.

4.4.5 Panika

Súčasťou prefabrikátu `BulletPrefab` je aj herný objekt`moraleController`, ktorý sa skladá z 2D kolíznej geometrie, ktorá je približne 20 krát väčšia než samotný náboj a skriptu `BulletMoraleController`. Pomocou funkcie `onTriggerEnter2D` získava kolízne geometrie nepriateľských jednotiek, okolo ktorých preletí vystrelený náboj. Následne pomocou `getComponent<>` získa ovládač nepriateľskej jednotky a zavolá funkciu `SetMoraleDamage()`, pomocou ktorej sa zníži morálka jednotky. Úprava morálky sa uskutočnuje v ovládači každej jednotky. Po vytvorení jednotky, vo funkcii `Start()` sa nastaví `InvokeRepeating()`(funkcia, cez ktorú sa môže v určitom časovom intervale pravidelne volať iná funkcia) tak, aby sa každých 5 sekund zavolala funkcia `IncreaseMorale()`, ktorá, v prípade, že jednotka nemá 100% morálku, zvýši morálku o 10. Na záver sa vo funkcii `FixedUpdate()` kontroluje morálka. Ak je na nule, tak sa preruší akýkoľvek boj. Skontroluje sa, či je jednotka schovaná v kryte alebo je na otvorenom priestranstve. Ak nie je v kryte, tak sa otočí o 180 stupňov a začne ustupovať. Jednotka sa zbaví paniky až v momente, keď jej morálka stúpne aspoň na 50%.

4.4.6 Hlavný ovládač

Skript `RedController.cs` (prípadne `BlueController.cs` pre modrú stranu) je hlavným ovládačom umelej inteligencie. Pri spustení hry sa volá funkcia `Start()`. V nej sa pomocou funkcie `Physics2D.OverlapCircleAll` získa zoznam jednotiek danej strany, ktorý sa v ďalšom kroku využije pri tvorbe bojových skupín. Vytvorí sa ďalší zoznam, ktorý bude uchovávať bojové skupiny jednotiek. Zoznam jednotiek, ktorý sa získal v predošlom kroku, sa dá do cyklu, v ktorom sa pracuje s jednotlivými jednotkami. Každá jednotka skontroluje, či sa už v zozname bojových skupín nachádza aspoň jedna skupina. Ak áno, tak sa porovná pozícia jednotky s pozíciami skupín. Ak je jednotka dostatočne blízko a zároveň skupina nie je moc veľká, tak sa jednotka pridá do skupiny. V opačnom prípade jednotka vytvorí novú skupinu, ktorá sa pridá do zoznamu bojových skupín. Následne sa k bojové skupiny rozšíria tak, že sa vytvorí nová inštancia triedy `customGroup`, ktorá berie ako argumenty samotnú bojovú skupinu, jej druh (môže sa jednať buď o prieskumný druh skupiny, úderný druh skupiny alebo o tank), pozíciu na mape a stav skupiny. Existujú dva stavy, buď je skupina aktívna a to znamená, že vykonáva nejakú úlohu, alebo že je pasívna a teda čaká na ďalšiu bojová úlohu. Bojové skupiny sa uložia do verejnej premennej `listOfGroups`, ktorá sa bude využívať aj v iných skriptoch. Po vytvorení všetkých počiatočných skupín a po zadaní počiatočných úloh sa musia dané úlohy poslať aj samotným jednotkám. Toho sa dosiahne tak, že pomocou funkcie `GetComponent<RiflemanRedController>()` získame ovládače jednotiek a následne im pošleme všetky dôležité údaje pomocou funkcií `SetTastType()`, `SetGroupID()`, `SetMovePosition()` a `SetState()`. Na konci funkcie `Start()` sa pomocou novej inštancie triedy `State`, viz sekcia 4.4.7, vyhodnotí súčasný stav hry a vytvorí sa abstraktná verzia hry, ktorá má podobu šachovnice pozostávajúcej z buniek. Následne sa pomocou získaného stavu vytvorí nový `MCPlan`, viz. sekcia 4.4.9, ktorý sa uloží a začne vykonávať.

Najdôležitejšia časť hlavného ovládača sa nachádza vo funkcii `FixedUpdate`. V tejto funkcii sa kontroluje stav bojových skupín. Ak bojová skupina nie je aktívna, tak sa skontroluje súčasný plán, v ktorom je tiež uložený zoznam bojových skupín a k ním je priradený zoznam úloh. V prípade, že bojová skupina nie je v aktívnom stave a daná skupina stále

nedokončila všetky svoje úlohy, tak sa jej priradí bojová úloha, ktorá je ďalšia v poradí. Ak dokončila všetky úlohy, ktoré jej boli naplánované, tak sa vygeneruje nový `MCPlan`. Avšak ak je skupina aktívna, tak sa overí, či nedokončila aktuálnu bojová úlohu a to tak, že sa porovná jej aktuálna pozícia s pozíciou, kam sa mala dostať aby splnila bojovú úlohu. Ak sa tam bojová skupina nachádza, tak sa overí typ úlohy. Ak sa jedná o útok alebo o prieskum, tak sa úloha vyhodnotí ako dokončená a stav skupiny sa zmení na `Inaction`. Ak sa jedná o obranu alebo o rezervu, tak sa spustí časovač a až po jeho uplynutí sa stav skupiny zmení na `Inaction`.

4.4.7 Stav

Hlavnou náplňou triedy `State` je využitie informácii, ktoré má umelá inteligencia k dispozícii k vyhodnoteniu situácie na bojisku a následné vytvorenie abstraktnej verzie bojiska, ktorá sa využije pri simulácii plánov. Na úvod sa v konštruktore triedy, pomocou pozícii rohov mapy a zadaného počtu buniek, vypočíta šírka a výška jednej bunky. Následne sa začnú vytvárať samotné bunky. Tie sa pridajú do zoznamu buniek. Každý zoznam buniek reprezentuje stĺpec buniek. Zoznam sa potom pridá do ďalšieho zoznamu, čím vznikne zoznam zoznamov buniek (`List<List<Cell>>`) a následne sa pokračuje na ďalší stĺpec. Každá bunka uchováva informácie o tom, koľko jednotiek sa nachádza v danej bunke, aké sú tam objekty, hodnotu bunky či odkazy na všetky susedné bunky, avšak na začiatku sú bunky do veľkej miery naplnené len provizórnymi dátami, napríklad údaje o bojovej sile jednej či druhej strany sa na začiatku rovnajú nule. Po vytvorení celého bojového poľa sa bunky začnú napĺňať dátami, ktoré zodpovedajú skutočnému bojisku. Najskôr sa pridajú dáta o spriatelených jednotkách a strategických objektoch a to pomocou zoznamu 2D kolíznych objektov, ktorý sa získal pomocou funkcie `Collider2D.OverlapCircleAll()`. Potom sa na základe zistených nepriateľských pozícií aktualizujú aj dáta o nepriateľských jednotkách a na záver sa doplnia ukazatele na susedné bunky. Ďalšou dôležitou časťou je funkcia `SimulateCombat()`, ktorej úlohou je simulovať boj. `SimulateCombat()` zoberie dva plány, jeden ktorý bol vygenerovaný pre umelú inteligenciu 4.4.9 a druhý pre oponenta 4.4.9. Postupne začne porovnávať polohu všetkých bojových skupín normálneho plánu s tými, ktoré sa nachádzajú v pláne pre oponenta. V prípade, že sa dve skupiny nachádzajú v dostatočnej blízkosti na to, aby mohli bojovať, sa odsimuluje boj za použitia Lanchesterovho mocninového pravidla 2.7. Na základe výsledku boja sa aktualizujú bojové skupiny a simulovaný stav hry. Potom sa u každej bojovej skupiny, ktorá v danom kroku bojovala, nastaví parameter `fought` na `true`, čím sa zaručí, že v danom kroku zaútočí len raz. Po odsimulovaní všetkých skupín sa u všetkých skupinách vráti parameter `fought` do pôvodného stavu.

4.4.8 Plán

Medzi najdôležitejšie funkcie triedy `Plan` patria `SetGroupTask()`, `SetOpponentGroupTask()`, `UpdateGroupTask()` a `IsCompleted()`. `SetGroupTask()` má 3 parametre, prvým je bojová skupina triedy `CustomGroup`, druhý je zoznam úloh, `List<Task>` a posledným parametrom je pozícia bojovej skupiny. Funkcia pomocou týchto troch parametrov vytvorí nová inštanciu triedy `PlanForGroup` a uloží si ju do lokálneho zoznamu. `PlanForGroup` uchováva odkaz na bojovú skupinu, zoznam úloh, polohu skupiny a počet splnených úloh. `SetOpponentGroupTask()` funguje podobne, s tým rozdielom, že prvým argumentom je len číslo reprezentujúce veľkosť nepriateľskej skupiny. `UpdateGroupTask()` sa volá v prípade, že sa dokončila bojová úloha a jediné čo robí je, že zvýši počet splne-

ných úloh. `IsCompleted()` kontroluje, či niektorá bojová skupina nedokončila všetky úlohy. V prípade že áno sa plán považuje za splnený.

4.4.9 Monte-Carlo

Monte-Carlo plánovanie prebieha v skripte `MCPlan.cs`. Pri vytváraní novej inštancie triedy `MCPlan` sa musí zadať 5 argumentov, prvý je momentálny stav hry, druhý je počet simulácií, tretí argument je maximálna doba trvania každej simulácie, štvrtým je agresivita umelej inteligencie a posledným argumentom je majiteľ plánu a teda či sa generuje plán pre červenú alebo modrú stranu. Ako prvé sa vytvorí cyklus, ktorého počet opakovaní záleží od druhého argumentu. V cykle sa volaním funkcie `generatePlan()` vygeneruje plán pre umelú inteligenciu, následne sa zavolá funkcia `evaluatePlan()`, ktorá vyhodnotí plán a vráti číslo korešpondujúce s efektivitou plánu. V prípade, že sa jedná o prvý vyhodnotený plán alebo, že je jeho hodnota vyššia než hodnota predchádzajúceho plánu, tak sa uloží hodnota plánu a samotný plán.

EvaluatePlan

Funkcia má 4 parametre, prvý parameter je plán pre umelú inteligenciu, druhý je aktuálny stav hry, tretí je počet simulácií a posledný parameter je maximálna doba trvania každej simulácie. Na začiatku sa vytvorí `for` cyklus, počet opakovaní záleží od tretieho argumentu. Vo vnútri cyklu sa zavolá funkcia `simulatePlan()`, ktorej úlohou je simulácia boja za použitia vygenerovaného plánu pre umelú inteligenciu a plánu pre protivníka, ktorý sa vygeneruje v danej funkcii. Vráti sa číselná hodnota reprezentujúca úspech plánu, tá sa uloží. Po skončení cyklu sa vyberie najvyššia hodnota a tá sa vráti späť do hlavnej funkcie.

SimulatePlan

Na úvod sa spraví hĺbková kópia aktuálneho stavu hry za použitia funkcie `State.CopyState()`. Volaním funkcie `generateOpponentPlan()` sa vygeneruje plán pre protivníka. Boj sa simuluje po krokoch, každý krok reprezentuje 5 sekúnd reálneho času. Dobu 5 sekund som zvolil z dôvodu, že to je doba za ktorú jeden pešiak priemerne zničí iného pešiaka, čím sa lahšie simuluje boj jendnotiek. Simulácia prebieha tak, že sa vo `while` cykle volá funkcia `simulatePlanStep()`, ktorá vyhodnotí momentálny stav a vráti nasledujúci stav, ktorý sa uloží a použije v nasledujúcej iterácii cyklu. Cyklus beží dokým nevyprší maximálny čas simulácie alebo sa minimálne jeden z plánov nedokončí. Aktuálny stav plánu sa zistí funkciou `Plan.IsCompleted()`. Po skončení cyklu sa pomocou funkcie `State.Evaluate()` vyhodnotí finálny stav hry. Hodnota reprezentujúca finálny stav je výstupom tejto funkcie.

SimulatePlanStep

Vstupnými argumentmi sú: plán umelej inteligencie, plán protivníka a aktuálny stav hry. Najskôr sa vo `foreach` cykle aktualizuje poloha bojových skupín protivníka. To sa docieli tak, že sa od pozície kam sa presúvajú odpočíta ich momentálna pozícia. Následne sa výsledný vektor normalizuje. Normalizovaný vektor sa zväčší v závislosti od rýchlosti jednotiek, ktoré sa presúvajú. Tento vektor sa pripočíta k pôvodnej pozícií, čím sa získa nová pozícia. Na hracom poli sa nájdu na základe pôvodnej a novej pozície bunka z ktorej daná skupina odchádza a bunka do ktorej sa presúva. Ich dáta sa následne aktualizujú. Potom, čo

sa aktualizuje poloha všetkých bojových skupín protivníka, sa podobný proces zopakuje pri bojových skupinách, ktoré patria umelej inteligencii. Po presune všetkých bojových skupín sa v ďalšom cykle odsimuluje boj volaním funkcie `State.simulateCombat()`. V nasledujúcej fáze sa skontroluje, ktorá strana ovláda strategické objekty. Pozícia každého objektu sa porovná s pozíciou každej bojovej skupiny. V prípade, že sa nejaká bojová skupina nachádza v blízkosti objektu a objekt je ovládaný druhou stranou, tak sa `Object.ControlledBy` aktualizuje. Na záver sa overí, či niektorá bojová skupina nesplnila svoju bojovú úlohu a to tak, že sa porovná ich momentálna pozícia s pozíciou kam sa majú podľa bojovej úlohy presunúť. Ak sa úspešne presunuli na zadanú pozíciu, tak sa overí typ úlohy. V prípade, že sa jedná o útok alebo o prieskum, tak je úloha splnená a bojová skupina dostane novú úlohu. Ak sa jedná o obranu tak sa bojová skupina na dobu dvoch krokov pridá do zoznamu brániacich sa skupín, čím v nasledujúcich dvoch krokoch vyhne prideleniu novej úlohy.

GeneratePlan

Prí generovaní plánov sa do maximálnej miery využíva náhodnosť. Najskôr sa vytvorí nová inštancia triedy `Plan`. Následne sa za použitia cyklu pre každú bojovú skupinu typu `CustomGroup` vytvorí zoznám úloh. Tvorba prebieha nasledovne. Na začiatku sa vygeneruje náhodné číslo v rozsahu 0 až 100. To sa vynásobí hodnotou 1.33 v prípade, že sa zvolila nízka agresivita alebo hodnotou 0.66 v prípade, že sa zvolila vysoká agresivita. V závislosti od výsledného čísla a typu jednotky sa vyberie druh úlohy. Na výber je medzi špecifickou úlohou pre daný typ jednotky, `assault` pri údernej skupine, `recon` pri prieskumnej skupine a `tank` pri tanku a dvomi generickými druhmi úloh, `defense` a `reserve`. Cieľom úlohy typu `Assault` je zneškodnenie pozície nepriateľa alebo zabratie strategického objektu. Pozícia, na ktorá má bojová skupina zaútočiť sa vyberá náhodne. Pointou úlohy typu `Recon` je vyhľadávanie a nahlasovanie pozícií nepriateľa a vyhnutie sa samotnému boju. Úloha typu `tank` je podobná typu `assault` s tým rozdielom, že sa `tank` vyhýba budovám. Pri `defense` sa buď vyberie náhodný strategický objekt, ktorý je pod kontrolou strany, ktorá vytvorila daný `MCPlan` alebo sa zavolá funkcia `closestEnemyPosition()` a čím sa získa obranna pozícia v blízkosti predpokladaných pozícií nepriateľa. Cieľom úlohy typu `reserve`, je presunúť bojovú skupinu do týlu a tým vytvoriť rezervu, ktorá môže neskôr reagovať na kroky nepriateľa. Potom, ako sa vytvorí zoznam pozostávajúci z 10 úloh sa zavolá funkcia `Plan.SetGroupTask()`, ktorej náplň je popísaná v kapitole 4.4.8.

GenerateOpponentPlan

Na začiatku sa vytvorí nová inštancia triedy `Plan`. Generovanie plánu pre oponenta prebieha podobne ako generovanie plánu pre umelú inteligenciu s tým rozdielom, že pri vytváraní plánu pre oponenta nie sú k dispozícií presné informácie o všetkých jednotkách oponenta. K dispozícií sú len informácie o nahlásených pozíciách nepriateľa a o celkovej sile nepriateľa. Preto z pozícií oponenta vytvorí bojové skupiny. V prípade, že počet jednotiek v takto vygenerovaných skupinách je menší než celkový predpokladaný počet jednotiek oponenta, sa vygenerujú náhodne veľké bojové skupiny oponenta až dokým sa sila všetkých vygenerovaných bojových skupín oponenta nerovná celkovej predpokladanej sile oponenta. Pre všetky bojové skupiny sa vytvorí zoznam úloh a následne sa pre každú bojovú skupinu vygeneruje `PlanForOpponentGroup`, ktorý uchováva informácie o veľkosti bojovej skupiny, o jej úlohách, aktuálnej polohe a počtu splnených úloh. Tá sa priradí do k plánu pomocou funkcie `Plan.SetOpponentGroupTask()`. Rozdiel medzi `PlanForGroup` a `PlanForOpponentGroup`

je, že jeden má odkaz na bojovú skupinu, zatiaľ čo druhý má uloženú len veľkosť oponentovej skupiny.

4.5 Používateľské rozhranie

. Používateľské rozhranie pozostáva z 5 textových polí, `Text0` uchováva, `Text1` počet objektov pod kontrolou červenej strany, `Text2` ukazuje počet neobsadených objektov, `Text4` počet objektov pod kontrolou modrej strany a `Text5` bojová silu modrej strany. Všetky textové polia sú pod kontrolou skriptu `ObjectivesController.cs`. Ďalej sú súčasťou užívateľského rozhrania aj 2 tlačítka, `Restart` a `Quit`. `Restart` pomocou `SceneManager.LoadScene(SceneManager.GetActiveScene().name)` reštartne scénu do pôvodného stavu. Quit vypne aplikáciu pomocou `Application.Quit()`.

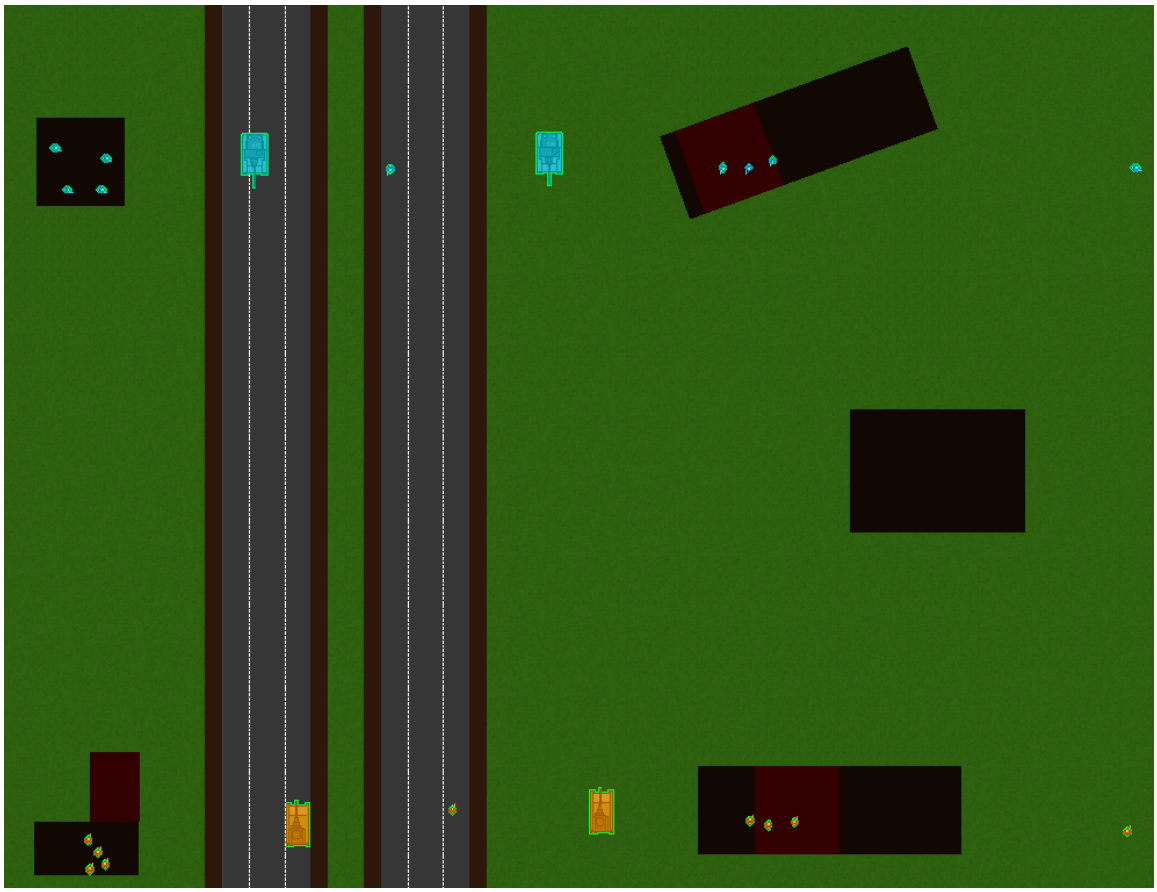
Kapitola 5

Vyhodnotenie

V tejto časti sa pomocou praktických testov vyhodnotí efektivita umelej inteligencie.

5.1 Úvod do testovania

Testovanie pozostáva z vyhodnocovania bitiev medzi červenou a modrou stranou. Každá strana má 2 tanky, a 9 strelcov. Súčasťou testov je aj skúška rôzných stupňov agresivity umelej inteligencie, ktorá môže byť pasívna, neutrálna alebo agresívna.



Obr. 5.1: Rozostavenie jednotiek v testovacom scenári

Každý test trvá 1 minútu. Po uplynutí minúty sa vyhodnotí stav hry, určí sa víťaz a hra sa vráti do pôvodného stavu. Testovanie je riadené skriptom `GlobalObject.cs`, ktorý ukladá skóre a priebežne resetuje scénu. Vyhodnocovanie testov prebieha dvomi spôsobmi.

1. Násobiacie vyhodnotenie: počet modrých objektov * bojová sila modrej strany sa poronáva s počtom červených objektov * bojová sila červenej strany,
2. Sčítací vyhodnotenie: počet modrých objektov + bojová sila modrej strany sa poronáva s počtom červených objektov + bojová sila červenej strany.

5.2 Modrá umelá inteligencia verzus červená umelá inteligencia

Na úvod sa proti sebe postavila neutrálna modrá umelá inteligencia a neutrálna červená. Týmto testom sa zistí, či má nejaká strana výhodu, napríklad kvôli lepšej počiatočnej pozícii jednotiek alebo objektov.

Najskôr sa uskutočnilo 191 testov, pri ktorých sa použilo násobiace vyhodnotenie. Z nich modrá strana vyhrala 79 testov (41.36%), 5 testov (2.62%) skončilo remízou a 107 (56.02%) testov vyhrala červená strana. Následne sa uskutočnilo 297 testov, pri ktorých sa použilo sčítacie vyhodnotenie. Modrá strana vyhrala 127(42.76%), 18 testov (6.06%) skončilo remízou a červená strana vyhrala 153 (51.51%) testov. Testy ukazujú, že červená strana má miernu výhodu. Pri zvyšných testoch sa robili obe, násobiace aj ščítacie, vyhodnotenia naraz.

Neutrálna modrá verzus agresívna červená

Pri tejto konfigurácii umelých inteligencií sa spravilo 388 testov. Výsledok, ktorý sa získal pomocou násobiaceho vyhodnotenia je:

modrá	remíza	červená
126	4	255
32.47%	1.03%	65.72%

Výsledok, ktorý sa získal pomocou sčítacieho vyhodnotenia je:

modrá	remíza	červená
143	15	230
36.85%	3.86%	59.27%

Po spriemerovaní výsledkov, červená strana vyhrala v 62.50% testov, čo je o 8.73% viacej než v základnom scenári, v ktorom vyhrala v 53.77% testov. Efektivita sa zvýšila o 16.24%.

Neutrálna modrá verzus pasívna červená

Pri tejto konfigurácii umelých inteligencií sa spravilo 270 testov. Výsledok, ktorý sa získal pomocou násobiaceho vyhodnotenia je:

modrá	remíza	červená
147	5	118
54.44%	1.85%	43.70%

Výsledok, ktorý sa získal pomocou sčítacieho vyhodnotenia je:

modrá	remíza	červená
145	15	110
53.70%	5.55%	40.74%

Po spriemerovaní výsledkov, červená strana vyhrala v 42.22% testov, čo je o 11.55% menej než v základnom scenári, v ktorom vyhrala v 53.77% testov. Efektivita sa znížila o 21.49%.

Pasívna modrá verzus agresívna červená

Pri tejto konfigurácii umelých inteligencií sa spravilo 1039 testov. Výsledok, ktorý sa získal pomocou násobiaceho vyhodnotenia je:

modrá	remíza	červená
331	9	699
31.86%	0.87%	67.28%

Výsledok, ktorý sa získal pomocou sčítacieho vyhodnotenia je:

modrá	remíza	červená
380	44	619
36.57%	4.23%	59.57%

Po spriemerovaní výsledkov, červená strana vyhrala v 63.43% testov, čo je o 9.66% viacej než v základnom scenári, v ktorom vyhrala v 53.77% testov. Efektivita sa zvýšila o 17.97%.

5.3 Zhrnutie testov

Je zrejmé, že červená strana má lepšiu štartovaciu pozíciu, čo jej zabezpečuje vyššiu percentuálnu úspešnosť. Táto výhoda sa dá korigovať správnou konfiguráciou umelej inteligencie. Agresivita umelej inteligencie hrala vo výsledkoch dôležitú rolu. Strana, ktorá bola viac agresívna mala vždy vyššiu úspešnosť, než v základnom scenári, v ktorom mali obe umelé inteligencie nastavenú neutrálnu agresivitu. Dôvodom úspechu väčšej agresivity môže byť, že v hre nie dostatočne odmenené bránenie objektov. To by sa dalo napraviť, napríklad vyššími bonusmi pre jednotky, ktoré bránia objekty alebo predĺžením doby, po ktorú majú jednotky brániť objekt. Tým by sa znížila pravdepodobnosť, že by jednotky boli zachytené nepriatelskými jednotkami, počas presunu k inému objektu.

Kapitola 6

Záver

Cieľom tejto práce bol návrh a implementácia umelej inteligencie pre strategickú hru v Unity. Z rôzných metód, ktoré sa používajú pri vývoji umelej inteligencie pre strategické hry, som si vybral relatívne málo rozšírenú metódu Monte Carlo. V rámci metódy Monte Carlo, som vytvoril abstraktnú verziu hry, ktorá slúžila na simuláciu bitiev za použitia náhodne vygenerovaných plánov. Po množstve testov sa najefektívnejší plán uložil a začal vykonávať priamo v hre. Pre simuláciou bojov, v rámci metódy Monte Carlo, som využil Lanchesterovo mocninové pravidlo. Ak by som pokračoval vo vývoji tejto hry, tak by som zvýšil obranné bonusy jednotiek, ktoré sa dajú získať, keď su jednotky schované v kryte a to z dôvodu, že testovanie dokázalo príliš veľkú efektivitu agresívnych plánov. Tiež by som zväčšil mapu, pretože v momentálnej podobe je pre umelú inteligenciu pomerné náročne obísť jednotky nepriateľa. Z pozorovania som tiež zistil, že efektivita plánu sa zvyšovala so zvyšujúcim sa počtom simulácií. Avšak množstvo simulácií je obmedzené výpočetným výkonom počítača. Existuje možnosť, že by sa dal vytvoriť efektívny plán rýchlejšie, ak by sa vytvorilo viacero abstraktných verzií hry. Niektoré by boli ďaleko abstraktnejšie, než je tá ktorú používam. Tie by slúžili na zvolenie všeobecnej stratégie, napríklad, či sa má útočiť, brániť, prípadne v ktorej časti mapy. Po vybraní všeobecnej stratégie by sa začala vyhodnocovať konkrétna stratégia nasadenia jednotiek na menej abstraktných verziách hry. Výsledky testov a moje vlastné pozorovanie ma privádza k záveru, že Monte Carlo metóda je vhodná ná vývoj umelej inteligencie pre strategické hry v reálnom čase.

Literatúra

[1] [online]. unity.com [cit. 2021-11-25]. Dostupné z: https://unity.com/our-company.

[2] *Age of Empires 2 is better than ever, 20 years later* [online]. [cit. 2021-12-12]. Dostupné z: https://www.polygon.com/2019/12/4/20994266/age-of-empires-2-definitive-edition-pc-windows-review.

[3] *ALGORITHMS - DEPTH FIRST* [online]. [cit. 2021-12-26]. Dostupné z: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/depth.html.

[4] *ALGORITHMS - MINIMAX* [online]. [cit. 2021-12-26]. Dostupné z: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/minimax.html.

[5] *ALGORITHMS: BREADTH - FIRST* [online]. [cit. 2021-12-26]. Dostupné z: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/breadth.html.

[6] *AlphaGo is the first computer program to defeat a professional human Go player, the first to defeat a Go world champion, and is arguably the strongest Go player in history.* [online]. [cit. 2021-11-27]. Dostupné z: https://deepmind.com/research/case-studies/alphago-the-story-so-far.

[7] *Classical games: Garry Kasparov beat Deep Blue (Computer) 4 to 3, with 5 draws* [online]. chessgames.com [cit. 2021-1-21]. Dostupné z: https://www.chessgames.com/perl/chess.pl?pid=29912&pid2=15940.

[8] *Emanuel Lasker* [online]. chessgames.com [cit. 2021-1-21]. Dostupné z: https://www.chessgames.com/perl/chessplayer?pid=19149.

[9] *Monte Carlo Simulation* [online]. [cit. 2021-12-28]. Dostupné z: https://www.ibm.com/cz-en/cloud/learn/monte-carlo-simulation.

[10] *Čínsky drak a námorníctvo* [online]. [cit. 2021-12-12]. Dostupné z: https://www.sector.sk/recenzia/30272/wargame-red-dragon.htm.

[11] *Návrat do éry kráľov* [online]. [cit. 2021-12-12]. Dostupné z: https://www.sector.sk/recenzia/36372/age-of-empires-ii-definitive-edition.htm.

[12] *Order of execution for event functions* [online]. [cit. 2021-12-20]. Dostupné z: https://docs.unity3d.com/Manual/ExecutionOrder.html.

[13] *Stanford-ITEP Match* [online]. chessgames.com [cit. 2021-1-21]. Dostupné z: https://www.chessprogramming.org/Stanford-ITEP_Match.

[14] *Wilhelm Steinitz* [online]. chessgames.com [cit. 2021-1-21]. Dostupné z: https://www.chessgames.com/perl/chessplayer?pid=10421.

[15] Adams, E. *The Designer's Notebook: Kicking Butt by the Numbers: Lanchester's Laws* [online]. [cit. 2022-5-7]. Dostupné z: https://www.gamedeveloper.com/design/the-designer-s-notebook-kicking-butt-by-the-numbers-lanchester-s-laws.

[16] Churchill, D. *A History of Starcraft AI Competitions (and UAlbertaBot)* [online]. 2016 [cit. 2021-1-21]. Dostupné z: http://www.cs.mun.ca/~dchurchill/starcraftaicomp/history.shtml.

[17] Davis, R. a King, J. J. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project* [print]. Addison-Wesley, 1984 [cit. 2021-11-25]. Dostupné z: http://www.shortliffe.net/Buchanan-Shortliffe-1984/MYCIN%20Book.htm.

[18] Dickenson, M. *A Model of Conflict: Iwo Jima* [online]. [cit. 2022-5-7]. Dostupné z: https://mattdickenson.com/2012/04/20/a-model-of-conflict-iwo-jima/.

[19] Drugalya, Y. *How do we use code coverage for Unity?* [online]. blog.unity.com, 2014 [cit. 2021-11-25]. Dostupné z: https://blog.unity.com/technology/how-do-we-use-code-coverage-for-unity.

[20] Lazar, D. *Understanding the Minimax Algorithm* [online]. [cit. 2021-12-25]. Dostupné z: https://towardsdatascience.com/understanding-the-minimax-algorithm-726582e4f2c6.

[21] Newborn, M. *Computer Chess*. 1. vyd. Academic Press, 1975.

[22] Radigales, A. *Lanchester's Laws of Combat* [online]. [cit. 2022-5-7]. Dostupné z: http://www.doolanshire.net/2017/08/30/lanchesters-laws-of-combat/.

[23] Schaeffer, J. *Monte Carlo Planning in RTS Games* [online]. [cit. 2022-3-21]. Dostupné z: https://webdocs.cs.ualberta.ca/~jonathan/PREVIOUS/Papers/Papers/mcsearch.pdf.

[24] Team, A. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II* [online]. 2019 [cit. 2021-1-21]. Dostupné z: https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii.