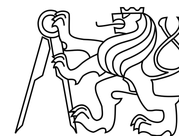
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Simulace inteligentního chování zví at v dynamickém herním prost edí
Student:	Bc. Xeniya Valentova
Vedoucí:	Ing. Marek Žehra
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

* Prove te rešerši algoritm um lé inteligence používané v sou asných po íta ových videohrách se zam ením na simulaci inteligentního chování zví at v dynamickém herním prost edí.
* Vybrané algoritmy popište.
* Nastudujte a popište herní engine Unity 3D.
* S jeho využitím navrhn te a implementujte herní modul využívající tyto algoritmy v dynamickém herním prost edí.
* Na p ípadových studiích ov te výsledné chování simulovaných zví at.
* Definujte a vyhodno te vhodné metriky jejich inteligentního chování.
* Výsledný modul otestujte a zdokumentujte.

Seznam odborné literatury

Dodá vedoucí práce.

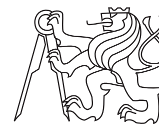
L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
d kan

V Praze dne 6. íjna 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství



Diplomová práce

Simulace inteligentního chování zvířat v dynamickém herním prostředí

Bc. Xeniya Valentova

Vedoucí práce: Ing. Marek Žehra

14. února 2017

Poděkování

Děkuji Ing. Marku Žehrovi za pomoc při vedení této diplomové práce. Mé poděkování patří též Filipu Vondráškovi za spolupráci při získávání údajů pro výzkumnou část práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 14. února 2017 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Xeniya Valentova. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Valentova, Xeniya. *Simulace inteligentního chování zvířat v dynamickém herním prostředí.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato práce se zabývá výzkumem a implementací různých druhů rozhodovací logiky umělé inteligence zvířat v dynamickém herním prostředí. V práci je zahrnuto procedurální generování terénu, v němž se zvířata musí navigovat, algoritmus hledání cest v tomto vygenerovaném prostředí, rozsáhlá rešerše možných řešení rozhodovací logiky a prohledávání stavového prostoru i samotná implementace nejvýznamnějších druhů logiky. Z výsledků práce je patrné, že všechny důležité implementace dosahují srovnatelných výsledků. Nejvýraznějším rozdílem je samotná práce pro vývojáře, protože různé přístupy zahrnují odlišnou míru práce podle toho, nakolik je daná umělá inteligence ambiciózní.

Klíčová slova AI, simulace inteligence, video hry, procedurální terén

Abstract

This thesis deals with the reseach and implementation of different kinds of animals artificial intelligence decision making in a dynamic game environment. The work includes procedural terrain generation, in which the animals have to navigate, a pathfinding algorithm, extensive research of possible solutions for decision making logic and state space search and the actual implementation of the most important kinds of the logic. Based on the results, it is apparent that

all imporatnt implementations achieve comparable results. The most notable is the work for developers, because different approaches involve different levels of work according to how the given artificial intelligence is ambitious.

Keywords AI, intelligence simulation, video games, procedural terrain

Obsah

Úvod **1**
Motivace . . . 1
Cíle . . . 1
Struktura práce . . . 2

I Teoretická část: Analýza a návrh **3**

1 Simulace inteligentního chování ve hrách **5**
1.1 Stručný úvod do problematiky . . . 5
1.2 Akademické pojetí AI . . . 7
1.3 AI ve hrách . . . 9

2 Nejrozšířenější architektury rozhodování ve videohrách **11**
2.1 Ad-hoc pravidla . . . 12
2.2 FSM . . . 12
2.3 Behaviorální stromy . . . 14
2.4 Hierarchické konečné automaty . . . 18
2.5 Plánování . . . 18
2.6 Teorie užitků (Utility-based) . . . 24
2.7 Učící se algoritmy . . . 26
2.8 Stručné shrnutí rešerše . . . 26

3 Systémy pohybu **29**
3.1 Použité pojmy . . . 29
3.2 Přehled používaných metod . . . 31
3.3 Navigační síť . . . 35

4 Dynamické prostředí **37**
4.1 Procedurální generování terénu . . . 37

4.2 Perlinův šum 38
4.3 Proměnlivost světa 39
4.4 Vliv na AI 39

5 Unity 3D engine **41**
5.1 Stručný přehled enginu 41
5.2 Vymezení klíčových pojmů v Unity 3D 42

6 Cílové chování **45**
6.1 Zvířata 45
6.2 Životní statistiky 45

II Implementační část **47**

7 Zvolené technologie **49**
7.1 Unity 3D 49
7.2 Programovací jazyk 49
7.3 Prostředí 49
7.4 Použité assety 49

8 Generování terénu **51**
8.1 Dynamické generování terénu 51
8.2 Naplnění světa 53

9 Pathfinding **55**
9.1 Implementace A* 55
9.2 Optimalizace 55
9.3 Penalizace 56

10 Struktura rozhodovací logiky **59**

11 Konečné automaty **61**
11.1 Návrh 61
11.2 Implementace 61
11.3 Testování a ladění 63

12 Behaviorální stromy **65**
12.1 Návrh 65
12.2 Implementace 65
12.3 Testování a ladění 68

13 GOAP **71**
13.1 Návrh a implementace 71
13.2 Možné optimalizace 73

13.3 Testování a ladění 73

14 Porovnání dosažených výsledků **75**

Závěr **77**

Literatura **79**

A Seznam použitých zkratek **83**

B Uživatelská příručka **85**
B.1 Unity 3D 85
B.2 Import modulu do projektu 85
B.3 Požadovaná hierarchie scény 85
B.4 Generování terénu 86
B.5 Naplnění světa objekty 86
B.6 Přidání zvířat 87
B.7 Volba architektury chování 87
B.8 Testování modulu 87

C Obsah přiloženého CD **89**

Seznam obrázků

1.1 CAPTCHA . . . 9
1.2 Model AI ve videohrách podle Millingtona [16] . . . 10

2.1 FSM ilustrující chování robota během války strojů proti lidstvu. . 13
2.2 Logika rozhodování v FSM a BT. . . . 17
  (a) Logika rozhodování v FSM . . . 17
  (b) Logika rozhodování v BT . . . 17
2.3 Struktura HFSM. . . . 18
2.4 Porovnání struktury FSM a GOAP. . . . 21
2.5 Příklad složených a primitivních akcí v hierarchickém plánování. . 22
2.6 Utility-based rozhodovací systém. . . . 25

3.1 Navigační síť . . . 36

4.1 Perlinův šum. . . . 39
  (a) Jednorozměrné Perlinovy šumové funkce s různými parametry . . . 39
  (b) Výsledný šum . . . 39

8.1 První etapa generování terénu. . . . 51
  (a) Vygenerovaný Perlinův šum . . . 51
  (b) Barevná mapa regionů . . . 51
8.2 Generování omezeného terénu pomocí falloff mapy. . . . 52
  (a) Falloff mapa . . . 52
  (b) Perlinův šum s naloženou maskou falloff mapy . . . 52
  (c) Výsledná barevná mapa ostrova . . . 52
8.4 Ukázka generovaného terénu s objekty. . . . 53
8.3 Výsledek generování terénu. . . . 54
  (a) Příklad výsledného terén . . . 54
  (b) Polygonová síť . . . 54
  (c) Flat shading . . . 54

9.1 Ukázka vygenerované mřížky pro hledání cest s neprůchozími nody. 56
9.2 Halda 56
9.3 Preferovaná cesta s použitím penalizačního systému. 57

10.1 Implementace rozhodovací logiky. 60

11.1 Zjednodušená ukázka konečného automatu chování býložravců. . . 61
11.2 Diagram tříd stavů implementovaného konečného automatu. 62

12.1 Ukázka návrhu behaviorálního stromu pro základní chování zvířat. 66
12.2 Ukázka návrhu behaviorálního stromu specifického chování predátorů. 67
12.3 Diagram tříd klíčových prvků behaviorálního stromu. 68

B.1 Import modulu do projektu 86
B.2 Ukázka MapGenerator komponenty 87
B.3 Komponenta Animal: Změna aktuální rozhodovací logiky zvířete. . 88
B.4 Komponenta AIManager: Změna rozhodovací logiky pro všechna zvířata ve scéně. 88

Úvod

Motivace

Motivací k sepsání této diplomové práce bylo v prvé řadě mé celoživotní nadšení počítačovými hrami – již od dětství jsem si uvědomovala, že hry jsou komplexním spojením několika různých odvětví oboru informačních technologií. Nicméně teprve poté, co jsem začala v herním průmyslu sama pracovat, jsem si pořádně uvědomila, nakolik je tento obor skutečně rozsáhlý a kolik vlastního úsilí je potřeba do něj vložit.

Když jsem pak v praxi viděla, jaké množství práce je potřeba investovat zejména do umělé inteligence, ale zároveň o jak moc zajímavé téma se jedná, rozhodla jsem se zjistit o umělé inteligenci co nejvíc a sepsat své poznatky do této diplomové práce, která se, jak doufám, stane vodítkem pro další herní vývojáře.

Cíle

Cíle této práce lze shrnout do následujícího seznamu:

- Provést rešerši algoritmů umělé inteligence používané v současných počítačových hrách se zaměřením na simulaci inteligentního chování zvířat v dynamickém herním prostředí.
- Nastudovat a popsat herní engine Unity 3D.
- Navrhnout a implementovat herní modul využívající vybrané algoritmy v dynamickém herním prostředí.
- Na případových studiích otestovat výsledné chování simulovaných zvířat.
- Definovat a vyhodnotit vhodné metriky jejich inteligentního chování.
- Otestovat a zdokumentovat výsledný modul.

Struktura práce

Tato práce je rozdělena na teoretickou a praktickou část.

Teoretická část obsahuje 6 kapitol. První zahrnuje úvod do problematiky umělé inteligence, definice klíčových pojmů, popisuje akademické pojetí AI a odlišnosti v kontextu vývoje videoher. Druhá, nejrozsáhlejší, analyzuje a porovnává nejznámější algoritmy rozhodování používané v umělé inteligenci. Kapitola nabízí obecný přehled klasických přístupů k AI ve vědě, detailněji se pak zaměřuje na simulaci inteligentního chování ve videohrách. Třetí kapitola je věnována hledání cest. Kapitola definuje pathfinding z matematického hlediska, popisuje a porovnává algoritmy prohledávání stavového prostoru, zaměřuje se na jejich výhody a nevýhody. Čtvrtá kapitola je zaměřena na popis zvoleného algoritmu generování dynamického prostředí videoher, ilustruje problémy, které tento algoritmus přináší při návrhu AI, a popisuje jaký vliv má na každou součást navrhované umělé inteligence. Pátá kapitola obsahuje analýzu Unity 3D, herního enginu zvoleného pro implementací výsledného modulu. Šestá kapitola obsahuje návrh cílového chování zvířat v implementovaném modulu.

Implementační část, jak je patrné z názvu, popisuje klíčové aspekty při implementaci a porovnání výsledků vybraných algoritmů, optimalizační kroky a také vhodné metriky inteligentního chování zvířat. Kromě toho tato část obsahuje výsledky testování implementovaného modulu.

Navíc práce obsahuje 3 přílohy: seznam použitých zkratek, uživatelskou příručku výsledného modulu a popis obsahu přiloženého CD.

Část I

Teoretická část: Analýza a návrh

Kapitola 1

Simulace inteligentního chování ve hrách

1.1 Stručný úvod do problematiky

1.1.1 Vymezení pojmů

1.1.1.1 Inteligence

Inteligenci můžeme definovat jako charakteristický znak některých živých organizmů, umožňující jim přizpůsobovat se a přijímat rozhodnutí v měnících se podmínkách prostředí, flexibilně reagovat na okolí. Tato vlastnost jim dává výraznou taktickou výhodu při dosahování jejich cílů.

1.1.2 Simulace inteligentního chování

Samotný pojem „inteligentní chování" je pořád tématem diskuzí. Ve většině případů je za etalon považován lidský rozum.

1.1.3 Inteligentní agent

Inteligentní entita, která vnímá svět okolo sebe pomocí senzorů a je schopná na získané vjemy reagovat a vykonávat na jejich základě racionální činnost [27]. Klade se důraz na vnější projev chování, který má být viditelný pro pozorovatele (v kontextu videoher pro hráče).

O multiagentních systémech mluvíme tehdy, pokud se systémy skládají z několika agentů, schopných nezávisle plnit delegované úkoly.

Při implementaci umělé inteligence neexistuje jednoznačná kategorizace typů agentů. Nejčastěji citované rozdělení nabídli Russel a Norvig ve své knize „Artificial Intelligence: A Modern Approach"[26]:

- *Reflexní agenti (Reflex agents)*: Reagují pouze na aktuální pozorování stavu světa. Zbytek historie vjemů je ignorován. Klíčová funkcionalita je trochu podobná sadě pravidel *podmínka → akce.*
- *Reflexní agenti založení na vnitřním stavu (State-based agents)*: Chování agentů je založeno na přesně určených stavech a definovaných přechodech mezi nimi. Patří sem způsob implementace pomocí například konečných automatů nebo stromů chování.
- *Cílově orientovaní agenti (Goal-based agents)*: Cílově orientovaní agenti, kteří se snaží dosáhnout určitého stavu světa, odpovídajícího jejich cílům. Příkladem implementace takového agenta může být plánovač.
- *Utility-based agenti*: Agenti s chováním založeným na funkci užitku, implementační technika má stejný název.
- *Učící se agenti (learning agents)*: Agenti upravující své chování v čase. Jsou implementovaní pomoci učících se algoritmů.

1.1.3.1 Umělá inteligence

Především kvůli multioborovosti této disciplíny neexistuje jednoznačná definice umělé inteligence (z anglického Artificial Intelligence – dále jen AI). Uvedeme stručnou historii tohoto oboru a ukážeme jak se měnil pohled na AI v čase.

Pojem umělá inteligence se zrodil v roce 1955[13], když ji John McCarthy, který je považován za otce AI, definoval jako „obor informatiky zabývající se tvorbou strojů vykazujících známky inteligentního chování“.

Marvin Minsky, spoluzakladatel MIT AI group, parafrázoval tento pojem v roce 1968 a právě jeho formulace je nejuznávanější definicí AI v současné době: „Umělá inteligence je věda o vytváření strojů nebo systémů, které budou při řešení určitého úkolu užívat takového postupu, který, kdyby ho dělal člověk, bychom považovali za projev jeho inteligence.“ Tato definice se zakládá na Turingově imitačním testu, popsanému detailněji v sekci 1.2.1.

Jiná známá definice AI pochází od Eleane Richové a Kevina Knighta: „Umělá inteligence se zabývá tím, jak počítačově řešit úlohy, které dnes zatím zvládají lidé lépe.“[25] Tato definice se zaměřuje na současný stav počítačových věd a nezahrnuje úlohy, které zatím neumí řešit ani člověk, ani počítač.

Je nutné zmínit taktéž Kotkovu definici, která zavadí důležitý pojem vnitřních modelů světa. Tato definice říká, že „umělá inteligence je vlastnost člověkem vytvořených systémů vyznačujících se schopností rozpoznávat předměty, jevy a situace, analyzovat vztahy mezi nimi, a tak vytvářet vnitřní modely světa, ve kterých tyto systémy existují, a na tomto základě pak přijímat účelná rozhodnutí, za pomoci schopností předvídat důsledky těchto rozhodnutí a objevovat nové zákonitosti mezi různými modely a jejich skupinami“[9]. Tato

formulace ve své podstatě zavádí základ matematického modelu umělé inteligence, který detailně rozebereme v kapitole 3. Přistupuje ke zmíněnému rozhodování jako k počátečnímu a cílovému stavu úlohy, dovoleným akcím a zadaným omezujícím pravidlům [13].

Je důležité rozlišovat akademickou definici umělé inteligence a pojetí tohoto pojmu z pohledu vývoje videoher.

1.2 Akademické pojetí AI

1.2.1 Turingův test

Alan Turing v roce 1950 navrhl a prezentoval v článku Computing Machinery and Intelligence test určující, jestli se systém umělé inteligence chová skutečně inteligentně. Ve své původní podobě test se zakládal na tzv. imitační hře[32]. Jádrem Turingova testu je názor, že inteligence je schopnost vést smysluplnou textovou konverzaci v přirozené řeči, je tedy zaměřen na sémantiku komunikace.

Turing používal porovnání s člověkem, kterého vzal jako příklad inteligentní entity.

Podle testu je systém inteligentní, pokud člověk, který s ním komunikuje pomocí textového terminálu, není schopen rozhodnout, zdali je na druhé stráně stroj, nebo jiný člověk.

Přestože Turingův test nepokrývá všechny aspekty očekávané od inteligentní entity, byl dlouhou dobu vnímán jako převládající měřítko schopností umělé inteligentní entity.

Turingův test je svým způsobem první definicí, která výrazně přispěla k formování umělé inteligence jako vědní disciplíny.

1.2.2 Kritika Turingova testu

Zřejmé – a nejvíc kritizované – nedostatky Turingova testu jsou:

- Nízká objektivita: Pouze jeden člověk rozhoduje o výsledku experimentu.
- Omezená oblast důvěryhodnosti: Testovaná entita pouze předstírá, že je člověk.
- Omezenost myšlení: Mentální procesy přemýšlející bytosti můžeme rozdělit na dvě komponenty[7]:

  Perforační: Složka, která se projevuje navenek (chováním) a může být popsána. Například činnost čtení knihy (otáčení stránek).

  Fenomenální: Vnitřní, skrytá, nepopsatelná složka. Například subjektivní pocit při čtení knihy (člověka baví příběh).

  Turingův test kontroluje pouze perforační složku entity.

1.2.2.1 Argument čínského pokoje

Jednu z nedokonalostí Turingova testu ilustruje argument čínského pokoje Jihna Searla. Tento experiment byl poprvé představen v roce 1980 v časopisu The Behavioral and Brain Sciences.

Tento myšlenkový pokus měl prokázat, že pro důkaz toho, že stroj skutečně uvažuje, nestačí pouze schopnost smysluplně odpovídat na položené dotazy

V původní formulaci tento hypotetický argument upravuje znění Turingova testu a říká, že testovaná entita může být schopná třídit tabulky s čínskými znaky podle pravidel a vytvářet smysluplné odpovědí v čínštině, aniž by rozuměla sémantické složce konverzaci[28].

Podle Searla bude počítač, který úspěšně prosel Turingovým testem, stále pouze imitovat porozumění a nemusí být inteligentní.

1.2.3 Blockhead

Ned Block nabízí jiný myšlenkový pokus, který funguje na podobném principu jako argument čínského pokoje[7]. Block říká, že každý přirozený jazyk obsahuje konečný počet syntakticky a gramaticky správných tázacích vět, stejně jako existuje konečný počet správných možných reakcí na každou otázku. Podle Blockova experimentu může existovat stroj, kterému autor říká Blockhead, jehož databáze obsahuje všechny možné otázky a odpovědi. Takový stroj je schopný složit Turingův test bez chápání významu konverzaci. Block stejně jako Searl tvrdí, že schopnost produkovat smysluplné odpovědi není postačující podmínkou inteligence.

1.2.4 Kategorizace AI

Vědecký pohled na problematiku rozlišuje slabou a silnou AI. Poprvé toto rozdělení zavedl právě Searl při popisu argumentu čínského pokoje.

Slabá AI Záměrem slabé AI je snaha vyrobit systémy, které *napodobují* lidské myšlenkové pochody. Nejedná se o replikaci, pouze o modelování. Slabou AI má podle Searla entita, která je schopná složit test čínského pokoje bez porozumění čínštině.

Silná AI (Občas Artificial General Intelligence.) Oborem silné AI je pokus o vytvoření systému, který *má vlastní mysl* (ve filozofickém významu pojmu). Zakládá se na názoru funkcionalizmu: mysl je algoritmická a médium zpracovávající algoritmy není podstatné[10]. V případě čínského pokoje má entita silnou AI, pokud si plně uvědomuje smysl konverzace.

Moderní věda zatím nedosáhla úrovně, kdy bychom skutečně potřebovali umět rozlišit umělou inteligenci od lidské pomocí testu s jednoznačným výsledkem. Zatím jsou častěji používané opačné testy, známé asi každému uživateli

Obrázek 1.1: CAPTCHA

Internetu, určené pro automatické rozlišení skutečných lidi od botů. Nazev tohoto testu, CAPTCHA, je akronymem pro „**C**ompletely **A**utomated **P**ublic **T**uring test to tell **C**omputers and **H**umans **A**part“ (plně automatický veřejný Turingův test k odlišení počítačů a lidí), v některé literatuře uváděný jako „**CAPT**ure **CHA**racters“ (zachytávání znaků). Existuje několik variant testu, například zobrazení obrázků s deformovaným textem a výzvou text vyplnit do políčka. Ukázka tohoto testu je ilustrována na obrázku 1.1.

1.3 AI ve hrách

Jako videoherní pojem je umělá inteligence programovou součástí hry, která musí přesvědčivě simulovat inteligentní chování nehratelných postav ve hře (lidské nebo zvířecí chování).

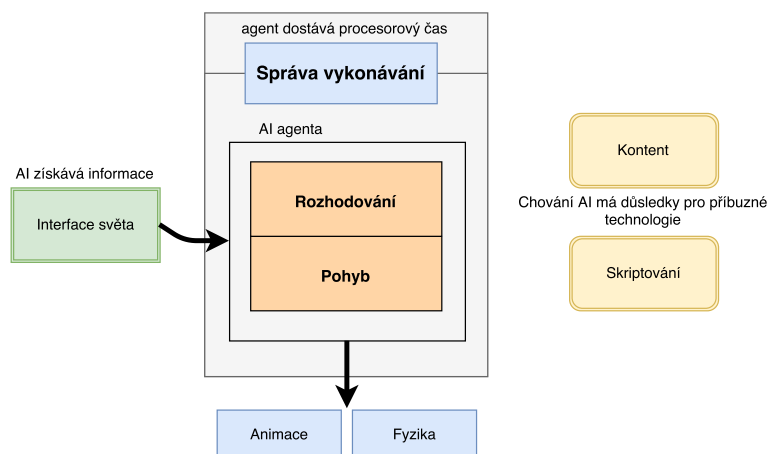
V porovnání s akademickým pojetím AI má umělá inteligence ve hrách jiný úkol. Nejde o to vytvořit skutečnou inteligenci, buďto slabou, nebo silnou, ale zlepšit prožitky hráče pomocí vytváření iluze inteligence. Na rozdíl od vědeckého přístupu je ve hrách suboptimální chování agentů často žádoucí.

Nejdůležitějšími vlastnostmi umělé inteligence ve hrách jsou:

- Koherence: AI musí v hráči vytvářet pocit, že je živou, myslící bytostí.
- Transparentnost: Chování AI je jednoduše vysvětlitelné (králík utíká před náhlým hlasitým zvukem; jiná postava utočí, když hráč začne boj první).
- Jednoduchost na úpravy: Přidání nového nebo změny existujícího chování musejí být intuitivní. Nejlepším možným scénářem je, když nastavení chování NPC postav může provádět designer.

Ian Millington a John Funge navrhují logický model AI ilustrovaný na obrázku 1.2. Tento model rozděluje úlohu AI do dvou hlavních sekcí: pohyb a rozhodování. Volitelně lze přidat strategickou vrstvu nad AI jednotlivých agentů, která ovládá AI davu.

Táto práce se drží právě zmíněného modelu a věnuje každé z jednotlivých složek kapitolu, kde je provedená analýza vhodných technik, architektur nebo algoritmů.



Obrázek 1.2: Model AI ve videohrách podle Millingtona [16]

1.3.1 Zvířata ve hrách

Většina moderních videoher s otevřeným světem či větším množstvím vnějších lokací, obzvlášť AAA tituly, implementují ve svých projektech AI fauny. Pobíhající zvířata vyvolávají dojem živého prostředí; světa, který funguje sám od sebe nezávisle na hráči.

Příkladem takových her může být Minecraft, kde zvěř plní nejen roli výplně světa, ale taktéž zdroj surovin (maso, peří. . .). Další příklady zvířat lze nalézt v sériích The Elder Scrolls, Dragon Age, Gothic a v mnohých dalších.

Kapitola 2

Nejrozšířenější architektury rozhodování ve videohrách

Tato kapitola se věnuje první složce AI modelu ilustrovanému na obrázku 1.2 – rozhodování.

Rozhodování zahrnuje volbu agenta co má dělat dále. Obvykle má každá postava řadu různých chování, která může provést. Rozhodovací systém musí zvolit, které z těchto chování je nejvhodnější v každém okamžiku hry.

Videohry mají za sebou dlouhou cestu, zvlášť za posledních několik desetiletí, stejně jako je tomu u rozhodovacích technik, které se používají v rámci AI těchto her. Některé z architektur rozhodovacích AI se však v průběhu času změnily jen málo, stává se z nich osvědčená technika na *určitý typ problémů*. A toto je velice důležitá myšlenka – ve své podstatě všechny níže popsané techniky dělají stejnou věc a je velice složité porovnávat je obecně a tvrdit, že jedna architektura je lepší než druhá. Většina architektur zvládne řešit skoro každý druh problémů, ale řešení často bude neoptimální nebo zbytečně složité. Proto si musíme uvědomit, že primární rozdíl mezi popsanými technikami je spíše v mechanizmu nežli v obsahu, který je stejný pro všechny:

- Agent zpracovává o okolním světě sadu informací, kterou používá ke generování své další akce.

- Vstupem do rozhodovacího systému je znalost, co má a/nebo umí agent, a stav prostředí, výstupem je požadavek na provedení akce.

Tato kapitola analyzuje architektury rozhodovacích AI, které jsou nejvíce používané ve videohrách, rozebírá jejích klady a zápory a popisuje problémy, na které se každá z technik chodí nejvíc.

2.1 Ad-hoc pravidla

Definované chování pomocí posloupnosti podmínek a výsledných reakcí lze sice nazvat triviální rozhodovací architekturou, ale, pochopitelně, taková struktura je naprosto neudržovatelná a nepřehledná pro jakoukoliv hru, která není v etapě prototypu.

2.2 FSM

Implementaci konečných automatů (dále FSM, z anglického Finite State Machines) najdeme téměř v každé hře počínaje od prvních dnů tohoto odvětví. I přes rostoucí popularitu složitějších architektur agentů budeme FSM potkávat skoro v každém projektu ještě dlouhou dobu.

Historicky je konečný stavový automat přísně formalizované zařízení používané matematiky k řešení problémů. Nejznámějším konečným automatem je pravděpodobně hypotetické zařízení Alana Turinga, které popsal ve svém článku „On Computable Numbers“ v roce 1936. Jednalo se o stroj předpovídající novodobé programovatelné počítače, který by mohl provádět na nekonečně dlouhém pruhu zapisovací pásky logické operace čtení, psaní a mazání symbolů.

V kontextu AI můžeme definovat FSM následujícím způsobem: Konečný stavový automat je zařízení (či jeho model), které má konečný počet stavů, v nichž se může v libovolném momentu v čase nacházet. FSM může buď reagovat na vstup přechodem z jednoho stavu do jiného, nebo vytvořením výstupu či provedením akce. FSM může v jednom okamžiku být pouze v jednom stavu.

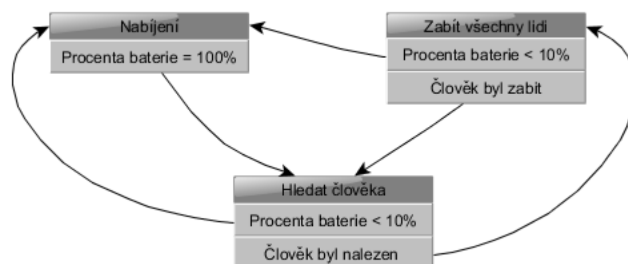
Hlavní myšlenka konečného stavového automatu je v rozložení chování objektů a dekomponování ho na snadno ovladatelné „bloky“ (stavy).

FSM jsou perfektním řešením pro jednoduché AI. FSM je definován stavy a podmínkami pro přechody mezi nimi. AI se dostane do nekonečné smyčky rozhodování a bude pokračovat v provádění současného stavu, dokud nebudou splněny podmínky pro přechod do jiného stavu.

Známé příklady použití FSM v implementaci AI ve videohrách jsou:

- Chování „duchů“ jako nepřátelského AI v Pac-Manovi je implementováno pomocí FSM. Obsahuje pouze dva stavy: pronásledování (Chase) a útěk (Evade). Přechod do stavu útěku je podmíněn vstupem od hráče, pokud dostane dočasnou schopnost zabíjení nepřátel. Přechod zpátky do stavu pronásledování závisí na časovači. Stav pronásledování je implementován jinak pro každého nepřítele.

- Quake měl implementované chování nepřátelských NPC pomocí FSM. Konečný automat obsahoval například stavy jako hledání zbroje (FindArmor), hledání možností léčení (FindHealth), hledání úkrytu (SeekCover) a útěk (RunAway).



Obrázek 2.1: FSM ilustrující chování robota během války strojů proti lidstvu.

- FIFA Football 2002 simulovala chování sportovců jako konečný automat. Dokonce chování celého týmu bylo implementováno pomocí FSM.
- Warcraft implementoval chování NPC postav pomocí klíčových stavů přesunutí na pozici (MoveToPosition), hlídky (Patrol) a dodržování nalezené cesty (FollowPath).

Schopnost přechodu z jakéhokoliv stavu do stavu jiného definováním podmínek velice usnadňuje proces návrhu chování agentů AI. Nicméně právě tato vlastnost je zároveň největší nevýhodou FSM, jelikož u AAA her FSM může snadno mít stovky stavů a při této velikosti se ladění a debugování stává velmi obtížným úkolem. Herní tituly velikosti AAA mívají víc než 500 stavů definovaného chování AI. FSM se v tomto případě stává naprosto neudržovatelným. Detailněji lze FSM shrnout do následujícího seznamu [22]:

- Složitá udržovatelnost: Přidání nebo odebrání chování vyžaduje změnu podmínek pro všechny ostatní stavy, které mají přechod na nový nebo starý stav. Změny chování v takovém počtu jsou náchylné k chybám.
- Špatná škálovatelnost: Velký počet stavů ztrácí výhodu grafické čitelnosti.
- Minimální znovupoužitelnost: FSM takřka nelze použít pro různé projekty, jelikož podmínky přechodů mezi stavy jsou uložené uvnitř stavů. Samotná struktura FSM vyžaduje vysokou provázanost stavů.
- Absence cílevědomosti: FSM ve své podstatě funguje jako Markovský proces: znalost toho, že stav je aktivní, je nezávislá na předchozích stavech. Koordinace stavů FSM pro dosažení různých a proměnlivých cílů je skoro nemožná.
- Paralelizace: Spuštění několika stavů FSM paralelně je velmi obtížné a obvykle vyžaduje externí moduly pro řešení konfliktů a uváznutí.

2.3 Behaviorální stromy

Behaviorální stromy byly poprvé použity v roce 2005 s cílem vyřešit problémy FSM při rozhodovacích procesech NPC ve hrách. Existuje několik definic BT, v této práci se budeme držet definice podle Ögrena [34] a Marzinotta [15].

Tento model se skládá z uzlů a hran. Pro dvojici uzlů spojených hranou platí, že odchozí uzel se nazývá rodičem (parent) a vstupní potomkem (child). Neexistuje žádný limit na to, kolik potomků uzel může mít. Uzly bez potomků jsou nazývány listy, uzel bez rodiče se jmenuje kořenový uzel, nebo pouze kořen.

Uzly, které se nacházejí mezi kořenem a listy, mohou být dvou typů: kompozitní (composite), nebo dekorační (decorator).

Každý podstrom definuje odlišné chování, které může být jednoduché (složené z pouze několika uzlů), nebo komplexní (složené z velkého počtu dalších podstromů).

Kořen BT generuje signál, který se jmenuje *tick*, dodržující určitou frekvenci f. Tick je propagován přes všechny podstromy, pomocí algoritmu definovaného v každém z uzlů zvlášť. V okamžiku, kdy tick dosáhne listu, provede uzel výpočet a vrátí hodnotu svého stavu. Možná ohodnocení (podle [2]) jsou: ÚSPĚCH (SUCCESS), SELHÁNÍ (FAILURE), VYKONÁVÁ SE (RUNNING) nebo CHYBA (ERROR). Dále budeme používat anglickou variantu názvu. Spočítaná hodnota je dále propagována zpět do kořene a až potom je proces vyhodnocení chování ukončen. Frekvence ticků je nezávislá na frekvenci řídící herní smyčky.

V sekci 2.3.0.1 definujeme detailněji každý typ uzlů podle kategorizace Pereiry [22].

2.3.0.1 Typy uzlů

Listy

Listy jsou nejprimitivnějšími bloky BT, mohou být dvou typů: podmínky a akce. Jsou rozděleny do kategorií podle jejich odpovědnosti.

Podmínky

Podmínečný list kontroluje, zda byla určitá podmínka splněna, a vrátí hodnotu svého stavu. Aby podmínku mohl zkontrolovat, porovnává nějakou cílovou proměnnou (například „vzdálenost do překážky“ s kritickou hodnotou pro tuto proměnnou („vzdálenost do překážky“ < 100 m). Pokud byla podmínka splněna, uzel ohodnotí sebe jako SUCCESS, jinak jako FAILURE. Tento list nevrací stav RUNNING. Graficky budeme zobrazovat tento typ listů jako elipsu.

Akce

Akce, na rozdíl od podmínek, mění stav agentů. Implementace akce je dána jejím chováním, může to být vykonání animace, pře-

hraní zvuků, nebo nějaká interní akce, která se neprojevuje navenek (ukládání souborů, změna hodnoty).

Akce vrací ohodnocení SUCCESS, pokud může být provedena, FAILURE, pokud z nějakého důvodu nemůže být dokončena. Po celou dobu vykonávání akce je vracen stav RUNNING.

Graficky tento list budeme zobrazovat jako čtverec.

Kompozitní uzly (Selektory) Hlavním účelem kompozitních uzlů (v některé literatuře se uvádí selektor, z angl. selector) je propagace ticků svým potomkům v určitém pořadí. Právě podle způsobu propagace můžeme rozdělit kompozitní uzly na tři skupiny.

Prioritní

Kompozitní prioritní uzly posílají ticky sekvenčně každému ze svých potomků, dokud některý z nich nevrátí ohodnocení SUCCESS, RUNNING nebo ERROR. Pokud ani jeden z potomků neohodnotí svůj stav takto, Selektor vrací FAILURE jako celkové ohodnocení svého podstromu. Graficky se zobrazuje jako čtverec s otazníkem.

Kód 2.1: Pseudokód implementace prioritního selektoru

```
1  for i = 1 to N:
2      state = Tick(child[i])
3
4      if state != FAILURE
5          return state
6
7  return FAILURE
```

Sekvenční

Funkcionalita sekvenčních selektorů se zakládá na sekvenční propagaci ticků svým potomkům, dokud jeden z nich nevrátí ohodnocení FAILURE, RUNNING nebo ERROR. Je to vhodný způsob vykonání série na sebe navazujících akcí a přeskočení vyhodnocení nevhodných podstromů. Graficky se zobrazuje jako čtverec s šipkou.

Kód 2.2: Pseudokód implementace sekvenčního selektoru

```
1  for i = 1 to N:
2      state = Tick(child[i])
3
4      if state != SUCCESS
5          return state
6
7  return SUCCESS
```

Paralelní Paralelní uzly posílají ticky všem svým potomkům najednou. Paralelizace je implementována lokálně pro každý uzel, aby se předešlo ztrátě kontroly nad pořadím vykonání stavů – častému

problému FSM. Paralelní selektory vracejí SUCCESS, pokud počet potomků, kteří ohodnotili svůj stav stejně, je větší než lokální konstanta S. Podobně je vráceno ohodnocení FAILURE, pokud víc než F potomků vrátilo FAILURE. V ostatních případech je vracen stav RUNNING.

Kód 2.3: Pseudokód implementace paralelního selektoru

```
1  for i = 1 to N:
2  state_i = Tick(child[i])
3
4  if nSucc(state) >= S:
5      return SUCCESS
6  else if nFail(state) >= F:
7      return FAILURE
8  else
9      return RUNNING
```

Dekorační uzly Dekorátor může upravovat chování svého jediného potomka pomocí změny hodnoty jeho stavu nebo změny frekvence posílání ticků. Například může plnit funkci cyklu a vyvolat několikanásobné vykonání stavu potomka nebo invertuje jeho ohodnocení stavu. Implementace vždy záleží na účelu dekorátora. Graficky se zobrazuje jako kosočtverec s jednoznačným krátkým popisem, co daný dekorátor dělá (například „Inv.“ pro inverter, „3 times“ pro repeater a tak podobně).

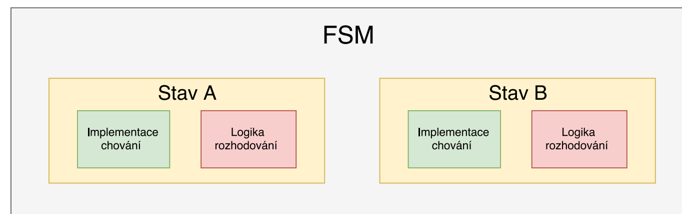
Vzhledem ke struktuře BT je průchod stromem vykonán pomocí dynamického DFS algoritmu.

2.3.0.2 Vlastnosti BT

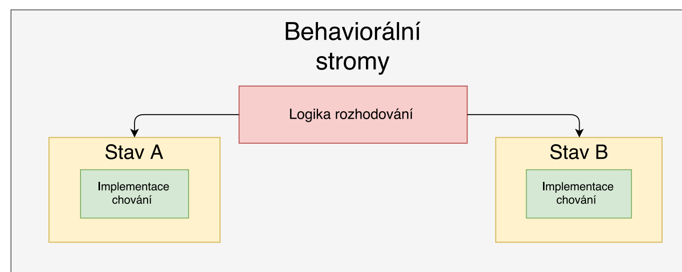
Pro porovnání s FSM provedeme analýzu stejných parametrů:

- Udržovatelnost: Je vhodné poukázat na odlišnost struktur FSM a BT, hlavně na rozdíly v pojetí implementace akcí a rozhodovací logiky. Jak již bylo zmíněno, struktura FSM se vyznačuje tím, že se agent vždy nachází v nějakém stavu, dokonce i pokud je to stav „nic nedělej“. Každý stav obsahuje vlastní rozhodovací logiku, která určuje podmínky případných změn, a popis (implementaci) chování agenta v tomto stavu (viz obrázek 2.2a. Jelikož logika definuje určité reakce na podněty okolního světa, pro velký počet případů je obecná a neměla by být závislá na stavu, ve kterém se nachází.

  Například logika definující přechod do stavu útěku, když agent slyší výstřel. V tomto zobecněném případě je agentovi jedno, co dělal v daný okamžik, jedinou vhodnou reakcí je schovat se. Pro FSM to znamená přidání redundantní logiky přechodu na stav schování se do všech ostatních stavů.



(a) Logika rozhodování v FSM



(b) Logika rozhodování v BT

Obrázek 2.2: Logika rozhodování v FSM a BT.

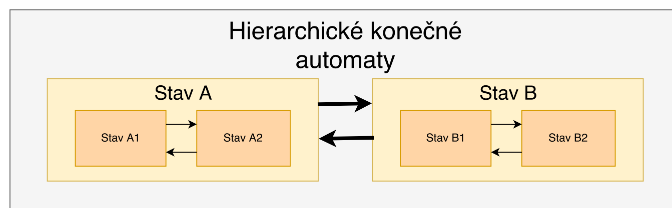
Na rozdíl od FSM jsou přechody v BT definovány samotnou strukturou architektury, nikoliv podmínkami uvnitř stavů. Právě proto jsou uzly na sobě nezávislé a přidání nebo odebrání nového chování (nebo dokonce celého podstromu chování) nevyžaduje změnu celého modelu systému – úpravy jsou jednoduché. Obrázek 2.2 ilustruje tento důležitý rozdíl.

- Škálovatelnost: Velký počet stavů může být jednoduše dekomponován na menší podstromy. Tato vlastnost velmi prospívá čitelnosti a vizualizaci celého modelu.

- Znovupoužitelnost: Jelikož jsou chování na sobě navzájem nezávislá, celé podstromy jsou také nezávislé. To umožňuje opětovné použití uzlů nebo dokonce celých podstromů v jiných stromech nebo i jiných projektech. Přenos stromu do jiného projektu nicméně vyžaduje docela velké (avšak jednoduché) úpravy kódu kvůli prioritám chování definovaných přímo v kódu.

- Cílevědomost: Ačkoliv jsou uzly v BT nezávislé, kvůli stromové struktuře modelu jsou stále spojeny. To umožňuje designerovi vytvářet podstromy pro daný cíl, aniž by byla ztracená flexibilita modelu.

- Paralelizace: V BT lze specifikovat paralelní uzly, které budou vykonávat všechny děti najednou, aniž by ztratily kontrolu nad vykonávaným modelem jako celkem. To je možné proto, že je paralelizace implementována lokálně v každém z potenciálně paralelních uzlů.



Obrázek 2.3: Struktura HFSM.

Behaviorální stromy byly použity například v Halo 2 a 3 nebo Spore, Just Cause 2, Metro 2033 a Metro: Last Light.

2.4 Hierarchické konečné automaty

Hierarchické konečné automaty (dále HFSM, z anglického Hierarchical FSM) jsou svým způsobem nadstavbou pro FSM, kdy je na FSM aplikována hlavní výhoda BT – stromová struktura. HSFM měly hlavně odstranit duplikovanou logiku přechodu.

Jak je patrné z názvu, v této architektuře existuje několik úrovní stavů. Vnitřní stavy mohou mít přechody pouze do jiných vnitřních stavů zapouzdřených do stejného „rodiče“. Vnějším stavům říkáme super-stavy (super-states), přechodům mezi nimi pak zobecněné přechody (generalized transitions).

Opětovné použití přechodů v HFSM není triviální záležitost, primárně proto, že na etapě designu musí být logika přechodu navržena pro použití v mnoha různých kontextech. Pokud tato struktura nebyla navržena univerzálně, pozdější úpravy jsou často složitější než návrh úplně nového modelu.

2.5 Plánování

Úkolem metod prohledávání je objevení sekvence akcí nebo stavů v grafu, která uspokojí cíl. Uspokojením cíle rozumíme buď dosažení specifikovaného cílového stavu, nebo jednoduše maximalizaci určité hodnoty založené na dostupných stavech. Rabin definuje plánovací systémy jako „rozšíření metod prohledávání, které dává důraz na podproblém nalezení nejlepší (nejjednodušší) sekvence akcí, které mohou být provedeny pro dosažení požadovaného výsledku, se zadaným počátečním stavem světa a přesnou definicí následků každé možné akce“ [23].

V porovnání s FSM architekturou má plánování úplně odlišný přístup: FSM diktuje agentovi, jak přesně se má chovat v každé situaci; plánovací systém říká agentovi, jaké jsou jeho cíle a možné akce a umožňuje mu rozhodnout se, jakou sekvenci akcí použít k dosažení cíle. Z tohoto úhlu pohledu jsou FSM procedurální, zatímco plánování je deklarativní.

Téměř všechny plánovací algoritmy jsou založeny na prohledávání. Uzly prohledávaného grafu odpovídají stavům, hrany jsou reprezentované přechody pomocí akcí. Prohledávání můžeme rozdělit na dva způsoby: progresivní prohledávání (v některé literatuře dopředné prohledávání) a regresivní (v některé literatuře zpětné prohledávání).

Progresivní prohledávání začíná v počátečním stavu a hledá cílový stav. Tento způsob je možné implementovat deterministicky: Například pomocí prohledávání do šířky, prohledávání do hloubky, uspořádaného prohledávání nebo hladového prohledávání. Dopředné prohledávání obnáší problém vysokého větvícího faktoru, kdy počet možností pro výběr akcí je velmi vysoký, v důsledku čehož je plýtváno prostředky na zkoušení i nepotřebných akcí.

Regresivní prohledávání začíná s cílem reprezentovaným množinou stavů a postupně hledá cestu k startovnímu stavu přes nalezené podcíle. Implementace může být stejně deterministická; situace s problémem větvení je sice lepší kvůli aproximovanému zacílení, ale pořad zůstává.

Všechny zmíněné algoritmy plánování potřebují zredukovat prohledávaný prostor pro zlepšení efektivity. Sice zčásti může pomoct nějaká „doporučovací“ heuristika nebo ořezávání prohledávaného prostoru, ale výsledné vylepšení bude velice omezeno a závislé na konkretním problému.

Výrazné zlepšení přináší systém STRIPS (Stanford Research Institute Problem Solver) navržený kolem 70. let sice pro obor robotiky, ale zároveň poskytující obecné řešení zmíněného problému nutnosti redukce prohledávaného prostoru. Zavádí pojmy předpokladů a efektů akcí a zajímá se pouze o část řešící předpoklady poslední přidané akce. Dále popsaná podkapitola 2.5.1 se zakládá právě na pojmech STRIPSu.

2.5.1 GOAP

GOAP (Goal-Oriented Action Planning) je AI systém umožňující agentům plánovat posloupnost akcí pro splnění určitého cíle. Konkrétní sekvence akcí závisí nejen na cíli, ale také na aktuálním stavu světa a konkretním agentovi. To znamená, že pokud je stejný cíl zvolen pro různé agenty či světové stavy, můžeme získat úplně jinou výslednou posloupnost akcí, což způsobuje, že AI vypadá dynamičtěji a mnohem realističtěji [20].

GOAP se zakládá na klíčovém pojmů atomického stavu světa. Je to nějaká vlastnost, která definuje jednu konkretní informaci důležitou pro agenta.

Vytváření plánu agenta probíhá s použitím následujících informací:

1. Množiny proveditelných akcí.

2. Popisu aktuálního stavu světa.

3. Definovaného cíle (požadovaného stavu světa).

Pro popis akcí musíme definovat předpoklady (preconditions), efekty (postconditions nebo effects) – oboje jsou vyjádřené zmíněnými atomickými stavy světa – a jejich cenu.

Popis aktuálního a požadovaného světa (relativní pro každého agenta) se skládá z jeho atomických stavů.

Plánovač pak na základě uvedených dat prohledává prostor možných akcí a sestavuje plán, který převede současný popis světa do požadováného stavu.

2.5.1.1 Princip GOAP

Ukážeme princip cíleně zaměřeného plánování na velice jednoduchém scénáři: Nešťastná princezna je uvězněná ve věži, kterou střeží drak.

Zavedeme následující atomické stavy se samovysvětlujícími názvy: `DrakJeNaživu`, `DržíSekeru`, `PrinceznaJeŠt'astná`.

Akce, které umí vykonávat princezna, jsou následující:

- **Odejít domů**,
  Předpoklady: `!DrakJeNaživu`
  Efekty: `PrinceznaJeŠt'astná`
  Cena: 1
- **Zabít draka**
  Předpoklady: `DržíSekeru`
  Efekty: `!DrakJeNaživu`
  Cena: 50
- **Počkat až drak zemře stářím**
  Předpoklady:
  Efekty: `!DrakJeNaživu`
  Cena: 150
- **Vzít sekeru**
  Předpoklady: `!DržíSekeru`
  Efekty: `DržíSekeru`
  Cena: 10

Aktuální situace je taková, že princezna sedí ve své věži, kde má náhodou pod rukou sekeru, a chce byt znovu šťastná. Definujme její aktuální svět pomocí atomických stavů:`DrakJeNaživu`, `!DržíSekeru`, `!PrinceznaJeŠt'astná`, požadovaný cíl: `PrinceznaJeŠt'astná`.

Možnosti princezny dosáhnout cíle jsou v tomto případě omezené:

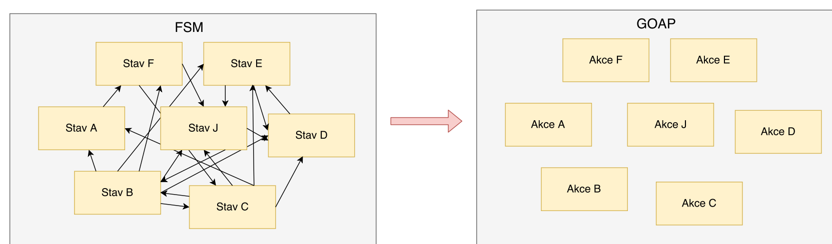
- Počkat až drak zemře stářím → Odejít domů (cena 151)
- Vzít sekeru → Zabít draka → Odejít domů (cena 61)

GOAP by na základě uvedených údajů naplánoval pro agenta-princeznu druhou, „levnější“, variantu.

2.5.1.2 Vlastnosti GOAP

Podobně jako v předchozích architekturách provedeme analýzu stejných parametrů:

- Udržovatelnost: GOAP, jako každý plánovač, rozděluje pojmy „co“ a „jak“ má agent dělat. V porovnání se špatně udržovatelným FSM principem převádí GOAP nepřehlednou směs propojení stavů na pouhou definici podporovaných akcí agenta. Obrovský rozdíl v přehlednosti (což znamená i v udržovatelnosti) je ilustrován na obrázku 2.4.



Obrázek 2.4: Porovnání struktury FSM a GOAP.

- Škálovatelnost: Přidání nových akcí, stejně jako odebírání, je velice jednoduché a neovlivňuje zbytek struktury.

- Znovupoužitelnost: Akce jsou na sobě navzájem nezávislé, logika plánování běžící na pozadí je schopná pracovat s jakýmikoliv akcemi založenými na STRIPS systému (obsahujícími efekty a předpoklady z definovaných atomických stavů světa). Případná změna možných stavů a zavedení nových akcí neovlivní algoritmus plánování. Znovupoužitelnost plánovačů je velice dobrá.

- Cílevědomost: Jak již plyne z názvu GOAP (Goal-Oriented Architecture Planner), celá tato architektura je cílevědomá. Agent definuje určitý cíl, naplánuje posloupnost akcí a dosáhne cíle.

- Paralelizace: Záleží na konkretní implementaci. Lze například zavést manažer plánování a spravovat požadavky od různých agentů paralelně.

2.5.1.3 Použití ve vývoji her

Nejznámější příklady AAA titulů, používajících GOAP při rozhodování AI agentů, jsou série F.E.A.R., S.T.A.L.K.E.R. a Deus Ex. Implementace GOAP se doporučuje pro imitaci lidského chování, jelikož komplexní systém plánování navrhuje uvěřitelnou simulaci lidských rozhodnutí: agent využívá objekty kolem sebe, dynamicky interaguje a reaguje na změny světa.

2.5.2 Hierarchické plánování

Vnitřní strukturou připomíná hierarchické plánování spíše Behaviorální stromy: dekomponuje akce na primitivní a složené, čímž se snaží zavést abstraktní obal primitivních úloh pro zjednodušení prohledávání.

Složená akce (v některé literatuře vysokoúrovňová akce, z anglického High-level action, HLA) je akce, kterou můžeme rozkládat pomocí dekompozice na podakce. Nemůže být přímo vykonaná.

Primitivní akce nemůže být dále dekomponovaná, ale zato může být vykonaná. Plní stejnou roli jako akce v GOAPu.

2.5.2.1 Klasické a hierarchické plánování

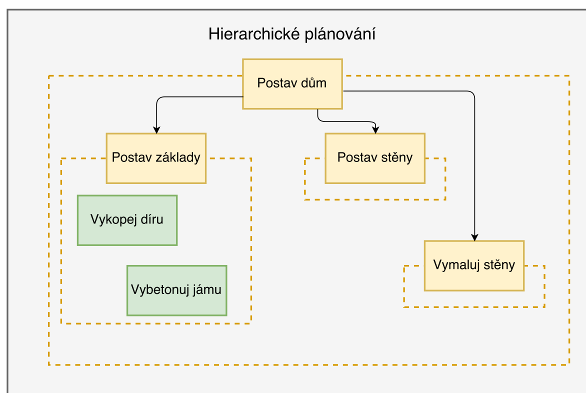
Abychom ukázali výhody hierarchického přístupu plánování, rozebereme příklad, kdy má agent za cíl postavit dům.

- **Klasický přístup**

  Klasické plánování pracuje s akcemi typu: *zvedni cihlu,dej cihlu na pozici X*, *vykopej díru na pozici Y*. Pro úkol stavení domu to znamená obrovský stavový prostor, který je potřeba prohledat.

- **Hierarchický přístup**

  Hierarchický přistup začíná pracovat s úkoly abstraktnější povahy: *postav základy*, *postav stěny*, *vymaluj zdi*. Dále je každá z větších úloh rozložená na menší. Tato struktura je znázorněna na obrázku 2.5.



Obrázek 2.5: Příklad složených a primitivních akcí v hierarchickém plánování.

Podobně jako GOAP, i hierarchický přístup pracuje s akcemi obsahujícími předpoklady a efekty a také vyžaduje přesnou definici stavů světa, ale princip a způsob plánování se zásadně liší od přístupu GOAPu[5].

2.5.2.2 Hierarchická síť úloh

Hierarchická síť úloh (z anglického Hierarchical task network, dále se bude používat zkratka HTN) je formálním systémem hierarchického plánování, nabírající větší a větší popularitu mezi moderními herními tituly[3]. Zavádí pojmy úkolová síť (cílový stav), operátor (provedení primitivního úkolu), metoda (systém provedení složeného úkolu) a plánovací doména (dvojice množin operátorů a metod). V této rešerši nebudeme uvádět formální definice každého pojmu, ale budeme se soustředit pouze na klíčové rozdíly v porovnání s klasickým plánováním[30].

Plánování vždy začíná složenou cílovou akcí, kterou se HTN snaží rozložit na sérii primitivních akcí.

Komplexní akce může mít několik metod provedení nebo se rozkládat na další složené akce.

Samotný proces plánování je rekurzivní: plánovač provádí dekompozici složených akcí, dokud nebude objevena splnitelná posloupnost primitivních akcí. V případě, že se rozložení nepodaří, plánovač označí větev jako slepou a bude zkoušet jiný rozklad nadřazené složené akce.

Jedna z variant této techniky využívá kritičnosti (v některé literatuře – ceny) přiřazené podmínkám operátorů. Obecněji jsou vyžadována specifická pravidla pro zpracování plánu.

2.5.2.3 Vlastnosti HTN

Podobně jako v předchozích architekturách provedeme analýzu stejných parametrů:

- Udržovatelnost: podobně jako GOAP nabízí sice pohodlnější systém údržby než reaktivní architektury, ale vyžaduje preciznost při definování metod, jelikož nevhodně navržena struktura metody má výrazný vliv nejen na rychlost plánování, ale i celkovou řešitelnost úloh [5]. Design samotné struktury je definující pro úspěšnost jejího použití.

- Škálovatelnost: Přidání a odebírání nových akcí je složitějším procesem než v GOAPu ze stejných, výše zmíněných důvodů.

- Znovupoužitelnost: Podobně jako v GOAPu je logika plánování schopna pracovat s jakýmikoliv konzistentně definovanými stavy, i když pro znovupoužití v jiném projektu bude vyžadovat větší úpravy než například GOAP.

- Cílevědomost: Stejně jako GOAP architektura, HTN se zakládá na cílevědomosti agenta, který ví, jak má vypadat požadovaný svět a plánuje jak se dostane do tohoto stavu.

- Paralelizace: Stejně jako u jiných plánovačů, možnost a efekty paralelizace jsou závislé na konkretní implementaci.

2.5.2.4 Použití ve vývoji her

Známé herní tituly úspěšně využívající HTN jako rozhodovací architekturu AI agentů jsou například Killzone 2 a 3, Max Payne 3 nebo Dark Souls. Tento princip je vhodný pro velké otevřené světy a pro velmi komplexní rozhodování: očekávané složité cíle, velmi věrohodné chování agentů, obrovský počet možných akcí. Jsou doporučené pro imitaci lidských postav [6].

2.6 Teorie užitků (Utility-based)

Architektury založené na teorii užitku (z anglického utility-based, dále se bude používat anglická varianta pojmu) jsou další možnosti implementace rozhodovací složky agentů. Podobně jako plánovače, Utility-based systémy nemají předem přesně definovaný seznam očekávaných chování, jsou méně strukturované než FSM nebo BT [24].

Tato architektura se zakládá na teorii užitků, popisující co můžeme chtít a jak máme ohodnotit potenciální rozhodnutí. Používá funkci užitku, jejímž cílem je zmapovat stavy světa na reálná čísla [1]. Architektura pracuje s očekávaným užitkem, představujícím průměr přes všechny potenciálně možné stavy. Na základě tohoto pak agent volí „nejlepší“ akci, která maximalizuje očekávaný užitek, a provádí ji.

Příklad takového rozhodování je dobře ilustrován v sérii The Sims, v samotném jejím rozhraní. Agent ohodnotí každou potenciální akci na základě kombinace aktuálních potřeb a schopnosti akce tuto potřebu uspokojit. Dále agent vypočítá průměrný užitek přes všechny možné výsledné stavy a z těchto výpočtů vybere nejvíc „vhodnou“ akci.

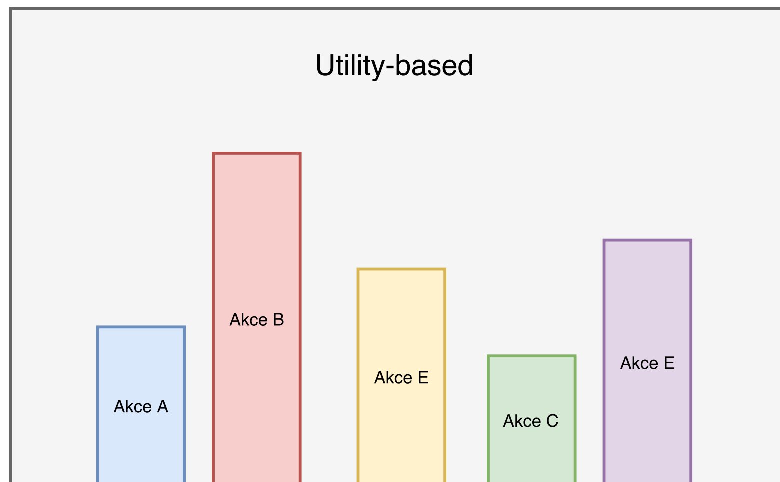
Jelikož se celé hodnocení zakládá na dobře navrženém seznamu akcí a definované váhy bere rozhodovací logika jako pouhé doporučení pro volbu akce, je občas složité předpovědět, co přesně zvolí agent v určité situaci. Na rozdíl od kupříkladu BT, kde je možné proiterovat celou stromovou strukturu a s jistotou říct, jak se zachová agent, v utility-based architektuře je nutné počítat s určitou ztrátou míry kontroly nad chováním.

Na jinou stranu je výhodou této struktury záruka, že pokud jsou akce navržené dobře, zachová se agent uvěřitelně i v případě nedefinovaných situacích.

2.6.0.1 Vlastnosti Utility-based systému

Provedeme rozbor dříve analyzovaných parametrů pro porovnání s ostatními popsanými přístupy:

- Udržovatelnost: Jelikož utility-based systémy řeší, která konkrétní akce se bude provádět dále, samotný kód rozhodování je na jednom místě a celý systém je přehledný: rozhodovací logika je docela jednoduchá proto, že, jak již bylo uvedeno, spoléhá na promyšlený design a původní nastavení akcí.



Obrázek 2.6: Utility-based rozhodovací systém.

- Škálovatelnost: Podobně jako u plánovače je přidání nebo odebrání nových akcí poměrně přímočaré: Stačí definovat základní nastavení a příslušné váhy a systém sám je dokáže jednoduše zapracovat do rozhodovací složky agenta. Pravě tato vlastnost je jedním z hlavních důvodů, proč jsou takové hry jako série The Sims snadno rozšiřitelné a přidání nového obsahu do nich je poměrně jednoduché a často nevyžaduje ani změny kódu hry.

- Znovupoužitelnost: Stejně jako u plánovačů, logika bežící na pozadí rozhodování je snadno znovupoužitelná v jiných projektech s minimálními úpravami. Jedinou podmínkou je formálně definovaný seznam akcí s příslušnými váhami.

- Cílevědomost: Zaměření na cíl v utility-based systému nedosahuje sice takové míry jako u goal-based systémů, které na cílevědomosti staví, ale nějaké povědomí o cílech existuje, na rozdíl od FSM a BT.

- Paralelizace: Stejně jako u jiných plánovačů, možnost a efekty paralelizace jsou závislé na konkretní implementaci.

2.6.0.2 Použití ve vývoji her

Systémy založené na užitkové funkci jsou dobře použitelné v různých typech videoher, ale jsou vhodnější (vzhledem k náročnosti návrhu) v situacích, kdy počet potenciálně proveditelných akcí je velký a kdy nemůžeme jednoznačně určit „správné“ chování. Utility-based architektury jsou hodně používané v moderních hrách. Nejznámějšími příklady jsou takové tituly jako již zmíněná série The Sims nebo Section 8: Prejudice.

2.7 Učící se algoritmy

Některé z klasických AI přístupů z akademického prostředí, jako jsou například genetické algoritmy a neuronové sítě, jsou jen zřídka používané při vývoji her [17]. Takové přístupy v kontextu videoher jsou obecně považovány za příliš náročné na vývoj (tím je myšlený i čas designu a ladění). Pro dosažení přijatelného výsledku tyto obecné přístupy musí být silně modifikované a specializované pro konkrétní hru. Analýza (a pochopení) důvodů výsledného chování u těchto algoritmů je obtížná, podobně jako testování a hlavně ladění směrem k víc očekávanému chování na výstupu.

Z tohoto důvodu vynecháme detailní analýzu vnitřní struktury a logiky klasických AI přístupů a rozbor jejich vlastnosti.

2.7.0.1 Použití ve vývoji her

Doposud jen velmi málo komerčních her našlo odůvodněné využití učících se technologií [14]. Mezi nimi jsou například Creatures, Supreme Commander 2 nebo Black & White.

2.8 Stručné shrnutí rešerše

Zrekapitulujeme klíčové výhody a nevýhody analyzovaných architektur z pohledu využití ve videohrách[14] do tabulky 2.1.

Na základě provedené analýzy byly zvoleny tři populární a zároveň nejvíc odlišné architektury pro implementaci a následné porovnání ve výsledném modulu. Hierarchické konečné automaty, stejně jako hierarchické plánování, se lehce podobají svou logikou behaviorálním stromům. Pravě proto byla pro implementaci zvolena základní forma konečných automatů, behaviorální stromy a nehierarchický GOAP.

Detaily implementace každého z přístupů jsou popsané v druhé části této práce v příslušných kapitolách.

Tabulka 2.1: Výhody a nevýhody architektur rozhodování AI agentů

Architektura	**Výhody**	**Nevýhody**
Ad-hoc pravidla	Triviální implementace	Naprosto neudržovatelný a nepřehledný systém dokonce i pro jednoduché chovaní

Tabulka pokračuje na další stránce

Tabulka 2.1 – pokračování z předchozí stránky

Architektura	Výhody	Nevýhody
Konečný automat (FSM)	Intuitivně pochopitelná struktura Jednoduchá implementace	Přibývající počet přechodů a stavů rychle zhoršuje přehlednost kódu a snižuje udržovatelnost
Hierarchický konečný automat (HFSM)	Rozděluje chování na moduly Intuitivně pochopitelná struktura	Složitá udržovatelnost ovlivněná počtem přechodů
Behaviorální stromy (BT)	Přehledná struktura, jednoduchá na pochopení a údržbu Snižuje provázanost logiky rozhodování a implementace konkretních stavů	Prioritu akcí je nutno manuálně nastavit.
Plánovače	Agent sám rozhoduje o posloupnosti akcí nutných k dosažení cíle Dobře zvládá nedefinované situace Velice jednoduchý proces přidaní nebo odebírání nových stavů a jejich zapojení do rozhodovací logiky	Lehká ztráta designérské kontroly v nastavení chování V závislosti na počtu akcí a na implementaci může být výpočetně náročné

Tabulka pokračuje na další stránce

Tabulka 2.1 – pokračování z předchozí stránky

Architektura	**Výhody**	**Nevýhody**
Architektury založené na funkci užitků	Agent má přistup ke všem akcím, není potřeba definovat omezení jeho chování Dobře zvládá nedefinované situace Nabízí variace chování: i malá změna podmínek může způsobit jiné, ale neméně uvěřitelné chování v určitých situacích	Vetší ztráta designérské kontroly v nastavení chování Vyžaduje precizně promýšlený návrh seznamu akcí V závislosti na počtu akcí a na implementaci může být systém výpočetně náročný
Učící se algoritmy	Agent může upravovat své chování během „učení“ Základní kostra chování může být naimplementována velmi rychle	Ztráta designérské kontroly v nastavení chování v plné míře Chování je neupravitelné.

Kapitola 3

Systémy pohybu

Tato kapitola se zaměřuje na druhou složku modelu AI uvedeného na obrázku 1.2 v první kapitole. Pohyb se skládá z algoritmů používaných v případech, kdy rozhodnutí přimělo agenta ke změně pozice.

Velké množství problémů, které musíme řešit algoritmicky, můžeme převést na prohledávání stavového prostoru. Pathfinding, neboli navigace na mapě, patří právě k tomuto typu úloh. Dále v této práci budeme používat anglický název pojmu.

Definujme klíčové pojmy z teorie grafů a nejčastěji používané algoritmy při řešení úloh prohledávání stavového prostoru.

3.1 Použité pojmy

3.1.1 Neorientovaný graf

Nechť E, V jsou libovolné disjunktní množiny a $\varrho : E \to V \bigotimes V$ zobrazení. Uspořádaná trojice $G = (E, V, \varrho)$, kde prvky množiny E jsou hranami grafu, prvky množiny V uzly grafu a zobrazení ϱ incidencí grafu, tvoří **neorientovaný graf** G. $V \bigotimes V$ je množina všech neuspořádaných dvojic prvků z V.

3.1.2 Orientovaný graf

O **orientovaném grafu** mluvíme tehdy, pokud ϱ značí zobrazení $E \to V \times V$, tj. zobrazení z množiny hran do uspořádaných dvojic uzlů.

3.1.3 Stavový prostor

Stavový prostor je *orientovaný prohledávatelný graf*, kde:

- Uzly představují **stavy** (současný popis stavu řešeného problému). Množinu stavů budeme značit S.

- Hrany reprezentují **akce** (způsobují přechod mezi stavy). Množinu akcí budeme značit A.

Stavový prostor je množinou všech stavů systému s operátory a akcemi, umožňujícími přechod mezi stavy.

3.1.4 Následník stavu

Nechť máme stavový prostor (S, A) a stav $s \in S$. Pokud $(s, s') \in A$ (ve stavu s můžeme použit akci s'), pak říkáme, že stav $s' \in S$ je následníkem stavu s.

3.1.5 Sled

Posloupnost uzlů a hran s nimi incidujících.

3.1.6 Tah

Takový sled grafu, ve kterém se hrany neopakují.

3.1.7 Cesta

Takový tah grafu, ve kterém každý uzel inciduje maximálně se dvěma hranami tohoto tahu.

3.1.8 Orientovaná cesta

Cesta v orientovaném grafu dodržující orientaci hran.

V kontextu stavového prostoru se orientovanou cestou ze stavu s_1 do stavu s_n rozumí posloupnost stavů a akcí, pro kterou platí následující podmínka: $\forall\ i \in \{1, 2, ..., n-1\} : a_i = (s_i, s_{i+1}) \land s_{i+1} \in \Gamma(s_i)$, kde $\Gamma(s_i)$ je množina všech následníku stavu s_i.

V některé literatuře se setkáváme s definicí cesty pouze jako posloupnosti akcí s tím, že stavy můžeme dopočítat.

3.1.9 Expanze

Nalezení množiny všech následujících uzlů.

3.1.10 Souvislý graf

Takový graf, kde pro každé dva vrcholy x a y existuje sled.

3.1.11 Strom

Souvislý graf neobsahující kružnici nazýváme strom.

3.1.12 Úloha prohledávání stavového prostoru

Problém řešící prohledávání stavového prostoru je zadán:

- Stavovým prostorem (S, A),
- počátečním stavem I, kde $I \in S$,
- množinou koncových stavů $G \subseteq S$.

Řešením teto úlohy je pak orientovaná cesta z počátečního do koncového stavu. Takovou cestu nazýváme plánem, metody hledaní cest pak metodami řešení úloh.

V praxi lze problém prohledávání stavového prostoru převést na dva typy úloh:

- Hledání cesty: Ve výsledku potřebujeme celou cestu, popis stavů, kterými procházíme, a hlavně seznam akcí, které aplikujeme. Koncový stav je znám a dobře popsán. Tento problém je převeditelný na konstrukci prohledávacího stromu.
- Hledání cílového stavu: Cesta ke koncovému stavu není tak důležitá jako samotný koncový stav (jeho popis).

Tato práce využívá pouze problémy hledání cesty.

Musíme brát v úvahu, ze úloha řešení problému prohledávaní stavového prostoru nemusí končit nalezením cesty. Často je součástí opakovaného průchodu cestou, což znamená, ze ne vždy potřebujeme optimální nejkratší cestu, ale rychle hledáme jakékoliv řešení dodržující alespoň potřebný směr.

Další specifickou částí problematiky hledání tohoto řešení je rozsah stavového prostoru, který je na rozdíl od podobných úloh (například hledaní cest v grafu) obrovský (občas i nekonečný)[8]. Vzhledem k tomuto je systematické prohledávaní často neefektivní. Kromě toho často v případě prohledávání stavového prostoru potřebujeme započítat i nějakou doplňující informaci, která může výrazně ovlivnit směr postupu hledaní. Tyto informace nazýváme *znalostními heuristikami.*

3.2 Přehled používaných metod

Existuje velký počet algoritmů řešících problém prohledávání stavového prostoru. Jak již bylo zmíněno, můžeme je rozdělit na dvě skupiny:

- *Neinformované*: Algoritmy, které nerespektují strukturu problému, nemají informaci o jeho vlastnostech a mají tendenci prohledávat irelevantní uzly. Nevyužívají znalostí o řešeném problému.

- *Informované*: Na základě informací o úloze definují nezápornou hodnotící funkci f stavů, která definuje vzdálenost od cíle, což umožňuje kontrolovat rozšiřování stavů a korigovat jej expanzí nejperspektivnějších stavů.

Do první skupiny spadají triviální algoritmy prohledávání stavového prostoru: Náhodné prohledávání, prohledávání do šířky (BFS) a prohledávání do hloubky (DFS). Pokročilejší metody (Dijkstra, heuristiky, Greedy search, A*) patří do skupiny informovaných algoritmů.

Uděláme stručný přehled a popis nejpoužívanějších algoritmů z obou skupin.

3.2.1 Neinformované algoritmy prohledávání stavového prostoru

Každý algoritmus při prohledávání generuje strom, který je podgrafem grafu stavového prostoru. Na začátku je vytvořen kořen stromu (počáteční uzel) a pak, dokud ani jeden list vytvořeného stromu není cílovým stavem, se opakuje výběr jednoho z listů a jeho expanze.

Tyto metody rozlišujeme podle pořadí expanze uzlů, které má zásadní vliv na efektivitu a optimalitu algoritmu.

Zavedeme terminologii stavů uzlů: čerstvý (fresh), otevřený (open) a uzavřený (closed).

3.2.1.1 Náhodné prohledávání

Náhodné prohládávání volí v každém kroku náhodný list k expanzi. Tento algoritmus je použitelný pouze pro nejjednodušší problémy, u kterých nepotřebujeme optimální řešení.

3.2.1.2 Prohledávání do šířky

Algoritmus prohledávání do šířky (BFS, breadth-first search) prochází graf počínajíc zvoleným uzlem. Po jeho zpracování a nalezení vygenerovaných potomků je uložíme do fronty a zpracujeme později. Expanze je provedena systematicky: Celé patro je prohledáno před pokračováním níž; prohledávané uzly dodržují pořadí podle vzdáleností od kořene.

Klíčové kroky jsou následující[13]:

1. Počáteční stav je přidán do fronty *vertices*. Pokud je počáteční stav zároveň cílovým, je prohledávaní ukončeno.

2. Pokud OPEN není prázdná, vyjmeme z ní stav, označíme ho indexem (i) a nastavíme mu příznak CLOSED. V opačném případě (fronta OPEN je prázdná) řešení neexistuje a prohledávaní je ukončeno.

3. Expandujeme stav i.

4. Pokud jsou již všichni následníci stavu i uzavřeni, nebo žádné nemá, pokračujeme krokem č. 2.

5. Všechny neuzavřené nalezené následovníky přidáme do *vertices* (na konec, jelikož fronta je založená na FIFO metodě).

6. Pokud je některý z nalezených následníků cílovým stavem, je prohledávaní ukončeno. Jinak pokračujeme krokem č. 2.

Tento algoritmus je úplný. To znamená, že BFS zaručuje nalezení optimální cesty (s nejmenším počtem hran), pokud cesta existuje[18]. Jeho nároky na paměť rostou exponenciálně s délkou cesty k cíli, což jej dělá skoro nepoužitelným pro řešení problémů AI v praxi.

3.2.1.3 Prohledávání do hloubky

Na rozdíl od BFS algoritmus prohledávání do hloubky (DPS, depth-first search) prioritně expanduje uzly s největší hloubkou. Často je potřeba omezovat největší povolenou prohledávanou hloubku. Po jejím dosažení je možné použít algoritmus navracení (backtracking).

Myšlenka DPS spočívá v následujících krocích [13]:

1. Počáteční stav je přidán do zásobníku *vertices*. Pokud je počáteční stav zároveň cílovým, je prohledávaní ukončeno.

2. Pokud *vertices* není prázdný, vyjmeme z něj stav, označíme ho indexem (i) a nastavíme mu příznak CLOSED. V opačném případě řešení neexistuje a prohledávaní je ukončeno.

3. V případě, že hloubka stavu i je maximálně přípustná, pokračujeme krokem č. 2.

4. Expandujeme stav i.

5. Pokud jsou všichni následníci stavu i již uzavřeni, nebo žádné nemá, pokračujeme krokem č. 2.

6. Všechny neuzavřené nalezené následníky přidáme do *vertices* (na začátek, jelikož zásobník je založen na LIFO metodě).

7. Pokud je některý z nalezených následníků cílovým stavem, je prohledávání ukončeno. Jinak pokračujeme krokem č. 2.

Nalezená cesta není zaručeně optimální. Kromě toho mají oba způsoby výraznou nevýhodu, že expandování je provedeno na mnohem vetším počtu uzlů než je potřeba. Právě proto neinformované algoritmy mohou být použité pouze v jednodušších problémech.

3.2.2 Informované algoritmy prohledávání stavového prostoru

Na rozdíl od předchozí skupiny, informované algoritmy prohledávání stavového prostoru mohou najít nejkratší a optimální cestu mnohem rychleji, jelikož z každého prohledávaného vrcholu vyrazí zhruba správným směrem. Kromě toho můžeme kontrolovat a korigovat expanzi uzlů. Jedním z nejčastěji používaných informovaných algoritmů prohledávání stavového prostoru je A*.

3.2.2.1 A*

Jak již bylo zmíněno, podstatnou změnou v porovnání s neinformovanými algoritmy je hodnotící funkce $f(u)$ definující vzdálenost k cílovému stavu (většinou vyjadřuje součet vzdáleností cílového uzlu od uzlu u). Avšak spolehlivost této funkce závisí na konkretním problému, a proto mnohem výhodnější uvažovat f jako funkci ve tvaru: $f(u) = g(u) + h(u)$, kde $g(u)$ představuje vzdálenost od počátku a h tvoří vzdálenost od u k cíli t. Tento přistup má výhodu v tom, že hodnotící funkce pak vyjadřuje délku cesty řešení procházející uzlem u [8]. Problém nastává při samotném výpočtu hodnot $g(u)$ a $h(u)$.

Jelikož nemáme možnost spočítat hodnoty přesně kvůli absenci pravidel pro výpočet, musíme použít nějakou aproximovanou hodnotu. Za běhu algoritmu máme možnost upřesňovat hodnotu vzdálenosti od začátku ($g(u)$), ale pro odhad vzdálenosti v zatím nevygenerované části stavového prostoru musíme použit heuristiku, což znamená že $h(u)$ plní roli heuristické funkci.

Klíčové kroky algoritmu jsou následující [13]:

1. Počáteční stav je přidán do seznamu *vertices*.

2. Pokud je *vertices* prázdný, řešení neexistuje a prohledávaní je ukončeno. V opačném případě z něj vybereme stav s nejnižší hodnotou f, označíme ho indexem (i) a nastavíme mu příznak CLOSED. Pokud během výběru najdeme několik stavů se stejnou nejnižší hodnotou, hledáme, který z nich je cílový, a zvolíme jeho; pokud ani jeden, zvolíme libovolný z nich.

3. Odstraníme stav i ze seznamu *vertices* a nastavíme mu příznak CLOSED.

4. Pokud je i cílovým stavem, prohledávání je ukončeno, máme řešení.

5. Expandujeme stav i. Pro každého následníka j stavu i spočítáme hodnotu $f(j)$ a zkontrolujeme, jestli se nachází v seznamu *vertices*, nebo zda má nastaven příznak CLOSED.

    - Je-li podmínka splněna a zároveň má aktuální stav větší hodnotu $f(j)$ než je hodnota aktuálně počítaného následníka, přidáme ho do seznamu *vertices* a změníme mu rodičovský uzel a ohodnocení $f(j)$ na nové.

- Pokud není podmínka splněna, přidáme ho do *vertices*.

6. Pokračujeme krokem č. 2.

Pravě tento algoritmus bude použit v této práci: jako implementace pathfinding systému a také při plánování posloupnosti akcí v GOAP.

3.3 Navigační síť

V enginu Unity 3D je jako datová struktura při hledání cest použita navigační síť. Navigační síť (z angl. Navigation Mesh) byla velmi populární strukturou pro pathfinding ve videohrách od roku 2000 [4], v omezené míře se nicméně používá doteď.

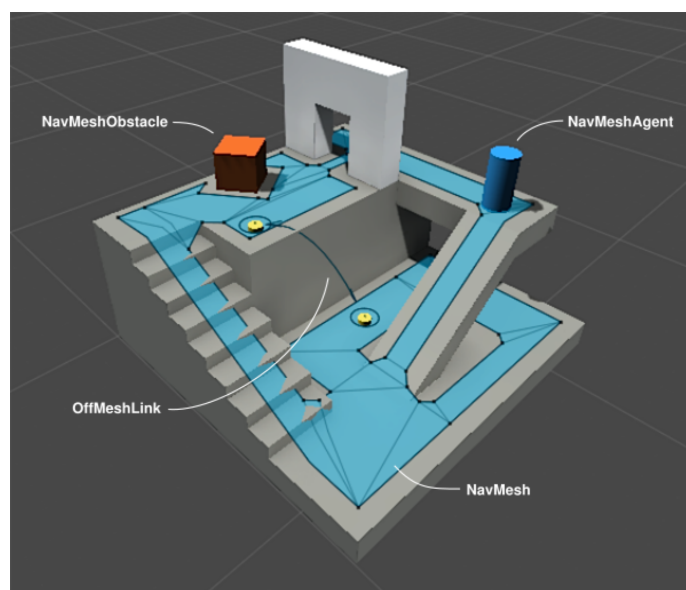
Navigační síť v Unity 3D obsahuje informace o průchozích blocích herního světa, neprůchozích překážkách a způsobech pohybu agentů. Síť umožňuje hledání cesty z jednoho bloku do druhého právě na základě těchto dat. Její klíčová myšlenka je ilustrována na obrázku 3.1.

Unity 3D na této struktuře spouští vlastní implementaci A* algoritmu. Jediný požadovaný vstup od herního vývojáře je pak definovat průchozí a neprůchozí bloky [33].

Navigační síť se vytváří v kompilačním čase podle geometrie scény, což dělá navigační síť nevhodnou volbou pro herní svět, který se v průběhu svého života mění.

Samozřejmě existují možnosti manuálního vytváření navigační sítě pro dynamicky generované světy. Ale jelikož se tato práce zaměřuje na zkoumání algoritmů simulace umělé inteligence a nikoliv primárně na hledání cest, implementace přepočítávání navigační sítě za běhu aplikace by byla zbytečně komplikovaná.

Při psaní této diplomové práce byla zvolena vlastní implementace datové struktury na udržení informací o světě a taktéž vlastní implementace A*.



Obrázek 3.1: Navigační síť
Zdroj: `https://docs.unity3d.com/Manual/nav-NavigationSystem.html`

Kapitola **4**

Dynamické prostředí

Z pohledu AI agenta můžeme prostředí rozlišovat na statické a dynamické. Podle Russela: „Pokud se prostředí mění během agentova uvažování, pak je prostředí z jeho hlediska dynamické, jinak je statické. Statická prostředí mají výhodu v tom, že během svého uvažování nemusí agent sledovat svět, a také se nemusí zabývat časem.“ [26].

Cílem této práce je navrhnout modul chování zvířat pravě v dynamickém prostředí. Tato kapitola stručně popíše možnosti simulování takového prostředí, především generování terénu a pak run-time dynamické změny okolí agentů, které nutí je sledovat svět kolem sebe. Kromě toho je na konci kapitoly provedena analýza, jaký vliv bude proměnlivé prostředí mít na AI v navrženém modulu.

4.1 Procedurální generování terénu

Pro generování terénu se často používá výšková mapa (z angl. height map), která uchovává hodnoty výšek vrcholů pro každou souřadnici terénu. Použití klasického generátoru pseudonáhodných čísel není nejvhodnější metodou pro generovaní výškových map kvůli tomu, že výsledek na výstupu, paradoxně, je příliš náhodný a nevytváří dojem přirozenosti.

Existuje velký počet možností generování výškové mapy: Metoda přesouvání středního bodu (midpoint displacement), nanášení částic (particle deposition), metoda náhodných poruch (random faults), Perlinův šum (Perlin noise), multifraktály a mnoho dalších.

V teto práci nebudeme detailně rozebírat a porovnávat každou z metod. Misto toho popíšeme pouze jedinou, zvolenou pro implementaci, Perlinův šum.

4.2 Perlinův šum

Perlinova šumová funkce byla navržena Kenem Perlinem v roce 1983 a poprvé představena světu na konferenci SIGGRAPH v roce 1985. Sice je v základu myšlenky algoritmu stále generátor pseudonáhodných čísel, ale mezi výstupními dvojicemi hodnot je prováděna interpolace. Tento krok trochu vyhladí výslednou funkci a odstraní nepřirozenou náhodnost. Může se jednat o interpolaci lineární, kvadratickou, kubickou nebo jejich kombinaci [31].

Ken Perlin při původním návrhu výpočtu šumu použil funkci *s_curve()* definovanou jako:

$$s_curve(t) = (t^2 * (3 - 2t))$$

. Vyhlazování probíhalo pomoci funkce *lerp()* a náhodných hodnot a a b:

$$lerp(t, a, b) = (a + t(b - a))$$

.

Perlinův šum vypočítaný takovým způsobem se však neaplikuje přímo, výsledek je stále nedostatečně přirozený. Pro dosažení nejlepších výsledků jsou jednotlivé šumové funkce (v některé literatuře označované jako oktávy) generovány s různými frekvencemi a amplitudami a pak jsou sečteny. Tento proces je ilustrován na obrázku 4.1

Perlinův šum je koherentní, což znamená že změny přechodů jsou gradualní.

4.2.1 Parametry

Nejdůležitější parametry při implementaci jsou [11]:

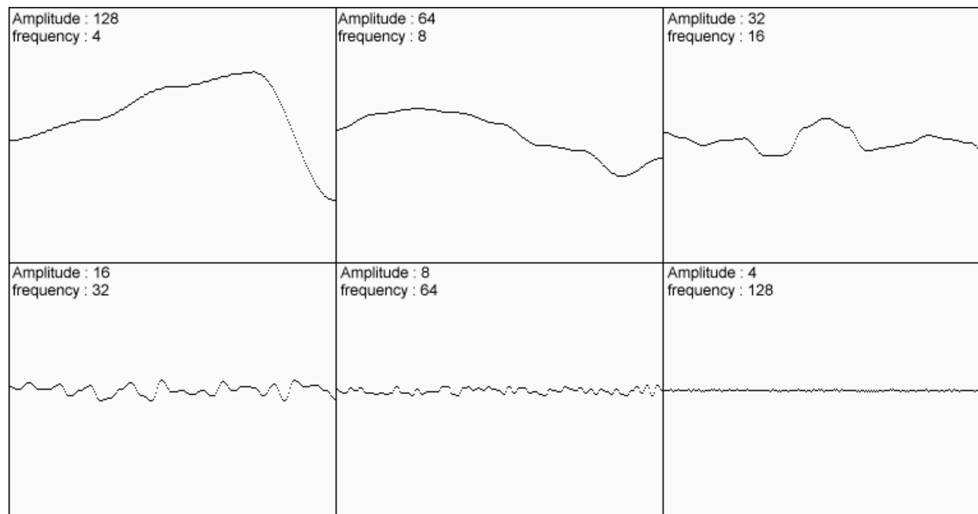
Oktáva Jedna z koherentních šumových funkcí. Počet oktáv říká, kolik funkcí skládáme pro výsledný Perlinův šum. Ovlivňuje detailnost výsledků.

Amplituda Maximální absolutní hodnota, kterou oktáva může produkovat, ovlivňuje výšku.

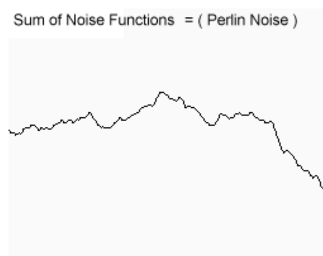
Frekvence Počet cyklů na jednotku délky, které produkuje oktáva.

Persistence Ovlivňuje jak rychle klesají amplitudy oktáv. Pro každou oktávu vyjadřujeme její amplitudu jako: $octave_i : amplitude = persistance^i$.

Lakunárnost (Lacunarity) Ovlivňuje jak rychle rostou frekvence oktáv. Pro každou oktávu můžeme vyjádřit její frekvenci následujícím vzorcem: $octave_i : frequency = lacunarity^i$.



(a) Jednorozměrné Perlinovy šumové funkce s různými parametry



(b) Součet funkcí z obrázku 4.1a

Obrázek 4.1: Perlinův šum.

4.3 Proměnlivost světa

V navrženém modulu se vyhneme modifikaci terénu za běhu hry, jelikož by to vyžadovalo příliš složitý a komplexní systém. Protože terén není hlavním tématem této práce, vystačíme si pro testovací účely s faktem, že samotné procedurální generování světa zaručuje, že pathfinding nebude připravený předem, ale bude vždy spočítaný agentem ad hoc.

Místo toho bude proměnlivost světa za běhu hry ovlivněna simulací flóry – čas od času se objeví stromy, které budou růst a budou plnit funkci překážek, květiny a tráva, které budou hlavním zdrojem potravy pro býložravce.

4.4 Vliv na AI

Pochopitelně, dynamický svět nejvíc ovlivňuje systém pohybu agenta. To, že svět není připraven předem, nutí agenta propočítávat cesty v run-time a zakazuje použití předem definovaných možných tras. Výkonnostně je to náročnější:

pro nalezení cesty agent musí nejdřív zmapovat okolí a až pak může použít jeden ze zmíněných v kapitole 3 algoritmů.

Kapitola 5

Unity 3D engine

Tato kapitola stručně popíše herní engine Unity 3D a vysvětlí, proč právě Unity 3D bylo zvoleno pro implementaci modulu simulace chování namísto jiných stejně výkonných herních enginů.

5.1 Stručný přehled enginu

Unity 3D je moderní, rychle se rozvíjející a velice populární mezi vývojáři videoher. Unity 3D bylo vydáno v roce 2005 společnosti Unity Technologies a v rozpětí dalších deseti let se stalo jedním z nejrozšířenějších herních enginů po celém světě.

Unity 3D nabízí bohaté vlastnosti, rychle se rozvíjející funkcionalitu a podporu pro více platforem. Podporuje různé předrenderované izometrické perspektivy prostředí a jeho široká škála nástrojů a snadné rozhraní dělá z Unity 3D jeden z nejpoužívanějších enginů pro vytvoření jakékoliv videohry. Výzkum analytické agentury Statista ukázal, že už v roce 2014 téměř 65 % herních vývojářů ve Velké Británii dávalo přednost Unity 3D před alternativami [29].

Samozřejmě má Unity 3D i své nevýhody. V následujícím seznamu se pokusíme shrnout klady a zápory tohoto enginu:

Klady:

- Populární volba mezi moderními videoherními vývojáři, z toho plynoucí velká podpora komunity.
- Dobré licenční podmínky v herním průmyslu.
- Snadné použití.
- Kompatibilní s téměř každou herní platformou.
- Rozsáhlá dokumentace a velký počet oficiálních příruček.
- Vhodné pro rychlé prototypování a snadný design.
- Oficiální vzdělávací akce po celém světě.

- Časté updaty přinášející novou funkcionalitu.

Zápory:

- Omezená sada nástrojů.
- Časté velké problémy s výkonem.
- Dokumentace, i přes svůj rozsah, má v některých místech značné nedostatky.
- Stále vysoký počet chyb.
- Časově náročné na výrobu AAA her se složitými a různorodými efekty.

5.2 Vymezení klíčových pojmů v Unity 3D

V této části budou definovány některé klíčové pojmy použité v této diplomové práci. Jsou nezbytné pro pochopení různých pracovních procesů a postupů v Unity 3D a také některých klíčových bodů této práce.

Projekt.
Projektem jsou myšleny všechny zdrojové soubory, se kterými pracuje vývojář. To jsou assety, herní skripty a rozšíření editoru, různé specifické soubory Unity (scény, prefaby).

Hra.
Hrou označujeme konečný výsledek projektů – zkompilovanou a spustitelnou aplikaci.

Scéna.
Scéna je, jednoduše řečeno, kolekce objektů, které se používají jako celek. V kontextu videoher můžeme koukat na scénu jako na úroveň nebo lokaci hry. Načtením scény odstraníme nepotřebné herní objekty z aktuální scény a načteme novou s jiným rozložením objektů.

Asset.
Asset je nějaký zdroj používaný ve hře – sprite, zvuk, textura, animace, 3D model, shader, doslova cokoliv, co se používá ve hře a může být *znovupoužito* vývojářem. Z assetů lze skládat balíčky (z anglického packages) a importovat sadu několika assetů do jiného projektu v Unity 3D. Výsledný modul této práce představuje právě sadu assetů, kterou bude možné importovat a používat v jiných projektech nezávisle na jejich struktuře.

Herní objekt (GameObject).
Základní třída všech entit, které mají instanci ve scéně.

Komponenta.
Komponenta je třída ve jmenném prostoru UnityEngine dědící ze třídy Object (nejedná se o klasickou třídu `object` obsaženou v C#, ale o třídu Object obsaženou taktéž ve výše uvedeném jmenném prostoru). Jedná se o základní třídu pro všechny skripty, které můžou být připojené k herním objektům.

Prefab.
Typ assetů umožnující kompletní uložení herního objektu spolu s jeho komponentami a vlastnostmi. Prefab pak představuje šablonu, na jejímž základě vývojář může vytvářet instance herních objektů ve scéně. Úprava vlastností konkretní instance bude mít vliv pouze na jeden upravovaný herní objekt, ale úprava vlastností prefabu ovlivní všechny herní objekty, které se zakládají na tomto prefabu.

Editor.
Editorem rozumíme spuštěnou instanci Unity 3D – v okně editoru probíhá nastavování scény, herních objektů, prefabů atp.

Inspektor.
Okno Inspektoru zobrazuje podrobné informace o aktuálně vybraném herním objektu v editoru. Zobrazuje všechny jeho připojené komponenty a jejich vlastnosti, což umožňuje vývojářům měnit a modifikovat funkčnost všech herních objektů ve scéně.

Kolider (Collider).
Kolidery jsou neviditelné komponenty, které definují tvar herního objektu pro účely fyzikálních kolizí. Tvar koliderу většinou neobkresluje přesně tvar objektu (takovýto přesný kolider se nazývá v angličtině mesh collider), ale bývá z výkonnostních důvodů zjednodušen. Kupříkladu kolider stromu se může skládat z jednoho kvádru pro kmen a koule pro korunu. Koliderу lze nastavit vlastnosti jako materiál (kov, guma. . .), zda se kolider bude v průběhu hraní hýbat či zda má fungovat pouze jako spínač (trigger), tj. přes takový kolider mohou objekty volně prostupovat, ale spustí přitom nějakou akci. Tyto akce a celkově chování, které má kolider spouštět, se definují v kódu metodami `OnCollisionEnter`, `OnCollisionStay` a `OnCollisionExit`. Pro spínače jsou metody identické, pouze místo slova `Collision` v názvech metod je použito slovo `Trigger`.

Kapitola 6

Cílové chování

6.1 Zvířata

Ve výsledném modulu budou zvířata logicky rozdělena na 2 skupiny: masožravci a býložravci. Jedinec z první skupiny bude označen jako predátor a bude se živit pouze živočichy, které uloví sám. Druhy zvířat uvnitř skupin budeme rozdělovat primárně podle životních statistik popsaných níže.

Tématem teto práce je simulace chování, ne animování a 3D modelování fauny. Proto je vizuální reprezentace řešena pouze modelem krychle a každá z podskupin zvířat má specifickou barvu. Případné pozdější nasazení skutečných animovaných 3D modelů bude fungovat s navrženým modulem a vyžadovat pouze triviální úpravy kódu pro podporu správného přepínání parametrů přidaného animačního kontroléru.

6.2 Životní statistiky

U všech zvířat budeme sledovat 3 základní statistiky:

Zdraví.
Nejdůležitější parametr života zvířete. Pokles jeho hodnoty upravuje toleranci nepohodlí při hledání cest a má celkově velký vliv na chování: zvíře se primárně zaměří na způsoby záchrany vlastního života a zvýšení hodnoty zdraví.

Hlad.
Očekávaně upravuje chování tak, že se zvíře pokusí najít jídlo podle preferovaného druhu potravy – začne lovit ostatní zvířata nebo bude hledat rostliny kolem sebe.

Únava
Nízká hodnota tohoto parametru donutí zvíře najít pohodlné a bezpečné

místo na odpočinek. Úroveň únavy má přímý vliv na rychlost pohybu zvířete.

Navrhovaný modul bude implementovat rozhodovací část inteligentního chování podle popsaných statistik třemi způsoby: pomocí FSM, behaviorálních stromů a GOAP systému. Klíčový teoretický popis všech tří přístupů byl popsán v rešeršní části 2. Detaily tykající se implementačních kroků se nalézají v příslušných kapitolách každého způsobu v implementační části práce.

Hlavní důraz při testování bude položen na uvěřitelnost chování: zvířata by měla adekvátně reagovat na změny životních funkcí, mimořádné události a změny ve svém okolí. Jejích chování by mělo být jednoduše vysvětlitelné pro uživatelé, logické a očekávané.

Část II

Implementační část

Kapitola 7

Zvolené technologie

7.1 Unity 3D

Pro vývoj byla použita verze 5.4.1f1. Testování však probíhalo také na novější verzi 5.5.0p4.

7.2 Programovací jazyk

Engine Unity nabízí podporu UnityScriptu (modifikovaný JavaScript), C# a Boo. Pro implementaci cílového modulu byl zvolen C#.

7.3 Prostředí

Jako vývojové prostředí bylo použito Visual Studio Enterprise 2015.

7.4 Použité assety

Při implementaci byly použity následující sady assetů:

- “Low Poly: Free Pack“ od AxeyWorks[1]: modely použité pro generování prostředí (stromy, kameny, keře).

[1] https://www.assetstore.unity3d.com/en/#!/content/58821

Kapitola 8

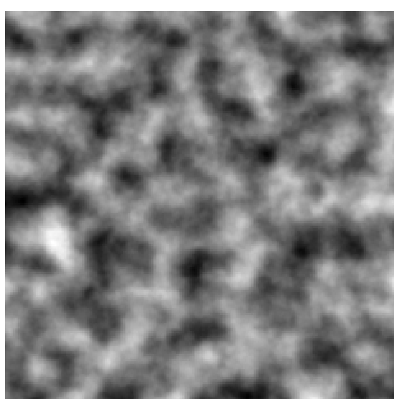
Generování terénu

8.1 Dynamické generování terénu

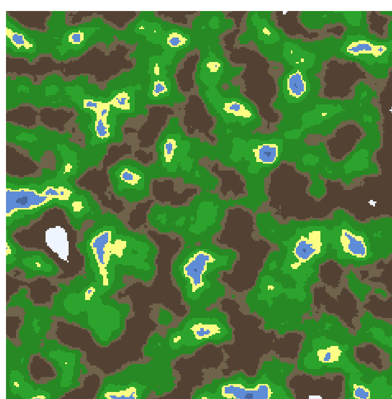
Implementace se zakládá na postupu doporučeném v sérii Sebastiana Lagua „Procedural Landmass Generation“ [11].

Modul generování terénu je implementován jako komponenta pro herní objekt v Unity 3D. Implementace prochází několika etapami, počínaje generováním výškové mapy pomocí Perlinova šumu podle hodnot nastavených uživatelem. Vygenerovaná mapa byla poté obarvena podle definovaných výškových regionů. Mezivýsledek teto etapy je zobrazen na obrázku 8.1.

Pro účely testování byly okraje mapy omezené. Místo nekonečného terénu byl tedy vygenerován ostrov. Toto omezení bylo implementováno generováním tzv. falloff mapy a použitím její masky na vygenerovaný šum z prvního kroku (konkretně: odečtením hodnot falloff mapy od hodnot výškové mapy).



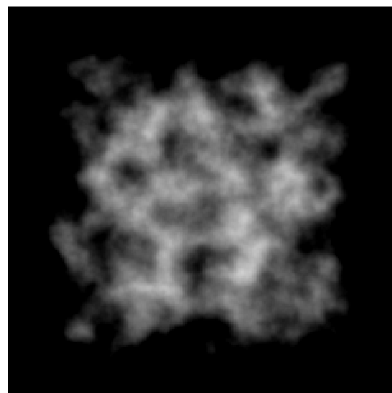
(a) Vygenerovaný Perlinův šum



(b) Barevná mapa regionů

Obrázek 8.1: První etapa generování terénu.

(a) Falloff mapa



(b) Perlinův šum s naloženou maskou falloff mapy

(c) Výsledná barevná mapa ostrova

Obrázek 8.2: Generování omezeného terénu pomocí falloff mapy.

Samotná falloff mapa se zakládá na následujícím vzorci:

$$f(x) = \frac{x^a}{x^a + (b - bx)^a}$$

Po několika iteracích testování byly zvoleny hodnoty $a = 3$ a $b = 2.2$, které dávaly nejlepší vizuální výsledek.

Tento krok je ilustrován na obrázku 8.2.

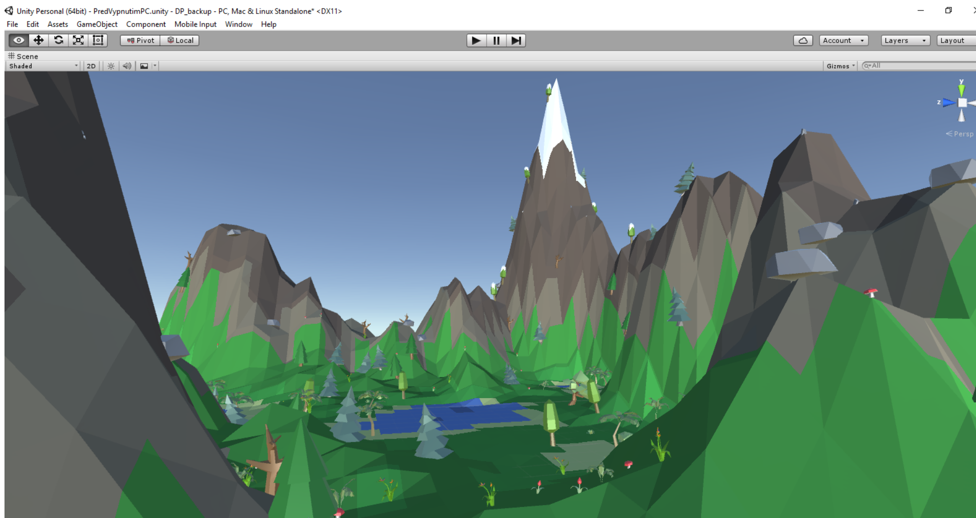
Třetím a finálním krokem v generování terénu byl převod 2D mapy do 3D světa a generování polygonové sítě (z angl. mesh) terénu stejného tvaru jako výsledný ostrov. Tato síť se také používá k detekování kolizí objektů.

Kromě toho v rámci vizuálních vylepšení výsledného terénu bylo naimplementováno konstantní neboli ploché stínování (z angl. flat shading). Výsledek této etapy je ukázán na obrázku 8.3.

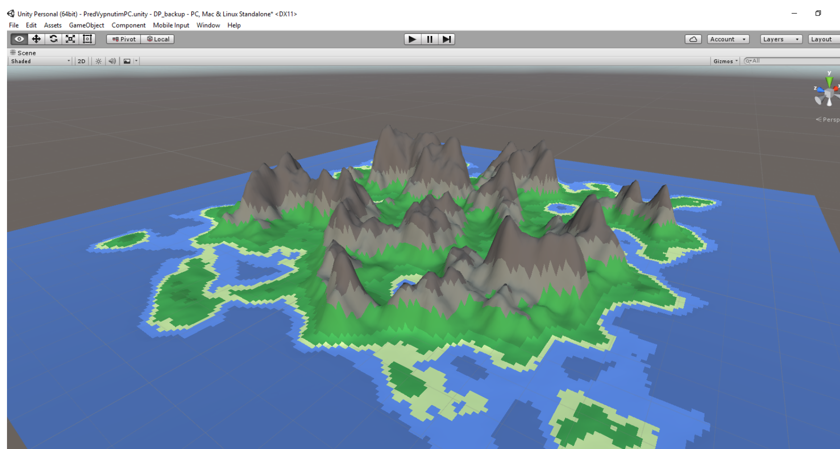
Využití flatshadingu vyžaduje zmenšení výsledného ostrova kvůli interním omezením Unity 3D: detailnější hrany terénu vyžaduji víc vrcholů v polygonové síti, ale maximální podporovaný počet je 65534.

8.2 Naplnění světa

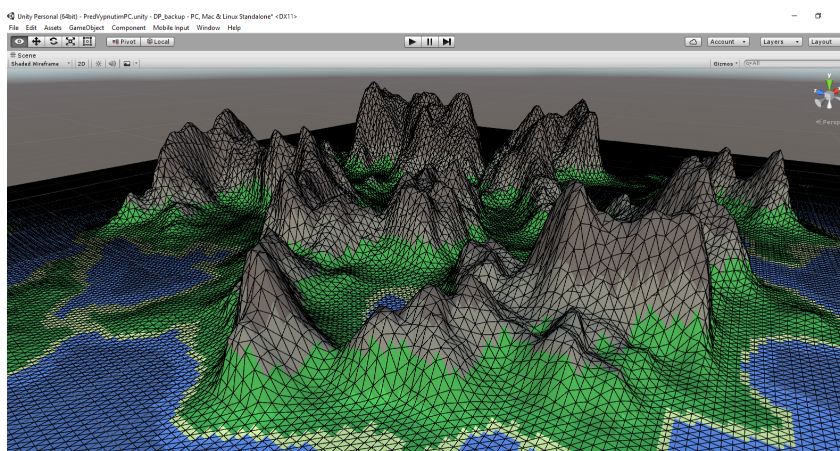
Vytváření objektů v generovaném světě, stejně jako samotné generování terénu, se zakládá na Perlinově šumu. Každý definovaný region obsahuje jednoduše rozšiřitelný seznam prefabů, které se mohou vyskytnout v konkretním biotopu. Přesná světová pozice pro nový objekt, hlavně výška, je vypočítána z polygonové sítě terénu, primárně z pole vrcholů sítě. Příklad dynamického naplnění světa objekty je ilustrován na obrázku 8.4. Aby zvířata nezemřela hlady, na základě vnitřního časovače se automaticky do světa generuje pro býložravce jídlo a pro predátory jiná zvířata.



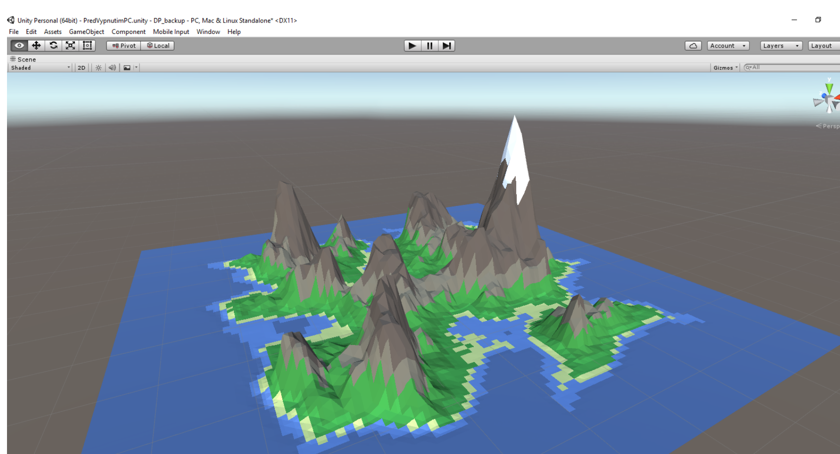
Obrázek 8.4: Ukázka generovaného terénu s objekty.



(a) Příklad výsledného terén



(b) Polygonová síť



(c) Flat shading

Obrázek 8.3: Výsledek generování terénu.

Kapitola 9

Pathfinding

9.1 Implementace A*

Jak již bylo zmíněno v sekci 3.2.2.1, klíčovým bodem při hledání cesty pomocí A* algoritmu je funkce ve tvaru $f(u) = g(u) + h(u)$, kde $g(u)$ představuje vzdálenost od počátku a $h(u)$ tvoří vzdálenost od u k cíli t. Čím blíž jsme k cíli, tím vyšší hodnotu má $g(u)$ a tím nižší hodnotu má $h(u)$.

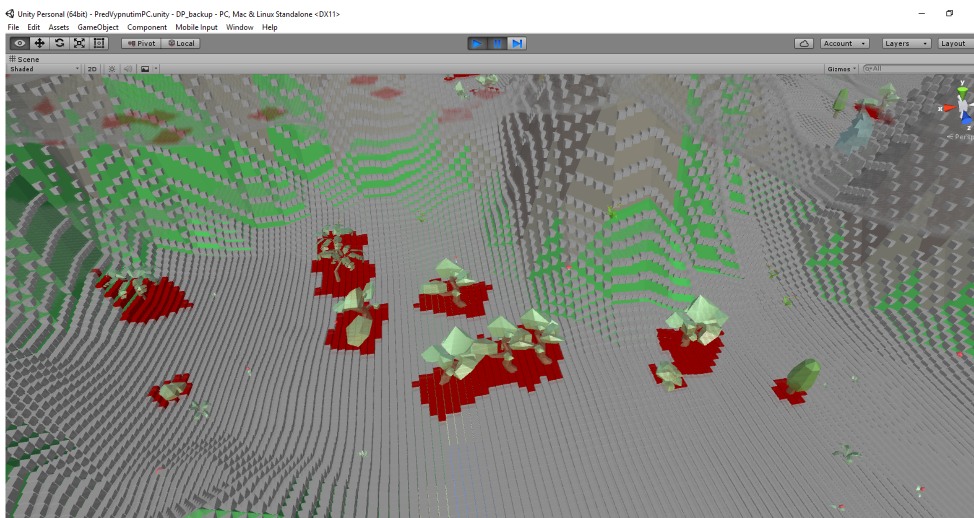
Při implementaci byly zvoleny následující orientační hodnoty: 10 pro pohyb v horizontálním nebo vertikálním směru a 14 pro pohyb ve směru diagonálním. Tato čísla vycházejí z jednotkové veličiny nodů: pokud bychom zvolili 1 jako cenu ortogonálního pohybu (horizontálně nebo vertikálně), pak by se cena diagonálního rovnala odmocnině ze dvou. Abychom se vyhnuli nutnosti počítat odmocniny při každém hledání cesty, běžnou praxí je zaokrouhlit odmocninu a použít desítkové násobky hodnot [12].

Systém hledání cest, podobně jako modul generující terén, je implementován jako komponenta herního objektu, který se nachází ve scéně. Tato komponenta podporuje uživatelské nastavení velikosti mřížky a velikosti jednotlivých nodů. Každé z těchto nastavení má pochopitelně výrazný vliv na výkon: vetší mřížka s vetším počtem nodů vyžaduje více času na hledání cesty.

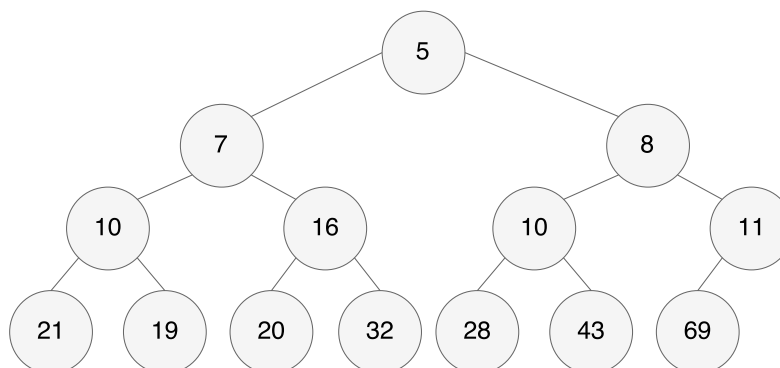
Ve výsledném modulu pro testovací ostrov velikosti 900x900 byla použita hodnota poloměru 1. Na obrázku 9.1 je ukázána mřížka vygenerovaná na základě těchto hodnot. Šedivý čtverec je 3D zobrazení potenciálně přístupné světové pozice, červený znamená node, kam zvíře nemůže jít. Velikost takových nepřístupných překážek je vypočítána pomocí velikosti koliderů objektů.

9.2 Optimalizace

Nejpomalejší část A*, jako u většiny prohledávacích algoritmů [21], spočívala v prohledávaní celého seznamu OPEN při hledání nodů s nejnižší hodnotou $f(u)$. Právě proto bylo rozhodnuto změnit její strukturu na haldu, konkrétněji na haldu typu Min-Heap.



Obrázek 9.1: Ukázka vygenerované mřížky pro hledání cest s neprůchozími nody.



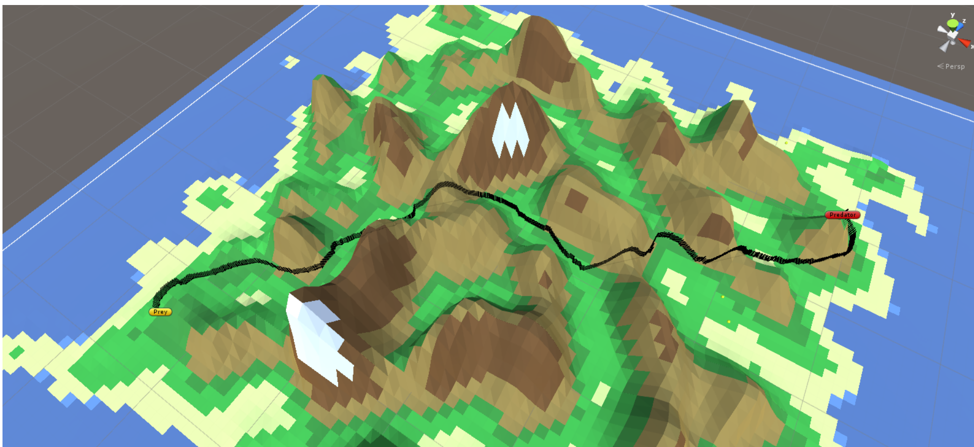
Obrázek 9.2: Halda

Jedná se o stromovou datovou strukturu, kde platí, že prvek s nejnižší hodnotou je vždy v kořenu stromu, a každý rodičovský uzel má vždy menší hodnotu než oba jeho potomci. Názorná ukázka struktury haldy je na obrázku 9.2. V případě haldy z obrázku, pokud bychom chtěli přidat nový prvek s hodnotou 6, stačilo by provést pouze 3 porovnání místo N porovnání pro N-prvkovou množinu jako v první iteraci implementace.

Tato optimalizace výrazně zlepšila výkon modulu pří hledání cest.

9.3 Penalizace

Pro vytváření pocitu přirozeného pohybu při hledání cest byl implementován penalizační systém. Na mřížku, kterou vytváří Pathfinding komponenta, je



Obrázek 9.3: Preferovaná cesta s použitím penalizačního systému.

aplikována výšková mapa z modulu generování terénu a výška nodu se započítává do ceny pohybu přes něj. Výsledkem tohoto kroku je to, že zvířata preferují cesty přes biomy s nevysokým modifikátorem výšky terénu a jejich pohyb přes mapu vypadá pak přirozeněji: místo toho, aby zvíře šplhalo přes horu, se ji pokusí obejít. Pokud je život zvířete v ohrožení, bude mít vyšší toleranci terénové penalizace.

Ukázka zmíněného systému je ilustrována na obrázku 9.3.

Kapitola 10

Struktura rozhodovací logiky

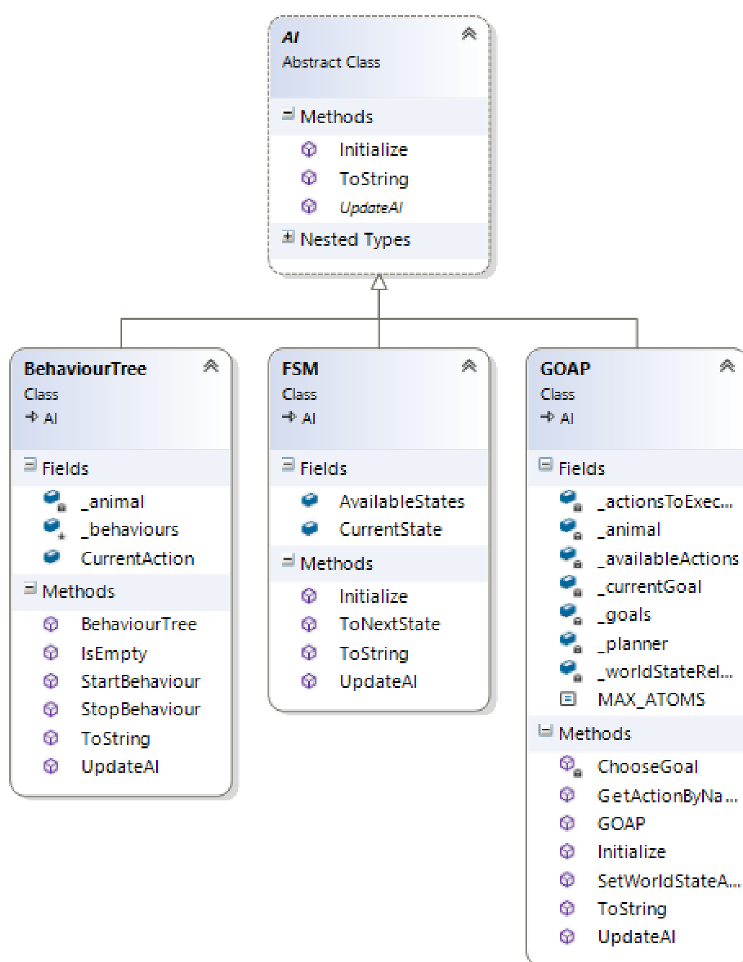
Jelikož je rozhodovací logika hlavním tématem této práce a rozhodující složkou v uvěřitelnosti chování AI agentů, právě tato část byla implementovaná několika způsoby pro porovnání v praxi.

Implementační struktura je zobrazena na obrázku 10.1. Třída `Animal`, představující zvíře, volá každý frame update na aktuálně použitou architekturu rozhodovací logiky (řádek číslo 10 v ukázkovém kódu 10.1).

Kód 10.1: Ukázka kódu kde každý frame probíhá update chování

```
1        /// <summary>
2        /// Update is called every frame
3        /// </summary>
4        void Update()
5        {
6            if (IsAlive())
7            {
8                StateText.text = String.Format("HP: {0} \n Energy: {1} \n
                      Hunger: {2} \n Current State: {3}, \n Target Food:
                      {4}", HP, Energy, Hunger, _ai.ToString(), TargetFood
                      == null ? "nothing" : TargetFood.gameObject.name);
9
10               _ai.UpdateAI();
11               HungerUpdate();
12               EnergyUpdate();
13               HpUpdate();
14           }
15       }
```

Architektura rozhodovací logiky může být přepnuta za běhu modulu pro všechna zvířata najednou nebo pro každé konkretní zvíře zvlášť pomocí dropdown menu na komponentě herního objektu zvířete. Tento proces je popsán v uživatelské příručce v části B.7.



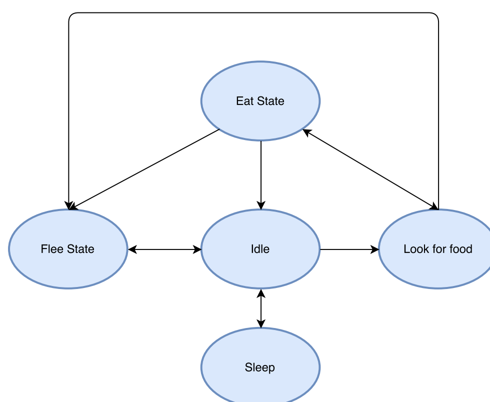
Obrázek 10.1: Implementace rozhodovací logiky.

Kapitola 11

Konečné automaty

11.1 Návrh

Zjednodušená ukázka implementovaného konečného automat chování pro býložravce je znázorněna na obrázku 11.1. FSM chování predátora obsahuje další stavy (například stav Hunt: Lov, během kterého predátor hledá ostatní zvířata).



Obrázek 11.1: Zjednodušená ukázka konečného automatu chování býložravců.

11.2 Implementace

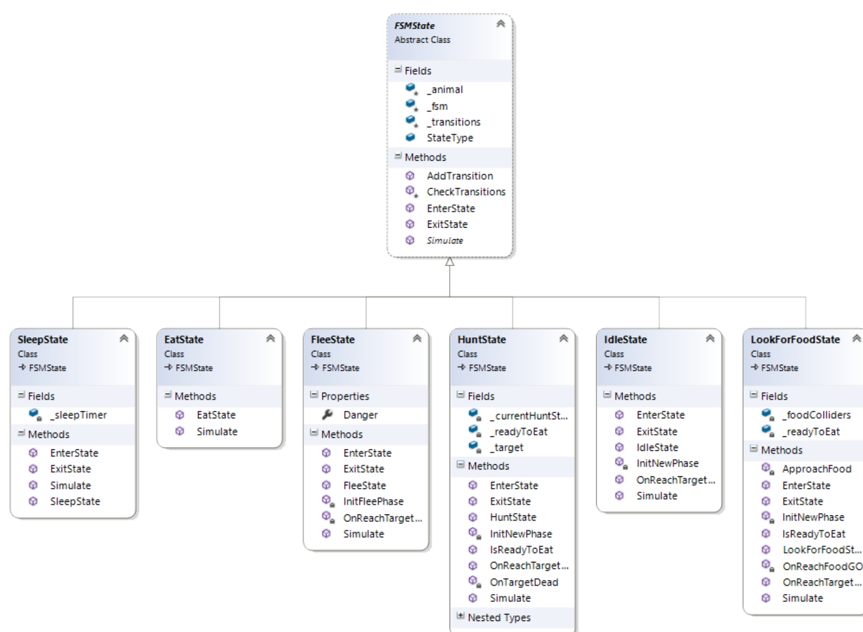
Klíčovou třídou je `FSM` implementující již zmíněnou třídu `AI`. `FSM` udržuje seznam povolených stavů s jejich přechody a, samozřejmě, aktuální prováděný stav `CurrentState`. Úkolem této třídy je provést simulaci aktuálního stavu a poskytnout na vyžádání instanci dalšího stavu. Metoda `UpdateAI()`, již zmíněná v ukázce 10.1, je velice stručná:

Kód 11.1: Ukázka kódu odpovídajícího za update chování v FSM

```
1    public override void UpdateAI()
2   {
3       CurrentState.Simulate();
4   }
```

Z diagramu tříd stavů na obrázku 11.2 vidíme, že každý stav implementuje abstraktní třídu `FSMState` a hlavně její metodu `Simulate()`, která definuje samotný charakter stavu.

Každý stav kontroluje ve své metodě `Simulate()` podmínky přechodu do jiných stavů a v případě jejích nesplnění vykonává dále kroky svého chování.



Obrázek 11.2: Diagram tříd stavů implementovaného konečného automatu.

11.2.1 Poznámky k implementaci

Jak již bylo zmíněno v rešeršní části práce, největším problémem FSM architektury je nutnost definovat každý stav a přechod z něj ručně v kódu. Důležitou roli hraje pořadí zadávaných přechodů, na základě nichž se pak vyhodnocuje aktuální stav zvířete. Toto pořadí má kritický vliv na rychlost reakcí a pochopitelně i na uvěřitelnost agenta. I pro tak malý a relativně jednoduchý konečný automat je definování stavů a přechodů velice nepřehledné a náchylné k chybám, což ilustruje ukázka kódu 11.2.

Kód 11.2: Ukázka definování přechodů a stavů pro zvíře

```
1       protected virtual void InitFSM()
2       {
```

```
3            _ai = new FSM();
4            var fsm = _ai as FSM;
5
6            var idleState = new IdleState(fsm, this);
7            var eatState = new EatState(fsm, this);
8            var lookForFoodState = new LookForFoodState(fsm, this);
9            var sleepState = new SleepState(fsm, this);
10           var fleeState = new FleeState(fsm, this);
11
12           idleState.AddTransition(new StateTransition(fleeState,
13               () => { return DangerDetected() != null; }));
14
15           idleState.AddTransition(new StateTransition(lookForFoodState,
16               () => { return IsHungry; }));
17
18           idleState.AddTransition(new StateTransition(sleepState,
19               () => { return IsTired; }));
20
21           lookForFoodState.AddTransition(new StateTransition(eatState,
22              () => { return lookForFoodState.IsReadyToEat(); }));
23
24           lookForFoodState.AddTransition(new StateTransition(fleeState,
25              () => { return DangerDetected(); }));
26
27           eatState.AddTransition(new StateTransition(fleeState,
28              () => { return DangerDetected(); }));
29
30           eatState.AddTransition(new StateTransition(lookForFoodState,
31               () => { return TargetFood == null; }));
32
33           eatState.AddTransition(new StateTransition(idleState,
34               () => { return HasEatenEnough; }));
35
36           sleepState.AddTransition(new StateTransition(idleState,
37                () => { return HasSleptEnough || IsHungry || WasAttacked;
                      }));
38
39           fleeState.AddTransition(new StateTransition(idleState,
40               () => { return fleeState.Danger == null; }));
41
42           fsm.AvailableStates.Add(idleState);
43           fsm.AvailableStates.Add(lookForFoodState);
44           fsm.AvailableStates.Add(eatState);
45           fsm.AvailableStates.Add(sleepState);
46           fsm.AvailableStates.Add(fleeState);
47       }
```

11.3 Testování a ladění

Testy byly prováděny skupinou 5 lidí. Během testování byly odhaleny menší nesrovnalosti pathfindingu a několik chybějících přechodů mezi stavy; všechny problémy však byly opraveny po nahlášení.

Testování implementovaného modulu probíhalo formou pozorování chování v předpřipravených stavech zvířete: hladové, v nebezpečí, unavené a kombinace těchto podmínek. Všechny životní charakteristiky jsou nastavitelné i za běhu aplikace pro každé zvíře zvlášť. Podle příslušných změn hned zvíře upravuje chování. Všechny testy očekávaných reakcí zvířete na změny životních funkcí byly úspěšně splněny.

Důležitým testovacím milníkem bylo zkontrolovat, jestli se na chování zvířete projeví, že v navrženém automatu nejsou podporované přechody mezi všemi stavy. „Centrálním" stavem je `Idle`, kde zvíře nemá žádné potřeby, které musí uspokojit, a pouze se prochází generovaným ostrovem. Při testování byl položen důraz na to, zda přechody z některých stavů do jiných přes `Idle` budou mít viditelný vliv na chování a budou kazit dojem inteligence zvířete. Očekávané chování při selhání testu by bylo takové, že by se zvíře zastavilo, přepočítalo novou cestu (a případně by začalo přesun do nejbližšího bodu na naplánované trase) a až potom by přešlo do skutečně požadovaného stavu. Selhání testu by vyžadovalo jiný návrh konečného automatu, kde každý stav má přímý přechod do všech potřebných stavů, což by zhoršilo i tak nepřehledné definování stavů a přechodů v kódu.

Všichni testeři potvrdili, že přechod mezi stavy vyžadujícími `Idle` jako mezistav není možné zpozorovat a změna stavů vypadá z pohledu uživatele naprosto plynule. Tato skutečnost je dána tím, že simulace stavů, a tím pádem i kontrola podmínek přechodů, probíhá každý frame. Toto potenciální problémové místo úspěšně prošlo testem.

Kapitola 12

Behaviorální stromy

12.1 Návrh

Na začátku implementace byla navržena struktura stejného chování jako v FSM pro býložravce (obrázek 12.1) a predátora (obrázek 12.2). Chování klíčových prvků zobrazených na diagramu je popsáno v analýze behaviorálních stromů (viz. 2.3.0.1 sekce rešeršní kapitoly). Sekvenční selektory na těchto obrazcích kontrolují potomky směrem shora dolů.

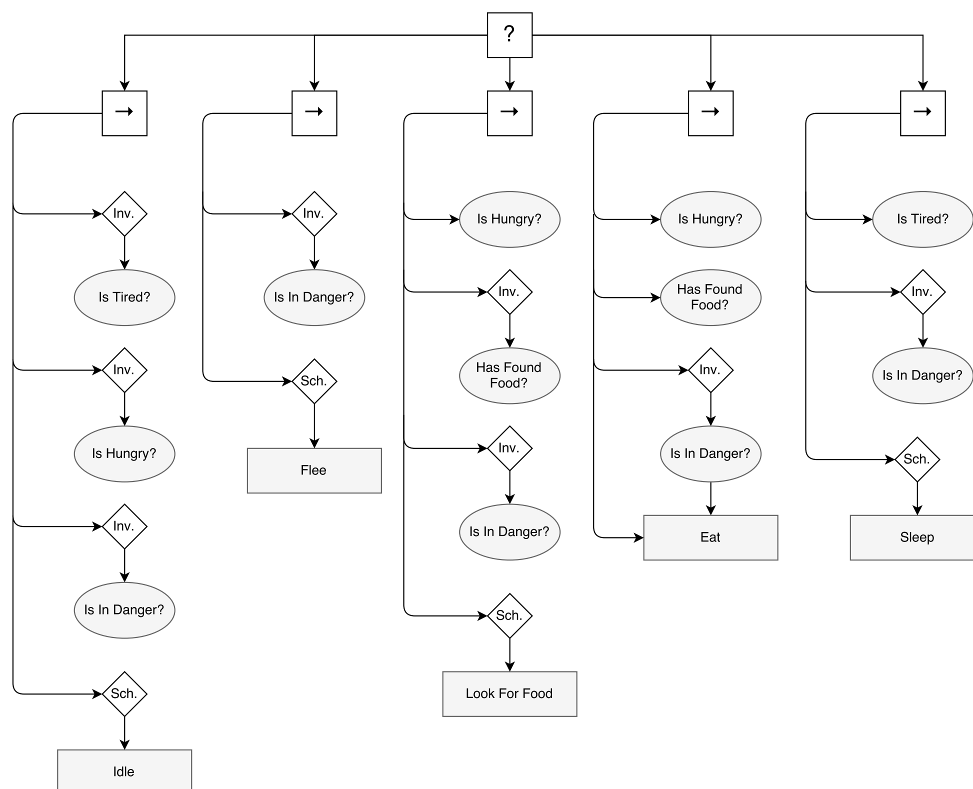
12.2 Implementace

Diagram tříd klíčových prvků behaviorálních stromů popsaných v rešeršní části práci je znázorněn na obrázku 12.3.

Stejně jako v kapitole o FSM, i zde si ukážeme, jak probíhá update chování v behaviorálních stromech. Tato metoda je uvedena v ukázce 12.1. Procházení stromu chování začíná vždy od kořene, kterým je v této práci hlavní selektor. Algoritmus zavolá tick na tento selektor, který, v souladu se svou definicí, zavolá tick na svého prvního potomka. Tímto začne zpracování a vyhodnocení prvního podstromu. Podobným způsobem každý z členů tohoto podstromu propaguje ticky dále dolů ve své hierarchii. Pokud v této větvi jeden z prvků vrátil jiný výsledek než RUNNING (typicky FAILURE, nebo SUCCESS), vrátí se tento výsledek prvnímu předkovi, který ví, jak jej zpracovat.

Na základě výsledku, jenž obdržel kořenový selektor stromu, je rozhodnuto následující chování:

RUNNING
: Pokud se z podstromu do kořenového selektoru vrátil RUNNING stav, UpdateAI končí a v dalším framu pošle tick pouze selektorům a sekvencím tohoto podstromu, v nichž je stále udržován index naposledy prováděného RUNNING potomka. Tím je zajištěn rychlý návrat do aktuálně prováděného uzlu.



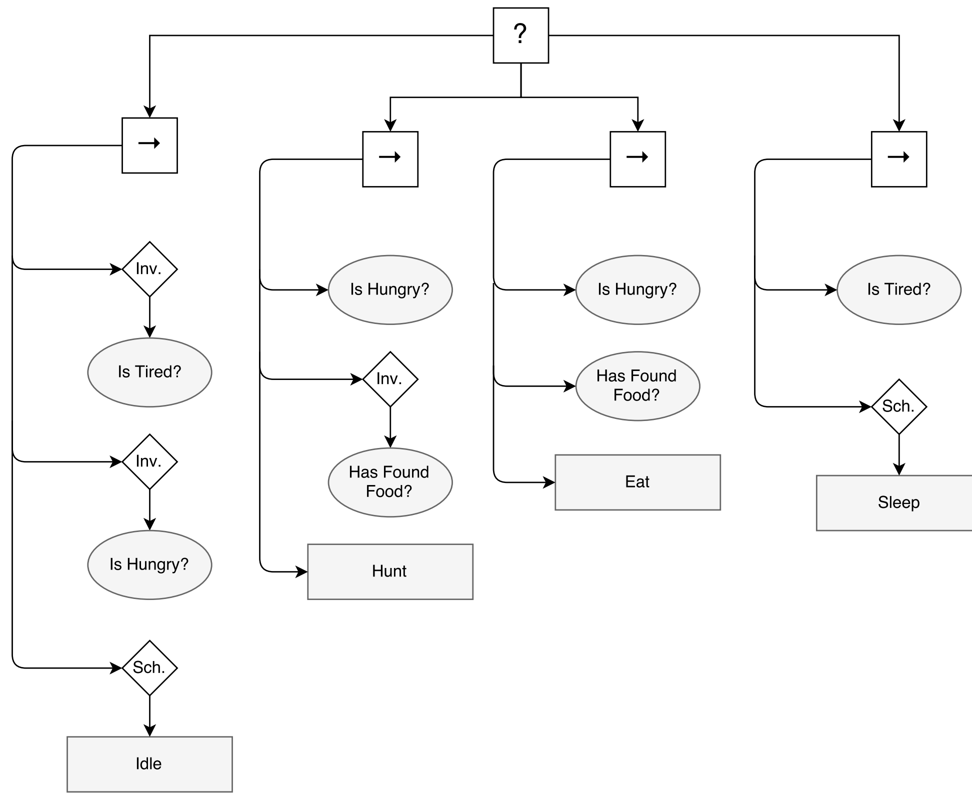
Obrázek 12.1: Ukázka návrhu behaviorálního stromu pro základní chování zvířat.

SUCCESS
: Vrátil-li se z podstromu do kořene stav SUCCESS, strom je resetován a v dalším framu znovu vyhodnocen.

FAILURE
: Pokud se z podstromu vrátil FAILURE stav, selektor ve stejném framu pokračuje tím, že posílá tick do dalšího dostupného podstromu.

Kód 12.1: Ukázka Update() metody v behaviorálních stromech

```
1   public override void UpdateAI()
2   {
3     if (IsEmpty())
4     {
5       _animal.StartBT();
6     }
7
8     for (int i = 0; i < _behaviours.Count; ++i)
9     {
10      CurrentAction = _behaviours[i];
```

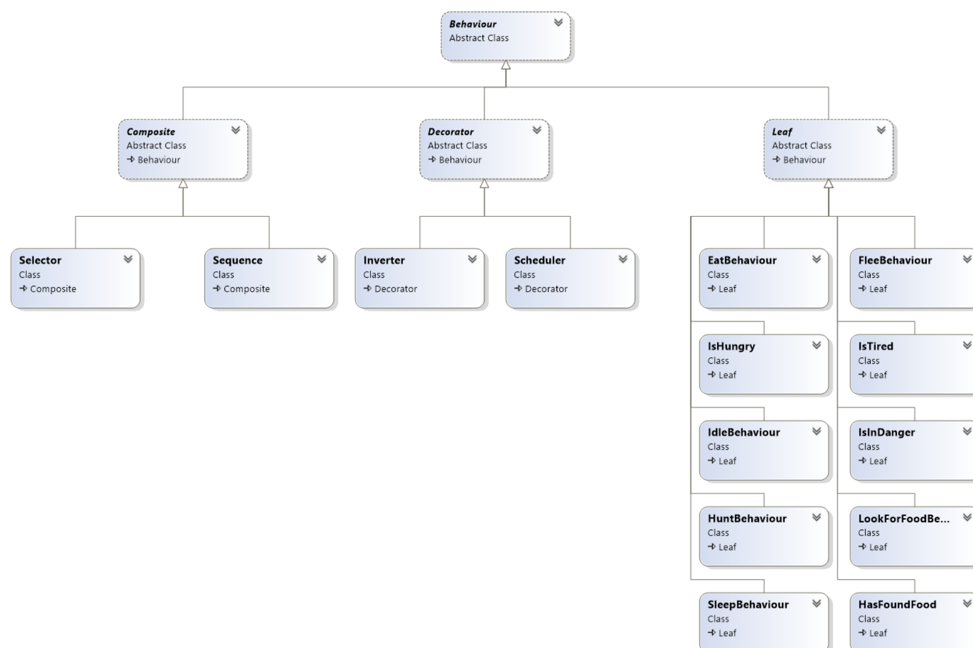


Obrázek 12.2: Ukázka návrhu behaviorálního stromu specifického chování predátorů.

```
11
12          CurrentAction.Tick();
13          if (CurrentAction.State != BehaviourState.Running)
14          {
15            StopBehaviour(CurrentAction, CurrentAction.State);
16          }
17        }
18        RemoveBehaviours();
19      }
```

12.2.1 Dekorátor: Scheduler

Problémem je stav, kdy některý z listů vrací RUNNING (například Idle stav, kdy zvíře bezcílně bloumá po okolí v situacích, kdy není ohroženo, nemá hlad, ani není unavené). Jelikož se v každém zpracování stromu rychle vracíme do RUNNING listu, je potřeba nějakým způsobem kontrolovat, zda již některá z podmínek není porušena. Jedním z možných řešeních je do `Update()` metody každého listu dát podmínky, při jejichž splnění se má vracet jiný stav. Tento přístup nicméně přináší nevýhodu, kdy bychom při přidání nového chování



Obrázek 12.3: Diagram tříd klíčových prvků behaviorálního stromu.
.

museli ve všech ostatních relevantních listech přidat novou podmínku: právě toto chování je největším nedostatkem konečných automatů a behaviorální stromy řeší tento problém správně navrženou vnitřní strukturou.

Byl proto naimplementován dekorátor nazvaný scheduler, kterému lze nastavit časový interval, v němž má znovu překontrolovat podmínky, které obyčejně kontroluje sekvence při prvním průchodu podstromem. Pokud časovač uzná, že je čas na kontrolu, překontroluje zadané podmínky a je-li alespoň jedna nesplněna, dekorátor vrátí FAILURE. Ve všech ostatních případech je klasicky zpracováno dítě dekorátora a vrácen jeho výsledek.

12.3 Testování a ladění

Testování rozhodovacích logik implementovaných pomocí BT probíhalo podobným způsobem jako v FSM. Opět byly pozorováním skupiny testerů otestovány očekávané reakce na změny životních funkcí a jejich kombinace a také chování zvířat ve specifických podmínkách terénu.

Během ladění a testování bylo objeveno několik nedokonalostí navržené struktury behaviorálních stromů. Jako výsledek analýzy testů byla odhalena nutnost zavedení nového dekorátoru scheduler již zmíněného v sekci 12.2.1 a následný design jiné struktury stromů. Kromě toho byly doplněny chybějící podmínky u některých sekvencí.

Po opravě zmíněných problémů byly všechny testy chování ohodnoceny jako úspěšné.

Kapitola 13

GOAP

Třída `GOAP` obsahuje hlavní udaje potřebné pro plánování chování, již zmíněné v kapitole 2.5.1: seznam možných akcí, aktuální stav světa a seznam potenciálních cílů. Uvedeme nejdůležitější implementační poznámky ke každému z těchto bodů.

13.1 Návrh a implementace

Již zmíněná třída `GOAP`, stejně jako třídy `FSM` nebo `BT`, implementuje abstraktní třídu `AI`.

13.1.0.1 Atomické stavy světa

Stavy světa představují nejdůležitější informace pro zvíře: pokles jeho životních funkcí pod kritickou hodnotu, objevená nebezpečí v blízkosti zvířete a podobné klíčové události.

Stav světa je uložen jako bitové pole a změny jeho stavu, simulování nebo provedení akce jsou implementované jako bitové operace. Všechny atomické stavy světa jsou definované ve třídě `World` jako `enum`, kde každá z položek má numerickou hodnotu mocniny dvojky.

Kód 13.1: Zjednodušená ukázka definice atomických stavů světa

```
1    [Flags]
2    public enum WorldAtomicState
3    {
4        IS_HUNGRY = 1,
5        FOUND_FOOD = 2,
6        IN_DANGER = 4
7    }
```

13.1.0.2 Akce

Každá akce dědící ze třídy `GOAPAction` obsahuje bitové pole efektů a předpokladů a taktéž si pamatuje bity ovlivněné těmito potenciálními světovými změnami. Akce může být očekávaně ukončená pro pokračování do dalších akcí podle plánu po vypršení nějakého časovače (například akce *Idle*, kde zvíře se prochází světem bez nějakého konkretního cíle) nebo po nastavení Success flagu akce (například akce *LookForFood*, kde zvíře hledá jídlo). Kromě toho akce může být neočekávaně přerušena kvůli mimořádným událostem, které popíšeme níž.

Třída `GOAPGoal` představuje potenciální cíle, které může zvíře provádět, a skládá se z očekávaného stavu světa popsaného maskou atomických stavů – bitů, které mají být cílem ovlivněné – a názvů pro přehlednější práci s cílem.

13.1.0.3 Cíle

Volba nového cíle probíhá vždy po dosažení současného cíle nebo v případě mimořádných události: cíl „Najíst se“ bude přerušen a nahrazen cílem „Dostat se do bezpečí“, pokud zvíře objeví predátora v nebezpečné blízkosti. Mimořádná událost zastaví aktuální akci, zruší připravený plán a sestaví nový v souladu s důležitějším cílem.

Samotný výběr nových cílů probíhá vyhodnocováním seznamu všech možných cílů zvířete podle definovaných preferencí s ohledem na aktuální stav životních funkcí.

13.1.0.4 Plánování

Samotné plánování se provádí ve třídě `GOAPPlanner`, která implementuje prohledávání prostoru akcí pomocí progresivního A* a vrací každému agentovi seznam akcí pro dosažené cíle. Plánovač sestavuje posloupnost akcí tak, že z výchozího stavu světa (aktuálního stavu zvířete uchovávaného jako bitové flagy) zkouší nasimulovat svět po provedení všech potenciálně možných akcí v tomto stavu, počínajíc nejvýhodnější. Algoritmus pokračuje, dokud mu nedojdou proveditelné akce nebo dokud simulovaný stav světa nebude stejný jako očekávaný stav světa v hledaném cíli. Pak algoritmus zrekonstruuje posloupnost akcí, přes které se do očekávaného stavu dostal, a vrátí ji zvířeti.

13.1.1 Update

Stejně jako v implementačním popisu předchozích dvou kapitol si ukážeme, jak probíhá Update v GOAP systému:

Kód 13.2: Zjednodušená ukázka definice atomických stavů světa

```
1       public override void UpdateAI()
2       {
3           if (_currentGoal == null)
```

```
4           {
5               //init
6               ChooseGoal();
7           }
8
9           CheckCurrentGoal();
10          if (_actionsToExecute.Count != 0)
11          {
12              var action = _actionsToExecute.First();
13
14              if ((action.WaitForSuccess && action.Success) || (!action.
                    WaitForSuccess && action.IsFinished()))
15              {
16                  //a new action (if any) will start execution next
                        frame
17                  action.ExitAction();
18                  ApplyActionEffects(action);
19                  _actionsToExecute.Remove(action);
20              }
21              else
22                  action.Execute();
23          }
24          else
25          {
26              ChooseGoal();
27          }
28      }
```

13.2 Možné optimalizace

Případným optimalizačním krokem by bylo regresivní prohledávání prostoru akcí [19], kdy cílový stav je začátkem hledání a aktuální stav zvířete je podmínkou ukončení prohledávání.

Nicméně, výrazné zlepšení by bylo znatelné pouze pro velký počet potenciálních akcí a cílů (řádově stovky akcí). V případě implementovaného modulu se počet pohybuje kolem několika desítek, což se nijak zásadně neprojevuje na výkonu.

13.3 Testování a ladění

Testování plánovače bylo zaměřeno hlavně na správnost nalezené posloupnosti akcí. Proto testování probíhalo pro zjednodušení na testovacím seznamu akcí: ne všechny se dostaly do finální verze modulu. Například bylo použito několik stejných akcí `LookForFood`, které se lišily pouze cenou provedení nebo jedním z předpokladů (například tím, že zvíře už předtím nalezlo jídlo atp.). Zavedení těchto akcí bylo zapříčiněno potřebou kontroly, že zvíře skutečně zvolilo nejoptimálnější cestu k dosažení cílového světového stavu. Tento krok garantoval správný výsledný plán.

Po odstranění menších nalezených problémů a zbytečných testovacích stavů začalo testování samotného chování. Tato etapa probíhala stejným způsobem jako u konečných automatů a behaviorálních stromů: skupina testerů sledovala chování zvířat a jejich reakce na změny životních statistik nebo svého okolí. Kromě toho bylo stejně jako u FSM a BT otestováno chování na specifickém terénu. Všechny testy byly úspěšné.

Kapitola **14**

Porovnání dosažených výsledků

Celkově, nejen při implementaci, ale i při testování, byl rozdíl mezi zvolenými přístupy velice výrazný. U procedurálních způsobů (obzvlášť FSM) mohl uživatel v každý okamžik jednoznačně říct, jak se bude zvíře chovat dál. Deklarativní přístup plánovače sice dával neméně důvěryhodné výsledky, ale byl trochu složitější na testování, jelikož uživatel mohl pouze předpokládat, jakou cestu zvolí zvíře pro dosažení cílů.

Nejlépe působící metriky jsou nastaveny na prefabech v implementaci a jsou uživatelům poskytnuty volně k libovolným úpravám.

Důležitým výsledkem této práce je, že finální chování zvířat implementované všemi způsoby je ve své podstatě stejné: všechna splnila test věrohodnosti a správně reagovala na změny světa nebo životních statistik. Je možné dosáhnout jakéhokoliv požadovaného výsledku všemi způsoby. Důležité je brát v úvahu, že některé přístupy řeší určité typy klasických problémů herních AI lépe a některé hůř. Volba způsobu implementace rozhodovací logiky pro herního agenta je hodně propojená s typem úloh, které bude agent řešit, okolním světem, žánrem hry, velikosti očekávaného stavového prostoru agenta a spoustou dalších faktorů.

Závěr

Cílemi práce bylo provést rešerši algoritmů umělé inteligence používané v současných počítačových hrách se zaměřením na simulaci inteligentního chování zvířat v dynamickém herním prostředí, nastudovat a popsat herní engine Unity 3D, navrhnout a implementovat herní modul využívající vybrané algoritmy, na případových studiích otestovat výsledné chování simulovaných zvířat, definovat a vyhodnotit vhodné metriky jejich inteligentního chování a na závěr otestovat a zdokumentovat výsledný modul.

Všechny tyto cíle byly splněny, přičemž hlavní důraz byl kladen na rešerši a implementaci algoritmů umělé inteligence. Krom toho je v této práci za pomoci Perlinova šumu naimplementováno procedurální generování terénu, v němž se zvířata pohybují, a entit vyskytujících se ve vygenerovaném světě, což zajišťuje dynamickou složku herního světa.

Umělá inteligence je rozdělena na rozhodovací logiku a hledání cest. Hledání cest je implementováno algoritmem A*, rozhodovací logika je vytvořena třemi způsoby – konečnými automaty, rozhodovacími stromy a cílově orientovaným plánováním. Tato práce všechny tyto algoritmy a mnohé další pro hledání cest i rozhodovací logiku důkladně popisuje v rešerši, ostatním vývojářům tedy nabízí komplexní pohled na svět umělé inteligence ve videohrách.

Výsledky práce ukázaly, že samotný výběr rozhodovací logiky příliš neovlivňuje chování umělé inteligence navenek. Hlavním rozdílem je především práce pro vývojáře, neboť samotný výběr logiky je spjat spíše s ambicióznosti výsledného projektu.

Při čtení této práce je velmi důležité si uvědomit, že přestože jak akademická, tak herní umělá inteligence sdílejí název, ve skutečnosti je na oba druhy pohlíženo ze zcela odlišného úhlu – akademický úhel pohledu se snaží vytvořit skutečně živého a přemýšlejícího tvora, zatímco hry se snaží vytvořit pouze takový dojem, neboť pro hráče by dokonalá umělá inteligence mohla být předvídatelná a často i frustrující, což je v rozporu s hlavními záměry her. V drtivé většině případů je tedy volen suboptimální přístup.

Implementovaný výsledný modul lze importovat do rozpracovaných pro-

jektů v enginu Unity 3D, kde bude s minimálními úpravami pro kompatibilitu použitelný. Modul nabízí chování na základě důležitých životních statistik, další chování je snadno rozšiřitelné.

Tato diplomová práce je přínosem pro akademické čtenáře, kteří se chtějí dozvědět, jakým způsobem jsou implementovány známé algoritmy v herní praxi, pro herní vývojáře, kteří si chtějí udělat komplexní představu o umělé inteligenci, ale i pro ostatní čtenáře, kteří se prostě o toto téma zajímají. Psaní práce bylo ve výsledku velmi zajímavé i pro mě, neboť jsem se v rámci výzkumu dozvěděla spoustu informací, ke kterým bych se jinak nedostala.

Literatura

[1] Barták, R.: *Umělá inteligence II: Racionální rozhodování, teorie užitku. [online].* [cit. 2017-02-07]. Dostupné z: `http://kti.ms.mff.cuni.cz/~bartak/ui2/lectures/lecture06.pdf`

[2] Champandard, A.: Understanding the Second Generation of Behavior Trees and Preparing for Challenges Beyond. AltDevConf 2012, únor 2012.

[3] Champandard, A. J.: *Planning in Games: An Overview and Lessons Learned [online].* [cit. 2017-02-01]. Dostupné z: `http://aigamedev.com/open/review/planning-in-games/`

[4] DeLoura, M. A.: *Game Programming Gems.* číslo sv. 1 in Charles River Media graphics, Charles River Media, 2000, ISBN 9781584500490.

[5] Erol, K.; Hendler, J.; Nau, D. S.: Semantics for Hierarchical Task-network Planning. Technická zpráva, University of Maryland, College Park, MD, USA, 1994.

[6] Humphreys, T.: Exploring HTN Planners through Example. In *Game AI Pro : collected wisdom of game AI professionals*, editace S. Rabin, kapitola 12, Boca Raton: CRC Press, 2014, ISBN 1466565969, s. 149–167.

[7] Katedra obecné lingvistiky, Filozofická fakulta Univerzity Palackého v Olomouci: *Analýza přirozeného jazyka (podklady k výuce) [online].* [cit. 2017-02-03]. Dostupné z: `http://oltk.upol.cz/isoj/data/uploads/vyukove-materialy-ke-staze/ka_01_analyza_prirozeneho_jazyka_podklady_k_vyuce_1.compressed.pdf`

[8] Kolář, J.: *Teoretická informatika.* České vysoké učení technické v Praze, 2009, ISBN 978-80-01-04331-8.

[9] Kotek, Z.; Mařík, V.; Zdráhal, Z.: Metody rozpoznávání a umělá inteligence. *Kybernetika ve výzkumu a výuce*, 1983: s. 16–30.

[10] Kovář, V.: *Umělá inteligence a filosofie [online]*. [cit. 2017-02-03]. Dostupné z: https://nlp.fi.muni.cz/uui/referaty2005/kovar_vojtech/uif_prezentace.pdf

[11] Lague, S.: *Procedural Landmass Generation: tutorial series*. leden 2016.

[12] Lester, P.: *A* Pathfinding for Beginners [online]*. 2005, [cit. 2017-01-20]. Dostupné z: http://www.policyalmanac.org/games/aStarTutorial.htm

[13] Mařík, V.; Štěpánková, O.; Lažanskỳ, J.: *Umělá inteligence*. Umělá inteligence, sv. 1, Academia, 1993, ISBN 9788020004963.

[14] Mark, D.: *AI Architectures: A Culinary Guide [online], [cit. 05-01-2017]*. říjen 2012. Dostupné z: http://intrinsicalgorithm.com/IAonAI/2012/11/ai-architectures-a-culinary-guide-gdmag-article/

[15] Marzinotto, A.; Colledanchise, M.; Smith, C.; aj.: Towards a Unified Behavior Trees Framework for Robot Control. *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, květen 2014: s. 5420–5427.

[16] Millington, I.: *Artificial Intelligence for Games (The Morgan Kaufmann Series in Interactive 3D Technology)*. CRC Press, 2006, ISBN 0124977820.

[17] Nareyek, A.: AI in Computer Games. *Queue*, ročník 1, č. 10, únor 2004: s. 58–65, ISSN 1542-7730, doi:10.1145/971564.971593. Dostupné z: http://doi.acm.org/10.1145/971564.971593

[18] Nilsson, N. J.: *Principles of artificial intelligence*. Tioga Pub. Co, 1980, ISBN 0935382011.

[19] Orkin, J.: Applying goal-oriented action planning to games. *AI Game Programming Wisdom*, ročník 2, č. 2004, 2004: s. 217–227.

[20] Orkin, J.: *Three states and a plan: the AI of FEAR [online], [cit. 03-01-2017]*. 2006. Dostupné z: http://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf

[21] Patel, A.: *Introduction to A* [online]*. [cit. 2017-01-25]. Dostupné z: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

[22] Pereira, R.: *An Introduction to Behavior Trees – Part 1 [online], [cit. 12-12-2016]*. červen 2014. Dostupné z: http://guineashots.com/2014/07/25/an-introduction-to-behavior-trees-part-1/

[23] Rabin, S.: *AI Game Programming Wisdom*. Rockland, MA, USA: Charles River Media, Inc., 2002, ISBN 1584500778.

[24] Rasmussen, J.: *Are Behavior Trees a Thing of the Past [online]*. duben 2016, [cit. 2017-02-05]. Dostupné z: www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.php

[25] Rich, E.; Knight, K.: *Artificial Intelligence*. Artificial Intelligence Series, McGraw-Hill, 1991, ISBN 9780070522633.

[26] Russell, S.; Norvig, P.: *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2002, ISBN 0137903952.

[27] Russell, S. J.: Rationality and Intelligence. *Artificial Intelligence*, ročník 94, 1997: s. 57–77.

[28] Searle, J. R.: Minds, brains, and programs. *Behavioral and Brain Sciences*, ročník 3, 1980: s. 417–424.

[29] Statista.com: *What game engine do you currently use? [online]*. listopad 2014, [cit. 2017-01-29]. Dostupné z: http://www.statista.com/statistics/321059/game-engines-used-by-video-game-developers-uk

[30] Surynek, P.: *Plánování a rozvrhování: Hierarchické plánování. [online]*. [cit. 2017-02-08]. Dostupné z: http://ktiml.mff.cuni.cz/~bartak/planovani/lectures/HTN.pdf

[31] Tišnovský, P.: *Perlinova šumová funkce a její aplikace [online], [cit. 01-01-2017]*. březen 2007. Dostupné z: https://www.root.cz/clanky/perlinova-sumova-funkce-a-jeji-aplikace/

[32] Turing, A. M.: Computing Machinery and Intelligence [online], [cit. 24-12-2016]. *Mind*, ročník 59, č. 236, 1950: s. 433–460, ISSN 00264423, 14602113. Dostupné z: http://www.jstor.org/stable/2251299

[33] Unity Technologies: *Unity User Manual: Building a NavMesh [online]*. [cit. 2017-01-31]. Dostupné z: https://docs.unity3d.com/Manual/nav-BuildingNavMesh.html

[34] Ögren, P.: Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees. *AIAA conference on Guidance, Navigation and Control, Minneapolis, MN*, srpen 2012.

Příloha A

Seznam použitých zkratek

AI Umělá inteligence, z anglického Artificial Intelligence.

BFS Prohledávání do šířky, z anglického Breadth-first search.

DFS Prohledávání do hloubky, z anglického Depth-first search.

FIFO First In, First Out, způsob vkládání do datové struktury a vyjímání z ní.

LIFO Last In, First Out, způsob vkládání do datové struktury a vyjímání z ní.

NPC Postava neovládaná hráčem, z anglického non-playable character.

AAA Označení špičkových titulů videoher s nejvyšším rozpočtem a velkou marketingovou propagací.

FSM Konečný automat, z anglického Finite state machine.

BT Behaviorální strom, z anglického Behaviour Tree.

GOAP Typ plánovače, z anglického Goal Oriented Action Planning.

HTN Hierarchická síť úloh, z anglického Hierarchical task network.

HLA Vysokoúrovňová akce, z anglického High-level action.

STRIPS Stanford Research Institute Problem Solver.

Příloha B

Uživatelská příručka

Popíšeme klíčové kroky práci s modulem.

B.1 Unity 3D

Pro vyzkoušení implementovaného modulu uživatel potřebuje mít nainstalovaný engine Unity 3D verze 5.4 nebo vyšší. Instalační soubor pro osobní edici, která je distribuována zdarma, lze stáhnout z oficiálních stránek: `https://store.unity.com/download?ref=personal`.

B.2 Import modulu do projektu

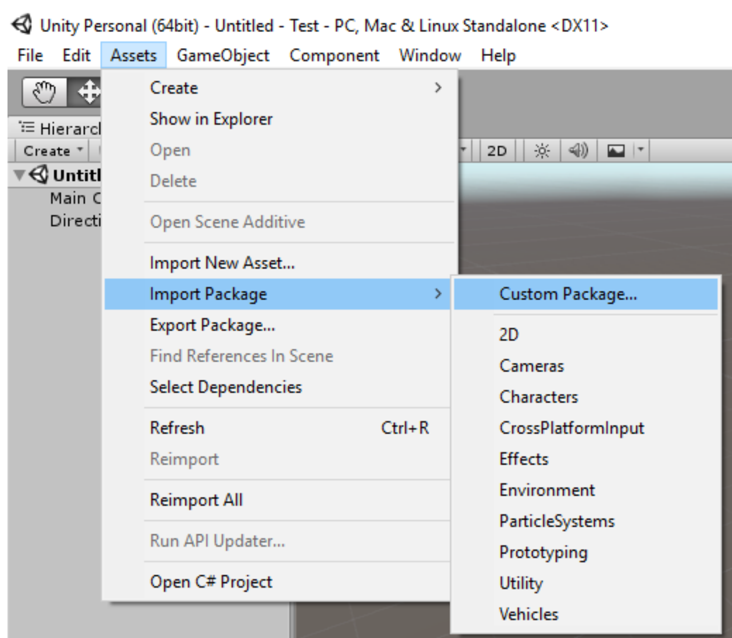
Importování výsledného modulu této práce probíhá způsobem doporučeným v dokumentaci enginu: uživatel zvolí v menu *Assets* → *ImportPackage* → *CustomPackage*. Tento krok je ilustrován na obrázku B.1. Dále uživatel zkontroluje, že všechny soubory modulu budou importované a potvrdí nastávající změny v projektu zmáčknutím OK.

B.3 Požadovaná hierarchie scény

Uživatel může použít předpřipravenou scénu `Showcase`, kterou otevře pomocí menu *File* → *OpenScene* a zvolí scénu `Showcase` ve složce *Scenes*.

Alternativně může uživatel založit vlastní scénu a přidat do ní následující objekty, nutné pro funkcionalitu:

- A*
- Terrain
- AIManager



Obrázek B.1: Import modulu do projektu

Všechny tyto objekty uživatel najde ve složce $Assets/ScenePrefabs$ a jejich instance ve scéně vytvoří přetažením prefabů do scény. Je nutné zkontrolovat všechny závislosti v komponentách.

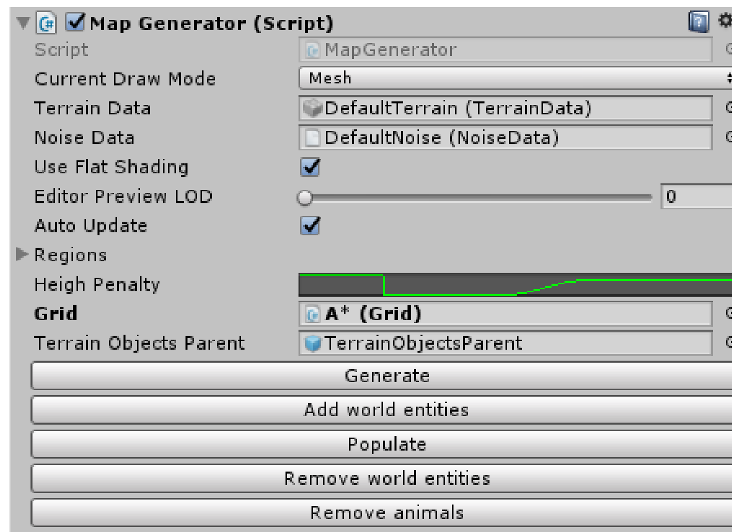
B.4 Generování terénu

Zodpovědnost za generování terénu nese herní objekt MapGenerator, který obsahuje komponentu `MapGenerator`. Požadované úpravy terénu a nastavení nových hodnot pro generovaní ostrovu je možné pomocí úprav assetů `DefaultNoise` a `DefaultTerrain`. Všechny nastavitelné hodnoty jsou popsané v kapitole 4. Samotné generování probíhá hned po změně jednoho z parametrů nebo po zmáčknutí tlačítka „Generate“ na zmíněné komponentě. Tato komponenta je zobrazena na obrázku B.2.

B.5 Naplnění světa objekty

Naplnění ostrova světovými entitami probíhá ve stejné komponentě `MapGenerator` stejného herního objektu. Pro přidání stromů, kamenů a ostatních ukázkových entit uživatel musí zmáčknout tlačítko „Add world entities“. Tlačítko „Remove world entities“ odstraní všechny herní objekty světových entit.

Definice samotných entit a jejich rozdělení podle biotopů se provádí ve stejně komponentě a nastavuje se při rozklepnutí pole „Regions“ definujícího



Obrázek B.2: Ukázka MapGenerator komponenty

barvu, hraniční výšku biomů a pole podporovaných entit pro daný region.

Jídlo se generuje náhodně po ostrovu podle vnitřního časovače.

B.6 Přidání zvířat

Za přidání a odstranění zvířat je zodpovědná stejná herní komponenta, která po zmáčknutí tlačítka „Populate“ náhodně umístí herní objekty reprezentující zvířata po ostrovu. Tlačítko „Remove all animals“ je odstraní ze scény.

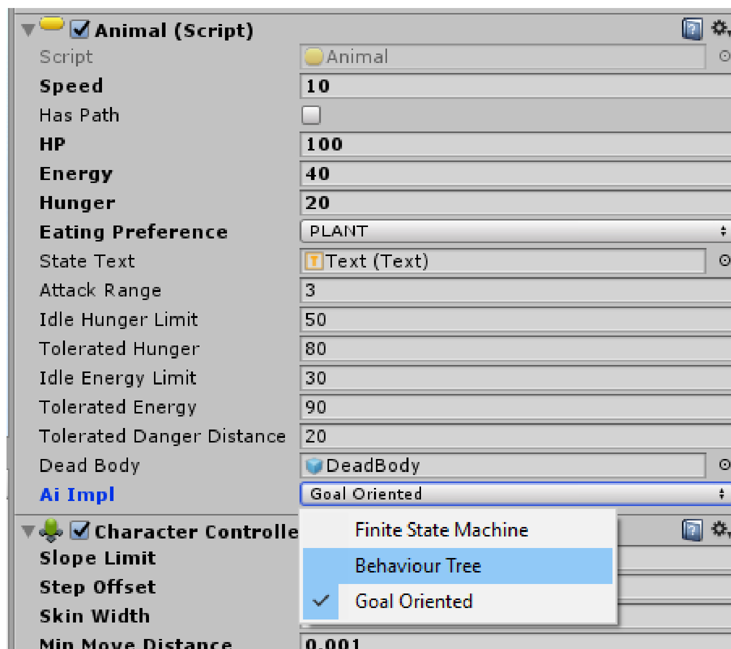
Kromě toho může uživatel manuálně umisťovat herní objekty zvířat přetažením prefabů na požadované místo scény.

B.7 Volba architektury chování

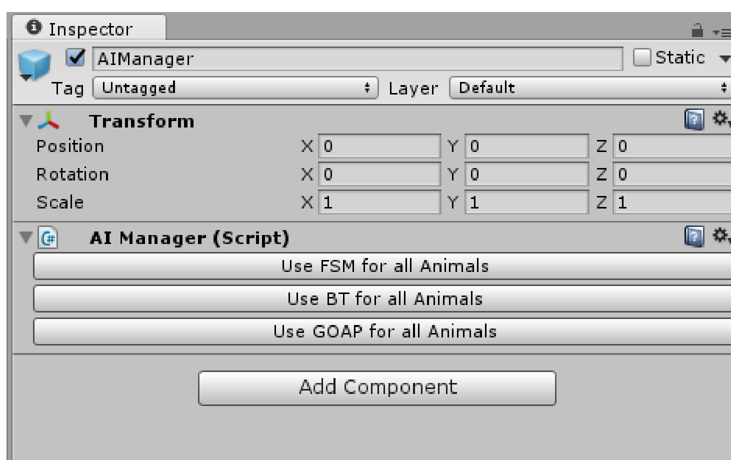
Proces změny logiky rozhodování je možný pro všechna zvířata ve scéně najednou nebo pro každé zvíře zvlášť. Přepínání chování všech zvířat najednou je možné po zvolení objektu `AIManager` ve scéně a zmáčknutí příslušného tlačítka. Chování konkretního zvířete může být nastaveno na jiný druh pomocí dropdown menu na komponentě `Animal` u každého zvířete zvlášť. Obě možnosti jsou zobrazeny na obrázcích B.3 a B.4.

B.8 Testování modulu

Pro finální testování modulu musí uživatel spustit scénu v editoru zmačknutím tlačítka „Play“.



Obrázek B.3: Komponenta Animal: Změna aktuální rozhodovací logiky zvířete.



Obrázek B.4: Komponenta AIManager: Změna rozhodovací logiky pro všechna zvířata ve scéně.

Příloha C

Obsah přiloženého CD

```
src
├── impl ........................ zdrojové kódy implementace
└── thesis ...................... zdrojová forma práce ve formátu LaTeX
text ............................ text práce
└── DP_Valentova_Xeniya_2017.pdf ... text práce ve formátu PDF
```