VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

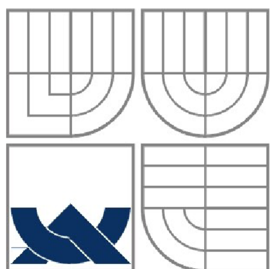
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SIMULACE KOLEKTIVNÍHO CHOVÁNÍ ENTIT VIRTUÁLNÍM SVĚTĚ

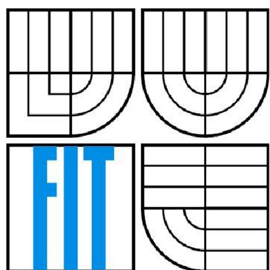
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE Tomáš Vymazal
AUTHOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SIMULACE KOLEKTIVNÍHO CHOVÁNÍ ENTIT VIRTUÁLNÍM SVĚTĚ

SIMULATION OF ENTITIES COLLECTIVE BEHAVOIR IN VIRTUAL WORLD

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
Tomáš Vymazal

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. Aleš Láník

BRNO 2011

Abstrakt

Tématem této práce je zhodnotit a porovnat možnosti řízení entit (agentů) ve virtuálním světě a jeden vybraný přístup realizovat ve formě programu. Pro implementaci byl vybrán přístup, kdy konečný automat, řídící agenta, podléhá evoluci pomocí genetických algoritmů. Tento přístup by měl přizpůsobit, případně vylepšit agentovo chování tak, aby odpovídalo zadaným požadavkům: zde např. aby se agent naučil težit zdroje ve virtuálním světě. Implementace je realizována pomocí modulu pro běh evoluce a pomocí modulu se 3d zobrazením, kde lze prohlížet chování agentů.

Abstract

Theme of this work is to evaluate and compare aviable paradigms for entity control in virtual worlds, and to implement one of these paradigms in application. Dynamic finite state machine upgraded using genetic algorithms has been chosen. This paradigm should make agent's behavior better and adapt agent to required state: i.e. make agent harvest resources in virtual world. Output of this work is application for running evolution and application for 3D view of agent's behavior.

Klíčová slova

agenti, virtuální svět, konečné automaty, evoluce, genetické algoritmy, 3d engine

Keywords

agents, virtual world, finite state machines, evolution, genetic algorithms, 3d engine

Citace

Tomáš Vymazal: SIMULACE KOLEKTIVNÍHO CHOVÁNÍ ENTIT VIRTUÁLNÍM SVĚTĚ
bakalářská práce, Brno, FIT VUT v Brně, 2011

SIMULACE KOLEKTIVNÍHO CHOVÁNÍ ENTIT VIRTUÁLNÍM SVĚTĚ

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Aleše Láníka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Tomáš Vymazal
11.5.2011

Poděkování

Rád bych na tomto místě poděkoval vedoucímu této práce, Ing. Aleši Láníkovi za jeho vedení a věcné rady, které mi v průběhu vypracování poskytl.

© Tomáš Vymazal, 2011

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah....1
1 Úvod....2
2 Inteligentní agenti....3
  2.1 Základní pojmy....3
  2.2 Možnosti řízení inteligentních agentů....10
  2.3 Genetické algoritmy a evoluční výpočetní techniky....17
3 Implementace....18
  3.1 Reprezentace světa, implementace v simulátoru....18
  3.2 Reprezentace znalostí agentů....23
  3.3 Návrh struktury FSM vhodné k evoluci....25
  3.4 Evoluce....31
  3.5 Zobrazení v režimu GUI....33
4 Závěr a porovnání ovládacích schémat agentů....36
  4.1 Boids a flocking....36
  4.2 Rozhodovací stromy....36
  4.3 Jednoduchá FSM....37
  4.4 Hierarchické FSM....38
  4.5 GravFSM....38
  4.6 Zhodnocení řídícího konceptu použitého v této práci....39
  3.1 Závěrečné zhodnocení práce....40
Literatura....42
Seznam příloh....43

1 Úvod

Autorovým úkolem v této práci bylo nastudovat možná řízení entit ve virtuálním světě a jedno z těchto řídících paradigmat implementovat. Pod pojmem virtuální svět si můžeme představit systém, který obsahuje určité objekty, podléhá určitým pravidlům a je v něm možné provádět různé interakce. Pojem entita můžeme pro zjednodušení zaměnit s pojmem agent tak, jak se používá na poli umělé inteligence. Jedná se tedy o samostatně funkční jednotku, která podle nějakého řídícího mechanismu operuje ve virtuálním světě, na základě událostí mění své vlastnosti a chování a mění také některé činitele ve virtuálním světě. Jako nejběžnější případ virtuálního světa s entitami si můžeme představit počítačovou hru, ve které se nachází nějaké město, obývané inteligentními bytostmi. V tom případě je virtuálním světem toto město a inteligentní bytosti jsou zmíněné entity.

Hlavní motivací bylo nastudovat možná řídící paradigmata pro ovládání entit ve virtuálním světě a jedno z těchto paradigmat implementovat. Autorovou hlavní motivací bylo prohloubit svoje znalosti v oblasti evolučního programování a toto zúročit při experimentech s ovládáním entit řízených pomocí konečných automatů. Evoluční přístup je v této práci aplikován tak, že chování entit je optimalizováno na nějakou požadovanou kvalitu, např. lepší schopnost natěžit zdroje, lépe ovládat boj s cizími entitami apod.

Následující kapitola se bude zabývat úvodem do problematiky inteligentních agentů, jejich řízení, virtuálními světy a evolučním přístupem k optimalizaci chování entit. Třetí kapitola shrnuje praktické znalosti a zkušenosti získané během vývoje a programování vybrané implementaci. Poslední kapitola zhodnocuje experimenty a nastudované vlastnosti různých ovládacích paradigmat, srovnává jejich vhodnost pro různé úlohy a také zhodnocuje výsledky této práce.

Hlavní částí práce byla implementace vybraného přístupu k řízení agentů. Byla vybrána cesta použití jednodušších konečných automatů, které jsou však naprosto editovatelné a upravitelné evolučními úpravami, zejména mutací a křížením. Pro tento přístup byla vybrána a navržena linearizovaná struktura konečného automatu a některé jeho vlastnosti, které umožní použít na FSM jakoukoliv operaci evoluce, přičemž vždy je z hlediska sémantiky FSM výsledkem validní řešení. Konkrétní implementace byla vytvořena v podmínkách virtuálního světa, kde si agenti musí hledat cestu rozsáhlým terénem pomocí pathfinding techniky, vyhýbat se překážkám, bojovat s cizímy kmeny a snažit se natěžit zdroje pro svůj kmen. Programová část má dva módy, kdy jeden slouží k evolučnímu zlepšování vlastností agentů a druhá slouží k prohlížení jejich chování v 3D režimu.

2 Inteligentní agenti

2.1 Základní pojmy

2.1.1 Agent

Agent je výpočetní proces s jedním centrem řízení a s určitým (lokálním) cílem. Agenti koordinují hlavně své znalosti, cíle, schopnosti a plány tak, aby je mohli využívat společně při akcích a řešení úloh. Agenti v multi agentním systému mohou ve své činnosti směřovat k jednomu globálnímu cíli nebo k jednotlivým odděleným cílům, které spolu mohou, ale nemusí souviset. Agenti sdílejí znalosti o úlohách a řešeních [1].

Obdobně můžeme v této práci obměnit definici agenta a doplnit ji následovně: agenti v této práci se shlukují do kmenů, kde agent má v rámci kmene za úkol maximalizaci vytěžených zdrojů ve virtuálním světě, případně usmrcení agentů ostatních kmenů. Dále můžeme podotknout, že agenti nemají pevné řízení, ale toto se díky simulované evoluci mění - konverguje k lepším ovládacím strukturám (zde FSM).

U agentů je také důležitá schopnost komunikace a sdílení informací. Toto je možné implementovat různými způsoby. Často používaný způsob je tzv. sdílená tabule, což je paměť, do které jsou všichni agenti schopni zapisovat a zaráz z ní číst. Tento koncept je poté nutné doplnit o vlastní paměťovou implementaci, zda se např. jedná o asociativní paměť, nějakou formu databáze (relační, objektová), nebo např. stromová (hierarchická) struktura.

Druhým často používaným typem uložení znalostí agentů a jejich sdílení jsou soukromé (lokální) paměti jednotlivých agentů, tyto však musí být doplněné o schopnost kopírování svých částí jiným agentům. Zde však může vzniknout problém, a to sice rozdíl kontextu dat v paměti agenta A (původního) a agenta B (cílového). Informace v paměti původního agenta může obsahovat informace, které jsou validní jen v kontextu agenta A a pokud by je měl agent B využívat, buď to vůbec není možné, nebo by je bylo nutné opravit. Tento problém byl řešen i v této práci a jeho řešení bude uvedeno v kapitole věnované implementaci.

Inteligentní agenti musí být schopni získávat informace ze svého prostředí a také toto prostředí ovlivňovat - jinak by jejich existence nebyla opodstatněná.

V této souvislosti můžeme zavést další pojem, a to sice pojem reaktivní agent. Tento vždy reaguje pouze na podněty z vnějšího světa. Má k dispozici předem známou množinu akcí, které reprezentují např. nějaké stereotypní plány [1].

Hlavní nevýhodou tohoto typu agenta je úplná nebo částečná absence využití nějaké formy vnitřní paměti, která uchovává agentův model světa.

Dříve bylo uvedeno, že agent musí být schopen získávat informace ze svého prostředí. Jaké informace a jakého typu to jsou? Toto silně závisí na problémové doméně, ve které jsou agenti nasazeni. Pokud by např. agenti operovali v kontextu operačního systému, kde by měli za úkol optimalizovat jeho běh, mohly by mít tyto vstupní informace formu např. dat o bežících procesech, vytížení CPU, zabrané paměti apod. Pro účely této práce to však bude jiný případ. Zde agenti figurují v rámci virtuálního světa, který je do značné míry obrazem světa skutečného. Hraje zde výraznou roli spatiální informace (poloha) o ostatních objektech ve světě, ať už se jedná o objekty statické nebo dynamické (jiní agenti).

Agent musí být schopen informace získávát a pokud je jeho svět založen na spatiálním modelu, měl by být agent také omezen ve schopnosti tyto informace získávat, např. na bázi vzdálenosti nějakého jiného objektu, jeho relativní poloze vůči agentovu pohledu, zákrytu za jiným objektem, apod. K získávání informací o tom, jaké objekty jsou v agentově pohledovém kuželu (line of sight), tudíž zajištění omezení daných spatiální povahou agentova světa, je využit tzv. raycasting.

Inteligentní agent musí být také schopen svůj okolní svět ovlivňovat, případně ovlivňovat svoje postavení v něm. Pokud bychom měli opět použít analogii s operačním systémem, v něm by měli mít agenti schopnost měnit priority procesů, případně příliš náročné procesy ukončovat apod. V této práci je hlavním agentovým výrazovým prostředkem pohyb, přemístění, čili změna pozice (spatiální informace) relativně vůči ostatním objektům ve virtuálním světě. K dodržení omezení daných simulací virtuálního světa však agent nesmí a nemůže procházet pevnými objekty, nemůže se za nulový čas přesunout na velkou vzdálenost (“teleport”), má omezenou rychlost, musí se řídit tvarem a reliéfem povrchu, apod. Aby byl agent schopný dodržet omezení daná jeho světem a navíc byl schopný se v něm pohybovat, potřebuje mít schopnost hledat si v tomto světě cestu k danému cíli, pokud možno co nejkratší. Oblasti umělé inteligence, která se zabývá tímto problémem se říká pathfinding a bude mu věnovaná jedna z následujících podkapitol.

2.1.2 Agentův model světa

Inteligentní agent, jak již bylo rečeno, může být reaktivní, nebo může využívat pamět pro rozhodování o tom, co a jak bude ve virtuálním světě dělat. Do této paměti si např. agent může ukládat důležité pozice ve virtuálním světě, pozice kde zahlédl nepřátele, kde je potrava, apod. Pokud bychom měli opět použít i analogii inteligentního agenta ze světa operačních systémů, zde by si agent mohl do paměti ukládat např. extrémní hodnoty využití CPU specifickými procesy. Toto by mu pak umožnilo lépe se rozhodnout o tom, který proces v případě většího vytížení systému ukončit, případně by mohl poslat zprávu (přes sdílenou pamět, tabuli, apod.) agentovi pro práci s oprávněními v systému, který by mohl upravit nastavení práv procesu.

Vraťme se ale k agentům v aktuální práci. V nadpisu je uvedena fráze "vnitřní model světa". Toto znamená, že aby agent byl schopný se sám rozhodovat, potřebuje mít informace o okolním světě, o interakcích v tomto světě, o možných akcích a potenciálních reakcích na ně. Ne vždy je potřeba ukládat všechny tyto informace, ale alespoň některé z nich by každý nereaktivní agent měl využívat. V této práci byli agenti obdařeni pamětí pro ukládání spatiálních informací o virtuálním světě, o agentech, které daný agent vidí, a o hodnotách charakterizujících samotného agenta. Odprosťme se teď od implementačních detailů a zaměřme se na to, jak by tato paměť měla být implementovaná. Již sme uvedli, že pamět a ukládání informací by mělo mít tu vlastnost, že pokud přeneseme část paměti do paměti jiného agenta (komunikace), bude tato přenesená informace stále použitelná a relevantní. Proto by informace v paměti měly být samopopisující se a vzhledem k agentovi i okolnímu světu absolutní. Pokud by nějaká informace v paměti agenta byla pro něho relativní (vůči němu), mohla by mít v paměti jiného agenta uplně jiný význam, což jistě není cílem - typickým příkladem je např. pozice. Za tímto účelem byla pamět rozdělená na 3 druhy:

- pozice objektů ve světě
- identifikátory ostatních agentů
- informace o aktuálním agentovi

V samotné agentově paměti je pak vytvořena hierarchie, která dává význam daným informacím. Tak např. pozice objektů ve světě jsou uloženy v oddělených seznamech, kde každý má specifický význam, např.:

- seznam pozic zdrojů ve světě
- seznam míst se zdroji
- domovská základna (seznam domovských základen)
- custom seznam, pro volný přístup
- apod.

Speciální pozici zde mají tzv. custom seznamy, do kterých si agent může volně pomocí speciálních instrukcí ukládat informace příslušejícího typu.

Díky tomu si již můžeme definovat 2 cesty, jak se informace do agentova modelu světa dostávají:

• agent si je zde uloží sám

• do paměti mu je uloží simulátor světa

O první možnosti jsme už mluvili, agent je vybaven speciálními instrukcemi, jejichž implementace zapisují specifická přijatá data do příslušných podpamětí. Když tedy agentovo FSM dá příkaz, data se zapíší.

Druhá možnost je automatická, agentem nijak nevynucovaná, má ji na starosti simulátor světa. Jedná se zejména o ukládání informací z pomyslného vizuálního percepčního centra agenta. Simulátor pomocí raycasting metody zjišťuje podle úhlu pohledu agenta a okolních překážek, na které objekty vidí a tyto mu jsou zapisovány do příslušných podpamětí v hlavní paměti. Důležité je zde uvést, že frekvence těchto zapisování je implicitně taková, že při každé aktualizaci AI smyčky se provede volání raycasting metody, a agentova “vizuální” pamět je aktualizována. Tato vlastnost simulátoru je ovšem nastavitelná, protože volání raycasting metody může být značně drahá operace (ve smyslu konzumace CPU času). Uvažme, že agent vysílá několik (desítek) paprsků ve směru, kterým se dívá, pro tyto paprsky jsou počítány kolize s objekty ve světě a podle nich je odvozen seznam viditelných objektů. S narůstajícím počtem agentů toto může být značná zátěž pro CPU.

K optimalizaci tohoto problému je možno přistoupit více způsoby. Za prvé, je důležité si uvědomit, že stav, kdy je každou aktualizaci agentovy AI smyčky naplněna znovu jeho vizuální podpaměť, je nadměrný luxus. Uvažme, že simulace běží v realném čase a nacházíme se v ideálním stavu, kdy aplikace běží na 60 AI aktualizací za sekundu. Tudíž i agentova vizuální paměť je obnovována každých 1000/60 ms. Pokud uvážíme, že agenti by měli v jistém smyslu napodobovat chování člověka, který má oční aparát s obnovovacím frekvencí 30 aktualizací za sekundu, můžeme se spokojit s tím, že agent si bude obnovovat vizuální pamět každý druhý snímek.

Další možností optimalizace je snížení počtu vysílaných paprsků do okolí. Tím pádem nebude nutné tolik výpočtů kolizí paprsků s geometrickými objekty ve scéně. Všechny tyto vlastnosti (a ještě velké množství dalších) jsou nastavitelné v konfiguraci simulace.

2.1.3 Virtuální svět a jeho model

Už bylo uvedeno, že, aby agent mohl ve virtuálním světě operovat, potřebuje si vytvořit svůj vlastní model tohoto světa. Ovšem i vlastní simulace potřebuje mít konzistentní model světa, který simuluje, jeho hierarchii, pozici a vlastnosti objektů apod.

Jednou z nejdůležitějších částí modelu světa (z hlediska AI) je navigační model pro agenty. Můžeme si jej představit jako grafovou strukturu, kde uzly jsou místa ve virtuálním světě a hrany jsou cesty mezi nimi. Díky tomuto grafu si mohou agenti ve světě hledat cesty a pohybovat se po něm. Obecně existuje mnoho technologií a paradigmat pro navigaci agentů ve virtuálním světě:

- systém waypointů
- navmesh
- gradient

Systém waypointů byl již stručně zmíněn. Jedná se o grafovou strukturu. Pohyb agenta spočívá ve vybírání správných uzlů tak, aby se po hranách mezi uzly mohl pohybovat ke svému cíli. Vzhledem k tomu, že systém waypointů je grafová struktura s pěvně danou matematickou strukturou, můžeme použít standartizované a prověřené techniky navigace agenta touto strukturou. Pro hledání cesty je v této práci využitý algoritmus A*.

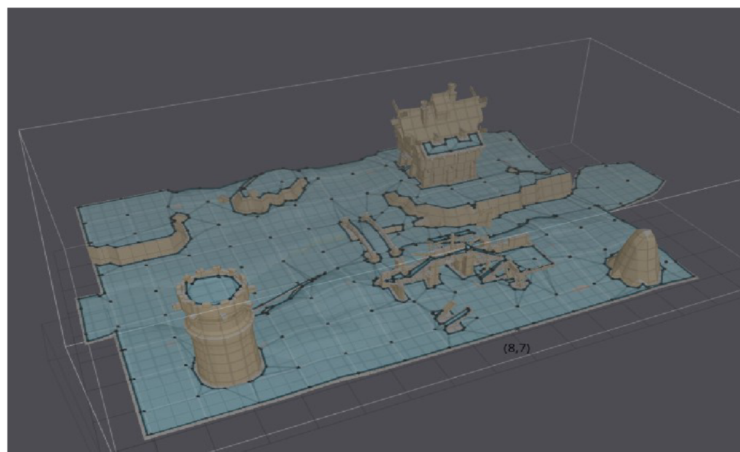
Důležitou věcí, která nebyla uvedena, je jak vlastně systém waypointů vzniká. Existují pouze dvě možnosti (případně jejich kombinace):

- dynamické generování z informací o světě
- pevně daná struktura navržená manuálně člověkem

Druhá zmíněná možnost má několik nevýhod. Pro rozsáhlé světy je ruční umisťování navigačních waypointů značně pracné. Je také vyžadován nějaký editor, specializovaná aplikace, která uživateli zobrazí strukturu světa a umožní mu do ní vkládat navigační informace. Další nevýhodou je, že při změně světa (např. přidání nové překážky) je nutné navigační informace přepracovat.

Prvně zmíněná možnost - dynamické generování - byla zvolena i v této práci. Implementační detaily budou zmíněny později, uveďme zde však jen to, že navigační data jsou počítána a generována podle reliéfu světa automaticky, tudíž není vyžadován návrhář. Nevýhodou tohoto přístupu mohou být chyby a nedostatky, kterých by se člověk při manuálním umisťování nedopustil.

Další možností navigace agentů je tzv. navmesh, který lze vidět na ilustraci 1. Jedná se o navigační strukturu složenou pomocí CG metod (constructive geometry), kdy volná prostranství, po kterých agenti mohou chodit, jsou reprezentována polygony. Tyto polygony jsou poté pomocí konstruktivní geometrie propojené do jedné velké struktury - navmeshe. V rámci navmeshe má agent pro svůj pohyb poměrně velkou volnost, z nejjednoduších metod pohybu zde můžeme uvést metodu zmenšování vzdálenosti od cílového bodu. Navmesh je opět možno vytvářet dynamicky i manuálně. V moderních počítačových hrách se prevážně používá tato metoda. Její nevýhodou však je, že algoritmus hledání cesty v navmeshi je značně složitější, než u systému waypointů - nepracuje se zde už s jednoduchou grafovou strukturou.



Ilustrace 1: Navmesh [4]

Poslední metoda navigace ve virtuálním světě je metoda gradientní. Agent se přesunuje pomocí metody postupného zmenšování vzdálenosti k cílovému bodu, přičemž detekuje kolize s neprůchodnými předměty a změnou směru se jim snaží vyhnout. Tato metoda se používala ve starších počítačových hrách, kde nebyl rozsáhlý a členitý virtuální svět. Dnes již tento přístup není příliš použitelný pro seriozní produkty.

Virtuální svět však není tvořen pouze navigační strukturou, ale musíme mít uložené i informace o poloze objektů v něm. Pro menší virtuální světy stačí mít tyto informace (pozice) uložené v lineárních datových strukturách, kde případné časté hledání nebude mít negativní dopady na výkon (v realtime aplikacích je nutno optimalizovat). Pro tyto účely většinou postačuje jednorozměrné nebo dvourozměrné pole, v případě dvourozměrného zde už hraje roli i spatiální informace - souřadnice ve 2D poli jsou v nějakém vztahu k objektům (k jejich pozici), které jsou v poli uložené.

Pokud bychom však chtěli pracovat s rozsáhlejšími světy, je nutné v rámci optimalizace zvolit nějakou sofistikovanější strukturu pro ukládání informací o objektech ve světě. Pokud je totiž agent v nějaké pozici v rozsáhlém světě a je nutné počítat raycasting metodou objekty, které vidí, v neoptimalizované verzi bychom museli spočítat kolizi s každým objektem světa. Pro rozsáhlé virtuální světy tato možnost není absolutně použitelná. Pokud však pracujeme s datovou strukturou, která respektuje spatiální rozložení objektů ve světě, můžeme počítat kolize pouze s objekty, které jsou danému agentovi blízko, čímž se zbavíme výpočtu kolizí všech ostatních objektů, které náš agent ani nemůže vidět.

Mezi tyto datové struktury, které reprezentují spatiální rozložení objektů ve světě, patří

- BSP stromy
- quadtree
- octree

2.1.4 Pathfinding

Aby mohli agenti ve virtuálním světě vykonávat činnosti, pohybovat se a interagovat s jinými agenty, je potřeba nějaká forma navigace. Jinými slovy, je třeba, aby agent byl schopen najít cestu ze svého aktuálního místa do místa cílového, kde mu bylo např. FSM naplánována nějaká činnost.

Na nalezenou cestu máme několik požadavků, z nichž nejdůležitější jsou:

• cesta by měla být co možná nejkratší

• cesta by neměla obsahovat cykly (vyplývá z prvního bodu)

• cesta musí respektovat reliéf povrchu a překážky na něm

Algoritmus hledání cesty přímo závisí na volbě datových struktur, které reprezentují virtuální svět a možné cesty v něm. V této práci byla k tomuto účelu zvolena struktura systému waypointů. Jedná se o grafovou strukturu, kde jednotlivé uzly grafu reprezentují místa ve virtuálním světě a hranami jsou reprezentovány cesty mezi nimi. Algoritmus pak prochází tuto grafovou strukturu, kdy začne na aktuálním uzlu, kde se nachází agent, postupně prochází všechny jeho okolní uzly, až dojde k cílovému uzlu. Při procházení uzlů si ukládá cestu, kterou zatím prošel, a při dosažení cílového uzlu si tuto cestu zpětně rekonstruuje.

V této práci byl pro vyhledávání cesty použit algoritmus A*, byl však ještě modifikován o hierarchický prvek tak, aby se samotným algoritmem A* nemusel procházet celý možný stavový prostor. Hierarchické použití A* je popsáno v praktické části textu, zde si definujeme samotný algoritmus A*.

Algoritmus A* je algoritmus pro procházení stavového prostoru. Stavový prostor je v tomto případě množina všech uzlů v grafové struktuře reprezentující možné cesty ve virtuálním světě. Samotný algoritmus A* patří mezi informované metody prohledávání. Pro tyto algoritmy platí, že podle charakteru úlohy je možné definovat hodnotící funkci f, která pro každý uzel stromu řešení určí jeho ohodnocení [2]. Hodnoty hodnotící funkce se pak používají k výběru uzlu pro expanzi. Odtud je intuitivně zřejmé, že pokud hodnotící funkce dobře postihuje vlastnosti a charakter úlohy, budou vždy expandovány "nejperspektivnější" uzly a zabrání se prohledávání cest, které nevedou k cíli (a které by byly zbytečné).

Algoritmus A* využívá k výpočtu funkce f navíc tzv. heuristiku, celkově tedy můžeme funkci f(i) v i-tém stavu jako součet ceny g(i) optimální (dosud nalezené) cesty z počátečního stavu do stavu i a ceny h(i), optimální cesty ze stavu i do některého z cílových stavů. [2]. Tedy (viz. vzorec 1):

$$f(i) = h(i) + g(i) \quad (1)$$

V každém kroku prohledávání bude tedy při aplikaci algoritmu A* vybrán k expanzi ten uzel, který má minimální hodnotu hodnotící funkce f(i). Ve většině případů a úloh však funkce g(i) a h(i) neznáme, a proto je nahrazujeme jejich odhady g*(i) a h*(i). [2]. Funkci h(i) potom nazýváme heuristickou funkcí. Zjednodušeně řečeno, hodnota heuristické funkce pro daný uzel nám udává, jak potenciálně výhodné je zvolit tento uzel.

2.2 Možnosti řízení inteligentních agentů

Aby se agent mohl ve virtuálním světě nějakým způsobem chovat, pohybovat se a interagovat, potřebuje nějaký mechanismus řízení. Už bylo uvedeno, že existují reaktivní agenti, jejichž reakce neuvažují využití paměti a reagují vždy jen na informace, které mají z aktuální události. My se zde však budeme zabývat agenty, kteří využívají pamět.

Ovládací aparát agenta je taková datová struktura nebo algoritmus, která zpracovává vstupní podněty z okolí agenta, dává je do kontextu s agentovou pamětí a na základě rozhodnutí vytváří agentovu reakci.

2.2.1 Statické řízení agenta

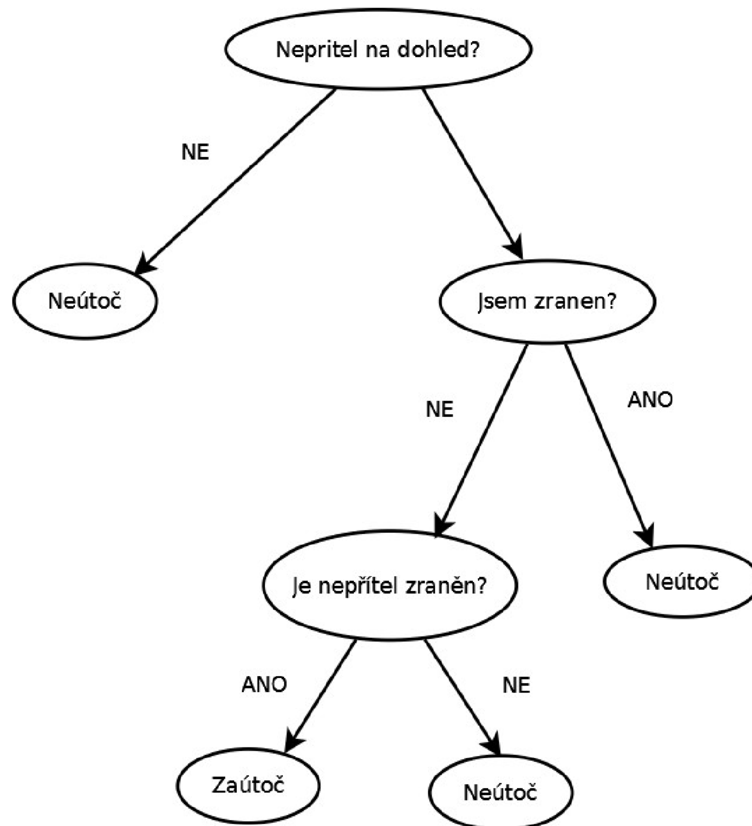
Jedná se o nejjednoduší možné řízení agenta, kdy ovládací aparát je přímo zakomponován do kódu aplikace. To znamená, že aktualizační funkce agenta zavolá přímo kód, který realizuje řídící aparát agenta. Tento kód by mohl např. pro agenta typu kořist (unikající před predátory, a vyhýbající se předchozím nebezpečným místům), vypadat ve zjednodušeném algoritmu tak, jako v příloze 6.

2.2.2 Rozhodovací strom

Obecně je cílem rozhodovacího stromu pomocí řetězce predikátů (realizovaného cestou od kořene k listu stromu) rozhodnout o příslušnosti posuzovaného objektu k nějaké třídě. Rozhodovací strom si tedy můžeme představit jako obecný strom, jehož uzly jsou nějaké dotazy na stav objektu nebo situaci ve stavovém prostoru a jednotlivé hrany vedoucí z uzlu reprezentují příslušné odpovědi na tyto dotazy. Pokud je odpověd kladná, pokračujeme uzlem do kterého vedla hrana příslušné otázky, až se takto dostaneme k listu stromu, který tvoří jedno z možných klasifikací/rozhodnutí o problému.

Pokud bychom měli rozhodovací stromy použít k ovládání agentů, bylo by vhodné použít větší množství rozhodovacích stromů, každé specializované na určitou část řízení agenta, např. rozhodovací strom pro pohyb agenta by měl v uzlech dotaz na vzájemnou vzdálenost s viditelnými objekty virtuálního světa a v listech by bylo rozhodnutí o tom, že agent se má otočit, případně přesunout nějakým směrem.

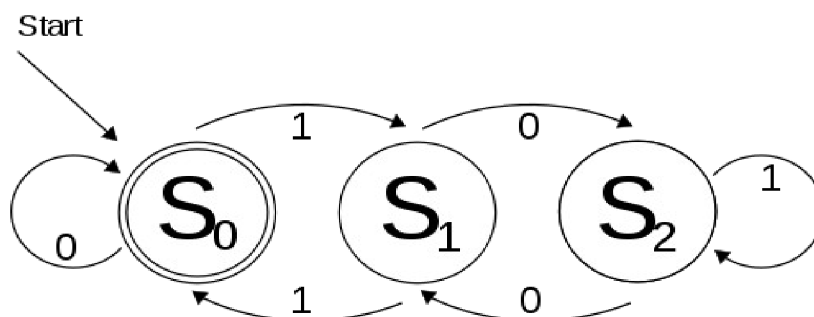
Jiným příkladem může být např. rozhodování agenta, zda má zaútočit na nepřátelského agenta, který se nachází na dohled. Popisovaný rozhodovací strom může vypadat např. jako na ilustraci 2:



Ilustrace 2: Příklad rozhodovacího stromu

2.2.3 Konečný automat

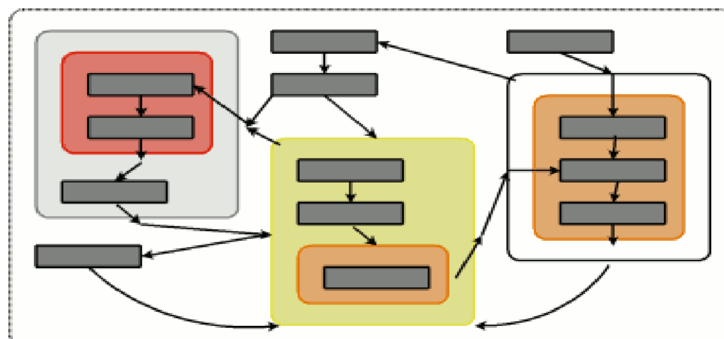
Konečné automaty (FSM - Finite State Machines) jsou velmi často používané aparáty pro ovládání agentů, např. v počítačových hrách. Konečný automat se dá popsat jako grafová struktura, kde uzly jsou stavy agenta (akce) a hrany jsou události (podmínky). Např. agent se může nacházet ve stavu "náhodný pohyb", tzn. je v uzlu reprezentující tento stav, dokud nenastane nějaká událost, např. "nepřítel spatřen". Pokud tato událost nastane, FSM změní aktuální stav pomocí přechodu po hraně "nepřítel spatřen" a přejde do nového stavu "stíhání nepřítele". Pokud se nachází v tomto stavu, je jeho řídící aparát na nižší úrovni nastaven tak, aby se pohyboval směrem k nepříteli. Tento způsob je použit v této práci. Příklad konečného automatu lze vidět na ilustraci 3.



Ilustrace 3: Příklad konečného automatu [5]

2.2.4 Hierarchický konečný automat

Jedná se o stromovou strukturu, kde každý uzel stromu je tvořen jednodušším nehierarchickým konečným automatem. Často je dostačující dvouúrovňová nebo tříúrovňová výška stromu. Opět můžeme použít příklad z minulé kapitoly, kdy na nejvyšší úrovni - v kořenu stromu je automat se dvěma uzly - "náhodný pohyb" a "stíhání nepřítele". FSM se tak dlouho nachází ve stavu náhodný pohyb než nedojde k události "spatřen nepřítel", kdy FSM přejde do stavu "stíhání nepřítele". Jakmile je v tomto stavu, popis a ovládání této činnosti převezme FSM o jednu úroveň níže, které má na starost ovládání při režimu stíhání nepřítele. Příklad hierarchického FSM je na ilustraci 4.



Ilustrace 4: Příklad schématu hierarchického FSM [6]

2.2.5 Boids - flocking

Toto ovládací paradigma se používá v jiných případech než ostatní zde zmíněné. Pomocí něho je relativně snadné simulovat pohyb velkých množství samostatných živočichů bez pevně daného vedení. Typickým příkladem může být hejno ryb v moři nebo hejno ptáků na obloze. Tato seskupení jsou zajímavá z toho hlediska, že hejna nejsou vedena žádným jedincem, a i kdyby byl nějaký vedoucí jedinec na špičče hejna (který jej může vizuálně vést) odstraněn, na funkčnost a pohyb hejna toto nebude mít žádný vliv. Autorem algoritmu zvaného flocking, který simuluje toto chování, je Craig Reynolds, výzkumník na poli Umělého

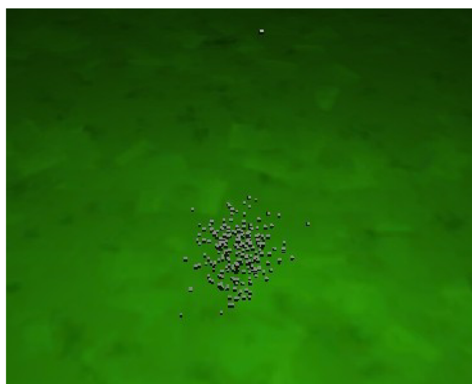
života [3]. Pro implementaci tohoto algoritmu je potřeba na každého jedince v hejnu uplatnit sadu jednoduchých pravidel (minimálně jsou potřeba tři, která budou zmíněna, je možno ještě další přidávat). Flocking je typickým případem jevu zvaného emergence, který lze definovat tak, že pokud se díváme na jednu úroveň chování nějakého subjektu a sledujeme jeho pravidla, tato pravidla se mohou na vyšší úrovni vnímání projevovat mnohem komplexnějším dojmem, než se jeví z pravidel na nižší úrovni. Totéž platí pro boidy, kteří implementují algoritmus flockingu: na jedné úrovni chování - na úrovni jedince - se aplikuje pár jednoduchých pravidel, když se však ale podíváme na boidy z jiné úrovně – úrovně hejna, toto hejno se chová značně komplexně.

Je nutné podotknout, že flocking nepatří mezi běžné aparáty pro ovládání agentů, někdy se ale používá v kombinaci s jinými aparáty, např. FSM. Ideální je tuto kombinaci rozložit do více úrovní, kdy např. v globálním měřítku se jedinec chová podle pravidel flockingu, ale svoje individuální činnosti řídí pomocí FSM.

Další možností kombinování je např. varianta, kdy se hejno po určitou dobu chová podle pravidel flockingu, a pak (např. po dosednutí hejna ptáků na zemi) se jednotliví členové hejna začnou řídit podle FSM.

Již jsme se zmiňovali, že pro simulaci flockingu stačí několik elementárních pravidel aplikovaných na každého jedince. V příloze [1] jsou tato pravidla v pseudokódu, patřičně okomentovaná [3].

Tato jednoduchá tři pravidla postačují k vytváření komplexního chování - simulace hejna živočichů. Samozřejmě, pokud bychom chtěli dosáhnout chování opravdu realistického, je třeba přidat ještě další modifikace - pravidla. Pokud bychom např. chtěli přidat pravidlo, aby boidové následovali nějakou předem danou cestu, stačí od vektoru pozice každého boida vždy odečíst rozdíl jeho pozice a pozice cílového bodu, a tento rozdíl vydělit nějakým koeficientem, udávajícím rychlost přibližování. Chování boidů lze vidět na ilustraci 5.



Ilustrace 5: Boidové v autorově simulaci

2.2.6 Dynamické fsm

Ve většině aplikací mají agenti svoje řídící FSM pevně dané, tzn. buď je natvrdo zapsáno v kódu ovládajícím agenta, nebo je načítáno jednou při startu simulace ze souboru. V této práci se ovšem pracovalo s dynamickými FSM, které podléhají modifikacím. Modifikace jsou takového druhu, že simulují účinky evoluce na daná FSM. Základní operace jsou tedy:

- mutace
- křížení

Podrobnosti budou uvedené v praktické části, zde však uvedeme jemný nástin problematiky. Největší problém při evoluci FSM je ten, že klasické FSM je grafová struktura, která se poněkud obtížně linearizuje - serializuje. Je to dáno tím, že v grafu jsou cykly, větvení, apod. Problémem je, že právě pro operace mutace a křížení je lineární reprezentace velmi výhodná. Autor této práce proto stál před problémem návrhu reprezentace FSM tak, aby bylo možné jej snadno křížit a mutovat. Podrobnosti k mutacím a křížení FSM budou uvedeny v praktické části, zde již jen stručně:

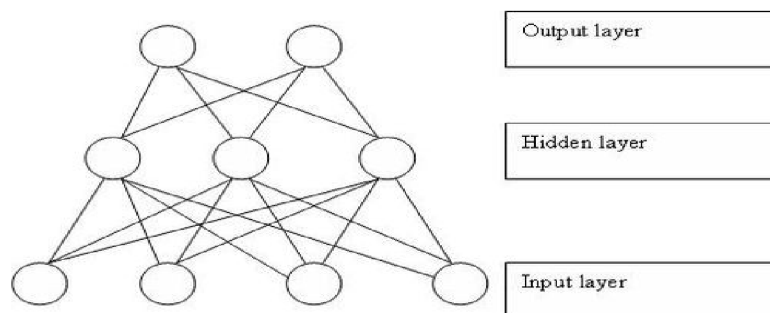
- mutace - záměna některých vlastností FSM - např. změna příkazu v uzlu FSM nebo změna jeho parametru, nebo změna události na hraně FSM.
- křížení - záměna celého podstromu - podgrafu FSM za jinou část (buď novou nebo od jiného agenta)

2.2.7 Fsm ovládané neuronovou sítí

Jednou z málo používaných metod, se kterou však ještě nebylo provedeno dostatečné množství experimentů, je kombinace FSM + neuronová síť. Neuronová síť je matematický (zjednodušený) model neuronových sítí v živých organismech. Existuje mnoho zapojení - topologií - neuronových sítí a také mnoho metod učení neuronových sítí. Neuronové sítě můžeme v zásadě dělit podle asistence při učení na:

- samoučící se neuronové sítě
- neuronové sítě učené s učitelem

Princip funkce neuronové sítě spočívá ve dvou módech činnosti. Mód učení probíhá tak, že na vstupní vrstvu neuronové sítě jsou předkládána data z učící množiny a po provedení výpočtu činnosti sítě se sledují data na výstupní vrstvě. Následně je spočítána chyba sítě - rozdíl mezi požadovaným výstupem a aktuálním výstupem sítě. Podle velikosti chyby jsou pak vlastnosti sítě zpětně upraveny - např. algoritmem back propagation. Schéma 3. vrstvé neuronové sítě lze vidět na ilustraci 6.



Ilustrace 6: Příklad 3 vrstvé neuronové sítě [7]

Mód činnosti neuronové sítě probíhá tak, že na vstupní vrstvu jsou přikládána vstupní data, proběhne výpočet činnosti sítě a následně jsou přečtena data na výstupní (nejvyší vrstvě) sítě.

Dosud jsme si nic neuvedli o základních stavebních prvcích neuronové sítě, nyní nastal čas to napravit. Neuronová síť je opět grafová struktura, skládající se z neuronů a propojení mezi nimi. Neuron je matematická aproximace biologického neuronu a jak už je obvyklé, existuje velké množství matematických aparátů, popisujících činnost neuronu. Nejjednoduší aproximace a první objevená se jmenuje perceptron. Je odvozena od činnosti neuronů zpracovávajích vjemy v nervové soustavě. Perceptron má několik vstupů (které jsou vedeny z výstupu jiných neuronů nižší vrstvy, nebo ze vstupů sítě) a jeden výstup. Tento výstup pak může být rozveden mezi větší množství dalších neuronů vyšší vrstvy. Výstup aktuálního neuronu spočívá tak, že neuron provede sumaci všech svých vstupů, její výsledek vloží na vstup transformační funkce a její výstup už je výstupem neuronové sítě.

Princip ovládání FSM neuronovou sítí pak spočívá v tom, že na vstup neuronové sítě jsou přivedeny vstupní hodnoty agenta - např. pozice předmětů ve virtuálním světě. Neuronová síť toto zpracuje a její výstup pak může indikovat povolení nebo zakázání některých hran v FSM.

2.2.8 Neuronová síť ovládaná Fsm

Jedná se o podobný princip jako u předešlého řešení. Rozdíl je v tom, že aparátem, který řídí agenta je zde neuronová síť - resp. její výstup.

Tuto implementaci si můžeme představit tak, že FSM sleduje vstupní hodnoty ve vstupní vrstvě neuronové sítě a pro určité hraniční hodnoty dochází v FSM k “události” - přesunu jednoho stavu do druhého. Přechod do jiného stavu může způsobit odpojení určitých propojení v neuronové síti, případně změnu hodnoty vah některých propojení. Co se týká samotných aktuátorů agenta - tedy částí, které ovládájí koncové akce, může být ovládací schéma následující:

Pokud může agent provést v daném čase N akcí, výstupní vrstva neuronové sítě má N výstupních neuronů - výstupů. Hodnota na každém z N výstupů potom udává, jak je vhodné použít akci přiřazenou k tomuto výstupu. Obvykle se bere největší nebo nejmenší hodnota. Příklad lze vidět na tabulce 1.

Tab. 1: možné akce a výstupy sítě

Akce	Otoč vlevo	Otoč vpravo	Jdi rovně	Stůj
Výstup sítě	0,8	0,7	0,95	0,35
Vybráno	Ne	Ne	**ANO**	Ne

2.2.9 GravFSM

Jedná se o autorem navržený aparát, který byl experimentálně zkoušen jen na omezených příkladech. Jednoduše si GravFSM můžeme představit jako uzly FSM a tzv. atraktory umístěné v N rozměrném prostoru. Klíčovou roli zde hraje gravitace. FSM začíná v předem daném stavu, kde je umístěn hmotný bod. Prostor, ve kterém je FSM implementováno, má tolik dimenzí, kolika hodnotami se dá popsat stav agenta ve virtuálním světě. Každý uzel FSM (akce) je umístěn v tomto prostoru. Kolem něj jsou v určité vzdálenosti atraktory, které mají za úkol "vystřelit" hmotný bod v aktuálním uzlu tím směrem, který odpovídá nejvíc danému atraktoru. Pro výpočet gravitace mezi atraktorem a hmotným bodem stačí spočítat euklidovskou vzdálenost mezi pozicí atraktoru a mezi hmotným bodem. Pokud se hodnoty budou sobě blížit, bude hmotný bod více přitahován k danému atraktoru, který ho vystřelí směrem k dalším uzlům.

Důvodem, proč autor této práce koncept GravFSM navrhl, byl tento. Gravitační FSM je snadné obohatit o koncept učení v kombinaci se simulovanou evolucí. Zároveň je snadné GravFSM evolvovat, tedy mutovat a křížit, protože veškeré tyto operace jen přesunují uzly a atraktory v prostoru. V neposlední řadě, je lehké GravFSM serializovat. Stačí si pro každý objekt uložit jeho pozici. Jako nevýhodu autor vidí to, že pokud se dá aktuální stav agenta popsat hodně proměnnými, roste počet dimenzí prostoru GravFSM a tím pádem se i zpomaluje výpočet gravitačních interakcí.

Evoluce může probíhat podle algoritmu uvedeného v příloze 7.

2.3 Genetické algoritmy a evoluční výpočetní techniky

V této práci je pro “zlepšování inteligence” jedinců - agentů - použito konceptu dynamického FSM v kombinaci s asociativní pamětí, přičemz FSM podléhá pravidlům Darwinovské evoluce.

Evoluce v této práci je konkrétně postavena na principech genetických algoritmů. Na začátek je vhodné si uvést několik pojmů. Jedinec je jednotka podléhající evoluci, jejíž vlastnosti se snažíme zlepšit. Populace je množina jedinců. Genotyp je datová struktura, která kodifikuje vlastnosti jedince, v případě přírody je to DNA, v případě genetického algoritmu (GA) je to datová struktura, ze které jsou odvozeny vlastnosti jedince, v případě genetického programování to např. může být nelineární seznam v jazyce LISP, apod. Fenotyp je jedinec, který vznikl kódováním z genotypu. Fitness function (hodnotící funkce, funkce vhodnosti) je funkce, která každému jedinci populace přiřazuje ohodnocení vzhledem k dané problémové doméně. Obvykle hledáme extrém této funkce. Genetický algoritmus (GA) je algoritmus, který využívá principů Darwinovské evoluce a přírodního výběru ke zlepšení populace jedinců.

Pseudokód Genetického algoritmu je v příloze 8.

3 Implementace

3.1 Reprezentace světa, implementace v simulátoru

Jak již bylo řečeno, pro chod simulace a testování agentů je nutné udržovat informace o virtuálním světě, tyto informace aktualizovat a poskytovat je agentům, navíc je nutné toto zajistit pokud možno v minimálním čase, aby agenti a jejich činnost mohla být posléze vykreslována a počítána v reálném čase. Nejdůležitější a základní datovou strukturou popisující svět je systém waypointů, podle kterého agenti řídí veškerou svoji navigaci.

3.1.1 Systém waypointů

Waypoint je základní prvek v této navigační struktuře. Pro každý waypoint je nutno ukládat a v paměti udržovat větší množství informací, zejména tyto:

- seznam jeho sousedů
- seznam vzdáleností k sousedům
- zda je waypoint povolen (viz. dále)
- segment, ve kterém se waypoint nachází
- pozice ve virtuálním světě
- jednoznačné identifikační číslo
- další informace

Seznam sousedů je lineární pole, které obsahuje ukazatele na sousední waypointy. Tím máme zajištěnou informaci o tom, kam může z daného waypointu agent jít. Pro to, aby mohla být použita heuristika při vyhledávání cesty jsou potřebné informace o vzdálenostech k sousedním waypointům. Toto zajišťuje druhý zmíněný seznam.

Waypoint může být dočasně zakázán. Pro uložení příznaku, zda je waypoint povolen, či zakázan, je třeba tento příznak pro každý waypoint uložit. Možnost zakázání waypointu je důležitá ze dvou důvodů:

- nízkoúrovňové hledání cesty - hledání cesty funguje na více úrovních a jakmile je nalezena vysokoúrovňová cesta (po segmentech), hledá se detailní cesta v síti waypointů. Už se však

neprohledává cesta ve všech waypointech, ale jen v těch, které patří segmentům v nalezené vysokoúrovňové cestě. Všechny ostatní waypointy jsou právě v této chvíli pomocí zmíněného příznaku zakázány

- místo je nedosažitelné - na místě waypointu může být v aktuální chvíli nějaký předmět, který brání průchodu - toto musí vzít algoritmus pro hledání cesty v úvahu a tuto překážku obejít

Ukázka zdrojového kódu popisujícího waypoint je v příloze [2]:

3.1.2 Generování systému waypointů

Systém waypointů musí být vygenerován tak, aby cesty v něm obsažené dodržovaly vlastnosti terénu ve virtuálním světě. To zejména znamená, že agent nemůže projít neprůchodnými předměty, musí je obejít, apod. To znamená, že cesta nemůže procházet místem, kde je neprůchodný předmět. K dosažení systému waypointů s takovými vlastnostmi byl v této práci použit algoritmus, který je uveden v příloze 9.

Zjednodušeně můžeme říci, že tento kód pro každou možnou cestu mezi aktuálním waypointem a jeho sousedy spočítá kolize s předměty v aktuálním segmentu (toto je jedna z výhod rozdělení systému waypointů na segmenty, nemusí se počítat kolize se všemy předměty, ale jen s těmi prostorově blízkými), a pokud ke kolizi nedojde, je tato cesta uložena.

Je nutno podotknout, že k tomuto generování dochází jen jednou při spuštění simulace. Existují ovšem i projekty, kde je nutné dělat tzv. realtime generování waypointů. Jedná se např. o počítačové hry se zničitelným terénem. Jako příklad můžeme uvést hru Battlafield: Bad Company 2, která disponuje plně zničitelným terénem. Zde vzniká problém, a to ten, že při např. zničení zdi ve virtuálním světě je vytvořena nová potenciální cesta pro hráče i pro počítačem řízené protivníky (NPC), a proto je nutné toto ošetřit. Přístupy k tomotu problému jsou v podstatě 3:

- předpočítané varianty systému waypointů pro různé zničené objekty - vzhledem k množství zničitelných objektů by bylo těchto variant obrovské množství, tudíž tato cesta je možná jen pro aplikace s ne absolutně zničitelným terénem
- přepočítat při každé destrukci celý systém waypointů - zde je problém, že jelikož hry zobrazují svůj průběh v reálném čase, bylo by nutné toto provést v řádu jednotek milisekund, což není při současném HW možné. Jinak by se totiž hra na okamžik zastavila a způsobilo by to pokles vykreslovaných FPS. Toto řešení není ani možné příliš paralelizovat, protože na výsledku algoritmu přímo závisí pohyb postav ve hře a není možné zastavit NPC při čekání na výpočet
- parciální přepočet ovlivněného místa - v současných aplikacích jediné možné řešení, v místě destrukce objektu (zdi, domu, apod.) se provede nový přepočet systému waypointů. Zde je však tento algoritmus náročný v tom, navíc okraje nového segmentu musí být znovu napojeny na existující okolní systém waypointů, což nemusí být jednoduché implementovat.

3.1.3 Segmenty waypointů

Aby bylo možné se systémem waypointů pracovat i na vyšší úrovni a mohl být použit princip hierarchického vyhledávání cesty, je systém waypointů rozdělen do segmentů. Platí, že jeden segment může obsahovat jeden, nebo více waypointů. Každý waypoint si udržuje ve své datové struktuře ukazatel na svůj rodičovský segment, a naopak, každý segment si udržuje seznam waypointů, které obsahuje.

Ukázka kódu segmentu waypointů je v příloze [3]:

Segment tedy obsahuje kromě informací o tom, jestli je povoleno na něho vejít (není tam překážka), také informace o tom, zda byl v rámci vyhledávání cesty navštíven, obsahuje pole s ukazateli na objekty a waypointy, které obsahuje, apod.

Na kódu segmentu lze vidět jednu často používanou šablonovou třídu standardní šablonové knihovny C++, a to třída set. Jedná se o množinu prvků, a protože umí rychle odpovědět (v konstantním čase), zda obsahuje nějakou hodnotu, zde se používá pro ukládání identifikátorů objektů a waypointů. Proto, když hledáme nějaký takový objekt, nejprve vytvoříme dotaz na tyto množiny, které nám rychle dají odpověď na to, zda objekt obsahují, a pokud ano, provedeme s ním operace. Pokud objekt obsažen není, můžeme položit dotaz sousedům tohoto segmentu.

Jak lze vidět v kódu, segment obsahuje také ukazatele na svoje sousedy. Toto je výhodné při hledání cesty, protože sousedé jsou uloženi podle směru, ve kterém se nachází (pomocí indexů typu LEFT, RIGHT, apod.), a pokud hledání cesty využívá informaci o směru, kde se nachází cíl cesty, může přímo využívat informace o sousedství segmentů k nalezení waypointů, které se např. nachází vlevo od daného waypointu (pokud i cíl se nachází směrem vlevo).

3.1.4 Generování světa

Virtuální svět v simulaci je částečně náhodně (procedurálně) generován a částečně načten ze souboru s popisem světa. Tato kombinace je výhodná z toho důvodu, že můžeme zajistit vždy alespoň částečně stejné podmínky pro všechny simulace evoluce, ale díky náhodnému generování zde můžeme vložit i prvek náhody.

Soubor s popisem světa je jednoduchý textový soubor, kde každý řádek reprezentuje jeden objekt, na každém řádku je uložena informace o typu objektu a sloupec a řádek, kde se nachází v poli segmentů.

Algoritmus procedurálního generování světa pak musí brát v úvahu už umístěné objekty a rozměry světa k tomu, aby mohl umístit další objekty. Rovněž by měl rovnoměrně distribuovat objekty různých typů - pokud to není pro účely simulace vyžadováno jinak. Pseudokód popisující generování světa je v příloze 10.

3.1.5 Hledání cesty

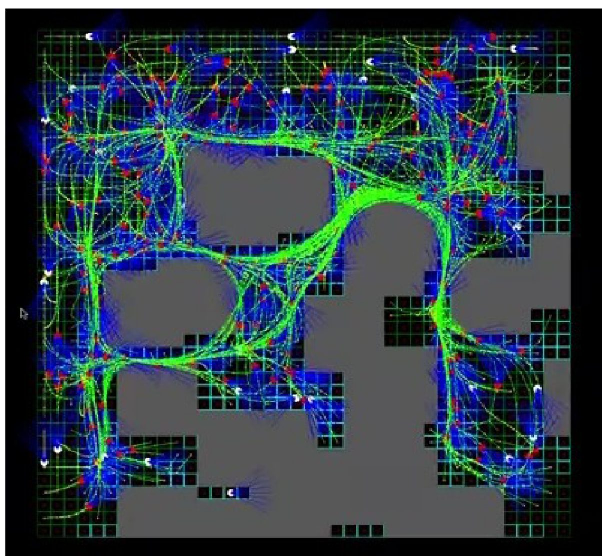
Hledání cesty probíhá v této práci hierarchicky, to znamená, že cesta je prvně vyhledána ve vyšších úrovních reprezentace světa a detailní cesta je vyhledávána pak už jen v těchto částech. Tento přístup značně urychlí prohledávání cesty pro rozsáhle světy.

Pseudokód vysokoúrovňového hledání cesty je v příloze 11.

Je zde využívána ta vlastnost, že prvně jsou prohledáváni ti sousedé aktuálního segmentu, kteří jsou ve směru k cílovému segmentu. Toto může značně zrychlit hledání cesty. Na začátku jsou všechny segmenty zakázány pro procházení při nízkoúrovňovém hledání, a po nalezení cesty, jsou povoleny jen ty, které jsou umístěny jen na této cestě. Tudíž nízkoúrovňové hledání bude procházet jen tyto segmenty.

Pseudokód nízkoúrovňového hledání cesty je v příloze 12.

Na pseudokódu lze vidět, že nízkoúrovňové hledání využívá výsledky hledání na vyšší úrovni a zakázané sousedy neprochází. Vyhledané cesty lze vidět na ilustraci 7.



Ilustrace 7: Zobrazení hledání cesty v ranné verzi této práce

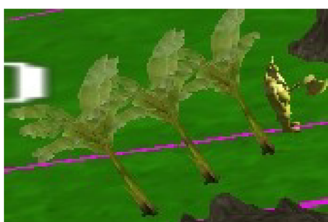
3.1.6 Objekty ve světě

V rámci zjednodušení obsahuje virtuální svět v této práci jen 3 typy objektů (pokud nepočítáme pohybující se agenty). Všechny tyto objekty jsou statické. Kromě nich můžeme uvažovat ještě čtvrtý typ objektu -

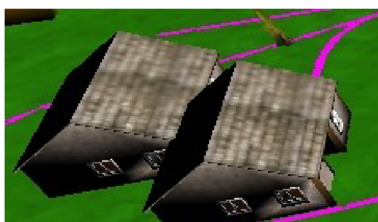
přenášené zdroje, které však nejsou reprezentovány graficky, ale jsou uloženy jen v kódu jako číslo ve vlastnostech agenta. Zbylé statické objekty jsou tedy tyto:

- strom
- budova
- skála

Strom (ilustrace 8) je využíván k těžení zdrojů, implicitně se jedná o nevyčerpatelný zásobník zdrojů, které agenti těží a mají za úkol nosit do základny. Stromů by němělo být v úrovních příliš mnoho, aby bylo zabráněno tomu, že agenti se k nim budou díky hustému zalesnění dostávat náhodně a nesmí jich být ani příliš málo, aby agenti měli vůbec možnost nějaké zdroje těžit.



Ilustrace 8: Strom v režimu 3D zobrazení



Ilustrace 9: Budova v režimu 3D zobrazení

Ilustrace 10: Skála v režimu 3D zobrazení

Budova (ilustrace 9) slouží jako základna agentů stejného kmene. Ti zde mají za úkol nosit zdroje, které natěží a díky tomu, že je donesou na základnu, zvyšuje se jejich individuální hodnocení (později využito při evoluci). Každý agent má přidělenou svoji základnu, která náleží jeho kmeni. V gui jsou agenti a kmeny odlišeny barvou, stejnou barvu má i jejich základna.

Skála (ilustrace 10) tvoří neprůchodný objekt, který se nachází ve světě, aby znesnadnil jeho procházení agenty, případně oddělil území jednotlivých kmenů. V implicitně nadefinovaném světě má každý kmen svoje oddělené uzemí, ke kterému je ve skalách jen několik průchodů.

3.1.7 Kontextové informace o světě

Aby agenti byli schopní zacházet a interagovat s objekty ve virtuálním světě, potřebují informace o těchto objektech. Tyto vlastnosti mají statickou formu, s časem se nemění a mají kontextový význam. Příklad: aby agent mohl vyzvednout ze stromu zdroje a přinést je na základnu, musí vědět, že ze stromu lze zdroje vyzvedávat a na základnu je lze nosit. Těmto informacím se říká kontextové, protože mají význam jen v kontextu aktuálního objektu. K uložení těchto informací máme víceméně 3 způsoby:

- každý objekt si nese kontextové informace s sebou
- každý agent si nese kontextové informace s sebou
- tyto informace jsou implicitně zapsány v programovém kódu

První dvě možnosti jsou vhodné pro rozsáhlé projekty, kde je potřeba často tyto kontextové měnit. Jako nejvhodnější způsob uložení těchto informací se jeví indexovaný seznam ukazatelů na funkce, které implementují danou činnost. K doplnění informací o činnosti je vhodný druhý seznam, který nese informace a povolené rozsahy pro příslušnou funkci, na kterou je v druhém poli uložen ukazatel.

3.2 Reprezentace znalostí agentů

Již bylo zmíněno, že, aby agent mohl operovat ve virtuálním světě a neomezoval se pouze na reaktivní chování, potřebuje nějakou formu paměti, ve které si může uchovávat krátkodobé i dlouhodobé informace. Informace uchovávané agenty v této práci lze dělit následovně:

- identifikátory
- pozice
- celá čísla

Identifikátory slouží k uložení informací např. jaké ostatní agenty právě daný agent vidí, v tomto případě jsou v příslušném podpoli uloženy identifikační čísla těchto agentů. Nízkoúrovňová implementace akcí agenta si pak může podle identifikačního čísla jiného agenta o tomto nalézt informace potřebné např. pro útok.

Jaká pole má agent k dispozici:

- pozice:
  - nebezpečná místa
  - místa se zdroji
  - místa, kde se pohybují nepřátelé
  - pole pro volné použití
  - druhé pole pro volné použití
- identifikátory
  - identifikátory přátelských agentů na dohled
  - identifikátory nepřátelských agentů na dohled
  - identifikátory objektů na dohled
  - těžitelné zdroje na dohled
  - pole pro volné použití 1
  - pole pro volné použití 2
- stavové informace agenta
  - aktuální zdraví
  - počet natěžených zdrojů (aktuálně)

   - počet natěžených zdrojů (globálně)
   - počet zranění způsobených jiným agentům

Tyto pole je možno dělit na dvě podskupiny, jedná se o pole:

- permanentní - jakmile do nich agent informaci zapíše, uchovává se zde, dokud není nutnost ji přepsat z důvodu nedostatku místa v agentově paměti
- aktualizované a přepisované simulátorem - toto platí pro většinu identifikátorových polí, konkrétně pro ty, které uchovávají informace o viditelných entitách. Tyto informace si nezapisuje agent sám, ale jsou mu poskytovány každý AI update simulátorem.

Ukázka kódu agentovy databáze znalostí:

```
class CDBDatabase {
private:
  CDBState                        m_agentState;
  // pole s pozicemi, klici a ciselnymi informacemi
  vector<CDBArray<SPosition2D> >          m_positionArrays;
  vector<CDBArray<DBKey_t> >    m_keyArrays;
  vector<CDBArray<int> >                m_numberArrays;
```

3.2.1 Přenášení znalostí mezi agenty

Aby se agenti mohli kolektivně chovat a komunikovat mezi sebou, musi existovat nějaký mechanismus sdílení informací. Jak jsme si již řekli, existují mechanismy sdílené paměti, sdílená tabule, kde agenti zapisují a čtou informace zapsané jinými agenty apod. V této práci byl zvolen jiný přístup. Jakmile agenti ze společného kmene si jsou dostatečně blízko, a pokud je k tomu jejich dynamické FSM naprogramované, mohou si přenést náhodně zvolenou informaci určitého typu, z vybraného podpole, mezi sebou. Např. FSM jednoho agenta může být naprogramované k tomu, aby při potkání agenta svého kmene k němu přenesl informaci o blízkých zdrojích.

Tuto činnost a koncept komunikace lze i podporovat ve fázi evoluce. Způsob, jak tohoto dosáhnout, je poměrně jednoduchý. Stačí přidat do agentových stavových informací čítač přenesených informací mezi agenty, případně ještě další čítač (v asociativním poli - mapě), jak který cílový agent této přenesené informace využil. Poté lze agenty s vysokou hodnotou tohoto čítače zvýhodňovat v procesu evoluce tak, že čítač je přetransformován do vzorce pro výpočet fitness funkce.

Jak již bylo zmíněno, je důležité, aby přenesené informace, to znamená informace, které jeden agent předá druhému, který nemusí mít ostatní informace stejné, měly správný význam i v tomto cílovém agentu. Proto musíme zajistit, aby:

- informace nebyly zatížené kontextem aktuálního agenta
- přenášené informace o pozicích měly absolutní, nikoliv relativní rozměr

Přenášení informací mezi agenty je realizováno ve FSM pomocí příkazu TalkTo, který má jeden parametr, specifikující který druh informací bude přenášen. Je také možné, že do cílového agenta

se přenese i informace, kterou už má, v tom případě ale není jeho funkčnost a informační integrita nijak zaručena. Důvod, proč je specifická informace ze sub pole agenta k přenesení vybrána náhodně je ten, že pokud by se při každém přenesení informace mezi dvěma agenty měly porovnávat obě jejich báze znalostí, mohlo by dojít ke zpomalení běhu simulace (protože k přenášení informací může docházet při větším množství agentů velmi často).

3.3 Návrh struktury FSM vhodné k evoluci

Protože FSM v této práci podléhá evoluci - a toto je jediný prostředek, jak jsou agenti schopní se přizpůsobovat okolí (kromě komunikace), je nutno mít samotné FSM uložené, nebo převoditelné do takové podoby, aby s ním bylo možno provést dvě základní operace, používané v genetických algoritmech:

- mutace
- křížení

I když tyto operace můžeme provádět i na nelinearizované struktuře, algoritmus pro ně je značně složitější, a i samotný fakt uchovávání FSM v nelinearizované (grafové) struktuře nám nepřináší žádné benefity, krom toho, že takové uložení je přirozenější. Na strukturu uložení FSM tedy máme následující požadavky:

1. možnost snadného křížení
2. možnost snadné mutace
3. práce s celou strukturou i s jejími částmi tak, aby nebylo poznat, že není uložena jako grafová struktura.

Způsob řešení tohoto problému bude uveden v následujících kapitolách, nyní si uvedeme, z čeho se FSM v této práci skládá.

- uzel
- množina predikátů
- jejich parametry
- množina příkazů
- jejich parametry
- relativní skoky do jiných uzlů

3.3.1 Predikát

Predikát si můžeme představit jako tvrzení o situaci ve virtuálním světě, které může být buď pravdivé, nebo nepravdivé. V případě, že interpret FSM provádí exekuci tohoto FSM a přečte tento predikát, a tento predikát má pravdivostní hodnotu true, provede interpret FSM příkaz, který je

asociovaný s tímto predikátem a následně skočí na uzel, jehož relativní adresa (vůči aktuálnímu uzlu) je udána skokem asociovaným s příkazem. Jako parametry predikátu mohou být pouze celá čísla (většinou identifikátory) nebo vnořené predikáty (je podporována jen jedna úroveň zanoření).

Nejdůležitějším prvek v kódu popisu predikátu je index m_idx, který je indexem do pole ukazatelů na funkce. Každá funkce, na kterou jsou zde ukazatele, implementuje jeden konkrétní predikát. Můžeme si zde uvést, jaké nejdůležitější predikáty může každý agent používat.

IsHealthFull - má agent plnou hodnotu života? Tzn. nebyl žádným jiným agentem zraněn. Plná hodnota života agenta je hodnota 100.

IsHealthLowerThan - má jeden číselný parametr. Predikát má hodnotu true, pokud je agentova hodnota života nižší parametr.

IsEnemyInSight - je na dohled nepřítel?

IsInRunAwayMode - je agent právě v módu útěku? Tento predikát je použit pro zjemnění rozhodování v FSM, kdy je např. nesmyslné útočit na nějakého cizího agenta, pokud je náš agent zraněn a uniká.

IsWorldObjectInSight - je na dohled nějaký objekt virtuálního světa? Počítají se zde veškeré objekty, kromě agentů samotných, agent však nemá možnost rozlišit druh objektu, k tomu slouží specializovanější predikáty.

IsInGotoResourceMode - je agent momentálně v módu přesunu ke zdroji? Opět tento predikát slouží ke zjemnění rozhodování, aby agent, který už se rozhodl pro přesun ke zdroji nebyl “vyrušen” jiným rozhodnutím.

NoEnemyInSight - opak predikátu IsEnemyInSight, v tomto případě má predikát pravdivostní hodnotu true, pokud v dohledu není žádný agent.

IsFriendInSight - predikát má pravdivostní hodnotu true, pokud na dohled je nějaký agent z jeho kmene.

IsTargetReached - tento predikát agent používá při plánování cesty. Nabývá pravdivostní hodnoty true, pokud agent stojí na místě - cíli své naposledy naplánované cesty. V tomto případě je většinou tento predikát napojen na příkaz plánující novou cestu.

IsResourceInSight - nabývá pravdivostní hodnoty true, pokud je na dohled nějaký zdroj. V tomto případě se většinou agent přepne do módu cesty ke zdroji, pokud není zrovna zraněný, a nebo neuniká před jiným agentem.

And - mechanismus vnořených predikátů, nabývá hodnoty true, pokud oba vnořené predikáty mají hodnotu true.

Or - mechanismus vnořených predikátů, nabývá hodnoty true, pokud alespoň jeden vnořený predikát má hodnotu true

3.3.2 Módy řízení agenta

Na tomto místě je vhodné uvést, že agent se řídí v podstatě dvěma módy řízení. Jeden je vysokoúrovňový a plánuje mu dlouhodobé cíle, a jeden je nízkoúrovňový, který detekuje nové situace a mění vysoceúrovňové plány. Nejlepší bude si toto předvést na příkladu. Např. agent dojde na cílové místo svojí naplánované cesty a po té si naplánuje vysokoúrovňový mód náhodné procházky. Nízkoúrovňové řízení se tedy začne řídit tímto příkazem a přesunuje agenta náhodně vybranou cestou. Přitom agent vyčkává nových podnětů od nízkoúrovňových "čidel". Jakmile uvidí zdroje, které je možné nosit na základnu, rozhodne se změnit svůj vysoceúrovňový plán na plán přesunu ke zdrojům.

3.3.3 Příkazy

Každý predikát, pokud je splněn, vyvolá svůj asociovaný příkaz. Příkaz je implementován jako funkce, která operuje nad daným agentem a mění jeho parametry a plány.

Nejdůležitější roli zde opět hraje členská proměnná m_idx, která obsahuje index do pole ukazatelů na funkce, které implementují příkazy. My si zde ty nejdůležitější příkazy uvedeme:

Chase - pronásleduj agenta, jehož identifikátor máš jako parametr

RandomWalk - náhodná procházka - jsou vybrány dvě místa ve virtuálním světě, které mezi sebou mají danou jistou minimální vzdálenost a mezi nimi je naplánovaná cesta

Runaway - utíkej před agentem, jehož identifikátor máš jako parametr. Útěk je naplánován tak, že jsou uváženy všechny nejbližší waypointy kolem aktuálního agenta, a k útěku je vybrán ten, který je nejvzdálenější od agenta, před kterým unikáme. Na tento waypoint je po té naplánována cesta.

SetLastSeenEnemy - do paměti - zásobníku je uložen ten nepřítel, jehož ID je uvedeno jako parametr tohoto příkazu.

AttackLastSeenEnemy - je naplánován útok na agenta, který byl předtím uložen pomocí příkazu SetLastSeenEnemy.

GotoResource - agent se přesune ke zdroji, jehož identifikátor je uložen v parametru tohoto příkazu.

AddLastSeenEnemyPositionToDangerous - uloží pozici, kde agent viděl naposledy nepřítele, do seznamu nebezpečných pozic ve virtuálním světě

AddLastSeenEnemyPositionToCustom - uloží pozici, kde agent viděl naposledy nepřítele, do seznamu pozic ve virtuálním světě

AddLastSeenResourcePositionToCustom - uloží pozici, kde agent viděl naposledy zdroje, do seznamu pozic ve virtuálním světě

AddLastSeenResourcePositionToResources - uloží pozici, kde agent viděl naposledy zdroje, do seznamu pozic zdrojů ve virtuálním světě

GotoCustomPosition - naplánuje cestu na pozici, jejíž index je v parametru tohoto příkazu a jež je uložená v poli pozic custom

SetLastSeenFriend - do zásobníku uloží naposledy spatřeného přítele - příslušníka stejného kmene

TellPosition - naposledy spatřenému příteli (nastavenému pomocí SetLastSeenFriend) přenese náhodně vybranou informaci o pozici ze svojí paměti do jeho - tímto je realizována komunikace

GotoBase - agentovi je naplánována cesta na základnu. Každý agent má přidělenou právě jednu základnu, která náleží jeho kmeni, a kam má za úkol nosit zdroje.

Příkazy mohou mít opět pouze celočíselné parametry, které mají buď význam jako celé číslo, udávající nějakou kvantitu, nebo identifikátor agenta nebo objektu virtuálního světa.

3.3.4 Generování uzlů predikátu a příkazů

Pokud chceme použít ke zlepšení vlastností agentů a potažmo FSM evoluční přístup, musíme zajistit možnost strojového generování celých FSM a jejich uzlů. Toto je potřeba z toho důvodu, že na počátku a v průběhu evoluce je potřeba vytvářet nové jedince. Sice by jsme mohli použít pro nového jedince napevno nastavené FSM, ale možnost náhodně generovaného jedince (jeho FSM) nám dává mnohem větší diverzitu v populaci, což je při evolučním přístupu vždy vítaný stav.

Samotný algoritmus generování uzlu je v příloze 12.

Ke generování parametrů predikátů a příkazů se používají struktury SCommandInfo a SPredicateInfo, které jsou uloženy v poli, přičemž jejich indexy odpovídají příslušným indexům dané implementace příkazu nebo predikátu. Obě struktury S*Info obsahují velikosti minimálních a maximálních přípustných hodnot pro predikát či příkaz.

Zdrojový kód struktur SPredicateInfo a SCommandInfo je v příloze [4].

Před spuštěnim simulace je nutné tyto mezní hodnoty nastavit pro všechny predikáty a příkazy. Je možno toto uložit a načíst ze souboru, nebo hodnoty inicializovat přímo v kódu. Ukázka inicializace v kódu (nastavení pro OR predikát) je v příloze [5].

V tomto případě inicializační kód signalizuje, že OR predikát musí mít dva parametry, což jsou hodnoty v rozsahu povolených indexů predikátů (pro uložení indexu na zanořený predikát se používají parametry nadřazeného predikátu). Poslední maximální povolená hodnota je IsResourceInSight, protože se zároveň jedná o poslední predikát. Způsob inicializace příkazů je podobný a nebudu ho zde uvádět.

Na tomto místě by bylo vhodné uvést ukázku kódu predikátu a příkazu, resp. některých jejich implementačních funkcí:

Predikát IsEnemyInSight:

```
bool IsEnemyInSightPredicate(CBaseAgent *_agent, CDBDatabase *_db, int _param1, int _param2, int _param11, int _param12, int _param21, int _param22)
{
 return _db->GetKeyArray(AIK_ENEMIES_IN_SIGHT).IsEmpty() == false;
}
```

Zde lze vidět, že predikát musí dostat ukazatel na agenta, se kterým má pracovata a rovněž na jeho databázi znalostí, případně je doplněn hodnotami parametrů. Tento konkrétní predikát zkontroluje na neprázdnost pole AIK_ENEMIES_IN_SIGTH, které obsahuje seznam identifikátorů nepřátelských agentů, kteří jsou na dohled.

Příkaz RandomWalk:

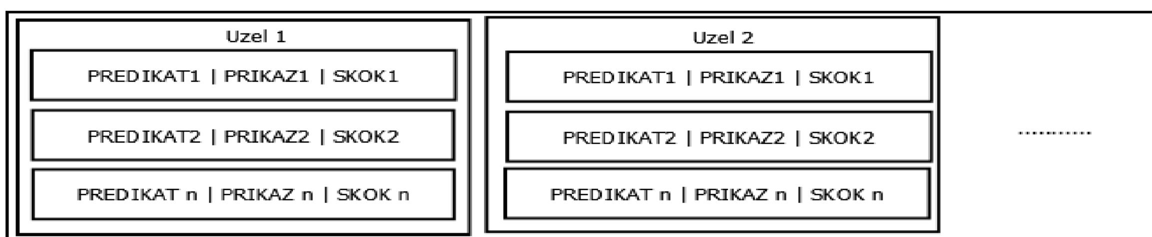
```
bool RandomWalkCommand(CBaseAgent *_agent, CDBDatabase *_db, int _param1, int _param2)
{
 #ifdef TEST
      stringstream ss;
      // vypsani testovaciho textu do logu
      ss << "agent " << _agent->GetAgentId() << " random walk";
      GetLog()->Log(ss.str(), LOGLEVEL_LOW);
#endif     // pokud je validni ukazatel na agenta
if (_agent)    {
// nastavime vysokourovnovy mod na randomWalk
      _agent->GetDatabase().GetState().SetState(AS_RANDOM_WALK);
      _agent->SetChasing(false);  //  zrusime pripadny mod stihani
      _agent->SetChased(false); // zrusime pripadny priznak stihaneho
      return true;
}
else
 return false;
}
```

3.3.5 Modifikace uzlů predikátu a příkazů

K modifikaci uzlů predikátů a příkazů dochází při mutaci uzlu. Tato činnost je popsána pseudokódem v příloze 13.

3.3.6 Uložení uzlů FSM

Uzly FSM jsou uloženy v poli za sebou, jak je vidět na ilustraci 11. Každý uzel v poli obsahuje množinu poduzlů, což jsou trojice (predikát, příkaz, skok), které mají takový význam, že interpret FSM prochází pole trojic od nejnižších indexů a zkouší pravdivost predikátů (tudíž na pořadí uložení trojic v poli záleží), pokud nějaký predikát dojde k pravdivostní hodnotě true, procházení se ukončí, provede se příkaz z trojice a skočí se na další uzel ve FSM daný hodnotou relativního skoku oproti aktuálnímu indexu uzlu ve FSM.



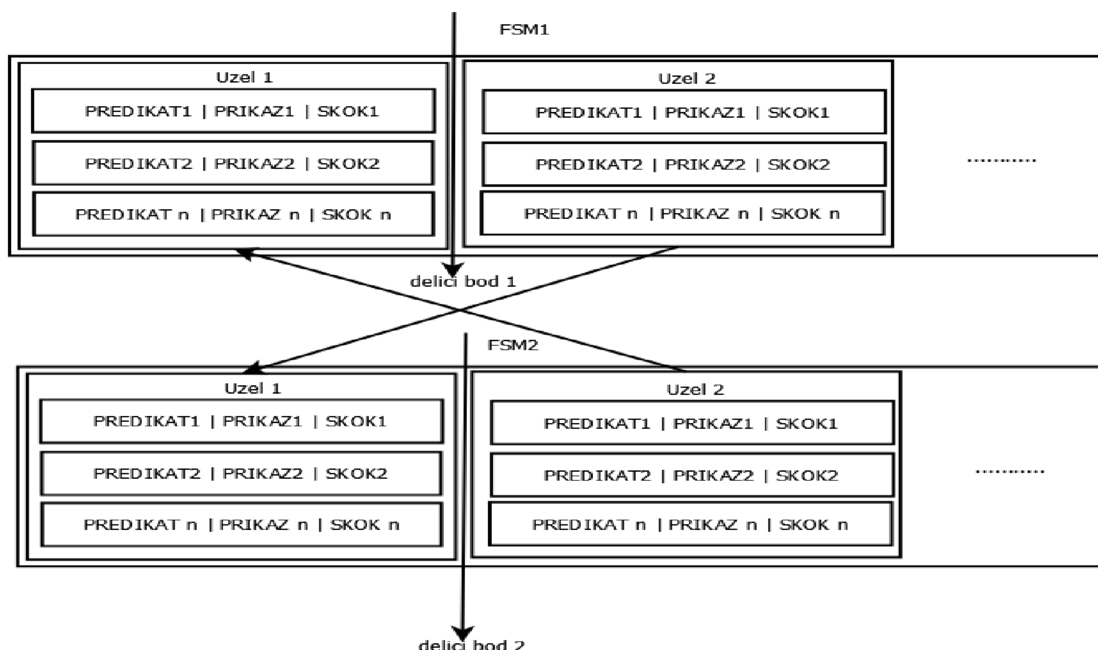
Ilustrace 11: Ukázka uložení FSM

3.3.7 Křížení FSM

Křížení je jednou z elementárních operací genetických algoritmů. Díky tomu, jak byla v této prácí navržena struktura FSM, křížení se snadno zjednodušší. Stručně řečeno, vše co je potřeba ke kříženi, stačí si zvolit indexy, kde rozdělíme první a druhé FSM, získáme "pod-FSM" z těchto rozdělení a ty zase následně konkatenujeme. Křížení je naznačeno na ilustraci 12.

Postup:

1. zvol náhodný index rozdělení prvního FSM v rozsahu 1 až délka prvního FSM - 1 (vynechání krajních bodů je z důvodu, že tímto dělením by nám vznikly opět původní FSM)
2. zvol náhodný index rozdělení druhého FSM v rozsahu 1 až délka tohoto FSM - 1
3. rozděl tyto FSM podle indexů vygenerovaných v předchozích krocích
4. vezmi první polovinu prvního FSM a první polovinu druhého FSM, spoj je (konkatenuj) a prohlaš je za prvního potomka
5. vezmi druhou polovinu prvního FSM a první polovinu druhého FSM, konkatenuj je a prohlaš za druhého potomka
6. totéž proveď analogicky pro první FSM a druhou polovinu druhého FSM
7. získáváme 4 potomky, které můžeme vložit do následující generace



Ilustrace 12: Příklad křížení FSM

3.4 Evoluce

Globálním cílem použití evoluce v této práci je dosažení lepší inteligence a chování agentů. Díky nastavení fitness funkce můžeme chování agentů optimalizovat na různá požadovaná schémata. Pro dosažení tohoto cíle stačí ve výpočtu fitness funkce zvýhodnit ta řešení, která mají velkou hodnotu příslušné statistické položky informující o úspěších agenta. Např.:

- množství natěžených zdrojů (položka GlobalResources ve statistikách agenta)
- množství zraněných nepřátelských agentů (položka GlobalHarm ve statistikách)
- velikost projíté cesty ve virtuálním světě (položka GlobalDistance)
- schopnost komunikace a její aktivní využívání (hodnocení neimplementováno)

Jako ideální se ukázalo ve fitness funkčni zohlednit více cílových jevů, ovšem každému přiřadit hodnotící koeficient, který mu přikládá určitou váhu. Např. by jsme měli zvýhodňovat agenty, kteří ušli velkou vzdálenost ve virtuálním světě, abychom předešli evolučnímu vznikání takových FSM, kde se agenti zacyklí na krátké trase, nebo se nepohybují vůbec. Přílišné lpění na této hodnotě a její nemístné oceňování však může mít i nežádoucí a často zajímavé účinky. Během ladění evolučního mechanismu totiž začali vznikat agenti, kteří se “naučili” využívat chyby v simulátoru k tomu, aby se “teleportovali” po virtuálním světě. Tito agenti totiž z pohledu hodnotící funkce ušli největší vzdálenost a natěžili nejvíce zdrojů (dovedli se mezi nimi rychle přesunovat), což tuto náhodně získanou schopnost posléze v běhu evoluce prosadilo do dalších generací. Pokud by mělo dojít k ručnímu programování FSM agentů, na tuto

chybu by asi nikdy autor nenarazil, protože je k jejímu vyvolání a destabilizaci simulátoru nutné vyvolat specifický řetězec příkazů v agentově FSM.

Jakmile autor narazil na toto chování, stál před úkolem, jak se tohoto nežádoucího jevu zbavit, protože z pohledu evoluce se jednalo o korektní chování. Nakonec se jako ideální měřítko indikující výskyt teleportačního jevu osvědčil počet požadavků na hledání cesty ve virtuálním světě. Jelikož k teleportaci docházelo mezi naplánovaným startem a cílem agentovy cesty, musel si agent vždy po teleportaci (která byla velmi rychlá a častá) zažádat o vyhledání nové cesty. Tak lze snadno rozeznat, že agent který má ve statistikách velké množství požadavků na pathfinding v krátkém časovém období, pravděpodobně používá teleportační chyby. Proto byl počet požadavků na hledání cesty zohledněn ve fitness funkci, a to tak, že jedince s vysokou hodnotou této veličiny evoluce znevýhodňuje.

3.4.1 Modul pro běh evoluce

Jako ideální postup se autorovi této práce jevilo vyvinout modul pro běh evoluce jako separátní spustitelný soubor, který nebude mít žádné uživatelské rozhraní, které by jen zpomalovalo běh evoluce. Modul evoluce je proto aplikace pro příkazovou řádku, která nevyužívá žádné grafické knihovny a pouze jedenkrát za sekundu vypíše krátké kontrolní informace, aby měl uživatel přehled o tom, zda a jak rychle evoluce běží.

Běh evoluce probíhá v krocích, které jsou popsány v příloze 14.

3.4.2 Optimalizace evoluce

Již byla zmíněna optimalizace evoluce z pohledu dosažení lepších výsledků, nyní ještě je nutno zmínit, jak evoluci optimalizovat z pohledu rychlosti běhu evolučního modulu. Jak vyplývá z postupu, kterým se evoluce ubírá a co se vše musí vykonat při jednom evolučním kroku, je zřejmé, že největší potřebu optimalizace má prvek simulující inteligenci agentů. Největšími zpomalujícími elementy zde jsou:

• raycasting - tedy výpočet viditelnosti agentů, počítá kolize mezi paprsky vrhanými agentem ve směru jeho pohledu a mezi okolními objekty. Náročnost tohoto algoritmu značně roste s počtem paprsků, které agent vrhá, přičemž kvalita chování se při velkém růstu počtu prvků nijak nezlepšuje, proto můžeme pro potřeby počet vrhaných paprsků značně omezit, a tuto hodnotu zvýšit až pro běh v režimu 3d zobrazení, kde je větší požadavek na reálné chování agentů

• hledání cesty - tato operace je naprogramovaná jako synchronní, a protože je pro velké virtuální světy časově relativně náročná (i přes hierarchickou optimalizaci), může způsobit zastavení celé simulace

na několik desítek milisekund, což při množství agentů, kterých se může pohybovat až kolem několika stovek (při evoluci) nebo desítek (při 3d zobrazení), může způsobit značné zpomalení běhu. Tato operace je ideálním kandidátem pro paralelizaci, tzn., že každý požadavek na hledání cesty by dostal přidělené vlákno ze zásobníku vláken aplikace a toto vlákno by paralelně beželo s během simulace a nezastavovalo by jej. Tento přístup není v této práci použit, protože se s ním nepočítalo od začátku a dodatečná paralelizace tohoto algoritmu v kombinaci s návrhem datových struktur v této práci by byla značně časově náročná. Jedná se však o jedno z možných rozšíření.

Jak vyplývá z postupu v minulé podkapitole, značný vliv na rychlost běhu evoluce bude mít i počet agentů v každé generaci. Tato hodnota také značně koreluje s rychlostí konvergence k suboptimálnímu řešení v evoluci. Jak se ukázalo, příliš malá hodnota množství agentů způsobuje nízkou diverzitu FSM v populaci, což vede k tomu, že nalezená a generovaná řešení nejsou dostatečně rozmanitá a evoluce konverguje daleko pomaleji k optimálním hodnotám.

Pokud je však agentů v simulaci příliš mnoho, dochází k jejich velké interakci, kdy se na malém prostoru ovlivňuje velké množství jedinců, a než aby jednotlivá FSM např. v plné síle ukázaly svoji schopnost vedení agenta k těžení velkého množství zdrojů, tento agent je zabit nějakým agentem z jiného kmene. Velké množství agentů rovněž vede k časovému zpomalení evoluce, protože v každém kroku evoluce se musí obsloužit větší množství na raycasting, hledání cesty, vzájemné kolize agentů, atd.

3.5 Zobrazení v režimu GUI

Pro zobrazení detailů chování agentů existuje v této práci testovací zobrazení v režimu 3D. Agenti se zde v “reálném čase” pohybují, interagují mezi sebou, vyhýbají se objektům atd. Veškeré objekty virtuálního světa zde nabývají své symbolické podoby, mají velikost, která odpovídá měřítkům virtuálního světa apod.

Pro zobrazení byl zvolen open source 3d engine Irrlicht, který je naprogramován, rovněž jako zbytek této práce, v jazyce C++. Tento engine podporuje všechny moderní zobrazovací API, tedy:

- DirectX 8
- DirectX 9
- DirectX 10
- OpenGL 2.0
- softwarový renderer

Mezi další přednosti tohoto enginu patří:

- strmá učící křivka vyplývající z jednoduchého API
- podpora otevřených datových formátů
    - 3d studio max modely
    - collada modely
    - jpg, bmp, png, tga textury
    - obj (lightwave) modely

    - podpora shaderů
    - multiplatformní podpora (OS X, Win32, Linux)
    - podpora vlastního GUI
    - dynamické osvětlení
    - animace 3d modelů

3.5.1 Informace o GUI

GUI verze se 3d zobrazením slouží k testování a ladění chování, které bylo získáno pomocí modulu evoluce. Tyto dva moduly jsou na sobě naprosto nezávislé. Pro účely testování a správného náhledu na chování agentů je možno, kromě např. pouze vizuálního doplňku v podobě dynamického stínování, různě upravovat parametry zobrazení, např.:

- zrychlení či zpomalení běhu simulace
- zvětšení či zmenšení velikosti kroku, který agent ujde za jeden snímek
- zastavení AI agentů (neprobíhá jejich aktualizace, žádné kroky ve FSM se neprovádějí)
- natočení scény
- posunutí scény
- vypsání informací do scény
    - o agentech
    - o objektech ve virtuálním světě
- přiblížení scény
- přepnutí pohledu na scénu “z pohledu agenta”, lze vidět na ilustraci 13.

Ilustrace 13: Náhled na scénu v režimu pohledu agenta

4 Závěr a porovnání ovládacích schémat agentů

4.1 Boids a flocking

Agenti typu boids a ovládací schéma jejich pohybu zvané flocking se hodí pouze pro velmi specifické nasazení. Použití lze vidět zejména při simulaci pohybu (pouze pohybu) velkého množství agentů, přičemž se zde nekladou téměř žádné požadavky na chování jednotlivce, ale pouze na chování hejna - davu. Proto se agenti typu boids hodí např. pro tyto použití:

- stáda zvířat
- hejna ptáků
- hejna ryb

Toto ovládací paradigma lze kombinovat např. s FSM, kdy tento ovládací prvek má na starosti řízení detailnějšího chování na úrovni jednotlivce. K přepnutí na mód ovládání pomocí FSM může dojít např. při vzniku nějaké události, třeba při dosednutí hejna ptáků na zem a jejich individuálním pohybu po povrchu.

Na závěr je vhodné poznamenat, že řídící schéma boids agentů není vhodné pro řízení interakce agentů s prostředím, např. přenášení předmětů je naprosto nevhodné a je nutné toto kombinovat s např. FSM. Je tedy vhodné poznamenat, že ovládací schéma boids se hodí pokud simulujeme chování agentů hlavně na makroskopické úrovni.

4.2 Rozhodovací stromy

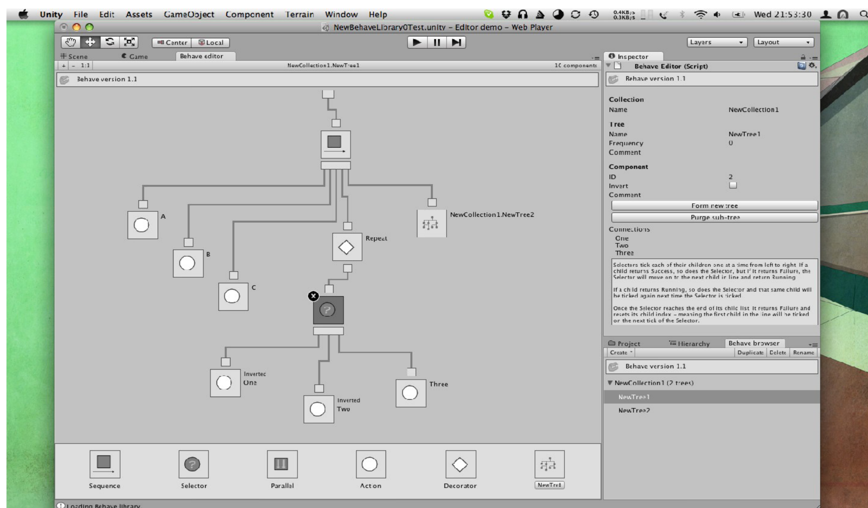
Toto ovládací schéma je vhodné zejména pro agenty a virtuální entity, kteří se nacházejí ve známém prostředí, kde jsou všechny vazby a interakce do předu známé a není příliš obtížné postihnout všechny možnosti pro návrháře rozhodovacích stromů. Rovněž je vhodné pro prostředí, kde reakce a chování agentů nejsou očekávány příliš komplexní, protože s rozsáhlými rozhodovacími stromy se uživateli špatně a neintuitivně pracuje.

Pro dosažení komplexnějšího chování takovýchto agentů je vhodný např. koncept, jako předvedl GOAP (Goal Oriented Approach Planner) v počítačové hře F.e.a.r., kde nepřátelé v (pro ně) předem

známém prostředí byli schopni aktivně využívat objekty v tomto prostředí, krýt se za nimi před střelbou hráče, hledat si nová krycí místa, případně si tato místa aktivně vytvářet (např. povalením skříně).

Toto ovládací schéma lze použít na nepřátele a virtuální entity v počítačových hrách a simulacích, kde není nutná ze strany agentů změna prostředí a jeho aktivní využívání.

Značnou nevýhodou zde je, že interakce a samotné prostředí musí být dopředu známé a jakákoliv např. dodatečná editace, např. přidáním nového interaktivního prvku musí být promítnuta i do ovládacího schématu rozhodovacího stromu. Totéž platí i pro GOAP planner.



Ilustrace 14: Editor rozhodovacích stromů v enginu Unity [8]

4.3 Jednoduchá FSM

Tyto mají zhruba stejnou (z pohledu návrháře) vyjadřovací schopnost, ale často je pro ně složitější manuální konstrukce. Konečné automaty je vhodné použít pro jednoduší ovládání entit ve virtuálních světech, např. popis stráží přemisťujících se mezi několika místy. Ovšem už je zde obtížnější popis řízení tzv. přerušení z činnosti, kdy např. stráže mohou spatřit hrozbu během svojí chůze a nějak aktivně se jí postavit. Zde už je potřeba víceúrovňové řízení.

Vylepšením tohoto konceptu jsou hierarchická FSM, které do značné míry řeší dva hlavní problémy tohoto konceptu: složitou konstrukci pro návrháře a jednoúrovňové řízení.

Dalším konceptem, který doplňuje tento stávající, je doplnění ovládání FSM o neuronovou síť, která podle svého naučení v učícím módu aktivně ovlivňujě FSM např zapínáním a vypínáním některých vazeb ve FSM, případně změnou predikátů a jejich parametrů, které se vyskytují ve FSM. Tento řídící koncept však nebyl v praxi zkoušen a autor se s ním nikde prakticky nesetkal, jedná se pouze o návrh doporučený autorem této práce.

Značnou nevýhodou zde je, že interakce a samotné prostředí musí být dopředu známé a jakákoliv např. dodatečná editace, např. přidáním nového interaktivního prvku, musí být promítnuta i do ovládacího schématu konečného automatu.

4.4 Hierarchické FSM

Vyjadřovací schopnost je (z pohledu návrháře a implementátora) v zásadě stejná jako u jednoduchých FSM, ale z hlediska návrháře je hierarchické uspořádání jednodušší na editaci, kdy si např. návrhář může jednotlivé úrovně řízení skrývat a pracovat pouze s aktuální úrovní, aniž by ho vyrušovaly imlementační detaily úrovní nižších či vyšších.

Toto ovládací schéma rovněž postihuje i druhý nedostatek jednoduchých FSM. Tímto nedostatkem je jednoúrovňovost. Hierarchická FSM umožnují jednodušší popis chování agenta, který má krátkodobé a dlouhodobé cíle, přičemž může být např. vyrušen nízkoúrovňovou událostí, která mu změní řízení na vyšší úrovni apod.

Zejména diky většímu usnadnění hierarchického náhledu a z něho vyplývající snadnosti popisu komplikovanějšího chování se hierarchická FSM hodí pro popis složitějšího chování virtuálních entit, jako jsou např. nepřátelé ve 3d akčních hrách využívající aktivně svoje prostředí, případně toto prostředí měnící. Tato technika byla použita např. v počítačových hrách Half-Life 2, kde je chování a diverzita nepřátel značně komplexní nebo např. ve hře Battlefield: Bad Company 2, kde se virtuální nepřátelé byli schopni vypořádat s destrukcí terénu a sami tuto destrukci i iniciovali za dosažením cile tvoření nových cest a útočných výhledů na hráče.

Značnou nevýhodou zde je, že interakce a samotné prostředí musí být dopředu známé a jakákoliv např. dodatečná editace, např. přidáním nového interaktivního prvku musí být promítnuta i do ovládacího schématu konečného automatu.

Podobně jako u jednoduchých FSM, autor navrhuje i zde toto schéma doplnit o kontrolu pomocí neuronové sítě pro dosažení více diverzitního chování (např. každý agent by měl neuronovou síť jinak naučenou, takže by měl svým způsobem individuální osobnost).

4.5 GravFSM

Zde se jedná o autorem navržený koncept, který dosud nebyl experimentálně ověřen, ale autor předpokládá dvě jeho velká pozitiva:

- automatická možnost učení a přizpůsobení
- snadné použití v evolvujícím prostředí

Možnost učení vychází přímo z konceptu GravFSM, kdy podle podílu na úspěchu agenta na dilčích činnostech jsou jednotlivé kontrolní hmotné body v agentově prostoru pozic k sobě iterativně přibližovány, čímž dochází k větší pravděpodobnosti jejich použití i v příštím případě.

Snadné použití v evolvujícím prostředí vychází z toho, že celé GravFSM je velmi snadné zmutovat nebo skřížit s jiným GravFSM. Mutace spočívá v pseudonáhodném přemístění některých hmotných kontrolních bodů nebo atraktorů a křížení spočívá ve výměně předem daného množství hmotných kontrolních bodu a atraktorů mezi GravFSM.

Diky těmto faktům můžeme GravFSM z počátku inicializovat náhodně a nechat ho evolučně se přizpůsobit prostředí.

Nevýhodou může být zpomalení výpočtu při větším počtu dimenzí prostoru pozic a rovněž delší doba učení a ladění, jelikož informace a "zkušenosti" jsou zde uloženy implicitně a nejsou na pozicích kontrolních bodů a atraktorů tak zřejmě vidět, jako např v případě jednoduchých FSM.

Jedná se o koncept navržený autorem této práce a je nutno jej vyzkoušet, v této práci byl zmíněn pouze jako zajímavost, nicméně autor by se k němu rád v budoucnosti vrátil.

4.6 Zhodnocení řídícího konceptu použitého v této práci

Základní koncept řídícího konceptu použitého v této práci je v podstatě stejný jako u jednoduchého FSM. Pozitivem je zde ovšem řešení jednoho problému klasického FSM, a to nevhodné struktuře vzhledem k evolučním úpravám. Proto bylo navrženo FSM se strukturou takovou, aby byla imunní vůči změnám jako je mutace a křížení a po aplikaci těchto operací vždy vznikl plnohodnotný, validní a použitelný objekt.

Bylo toho dosaženo použitím linearizované struktury a relativních skoků, které jsou v případě "překročení" konce FSM zarovnány pomocí operace modulu na jeho začátku.

Hlavním cílem této práce bylo vytvořit koncept, který by byl schopný umožnit agentům pomocí evoluce naučit se činnost, která jim nebyla programátorem dána - nebyla napevno naprogramována, ale bylo jí dosaženo evoluční optimalizací s fitness funkcí zvýhodňující ty jedince, kteří se chovají tak, jak je požadováno.

V tomto případě bylo dosaženo cíle, kdy jsme po agentech chtěli, aby se naučili nosit zdroje natěžené na jednom místě virtuálního světa na svoji domovskou základnu, případně během této činnosti se vyhnuli útoku nepřátelských agentů nebo tyto agenty napadly. Tohoto cíle bylo dosaženo, agenti se takto chovají a všechny 4 testovací kmeny v testovací úrovni jsou schopny natěžit omezené množští zdrojů.

Během evolučního procesu také vznikly ve FSM agentů a jejich chování různé artefakty, viz. např. schopnost využití chyby v simulátoru - teleportace, které ukazují, že evoluce je často velmi progresivní a najde si jakkoliv absurdní a agresivní cestu k tomu zvětšit hodnocení jedince pomocí fitness funkce.

4.7 Závěrečné zhodnocení práce

V této práci se autor zabýval nastudováním a implementací jednoho z ovládacích mechanismů pro řízení inteligentních agentů ve virtuálním světě. Pro implementaci byl vybrán přístup používající konečný automat, podléhající evoluci, s cílem naučení agentů činnosti, která je vyžadována a hodnocena přirozeným výběrem v evolučním algoritmu.

Tento cíl byl splněn, jelikož agenti se samovolně díky evoluci (která zvýhodňovala tyto jedince) "naučili" težit zdroje, nosit je na základnu a bojovat s cizími agenty, kteří náleží cizím kmenům. Byly vytvořeny dva programové moduly, které sdílejí některé části, jeden k simulaci a evoluci FSM agentů, přičemž zde není třeba žádná interakce ze strany uživatele. Druhý modul je režim se zobrazením v 3D enginu, který umožnuje pohyb s kamerou, zobrazuje 3D modely objektů virtuálního světa, lze měnit některé vlastnosti agentů a sledovat jejich chování, případně zrychlit či zpomalit simulaci chování agentů, nebo si její běh úplně zastavit.

Pro vypracování práce byly provedeny tyto kroky. Byla nastudována různá schémata ovládání entit ve virtuálních světech, s některými z nich byly provedeny experimenty a nakonec jedno zvolené schéma bylo implementováno v rámci programu v C++. O ostatních schématech je uvedeno detailní pojednání v teoretické části, protože s některými schématy nešlo provádět příliš kvalitativní či kvantitativní pokusy, je alespoň u experimentu s boidy (který autor implementoval zvlášť v enginu Unity) uveden obrázek skupinového chování.

Autor si prohloubil znalosti a zkušenosti v práci s evolučními výpočetními technikami a implementoval s nimi evoluci konečných automatů, jejichž problematiku rovněž nastudoval.

Veškeré zde uvedené ovládací přístupy byly nastudovány, často konzultována dokumentace komerčních produktů (např. Source SDK), zároveň s některými zde uvedenými paradigmaty byly provedeny detailní experimenty, přičemž nejdetailněji se autor zabýval evolucí konečných automatů. Autor rovněž implementoval chování boidů v separátním projektu.

Jednotlivé přístupy k ovládání virtuálních entit byly porovnány v poslední kapitole, kde bylo zároveň doplněno, s čím je vhodné který přístup kombinovat, případně pro kterou oblast je vhodný.

Při práci s hlavním projektem této práce bylo provedeno několik desítek řízených evolucí s populací agentů čítajících 10 - 2500 jedinců na jednu generaci, při počtu 1000 - 5000 AI kroků na jednu populaci. Za celou dobu experimentů se simulací bylo vyprodukováno téměř 4000 vybraných konečných automatů. Nejdelší běh evoluce (s největší velikostí populace) trval 22 hodin na stroji Core i7 Quad 2.4 GHz z 8GB RAM.

Pokud by byla možnost na této práci dále pokračovat, autor by se rád zaměřil na zlepšení možnosti komunikace agentů mezi sebou a větší prosazení této možnosti při zvýhodňování během evoluce. Větším a náročnějším cílem je realizace jiných paradigmat v prostředí této práce, např. FSM ovládaného neuronovou sítí nebo autorův koncept GravFSM, který potřebuje detailnější prozkoumání a více experimentů. Dalším možným rozšířením hlavně virtuálního světa by byla schopnost operace a umístění předmětů ve skutečně 3D rozměrech - v této práci je sice možný 3D mód zobrazení, ale veškeré děje se odehrávají v rovině.

Literatura

[1] Mařík a kolektiv, Umělá inteligence 2, Praha, Academia, 1997, ISBN 80-200-0504-8

[2] Mařík a kolektiv, Umělá inteligence 1, Praha, Academia, 1993, ISBN 80-200-0496-3

[3] Parker Conrad, Boids [online], c1995-2010, 23. května 2010, [cit. 11.5.2011], Dostupné na WWW: http://www.vergenet.net/~conrad/boids/index.html

[4] Mikko Mononen, Heightfield Layer Progress pt. 3 [online], 11.3.2011, [cit. 11.5.2011], Dostupné na WWW: http://digestingduck.blogspot.com/2011/03/heightfield-layer-progress-pt-3.html

[5] Wikipedia contributors, Konečný automat [online], 4.11.2010, [cit. 11.5.2011], Dostupné na WWW:
http://cs.wikipedia.org/wiki/Kone%C4%8Dn%C3%BD_automat

[6] Alex J. Champandard, The Gist of Hierarchical FSM [online], 5. 9., 2007, [cit. 11.5.2011], Dostupné na WWW: http://aigamedev.com/open/articles/hfsm-gist/

[7] Wikibooks contributors, Cognitive Psychology and Cognitive Neuroscience/Behavioural and Neuroscience Methods [online], 4.3.2011, [cit. 11.5.2011], Dostupné na WWW:
http://en.wikibooks.org/wiki/Cognitive_Psychology_and_Cognitive_Neuroscience/Behavioural_and_Neuroscience_Methods

[8] Emil Johansen, Preview – AngryAnt.com [online], 22.2.2009, [cit. 11.5.2011], Dostupné na WWW: http://eej.dk/angryant/wp-content/themes/angryant/images/BehaveTreeEditor.png

Seznam příloh

Příloha 1. Ukázky některých zdrojových textů a algoritmů

Příloha 2. CD/DVD s programem, zdrojovými kódy a návodem k použití

Příloha 1: zdrojové kódy a pseudokódy

1. Ovládání boidů

Kontrolní smyčka všech boidů:

```
PROCEDURE move_all_boids_to_new_positions()
     Vector v1, v2, v3
     Boid b
     FOR EACH BOID b
            //aplikace pravidel
            v1 = rule1(b)
            v2 = rule2(b)
            v3 = rule3(b)
       // vektory z jednotlivých pravidel jsou pricteny k aktuální
rychlosti
            b.velocity = b.velocity + v1 + v2 + v3
            // podle rychlosti je modifikována pozice
            b.position = b.position + b.velocity
     END
END PROCEDURE
```

Pozice každého boida je upravena pomocí sady pravidel, jak uvádí výše uvedený pseudokód.

Pravidlo 1:
"Boidové se snaží pohybovat k pomyslnému hmotnému bodu v centru ostatních boidů"

```
PROCEDURE rule1(boid bJ)
  Vector pcJ
  FOR EACH BOID b
       // aktuálního boida vynecháme
       IF b != bJ THEN
                         pcJ = pcJ + b.position
       END IF
  END
 // spočítáme průměr všech pozic
 pcJ = pcJ / N-1
 // výsledný vektor zmenšíme
 RETURN (pcJ - bJ.position) / 100
END PROCEDURE
```

Při aplikaci tohoto pravidla se spočítá pruměr ze všech vektorů pozic ostatních agentů. Výsledkem je vektor. Tento vektor je ještě třeba zmenšit, aby se boidové nepohybovali příliš rychle.

Výsledek: boidové se pohybují k pomyslnému středu hejna.

Pravidlo 2:
"Boidové si snaží udržet minimální vzdálenost od ostatních objektů (i jiných boidů)"

```
PROCEDURE rule2(boid bJ)
```

```
        Vector c = 0;
        FOR EACH BOID b
              IF b != bJ THEN
              // pokud vzdálenost vybraného boida prekročí hranici
                    IF |b.position - bJ.position| < 100 THEN
                    // odečteme rozdíl vzáleností od sumačního vektoru
                          c = c - (b.position - bJ.position)
                    END IF
              END IF
         END
         RETURN c
END PROCEDURE
```

Výsledkem je vektor, přes který v cyklu iterujeme a odčítáme od něho rozdíly vzdáleností aktuálního boida a právě iterovaneho boida.

Výsledek: boid se snaží udržet si vzálenost od svých nejbližších sousedů

Pravidlo 3:

“Boidové si snaží srovnat svoji rychlost s ostatními.”

```
PROCEDURE rule3(boid bJ)
        Vector pvJ
       // vypocet prumeru rychlosti ostatnich boidu
       FOR EACH BOID b
             IF b != bJ THEN
                   pvJ = pvJ + b.velocity
             END IF
        END
        pvJ = pvJ / N-1
       // zmenšení výsledného vektoru
       RETURN (pvJ - bJ.velocity) / 8
END PROCEDURE
```

Výsledek: každý boid se snaží leťet tak rychle, aby jeho rychlost zhruba odpovídala průměrné rychlosti ostatních boidů.

2. Ukázka zdrojového kódu waypointu

```
class CWaypoint: public CBaseObject {
private:
   // unikatni ID v ramci systemu waypointu
   unsigned int          m_waypoint_id;
   // citac waypointu - je nutny pri pridelovani ID
   static unsigned int        m_waypoint_counter;

   typedef CWaypoint * waypointPtr_t;

   // pole s ukazately na sousedni waypointy
   vector<waypointPtr_t>    m_neighbours;
   // pole se vzdalenostmi pro i-ty waypoint
   vector<float>            m_distances;
   // pokud je soused pritomen, jeho id je v mnozine
   set<int>                m_neighbourSet;    // COPY
   // rodicovsky segment
```

```
        waypointSegmentPtr_t    m_segmentPtr;        // COPY
        // position
        SPosition2D            m_position;        // COPY
         // byl navstiven? toto se pouziva pri hledani cesty pro vynechani
jiz navstivenych waypointu
        bool                   m_visited;         // COPY

        // odkud jsme do toho waypointu prisli (z jakeho waypointu)?
        waypointPtr_t           m_from;    // COPY

        // usla cesta - jak je dlouha globalni usla cesta az sem_
        double                  m_globalDistance;    // COPY
     …
```

3. Ukázka zdrojového kódu segmentu waypointů

```
     class CWaypointSegment : public CBaseObject {
     private:
      typedef CWaypointSegment * waypointSegmentPtr_t;
             // pole pointeru na sousedy
            waypointSegmentPtr_t    m_neighbours[LAST];
            // lin. pole s pointery na waypointy, ktere segment obsahuje
            vector<waypointPtr_t>    m_waypoints;
// mnozina IDu waypointu ktere mame v segmentu, pro snadne dotazovani
            set<int>                m_waypointSet;
            // linearni pole s ukazately na objekty, ktere segment obsahuje
            vector<CWorldObject*>    m_world_objects;
//mnozina identifikatoru objektu, ktere segment obsahuje
            set<int>                m_wo_set;
            static unsigned int        m_segment_counter;
            unsigned int            m_segment_id;
          // sirka segmentu
            static float            m_width;
             // ID agentu, pro ktere je tento segment zakazan
            // waypoint segment forbidden for int id of AI agent
            set<int>                m_agent_forbidden_flags;
             // ID skupiny agentu, pro ktere je tento segment zakazan
             // forbidden for group agent id
            set<int>                m_group_forbidden_flags;
             // pozice v poli segmentu
             // na jakem sloupci a radku se segment nachazi
            unsigned int            m_col;
            unsigned int            m_row;
          // pointer na pole segmentu - rodicovske pole
            CSegmentArray2D*        m_arrPtr;
             // byl segment navstiven pri hledani cesty?
             bool                    m_visited;
            // odkud jsme do segmentu prisli?
            waypointSegmentPtr_t    m_from;
            // true - muze se tudy planovat cesta
            // hierarchicke vyhledavani
            bool                    m_enabled;
            // true - volno / false - je zde prekazka
            bool                    m_forbidden;
             bool                    m_must_go;
        bool                    m_tmp_forbidden;
     …
```

4. Zdrojový kód struktur SPredicateInfo a SCommandInfo

Struktura SPredicateInfo:

```
struct SPredicateInfo {
 bool    m_parameterNeeded[6]; // je potreba i-ty parametr?
 int        m_parameterMax[6]; // maximalni povolene hodnoty
 int        m_parameterMin[6]; // minimalni povolene hodnoty
};
```

Struktura SCommandInfo:

```
struct SCommandInfo {
 bool    m_parameterNeeded[2]; // je i-ty parametr vyzadovan
 int        m_parameterMax[2]; // maximalni povolene hodnoty
 int        m_parameterMin[2]; // minimalni povolene hodnoty
};
```

5. Ukázka inicializace predikátu

```
 m_predicates[PN_OR] = OrPredicate;
 m_info[PN_OR].m_parameterNeeded[0] = true;
 m_info[PN_OR].m_parameterMax[0] = PN_IS_RESOURCE_IN_SIGHT;
// posledni pred PN_AND a PN_OR - nesmi se generovat vnoreny predikat
 m_info[PN_OR].m_parameterMin[0] = 0;
 m_info[PN_OR].m_parameterMax[1] = PN_IS_RESOURCE_IN_SIGHT;
// posledni pred PN_AND a PN_OR - nesmi se generovat vnoreny predikat
 m_info[PN_OR].m_parameterMin[1] = 0;
 m_info[PN_OR].m_parameterNeeded[1] = true;
```

6. Statické řízení agenta

1. vypočítej průměr všech vektorů pozic predátorů v okolí agenta - získáme "prostřední bod mezi predátory"
2. vypočítej novou pozici agenta tak, aby nová pozice agenta byla ve větší vzdálenosti od prostředního bodu mezi predátory než pozice předchozí
3. pro všechna nebezpečná místa v paměti najdi nejbližší nebezpečné místo
4. vypočítej novou pozici agenta tak, aby tato pozice byla ve větší vzdálenosti od nejbližšího nebezpečného místa než stará pozice agenta.

7. Evoluce GravFSM

1. vygeneruje N jedinců: GravFSM, kde jsou náhodně umístěny uzly a kolem nich jejich atraktory

2. spočítáme hodnotící funkci každého FSM - změříme jeho úspěšnost oproti referenčnímu řešení - např. Tím, že provedeme několik kroků simulace s agenty
3. pokud je nejlepší řešení dostatečné, konec
4. nejlepší jedince - jedince s největším hodnocením vybereme ke křížení a mutacím
5. na základě výsledku křížení a mutací vytvoříme novou populaci
6. skok na 2.

8. Evoluce v genetickém algoritmu

1. vytvoř N náhodných jedinců, každého s jiným genotypem
2. z genotypu vygeneruj testovatelné jedince
3. otestuj jedince a pro každého spočítej hodnotu fitness function (hodnotící funkci).
4. najdi nejlepší jedince - s nejlepší hodnotou fitness function a proveď s nimi křížení a mutace, z těchto potom vygeneruj novou populaci
5. (volitelné) do budoucí populace překopíruj i M nejlepších “rodičů”
6. pokud jsme dosáhli optimální velikosti hodnoty hodnotící funkce, konec
7. opakuj od kroku 2

9. Algoritmus generování systému waypointů

```
for (segment v poli segmentu)
{
    for (waypoint v segmentu)
    {
        for (soused waypointu)
        {
            usecka = spoj_mezi(soused, waypoint)
            for (predmet v predmety_v_segmentu)
            {
            if (nekoliduje(usecka, predmet))
            {
                pridej_cestu(usecka, soused, waypoint)
            }
            }
        }
    }
}
```

10. Algoritmus generování světa

```
 for pocet_objektu_ke_generovani {
        int typ = rand() % pocet_typu_objektu;
        do {
            sloupec = rand() % sirka_sveta;
            radek = rand() % sirka_sveta;
        } while (!pozice_neobsazena(sloupec, radek) )

        pridej_objekt(typ, sloupec, radek);
    }

bool pozice_neobsazena(sloupec, radek)
{
    segment = pole_segmentu.get(sloupec, radek)
    return segment->pocet_objektu() == 0;
}
```

11. Pseudokód vysokoúrovňového hledání cesty

```
bool path_find_high_level(start_segment, target_segment)
{
    vysledek = false;
    zakaz_vsechny_segmenty_pro_nizkourovnove_hledani();
    fronta.vloz(start_segment);
    while (fronta.neprazdna())
    {
        aktualni_segment = fronta.vyber();
        if (segmenty_souhlasi())
        {
            povol_segmenty_na_ceste_sem();
            vysledek = true;
        }
        fronta.vloz(vyber_nejperspektivnejsiho_souseda());
        fronta.vloz(ostatni_sousedy);
    }
    return vysledek;
}
```

11. Pseudokód nízkoúrovňového hledání cesty

```
bool path_find_low_level(start_waypoint, target_waypoint)
{
    vysledek = false;
    start_segment = start_waypoint->rodicovsky_segment;
    target_segment = target_waypoint->rodicovsky_segment;
    vysledek = path_find_high_level(start_segment, target_segment)
    if (!vysledek)
        return false;
    fronta.vloz(start_waypoint);
    while (fronta.neprazdna())
    {
        aktualni_waypoint = fronta.vyber();
        if (aktualni_waypoint == target_waypoint)
        {
            rekonstruuj_cestu();
            vysledek = true;
        }
        fronta.vloz(vyber_nejperspektivnejsiho_souseda());

        foreach (soused in waypoint->sousede)
        {
                if (soused->zakazan)
```

```
                    continue;
                fronta.vloz(soused);
            }
        }
        return vysledek;
    }
```

12. Algoritmus generování uzlu

1. do n ulož počet generovaných predikátů, nechť je to náhodná hodnota o maximální velikosti
2. nastav čítač na 1
3. pokud čítač je roven n, konec
4. vygeneruj predikát
    1. vyber index implementace predikátu, náhodnou hodnotu v rozsahu od 0 až po POCET_PREDIKATU
    2. načti informace o limitech parametrů predikátu ze struktury SPredicateInfo
    3. podle daných limitů vygeneruj náhodné hodnoty parametrů predikátu
5. vygeneruj příkaz
    1. vyber index implementace příkazu, náhodnou hodnotu v rozsahu od 0 až po POCET_PRIKAZU
    2. načti informace o limitech parametrů příkazu ze struktury SCommandInfo
    3. podle daných limitů vygeneruj náhodné hodnoty parametrů příkazu
6. vygeneruj náhodnout hodnotu relativního skoku v FSM o maximální hodnotě plánovaného počtu uzlů v FSM, přičemž skok může být i záporný, pokud je adresa uzlu nevalidní a je mimo pole uzlů, použij na tuto adresu operaci modulo, čímž se dostaneš do regulérního pole
7. zvyš čítač o 1
8. skok na bod 3.

13. Mutace uzlů FSM

```
    void MutateNode() {
      // nahodne vybereme index subuzlu (trojice predikat-prikaz-skok), //ktery        budeme
modifikovat
      int subNodeIndex = rand()% this->subNodesCount;
```

```
// vybereme, kterou cast poduzlu budeme mutovat (predikat, prikaz  // nebo skok)
 int randIndex = rand() % 3;
 int mutateOrGenerate = rand()%100;
 switch (randIndex)     {
        case 0:             // mutujeme predikat
        if (mutateOrGenerate > 50) {
               //   pozmeni   parametry   predikatu   podle   struktury   SPredicateInfo
               pozmenParametryPredikatu();
        }
        else
        {
               predikat = generujNovyPredikat();
        }
        case 1:             // mutujeme prikaz
        if (mutateOrGenerate > 50){
         // pozmeni parametry predikatu podle struktury SCommandInfo
                 pozmenParametryPrikazu();
        }
        else
        {
               predikat = generujNovyPrikaz();
        }
        case 2:
   // mutujeme skok
   step = - (pocetUzlu/2) + (rand() % pocetUzlu);
 }
}
```

14. Běh evoluce

1. nastav počáteční hodnoty, k je počet kroků evoluce (generací), l je počet kroků umělé inteligence za jednu generaci, m je počet agentů
2. načti a vytvoř virtuální svět, částečně podle popisu v souboru a částečně ho procedurálně vygeneruj
3. vytvoř první generaci, tj. m-krát proveď:
   1. vytvoř agenta
   2. přiřad mu náhodné FSM
   3. přiřaď agentovi kmen (každý agent musí být v nějakém kmeni)
   4. najdi volné místo ve virtuálním světě blízko základny agentova kmene
4. proveď k kroků simulace chování umělé inteligence, tj. v každém kroku proveď:
   1. zkontroluj kolize aktuálního agenta s jinými agenty
      1. pokud k ní dojde a agenti jsou nepřátelší, uprav hodnoty jejich životů a statistiky
   2. proveď příkazy, které jsou naplánované v agentově FSM, tzn. podle prekátů proveď příkazy a skoč na nový uzel FSM daný aktuální pozicí v FSM a velikostí relativního skoku

    3. pokud se má agent přesunout, uprav mu souřadnice
    4. vypočítej viditelnost - spočítej a identifikuj všechny objekty, které jsou v agentově zorném poli, uprav mu příslušné řídící struktury
    5. pokud je požadavek ze strany FSM na hledání nové cesty, proveď toto
    6. uprav agentovi řídící struktury podle aktuální situace
    7. mrtvé agenty znehybni
    8. pokud agent došel na cíl své cesty, změň příznaky pro predikáty ve FSM
    9. ulož hodnoty agentových dosažených cílů do jeho statistik
5. ohodnocení agentů:
    1. pro každého agenta spočítej velikost fitness funkce
    2. seřaď agenty podle velikosti hodnoty fitness funkce od největší po nejmenší
6. křížení a mutace:
    1. vezmi prvních n nejlepších agentů a skřížením jejich FSM vytvoř FSM pro potomky
    2. tyto FSM podle příslušných nastavení zmutuj
    3. vytvoř potomky s těmito novými FSM a zbytek nové generace naplň těmi nejlépe hodnocenými z předchozí generace
    4. ulož FSM nejlepšího agenta do souboru
7. pokud počet kroků evoluce nedosáhl K, skoč na bod 5
8. vypiš výsledky, ulož nejlepší agenty (jejich FSM) do souboru