České vysoké učení technické

Fakulta elektrotechnická

Katedra počítačů

logocvut

Bakalářská práce

Umělá inteligence v tahových strategiích

*Lukáš Beran*

Vedoucí: titul Michal Hapala titul

Studijní program: Softwarové technologie a management

Studijní obor: Softwarové inženýrství

Leden 2011

Poděkování

First of all I would like to thank my parents, my family, and especially my little sister. She definitely gave me a different view of this world.

Additionally, I would like to thank Dr. Vlastimil Havran for his kind leadership and invaluable remarks.

Finally, I would like to express my gratitude to my few very close friends, who always tried to help me accomplish things I wanted to achieve.

Declaration – Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 17. ledna 2011 …………………………………..

Lukáš Beran

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque porta vulputate dui eget convallis. Aliquam erat volutpat. Nulla facilisi. Vestibulum ante libero, mollis ac fringilla id, malesuada vel lectus. Cras id tellus dolor, vel consectetur ipsum. Aenean dignissim, sapien et viverra fermentum, lectus enim suscipit quam, id mollis neque dolor non magna. Integer volutpat est quis erat dapibus mattis. In a purus eget ligula egestas fringilla. Aenean semper risus a dolor pretium dapibus. Donec ac justo lacus. Mauris lobortis nulla lorem, vitae porttitor sapien. Sed malesuada tempus odio vitae mollis. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque porta vulputate dui eget convallis. Aliquam erat volutpat. Nulla facilisi. Vestibulum ante libero, mollis ac fringilla id, malesuada vel lectus. Cras id tellus dolor, vel consectetur ipsum. Aenean dignissim, sapien et viverra fermentum, lectus enim suscipit quam, id mollis neque dolor non magna. Integer volutpat est quis erat dapibus mattis. In a purus eget ligula egestas fringilla. Aenean semper risus a dolor pretium dapibus. Donec ac justo lacus. Mauris lobortis nulla lorem, vitae porttitor sapien. Sed malesuada tempus odio vitae mollis. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Abstrakt

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque porta vulputate dui eget convallis. Aliquam erat volutpat. Nulla facilisi. Vestibulum ante libero, mollis ac fringilla id, malesuada vel lectus. Cras id tellus dolor, vel consectetur ipsum. Aenean dignissim, sapien et viverra fermentum, lectus enim suscipit quam, id mollis neque dolor non magna. Integer volutpat est quis erat dapibus mattis. In a purus eget ligula egestas fringilla. Aenean semper risus a dolor pretium dapibus. Donec ac justo lacus. Mauris lobortis nulla lorem, vitae porttitor sapien. Sed malesuada tempus odio vitae mollis. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque porta vulputate dui eget convallis. Aliquam erat volutpat. Nulla facilisi. Vestibulum ante libero, mollis ac fringilla id, malesuada vel lectus. Cras id tellus dolor, vel consectetur ipsum. Aenean dignissim, sapien et viverra fermentum, lectus enim suscipit quam, id mollis neque dolor non magna. Integer volutpat est quis erat dapibus mattis. In a purus eget ligula egestas fringilla. Aenean semper risus a dolor pretium dapibus. Donec ac justo lacus. Mauris lobortis nulla lorem, vitae porttitor sapien. Sed malesuada tempus odio vitae mollis. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

„Some argue that in the very long term,

rendering may best be solved by some variant

of ray tracing, in which huge numbers of rays

sample the environment for the eye’s view of each frame.

And there will also be colonies on Mars,

underwater cities, and personal jet packs. “

Real-time Rendering, Möller and Haines, 1999

ונתווקת הדבא אל דוע  
„Our hope is not yet lost”

Hatikvah, 5th verse

Obsah

[Seznam obrázků xi](#_Toc284690420)

[Seznam tabulek xi](#_Toc284690421)

[1 Úvod 13](#_Toc284690422)

[1.1 Historie 13](#_Toc284690423)

[1.2 Iluze inteligence 13](#_Toc284690424)

[2 AI algoritmy 15](#_Toc284690425)

[2.1 Stavový automat 15](#_Toc284690426)

[2.2 Goal – driven 15](#_Toc284690427)

[2.3 Fuzzy logika 15](#_Toc284690428)

[2.4 Umělý život 15](#_Toc284690429)

[2.4.1 Mazlíčci 15](#_Toc284690430)

[2.4.2 Společenské simulace 15](#_Toc284690431)

[2.4.3 Hry na Boha 16](#_Toc284690432)

[2.4.4 Evoluční hry 16](#_Toc284690433)

[2.4.5 A-Life a tahové strategie 16](#_Toc284690434)

[2.5 Neuronové sítě 16](#_Toc284690435)

[2.6 Rozhodovací stromy 16](#_Toc284690436)

[2.7 Hledání cest 16](#_Toc284690437)

[2.7.1 Prohledávání do šířky 17](#_Toc284690438)

[2.7.2 Pár slov k A\* 18](#_Toc284690439)

[2.8 Genetické algoritmy 18](#_Toc284690440)

[2.8.1 Evoluce v přírodě 18](#_Toc284690441)

[2.8.2 Hledání cesty pomocí genetického algoritmu 19](#_Toc284690442)

[2.8.3 Selekce párů 19](#_Toc284690443)

[2.8.4 Kombinace křížením 20](#_Toc284690444)

[2.8.5 Mutace 20](#_Toc284690445)

[2.9 Skriptování 21](#_Toc284690446)

[2.10 Case Based Reasoning 22](#_Toc284690447)

[3 Frameworky a enginy 22](#_Toc284690448)

[4 Implementace 23](#_Toc284690449)

[4.1 Volba prostředí 23](#_Toc284690450)

[5 Pozorování 23](#_Toc284690451)

[6 Závěr 23](#_Toc284690452)

[7 Citovaná literatura 25](#_Toc284690453)

[A Ukázky kódu 25](#_Toc284690454)

[B Obsah CD 27](#_Toc284690455)

Seznam obrázků

**Nenalezena položka seznamu obrázků.**

Seznam tabulek

**Nenalezena položka seznamu obrázků.**

Seznam grafů

Nenalezena položka seznamu obrázků.  
Seznam ukázek kódu

**Nenalezena položka seznamu obrázků.**

# Úvod

V první kapitole, v druhé kapitole…

## Historie

Vývoj umělé inteligence v počítačových hrách započal s hrami samotnými. Již první grafická hra Tic Tac Toe, piškvorky 3 x 3 pole, z roku 1952 měla v sobě zabudovanou umělou inteligenci protihráče. [1] Totéž platí i pro následovníky „Tenisu pro dva“ nyní známějšího pod jménem Pong, nebo např. pro hru Pac Man.

Přestože v počátcích video her byly hry značně jednoduché včetně umělé inteligence, mnohé by mohla překvapit propracovanost a složitost umělé inteligence ve hře Pac Man. Jedná se o jednoduchou hru odehrávající se na jedné obrazovce, kde máte za úkol sbírat po bludišti kolečka představující jídlo a zároveň se nenechat chytit jedním ze 4 duchů.

Duchové nejen mají různé barvy, jména a přezdívky, ale také odlišné chování, čehož si málokdo z hráčů na první pohled všimne. [2]

Složitější umělé inteligence bylo potřeba vyvíjet právě s rozvojem strategických her (Civilization, Heroes of Might and Magic), kde bylo a je nutností udělat poměrně složitou AI, aby hra byla vůbec hratelná.

V minulosti byl vývoj umělé inteligence v ústraní. Mnohem důležitější bylo vyvíjet grafickou stránku hry, která zároveň znatelně vytěžovala CPU počítačů a tedy ani výpočetní výkon nezbýval pro AI. Vývoj umělé inteligence často probíhá až v posledních pár měsících tvorby hry, a tedy proto nemůže být AI dokonalá.

V současné době, kdy už je hráč nasycen téměř dokonalou fotorealistickou grafikou, kdy výkon počítačů je mnohem dál, je již dostatek prostoru pro rozvoj AI v počítačových hrách. [3]

## Iluze inteligence

Umělá inteligence v počítačových hrách má s umělou inteligencí všeobecně mnoho společného. Metody jako rozhodovací stromy, konečné stavové automaty nebo třeba neuronové sítě můžete najít v obojím.

AI ve hrách se v určitých aspektech liší od umělé inteligence např. řízení leteckého provozu. Nemusí být co nejchytřejší, co nejlepší. Již dávno není problém udělat inteligenci bota ve FPS střílečce neomylného, který by vždy dokázal při spatření hráče ho zabít jednou ranou z pistole, head shotem. Podobně lze udělat nepřekonatelného střelce v basketbalu, který trefí koš přes celé hřiště. Oba dva případy by potencionální hráče brzy odradily.

Existují hráči, kteří až absurdní obtížnost ocení, ale většina hráčů upřednostní, když polovinu času vyhrávají a polovinu času prohrávají. Vůbec nemusí být špatným nápadem korigovat obtížnost hry dle počtu výher a proher a tím jejich poměr zachovávat přibližně roven jedna ku jedné.

Je důležité zachovat určitou reálnost chování NPC, hráč by neměl mít pocit toho, že soupeř podvádí. Při vývoji hry Empire Earth měli testeři hlásit jakékoliv podezřelé chování soupeře, měli zapsat kdykoliv si mysleli, že jejich soupeř podvádí. Přestože se vždy nejednalo o podvod, tak se nahlášené problémy vždy prodiskutovaly a případně se umělá inteligence poté upravila. [4]

Podvádění soupeřů někdy nemusí být na škodu, ale nesmí to soupeř poznat. Co to vlastně je to podvádění ve hrách? NPC z pohledu vývojaře podvádí, pokud využívá věcí ve hře, které nemůže využít lidský hráč. Např. když od počátku hry soupeř ví, kde má hráč základnu a vysílá tam bez předchozího průzkumu své vojáky. Případně ve strategických hrách není neobvyklým jevem, když soupeř vytvoří pro obranu své základny jednotky z ničeho, ze surovin, které nemohl během hry získat. Oba ze zmíněných podvodů mohou být přípustné a zlepšit celkovou hratelnost hry, pokud je hráč nezaznamená.

Z pohledu laika může být podvádění i to, když se soupeř chová až moc dokonale, nejedná realisticky. Sem patří již zmíněná dokonalá střelba v FPS. Realističnost je z pohledu hráče hodně důležitá, ale jsou případy, kdy by mu mohla přijít otravná. Adventura, kde jsou dvě cesty, na konci každé z nich je truhla, v jedné z nich je klíč od té druhé. Hráč může zvolit jakoukoli z těchto dvou cest jako první. Pokud by zvolil jako první tu, jež ho vede k zamčené truhlici, musel by se vracet, jít druhou cestou, získat klíč a opět se vracet k první truhle. Tento nedostatek lze elegantně vyřešit tak, že první truhla, kterou hráč otevře bude obsahovat klíč od té druhé truhly. [5]

K dalším zajímavým přístupům patří podívat se na řešení problému AI z jiného pohledu. Nesnažit se dělat NPC přehnaně komplexní a inteligentní, ale vložit tu inteligenci do světa kolem ní. Známým příkladem je hra The Sims, simulátor lidí, kteří mají své potřeby jako je hlad, jež je potřeba uspokojit. Pokud bychom se na to podívali z pohledu reálného světa, tak postava, když dostane hlad, tak najde v domě nejbližší jídlo (lednička, hotové jídlo na stole), dojde k němu a začne jíst. V The Sims je to dělané jinak. Lednička do určitého poloměru vysílá zprávu „Můžu uspokojit tvůj hlad“. Pokud se simík dostane do blízkosti ledničky, vyhodnotí své aktuální potřeby a pokud mu nějaký objekt v jeho blízkosti oznamuje, že je může uspokojit, přesune se k němu. [5]

Do prostředí lze zanést mnohé rozmanité informace. Např. bodliny v 2D plošinovce, které ubírají životy všemu, co se jich dotkne. Místo programování logiky do hráče, že když je v blízkosti bodlin, změň animaci hráče, uber životy, případně uskoč se tato logika přiřadí do logiky bodlin. Tedy bodliny vědí, že mají hráči ubrat životy, změnit mu animaci, odstrčit ho. Tento přístup nejen zjednodušuje komplexnost logiky hráče, ale zjednodušuje přidávání nových herních prvků do hry.

Co je důležité si zapamatovat pro tvorbu herní AI je to, že hra nemusí být dokonale realistická, férová, neporazitelná. Pořád se pohybujeme v herním, v zábavním průmyslu a tedy dobrá AI má za úkol hlavně pobavit a být výzvou pro hráče. Musí umět bavit naprostého nováčka i zkušeného hráče hrající online turnaje.

# AI algoritmy

## Stavový automat

Určitě jste zpozorovali v mnoha hrách následující chování jednotek. Cestovali jste krajinou a z dálky jste uviděli skupinku nepřátelských goblinů stojících v hloučku. Když jste se k nim přiblížili na dostatečně krátkou vzdálenost, goblini vás zpozorovali a začali útočit. Zaměřili jste útok na jednoho z nich. Když jste mu ubrali většinu jeho života, začal utíkat od vás. Druhého jste stihli zabít rychleji než se mohl dát na útěk. Souboj pokračoval, ale vy jste zeslábli a také jste se dali na útěk. Z dálky jste mohli pozorovat, jak se ke skupince goblinů vrací ten, jež utekl a jemuž se postupně doplňuje zdraví.

### Konečný stavový automat

Pro zachycení tohoto chování je vhodné využít konečné automaty. Konečný stavový automat (Finite State Machine, FSM) lze definovat jako uspořádanou pětici (Q, Ʃ, δ, q0, F), kde Q je konečná neprázdná množina stavů. Ʃ je konečná množina vstupních symbolů. δ je přechodová funkce, která lze definovat δ : Q × Ʃ → Q. Počáteční stav q0 ∈ Q. Množina koncových stavů F ⊆ Q.

Definice může být matoucí, především nemusí být jasné, co se zde míní konečnou množinou vstupních symbolů Ʃ. Vysvětlím to na příkladu AI goblinů. Množina všech stavů Q obsahuje stavy čekání, útok, útěk, smrt. Počátečním stavem q0 je čekání. Každý goblin se po svém vytvoření nachází v tomto stavu. Konečný stav je zde pouze jediný, a to je stav smrt. Symboly Ʃ si lze představit jako podmínky, které mohou vést ke změně z jednoho stavu do druhé. Zde např. podmínky životů ≤ 0, životů ≤ 20, vzdálenost hráče ≤ 50, vzdálenost hráče ≥ 75.

δ je dle definice kartézský součin množiny všech stavů a množiny symbolů. Jinak řečeno musíme definovat pro každý stav, co se stane při splnění každé z podmínek. Máme 4 stavy a 4 podmínky, což je dohromady 4 \* 4, 16 případů přechodu. U stavových automatů použitých pro tvorbu AI nás nebudou zajímat všechny možné případy. V jednotlivých stavech bude kontrolovat jen ty podmínky, které mění jeden stav na druhý. Např. ve stavu čekání nás budou zajímat pouze podmínky životů ≤ 0, životů ≤ 20, vzdálenost hráče ≤ 50. Pokud ve stavu čekání bude splněna podmínka životů ≤ 0, nový stav bude smrt, podmínka životů ≤ 20, nový stav bude útěk, podmínka vzdálenost hráče ≤ 50, nový stav bude útok. Poslední 4. podmínka vzdálenost hráče ≥ 75 v aktuálním stavu nemá vliv na změnu stavu, a nebude nás zajímat. Pokud bychom počítali ve hře pouze se zbraněmi s dosahem nižším než je 50, stačila by nám reakce pouze na podmínku vzdálenost hráče ≤ 50, jelikož by nemohlo dojít ke snížení životů goblinů.

Při navrhování stavových automatů je vhodné si nakreslit diagram obdobný tomu na následujícím obrázku, který znázorňuje kompletní stavový automat pro chování goblinů. K vytváření diagramu můžete využít některý z CASE nástrojů jako je např. Enterprise Architect, ale pro začátek si vystačíte s tužkou a papírem.

OBRAZEK OBRAZEK OBRAZEK IMAGE OBRÁZEK

### Implementace

Způsobů, jak implementovat FSM je mnoho, mezi nejjednodušší varianty patří využití if-then-else podmínek pro změnu stavu a switch/case pro výběr aktivní akce dle stavu automatu.

Tato varianta je jednoduše naprogramovatelná, snadno pochopitelná a i rychlá v provozu. Přestože všechny zmíněné výhody zní úžasně, pro pokročilé programátory bych doporučil některé ze složitějších řešení založených na polymorfismu.

Všímavější čtenář si všimne copy-paste vady. Ve všech stavech je zkopírována podmínka pro změnu stavu na DEATH. V případě složitějšího automatu s více stavy byste stejnou podmínku museli kopírovat i do nově přidávaných stavů, což dělá přidávání nových stavů komplikované. Mohli bystě při rozšíření množiny stavů tuto podmínku zapomenout zkopírovat. Říkáte si možná, že jde pouze o jednu podmínky, kterou si nezapomenete ohlídat, což je možná v tomto případě pravda, ale těch podmínek může být víc a složitější.

switch(state)

case DEATH :

case IDLE :

if(hitpoints < 0) state = DEATH;

elseif(hitpoints < 20) state = FLEE;

elseif(playerDistance < 50) state = ATTACK;

case ATTACK :

if(hitpoints < 0) state = DEATH;

elseif(hitpoints < 20) state = FLEE;

elseif(playerDistance > 70) state = IDLE;

case FLEE :

if(hitpoints < 0) state = DEATH;

elseif(playerDistance > 70) state = IDLE;

switch(state)

case DEATH : deathAction();

case IDLE : idleAction();

case ATTACK : attackAction();

case FLEE : fleeAction();

Pseudokód jednoduchého stavového automatu pro chování goblina.

Zkopírovaný kód má i další nevýhody. Např. podmínka pro smrt se může rozrůst o kontrolu, jestli se goblin neocitl v lávě. To se mohlo stát při zmateném útěku i při bezhlavém útoku na vás. Chyby se v takovém kódu špatně hledají.

Pokud byste chtěli mít ve hře více nepřátel s rozdílnými způsoby chování, např. hraničáře, který by měl rozdělen útok na stav útok z dálky lukem a na útok zblízka, tak by to vedlo opět ke kopírování kódu, protože např. stavy a přechody na IDLE a FLEE by zůstaly i pro hraničáře stejné. Takový kód se špatně udržuje a rozšiřuje. [6]

Obvyklé rozšíření stavového automatu přidává akce, které se mají vykonat pouze jednou při přechodu z jednoho stavu do druhého. Můžou být potřebovány jak při vstupu do nového stavu, i při výstupu z něj. Příkladem může být potřeba při smrti goblina změnit jeho animaci a přičíst zkušenosti hráči.

Lepší řešení využívá polymorfismu a návrhového vzoru State pattern [7], jehož název napovídá, že zde nalezne využití.

triggeredTransition = null;

for transition in currentState.getTransitions()

if(transition.isTriggered())

triggeredTransition = transition

break;

if (triggeredTransition)

targetState = triggeredTransition.getTargetState();

actions = currentState.getExitActions();

actions += triggeredTransition.getActions();

actions += targetState.getEntryActions();

currentState = targetState;

return actions;

else return currentState.getActions()

Pseudokód update funkce FSM automatu dle

Každý stav má určené akce, které se vyvolají při přechodu do něj (getEntryActions), při odchodu z něj (getExitActions) a akce v případě, že se stav nemění (getActions). Dále má definovanou množinu přechodů do jiných stavů (getTransitions), které mají metodu isTriggered(), jež určí, jestli přechod má nastat. Pokud ano, tak pomocí metody getTargetState se určí následující stav. Navíc stejně jako v ukázce můžou být definovány akce s pojené s přechodem (getActions).

V update funkci se nejdříve zkontrolují všechny přechody, jestli některý z nich nastal. Pokud ano, zavolají se výstupní akce současného stavu, akce přechodu a vstupní akce nového stavu. Jestliže nedojde ke změně stavu, zavolají se akce hlavní, jako např. idleAction.

### Vylepšení FSM

Druhé řešení je již o mnoho pružnější než první. Jednoduše lze přidávat další stavy a přechody mezi nimi, starat se o různá chování jednotlivých druhů monster. Rozšíření na hraničáře se dvěma druhy útoku, či na bereserkera, který nezná strach a nemá akci FLEE, je již jednoduché.

Problematická může být metoda isTriggered(), která je u přechodů. Někdy v ní může být pouze jednoduchá podmínka, jindy složená z mnoha podmínek provázaných spojkami. Můžou vzniknout přechody s podmínkami, které se od sebe moc neliší. Např. jednou se kontroluje počet životů ≤ 0, u nelétavých nepřátel se navíc kontroluje, jestli nespadli do lávy. Nabízené řešení vede ke kopírování první z podmínek do obou stavů. Tento problém lze napravit pomocí návrhového vzoru Composite [7]. Konkrétní řešení můžete najít v [8].

Problém s kopírováním podmínky smrti přetrvává i do druhé verze, i když zde to již není tak špatné jako ve switch verzi FSM. Zůstává zde nutnost registrovat přechod kontrolující zdraví goblina do všech jeho stavů. Jedno nabízené řešení je implementovat globální stavy, které se kontrolují každý frame nezávisle na stavu lokálním. [9] Druhou možností je implementovat hierarchické stavové automaty, kde každá jednotka AI může být ve více stavech zároveň, je implementováno více stavových automatů, které se liší svou prioritou. Kontrola smrti by byla využita v FSM s vyšší prioritou než FSM obsahující stavy IDLE, ATTACK, FLEE. Více informací o hierarchických FSM lze získat z [8].

## Goal – driven

## Fuzzy logika

## Umělý život

Artificial Life, A life, umělý život. Další ze způsobů, který lze využít pro tvorbu AI, ale i jako základní princip hratelnosti. Jedná se o simulaci života, kde hráč přímo, či nepřímo ovládá jednu, nebo několik životních forem(lidí, zvířat, příšer), které bez zásahu hráče dokážou ve hře samostatně myslet, pohybovat se, žít.

Prapůvod umělého života je v Conveyově simulaci Game of Life. Jde o svět tvořený 2D mřížkou. Každá buňka mřížky představuje buď živou, nebo mrtvou buňku. Život se v tomto světě rozvíjí v tazích, kolech, kde několik pravidel určuje, jestli v následujícím tahu vznikne nová buňka na místě, kde žádná není, nebo zanikne tam, kde je, nebo jen stávající buňka bude pokračovat ve svém životě. Pravidla pro život v dalším kole jsou celkem čtyři a odvíjí se pouze podle počtu živých buněk v okolí v kole současném.

Přestože se jedná o jednoduchou simulaci, bylo v tomto světě o 4 pravidlech objeveno několik stovek vzorů skupin buněk se specifickým chováním rozdělených do několika kategorií. Patří jsem např. glider, skupina buněk, která vlivem zanikání a vzniku nových buněk působí dojmem, že se celá skupina pohybuje jedním směrem. [6] Game of life není stále zcela prozkoumána, každým rokem vědci z celého světa objevují nové vzory.

Simulaci života můžeme vidět v mnohých hrách, které jdou rozdělit do několika skupin. [7]

### Mazlíčci

Žánr, kde se staráte o své virtuální zvířátko, reálné, či smyšlené. Musíte ho krmit, cvičit, za prohřešky trestat. Zvíře je schopné se od hráče učit. Mazlíčci jsou většinou roztomilí a svými emocemi, chováním dávají hráči najevo, co s nimi má dělat.

Oproti jiným A-Life hrám se zvířátka často nemůžou rozmnožovat, umírat. Neznamená to ale, že nemůžete v takových hrách dělat špatná rozhodnutí. Pokud se chováte špatně ke svěřenému zvířeti, může od vás utéct.

Mezi známé zástupce tohoto žánru patří např. Tamagotchi, Nintendogs, nebo Neopets.

### Společenské simulace

Kdo by neznal hru The Sims, která odstartovala tento žánr. Úkolem je starat se o jednu, čí více osob, které mají několik základních potřeb (hygiena, hlad, zábava, potřeba wc, kamarádi apod.), jež je třeba udržovat. Na rozdíl od starání se o mazlíčky v těchto hrách jsou důležité sociální vazby s ostatními osobami v rodině a v sousedství, které je třeba udržovat a rozvíjet pro lepší pokrok ve hře.

### Hry na Boha

Populous, Black and White, Dungeon Keeper, zástupci tohoto žánru. V Dungeon Keeperu se impové sami od sebe se starají o zpevňování stěn, těžbu drahokamů, či zabírají nová území. Vy jako „Bůh“ je můžete trestat, obětovávat, dávat jim nepřímo rozkazy, nebo je jen sledovat. I ostatní příšerky žijí svým životem, chodí spát, jíst, cvičit se a nechávají si od vás zaplatit za obranu vašeho království. Vy je můžete vzít a přesunout tam, kde jsou zrovna potřeba. Např. poslat je objevovat nová kouzla, nebo zdokonalovat se v útoku.

### Evoluční hry

Hry, v nichž se staráte o populace několika generací, které se množí, vylepšují, přenášejí nové lepší vlastnosti do dalších generací. V těchto hrách můžete nechat organismy se vyvíjet vlastní cestou, nebo můžete do vývoje zasahovat změnou prostředí, či přidáním nově navržených vlastních forem života.

Populárními hrami jsou např. Spore, či Evolution : The Game of Intelligent Life .

### A-Life a tahové strategie

Původní experiment Game of Life se odehrává v jednotlivých krocích, ale stěží to lze označit za hru, jde jen o matematický experiment.

Nepodařilo se mi nalézt žádnou tahovou strategii spojovanou s metodou umělého života. Důvodem může být, že ve všech zmíněných hrách bylo podstatné, aby se bez hráčova zásahu vše hezky hýbalo, žilo vlastním životem a aby to bylo zábavné jen sledovat. Sledovat v reálném čase.

A to může být hlavní překážkou pro zdárné využití této myšlenky v tahových strategiích, kde by čekání na odkliknutí dalšího tahu narušovalo plynulý život.

## Neuronové sítě

## Rozhodovací stromy

## Hledání cest

Jedná se o speciální problém, který spadá do problematiky umělé inteligence. Hráč chce vyslat jednotky do soupeřovy základny. Neurčuje jednotkám jednotlivé kroky, určí jen požadovanou pozici a zbytek se udělá automaticky.

Úkol je zřejmý, nalézt nejoptimálnější a zároveň věrohodnou cestu mezi dvěma místy na mapě. Nejoptimálnější často znamená nejkratší cestu, ale nemusí tomu být tak. Např. pokud v nejkratší cestě bude minové pole, tak by se mu měly jednotky umět vyhnout. Minimálně jednotky ovládané počítačem.

Zachování věrohodnosti chování a přirozenosti je také důležité. Nemusí působit dobře, pokud jednotky se pohybují s naprostou přesností k cíli a obcházejí horu, kterou ještě hráč prozatím neobjevil, a tedy jednotky by o ní neměly vědět.

Dalším problémem, který je třeba v této kategorii řešit, je přesouvání více jednotek naráz. Působilo by nepřirozeně, kdyby všechny jednotky se snažily jít nejkratší možnou cestou, a tedy šly jeden za druhým jako vláček.

S hledáním cest je nemálo problémů, ale pro řešení základní úlohy, hledání nejkratších cest, bylo již vymyšleno několik funkčních algoritmů.

Hledání (nejkratších) cest je jedním ze stěžejních problémů teorie grafů. V tomto případě grafem se nemíní grafické zobrazení tabulkových hodnot na osách x a y, ale graf je množina uzlů a hran mezi nimi. Příkladem grafu může být bitevní pole složené ze šestiúhelníků z tahové strategie HOMAM 3, kde co jeden šestiúhelník, to jeden uzel grafu. Každý uzel má až šest hran, které znázorňují možnost pohybu na sousední šestiúhelníky.

Mezi algoritmy řešící tuto úlohu patří prohledávání do šířky, prohledávání do hloubky, heuristické prohledávání(sem patří ve hrách velice populární algoritmus A\*), Dijkstrův algoritmus, Floyd-Warshallův algoritmus, Bellman-Fordùv algoritmus.

### Prohledávání do šířky

Prohledávání do šířky je poměrně jednoduchý algoritmus snadno použitelný v počítačových hrách. Průběh vyhledávání připomíná vlnu rozšiřující se na hladině vody, když na její povrch dopadne kapka vody.

Základní princip algoritmu vysvětlím na příkladě čtvercové 2D mapy, kde je označen jeden čtverec jako začátek(např. by to mohla být aktuální pozice jednotky), jeden čtverec jako cíl(např. místo, kam hráč klikl myší a chce poslat jednotku) a některé ze čtverců jsou označené jako neprůchozí zdi. V naší imaginární hře s 2D mapou se můžou jednotky pohybovat pouze do 4 směrů, ne diagonálně.

Vyhledávání probíhá v několika krocích. Prvně se zkontroluje, jestli se začátek neshoduje s cílem, pokud ano, „cesta“ nalezena. Pokud ne, podívám se postupně na jednotlivé sousedy startovního pole, jestli ony nejsou cílem. Až zkontroluji všechny a nenacházím-li mezi nimi cíl, tak zkontroluji sousedy sousedů startovního pole. Když ani poté cíl nenacházím pokračuji stejným způsobem, otestuji všechny čtverce vzdálené na tři kroky od startu. Tento postup se opakuje dokud se nenarazí na cíl, nebo se nezjistí, že cíl není dosažitelný(např. je na ostrově, na který nevede most).

Jak se pozná, že k cíli se ze startu nelze dostat? Při procházení mapy „vlnou“ od startu si označuji čtverce, na kterých už jsem byl a hledal cíl, pokud se dostanu do fáze, že všechna pole dostupná ze startu jsou označená a ani jedno z nich nebylo cíl, značí to, že cíl je nedostupný.

Tento algoritmus s jistotou najde nejkratší cestu do cíle, pokud existuje a umí i zjistit, že taková cesta neexistuje. Jeho nevýhodou je, že hledá cíl ve všech směrech a prohledává velké množství prostoru, což se projeví především ve velkých prázdných plochách bez zdí. Pokud start a cíl budou vzdálené deset polí, tak než se cíl nalezne, projdou se všechna pole do vzdálenosti devět od startu. Druhý problém je ještě výraznější, pokud na mapě velké tisíc krát tisíc polí neexistuje cesta mezi startem a cílem, tak se prohledá až milión polí než se zjistí, že cesta neexistuje.

První problém lze částečně vyřešit vysláním vln zároveň z cíle a startu proti sobě. U příkladu polí vzdálených deset polí od sebe můžeme porovnat obsahy dvou kruhů o poloměru pět a jednoho o poloměru deset. Konstantu pí můžeme vynechat a máme to 2 \* 5 \* 5 = 50 ku 10 \* 10 = 100, tedy u dvou vln se v tomto případě prohledá polovina polí oproti vlně jedné.

Zamezit prohledávání miliónu polí lze jednoduchou podmínkou. Pokud se nenalezne cíl do vzdálenosti 30, 50, … vyhledávání se ukončí jako neúspěšné. Toto vylepšení není ideální, protože teď algoritmus nezaručuje, že cestu nalezne, přestože existuje. Pokud zvolíte limit 50, co když by byl cíl ve vzdálenosti 51? Hranici, kdy by mělo vyhledávání skončit, není jednoduché najít.

### Pár slov k A\*

A\* patří k nejpoužívanějším vyhledávacím algoritmům v počítačových hrách. S algoritmem prohledávání do šířky má více společného než se na první pohled může zdát. Prohledávání do šířky je vlastně speciálním případem A\*.

U prohledávání do šířky jsme postupně kontrolovali nejdříve všechny čtverce vzdálené jedna od startu, poté 2, 3, 4, atd. V jakém pořadí se budou kontrolovat čtverce, bylo dáno pouze jejich vzdáleností od startu. U algoritmu A\* k tomu přibude druhý parametr, a to předpokládaná vzdálenost k cíli. Čtverce se prohledávají v pořadí dané součtem vzdálenosti od počátku a předpokládané vzdálenosti do cíle. Čtverec, který má tento součet menší než jiný bude testován před ním.

Pod předpokládanou vzdáleností si lze představit odhad vzdálenosti se zanedbáním terénů, zdí mezi kontrolovaným čtvercem a cílem. Např. u příkladu se čtvercovou mapou dobře funguje jako odhad tzv. Manhattonská metoda. Odhad touto metodou se spočítá jako součet absolutních hodnot rozdílu x-ové a y-ové souřadnice zkoumaného čtverce a cíle. Příklad: čtverce se souřadnicemi [3, 4] a [0, 7] mají vzdálenost |3-0| + |4-7| = 6. Mezi další metody patří např. euklidovská vzdálenost.

Algoritmus lze snadno vylepšit, aby podporoval různé typy povrchů (cesta , bažina), aby znemožňoval dokonalou navigaci v neprozkoumaném prostředí apod.

## Genetické algoritmy

### Evoluce v přírodě

Genetické algoritmy patří k algoritmům inspirovaných přírodou, zde konkrétně evolucí. V přírodě přežijí pouze nejsilnější živočichové, obecněji živočichové lépe přizpůsobení prostředí, v kterém žijí. Je-li myš rychlejší než ostatní, má o něco větší šanci přežít, spářit se a přenést své geny na potomstvo. Geny obou myších rodičů se zkříží a vznikne nový potomek, který zdědí vlastnosti úspěšných rodičů.

Samotné křížení genů vybraných lepších jedinců není jediným důležitým prvkem v evoluci. Je dobře známo, že nejdříve žili živočichové pouze ve vodě a až později se přesunuli na souš a byli schopni dýchat atmosférický vzduch. Kdyby bylo v přírodě pouze křížení, tak by kombinací genů zodpovědných za tvorbu žáber nikdy nevznikly geny pro vznik plic. K tomu je zapotřebí mutace, která vznáší do DNA potomka geny, jež neměl ani jeden z jeho rodičů.

Často vlivem mutace vzniknou jedinci, kteří nejsou schopni v přírodě dlouho přežít. Příkladem mohou být albíni, tedy živočichové, kteří místo maskující barvy srsti mají srst bílou. Pro člověka to působí jako nádhera, ale v přírodě je to spíše na obtíž. Albín se mnohem hůře schovává před predátory.

Avšak čas od času mutace dokáže vytvořit nového jedince schopnějšího přežít v daném prostředí, a tedy tím přenášet nově vzniklý gen do dalších generací.

Tímto končím krátké připomenutí biologie střední školy. Genetické algoritmy mají mnohé využití, nejsou úzce spjaty s umělou inteligencí.

Pomáhají nalézt řešení daného problému, ale nezaručují nalezení nejlepšího řešení, ani nalezení nějakého řešení.

### Hledání cesty pomocí genetického algoritmu

Zkusme vyřešit problém hledání cesty mezi dvěma místy z předchozí kapitoly pomocí genetického algoritmu. Budeme hledat cestu v 2D mřížce mezi dvěma čtverci skrz bludiště. Jsou povoleny 4 směry pohybu, tedy není povolen diagonální pohyb.

Hledané řešení genetickým algoritmem bude posloupnost příkazů nahoru U, vpravo R, dolů D, vlevo L, která dovede hráče ze startu do cíle. Hledaná posloupnost bude vyšlechtěným potomkem vzniklým křížením a mutací jiných posloupností příkazů, které sice nevedly k cíli, ale postupně se mu přibližovaly. Zbývá určit, podle čeho vybírat úspěšné a neúspěšné jedince. Kterým dát možnost se křížit a přiblížit se hledanému řešení, a které nekompromisně zahodit. Pro tento specifický problém bude úspěšnost organismu dána vzdáleností od cíle, kam by se hráč dostal, kdyby se pohyboval dle posloupnosti instrukcí. Čím menší vzdálenost, tím lepší. Pokud nulová, nalezli jsme řešení. Zde nutno podotknout, že nalezená cesta nemusí být nejkratší, tímto způsobem se může nalézt nějaká cesta mezi startem a cílem.

Algoritmus bude fungovat následovně. Na začátku si určíme, s jak velkým potomstvem budeme pracovat. Mějme např. populaci o 100 kusech. Každý kus je na začátku inicializován náhodnou posloupností URDL a je mu vypočítána hodnota fitness (jak moc je úspěšný při hledání cíle). Z těchto 100 kusů se vybírají dvojice, které se budou křížit a mutovat, a tak vytvářet novou generaci potomků, jimž se určí hodnota fitness. Následně se kroky selekce dvojic, křížení a mutace, vznik nových potomků opakují v jednotlivých generacích dokud nevznikne potomek s ideální hodnotou fitness, tedy ten, který řeší úlohu.

### Selekce párů

Je více způsobů, jak vybírat vhodné dvojice pro páření. První, co by asi každého napadlo, kombinovat pouze ty nejlepší. Což na první pohled může vypadat jako skvělý nápad, ale trpí nedostatkem, že může nalézat pouze lokálně nejlepší řešení, ne globálně. V našem případě to znamená, že algoritmus ztratí ve slepé uličce, která končí blízko cíle, ale před cílem je zeď. Nebere to vůbec v úvahu možnost, že hledaná cesta k cíli směřuje od startu nejdříve směrem od cíle. K tomuto typu selekce patří elitářství (elitism), kde je zaručeno, že n nejlepších kusů bude zachováno do další generace, či selekce setrvalého stavu (steady state selection), kdy se do další generace zanechá např. 4/5 populace a zbylá pětina se vytvoří křížením.

Druhým způsobem je výběr proporcionálně k úspěšnosti jedince. Čím úspěšnější jedinec (větší fitness), tím má větší šanci, že bude vybrán ke křížení se. K této metodě patří selekce ruletou (roulette wheel selection). Kolo rulety je rozděleno na n výřezů dle velikosti populace, v našem případě 100. Velikost výřezu je dána velikostí fitness daného potomka. Pokud má jeden potomek fitness 2 a druhý 6, tak ten se 6 má třikrát větší výřez na kole než ten s fitness 2 a tím i třikrát větší šanci, že bude vybrán pro křížení se. Nevýhodou tohoto řešení je, že nemáte jistotu výběru nejlepších jedinců. Může se s malou pravděpodobností stát, že řešení blízké cíli bude zahozeno. Dobré je tento přístup kombinovat s předchozím, vybrat např. 5 nejlepších, kteří mají jistotu přežití do další generace populace a se zbytkem provést selekci ruletou.

Dalším typem selekce je selekce turnajem (tournament selection), která poměrně úspěšně eliminuje nevýhody předchozích dvou. Při výběru potomka ke křížení se náhodně vybere ze všech potomků n jedinců a z nich se vyberou dva nejlepší(s největším fitness), kteří se zkříží.

### Kombinace křížením

Ke křížení nedochází vždy po výběru dvou jedinců. Jestli ke křížení dojde určuje programátorem zvolená hodnota. Příkladem může být 0,7. Ze 70% je šance, že dojde ke křížení, ze 30% dojde pouze k zachování rodičů ze současné generace do generace následující.

I zde se může vymyslet mnoho způsobů, jak křížit potomstvo. Křížení i mutace se odvíjí od toho, jak máme zakódované jednotlivé jedince. U hledání cesty jsem zvolil výčet 4 prvků URDL, které mohou být reprezentovány v programu celými čísly.

Mějme dvě zkrácené trasy RDDLRULRU a LUDDLRLUR. Na těchto trasách ukážu několik možností jejich křížení.

Křížení jedním bodem (single-point crossover). Zvolí se náhodně pozice v řešení, od té pozice se řešení dvou křížených jedinců roztrhnou a konce řešení se zamění. Např. pro pozici 5 v našem příkladě :

RDDLRULRU a LUDDLRLUR

X

RDDLLRLUR a LUDDRULRU

Křížení dvěma body (two-point crossover) je obdobné. Zvolí se v řetězci dvě pozice, místo mezi nimi se vystřihne a zamění se mezi sebou. Např. pro náhodně zvolené pozice 2 a 5.

RDDLRULRU a LUDDLRLUR

X

RUDDLULRU a LDDLRRLUR

Nic nebrání v tom si zvolit bodů více (multi-point crossover), a tak zaměňovat libovolné kousky řetězců.

RDDLRULRU a LUDDLRLUR

X

RDDDLULUU a LUDLRRLRR

### Mutace

Stejně jako v přírodě i v genetických algoritmech nedochází k mutacím při každém křížení. Pravděpodobnost mutace se nastavuje obdobně jako pravděpodobnost křížení, ale zde na mnohem menší hodnotu. Pravděpodobnost může být například v jednotkách promile, tedy např. 0,005.

Nejjednodušším typem mutace v našem případě může být záměna jednoho či více příkazů za jiné. RDDLRULRU > ULDLRULRU.

Jinou motací může být prohození jednoho příkazu s jiným, skupiny příkazů s jinou. RDDLRULRU > RUDLRDLRU. Můžeme i otočit pořadí několika prvků. > RDDURLLRU.

V podstatě způsob mutace, či křížení je ponechán pouze naší představivosti, určitě po zamyšlení by vás napadly další metody pro mutaci. Není asi k udivení, že v [8] Mat Buckland tvrdí, že být dobrý v genetickém programování není jen věda, ale také umění. Mimo obdobného příkladu hledání cesty můžete v jeho knize nalézt příklad využití genetického algoritmu pro řešení problému obchodního cestujícího, či pro ovládání lunárního vozítka ze známé hry Moon Lander.

## Skriptování

V případě, že už jste v minulosti tvořili počítačovou hru, narazili jste pravděpodobně na problém podobný následujícímu. Představte si, že tvoříte klasickou strategickou hru jako je Heroes of Might and Magic. Zjistíte, že jednotka rytíře je příliš slabá, zvětšíte jí tedy sílu. Poté musíte hru znovu zkompilovat a slinkovat a znovu spustit. Objevíte v textech překlep, gramatickou chybu, opravíte jí, opět kompilujete, linkujete a spouštíte hru. Všechno trvá nějakou dobu a zdržujete vývoj. Zvlášť např. u vyvažování jednotek, kdy budete upravovat jejich atributy poměrně často než dosáhnete kýženého výsledku.

Herní data zakomponovaná ve zdrojových kódech mají i jiné nevýhody. Scénárista a pisatel textů, či designér hry nepotřebuje a někdy i nesmí mít přístup ke zdrojovým kódům. Nemusí to být vůbec technicky znalí lidé, které by kód mohl rozptylovat a mohli by do něj nepozorností zanést chyby.

Řešením je herní data vytáhnout ze zdrojových souborů do externích textových souborů, skriptů. Skript může sloužit k jednoduchému ukládání páru vlastnost – hodnota, ale lze je využít pro mnohem komplikovanější účely jako je vytváření nových herních událostí, či dokonce lze pomocí skriptů řídit chování programu.

<monster>

<name>Ogre</name>

<hitpoints>50</hitpoints>

<firststrike />

</monster>

Jednoduchý XML skript

Při použití skriptu stačí restartovat hru a je změna aktivní v ní. Skripty můžete načítat nejen na začátku spuštění programu se hrou, ale i např. průběžně během hraní. Pak již nemusíte restartovat hru, změna se projeví okamžitě a můžete vyvíjet „on the fly“. Nevýhoda tohoto přístupu je zřejmá, opakované načítání z disku zbytečně zatěžuje procesor.

Jazyk, ve kterém budou psány skripty do vaší hry, si sami určíte. Může být jednoduchý stejně jako ukázkový kus skriptu definující obra. Jiný druh skriptu můžete využít pro definici úkolů ve hře. Takový skript může být posloupnost kroků, které musí hráč udělat, aby získal svou odměnu.

ODMENAGP=500;

ODMENAXP=1500;

DOJIT\_NA(80,40); // Strom poznani

DOJIT\_NA(100,42); // Strom vsevedeni

PRISERA = VYTVORIT\_MONSTRUM(DRAK, 90, 41);

ZABIT(PRISERA); // Zabit draka

Příklad skriptu definující úkol ve hře.

Definujete si vlastní jazyk DSL (Domain specific languge), jeho slova (DOJIT\_NA, VYTVORIT\_MONSTRUM) a gramatiku (příklad jeden příkaz na řádku, příkaz zakončen středníkem, cokoliv za středníkem je komentář). Takový jazyk je dobře srozumitelný ne technicky vzdělaným lidem, a tak se může zapojit do projektu mnohem více lidí, kteří můžou vidět problém zcela z jiného úhlu a tím přispět ke vzniku dobré hry.

Takový jazyk můžete dále rozšiřovat. Přidávat příkazy větvení, smyček, či dát možnost definovat vlastní funkce. Větší složitost dává možnost vytvářet více zajímavých a rozličných úkolů. Nevýhodou je, že již ne každý může vytvářet dané skripty. Pokud posadí neprogramátora k takto rozšířeným možnostem skriptování, je pravděpodobné, že většinu jeho nabízených možností ani nikdy nevyužije. Lépe je příkazy a konstrukce postupně přidávat, aby si na ně všichni zvyknuli.

Jazyky DSL se můžou buď interpretovat, nebo kompilovat. Při interpretaci se v programu čte textový soubor řádek po řádku a rozeznávají se textové příkazy. Tento způsob je rychlejší při vývoji. Stačí uložit upravený textový soubor a je vše hotovo. Dalším plusem je, že hry využívající interpretové skripty jsou snadno využitelné modaři, fanoušky, kteří si chtějí hru upravit dle vlastních potřeb a tím prodloužit životnost hry. Může to prodloužit prodejnost hry za její původní plnou cenu až o 8 týdnů. [9] Mínusem je větší objem dat a rychlost interpretace, která je oproti využití předkompilovaných skriptů pomalejší.

Předkompilování urychlí běh skriptů ve hře a skripty poté zaberou méně místa na disku, což je stále aktuální problém při vývoji her na mobilní zařízení. Nevýhodou je pomalejší vývoj, kdy při každé změně skriptu se musí překompilovat a také to zhorší přístup modařům. Zamezení přístupu hráčů ke změně dat může být občas i záměrné, např. u her, které jsou určeny pro online hraní a turnaje. Zamezí se tak možnosti podvádění hráčem.

Před vytvářením vlastního skriptovacího jazyka je třeba se zamyslet, jestli by nešel využít nějaký již existující volně dostupný jazyk. Takový jazyk bude odzkoušen širokou základnou uživatelů, bude bez chyb a pravděpodobně bude mít stránku, kde budete moci napsat své připomínky na vylepšení jazyka. Nevýhodou využití cizího skriptovacího jazyka je, že časem může nastat situace, kdy budete potřebovat nějakou vlastnost, jež jazyk nepodporuje a nepůjde si ji dodělat. Pak můžete přednést svoji prosbu, ale při vývoji hry nemůžete spoléhat, že někdo třetí vaši prosbu vyslyší a vyplní.

Mezi nejznámější skriptovací jazyk používaný v počítačových hrách určitě patří jazyk Lua, který je populární také svou syntaxí podobnou jazykům z rodiny C. Dalšími možnostmi jsou např. Python, či Scheme. Stručný přehled skriptovacích jazyků naleznete v [9], základy jazyku Lua můžete pochytit z knihy [10].

## Case Based Reasoning

# Frameworky a enginy

Herních frameworků a enginů existuje spousta. Jsou poměrně známé obecné frameworky jako je Unity, nebo frameworky zaměřené například na grafiku Ogre3D, fyziku PhysX apod. Nabízí se otázka, jestli existuje nějaký framework určený pro tvorbu AI do her, který by byl obecně použitelný, nebo alespoň použitelný pro tahové strategie.

Existují open source projekty implementující jednu ze zmíněných metod z předchozí kapitoly, např. pro fuzzy logiku, neuronové sítě, či rozhodovací stromy, ale tyto projekty nebyly zamýšlené pro využití v herním průmyslu, mají sloužit hlavně k vědeckým účelům, data miningu a statistice. Seznam takových projektů můžete nalézt na [9], kde jsou jejich stručné popisy, licence a odkazy na jejich domovské stránky. Pro Case Base Reasoning mohu zmínit jColibri[10], který je napsán v Javě a není tedy moc vhodný pro využití ve hrách.

Jako obecně použitelné řešení se může jevit SOAR[11]. SOAR je architektura pro vývoj obecných systémů, které vykazují inteligentní chování. Je ve vývoji již od roku 1983 a nyní je ve verzi 9. Na stránce projektu jsou k dispozici ke stažení jednoduché hry využívající tuto architekturu. Bohužel se jeví pouze jako ukázka toho, k čemu lze SOAR přizpůsobit, ale není zde reference na nějakou konkrétní nejlépe komerční hru využívající tuto architekturu. A pokud taková hra nevznikla za téměř třicetileté trvání SOARu, tak asi není k tomuto účelu využitelná.

Mezi ukázkami je využití této architektury pro řešení hry hanojských věží, pro logickou hádanku obdobné převozníkovi, jedné loďce, s dvěma místy, kozou, vlkem a zelím. Najdeme zde i akční 2D hru viděnou z ptačí perspektivy, tanky, které po sobě střílí. Pokud si spustíte ukázku TankSOAR, po chvíli si můžete všimnout jedné věci. Souboje tanků nejsou vůbec zábavné. Na mnoho kol se dokážou zaseknout na jednom místě, protože ani jeden nechce vyjet ze svého výhodného úkrytu. Taková strategie je inteligentní, je pochopitelné, že se tank nesnaží vyjet před hlaveň druhého, ale sledovat minutu vyčkávající tanky na jednom místě vás unudí.

Jinou možností je využít kompletního enginu, který mimo herního, grafického enginu obsahuje i umělou inteligence. Sem patří např. Cipher engine[12], jenž pro své rozmanité využití (FPS, RPG, závodní hry) obsahuje pouze základní jednoduchý framework pro budování AI a chování. Visual 3D Game engine[13] obsahuje mimo dalšího editor stromů chování.

V případě volby konkrétního žánru hry jako je FPS nebo RTS je možné využít kompletní řešení pro daný typ hry, které má v sobě zabudované hotové řešení pro umělou inteligenci. V základu jsou, bohužel, všechny FPS sobě podobné jako vejce vejci a podobně je tomu i u RTS. Pro tvorbu FPS lze zmínit od roku 2005 open source Quake engine, který lze stahovat z [14]. Obdobně pro real-time strategie existuje OpenRTS engine [15].

Jako poslední bych zmínil frameworky vyvíjené pro použití ve hrách, které jsou ale použitelné pouze k nějakému konkrétnímu účelu. Patří sem Recast, jež umožňuje implementaci pathfindingu ve 3D světě na základě jeho geometrie. Dále jsou zde mohutnější systémy, které umožňují simulace lidí a dopravy. Jmenovitě AI implant a Alive!.

Žádné z nalezených řešení mi nepřišlo vhodné pro implementaci do tahové strategie. Buď se jedná o příliš specifická řešení pro konkrétní žánry a problémy, nebo o řešení nevhodné pro vývoj her. Nenašel jsem žádnou strategii úspěšně využívající nějaký obecný framework, a tedy je stále potřeba si napsat vlastní AI využívající některou z metod předchozí kapitoly.

# Implementace

## Volba prostředí

# Pozorování

# Závěr

# Citovaná literatura

1. **Lackore, Jason.** Survey on the use of artificial intelligence in video games. *Jason's Website.* [Online] [Cited: leden 20, 2011.] http://jlackore.com/Documents/Survey%20on%20the%20Use%20of%20Artificial%20Intelligence%20in%20Video%20Games.pdf.

2. **Birch, Chad.** Understanding Pac-Man Ghost Behavior. *Game Internals.* [Online] prosinec 2, 2010. [Cited: leden 20, 2010.] http://gameinternals.com/post/2072558330/understanding-pac-man-ghost-behavior.

3. **Tozour, Paul.** The Evolution of Game AI. [book auth.] Steve Rabin. *AI Game Wisdom.*

4. **Scott, Bob.** The Illusion of Intelligence. [book auth.] Steve Rabin. *AI Game Wisdom.*

5. **Kirby, Neil.** Solving the right problem. [book auth.] Steve Rabin. *AI Game Wisdom.*

6. Glider. *Conway Life.* [Online] červenec 23, 2010. [Cited: leden 20, 2011.] http://www.conwaylife.com/wiki/index.php?title=Glider.

7. Life Simulation Game. *Wikipedia.* [Online] leden 17, 2011. [Cited: leden 20, 2011.] http://en.wikipedia.org/wiki/Life\_simulation\_game.

8. **Buckland, Mat.** *AI Techniques for Game Programming.* Ohio : Premier Press, 2002.

9. **Millington, Ian.** *Artificial Inteligence For Games.* s.l. : Morgen Kaufmann, 2006.

10. **Buckland, Mat.** *Programing AI by Example.* s.l. : Wordware Publishing, 2005.

11. Machine learning software. *DMOZ.org.* [Online] [Cited: únor 1, 2011.] http://www.dmoz.org/Computers/Artificial\_Intelligence/Machine\_Learning/Software/.

12. jCOLIBRI CBR Framework. *Group for Artificial Intelligence Applications.* [Online] [Cited: únor 1., 2011.] http://gaia.fdi.ucm.es/grupo/projects/jcolibri/jcolibri2/index.html.

13. SOAR. *SOAR.* [Online] [Cited: únor 1., 2011.] http://sitemaker.umich.edu/soar/home.

14. *Cipher Engine.* [Online] [Cited: únor 1., 2011.] http://www.cipherengine.com/.

15. *Visual 3D Game Engine.* [Online] [Cited: únor 1., 2011.] http://www.visual3d.net/.

16. Quake 3 1.32 Source Code. *File Shack.* [Online] [Cited: únor 1., 2011.] http://www.fileshack.com/file.x?fid=7547.

17. OpenRTS. *Lible Game Wiki.* [Online] [Cited: únor 1., 2011.] http://libregamewiki.org/OpenRTS.

18. **Belis, Mary.** Computer and Video Game History. *About.com Inventors.* [Online] [Cited: leden 20, 2011.] http://inventors.about.com/library/inventors/blcomputer\_videogames.htm.

1. Ukázky kódu
2. Obsah CD

|  |  |
| --- | --- |
| ┐ ├───results  │ ├──cell  │ └──x86  ├───src  │ ├──cell rt  │ ├──x86 client  │ └──x86 rt  └───thesis | Data files with results measured on Cell  Data files with results measured on x86  IBM Cell Ray Tracer source  x86 to Cell Client source  x86 Ray Tracer source  PDF and Microsoft Word versions of this text |