Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Lukáš Bajer

Pohyb v projektu Enti

Katedra software a výuky informatiky

Vedoucí bakalářské práce: Mgr. Cyril Brom

Studijní program: Informatika, Programování

2007

Velký dík za vedení práce, podnětné připomínky i trpělivost patří Mgr. Cyrilu Bromovi. Za pomoc při řešení počátečních problémů v orientaci ve vnitřnostech projektu ENTI bych rád poděkoval i RNDr. Ondřeji Bojarovi a Mgr. Milanu Hladíkovi. Třetí poděkování pak směřuje knihovnickému personálu Městské knihovny v Novém Boru, díky jehož ochotě vznikla nemalá část této práce i v červenci 2007.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 10. srpna 2007 Lukáš Bajer

Obsah

Název práce: Pohyb v projektu ENTI
Autor: Lukáš Bajer
Katedra (ústav): Katedra software a výuky informatiky
Vedoucí bakalářské práce: Mgr. Cyril Brom
e-mail vedoucího: brom@ksvi.mff.cuni.cz

Abstrakt: Projekt ENTI je simulátor prostředí, které je podobné lidskému světu. Žijí v něm autonomní agenti nazývaní enti, kteří se o svět starají. K naplňování svých úkolů a životních potřeb potřebují často hledat po svém světě cestu. Tato práce je zaměřena na skripty, které toto hledání a následné procházení cest řídí. Pohyb entů mezi místnostmi vylepšuje hierarchickou verzí algoritmu A*, čímž snižuje nároky na procesor při hledání delších cest. Dále pak rozšiřuje skripty pro pohyb po místnosti, sledování a vyhýbání se cizím entům a hledání předmětů.

Klíčová slova: umělá inteligence, hledání cest, grafové algoritmy, hierarchický A*

Title: Movement in The Ents Project
Author: Lukáš Bajer
Department: Department of Software and Computer Science Education
Supervisor: Mgr. Cyril Brom
Supervisor's e-mail address: brom@ksvi.mff.cuni.cz

Abstract: The ENTS project is a simulator of an environment which is similar to our common world. In this environment, there live autonomous agents called ents. They take care of their world. To fulfil their goals and satisfy their daily needs, they often have to look for a path around the world. This work is focused on scripts which are responsible for this pathfinding. The ents' movements are improved by hierarchical version of the A* algorithm. Thanks to this, demands on CPU during looking for longer paths are considerably decreased. In addition, ents' scripts are enhanced by better movement around a room, other ents following and avoiding, and "lazy" picking up objects.

Keywords: artificial intelligence, pathfinding, graph algorithms, hierarchical A*

Kapitola 1

Úvod

Snahy napodobit nebo modelovat lidské myšlení existují již poměrně dlouho. Praktické realizace vznikaly nejprve jen pomocí podvodů (například známý šachový automat Wolfganga Kempelena [13]) nebo v dílech literárních autorů (viz např. [3]), ale ve druhé půli dvacátého století již technické podmínky umožnily tento obor rozvinout i ve skutečnosti.

Postupně se rodí počítačové programy, které jsou schopny se na základě složitějších vstupních informací samy rozhodovat. S rostoucí výpočetní silou počítačů lze konstruovat složitější a propracovanější rozhodovací systémy a autonomní agenti, jak se tyto programy nazývají, nacházejí uplatnění nejen ve vědeckých laboratořích, ale i v průmyslu nebo zábavě.

Projekt ENTI je příkladem platformy pro práci s autonomními agenty. I když nás projekt po spuštění přivítá pěkným uživatelským rozhraním, patří spíše k projektům vědeckým. Je to diskrétní kolová simulace, která si klade za cíl zjistit, do jaké míry a za jakou cenu lze simulovat prostředí podobné lidskému světu (více viz [1]). Ve zjednodušeném modelu rodinného domku žijí postavy, které se nazývají *enti*. Jsou řízeni počítačovými programy napsanými převážně ve speciálním jazyce E. Elementárními úkoly, tzv atomickými instrukcemi[1], jsou schopni ovlivňovat svět kolem nich, ale umějí řešit i komplexnější úlohy[2] nebo obstarávat své životní potřeby.

Uživatel do světa vstupuje v roli jednoho z entů, který má stejná práva i omezení jako enti strojoví. Může tak stejně jako ostatní enti brát předměty, zavírat dveře nebo chodit na záchod. Tento druh práce nebo zábavy by nás

[1] např. udělej krok, otevři dveře, seber předmět před sebou

[2] např. zalej květiny na zahradě

ale pravděpodobně po čase přestal bavit. Možnosti projektu však tímto nekončí: zkušenější uživatelé mohou vytvářet své vlastní skripty, podle kterých se pak budou jejich enti chovat.

1.1 Cíl práce

Autorům nových entů se nabízí poměrně široká nabídka předpřipravených funkcí a skriptů, které mohou použít. Tato práce se zabývá skripty, které souvisí s jejich pohybem. Vylepšuje a opravuje některé stávající a doplňuje nové.

Konkrétní cíle jsou následující:

Zrychlení prohledávacích algoritmů. Čas v projektu ENTI běží po kolech, která trvají, dokud všichni enti nepošlou atomickou instrukci. Na výpočty tedy mají enti času teoreticky neomezeně, uživatel ho však vnímá. Prvním cílem je zmenšení časové náročnosti prohledávání mezi různými místnostmi aplikací hierarchické formy algoritmu A*.

Věrohodnější chování entů. Pohyb entů se v některých případech značně liší od toho, jak by se chovali lidé. Druhým cílem je některé nedostatky odstranit, konkrétně „naučit“ enty projít zatarasenou cestou (tzv. „bagrovat“), lépe pronásledovat a vyhýbat se ostatním entům a hledat předměty i při jiných činnostech.

1.2 Struktura dokumentu

Práce je rozdělena do sedmi kapitol. První z nich právě čtete. Uvádí do problému a vytyčuje základní cíle, kterých práce dosáhla. V *kapitole 2* najdete stručný a poněkud teoretický popis algoritmů pro hledání cest, mezi kterými jsme před jejich implementací volili. *Kapitola 3* hlouběji popisuje projekt ENTI, zejména pak vlastnosti a specifika, které bylo nutné vzít v úvahu při výběru algoritmů i během jejich aplikace. *Kapitola 4* je vedle softwarového díla vlastním jádrem práce, neboť nabízí popis konkrétně použitých algoritmů i s důvody, proč bylo nebo nebylo vhodné je použít. *Kapitola 5* je zaměřena přímo na změněné nebo nové skripty a nabízí bližší popis vlastní implementace a jejích problémů, stejně jako příklady jejich použití. *Kapitola 6* nabízí další pokračování práce, krátce diskutuje výhody a nevýhody použitých řešení a shrnuje dosažené výsledky.

Nedílnou součástí práce jsou kromě tohoto textu i zdrojové soubory, kterými je potřeba před kompilací projektu ENTI nahradit soubory původní. Více viz „Instalační příručka“ umístěná na přiloženém CD.

Kapitola 2

Algoritmy pro pathfinding

Standardem dnešních algoritmů na prohledávání dlaždicového prostoru je jistě algoritmus A* [5]. Pro jedno nalezení skutečně nejkratší cesty je se správnou heuristikou dokonce nejlepší možný. Aby však ve své klasické podobě našel s jistotou cestu mezi startovní a cílovou pozicí, musí při hledání zpracovat alespoň všechny dlaždice na dané cestě, spíše však (vlivem překážek) ještě o poznání více. To může vyžadovat příliš mnoho času a paměti. Vznikly sice mnohé zrychlení použitím lepších datových struktur nebo ořezáváním prohledávaného stromu (viz např. [7]), výrazného zlepšení však dosáhneme až použitím nějakého systematického přístupu.

Po krátkém nahlédnutí do klasického algoritmu A* se zaměříme na metody, které výměnou za zaručení optimality řešení získají podstatné zrychlení a zmenšení paměťových nároků. Dále nás budou zajímat pouze algoritmy pracující nad dlaždicovými mapami a naopak až na výjimky se nebudeme zabývat technikami steeringovými (viz např. [12]). V podstatě mimo náš zájem stojí také přístupy, které začínají s dopředu neznámým terénem a až cestou zjišťují topologii prostoru – enti znají design místností i předmětů v nich po startu dokonale, stačí pouze reagovat na změny.

Tato kapitola si nebere za cíl předvést vyčerpávající přehled všech algoritmů a technik, které spadají do vymezeného prostoru. Popisuje pouze ty, které nás nějakým způsobem zaujaly a u kterých se zdálo, že budou v dosti specifickém prostředí neprocedurálních jazyků projektu ENTI relativně snadno implementovatelné.

2.1 Algoritmus A*

A* patří do skupiny algoritmů, které hledají optimální cestu v ohodnocených grafech. Vyznačuje se použitím heuristiky, kterou usměrňuje hledání v nejslibnějším směru. Jeho jednoduchost o něm umožnila za určitých podmínek dokázat všechny podstatné vlastnosti jako správnost, konečnost nebo optimalitu.

Algoritmus po krocích prohledává zadaný graf. Začíná startovním vrcholem s, který označí jako *navštívený*, tj. umístí ho do seznamu *open*. V každém kroku jeden z navštívených vybere, prohlásí za *probraný*, tj. umístí ho do seznamu *closed*, a zpracuje jeho sousedy. Algoritmus končí v okamžiku, kdy je do seznamu *open* umístěn cílový vrchol c (pseudokód viz obrázek 2.1).

Aby mohl algoritmus co nejrychleji nalézt nejkratší cestu k cíli, musí správně vybrat, který z vrcholů v seznamu *open* v daném kole zpracovat. Algoritmus A* vybírá vrchol n, který má nejnižší hodnotu ohodnocovací funkce $f(n)$. Její hodnota se spočítá jako součet $f(n) = g(n) + h(n)$, kde

- $g(n)$ je nejkratší vzdálenost ze startovního vrcholu s do n přes doposud probrané vrcholy a
- $h(n)$ označuje odhad nejkratší možné vzdálenosti z n do cílového vrcholu c. Tento odhad nesmí být větší než skutečná nejkratší vzdálenost z n do cíle.

```
A*(start, cíl) {
    přidej (start, 0) do open
    while open neprázdný
        vezmi n nejlepší z open
        if (n == cíl) return „úspěch“
        přidej n do closed
        for (all x ∈ Succ(n) and x ∉ closed)
            if (x ∉ open)
                přidej (x, f(x)) do open
            else
                aktualizuj (x, f(x)) v open, pokud je třeba
    return „neúspěch“
}
```

Obrázek 2.1: Pseudokód jednoduché varianty algoritmu A*

Pro každého souseda $n' \in Succ(n)$ takto vybraného vrcholu je spočtena hodnota $f(n') = g(n') + h(n')$. Pokud n' ještě není v *closed*, přidá se s hodnotou $f(n')$ do seznamu *open*[1]. Jestliže již v *closed* je, přeřadí se do *open* jenom v případě, že je jeho nová hodnota $f(n')$ nižší než původní[2].

Po dosažení cílového vrcholu c se vlastní cesta získá postupně pomocí zpětných ukazatelů na předka, tj. ukazatelů na ten vrchol, ze kterého byl daný naposledy přidán do seznamu *open*.

Úspěch algoritmu je silně závislý na přesnosti odhadu vzdálenosti do cíle $h(n)$. Podhodnocení vede ke zbytečnému probírání vrcholů navíc, je-li však odhad větší než skutečnost, nalezení nejkratší cesty není zaručeno.

V porovnání s algoritmy, které heuristiku nepoužívají (například s Dijkstrovým algoritmem [4]) sice A* přináší značné zlepšení, ale zejména na delších vzdálenostech může zpracovávat vrcholů stále velmi mnoho.

2.2 Hierarchical Pathfinding A*

Jak již sám název napovídá, Hierarchical Pathfinding A* (dále HPA*) je typickým zástupcem hierarchických plánovacích metod. Poměrně jednoduchým způsobem rozdělí prohledávaný prostor a postaví jednodušší graf (ve smyslu počtu vrcholů), ve kterém lze hledat podstatně rychleji než v původním plánu. Tato „abstraktní cesta“ se pak podle potřeby „zjemní“ na konkrétní sled dlaždic.

Algoritmus si v iniciační fázi rozdělí danou mapu na tzv. *klastry*, což jsou pravidelné čtvercové oblasti. Praktické testy ukázaly (viz [2], str. 19), že vhodnou velikostí pro standardní mapy je 10×10 políček. Vybraná políčka, která bezprostředně sousedí s volnými políčky sousedních klastrů, pak určí za tzv. *průchody*. Tyto průchody se spojí jednak se sousedními průchody z jiných klastrů, jednak navzájem v rámci jednoho klastru, pokud mezi nimi existuje cesta (viz obrázek 2.2). Délky cest v rámci jednoho klastru se ukládají. Tímto způsobem se vytvoří tzv. *abstraktní graf*.

Postup vytváření klastrů a abstraktního grafu lze iterovat a vytvářet tak zmiňovanou hierarchii. Lze tedy „poodstoupit“ výš a budovat nad klastry dosavadními klastry větší (cca dvojnásobné) velikosti. Tím se ještě zmenší velikost abstraktního grafu a hledání více urychlí.

[1] Místo přidání již zařazeného vrcholu lze pouze aktualizovat hodnotu $f(n')$

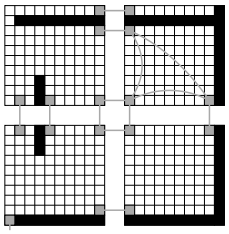
[2] Při použití tzv. *monotónní metriky*, což je případ většiny praktických aplikací, se vrcholy v *closed* již nepřehodnocují.

Při hledání určité cesty se s rostoucími klastry stává náročnější operace začlenění startovního a cílového políčka dané cesty do abstraktního grafu, zvětšování klastrů tedy není vhodné dělat do nekonečna (pro mapy velikosti cca 100×100 jsou vhodné 2–3 vrstvy hierarchie). Vlastní nalezení cesty v takto doplněném abstraktním grafu je naopak jednoduché a rychlé, použije se standardního algoritmu A* na graf, který má cca setinový počet vrcholů oproti původní mapě.

Abychom získali seznam políček, kudy skutečně povede cesta po mapě, je nutné provést tzv. *path refinement*, neboli zjemnění cesty. Tato operace se s výhodou může provádět postupně až když je potřeba konkrétní úsek cesty. Při programování umělých bytostí se totiž může snadno stát, že se řízení předá jiné části programu, nebo že naplánovaná cesta selže, takže počítání celé cesty dopředu může být často zbytečné.

Poslední, nepovinnou fází je tzv. *vyhlazení cesty*. Protože je počet *průchodů* mezi klastry omezen, mezi místy, které by se daly spojit přímou čárou, může vést cesta zbytečně klikatě. Vyhlazení tedy postupuje po spočtené cestě ještě jednou a kontroluje, zda-li nelze dlaždice průchodů v následujícím klastru spojit přímou čárou. V kladném případě původní cestu nahradí cestou jednodušší a kratší.

Autoři v [2] uvádějí, že praktické testy ukázaly až desetinásobné zrychlení oproti optimalizovanému A*. HPA* spotřebovává na svůj OpenList přibližně třetinu paměti oproti běžnému A*, počet navštívených dlaždic je až desetkrát menší. Produkuje o cca 6 % delší cestu než A*, když se však zařadí postprocessing (smoothing), jeho chyba je kolem 0,5 %. V uvedeném článku používají autoři osmisměrné dlaždice, ale nenašli jsme důvod, proč by algoritmus neměl stejně dobře fungovat i s dlaždicemi čtyřsměrnými, které používá projekt Enti. Dopředu nebylo jasné, do jaké míry budou moci enti ukládat informace o abstraktním grafu, celkově se ale HPA* ukázal jako dobrý kandidát. Za zmínku stojí i to, že se na algoritmu dále pracuje, což dokládají drobná zlepšení uvedená v [8].



Obrázek 2.2: Klastry a průchody algoritmu HPA*

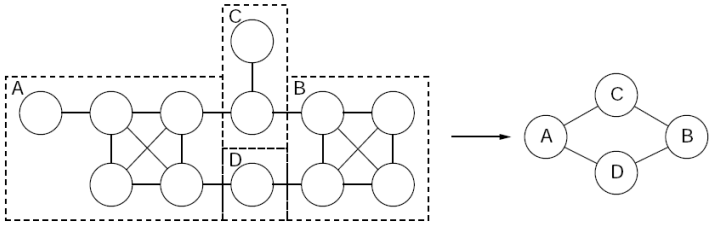
2.3 Partial-Refinement A*

Podobně jako v předchozím případě, i tento algoritmus používá hierarchii abstraktních grafů, její struktura i práce s ní je ale odlišná. Algoritmus se už na základní mapu dívá jako na graf: vrcholy odpovídají dlaždicím a hrana mezi nimi vede v případě, že spolu odpovídající dlaždice sousedí. Berou se při tom v úvahu mimo čtyř základních směrů i čtyři směry diagonální. Na rozdíl od HPA*, který překrývá původní mapu strukturou shora, Partial-Refinement A* (dále PRA*) slučuje vrcholy na základě lokálních vlastností grafu.

Algoritmus hledá ve zpracovávaném grafu *kliky*, které nahrazuje v budované (vyšší) vrstvě jedinými stavy. Spojováním klik je zaručeno, že všechny původní vrcholy náležící k novému stavu jsou od sebe vzdáleny jeden krok. Tedy téměř: ke klikám se do jednoho stavu přidávají také přilehlí *sirotci*, což jsou vrcholy připojené k dané klice pouze/právě jednou hranou (viz obrázek 2.3). Celý algoritmus zpracování grafu je určen k iteraci, dokonce ještě více než u HPA* – abstrakce končí v momentě, když už zbyde jenom jeden vrchol. Celá sada vrstev tvoří jakýsi strom, který PRA* využívá.

PRA* pro počáteční (*start*) a koncový (*goal*) vrchol nejdříve zkonstruuje takovou část stromu, aby *start* a *goal* byly v listech a aby měly jediného společného předka. Poté určí hladinu, od které začne iterovaně vylepšovat cestu – nesmí být ani moc hluboko (algoritmus by provedl příliš mnoho kroků) ani moc mělko (výsledná cesta by se mohla od optimální hodně lišit). V každé iteraci vezme v aktuální hladině cestu z vrcholu „obsahující“ *start* do vrcholu, který vznikl abstrakcí vrcholu *goal*, rozbalí jednu úroveň níž a v tomto „pásu“ najde nejkratší cestu pomocí A* (pseudokód viz obrázek 2.4).

Také PRA* umožňuje vylepšovat cestu postupně (tj. jaksi cestou), dokonce ho lze přímo parametrizovat, kolik dlaždic dopředu má počítat – na tuto vlastnost je



Obrázek 2.3: Konstrukce abstraktního grafu PRA* (zdroj: [14])

připraven lépe než HPA*. Autoři v [14] nabízí rychlostní srovnání s mírně optimalizovaným A*. Časová úspora oproti A* rychle roste s délkou hledané cesty: cestu délky 400 dlaždic PRA* najde až desetkrát rychleji, naopak na malých mapách (cesty délky cca 50 a méně) může PRA* dokonce mírně ztrácet. Délka nalezené cesty na testovaných mapách se vůči cestě A* neliší o více než 5 %, v 95 procentech případů je pak delší o méně než 0,5 %. V článku [14] autoři popisují algoritmus s osmisměrnými dlaždicemi, což je dle našeho soudu poměrně klíčová podmínka: v grafu ze čtyřsměrných dlaždic mají kliky velikost nejvýše 2. Na druhou stranu tím, že PRA* při stavbě abstraktního grafu vychází jen z přístupných dlaždic, může být například pro řídké mapy efektivnější, navíc striktně nevyžaduje celou mapu v jedné čtvercové síti, což by se mohlo v prostředí projektu ENTI ukázat jako výhoda.

```
PRA*(abstraktníGraf, start, goal) {
    PostavHierarchiiAbstrakcí(start, goal)
    s = VyberStartovníÚroveň(start, goal)
    vyprázdni cestu
    for každou úroveň i = s to 1
        cesta = ZjemniCestu(cesta, start[i], ocas(cesty))
    return cesta
}

ZjemniCestu(cesta, start, goal) {
    return aStar(start, goal) omezený na vrcholy v cestě
}
```

Obrázek 2.4: Pseudokód algoritmu PRA*

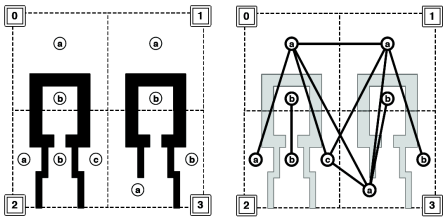
2.4 Další metody

Memory-Efficient Abstractions

V článku [15] je představen další přístup ke generování abstraktních grafů. Do jisté míry je kombinací dvou předešlých metod. Stejně jako HPA* rozdělí původní plán na pravidelnou síť *sektorů*. Sektory dále rozdělí na souvislé oblasti, tzv. *regiony*, které se stávají vrcholy budoucího abstraktního grafu. Ten vzniká spojením těch regionů, které mezi sebou sousedí (viz obrázek 2.5).

Vlastní algoritmus je pak podobný HPA* – nejdříve se najde pomocí A* cesta v abstraktním grafu, která se podle potřeby zjemní a následně vyhladí. Z algoritmu PRA* si tento přístup převzal omezení hledání v nižších patrech jen na „pás“ tvořený regiony z cesty v abstraktním grafu.

Článek se dále detailně zabývá postupným zjemňováním, vyhlazováním nebo paměťovou efektivitou, to však pro naše použití není podstatné.



Obrázek 2.5: Sektory a regiony Memory-Efficient Abstractions (zdroj: [15])

Incremental heuristic search methods

Typickým zástupcem této skupiny algoritmů je D* Lite [9]. Tyto metody používají heuristiku, aby omezily a zacílily hledání, a zároveň využívají dříve spočtených cest z minulých běhů algoritmu. Opravují jen to, co se změnilo a která část již prohledané mapy je díky tomu potřeba opravit. Šetří tak nemalé množství prohledávání, neboť změny často zasahují jen malou část mapy (např. kvůli zamčeným dveřím).

D* Lite začíná hledáním cesty od cílové dlaždice ke startovní. Jeho postup při prvním hledání je v podstatě totožný s hledáním běžného A*, jen kromě odhadu vzdálenosti z cílové dlaždice, hodnoty g, používá odhad ještě jeden, hodnotu rhs.

Je to jakýsi jednokrokový výhled, přesněji minimum ze součtů hodnot $g(s')$ sousedů dané dlaždice a ceny přechodu z tohoto souseda $c(s', s)$:

$$rhs(s) = \begin{cases} 0 & \text{pokud } s = s_{start} \\ \min_{s' \in Pred(s)}(g(s') + c(s', s)) & \text{jinak.} \end{cases}$$

Tyto hodnoty uchovává u každé dlaždice a aktualizuje je v okamžiku, když nalezne změny v mapě. Změnou jedné z hodnot se stane dlaždice nekonzistentní, což může mít za následek aktualizaci hodnot i dlaždic sousedních.

Výsledná cesta se pak najde projitím dlaždic s z cílové dlaždice s_{goal} tak, že se vždy vybírá předchůdce $s' \in Pred(s)$ s minimální hodnotou $g(s') + c(s', s)$.

I tato skupina algoritmů je kandidátem na použití v projektu ENTI, protože změny v konfiguraci světa většinou nebývají obrovské. D* Lite mezi nimi vyniká svou jednoduchostí a výborným popisem.

Závěr

Z uvedených algoritmů byl po zvážení výhod a nevýhod vybrán HPA*, který byl upraven tak, aby co nejlépe pracoval v prostředí projektu ENTI. Více o tom, jaké důvody k výběru vedly a jak byl HPA* upraven, se dočtete v kapitole 4.1 na straně 24.

Kapitola 3

Projekt Enti

3.1 Struktura projektu Enti

Projekt Enti není monolitický – lze na něho spíše nahlížet jako na soubor několika úzce provázaných softwarových nástrojů[1]. Nejdůležitějšími jsou *server prostředí*, *prohlížeč* a `ent`[2].

Server prostředí je ústřední složkou projektu. Jeho úkolem je řídit veškeré fyzikální děje v modelovaném světě a komunikovat s enty a prohlížečem. Server prostředí od nich přijímá informace o jejich činnosti, opačným směrem pak posílá údaje o změnách[3], které ve světě nastaly.

Prohlížeč je uživatelským rozhraním projektu. Stejně jako `ent` je to klient, který se připojuje k serveru prostředí. Prostřednictvím prohlížeče může uživatel do světa vstupovat jako jeden z entů a sledovat a ovlivňovat tak dění ve světě.

Program `ent` je komponentou projektu, jež reprezentuje jednoho enta – strojovou bytost obývající modelovaný svět. Tato bytost je schopna vnímat děje ve svém bezprostředním okolí a tyto vjemy shromažďovat ve své paměti[4]. Její chování a rozhodování je řízeno skripty, pomocí kterých může skládat jednoduché úkony (tzv. atomické instrukce) do složitějších úloh.

[1] citace z [1], část Uživatelská příručka, str. 15

[2] Stejně jako v [1] označuje slovo ent napsané normálním písmem simulovanou bytost. Totéž slovo, ale neproporcionálním písmem, je název spustitelného programu, který bytost řídí. Enti v množném čísle a kapitálkami pak občas budeme používat místo sousloví „projekt Enti“.

[3] citace z [1], část Uživatelská příručka, str. 17

[4] citace z [1], část Uživatelská příručka, str. 19

Ke svému běhu potřebují ENTI i tzv. *balík světa*. Je to sada konfiguračních souborů, která popisuje simulovaný svět. Lze s nimi měnit vzhled a strukturu světa. Obsahuje též některé skripty a další konfigurace pro jednotlivé enty.

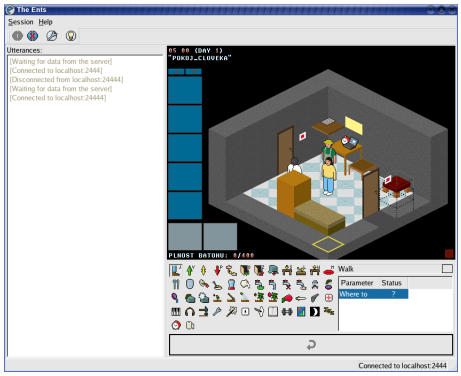
3.2 Svět entů

Jak bylo uvedeno výše, jedním z cílů projektu ENTI je simulovat svět podobný světu našemu. Autoři zvolili některá zjednodušení, která tuto práci a konečný výběr algoritmů podstatným způsobem ovlivňují. Jde zejména o následující (viz [1], část Teoretická dokumentace, str. 10):

Diskrétní prostor a objekty. Prostor světa entů je rozdělen na malé dlaždice ve čtvercové síti. Na dlaždicích mohou být umístěny předměty, včetně entů samotných. Předměty mohou (podle určitých pravidel) zase obsahovat předměty. Prostor je dále rozdělen do samostatných úseků, tzv. místností.

Diskrétní běh času. Simulace je prováděna po kolech, v každém kole všichni enti dostanou celý popis místnosti, kde se nachází, o změnách v jiných místnostech se naopak nedozví nic. V každém kole posílá každý ent serveru atomickou instrukci a mezi instrukcemi provádí výpočty.

Omezený čas i prostor. Entí čas je zdola omezen startovním okamžikem projektu. Entí prostor je omezený, v simulovaném světě je dopředu dán počet i velikost místností, místnosti ani dlaždice nemohou přibývat ani ubývat.



Obrázek 3.1: Spuštěný prohlížeč ENTŮ

Enti se mohou pohybovat ve čtyřech směrech. Neexistuje stav „natočení“ ani instrukce pro otáčení – enti mohou udělat v dalším kole krok na kteroukoli ze čtyř sousedních dlaždic.

Místnosti jsou mezi sebou propojeny dveřmi. Mezi některými místnostmi mohou vést schody, které svět rozdělí do několika pater. Entův pohled na topologii místností je však jednodušší než by se mohlo z uvedeného popisu zdát: ent jen ví, které místnosti jsou propojeny kterými dveřmi. Schody od dveří v podstatě nerozlišuje. Veškeré informace o umístění předmětů a ostatních entů jsou vždy relativní k jednotlivým místnostem, žádné absolutní souřadnice neexistují a vlastně je ani žádná část projektu nepotřebuje.

3.3 Programování entů

Jak již bylo několikrát zmíněno, chování entů je řízeno skripty. Většina z nich je napsaná v jazyce E. Tyto skripty jsou interpretované přímo `ent`em, a tak je lze poměrně snadno měnit nebo rozšiřovat a do značné míry tak ovlivňovat chování každého enta zvlášť. Některé výpočtové skripty nebo funkce pro práci s databází jsou však v jazycích Mercury [11] nebo C a kompilují se přímo jako součást `enta`. Jejich změny pro běžné uživatele příliš přívětivé nejsou (často je potřeba měnit několik souborů na různých místech, neexistuje debugger), odměnou je však vyšší rychlost a do jisté míry i spolehlivost.

Pojďme se na jazyk E podívat trochu podrobněji. Základní strukturou pro zápis algoritmů je tzv. *skript*. Je na cestě mezi prologovským termem a procedurou v C (jakkoli se to zdá nemožné). Zatímco výběr toho, který skript se spustí, se děje podle ryze specifických pravidel, vykonávání příkazů uvnitř skriptu probíhá do značné míry procedurálně. K dispozici jsou i příkazy jako `repeat`, `if`—`then` nebo `for`-cyklus, na druhou stranu lze použít verzi s backtrackingem i bez něj.

K výběru toho, který skript z několika variant se má použít, slouží *hodnotící funkce*. Ta existuje pro každý skript. Na základě stavu enta a jeho okolí určí, jak moc je pro enta daná verze skriptu zajímavá, a interpret pak vybere nejslibnější variantu.

Podobný systém používají *priority*. S jejich pomocí jazyk E dovoluje provádět více úloh najednou a jen mezi nimi přepínat podle toho, která z nich je nejdůležitější (má nejvyšší prioritu). Tyto priority se navíc mohou měnit v čase, takže lze elegantně plánovat vykonávání různých úkolů, obstarávání životních potřeb apod.

Dalším neméně zajímavým jevem je tzv. *interrupt*. Při programování umělých bytostí je často potřeba zajistit po určitou dobu platnost nějakých podmínek. Interrupty jsou skripty, které spoustu jinak nutných testů nahradí elegantním řešením: po jejich aktivaci (tzv. „navěšení“) se začnou provádět jakmile začne platit námi zvolená podmínka. Příklad: jestliže ent potřebuje získat konev, ale o žádné neví, lze navěsit interrupt „seber konev a skonči“ na podmínku „vidím konev“. Stačí pak procházet svět a jakmile ent konev uvidí, sebere ji a skript úspěšně skončí.

Jazyk E má ale i nevýhody. Rozlišuje jen čtyři datové typy: *handle* (v podstatě čtyřbajtové celé číslo sloužící k reprezentaci všech entů, místností, objektů i čísel), *seznam* (běžná datová struktura neprocedurálních jazyků, může obsahovat objekty typu handle), *member* (struktura s pevně danými položkami nebo jinak také řádek relační tabulky, používaná ke konkretizaci předmětů) a *superseznam* (seznam memberů). Zcela chybí například pole, se kterými by mnohé algoritmy pracovaly podstatně efektivněji.

Omezené jsou i možnosti ukládání dat mezi běhy skriptů. Existují jen dvě možnosti: *globální proměnné* a *entova databáze*. První z nich je sice jednoduchá a patrně i rychlá, nelze ale použít ve funkcích psaných v Mercury nebo C. Přidávat nový typ záznamu do entovy databáze tak, aby byl přístupný i z Mercury/C-funkcí, sice lze, vyžaduje to však velké úsilí v konfiguraci mnoha dalších souborů, přidávání nových handlů atd. Do entovy databáze navíc nelze ukládat nic jiného než handly, takže uchovávání většího množství dat, například určité informace ke každé dlaždici na mapě, by databázi velmi zaplnilo a zpomalilo.

3.4 Pohybové skripty

Skripty a funkce, které jsou zaměřeny na entův pohyb a navigaci, jsou soustředěny v knihovnách skriptů jazyka E i ve zdrojových souborech napsaných v Mercury. V první skupině najdeme především řídící výkonné skripty, které obsahují „vyšší logiku“, atomické instrukce a často využívají interrupty, například `jdiNaDlazdici/5`, `jdiDoMistnosti/3` nebo `jdiKEntovi/2`. Do druhé skupiny patří hlavně výpočetní funkce hledající vlastní cestu po dlaždicích nebo mezi místnostmi, často komunikující s databází. Příkladem jsou funkce `cSmerKDlazdici/6` nebo `cDvereSmeremK/6`.

První cíl této práce (viz kap. 1.1, str. 7), zrychlení prohledávacích algoritmů, se bude naplňovat především ve druhé skupině, tedy v Mercury. Naopak pro skripty, které budou zlepšovat věrohodnost chování entů, nabízí jazyk E poměrně širokou

paletu nástrojů. Většina změn spadajících do této oblasti se tedy bude odehrávat ve zdrojových souborech ze skupiny první. Konkrétní popis toho, co a jak bylo zlepšeno, najdete v následujících dvou kapitolách.

Kapitola 4

Použité postupy a algoritmy

Cíle této práce byly sice vytyčeny v kapitole 1.1, nikoliv však příliš konkrétně. Následující kapitola představí problémy, kterými jsme se zabývali, podrobněji, a to včetně obecného popisu jejich řešení. Jde o hierarchické hledání cest pomocí varianty algoritmu A* (4.1), tzv. bagrování (4.2), pronásledování (4.3) a vyhýbání cizím entům (4.4) a sbírání předmětů při jiných činnostech, tzv. on-line hledání (4.5). Implementační detaily a příklady použití naleznete v kapitole 5.

4.1 Hierarchické hledání cest

Přestože virtuální svět ENTŮ je typicky pouze několik místností veliký, enti se v něm musí rychle a přesvědčivě pohybovat. Najít cestu musí v podstatě k jakémukoli úkolu – zahradník musí pro zeleninu dojít na zahradu, umýt se musí enti u umyvadla nebo ve sprše apod. Stejně jako v jiných podobných aplikacích (nejčastěji v počítačových hrách), hledání cest je nejen hojně používáno, ale navíc jako takové zabírá poměrně mnoho výpočetního času. To zjistíme zejména v případě, když projekt ENTI spustíme na starším počítači (cca 1,3 GHz a méně). Cíl je tedy zřejmý: hledání cest zefektivnit při současném zachování nebo dokonce zlepšení věrohodnosti pohybu.

Jak už jsme uvedli, původní představa entů o topologii místností byla velmi jednoduchá: enti pouze věděli, která místnost je se kterou spojena dveřmi. Je sice pravdou, že na to, aby došli tam, kam potřebují, to stačí, ale tento přístup neumožňuje k hledání cesty mezi různými místnostmi použít lepší algoritmus než Dijkstrův. Ten se v případě ENTŮ, kde jsou všechny místnosti podobně velké, moc neodlišuje od prohledávání do šířky. Enti si navíc spočtenou cestu přes místnosti

nepamatovali – vyrazili do první místnosti, kudy cesta vedla, a tam celou cestu počítali znova.

Původní návrh hledání cest mezi dlaždicemi v různých místnostech byl do jisté míry hierarchický již od začátku – nejdříve si enti naplánovali, přes které místnosti půjdou, a v nich pak hledali konkrétní cestu po jednotlivých dlaždicích. Neumožňoval však ve vyšších vrstvách použít žádnou heuristiku.

Absolutní souřadnice dlaždic

Jako první krok k efektivnějším algoritmům jsme proto napsali sadu skriptů a Mercury-funkcí, které všechny místnosti zasadily do jakési virtuální trojdimenzionální sítě, tj. každé místnosti přiřadily souřadnice (x, y, z) jejího severozápadního rohu (hodnoty jsou relativní vůči startovní pozici enta). Souřadnice v rámci místnosti jsou přímo uloženy v prvním a druhém bajtu handlu dané dlaždice, pozice konkrétních dlaždic jsou tedy prostým součtem dvou vektorů po složkách.

Postup získání této sady souřadnic je poměrně jednoduchý: ent (virtuálně) prochází jednotlivé místnosti do šířky. Každé dveře, na které narazí, jsou již v prošlé části mapy, a jejich absolutní souřadnice jsou tedy známy. Pomocí relativní pozice dveří v nové místnosti se pak snadno spočítá pozice jejího severozápadního rohu.

Procházení probíhá po jednotlivých patrech – pokud ent narazí na schody, počká s jejich projitím, dokud není prošlé celé předchozí patro. Svět entů nerozlišuje, zda schody vedou nahoru nebo dolů. Čísla pater (souřadnice z) tedy nemusí odpovídat zamýšlenému návrhu světa; je to spíše pořadí, v jakém se jednotlivá patra zpracovávala. Aby se však topologie pater neztratila, ent si vytváří paralelně další strukturu, tzv. *graf pater*. Tam ukládá, které patro je kde a se kterým spojeno schody. K jeho využití se vrátíme později.

Tento krok umožnil u každé dlaždice rychle zjistit její přibližnou vzdálenost od kterékoli jiné. Protože enti chodí jen do čtyř směrů, nejlepší metrikou je jistě Manhattanská, která vzdálenost rychle spočítá jako součet absolutních hodnot rozdílů odpovídajících souřadnic

$$d(A, B) = |A_x - B_x| + |A_y - B_y| + |A_z - B_z|.$$

Jak bylo uvedeno v předchozím odstavci, význam vzdálenosti v ose z je diskutabilní, pro naše účely však stačí.

Výběr, úpravy a aplikace HPA*

V tomto bodě již bylo třeba rozhodnout, který z algoritmů uvedených v kapitole 2 aplikovat. HPA* se nabízel svou jednoduchostí, v prostředí projektu ENTI však má své slabiny: mapa může být poměrně řídká a není zaručeno, že se místnosti po zasazení do 3D sítě nebudou překrývat. Další v pořadí, PRA*, se zdál na první pohled jako dobrý kandidát: abstraktní graf vychází přímo z dané mapy, nabízí velkou variabilitu, navíc je algoritmus aktivně používán a vyvíjen. Po bližším zkoumání byl ale zavržen: požadavek na osmisměrné dlaždice je příliš silný, nad to by bylo přinejmenším nepříjemné ukládat do entovy databáze množství abstraktních grafů. D* Lite a jiné algoritmy ze skupiny *incremental heuristic search metods* byly zavrženy kvůli omezenosti entovy databáze, navíc nedosahují takových zlepšení v rychlosti jako hierarchické přístupy. Poslední z uvedených, Memory-Efficient Abstractions, využívá zajímavého přístupu v dělení mapy na souvislé oblasti, ale nepředstavuje příliš komplexní řešení.

Po zvážení kladů a záporů popsaných metod jsme nakonec zvolili svůj vlastní přístup. Je založen na stejné myšlence jako HPA*, naplno ale využívá dodatečných informací a implementačních detailů naší konkrétní aplikace, tedy projektu ENTI. Postup uvedený v [2] jsme upravili především v těchto bodech:

Klastry nahrazeny místnostmi. Místo klastrů ("clusters" v článku [2]) velikosti cca 10×10 polí jsme využili nativního rozdělení světa na místnosti. Toto řešení přináší mnohé výhody. Velmi dobře, přesně a výstižně jsou definované průchody mezi klastry ("enters" a "transmissions" v [2]) pomocí dveří a schodů mezi místnostmi. Je tím zaručena vysoká přesnost heuristiky (přesněji to vlastně ani nejde – enti nemohou projít přes zeď), není potřeba path smoothing (není kudy cestu zlepšit) a navíc je celá implementace o poznání jednodušší. Existují samozřejmě i záporné stránky. Místnosti jsou obvykle menší než 10×10 polí a průchodů může být více než v originálním řešení HPA*, takže hledání v abstraktním grafu je náročnější.

Líné počítání vzdáleností. Původní podoba HPA* navrhuje předpočítat vzdálenosti mezi průchody (v našem případě mezi dveřmi) v každém klastru dopředu. Naše práce používá jiný přístup: nejdříve se používá odhad Manhattanskou metrikou. Přesná vzdálenost se spočítá až v okamžiku, kdy ent skutečně cestu mezi dveřmi potřebuje naplánovat a projít. Spočtenou vzdálenost si pak uloží do databáze a příště ji již použije místo odhadu.

Toto chování navíc odpovídá chování lidí – chce-li člověk rychle odhadnout

nějakou vzdálenost, může se snadno dopustit chyby. Pokud si ale nějakou trasu opravdu projde, odhad se podstatným způsobem zpřesní.

Druhá abstraktní vrstva tvořená patry. Autoři v [2] vůbec neuvažují, že by mapa mohla vybočit ze dvou rozměrů. Bylo proto potřeba najít způsob, jak nejlépe vyřešit hledání v několika patrech. Celkem přirozeně se nabídlo řešení vytvořit mezi nimi druhou abstraktní vrstvu. Právě zde se uplatní *graf pater* generovaný při ukládání absolutních pozic místností (viz str. 23).

Hledání jen v první abstraktní vrstvě. Podobnost v entově vnímání dveří a schodů umožnila jednoduše implementovat poměrně překvapivou věc: plánovat cestu mezi místnostmi v různých patrech lze stejně dobře použitím obou abstraktních vrstev (tzn. nejdříve zjistit, přes která patra a schody je potřeba jít, a pak teprve naplánovat nejkratší cesty přes místnosti mezi schody v každém patře) i když se bude hledat jen ve vrstvě první (na schody se bude ent dívat jako na obyčejné dveře a druhá abstraktní vrstva se nepoužije). První přístup je pro delší cesty rychlejší a zřejmě i podobnější lidskému jednání, druhý pak umožňuje najít cesty o něco kratší (používá se přesnější heuristika, patra lze opouštět a zase se do nich vracet).

Vlastní algoritmus nalezení cesty se od původního algoritmu HPA* v podstatě neliší (pseudokód viz obrázek 4.1). Používá-li se graf pater, najde se nejdříve, přes které schody je potřeba projít. Tato cesta se pak zjemní o sled dveří tvořící nejkratší cestu mezi schody. Pokud se druhá abstraktní vrstva nepoužije, získáme sled dveří a schodů rovnou. Za zmínku stojí, že právě v tomto bodě hrají klíčovou roli absolutní souřadnice dlaždic, dveří a schodů. Účinně lze totiž používat algoritmus A* i při hledání mezi různými místnostmi, což původně nebylo možné.

Ent spočítá cestu přes aktuální místnost k nejbližším dveřím a vyrazí k nim. Cestu přes další místnosti počítá vždy až když do nich vstoupí. Jednak se tímto rozděluje zátěž procesoru na víc částí, čímž se běh Entů stává plynulejším, jednak se nepočítá zbytečně dopředu v případě, že je nalezená cesta neprůchodná nebo je potřeba přeplánovat úlohu, a v neposlední řadě se stává hledání přesnější, neboť ent získá po vstupu do místnosti aktuální přehled o všech předmětech i entech, kteří se v ní nachází.

Popsaná metoda snižuje časové nároky na hledání cest, což by se mělo výrazněji projevit zejména u delších vzdáleností. Kromě popsaného hierarchického přístupu a úspory hledání použitím algoritmu A* se čas uspoří i odstraněním hledání celé cesty v každé místnosti znovu – skripty si cestu pamatují a k přeplánování dochází až když selže některý z podskriptů.

```
cNajdiCestuDo(start, cíl, přesnost, použijPatra) {
    if ((StejnéPatro(start, cíl) and použijPatra == true)
            or použijPatra == false) then
        omezNaJednoPatro = použijPatra
        (dlaždicePředeDveřmi, vzdálenost) =
            aStar(start, cíl, přesnost, omezNaJednoPatro)
    else
        omezNaJednoPatro = true
        dlaždiceUSchodů = aStarVraťJenPatra(start, cíl, přesnost)
        (dlaždicePředeDveřmi, vzdálenost) =
            ZjemniPatra(start, cíl, dlaždiceUSchodů, omezNaJednoPatro)
    return (dlaždicePředeDveřmi, vzdálenost)
}

ZjemniPatra(start, cíl, dlaždiceUSchodů) {
    omezNaJednoPatro = true
    abstraktníCesta = [start, dlaždiceUSchodů, cíl]
    dlaždicePředeDveřmi = [ ]
    vzdálenost = 0
    do
        (s, g) = VezmiPrvníDvojiciZ(abstraktníCesta)
        (novéDlaždice, nováVzdálenost) =
            aStar(s, g, přesnost, omezNaJednoPatro)
        dlaždicePředeDveřmi = dlaždicePředeDveřmi + novéDlaždice
        vzdálenost = vzdálenost + nováVzdálenost
    while neprázdná abstraktníCesta
    return (dlaždicePředeDveřmi, vzdálenost)
}
```

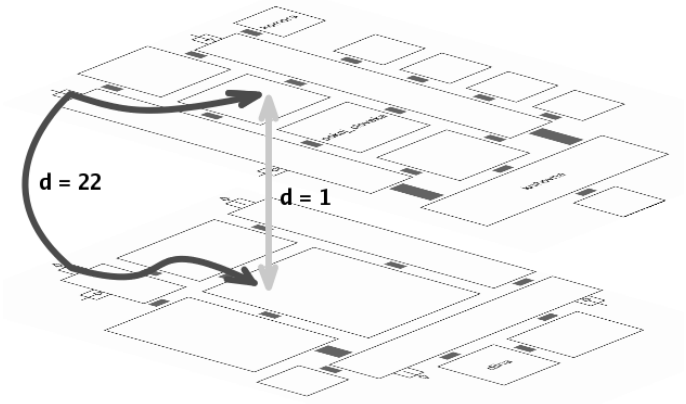
Obrázek 4.1: Pseudokód hledání mezi různými místnostmi

Poznámka o měření vzdáleností

Algoritmus A* potřebuje k odhadu vzdálenosti do cíle rychlou a tzv. přípustnou metriku. V případě čtyřsměrných dlaždic se v naprosté většině případů používá již zmíněná metrika Manhattanská a ani v této práci není použito jiné. Je to jakýsi dolní odhad vzdálenosti do cíle.

Problém se ale objevuje ve vertikálním směru. Jak již bylo uvedeno výše, význam rozdílu v souřadnicích z příliš neodpovídá vzdálenosti pater mezi sebou. Navíc vzdálenost mezi dlaždicemi přímo nad sebou bude nejspíš ve skutečnosti mnohem delší než 1, neboť cesta musí vést přes nějaké schody – heuristika je silně podhodnocená (viz např. obrázek 4.2). Následky nejsou katastrofální, popsaný jev ale může vést v případě vynechání druhé abstraktní vrstvy (grafu pater) ke zbytečnému hledání v necílových patrech do míst, kudy cesta nepovede. Do jiných pater lze totiž vstoupit pouze po schodech, hledání jinam tedy nemá smysl. Problém je naopak zcela odstraněn, když se druhá vrstva použije, neboť hledání cesty je pak omezeno vždy jen na jedno patro.

Druhý typ vzdáleností, se kterými A* pracuje, je vzdálenost dosud uražená. Ta by měla být co nejpřesnější, neboť při použití špatného odhadu nemusí algoritmus najít cestu nejkratší. Popsané algoritmy sčítají vzdálenosti uražené přes jednotlivé místnosti. Na jejich měření lze použít tři stupně přesnosti: (1) odhad Manhattanskou metrikou (nejméně přesný), (2) vzetí vzdálenosti z entovy databáze, pokud tam je, jinak použití předchozího postupu, (3) přesné naplánování cesty



Obrázek 4.2: Vzdálenosti ve vertikálním směru. Heuristika ($d = 1$) se může od skutečnosti ($d = 22$) značně lišit.

mezi dveřmi a uložení nové vzdálenosti do databáze (to i v případě, když v databázi byla stará hodnota). Je na autorovi, který skript používá, jakou přesnost a rychlost zvolí.

4.2 Bagrování

Zatímco předchozí část (4.1) se věnovala spíše teoretickému a algoritmickému zlepšení odpovídající prvnímu cíli této práce, následující části se zaměří na praktičtější a viditelnější chování entů.

Při hledání cesty po jedné místnosti se používal algoritmus A* i v původní verzi Entů. V některých případech však nefungoval správně. Příkladem je situace, kdy byla cesta na cílovou dlaždici zahrazena předměty, které lze sebrat. Když má člověk zablokovanou chodbu věcmi, posbírá je, přemístí a chodbu tak uvolní. Tohoto však enti nebyli schopni.

Náprava byla poměrně jednoduchá. Zásah vyžadoval algoritmus A* tak, aby zaplněné dlaždice správně penalizoval, ale aby cestu přes ně dokázal najít. Ve skriptu, který má na starosti vlastní pohyb, pak již stačilo přidat schopnost nejdříve předměty posbírat, pokud se ent na naplánovanou dlaždici nevejde.

S pohybem po místnosti souvisí i další podproblém. Pokud se enti chtěli dostat na dlaždici ve stejné místnosti, ve které stáli, hledali cestu pouze po této aktuální místnosti. Jestliže je však místnost zatarasena předměty, může být výhodnější použít okolní místnosti a překážku obejít jinudy.

Problém byl postupně vyřešen dvěma způsoby. První testuje cesty okolními místnostmi v případě, že cesta přes aktuální je delší než dvojnásobek Manhat-

Obrázek 4.3: Bagrující ent

tanské vzdálenosti. Postupně zkouší cesty všemi dveřmi, zda-li nemůže být kratší. V kladném případě enta pošle touto cestou, nenajde-li žádnou kratší cestou, ent se vydá aktuální místností.

Druhý přístup využívá algoritmu z předchozí části. Upravením skriptu pro hledání cesty mezi různými místnostmi jsme získali obecnější variantu, která umožňuje hledat cestu mezi kterýmikoli dlaždicemi, tedy i dlaždicemi jedné místnosti. Bere při tom v úvahu i cesty přes místnosti okolní, na což používá uvedenou úpravu algoritmu A*.

4.3 Pronásledování

Úkolem skriptu `jdiKEntovi/2` je enta dovést k jinému specifikovanému entovi (dále „oběti“). V původní verzi se snažil enta nasměrovat vždy přímo na svoji oběť. Pokud se hnula, skript přeplánoval cestu na novou pozici a ent vyrazil novým směrem.

Pokud ale budeme zkoumat chování lidí, zjistíme, že v mnoha případech své oběti nadběhnou a zkrátí si tak cestu, zvláště pak v případě, kdy se jim jejich cíl nesnaží utéci. V entím světě čtyřsměrných dlaždic to lze vyjádřit přesněji. Označíme-li

- změnu souřadnic, o které se změní pozice naší oběti za jedno kolo, jako $(o_{\Delta x}, o_{\Delta y})$, kde $o_{\Delta x}, o_{\Delta y} \in \{-1, 0, 1\}, |o_{\Delta x} + o_{\Delta y}| = 1$,
- pozici oběti jako (o_x, o_y),
- a pozici našeho enta jako (e_x, e_y),

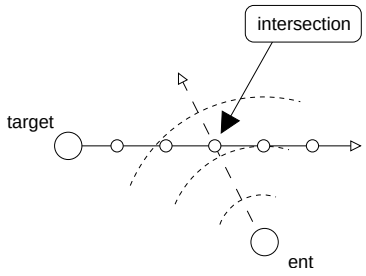
lze v dané místnosti alespoň trochu nadběhnout pokud

$$\mathrm{sgn}(e_x - o_x) = \mathrm{sgn}(o_{\Delta x}) \neq 0 \quad \vee \quad \mathrm{sgn}(e_y - o_y) = \mathrm{sgn}(o_{\Delta y}) \neq 0.$$

Použijeme techniku „pronásledování“ (v orig. “pursuit”) popsanou v [12] a upravenou pro prostředí čtyřsměrných dlaždic. Z pozice oběti v aktuálním a předchozím kole lze odhadnout, kudy se naše oběť bude pohybovat, pokud se nezastaví ani nezmění směr. Nebo přesněji, za kolik kol bude na které dlaždici. Pokud je splněna výše uvedená podmínka, je možné z této informace snadno spočítat pravděpodobné místo střetu, na které můžeme našeho enta poslat (viz obrázek 4.4).

Pokud uvedená podmínka splněna není, lze ještě zkusit jinou možnost, a to odhadnout dveře, do kterých naše oběť míří. Jestliže ent bude schopný dorazit do vedlejší místnosti dříve než oběť, zkusí jí nadběhnout sousedními dveřmi.

Jestliže selže i tato možnost, nezbývá než se vydat přímo za naší obětí s tím, že jí třeba bude možné nadběhnout později.



Obrázek 4.4: Pronásledování – místo střetu

4.4 Vyhýbání se ostatním entům

V původní verzi se enti při plánování cesty přes místnost snažili vyhnout pouze dlaždicím, na kterých ve chvíli hledání někdo stál. To však bylo zvláště na delší vzdálenosti téměř bezpředmětné – situací, kdy se cizí enti nehýbou, je v jejich životě jen málo.

Naše metoda na zlepšení vychází z postupu uvedeného, stejně jako v předchozím případě, v [12], konkrétně „předcházení kolizím“ (v orig. “unaligned collision avoidance”). Odhadnou se místa, kde by mohlo dojít ke kolizím (pokud se cizí enti budou pohybovat stejně jako v posledním kole), a ent se bude snažit vyhnout těmto místům očekávané srážky.

První část řešení je do jisté míry analogická s předchozím případem: sledování ostatních entů v místnosti a odhad toho, kterým směrem půjdou. Dobře nám při tom poslouží jazyk E a jeho interrupty, které dovolí sledovat pohyb ostatních entů bez enormního množství testů a podmínek. V dalším kroku se pak pozice a směry, kterým ostatní enti jdou, předají výpočtovým Mercury-funkcím, které odhadnou budoucí cesty ostatních entů a vypočtené dlaždice v příslušných kolech při prohledávání algoritmem A* penalizují. Tím je dosaženo požadovaného chování – dlaždice, na které míří ostatní enti, A* nevybere a ent se jim tedy vyhne.

4.5 On-line hledání

Pokud chce ent najít určitý předmět, nejdříve se podívá do své databáze. Jestliže tam odpovídající předmět najde, jde tam, kde si myslí, že je, a vezme ho. V případě, že o žádném takovém neví, vyrazí na procházku celým světem a snaží se najít nějaký splňující daná kritéria.

Tato práce představuje ještě jiný přístup k hledání věcí, které (zatím) ent nevidí. Dal by se nazvat například „hledání mimoděk“. Hlavní myšlenka tkví v tom, že ent předměty aktivně nehledá, ale sbírá až v okamžiku, kdy je uvidí při nějaké jiné činnosti.

Ent si udržuje seznam předmětů (přesněji superseznam memberů popisujících předměty), které by rád našel. Zároveň si nastaví globální interrupt, který spustí sbírání předmětů až ve chvíli, kdy nějaké hledané vidí. Pro uživatele–programátora jsou k dispozici skripty, které předměty přidávají a ubírají ze seznamu hledaných. V podstatě v každé chvíli lze také zjistit, které předměty už ent sebral a které ještě na sebrání čekají.

Uplatnění v praktických skriptech najde jak verze nová, tak i původní. Některé věci ent potřebuje nutně a další činnost se bez nich neobejde (potřebuje-li si ent uvařit jídlo, neobejde se bez surovin k jeho výrobě). V jiných případech je však možné hledání dočasně přerušit, začít dělat něco jiného a mezitím se pokusit předměty najít. Například hudebníkův hlavní úkol je chodit po místnostech a hrát. Různé hudební nástroje si může navěsit do svého „on-line hledače“, v klidu pokračovat v hraní na původní nástroj a nový sebrat až v okamžiku, když se k němu dostane.

Obrázek 4.5: Scéna s vyhýbáním tří entů

Kapitola 5

Implementace a modelové situace

5.1 Hierarchické hledání cest

Programovací jazyk

Zlepšení efektivity hledání cest a výběr a implementace nových prohledávacích algoritmů je klíčovou částí této práce. Jak jsme již naznačili dříve, většina změn se odehrála ve zdrojových souborech jazyka Mercury. Mercury je deklarativní jazyk se silnou kontrolou syntaxe, typů a druhu determinismu (tzn. jestli a případně kolikrát může predikát uspět). To jazyku umožňuje odhalit mnoho chyb již při překladu a podstatně zvýšit rychlost výsledných programů v porovnání s jinými deklarativními jazyky. Jako assembler používá jazyk C, čímž jsou výsledné programy do jisté míry přenositelné.

Popsané výhody jako rychlost, silná kontrola a typová čistota jsou zvláště výrazné vůči jazyku E. Jazyk E má i jiná omezení: do skriptu například nelze předat více než patnáct proměnných, což se vzhledem k překladu každého `if`-větvení jako podskriptu ukázalo pro složitější konstrukce jako do značné míry omezující. Vše spolu faktem, že některé prohledávací funkce již byly v Mercury napsány, bylo kritériem při výběru Mercury. Teoreticky by bylo možné funkce psát i v jiných jazycích (přímo se nabízelo C), Mercury však poskytuje podobnou neprocedurální filozofii jako E – typickým příkladem je bezproblémová práce se seznamy a jejich převod do/z E. Přestože domovská stránka projektu Mercury [10] působí dosti amatérským a nekomerčním dojmem, jeho funkčnost a množství kontrol, které skutečně provádí, přesně odpovídá specifikaci.

Jazyk Mercury s sebou pochopitelně přinesl i problémy. Typickým příkladem je absence debuggeru a omezenost na výpisy na standardní výstup. Na domovské stránce je sice uvedeno, že autoři žádné ladící prostředky nepoužívají, v praxi bychom je ale, myslím, ocenili.

Další problémy vyplývaly z propojení s projektem ENTI. Nejvíce času zaplnila správná komunikace s entovou databází. Chyběl příkaz pro zápis, takže bylo potřeba se ponořit do zdrojových souborů nejen Mercury, ale i C++, ve kterým je entova databáze napsána. Ještě náročnější pak bylo přidání nových tzv. „smyslů“, které byly nutné pro reprezentaci grafu pater a uložení absolutních pozic místností – změny vyžadovaly i konfigurační soubory celého projektu ENTI.

Řešení

Všechny popsané problémy se podařilo vyřešit, a tak již mohla následovat vlastní implementace upraveného HPA*.

Přiřazení absolutních souřadnic místnostem zařídí funkce `cNastavPozice-Mistnosti/1` s jedním vstupním argumentem udávajícím dlaždici, která bude mít souřadnice $(0, 0, 0)$ (nejjednodušší je předat dlaždici, kde ent právě stojí). Tuto funkci je třeba zavolat na začátku entova života, zejména před prvním použitím hledacích funkcí. Projde všechny dostupné místnosti a souřadnice jejich severozápadních rohů uloží do entovy databáze ke smyslu `HSmistnost_abspoziceH` ve formátu [*handle–místnosti*, *handle–souřadnice–X*, *handle–souřadnice–Y*, *handle–souřadnice–Z*][1].

Za krátkou poznámku stojí, že v ukázkovém světě, který je dodáván spolu s projektem ENTI, nelze všechny místnosti díky jejich velikostem do krychlové sítě správně umístit. Dochází tak k jejich překryvům a/nebo zdem tlustším než 1 dlaždice. Není to však klíčové – souřadnice se používají vždy jen pro heuristiku tak, že k dlaždici se zjišťuje souřadnice, nikoliv naopak.

Vlastní nové hledání cesty, využívající variantu algoritmu A*, mají na starosti dvě funkce: `cNajdiCestuDvermi/8` a `cNajdiCestuDo/8`. Zásadní rozdíl je v tom, že druhá umožňuje hledat cestu nejdříve po patrech a teprve následně ji zjemnit na jednotlivé místnosti, první se dívá na schody jako na normální dveře (viz 4.1, str. 25).

Součástí práce nebylo pouze napsání funkcí nových, ale i nahrazení původních funkcí bez A* novými verzemi. Přemostěny tak byly staré funkce

[1] „handle–souřadnice“ znamená handle čísla, které se rovná souřadnici. Např. handle pro číslo 0 je totiž ve skutečnosti číslo 18120704.

`cDvereSmeremK/6` a `cOdhadVzdalenosti/4` funkcí `cNajdiCestuDvermi/8` a upraven byl také skript `jdiDoMistnosti/1` tak, aby ent nehledal vždy celou cestu znova, jakmile vejde do nové místnosti, ale aby si ji pamatoval a používal, dokud to jde.

Testy

Testování probíhalo na počítači s procesorem Intel Mobile Celeron 1.8 GHz a 512 MB operační paměti v prostředí Linux Fedora Core 5. Vždy byla provedena tři nezávislá měření téhož nastavení, z kterých se vypočítal aritmetický průměr.

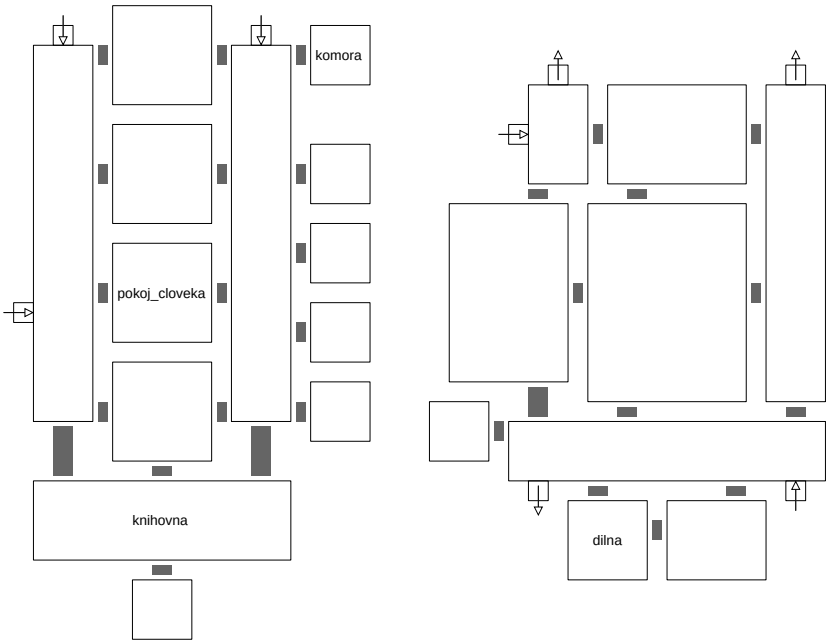
Jazyk E by měl umožňovat měření reálného času, který uběhne v daném skriptu, potřebné funkce (`pushtime`, `poptime`) však nejsou implementovány. Proběhlo tedy pouze měření výpočtových funkcí v Mercury. Aby byly časy vůbec měřitelné, jeden predikát proběhl vždy desetkrát těsně za sebou se stejnými parametry, časy jednoho běhu lze tedy jednoduše spočítat vydělením deseti. U každé trasy jsme měřili tři různé varianty – jako první starou funkci `cDvereSmeremK/6`, jako druhou a třetí novou `cNajdiCestuDo/8`, přičemž ve druhém případě se použil graf pater (tedy hierarchická varianta), ve třetím nikoliv.

Není zcela jasné, do jaké míry byl měřen skutečně jen čas, který byl stráven v daném predikátu, jestli měření ovlivňují i ostatní běžící procesy a jak se do měření promítla volání cizích funkcí, například hledání a zápisy do entovy databáze. Přesto hlavním výsledkem jsou hodnoty relativní vůči sobě. Ač jsou leckdy překvapivé, mají vypovídající hodnotu.

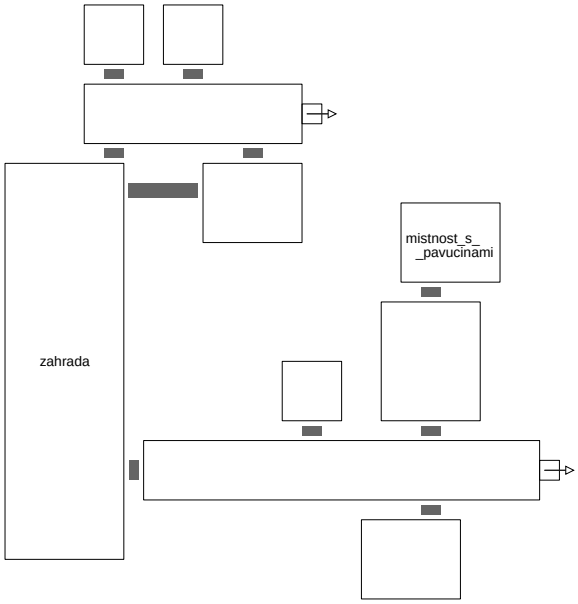
Než se však dostaneme k samotnému měření, uvědomme si, jaké výsledky můžeme očekávat. V části 2.2 (str. 11) bylo uvedeno, že HPA* může být až desetkrát rychlejší než obyčejné A* a může navštívit až desetkrát méně vrcholů. Nahlédnutím do článku [2] však zjistíme, že tyto výsledky autoři naměřili u cest délky cca 400. Ukázkový svět, na kterém probíhalo testování, takových rozměrů zdaleka nedosahuje. Další změnou je fakt, že vlastně porovnáváme algoritmus Dijkstrův s hierarchickým A*.

První fakt ukazuje spíše na menší zrychlení, druhý by měl naopak rozdíl zvětšovat. Dá se tedy očekávat, že zrychlení bude měřitelné, a to výrazněji s přibývající vzdáleností startovní a cílové pozice.

V ukázkovém světě (mapa viz obrázek 5.1 a 5.2) jsme k testování zvolili čtyři trasy. Všechny vedly z prvního patra, z toho první tři z pokoje člověka. První cesta směřovala na zahradu, což je přibližně napříč celým světem přes všechna tři patra, druhá cesta vedla do knihovny, na což stačilo pouze vyjít na chodbu



Obrázek 5.1: Mapa ukázkového světa: patro a přízemí



Obrázek 5.2: Mapa ukázkového světa: sklep

a vejít do cílové místnosti, třetí do místnosti s pavučinami, která je, stejně jako zahrada, ve sklepě (ent musel tedy projít přes dvoje schody). Poslední cesta vedla z komory do dílny v přízemí. Horizontálně jsou místnosti na opačných stranách domu, ale nejkratší cesta vede díky tomu, že schody nejsou přesně nad sebou, na první pohled oklikou.

V tabulce 5.1 jsou uvedeny výsledky měření. V prvním sloupci je varianta, jakou se hledalo, v dalších dvou délka nalezené cesty v dlaždicích a počet místností, kterými prochází. Následuje čas deseti běhů algoritmu v sekundách (průměr ze tří měření), počet navštívených vrcholů (vrchol odpovídá dlaždici přede dveřmi) a počet *různých* vrcholů, které algoritmus při svém běhu navštívil.

První hledání, z pokoje člověka na zahradu, dopadlo v zásadě podle očekávání. Vzhledem ke střední délce hledané cesty je již i přes nezanedbatelnou režii, se kterou je (hierarchický) A* spojen, rozdíl v čase proti Dijkstrovu algoritmu dobře patrný. Mezi hierarchickou a nehierarchickou verzí významný rozdíl není, což jsme však čekali – hierarchická verze se dle [2] začíná vyplácet až od cest délky cca 50. V počtu navštívených vrcholů je rozdíl vůči staré verzi algoritmu velmi markantní: hierarchická verze jich navštívila asi sedmdesátkrát méně, nehierarchická pak asi dvaadvacetkrát méně.

typ	poč. dlž.	poč. míst.	průměr. čas	cel. expnd.	růz. expnd.
pokoj_cloveka → zahrada					
cDvere	38	6	0.457	10626	84
cNajdi/2	36	6	0.280	150	15
cNajdi/1	38	6	0.277	480	41
pokoj_cloveka → knihovna					
cDvere	16	3	**0.030**	540	35
cNajdi/2	16	3	0.243	90	9
cNajdi/1	16	3	0.243	110	11
pokoj_cloveka → mistnost_s_pavucinami					
cDvere	43	6	0.623	14260	88
cNajdi/2	**76**	**9**	0.273	260	21
cNajdi/1	43	6	0.403	2120	70
komora → dilna					
cDvere	36	6	0.323	7480	86
cNajdi/2	35	6	**0.077**	320	25
cNajdi/1	36	6	**0.067**	310	31

Tabulka 5.1: Výsledky testů

Na druhém hledání bylo poměrně překvapivé, jak moc může algoritmus A* na krátkých vzdálenostech ztrácet v čase. Důvodem může být, že naše implementace v podstatě nebyla optimalizovaná, přesto je rozdíl značný. Počet navštívených vrcholů se ukazuje jako podstatně stabilnější ukazatel – A* navštívil zhruba pětinu; hierarchická a nehierarchická verze se zásadně neliší, neboť místnosti byly ve stejném patře.

Třetí příklad ukázal typickou vlastnost hierarchického přístupu. Je sice přibližně dvaapůlkrát rychlejší a navštívil asi jednu padesátinu vrcholů vůči Dijkstrovu algoritmu, díky „děravosti“ sklepního patra (a tudíž nepřesné heuristice) však nedokázal najít nejkratší cestu. Nehierarchická verze si za cenu poměrně velkého počtu navštívených vrcholů poradit dokázala a díky střední vzdálenosti dokázala Dijkstrův algoritmus porazit i v čase.

Na poslední trase potvrdil Dijkstrův algoritmus stabilitu svých výsledků a čas ani počet navštívených vrcholů nevybočují z vysledovaného chování. Lehce zvýšený počet navštívených vrcholů u algoritmů A* se dá vysvětlit nutností vypořádat se z faktem, že dlaždice, které sousedily s použitými dveřmi mezi prvním patrem a přízemím, jsou v horizontálním směru cca 6 dlaždic vzdálené. Vysvětlení, proč se najednou tak výrazným způsobem liší čas A* algoritmů, se nám ale nalézt nepodařilo.

Celkově lze říci, že ve většině případů ke zrychlení skutečně došlo. Nové algoritmy ztrácí v případě, že startovní a cílové pozice jsou blízko sebe. Vezmeme-li ale v úvahu, že nové skripty vrací cestu celou, nejenom první místnost, zrychlení je na delších cestách ve skutečnosti ještě výraznější a na krátkých se pravděpodobně výsledky vyrovnají, takže naše práce měla smysl.

5.2 Bagrování

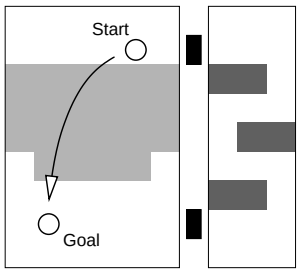
Aby enti dokázali nacházet a správně zvažovat náročnost cesty přes zaplněné dlaždice, bylo potřeba změnit predikát `cSmerKDlazdici/7`. Původní verze algoritmu A* ignorovala dlaždice, kam se ent nevešel. Další změna pak byla ve skriptu `jdiNaDlazdici/5`, který potřeboval upravit tak, aby dokázal z plné dlaždice nejdříve předměty posbírat.

Další podproblém, schopnost uvažovat i cesty okolními místnostmi, jsme nejdříve řešili pouze ve skriptu `jdiNaDlazdici/5`. Jak již bylo naznačeno v části 4.2 (str. 28), pokud je cesta přes aktuální místnost velmi dlouhá (delší než

dvojnásobek teoreticky nejkratší vzdálenosti), zkouší se postupně cesty přes všechny dveře i místnostmi sousedními.

Druhé řešení se nabídlo po zlepšení prohledávání cest mezi různými místnostmi pomocí A*. Použijeme-li na projití místnosti skript `jdiDoMistnosti/1`, prohledávací funkce `cNajdiCestuDo/8` bude zkoušet okolní místnosti automaticky.

Na přiloženém CD je video `geometrie.avi`, které demonstruje dosud popsaná zlepšení. Zelený ent je ent ovládaný počítačem. Modro-žlutý je ovládán uživatelem, jeho pohyb je nutný, aby bylo vidět chování strojového enta a nebudeme se jím dále zabývat. Úkol zeleného enta je dojít z dlaždice téměř v nejpravějším rohu (souřadnice $(2, 5)$, viz obrázek 5.3) do téměř nejlevějšího rohu $(8, 2)$ a poté zase zpět. Cestu má zahrazenou kyblíky a konvemi, a tak nastane případ, kdy je spočítaná cesta přes aktuální místnost velmi dlouhá. Zkusí tedy cestu přes okolní místnosti a zjistí, že existuje a mohla by být kratší. Na cílovou dlaždici tedy dojde přes sousední místnost. Při plánování cesty v sousední místnosti si zapamatuje vzdálenost mezi dveřmi, která je díky židlím poměrně velká, a tak když ent plánuje zpáteční cestu $(8, 2) \rightarrow (2, 5)$, raději sesbírá předměty v aktuální místnosti, protože už ví, jak daleko je to díky židlím přes místnost sousední.



Obrázek 5.3: Topologie místností v příkladu z videa na CD

5.3 Pronásledování

Základní kostra skriptu `jdiKEntovi/2` zůstala stejná, změnila se ale část, která se stará o dojití až k entovi v aktuální místnosti. Skript kombinuje plánovací přednosti jazyka E a na početní výkony volá funkce napsané v Mercury.

Skript navěsí interrupt, který sleduje pohyb oběti. Z její pozice v předchozím a aktuálním kole zjišťuje směr, kterým se oběť za poslední kolo hnula. Zjistí-li

ent, že oběť změnila směr, spustí celý skript znovu a přeplánuje cestu. Pokud ent ví, kam oběť směřuje, zjistí, jestli jí může nadběhnout (vzorec viz část 4.3, str. 29). V kladném případě spočítá její cestu a vyrazí na nejlepší políčko tak, aby jí co nejdříve potkal. Jestliže nadběhnout nemůže, zkusí zjistit, nelze-li dojít kratší cestou do místnosti, kam oběť míří. Najde dveře, které jsou nejblíž průsečíku její cesty s nejbližší stěnou. Spočítá dvě vzdálenosti: vzdálenost d_1 oběti od těchto dveří a vzdálenost d_2 průsečíku její cesty se stěnou od daných dveří. Pomocí podmínky

$$(5 - 2d_1) + (5 - 2d_2) > 0$$

zkusí odhadnout, mají-li dveře šanci na to, že do nich oběť skutečně míří. Jestliže je podmínka splněna, ent zjistí, může-li dojít do středu místnosti, kam míří tyto dveře, kratší cestou. Jestliže ano, zkusí nadběhnout takto. Jestliže ne, nezbývá, než se vydat přímo za obětí.

Dvě připravená videa ukazující obě popsané situace, kdy ent může oběti nadběhnout, najdete na CD disku. Zelený ent je strojový a snaží se dohnat modrožlutého enta ovládaného uživatelem. První video, `nadbihani_1.avi`, představuje případ, kdy ent může své oběti nadběhnout a využít tak své poziční výhody i když oběť mění směr. Dobře je patrné, že entovy reakce jsou o kolo zpožděny, neboť musí nejdříve zjistit, kam oběť míří. Na druhém videu, `nadbihani_2.avi`, je ukázka úspěšného odhadu, do kterých dveří oběť míří, a následné nadběhnutí kratší cestou.

5.4 Vyhýbání se ostatním entům

Abychom mohli sledovat pohyb ostatních entů v místnosti, navěsíme ve skriptu `jdiNaDlazdici/5` podobný interrupt jako v předchozím případě. Bude se spouštět ve chvíli, kdy existují enti, kteří jsou blíže než zadaný poloměr – na enty, kteří jsou dále, se nebere ohled. Z testů vyšla pro poloměr nejlépe hodnota 4, neboť je-li to méně, ent nestačí uhnout, je-li to více, zbytečně se zatěžuje procesor, navíc se pohyb sledovaných entů může před střetem ještě dosti změnit.

Pozice a směry takto sledovaných entů se předají funkci `cSmerKDlazdici/7`, která spočítá pravděpodobné cesty entů. Dlaždice, po kterých enti zřejmě půjdou, v příslušných kolech penalizuje. Při testování jsme zjistili, že penalizovat pouze dlaždici, na které ent bude v příslušném kole, nestačí. Je to tím, že není dáno pořadí, v jakém enti vykonají své atomické instrukce, a navíc ent při svém kroku často blokuje původní i cílovou dlaždici najednou. Je tedy potřeba omezit dlaždici před i za tou, kterou odhadnou výpočetní funkce.

Přiložené CD obsahuje video `vyhybani.avi`, které popsané zlepšení ukazuje názorněji. Dva strojoví enti jdou proti sobě, třetí, lidský, kolmo na jejich směr. V případě prvního vyhýbání je možné pozorovat jev, který známe i z chování lidí: enti se snadněji vyhýbají těm, kteří jdou kolmo. Jdou-li enti přímo proti sobě, může se stát, že se začnou vyhýbat na stejnou stranu. Následně ale občas raději kolo počkají, než vyrazí na již navštívené dlaždice, takže se poměrně snadno vzájemně obejdou.

5.5 On-line hledání

Entovy skripty byly rozšířeny o tři nové. První z nich nese název `hledejASbirejLazy_Pridej/2`. Vytvoří nebo rozšíří seznam předmětů, které ent bude „líným“ způsobem hledat (popsaným v kapitole 4.5 na str. 31). Jedinou možností, jak tento seznam (přesněji superseznam) reprezentovat, byly globální proměnné. Je potřeba, aby byl přístupný z více různých skriptů – entova databáze ani fakta použít nelze, protože do nich lze ukládat pouze typ handle, nikoliv seznam nebo superseznam. Druhá a třetí funkce `hledejASbirejLazy_Uber/1` a `hledejASbirejLazy_VsechnoZrus/0` jsou opakem funkce první, mají na starosti předměty ze seznamu ubírat.

Příklad použití můžeme nalézt v ukázce `on-line-hled.avi` na vloženém CD. Stejně jako v ostatních případech je žluto-modrý ent ovládán uživatelem a druhý, tentokrát hnědo-bílý, je ent strojový. Jeho úkolem je projít všechny místnosti a v každé krátce zahrát na všechny hudební nástroje, které v té chvíli má. Ještě před cestou do první místnosti, pokoje knihovníka, však přidá do svého on-line hledacího seznamu všechny tři druhy hudebních nástrojů, které ve světě entů existují: flétnu, harmoniku a housle.

V pokoji knihovníka uspěje podmínka globálního interruptu navěšeného skriptem `hledejASbirejLazy_Pridej/2` tím, že ent uvidí housle. Sebere je, zahraje na ně a vydá se do sousední chodby. Tam, ani v další místnosti, pokoji člověka, žádné nové nástroje nenajde, zahraje tedy pouze na housle. Až v pokoji hudebníka a kuchaře najde postupně flétnu a harmoniku, takže se jeho nástrojářský park rozšíří a v poslední místnosti si už zahraje na nástroje všechny tři.

Kapitola 6

Závěr

6.1 Budoucí práce

Obor umělé inteligence se neustále rozvíjí a s rostoucím výkonem počítačů se nabízí mnoho možností, jak chování umělých bytostí dále zlepšovat. Chceme-li zůstat u pohybových skriptů v projektu ENTI, další pokračování by se mohlo ubírat následujícími směry.

Optimalizace A* a HPA*. Z testů vyplynulo, že při hledání velmi krátkých cest nová implementace s heuristickými algoritmy zaostává nad původní verzí. Obecně platí, že čím složitější algoritmus, tím větší nutná režie – toto chování je tedy přirozené. Přesto by jistě bylo zajímavé se v budoucnu pokusit tento deficit odstranit.

Dlouhodobé sledování oběti. Při pronásledování entů se nové skripty snaží co nejdříve reagovat na nastalé změny v pohybu oběti a vždy hledat nejkratší cestu na nové místo střetu. Bez zajímavosti by ale jistě nebylo sledovat a odhadovat entovu dlouhodobější strategii a pokusit se ji využít, například k nadbíhání do sousedních místností. Implementovat by se daly i jiné techniky umělé inteligence jako tzv. reinforcement, tedy na základě úspěchu nebo neúspěchu posledního postupu upravovat příští strategie.

Snížení náročnosti vyhýbání. Pokud by se podařilo najít nějaký nový způsob v uchovávání většího množství dat, nejlépe v poli, hardwarová náročnost vyhýbání

by se pravděpodobně dala snížit aplikací některé z *incremental heuristic search methods*. Cesta přes místnost by se pak nemusela hledat při každé změně směru druhých entů celá znovu, ale aktualizovat by se mohla jen inkriminovaná část mapy.

Buzení on-line hledacím skriptem. Pokud by se do jazyka E doplnilo v dokumentaci uvedené usínání a probouzení skriptů, byla by možnost skript závislý na sebraných předmětech po navěšení hledaných předmětů uspat a po nalezení potřebného vybavení opět vzbudit. Tento postup by pak spojoval výhody obou nyní dostupných variant.

6.2 Další aplikace

Ač se úpravou a implementací nových prohledávacích algoritmů zabývá podstatná část této práce, jako obecné prostředí pro jejich testování se projekt ENTI, dle našeho názoru, příliš nehodí. Pathfinding v ENTECH je totiž dosti specifický – svět je pevně rozdělen na místnosti, enti se pohybují po poměrně velkých dlaždicích jen ve čtyřech směrech, je použit diskrétní čas, není vyřešené ukládání většího množství dat apod.

Kdybychom chtěli postupovat opačným směrem a použít námi upravenou verzi HPA* v jiných aplikacích, ve většině případů bychom asi také narazili na nemalé množství problémů. Jeden přínos, se kterým jsme se nesetkali v žádných jiných zdrojích, však tato část práce přinesla: použití hierarchické formy A* na *mapy s více patry*. V dnešní době těžko najdeme 3D hru, jejíž svět by nebyl víceúrovňový. I když přesně nevíme, jak řeší více pater například boti v Unrealu, věříme, že náš přístup je dobrým kandidátem na zlepšení.

Právě to, co je pro „nízkoúrovňové algoritmy“ nevýhodou, je vítaným zjednodušením v algoritmech úrovně vyšší. Pro plánování celých úloh a přepínání mezi nimi nabízí jazyk E praktické nástroje, které spolu s množstvím zjednodušení (například diskrétní čas i prostor) dávají možnost soustředit se na daný problém a nezabývat se implementačními detaily.

Praktický přínos práce v jiných programech a aplikacích tedy bude spíše v postupech spadajících do druhého cíle práce, tedy ve věrohodnějším chování agentů.

Příkladem může být pronásledování entů. Popsaná metoda, která dokáže jednoduchým způsobem učinit chování postav uvěřitelnějším, by dobře mohla najít uplatnění například ve strategiích nebo jiných simulacích využívajících dlaždicové mapy.

6.3 Shrnutí

Prvním cílem práce bylo zrychlení prohledávacích algoritmů. Abychom ho naplnili, provedli jsme průzkum současných algoritmů a přístupů, které rozvíjí základní prohledávací algoritmus A*. Práce přináší přehled nejvýznamnějších zástupců. Pro použití v projektu ENTI byl vybrán a dále upraven HPA*. Je to hierarchická úprava algoritmu A*, která ve své základní verzi abstrahuje pravidelné oblasti a staví z nich graf. Tento postup se může opakovat a lze tak vytvořit z abstrakcí hierarchii.

Z praktických testů plyne, že výsledný algoritmus skutečně zrychlení na středních a delších vzdálenostech přináší. Použití dvou vrstev abstraktních grafů výsledky ještě o málo zlepšuje, nezaručuje však nalezení nejkratší cesty. Objevil se (do jisté míry očekávaný) negativní jev – při hledání krátkých cest může být nový algoritmus pomalejší než původní. To potvrzuje teorii, že nové, složitější algoritmy mají větší nutnou režii, čímž se vyplácí až při rozsáhlejších problémech.

Druhý cíl, věrohodnější chování entů, se realizoval ve čtyřech základních zlepšeních. Uvolňování cesty zatarasené předměty, tzv. bagrování, funguje rychle a podle předpokladů. Při používání příbuzného vylepšení, hledání cesty okolními místnostmi, je potřeba opatrně používat jeho první verzi. Ve složitějších skriptech může způsobovat nepřirozené chování například tím, že se ent bude rychle vracet do míst, kde byl před chvílí. V druhém přístupu k obcházení, který používá upravený HPA*, jsme slabiny nenašli.

Pronásledování ostatních entů probíhá tak, jak jsme si představovali. V nejhorším případě se enti chovají stejně jako podle původních skriptů, pokud ale oběť utíká „správným“ směrem, zlepšení je zřetelně vidět.

Vyhýbání kolizím s ostatními enty je patrně nediskutabilnější částí práce. Pravdou je, že enti do sebe naráží jen ve výjimečných případech. Cena, která je za to zaplacena, je však poměrně vysoká: již při třech vyhýbajících se entech můžeme cítit zpomalení chodu celého systému. Množství interruptů, které je

potřeba dohromady navěsit, totiž roste s druhou mocninou počtu entů v místnosti. Interrupty v jazyce E, které se zprvu zdají jako praktické zjednodušení, se tak díky malé rychlosti zpracování stávají slabinou.

Poslední novou schopností entů je tzv. on-line hledání, neboli hledání předmětů během jiných činností. Zátěž procesoru je zvýšena o jeden krátký interrupt, a tak tato část funguje nejen správně, ale i rychle.

Z uvedeného plyne, že i přes jistá negativa, způsobená převážně implementačními detaily, byly stanovené cíle splněny a projekt ENTI tak byl v uvedených bodech skutečně rozšířen a vylepšen.

Literatura

[1] Bojar, O., Brom, C. et al.: *Dokumentace studentského softwarového projektu ENTI*. MFF UK, Praha, 2002.

[2] Botea, A., Müller, M., Schaeffer, J.: *Near Optimal Hierarchical Path-Finding*. Journal of Game Development, 1:7–28, 2004.

[3] Čapek, K.: *R. U. R., Rossum's Universal Robots*. Artur, Praha, 2004.

[4] Dijkstra, E. W.: *A note on two problems in connection with graphs*. Numerische Mathematik 1, 269–271, 1959.

[5] Hart, P., Nilsson, N., Raphael, B.: *A formal basis for the heuristic determination of minimum cost paths*. IEEE Transactions on Systems Science and Cybernetics, Vol. 4, Issue 2, p. 100–107, 1968.

[6] Henderson, F. et al.: *The Mercury Language Reference Manual*. The University of Melbourne, Melbourne, 2002.

[7] Higgins, D. F.: *Pathfinding Design Architecture, How to Achieve Lightning-Fast A*, AI Game Programming Wisdom*. Charles River Media, Inc., 2002.

[8] Jansen, M. R., Buro, M.: *HPA* Enhancements*. University of Alberta, Alberta, 2007.

[9] Koenig, S., Likhachev, M.: *D* Lite*. American Association for AI, Atlanta, Pittsburgh, 2002.

[10] *The Mercury Project* – homepage [online]. [cit. 30.7.2007]
Dostupný z WWW: <`http://www.cs.mu.oz.au/research/mercury/`>

[11] *The Mercury Library Reference Manual*. The University of Melbourne, Melbourne, 2002.

[12] Reynolds, C. W.: *Steering Behaviors For Autonomous Characters* [online]. Sony Computer Entertainment America, California, 1999. [cit. 30.7.2007]

Dostupný z WWW: <http://www.red3d.com/cwr/steer/>

[13] Skalský, V.: *Wolfgang Kempelen: Bratislavský faust* [online].

[cit. 31.7.2007] Dostupný z WWW:
<http://www.czsk.net/svet/clanky/osobnosti/kempelen.html>

[14] Sturtevant, N., Buro, M.: *Partial Pathfinding Using Map Abstraction and Refinement*. The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, 2005.

[15] Sturtevant, N. R.: *Memory-Efficient Abstractions for Pathfinding*. American Association for AI (www.aaai.org), Alberta, 2007.

Příloha: Obsah CD

Součástí práce je přiložený CD-ROM, který obsahuje

- videa, která ukazují jednotlivá zlepšení chování entů (popis v textu)
- stručnou instalační, uživatelskou a programátorskou dokumentaci
- instalaci projektu ENTI
- zdrojové soubory, kterými je potřeba nahradit některé soubory původní
- testovací světy a skripty, které demonstrují popsaná vylepšení
- tuto práci ve formátu PDF