VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PLÁNOVÁNÍ POHYBU OBJEKTU V 3D PROSTORU

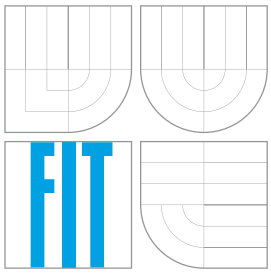
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

RADIM KRČMÁŘ

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PLÁNOVÁNÍ POHYBU OBJEKTU V 3D PROSTORU

PATH PLANNING IN 3D SPACE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

RADIM KRČMÁŘ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAROSLAV ROZMAN

BRNO 2010

Abstrakt

V následující práci jsou předvedeny základy plánování v prostoru se zaměřením na pravděpodobnostní plánování. Oborem úzce spjatým je detekce kolizí, s užitím lineární algebry je vytvořen systém pro kolize objektů v libovolném počtu rozměrů. Jsou popsány základní možnosti vizualizace trojrozměrných dat. Vybrané algoritmy byly implementovány v haskellu a užity k vytažení ježka z klece.

Abstract

Following thesis presents basics of motion planning, focusing on sampling-based algorithms. System for collision of objects in arbitrary dimensional space is created using linear algebra. Basic options for visualizaton of three-dimensional data are described. Selected algoritms were implemented in haskell and used to pull hedgehog out of the cage(popular disentanglement puzzle in Czech Republic).

Klíčová slova

haskell, plánování pohybu, pravděpodobnostní algoritmy, vizualizace dat ve 3D, detekce kolizí, lineární algebra

Keywords

haskell, motion planning, sampling-based algorithms, data visualization in 3D, collision detection, linear algebra

Citace

Radim Krčmář: Plánování pohybu objektu v 3D prostoru, bakalářská práce, Brno, FIT VUT v Brně, 2010

Plánování pohybu objektu v 3D prostoru

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jaroslava Rozmana

. . . . . .
Radim Krčmář
18. května 2010

Obsah

Kapitola 1

Úvod

Plánovací algoritmy vykonávají dříve lidskou práci v řadě oborů, od jednoduchých, jako balení potravin,[1] kde dosahují vyšší efektivity, až po skládání proteinů,[2] na které lidská představivost nestačí.

V kapitole 2 si představíme základní rozdělení plánovacích algoritmů a pozornost věnujeme pravděpodobnostním algoritmům, které jsou dostatečně obecné, aby zvládly všechny výše uvedené příklady. Pravděpodobnostní plánování využívá detekcí kolizí, která je popsána pro základní trojrozměrné objekty v kapitole 3. Jejich sílu ověříme na problému vytažení ježka z klece v kapitole 5, u kterého doufám, že jej lidé nepřenechají strojům, až bude se zdát příliš těžký. Výsledek plánování bude zobrazen pomocí trojrozměrné grafiky, dle kapitoly 4.

[1] Ukázka párkárny http://www.vincentabry.com/en/abb-robotics-sausage-robot-489.
[2] Folding@home http://folding.stanford.edu/.

Kapitola 2

Plánování

Plánováním myslíme řešení problému přesunu robota z jedné konfigurace do druhé, bez lidského přispění v průběhu.

2.1 Základní pojmy

Robot

Robotem budeme označovat objekt, pro který budeme chtít najít cestu.

Pracovní prostor

n rozměrný euklidovský prostor, ve kterém se robot vyskytuje, pro praktické problémy nabývá obvykle dvou, či tří rozměrů.

Stupně volnosti

Skalární veličina popisující počet rozměrů, kterých nabývá konfigurační prostor daného robota. Robot má vždy alespoň $\frac{1}{2}d(d+1)$ stupňů volnosti, kde d určuje počet rozměrů pracovního prostoru. Další stupně volnosti získává robot klouby, či deformačními schopnostmi.

Konfigurace a konfigurační prostor

Konfigurace je uspořádaná n-tice hodnot, kde n je počet stupňů volnosti. Konfigurační prostor je poté množina všech konfigurací, budeme jej značit C.

Volný konfigurační prostor

Volný konfigurační prostor, dále C_{free}, je podmnožinou konfiguračního prostoru. Každá konfigurace navíc nekoliduje s žádným jiným objektem.

$$C_{free} = \{k \in C | robot(k) \cap World = \varnothing\}$$

Holonomicita

Robota nazýváme holonomickým, pokud má všechny stupně volnosti ovladatelné.

Plán

Uspořádaný seznam volných konfigurací, ve kterém jsou sousedé dosažitelní.

$$plan(A, B) = A \to k_0 \to k_1 \to \cdots \to k_n \to B, kde\{A, B, k_i\} \in C_{free}$$

Graf

Množina vrcholů a hran mezi nimi.

$$G = (V, E), V = \{v_0, v_1, \cdots, v_n\}, E = \{(v_i, v_j) | v_i, v_j \in V\}$$

Strom

Graf s právě jednou cestou mezi libovolnými dvěma uzly.

Mapa

Množina vzájemně disjunktních spojených grafů. U disjunktních grafů očekáváme, že pokrývají části konfiguračního prostoru, které jsou neprostupně odděleny překážkami.

$$M = \{g | g_i, g_j \in M^2, g_i \neq g_j \implies g_i \cap g_j = \varnothing\}$$

2.2 Rozdělení plánovacích algoritmů

Budou uvedeny základní postupy na vyhledávání cesty mezi dvěma body v prostoru,

2.2.1 Vzorkující algoritmu

Nepracují se spojitým prostorem, ale diskretizují jej, základní rozdíl je ve způsobu diskretizace. Uvedeme si postupně se zobecňující algoritmy.

Procházení mřížkou

Konfigurační prostor je vzorkován do pravidelné mřížky, po které se můžeme pohybovat na sousední vrcholy. Jedná se o jednoduchý model, který není příliš vhodný pro aplikaci, jeho výkon velmi záleží na hustotě mřížky, která se však v průběhu nemění – ve složitějších problémech tak je buď příliš hrubá, takže robot nenajde cestu, nebo velmi jemná a výpočet trvá věky.

Potenciálová pole

Pozměnění algoritmu procházení mřížkou. Pracují s konfiguračním prostorem rozšířeným o jednu dimenzi, která vyjadřuje potenciál přitahující robota do cílové pozice, síla v každém bodě je obvykle součtem transformovaných vzdálenosti od cíle a od překážek. Pohyb poté ze startovního bodu probíhá s cílem zvětšovat potenciál, dokud není dosaženo cíle. Uvázne-li robot v lokálním minimu, je cesta prohlášena za neexistující.

Vyjádřeno jinou abstrakcí jsou potenciálová pole instancí algoritmu A* nad pravidelnou mřížkou s netriviální heuristickou funkcí a výběrem pouze jediného (nejlepšího) souseda k další expanzi.

Pravděpodobnostní algoritmy

Mřížka může být nepravidelná; uzly jsou generovány náhodně, což vede k pokrytí celého, i spojitého, konfiguračního prostoru v limitě [9]; a postupně, což dovoluje algoritmu reagovat na změny rozložení v případě vysokého počtu kolizí.

V současnosti se jedná o nejpoužívanější přístup k řešení problémů s vysokým počtem rozměrů konfiguračního prostoru. Nevýhodou může být pouze pravděpodobnostní úplnost. V případě neexistence řešení nemůže určit její neexistenci, jinak ale vždy najde cestu. Předchozí uvedené vzorkující algoritmy však nebyly úplné, takže v porovnání s nimi jde o značný pokrok.

2.2.2 Kombinatorické algoritmy

Narozdíl od vzorkujících algoritmů jsou úplné i nad spojitým prostorem. Také se neopírají o detekci kolizí, z čehož pramení značná závislost na reprezentaci světa. Pro svou složitost jsou vhodné pouze na problémy s nízkým počtem rozměrů konfiguračního prostoru, však pro takové problémy poskytují přesné řešení.

Rozklad na buňky

Rozdělují prostor na buňky. Rozdělení by mělo splňovat tři podmínky

- Cesta z každého místa jedné buňky do sousední musí existovat.
- Sousední buňky musí být zjistitelné, aby se z nich dala poskládat mapa.
- Musí být možno zjistit, které buňce náleží počátek a konec plánované cesty.

Ve dvou dimenzích můžeme problém převést na triangulaci složitého polygonu, užívajíce středy hran k vytvoření mapy, ovšem robot musí v tomto případě být bodový. Dostatečně obecný plánovač, schopný vyřešit komplikovaný problém ve třech rozměrech budeme hledat marně.

Geometrické výpočetní algoritmy

Jsou schopny vyřešit i PSPACE-těžké problémy, jakým je například problém stěhování klavíru, platí za to značnou teoretickou i implementační obtížností a vzhledem k našemu zaměření na pravděpodobnostní algoritmy budou prezentovány pouze jako odkaz do literatury [4, 9].

2.3 Princip pravděpodobnostního plánování

Literatura [4, 9] rozděluje plánovací algoritmy na jednodotazové a vícedotazové, kde klíčem je vytváření mapy;

- jednodotazové vytvářejí cestu mezi dvěma body. Přidáváním náhodných uzlů do stromu ukotveného v počátečním bodě se snaží dojít do situace, kdy přidaný uzel lze napojit na koncový bod.
- vícedotazové vytvoří z množiny náhodných uzlů graf, který je potom použit jako základ pro vyhledávání a již se do něj pouze zapojují koncové body hledané cesty.

i jednodotazové algoritmy lze interpretovat, jako vícedotazové. Liší se pouze v tom, že graf je vždy jeden strom; dále proto již nebudou rozlišovány a i jednodotazové budeme brát, jako vytvářející mapu.

Vytvoření mapy

Mapu obecně získáváme spojením grafů rozšířených.

- Přidání grafů složených z jednoho vrcholu k množině již existující grafů.
- Rozšiřování grafů přidá do grafu nový vrchol a hranu od existujícího vrcholu grafu k novému vrcholu, pokud by při přesunu po hraně nedošlo ke kolizi.
- Spojování grafů je, ze zkušeností, časově nejnáročnější fází algoritmu, kdy se graf x snaží spojit v graf se všemi grafy, které jsou z grafu x, kterému náleží dostupné. Vznikne množina spojených grafů, kdy žádný z grafu se nemůže spojit s žádným jiným, tedy mapa.

Napojení počátku a konce

Počáteční a koncový bod musí být napojen do stejného grafu náležícího mapě. Speciální případy, jaké byly dříve označovány jako jednodotazové, kupříkladu vytvářely mapu ze dvou stromů ukotvených v počátku a konci a v tomto kroku pouze ověříme, že mapa je nyní složena jen z jednoho stromu, což znamená úspěšné spojení, jinak můžeme s jistotou prohlásit, že cesta nebyla nalezena.

Vyhledání cesty

V této fázi již máme vybrán graf, kterému náleží uzly, mezi nimiž hledáme cestu; nejjednodušším způsobem jejícho nalezení je použít některý z existujících grafových algoritmů, například A*.

2.4 Algoritmy

2.4.1 Vytvořející grafu

PRobabilistic roadMaps

Základním algoritmem pro pravděpodobnostní plánování. Z konfiguračního prostoru vybere náhodnou množinu bodů, které se následně pokusí spojit do mapy. Existuje řada rozšíření, takže se ve své původní podobě používá snad pouze k uvedení pravděpodobnostních algoritmů. Jedno z nich bude následovat a jelikož se neliší o mnhoho, bude pseudokód uveden u něj.

Sampling-based Roadmap of Trees

SRT je flexibilním algoritmem, který vzniká přidáním fáze expanze o určitý počet vrcholů k algoritmu PRM a tím, že se může expandovat pouze jeden strom se chová i jako samotný algoritmus rozšiřování stromů.

`vstup:`
nT – počet grafů

nE – počet rozšíření každého grafu

`výstup:`

R – roadmapa = množina nejvýše nT grafů

`algoritmus:`

$R \leftarrow \varnothing$

$T \leftarrow nT$ **krát udělej:**

$q \leftarrow$ nahodná pozice v C_{free}

$t \leftarrow$ strom ukotvený v q rozšířený až o nE uzlů

pro každý t **z** T **udělej:**

$U \leftarrow$ grafy z R, na které lze napojit t

$R \leftarrow R \backslash U \cup \left(t \cup \bigcup_{u \in U} u\right)$

2.4.2 Vytvářející strom

Rapidly-exploring Dense Tree

Je pravděpodobnostní stromovou strukturou, v limitě vyplňující celý prostor.

Kořen stromu je rozšiřován o náhodně vybraný nový uzel a hranu mezi nejbližším uzlem stromu a novým uzlem, nenastane-li kolize v novém uzlu a při přesusunu po hraně.[1]

`vstup:`

q_0 – počáteční bod

N – počet pokusů o rozšíření

`výstup:`

T – strom o nejvýše N uzlech

`algoritmus:`

$T \leftarrow$ nový strom obsahující pouze q_0

N **krát udělej:**

$q_{rand} \leftarrow$ náhodná pozice v C_{free}

$q_{closest} \leftarrow$ nejbližší uzel ze stromu T k uzlu $rand$

pokud je bez kolize spojení $q_{closest}$ a q_{rand}**:**

rozšiř T o hranu $(q_{closest}, q_{rand})$ a uzel q_{rand}

Nová konfigurace by měla být v kinodynamických a neholonomických úlohách ještě přizpůsobena, aby splňovala podmínky dané systémem.[2]

EST

Od RRT se základně liší výběrem prvku ze stromu, na který napojíme nový uzel. U EST nejdříve náhodně vybereme uzel stromu, který budeme rozšiřovat a poté až nový uzel. Jelikož RRT si samo vybírá nejbližší uzel z grafu, může být vzdálenost mezi vybraným a novým uzlem u EST značná vzdálenost, neprospívající pravděpodobnosti nalezení bezkolizní spojnice a proto je vždy konfigurační prostor u výběru nového prvku omezen na okolí rozšiřovaného prvku.

[1] Hrana je z výkonostních hledisek diskretizována a krok nastaven, aby stále rozumně aproximovalo spojité posouvání.

[2] Letadlo zrychluje pouze kolmo k ose trysky. Konfigurace například může být upravena na posunutí prostorem o konstantu a maximální otočení směrem k původní konfiguraci.

2.4.3 Vyhledání cesty

A*

Rozšířením Dijkistrova algoritmu [5] o heuristickou funkci získáme A*.

Jedná se o informovaný algoritmus, preferující nejslibnější cesty – ohodnocení je provedeno jako součet již uražené cesty od počátku a heuristický odhad vzdálenosti do cíle.

Na heuristiku klademe obvykle dvě očekávání. Měla by být optimistická, tedy pro každý vrchol neříci, že cíl je dále, než je délka nejkratší cesty do něj – bude-li toto splněno, pak alogoritmus A* vrací optimální cestu grafem; a silnější podmínku monotonosti, kdy současná cesta z počátku do daného vrcholu sečtená s odhadem cesty do cíle nesmí překročit skutečnou délku cesty od počátku do konce – poté A* nemůže navštívit žádný vrchol podruhé lepší cestou, je tedy možné pamatovat si i prošlé uzly a při dalších navštíveních je znovu již nerozšiřovat.

Pseudokód by byl značně neelegantní a bude uveden pouze u zjednodušení algoritmu.[3]

BFS

Nastavením heuristické funkce A*, aby vracela vždy nulu[4] a nastavením vzdálenosti mezi uzly na konstantu, získáváme algoritmu BFS, který lze zapsat jednodušeji.[5]

`vstup:`
    A, B – uzly, mezi kterými chceme najít cestu
    G – graf
`výstup:`
    cesta z A do B, existuje-li

`algoritmus:`
    open ← `[A]`
    dokud *open* není prázné**:**
        active ← vyjmi první z *open*
        pokud *active* $= B$**:**
            skonči a vrať cestu z A do B
        jinak:
            zařaď všechny uzly v G sousedící s *active* do *open*
    open je prázdné $\implies$ cesta neexistuje

[3] V případě zájmu je, jakožto i všechny ostatní informace, dostupný na wikipedii [16].
[4] Tato heuristika je monotóní a tedy i optimistická.
[5] Předpokládáme-li, že uzly určené k rozšíření uchovaváme ve frontě.

Kapitola 3

Detekce kolizí

U plánování jsme potřeboali omezit konfigurační prostor o kolidující konfigurace, čehož bylo dosaženo kontrolou průniku bodů, ze kterých byl robot složen s body tvořícími okolní svět.

Kapitola také odhalila nepříjemné mezery mého vzdělání, když po třech letech na vysoké škole jsem neznal význam skalárního a vektorového součinu. Nemluvě o schopnosti odvodit jednoduché vzorce, které budou následovat.

3.1 Pracovní prostor

Současné pochopení pozorovatelného světa, ač nepovažuje se za úplné, je mnohem komplikovanější, než budeme potřebovat; zavedeme tedy abstrakci nazvanou diskrétní prostor.

3.1.1 Diskrétní prostor

Postaven je z buněk v n dimenzích, které mohou být vyplněny objekty;[1] o každé buňce můžeme říci, jaký objekt ji vyplňuje a známe její sousední buňky, což dále zjednodušíme spočetnou pravidelnou mřížkou.

Detekce kolizí v takovémto prostoru je algoritmicky jednoduchá, stačí najít buňku s dvěma a více objekty, třeba postupným procházením celého (potenciálně nekonečného) prostoru; dnešní výpočetní architektury[2] však problémy tohoto typu by pro přesné výsledky,[3] a z nich vyplývající extrémní počty buněk i pro jednoduché operace, v makrosvětě nemusely skončit ve stejném universu.

Zavedením hierarchické struktury by se dalo dosáhnout výrazného zrychlení samotné detekce, ovšem pří požadavku na měnitelný svět by udržování hierarchie přineslo podobné čekací doby.

3.1.2 Algebraický prostor

Nevýhody předchozího modelu nás vedou k další abstrakci, kterou se můžeme dostat k algebraickému prostoru, jež je složen z množiny objektů popsaných algebraicky, tedy obvykle rovnicemi a nerovnicemi, které vyjadřují nekonečnou množinu bodů, ze kterých je objekt

[1] V původním světě může být v jedné buňce pouze jeden objekt, však tímto bychom nikdy nemohli počítač kolize.

[2] Pravděpodobně také všechny uskutečnitelné počítačové architektury.

[3] Už od úrovně mikrometrů.

složen. Výhodou oproti předchozímu popisu je možnost jednoduché změny, která se dotkne pouze jednoho objektu.

Pro zjištění kolize je třeba vzít rovnice všech objektů a vzájemně zjistit, mají-li řešení, v tom případě kolidují. Měl-li by počítač symbolicky řešit soustavy (ne)rovnic, také bychom čekali relativně dlouho; vytvářejí se tak obvykle parametrizované objekty, které mezi sebou mají ručně vytvořené kolizní funkce.

V praxi se nad algebraickým popisem používá velmi hrubý hierarchický diskrétní model, který omezí objekty, které se mezi sebou ověřují [6].

3.2 Kolize ve vyšších rozměrech

Máme-li dva n rozměrné objekty v m rozměrném prostoru, kde $m > n$, pro které umíme určít, zdali kolidují s objektem v prostorech o rozměru n a nižších, a chceme zjistit, zdali kolidují i v prostoru o rozměru m, využijeme vlastnosti průniku nekonečných n rozměných objektů, které jsou rozšířením objektů, v m rozmerech. Rozdělme jejich základní polohy do dvou kategorií rovnoběžné a nerovnoběžné.

- rovnoběžné lze dále rozlišit na dva případy – jsou shodné a tedy tvoří n rozměrný prostor, či shodné nejsou a poté se nikdy neprotnou, čímž vytváří bezrozměrný prostor.

  V bezrozměrném prostoru kolize nenastává, však v n rozměrném prostoru může a jelikož umíme určit průnik n rozměrných objektů v n rozměrném prostoru, učiníme tak.

- nerovnoběžné vytvářejí $n-1$ rozměrný prostor, který využijeme pod pojemnováním C – provedeme průnik našich n rozměrných objektů s C, čímž získáme bezrozměrné, či $n-1$ rozměrné objekty v C, nejsou-li oba bezrozměrné, mohla nastat kolize, kterou určíme, jako průnik $n-1$ rozměrných objektů.

Abychom z tohoto mohli těžit, je třeba si definovat kolize základních n rozměrných prostorů. Dále budeme pracovat nejvýše ve třech rozměrech a tedy bude stačit definovat prostory pro $n < 3$.

3.3 Výpočet vzdálenosti v algebraickém prostoru

3.3.1 Primitiva

Základní n rozměrné objekty, odvozovány budou funkce

- $d(x, y) = z$ vyjadřující vzdálenost mezi objektem x a objektem y
- $c(x, y) = (px, py)$ pro nejbližší dva body
- $i(x, y) = z$ pro objekt, který vytvářejí svým průnikem.

pro základní trojrozměrné podprostory

- Bod – *Point* p, kde p je pozice bodu v prostoru
- Přímka – *Line* p v, kde p je bod náležící přímce a v směrový vektor
- Plocha – *Plane* p n, kde p je bod náležící ploše a n normálový vektor

Bod—Bod

$$d(Point\ P,\ Point\ Q) = \|P - Q\| \tag{3.1}$$

Bod—Přímka

mějme bod Q a přímku zadanou bodem P_0 a vektorem p.

Přímku vyjádříme parametricky

$$P(t) = P_0 + t * p \tag{3.2}$$

nejbližší bod $P(t_p)$ bude na kolmici k přímce, procházející bodem Q; využijeme skalárního součinu k pravoúhlé projekci bodu na přímku.[4]

$$\frac{p \cdot (P_0 - Q)}{\|p\|} = \|P_0 - Q\| \cos\alpha = t_p \tag{3.3}$$

$$d(Point\ Q,\ Line\ P_0\ p) = \|P(t_p) - Q\| \tag{3.4}$$

Bod—Rovina

bod Q a rovina s normálovým vektorem n a bodem P, na ní ležícím.

Stejnou projekcí, která v minulém případě vrátila parametr nejbližšího bodu, dostaneme nyní přímo vzdálenost, kterou ještě musíme zbavit znaménka, jež značí, na které straně plochy se bod nachází [7]

$$d(Point\ q,\ Plane\ n\ p) = \frac{|n \cdot (p - q)|}{\|n\|} \tag{3.5}$$

Nejbližší bod na ploše je ve vzdálenosti od bodu q směrem po kolmici k ploše. Skalární součin vrací vzdálenost se znaménkem vzledem k normále a proto nám bude fungovat v obou poloprostorech.

$$c(Point\ q,\ Plane\ n\ p) = \left(q,\ q - n\frac{n \cdot (p - q)}{\|n\|}\right) \tag{3.6}$$

Přímka—Přímka

mějme dvě přímky zadané bodem a vektorem P_0, p a Q_0, q přímky vyjádříme jako

$$P(t) = P_0 + t * p \tag{3.7}$$
$$Q(t) = Q_0 + t * q \tag{3.8}$$

A budeme chtít najít parametry t_p, t_q, ve kterých jsou si body $P(t_p)$ a $Q(t_q)$ nejblíže. O nejbližších bodech víme, že jejich spojnice

$$\begin{aligned} r &= P(t_p) - Q(t_q) \\ &= P_0 + t_p * p - Q_0 - t_q * q \\ &= r_0 + t_p * p - t_q * q \end{aligned} \tag{3.9}$$

[4] http://en.wikipedia.org/wiki/Scalar_resolute

bude v n-rozměřném prostoru protínat obě přímky v pravém úhlu.

$$p \cdot r = 0 \qquad q \cdot r = 0 \tag{3.10}$$

$$p \cdot r_0 + (p \cdot p)t_p - (p \cdot q)t_q = 0 \qquad q \cdot r_0 + (q \cdot p)t_p - (q \cdot q)t_q = 0 \tag{3.11}$$

a osamostatněním t_p a t_q při zavedení substituce $a = p \cdot p, b = p \cdot q, c = q \cdot q, d = p \cdot r_0, e = q \cdot r_0$

$$t_p = \frac{be - cd}{ac - b^2} \qquad t_q = \frac{ae - bd}{ac - b^2} \tag{3.12}$$

jsme získali parametry nejbližších bodů [15, 14].

$$c(Line\ P_0\ p,\ Line\ Q_0\ q) = (Point\ P(t_p),\ Point\ Q(t_q)) \tag{3.13}$$

Nyní máme ještě možnost bez další práce vyjádřit nejbližší bod dvou úseček, či úsečky a přímky – vyjádříme-li si je jako počáteční bod P_0 a vektor p, pro který $P_0 + p$ je koncovým bodem úsečky, pak t_p vypočtené v přechozích krocích lze po oříznutí intervalem $[0; 1]$ použít pro výpočet nejbližšího bodu stejným způsobem, jako pro přímku [12].

Přímka—Rovina

Omezíme-li se do tří dimenzí, má přímka dvě možné pozice vzhledem k ploše: je s ní rovnoběžná a poté se má buď nekonečno, nebo žádný průsečík a každý bod přímky je v obou případech nejbližším k ploše, či je různoběžná, z čehož vyplývá jeden průsečík, který je zároveň nejbližším bodem.

V případě rovnoběžnosti se vzdálenost redukuje na, již představenou, vzdálenost bodu a roviny. U nerovnoběžné je však vzdálenost vždy nulová, ale jak jsme si dříve řekli, budeme z primitiv později skládat složitější tvary, pro které bude potřeba určit vzdálenosti a proto budeme hledat pozici průsečíku, která se bude později hodit.

Rovina je dána bodem P_0 a normálovým vektorem n, přímka bodem Q_0 a vektorem q. Pro všechny body P roviny platí

$$n \cdot (P - P_0) = 0 \tag{3.14}$$

a hledáme takový bod P, který by zároveň splňoval i rovnici přímky

$$Q(t) = Q_0 + t * q = P \tag{3.15}$$

zpětným dosazením a následnou úpravou

$$\begin{aligned} n \cdot (Q_0 + t * q - P_0) &= 0 \\ n \cdot (Q_0 - P_0) + (n \cdot q)t &= 0 \\ t &= \frac{n \cdot (P_0 - Q_0)}{n \cdot q} \end{aligned} \tag{3.16}$$

získáme parametr průsečíku.

$$i(Line\ Q_0\ q,\ Plane\ P_0\ p) = Point\ Q(t),\ \text{pokud}\ p \nparallel q \tag{3.17}$$

Rovina—Rovina

Dvě roviny mají stejné základní pozice, jako přímka a rovina a také vzdálenost mezi rovnoběžnými rovinami převedeme na vzdálenost bodu od roviny. V případě různoběžnosti je však průsečíkem přímka.

$$L(t) = L_0 + t * l \tag{3.18}$$

Opět využijeme tří dimenzí a směrnici přímky určíme jako vektor pravoúhlý k oběma normálám ploch.

$$l = q \times p \tag{3.19}$$

zbývá určit nějaký bod náležící přímce, využijeme již odvozeného vztahu průsečíku přímky a roviny [12].

$$L_0 = i(Line\ P_0\ (l \times p),\ Plane\ Q_0\ q) \tag{3.20}$$

$$i(Plane\ P_0\ p,\ Plane\ Q_0\ q) = Line\ L_0\ l,\ \text{pokud}\ p \nparallel q \tag{3.21}$$

3.3.2 Objekty v prostoru

Ve jednom rozměru je jediným konvexním objektem úsečka, jejíž definici jsme mimochodem odvodili již při definici přímky. Ve dvou rozměrech je situace zajímavější a můžeme vytvářet složité útvary, o všech se však dá říci, že jsou složeny z trojúhelníků [3].[5] Všechny ostatní objekty, které zde nebudou představeny je možno dohledat v literatuře, pomocí maticového kolizního přehledu [1].

Trojúhelník

Trojúhelník—Bod Vybereme body p_1, p_2, p_3, jeden z každé strany trojúhelníka a z nich utvoříme tři lineárně nezávislé normály n_1, n_2, n_3 k stranám trojúhelníka, definující poloroviny, jejichž "kladná" část bude mimo trojúhelník. Jsou-li všechny skalární součiny $n_i \cdot (p_i - q)$ záporné, je bod uvnitř trojúhelníku a vzdálenost tedy nulová, jinak minimum jejich absolutních hodnot nám dá vzdálenost bodu k trojúhelníku.

Trojúhelník—Přímka Leží-li kterýkoliv bod v jiné polorovině, rozdělené přímkou, prochází přímka trojúhelníkem.

Trojúhelník—Trojúhelník Stejnou operací se pokusíme převést problém na Úsečka-Úsečka, což se však v případě trojúhelníku ležících na stejné ploše nemusí podařit.

Poté se trojúhelníky protínají pouze v případě, že alespoň jeden bod některého trojúhelníka leží uvnitř druhého.

Kruh

S kruhem je zjištění vzdálenosti velmi podobné, jako s bodem, pouze odečteme průměr kruhu a byla-li by výsledná vzdálenost menší nule, nastavíme ji na nulu.

$$d(Kruh\ p\ r, x) = max(0,\ d(Point\ p,\ x) - r) \tag{3.22}$$

[5] Uvedený algoritmu se však nepoužívá, jelikož i přes svou lineární časovou složitost je rychlejší implementovat a problém vyřešit s nadlineárním algoritmem.

Kruh—Kruh

$$d(Kruh\ p\ r,\ Kruh\ q\ s) = max(0,\ d(Point\ p,\ Point\ q) - (r + s)) \tag{3.23}$$

3.3.3 Skládání primitiv

Jelikož sklánáním lze utvořit nekonečně objektů, uvedeme pouze jeden pro ukázání principu.

Válec

Jedná se trojrozměrné těleso vzniklé opakováním kruhu na každém bodu úsečky, kde plocha kruhu je kolmá ke směrnici úsečky. K definici tedy stačí poloměr r a úsečka s bodem p a směrnicí v

3.3.4 Kolize mezi složenými objekty

Kolize složených objektů nastává, koliduje-li jakákoliv dvojice primitiv z vybraných objektů.

Kapitola 4

Vizualizace

Vizualizaci používáme pro lepší pochopení složitých systémů, pro které naše představivost není dostatečně natrénována a rychlého ověřování výstupu, jelikož mozek je schopen zpracovávat komplexní obrazová data efektivněji, než jejich ekvivalentní textový popis.

Pro vizualizaci by byla vhodná reprezentace pomocí více dvourozměrných obrazů, ukazovaných současně, které je mozek zvyklý interpretovat; však i přes dlouhou dobu vývoje technologie schopné dodat "trojrozměrný"[1] obraz, je ekonomicky nedostupná.[2]

Zobrazování tak probíhá jednoduchou projekcí do dvou dimenzí, které současné zařízení zvládají. Informace, ztracené projekcí, získáváme dalšími projekcemi stejného systému z jiného směru v jiném čase, díky čemuž jsme nakonec schopni sestavit trojrozměnou představu.

Pro tuto část práce nebylo, jako v předchozích, napsáno vše od základů. Budou představeny nejběžnější vizualizační rozhraní.

4.1 Používaná rozhraní

4.1.1 3D vykreslování

Existuje celá řada rozhraní, která nejsou používána; může to být z důvodu hardwaru, pro který byla navržena, upadlého v zapomnění,[3] či byla pouze nahrazena lepším návrhem. Následující dvě představená rozhraní si na nezájem nemohou stěžovat.

Open Graphics Library

Zkracováno na OpenGL, je API[4] pro zobrazování až trojrozměrné grafiky. Roku 1992 vydala SGI[5] úpravu své IRIS GL™, jako první specifikaci OpenGL a ustanovila komisi[6] starající se o změny rozhraní. Záměrem bylo donutit výrobce grafických karet k podpoře jednoho API, jež by umožnilo tvořit na hardwaru nezávislé aplikace. Již od počátku bylo pod svobodnou

[1] Pro skutečný trojrozměrný obraz by byla třeba býti alespoň čtyřrozměrnou bytostí.

[2] Nejlepším počinem současnosti, zdá se být systém CAVE [17]. V České republice existují dvě pracoviště, jež se chlubí vlastní jeskyní – katedra geografie přírodovědecké fakulty univerzity Jana evangelisty Purkyně v Ústí nad Labem a institut intermedií při fakultě elektrotechnické Českého vysokého učení technického v Praze.

[3] Herní konzole.

[4] Application Programming Interface.

[5] Silicon Graphics, Inc.

[6] OpenGL Architecture Review Board.

licencí, což spolu s relativně nízkoúrovňovým návrhem, dovolujícím více optimalizovat, než tehdejší konkurence,[7] vedlo v devadesátých letech dvacátého století k postavení OpenGL na první místo při výběru rozhraní pro solventní projekty, jak můžeme vidět z existence na trhu i po dvaceti letech, kdy grafické karty vykázaly značný pokrok [11].

Direct3D

Součástí DirectX API, vyvíjeného Microsoft Corporation, od roku 1995. S cíly stejnými, jako OpenGL, se i přes uzavřenost API, ze které plyne nemožnost[8] užití, mimo microsoftem podporovaných platforem,[9] stal nejrozšířenějším API pro vývoj her, tento úspěch nelze nepřiřknout aktivnímu marketingu.

Ostatní

Ač neexistují pouze dvě výše zmíněné rozhraní, použití všech zbývajících je v dnešní době velmi úzké. Pixar používá pro své filmy otevřené rozhraní RenderMan [10] narvžené na zobrazování fotorealistických scén. X3D je ISO standard pro popis trojrozměrné grafiky v xml souborech, vycházející z VRML97 [13]. Glide je API od 3dfx, udržované při životě z dob slávy jejich grafik,[10] kdy těžilo s úzkého provázání s harwarem. Existují i mnohá další, pravidlem však je, že pokud stejná firma zároveň nevyrábí i grafické adaptéry, musí být rozhraní postaveno nad OpenGL, či Direct3D, aby získalo alespoň rozumnou rychlost pro vykreslování.

4.1.2 Interakce s uživatelem

Pro zobrazení v operačním systému je ještě potřeba vytvořit prostor, ve kterém se může vizualizace prezentovat. Nejvíce knihoven se věnuje zapouzdření OpenGL, jelikož je otevřeným standardem.

OpenGL Utility Toolkit

Základní nástroje pro práci s OpenGL, umožňuje správu oken, vstupních zařízení a jednoduchých kontextových menu. Původní projekt[11] se již nevyvíjí a namísto něj se používá svobodná implementace freeglut,[12] která podporuje i vývojáři žádané rozšíření.[13]

Simple DirectMedia Layer

Podobně jako GLUT, i SDL zapouzdřuje platformně specifické rozhraní, více se však zaměřuje na mutimédia a dovoluje tak i pracovat se zvukem, optickou mechanikou a programovými vlákny [8].

---

[7] Zejména PHIGS [18].

[8] Možnost existuje, kupříkladu ve formě WINE `http://winehq.org`, nedá se však považovat za schůdnou.

[9] Windows a Xbox.

[10] Druhá polovina devadesátých let.

[11] Tvůrce Mark J. Kilgard `http://www.opengl.org/resources/libraries/glut/`.

[12] Tvůrce Pawel W. Olszta, dnes udržuje Steve Baker `http://freeglut.sf.net/`.

[13] Například není třeba mít v programu nekonečnou hlavní smyčku, ale je možno si vyžádat pouze zpracování jedné události.

Kapitola 5

Implementace

5.1 Haskell

Nejzajímavějším prvkem práce je bezesporu implementace v haskellu, ač vzniklý roku 1990, stále nerozšířil se příliš za akademickou půdu pravděpodobně proto, že se jedná o čistě funkcionální jazyk[1] a programátoři nechtějí přemýšlet, bez "vedlejších efektů".[2] Z dalších výhod haskellu nelze nezmínit líné vyhodnocování dovolující pracovat s nekonečnými strukturami a dávající, spolu s základním dobrým zvykem funkcionálního programování, tedy psaním generických, skládatelných funkcí, funkcím vlastnosti specializovaných.[3] Ve funkcionálních jazycích jsou hodnoty neměnné, a všechny "změny" jsou provedeny vytvoření nového objektu. Máme tak jistotu, že data, se kterými pracujeme, budou vždy konzistentní i v případě konkurentních aplikací.

5.1.1 Typový systém

Jelikož se jedná o staticky typovaný jazyk, je všechna kontrola prováděna v době kompilace, vlastnost velmi výhodná pro programování časově náročných algoritmů, jako v našem případě, kde uděláme-li například chybu z nepozornosti,[4] nemusíme čekat mnoho minut, až program zhavaruje, abychom nesrovnalost odstranili.[5]

Haskell umí odvozovat typy ze samotného použití funkcí ve zdrojovém kódu, ale explicitní deklarace pomáhá udržovat mentální obraz algoritmu a i v této práci budou anotace prezentovány s funkcemi, jelikož zároveň vypoví mnoho o významu funkce a v kombinaci s rozumně pojmenovanými datovými typy slouží ve zdrojových kódech, jako dokumentace.

Pro vyjádření, že `f` je typu `t`, napíšeme `f :: t`. Samozřejmě bychom chtěli i silnější vyjadřovací prostředky a tak přidáme operátor dovolující aplikaci typu `t` na `f`, dávající

[1] Přídavné jmého čistě znamená, řečeno "imperativní terminologií", že cokoliv funkce udělá, může se projevit pouze v jejím výstupu, jež musí záviset pouze na vstupních hodnotách.

[2] Typickým příkladem programování s vedlejším efektem je změna prvku uvnitř pole v C, kterážto mění globální stav přístupný i mimo funkci.

[3] Příkladem buď výběr nejnižších n prvků z neseřazeného pole velikosti m. Haskellová funkce `sort`$[O(m \log m)]$ a `take n`$[O(n)]$ svým spojením `take n . sort` získají složitost $O(m + n \log m)$, což oceníme zejména při výběru malého počtu nejmenších prvků a hlavně, nebylo třeba psát specializovanou funkci, jak by tomu bylo u nelíných jazyků, jež by jinak musely nejdříve seřadit celé pole $[O(n + m \log m)]$. (funkce `sort` i`take n` mají typ `[a] -> [a]` a spojovací ještě se objevila spojovací funkce, která má stejný význam, jako v matematice užívaná funkce "po" `(.) :: (b -> c) -> (a -> b) -> (a -> c)`) [http://apfelmus.nfshost.com/articles/quicksearch.html]

[4] předání seznamu, místo jeho prvku

[5] Existuje rčení: "Pokud se zkompiluje, pak funguje správně."

vzniku typu `u`, `f :: t -> u`. Alternativně můžeme pojmenovat `f` jako funkci mající za parametr typ `t` a vracející typ `u`. Více "parametrů" bychom zaznamenali `f :: a -> b -> c`.[6]

Uvedeme ještě jeden operátor umožňující nám klást obecné předpoklady na typ funkce. `f :: C a => a -> b` znamená, že typ `a` musí splňovat podmínku `C`.

Polymorfismus

S polymorfismem jsme se setkali v minulé sekci, kde byl každý uvedený typ polymorfický, vyjádřen typovými proměnými. Polymorfusmus nám dovoluje psát funkce, u kterých nezáleží na konkrétním typu, když například vybíráme první prvek ze seznamu, pak nemusíme vědět, je-li jím číslo, či řetězec, jelikož s vlastním prvkem nic neprovádíme.[7]

Haskell dále rozšiřuje polymorfismus pomocí typových tříd, definujích množinu funkcí, jež musí být pro daný typ definovány a ty je možno užívat v těle funkcí, které používají typové třídy jako podmínky pro polymorfické typy; vzpomeňme na operátor `=>`. Výhodou (a i nevýhodou) oproti předchozímu typu polymorfismu je nutnost definice typové třídy pro každý typ zvlášť.[8] Ve své podstatě je jedná o třídní objektový systém, kde mnohonásobná dědičnost je povolena a kolize jsou řešeny skrýváním a pojmenovaným importem modulů. Typové třídy navíc mohou poskytovat i výchozí definici závislou na jiné funkci ze stejné typové třídy, či třídy uvedené v typové podmínce.[9]

5.2 Planování

Implementován byl algoritmus SRT s použitím RRT expanze stromů a BFS pro vyhledání cesty, jejich pseudokódy jsou uvedeny v kapitole 2, pro použití jsou algoritmy skryty v třídě `Plan` a v konfiguračním souboru je možno i dodat vlastní implementaci.

Interpolace

Pro interpolaci byl zvolen algoritmus, jehož vzorkovací frekvence je odvislá od vzdálenosti a úhlu, a použije právě takový krok, aby robot nemohl minout i velmi malou překážku, ale zároveň nebyl příliš jemný, čímž by se zvedla výpočetní náročnost.

[6] V Haskellu má každá funkce pouze jeden parametr. `f :: a -> b -> c` je funkce přijímající typ `a` a vracející funkci `fa :: b -> c`, mohli bychom psát `f :: a -> (b -> c)`, ovšem pro pohodlnost tak nečiníme a řekneme, že `->` je zprava asociativní. Na druhou stranu `f :: (a -> b) -> c` přijímá funkci přijímající `a` vracející `b` a po aplikaci této funkce vrací `c`

[7] Mějme příklad jednodušší funkce `id :: a -> a`, která pro vrátí svůj vstup. Je definována `id a = a`, kde vidíme, že s prvkem nic neprovádí a proto může mít polymorfní typ. Aplikací skutečného typu `Int`, reprezentujícího přirozená čísla, ale získáme `id 1 :: Int` a nikoliv `id 1 :: a`, jelikož typové proměnné jsou na sebe vázané.

[8] Chceme-li například vyjmout ze seznamu duplicitní záznamy funkcí `nub :: Eq a => [a] -> [a]`, musí být nejprve pro chtěný typ definována třída `Eq a`. (definice se provádí vytvořením instance třídy a definováním funkce `(==) :: x -> x -> Bool` pro zvolený typ `x`)

[9] Haskellem definovaná třída `Ord a`, která značí, že daný datový typ je možno seřadit, má plný typ `Eq a => Ord a`, tedy datový typ musí mít definovánu nad sebou rovnost a poté stačí definovat funkci `<=` a zbytek si sám odvodí z již definované funkce `==`. Samozřejmě je možno přepsat všechny porovnávací funkce, dájí-li se lépe optimalizovat.

Spojování uzlů

Probíhá vždy přímkou, jelikož pracujeme s pravděpodobnostními algoritmy a vytváření spoje s použitím dalších uzlů by bylo pouze duplikací kroku rozšiřování stromu.

Optimalizace cesty

Základní technikou je zkoumání cesty od počátku a v každém uzlu se zkusit napojit na první možný při postupu odzadu, až v nejhorším případě dojdeme na následující uzel a od něj začne zkoumání nanovo. Uzly také můžeme prokládat křivkami, kterézto mají největší smysl u kinodynamických úloh, kde se odvíjejí od vlastností robota.

5.3 Kolize

Zvolený algebraický model světa byl přímočaře přenesen do haskellu, jelikož již notace použitá v kapitole 3 odpovídá deklaraci haskellových datových typů, nebude zde již opakována.

Pro kolize je vytvořena třída `Collide a b` s funkcí `collide :: a -> b -> Bool` a množina typů reprezentujících základní geometrické útvary, které mohou být vzájemně testovány na kolize jsou všechny členy třídy `Collide`.

Složené útvary jsou popsány rekurzivní datovou strukturou `Objekt`, kde každý `Objekt` je buď popsán pozicí vzhedem k nadřazenému objektu a množinou podřazených `Objekt`ů, či tvarem, který reprezentuje.

$$Objekt \;=\; Pozice\ [Objekt] \mid Tvar \tag{5.1}$$

Tato hierarchie dovoluje nerozlišovat mezi prostorovými rozměry Tvaru, kterýžto je vždy definován vzhledem k počátku, jímž je pohybováno a bod, úsečka, trojúhelník, či koule tak mohou být navzájem testovány na kolize jednotným způsobem.

5.3.1 Pozice

Prostorové pozice jsou vyjádřeny pomocí vektorů pro posun a kvaternionů pro rotaci.

Výhoda kvaternionů oproti rotačním maticím je nižší paměťová i výpočetní náročnost, nejsou však zobecnitelné do vyšších dimenzí. Pro popis rotace postačuje reálné číslo a trojrozměrný vektor $Quaternion = (Real, Vector)$ Složení rotací kvaternionů $q1(r1, i1)$ a $q2(r2, i2)$ provedeme jako

$$q12 = (r1r2 - i1 \cdot i2, r1 * i2 + r2 * i1 - i1 \times i2) \tag{5.2}$$

5.4 Vizualizace

Jak bylo uvedeno v kapitole 4, není vykreslování postaveno na vlastní knihovně. Volba implementačního rozhraní však díky použí linuxu a haskellu nebyla komplikovaná, jelikož jedině OpenGL je připraveno na tuto kombinaci.[10]

[10] Aplikace mohla býti ještě pod F# a Direct3D, však MSDNAA zabránilo, po druhém pádu Windows 7 v průběhu stahování Microsoft Visual Studio 2010, získání vývojových nástrojů, čímž zůstala pouze haskellová implementace.

Hlavní nevýhodou je samotný haskell, který manipulaci s "neuchopitelným stavem" může provádět pouze v IO monádě, kterážto je velmi efektivní koncept, jak dostat vedlejší efekty do čistého světa. Platí se za to ovšem spolehlivostí kódu[11] a potěšením z programování. Z druhého důvodu byl vytvořen pouze jednoduchý vykreslovač, u kterého interaktivita končí na možnosti otáčet pohled, animovat, a skrývat některé komponenty; jeho výhodou je rozhraní obsahující pouze jednu funkci přijímající seznam objektů k vykreslení. Zbytek výpočtu je možno provádět mimo IO monádu.

U kolizí byla uvedena definice objektu, které používá i vizualizace.

Pro vizualizaci bylo použito rozšíření o existenciální kvantifikátory užité, aby umožnilo mít seznam různých typů shopných se vykreslovat, neb jinak seznam může obsahovat pouze jeden typ.[12]

5.5 Ovládání

Práce s programem probíhá ve dvou fázích.

5.5.1 Definice robota

Robota definujeme pomocí haskellového kódu. Vytvoříme datový typ robota, ideálně odvozený od `Object3D`,[13] a okolní překážky. Pro pohodlnou a neopakující se definici plánování je vytvořena třída `Plan`, které stačí pro úplnou definici stačí, abychom předtím definovali vytvořený datový typ robota pro třídu `Robot`

Třída Robot

Pro úplnou definici je třeba dodat tři funkce.

- `configuration :: StdGen -> (a, StdGen)` vrácející náhodnou konfiguraci. Očekávána je funkce, která již bere v potaz prostorové omezení konfiguračního prostoru.
- `free :: a -> Bool` ověřující, zdali dochází ke kolizi. Funkce takto definovaná, by měla nést informaci o okolním světě, se kterým se robot sráží.
- `interpolate :: a -> a -> [a]` tvořící seznam poloh, které robot po cestě mezi dvěma body zaujme.

5.5.2 Definice vizualizace

Můžeme také nastavit objekty, které budeme chtít zobrazovat. Nastavení se provádí předáním seznamu objektů k vykreslení funkci `GUI.visualize :: [Renderable] -> IO ()`. Každému objektu lze přiřadit několik vlastností, alespoň však jednu.

- `Static :: Render a => a -> Renderable` bude vždy vykreslovat získaný objekt.
- `Optional :: Render a => a -> Renderable` bude vykreslovat pouze po přepnutí volby v gui.

[11] `add(a,b)` vracející `a+b` se v IO monádě může chovat stejně, jako obdobná funkce v C – stáhne poštu, vynuluje paměť a následně bude chtít vrátit `a+b`.

[12] Technicky stále obsahu pouze jeden typ vyjadřující vykreslitelnost obalených typů.

[13] Zadarmo tak dostaneme již kompletní detekci kolizí i vykreslování.

- `Animation :: Render a => Int -> [a] -> Renderable` vyžaduje seznam vykreslitelných stavů, nejlépe získaných interpolací, a číslo udávající krok kterým se bude seznam procházet, poté se tento seznam cyclicky vykreslovat..
- `NoLighting :: Render a => a -> Renderable` zruší počítání světla pro daný objekt, čímž dostane rozdílnou podobu od podobných osvětlených objektů.

5.5.3 Vytvoření a vizualizace dat

Hotový definiční program kompilujeme s knihovnou, čímž vznikne spustitelný soubor pro další práci. Pro kompilaci byl zkoušen pouze Glasgow Haskell Comiler,[14] a velmi pravděpodobně je také jediným použitelným překladačem, jelikož podporuje téměř všechna rozšíření haskellu a mnoho z nich je v programu použito.

```
$ ghc --make program
  ...
$ ./program --help
Usage: program [Options]
  -m File  --roadmap=File  roadmap datafile
  -r File  --route=File    route datafile
  -n       --new           start with empty datafiles
  -c       --continue      load roadmap and route from datafiles
  -v       --visualize     render the results
  -e Int   --expand=Int    number of vertices to expand per graph
  -g Int   --graphs=Int    number of graphs to generate
  -s Int   --seed=Int      seed for random number generator
  -h       --help          what do you think, you are reading?
```

Prakticky se nejrychleji dosáhlo výsledku při postupném přidávání stromů a vrcholů, až do chvíle, než byla nalezena cesta, však s dostatkem času není problém nastavit velké rozšíření již v počátku, které je pohodlnější.

```
$ ./program --new -e 100 -g 30 -s 0 -m roadmap -r route
$ ./program --visualize             -m roadmap -r route
```

a pokračovat s měnícím se počtem přidávaných grafů a uzlů

```
$ ./program --continue -e $X -g $Y -m roadmap -r route
$ ./program --visualize            -m roadmap -r route
```

Ve vizualizačním módu se zobrazí okno s objekty vybranými definičním souborem, animace se pouští od počátku a stále cyklí, volitelné objekty jsou skryty. Ovládání se provádí klávesnicí a myší, kde myš po stistu levého, či pravého tlačítka spolu s pohybem ovládá jednu ze čtyř os (rotace kolem tří základních os a přibližování). Klávesové zkratky jsou `Esc, Control-q, Enter, q` pro ukončení a `o` pro přepnutí, zdali-se vykreslují volitelné objekty.

[14] http://www.haskell.org/ghc/, jiné překladače jsou například hugs http://www.haskell.org/hugs/, či nhc98.http://www.haskell.org/nhc98/

5.6 Ježek v kleci

5.6.1 Model

Vzorem pro vytvořený model ježka v kleci byl kus zapůjčený, vedoucím práce, ing. Rozmanem. Výsledek však byl ještě idealizován, pro vytažení v rozumném čase. Před idealizací byl algoritmus puštěn po dobu pěti dní na slabém stroji intel atom a výsledkem nebyla mapa schopná vytáhnout ježka, zejména to bylo způsobeno nízkou pravděpodobností pro nalezení volné konfigurace mezi mřížemi, na což bylo reagováno uvedeným způsobem.

- odstanění základen klece, kteréžto nemá na výšledek vliv, jelikož i před jejich odstraněním mohl ježek být vždy posunut nahoru, či dolů tak, aby nedošlo ke kontaktu se základnou a konfigurační prostor je omezen způsobem nedovolujícím ježku se skrze díru dostat. Výhodou je lepší možnost pozorovat cestu vytahovaného ježka.
- zpravidelnění odstupů mezi mřížemi přineslo další možnost omezení konfiguračního prostoru, který stačilo zmenšit pouze na jednu pětinu. Problém změna však nijak nemění, neb ježek se může dostat ven vždy pouze jedním otvorem a u pravděpodobnostních algoritmů by pouze projití stejné cesty trvalo déle, než s prostorem omezeným pouze na správný východ.

Výsledkem je svět o dvou objektech – kleci a ježkovi. Klec je složena z pěti válců postavených na vrcholu pravidelného pětiúhelníku a ježek z koule a dvanácti válců. Dva válce vstupují kolmo do pólů a zbylé tvoří dva pravidelné pětiúhelníky na rovnoběžkách v třetinách poledníku. Každý osten má jinou délku.

5.6.2 Vytažení z klece

Pro extrémní časovnou náročnost, umocněnou procesorem intel atom, bylo vyhledávání možných cest ukončeno po prvním úspěšném nálezu.

Tento vznikl asistovaným zužováním množiny map na základě odhadu, zda mají šanci po malém množství nových vrcholů vytvořit možnou cestu. Z osmnácti úvodních grafů složených z 10 až 100 stromů, rozšířených o 500 až 50 vrcholů, byly vybrány dva, postupně rozšiřované o 100 vrcholů v každé iteraci a jeden z nich po zhruba tisíci vrcholech uspěl.

Výsledná cesta má deset uzlů a z kořenového adresáře přiloženého CD je možno ji prohlédnout příkazem

```
(cd)$ ./program/hedgehog -v -r data/hedgehog_route
```

Kapitola 6

Možná rozšíření

6.1 Dynamické objekty

Současná implementace vyhledávacích algoritmů předpokládá, že robot je jediným pohyblivým prvkem, což však v reálných problémech nelze zajistit.

Přidáním času, bychom mohli docílit reprezentace pohyblivého světa. Jeho připuštění nemůže být stejně jednoduché, jako přidání dalšího rozměru do euklidovského prostoru, jelikož i intuitivně tušíme se nemůžeme pohybovat časem dle libosti.

Problémem zůstává nutnost dokonalého odhadu, jak se budou objekty v budoucnu chovat. Kompenzovat nepřesnosti odhadu by se dalo zavedením "fuzzy" objektů, které by s jistou pravděpodobností zabíraly více prostoru ve svém okolí, než je jejich skutečný objem, kde do pravděpodobnosti by se promítla domělá přesnost našeho odhadu a míra přípustného rizika nezjištění srážky.

6.2 Distribuce náhodných konfigurací

Základní model má stejnou pravděpodobnost každého bodu v konfiguračním prostoru, ale jelikož v okolí překážek je pravděpodobnost volné konfigurace mnohem nižší, nejsou komplikovaná místa tak hustě pokryta, jako volná prostranství, což neoceníme.

Existuje celá řada technik, jak se danému problému vyhnout.

Bridge-test [4, 9] generuje vždy dvojici konfigurací a jsou-li obě kolizní, přidá konfiguraci uprostřed, pokud je volná.

Můžeme také nad konfiguračním prostorem vytvořit mřízku, a uchovávat informaci o počtu konfiugrací v každé buňce, čehož by šlo využít pro pravděpodobnosti výběru konfigurace z málo kolizních buňek.

6.3 Spojení s realitou

Přidáním dynamického rozpoznávání okolí, schopného vytvořit aproximovaný virtuální svět a zpětným napojením plánovacích algoritmů na ovládaného robota by bylo možno vytahovat skutečného ježka z klece. Toto rozšíření by vyžadovalo nejvíce úsílí z uvedených, po implementaci by však jako jediné mělo šanci na uplatnění.

Kapitola 7

Závěr

Byly představeny plánovací algoritmy, detekce kolizí a základy haskellu, zájemci o tato témata mohu doporučit knihy [9], [6], a [2], z nichž každá je dle mého názoru nejlepší v oboru (z čerpaných zdrojů).

Praktická část splnila očekávání vytvořením programu užívajícím pravděpodobnostní plánování k vytažení ježka z klece. K tomuto účelu byl vytvořen obecný systém, ve kterém mohou být modelovány a řešeny problémy vyžadující plánování pohybu. Možnosti jeho rozšíření jsou diskutovány v kapitole 6.

Literatura

[1] Akenine-Möller, T.; Haines, E.; Hoffman, N.: Object/Object Intersection. http://www.realtimerendering.com/intersections.html, [cit. 2010-05-18].

[2] Bryan O'Sullivan, D. S.; Goerzen, J.: *Real World Haskell*. O'Reilly Media, 2008, ISBN 978-0-596-51498-3, 720 s., také online http://book.realworldhaskell.org/read/.

[3] Chazelle, B.: Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, ročník 6, č. 5, 1991: s. 485–524, ISSN 0179-5376, doi:http://dx.doi.org/10.1007/BF02574703, také online http://www.ime.usp.br/~walterfm/cursos/mac0331/2006/chazelleT.pdf.

[4] Choset, H.; Lynch, K. M.; Hutchinson, S.; aj.: *Principles of Robot Motion: Theory, Algorithms, and Implementations*. Cambridge, MA: MIT Press, June 2005, ISBN 0-262-03327-5.

[5] Dijkstra, E. W.: A Note on Two Problems in Connexion with Graphs. In *Numerishe Mathematik 1*, 1959, s. 269–271, také online http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf.

[6] Ericson, C.: *Real-Time Collision Detection*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004, ISBN 1-558-60732-3.

[7] Howard, A.: *Elementary Linear Algebra*. Wiley, osmé vydání, 2000, ISBN 0-471-17055-0.

[8] Lantinga, S.: Simple DirectMedia Layer Introduction. http://www.libsdl.org/docs/, [cit. 2010-05-18].

[9] LaValle, S. M.: *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006, také online http://planning.cs.uiuc.edu/bookbig.pdf.

[10] Pixar: The RenderMan Interface Specification. http://renderman.pixar.com/products/rispec/, [cit. 2010-05-18].

[11] Silicon Graphics: Open Graphics Library. http://www.sgi.com/products/software/opengl/?/overview.html, [cit. 2010-05-18].

[12] Sunday, D.: Distance between Lines and Segments with their Closest Point of Approach. http://softsurfer.com/Archive/algorithm_0106/algorithm_0106.htm, [cit. 2010-05-18].

[13] Web3D Consortium: Open Standards for Real-Time 3D Communication. http://www.web3d.org/x3d/, [cit. 2010-05-18].

[14] Weisstein, E. W.: Line-Line Distance. http://mathworld.wolfram.com/Line-LineDistance.html, [cit. 2010-05-18].

[15] Weisstein, E. W.: Line-Line Intersection. http://mathworld.wolfram.com/Line-LineIntersection.html, [cit. 2010-05-18].

[16] Wikipedia: A* search algorithm. http://en.wikipedia.org/wiki/A*_search_algorithm, [cit. 2010-05-18].

[17] Wikipedia: CAVE Automatic Virtual Environment. http://en.wikipedia.org/wiki/Cave_Automatic_Virtual_Environment, [cit. 2010-05-18].

[18] Wikipedia: Programmer's Hierarchical Interactive Graphics System. http://en.wikipedia.org/wiki/PHIGS, [cit. 2010-05-18].

Příloha A

Obsah CD

Tři základní složky.

`program` zdrojové kódy a přeložená implementace pro běh na linuxech CVT FIT.

`data` cesta a mapa schopná vytáhnout ježka z klece, vhodná k vizualizaci.

`text` zdrojové kódy a přeložná bakalářská práce ve formátu pdf.