VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

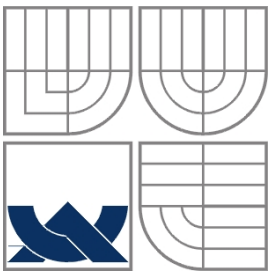
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VIZUALIZACE HLEDÁNÍ CESTY PRO ROBOTA

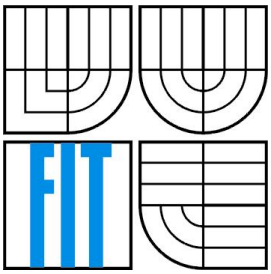
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE Vít Sykala
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VIZUALIZACE HLEDÁNÍ CESTY PRO ROBOTA

VISUALISATION OF PATH-FINDING FOR ROBOT

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — Vít Sykala
AUTHOR

VEDOUCÍ PRÁCE — Ing. Jaroslav Rozman
SUPERVISOR

BRNO 2007

Abstrakt

Tato bakalářská práce slouží k vysvětlení funkce algoritmů na vyhledávání a plánování cesty robota ve známém prostředí. V první části se věnuje rozdělení a vysvětlení algoritmů Road map a Buněčné dekompozice. Dále je zde popsán vývoj appletů na vizualizaci těchto algoritmů. Konkrétně na vizualizaci algoritmu pro sestavení Grafu viditelnosti a algoritmu pro Lichoběžníkovou dekompozici. Jako součást této bakalářské práce vznikla také webová prezentace těchto algoritmů, kde jsou k vidění a odzkoušení i zmiňované applety.

Klíčová slova

vizualizace algoritmu, plánování cesty, pohyb robota, Road mapa, Buněčná dekompozice, Lichoběžníková dekompozice, Morseova dekompozice, Graf viditelnosti, Voronoiův graf,

Abstract

This bachelor´s thesis is explanation of algorithms for finding and planning robot motion in known space. First part is about margin and explanation of algorithms: Roadmaps and Cell decomposition. Next part is about progress of creating Java applets for visualization of these methods. Visualization of create Visibility graph and Trapezoidal decomposition in the concrete. Web presentation about these algorithms was too created as a part of this bachelor´s thesis. Here can be seen described Java applets.

Keywords

visualization algorithms, path planning, robot moving, Roadmap, Cell decomposition, Trapezoidal decomposition, Morse decomposition, Visibility graph, Voronoi diagram.

Citace

Vít Sykala: Vizualizace hledání cesty pro robota, bakalářská práce, Brno, FIT VUT v Brně, 2008

Vizualizace hledání cesty pro robota

Prohlášení

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

.....
Vít Sykala
28. 4. 2008

Poděkování

Chtěl bych poděkovat vedoucímu práce Ing. Jaroslavu Rozmanovi, který mi byl při zpracovávání bakalářské práce vždy nápomocen.

Obsah

1 Úvod

Robotika je dnes velice aktuálním oborem, který má široké využití v průmyslové výrobě, při domácích pracích, ale i v lékařství a mnoha dalších oborech lidské činnosti. Tento obor se zabývá konstrukcí robotů a jim podobných zařízení a to jak konstrukcí hardware, tak i software. Jedním z problémů, kterými se robotika zabývá, je **hledání a plánování cesty robota** známým nebo neznámým prostorem. Existuje mnoho způsobů, jak hledat či plánovat cestu robota, tyto algoritmy se dělí do následujících skupin: Bug algoritmy, Potenciálová pole, Buněčná dekompozice, Road mapy, …

Cílem této práce je prozkoumat algoritmy **Road map** a **Buněčné dekompozice** a následně sestrojit Java applety pro vizualizaci jednoho algoritmu z každé skupiny. Tyto applety budou umístěný na stránkách zabývajících se těmito algoritmy a budou sloužit k experimentování a lepšímu pochopení daného algoritmu.

Následující kapitola popisuje tyto algoritmy. Po přečtení by čtenář měl získat ucelenou představu o algoritmech pro hledání cesty robota a být schopen kterýkoliv z nich implementovat pro to, co bude potřebovat.

Kapitola 3 popisuje vytvářené Java applety. První z nich demonstruje sestavení a používání **Grafů viditelnosti** jako příkladu Road mapy a druhý **Lichoběžníkovou dekompozici** jako typický příklad Buněčné dekompozice. Je zde popsán jak návrh těchto appletů, tak i návod k jejich používání.

2 Algoritmy plánování

Algoritmy plánování mají typicky 2 fáze. První fáze je převedení prostoru do požadovaného tvaru pro daný algoritmus (typicky na polygonální prostor) a následné zmapování prostoru podle daného algoritmu. Výstupem je zmapovaný prostor nachystaný pro hledání cesty mezi dvěma pozicemi. Druhá fáze má jako vstup výchozí, cílovou pozici robota a zmapovaný prostor z první fáze. Výstupem je cesta mezi těmito body. U této fáze se předpokládá její několikanásobné opakování s různými vstupy.

Jsou zde tedy dvě otázky: jak zmapovat prostor a jak v této mapě následně vyhledávat cestu. A na tyto otázky odpovídají právě algoritmy **Road map** a **Buněčné dekompozice**.

2.1 Road map

Pojem Road map by se dal volně přeložit jako mapa klíčových cest. Má stejnou vyjadřovací schopnost jako mapa dálnic, drážní mapy kolejních koridorů a mnoho dalších.

Road map algoritmy vytvářejí mapu klíčových cest (Road mapu) daným prostředím. Tyto algoritmy využijeme především, víme-li, že daným prostředím budeme plánovat více cest. Pro dané prostředí se vytvoří mapa klíčových cest robota. Neobsahuje úplně všechny cesty, ale pouze ty hlavní zajišťující v případě robota průchod mezi překážkami.

2.1.1 Dělení Road map podle vzniku

Podle toho, jak daná Road mapa vzniká, se dají Road mapy rozdělit:

- **Mapy viditelnosti:**
  Vznikají na základě viditelnosti vhodně zvolených bodů.
- **Deformací prostoru vzniklé mapy:**
  Vznikají deformací prostoru na prostor jednodušší. Každý bod původního prostoru má své zobrazení v tomto nově vzniklém prostoru.
- **Další méně významné** a méně zdokumentované skupiny

2.1.2 Definice Road mapy

Road mapa je množiny jednodimenzionálních křivek, kde platí pro všechny startovní pozice robota q_{start} a cílové pozice robota $q_{cíl}$:

- Dostupnost pro start: pro každý startovní bod z volného prostoru existuje cesta do nějakého uzlu RM, který je pro algoritmus považován za startovní q'_{start}
- Dostupnost pro cíl: totéž pro cílový bod
- Spojitost: v grafu existuje cesta mezi body q'_{start} a $q'_{cíl}$

2.1.3 Vlastnosti

Road map algoritmy mají také velice příznivé vlastnosti, co se týče nalezení cesty. Pro sestrojování a používání Road map platí:

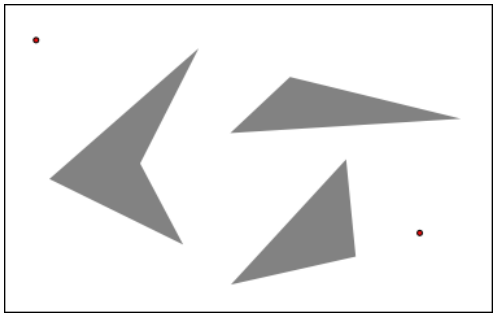
- pokud cesta existuje, tak je nalezena
- pokud je cesta nalezena, tak je to ta nejoptimálnější vzhledem k ohodnocení hran
- pokud cesta nalezena není, znamená to, že cesta neexistuje, lze to i matematicky dokázat

2.1.4 Grafy viditelnosti

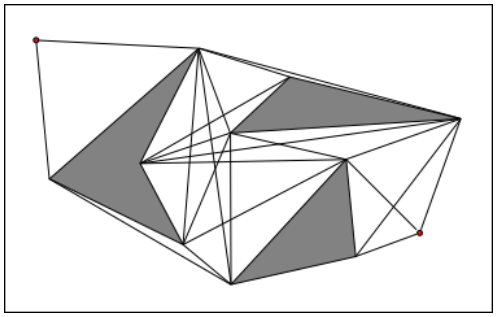
Algoritmus je určený především pro hranaté překážky, protože jednotlivé vrcholy vzniklého grafu sdílejí jednotlivé rohy překážek. V případě, že bychom chtěli použít na oblé objekty, bylo by nutno nahradit tyto oblé objekty dostatečnými polygony. Ovšem nárůst vrcholů má vliv na dobu při sestavování grafu viditelnosti. Další možností je přímo v algoritmu pro sestavení viditelnosti počítat tečná místa, čímž by se vyřešil problém oblých překážek, ale algoritmus by stále nebyl optimálně rychlý, i když někdy to nevadí a pro některé případy je vhodné, aby graf viditelnosti byl co nejpřesnější, protože se bude používat dlouho a pro tento účel je to optimálním řešením.

2.1.4.1 Definice Grafu viditelnosti

Graf viditelnosti je Road mapa definovaná jako dvojdimenzionální polygonální zobrazení prostoru. Uzly v_i obsahují startovní i cílovou konfiguraci, vrcholy všech překážek. Hrany e_{ij} jsou spojnice vrcholu v_i a v_j pro, které platí: leží ve volném prostoru (nepronikají žádnou překážkou). Obvodové hrany překážek mohou patřit mezi hrany grafu (viz Ilustrace 2).



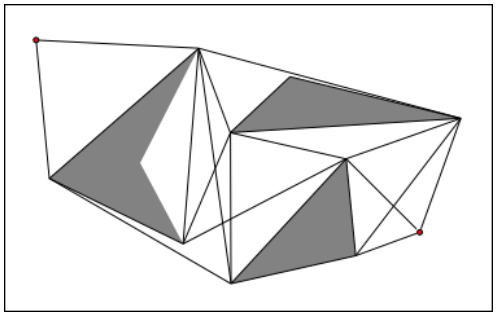
Ilustrace 1: Prostor, ve kterém se bude vyhledávat.



Ilustrace 2: Graf viditelnosti

2.1.4.2 Minimalizace počtu hran:

Je vidět, že některé hrany jsou v obrázku nadbytečné a nikdy při hledání cesty nebudou využité, proto tyto nadbytečné hrany z obrázku odstraníme. Redukcí počtu hran se zvýší rychlost následného prohledávání grafu pro nalezení cest.



Ilustrace 3: Eliminace hran grafu viditelnosti

Pro odstranění nepotřebných hran postupujeme tak, že nalezneme separující (separating) a obalující (supporting) hrany. A všechny ostatní hrany vynecháme. Separující hrany jsou takové, kde obě spojovaná tělesa leží na opačných stranách hrany. Obalující hrany jsou takové, kde obě spojovaná tělesa leží na stejné straně hrany. Tyto definice ovšem platí pouze pro konvexní tvary překážek. Obecně tedy musíme vzít polopřímky tvořící úhel pro daný vrchol a zjistit, jestli splňují podmínku pro separující nebo obalující hranu. Řečeno jednoduše: protáhneme-li hranu o malinký kousek a ona vejde skrz vrchol do překážky, tak ji vynecháme. Pokud půjde mimo, tak patří mezi separující nebo obalující hrany, tudíž je nutné ji zachovat (viz Ilustrace 3).

2.1.4.3 Konstrukce grafu viditelnosti

Pro tvorbu grafu viditelnosti si zavedeme množinu vrcholů $V=\{v_1, v_2,.., v_n\}$ a pro každý vrchol z této množiny nalezneme všechny hrany grafu viditelnosti zjištěním, které ostatní vrcholy jsou z něj viditelné. Nejběžnějším způsobem je zkusit udělat hrany mezi tímto vrcholem v a všemi ostatními vrcholy v_i a zkontrolovat, zdali se tato úsečka neprotíná někde s některou obrysovou hranou překážky. Jak vidíme složitost tohoto přístupu je kubická $O(n^3)$.

Samozřejmě existuje i efektivnější způsob, jak zjistit, které vrcholy jsou viditelné z vrcholu *v* otáčením paprsku okolo bodu. Tyto algoritmy jsou známy jako „Plane sweep algorithms“, což lze volně přeložit jako uhlazovaní zametáním. Celá optimalizace spočívá ve snížení počtu prověřovaných hran pouze na ty, které připadají v úvahu (viz Algoritmus 1).

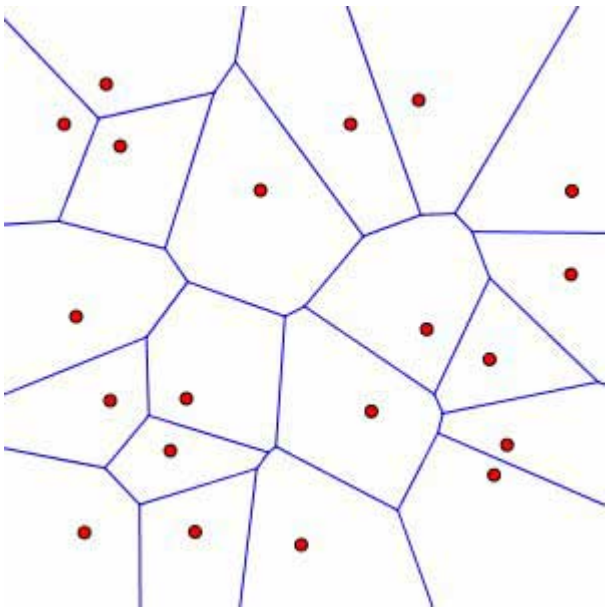
```
zkoumaným vrcholem veď vodorovnou osu x vypočti uhel úhel
ostatních vrcholů vzhledem ke kladné poloose x tyto vrcholy
seřaď od úhlu 0 do uhlu 2Π
vytvoř seznam S, do kterého vlož všechny hrany, které proťala
pomocná osa x
foreach (v_i v pořadí daném v bodě 1){
      if(spojnice v_i a v neprotíná hranu z množiny S){
                  přidej tuto hranu do seznamu výsledných hran
      }
      if (v_i je začátkem hrany, která není v množině S){
                  přidej tuto obrysovou hranu do množiny S
      }
      if (v_i je koncem hrany, která je v množině S){
                  odeber tuto obrysovou hranu z množiny S
      }
}
```

Algoritmus 1: nalezení všech viditelných hran

Tímto způsobem jsme schopni se dostat na složitost O(n*2* log(n)), což je výhodnější, je také snadné rozšířit tento algoritmus o kontrolu toho, zda daná hrana je separující nebo obalující, jak je popsáno dříve.

2.1.5 Obecný Voronoiův diagram

Voronoiův diagram dostaneme, pokud budeme všechny překážky v prostoru rovnoměrně zvětšovat všemi směry. Voronoiův diagram vznikne v těch místech, kde dojde k průniku těchto narůstajících překážek.



Ilustrace 4: Voronoiův diagram pro bodové překážky

2.1.5.1 Definice

Obecný Voronoiův diagram je definovaný jako množina bodů, které mají od nejbližších konvexních překážek stejnou vzdálenost.

Minimální počet překážek tvořících neprázdný Voronoiův diagram jsou dvě.

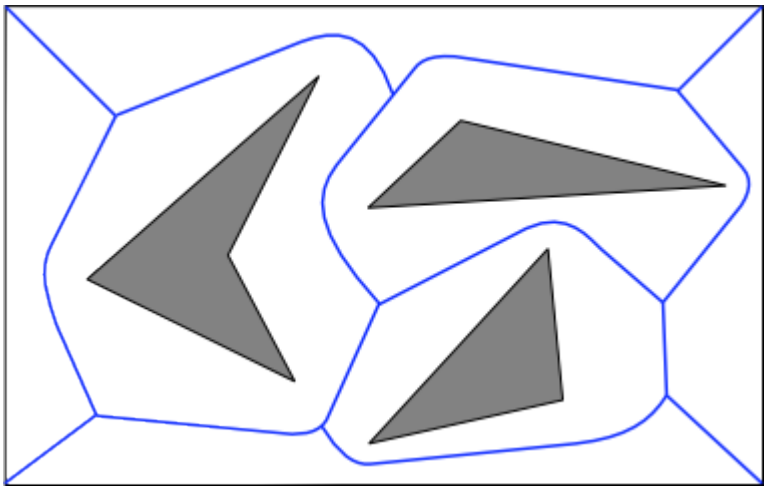
2.1.5.2 Konstrukce

Voronoiův diagram lze sestavit dvěma způsoby, buď plně teoreticky vypočítat kudy povedou jednotlivé části a čím budou tvořeny, nebo pomocí senzorů robota. Obě řešení mají svá pro a proti.

Výpočet Voronoiova diagramu lze provádět opět pouze nad polygonálním prostorem. V polygonálním prostoru k sobě mohou být dvě překážky nejblíže buď rohem a hranou nebo dvěma hranami nebo dvěma rohy. Pokud jsou dvě překážky k sobě nejblíže dvěma hranami nebo dvěma vrcholy, pak je Voronoiův diagram mezi nimi tvořen osou těchto hran, respektive vrcholů. Pokud jsou u sebe nejblíže vrcholem a hranou, pak je Voronoiův diagram mezi nimi tvořen parabolou, což je logické z definice paraboly, neboť parabola je definovaná jako množina bodů, které mají stejnou vzdálenost od ohniska (bod) a řídící přímky.

Další možností, jak sestavovat Voronoiův diagram, je pomocí senzorů robota, kdy robot jede a snaží se být co nejdále od všech překážek. Místa, kudy jede, jsou jednotlivé části hledaného diagramu. Dá se tak diagram vytvářet a editovat za chodu a robot, čím více úkolů plní, tím lépe má svůj vnitřní Voronoiův diagram sestaven. Ale rovněž se může takto vybavený robot vypustit

do prostoru a nechat, aby prostor zmapoval do posledního detailu. A v této mapě vyhledával cestu stejně, jako jo to dělá v mapě vzniklé jako graf viditelnosti.



Ilustrace 5: Přibližný tvar obecného Voronoiova diagramu

Používání Voronoiova diagramu spočívá v nalezení nejbližší překážky od startovního a cílového bodu a od nich se vydat směrem maximálně se vzdalujícím od této překážky, dokud se nenarazí na Voronoiův diagram. Poté už jen zbývá nalézt cestu mezi těmito dvěma body na Voronoiovu diagramu.

2.1.5.3 Vlastnosti

Voronoiův diagram má dobré vlastnosti, co se týče nalezení cesty, ale není triviální jej sestavit a to ani teoreticky. Díky existenci algoritmů pro sestavení Voronoiova diagramu pomocí senzoru robota vypadá tento algoritmus jako velice přijatelný pro nasazení v praxi.

2.2 Buněčná dekompozice

Další možností, jak zmapovat prostor, je rozložit jej do buněk reprezentujících části volného prostoru. Tyto buňky následně sdružit do takzvaného grafu sousednosti, ve kterém uzly jsou reprezentací buněk volného prostoru a hrany vyjadřují sousednost buněk. Sousední buňky, tj. oblasti mající společnou hranici jsou v grafu reprezentovány jako uzly spojené hranou.

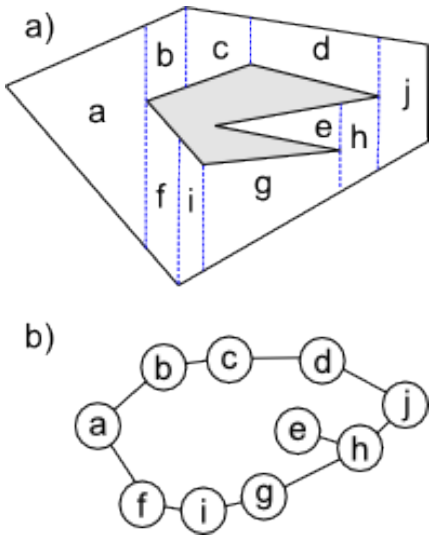
Plánování v takto reprezentovaném prostoru se potom skládá ze dvou částí. Zaprvé najít, kterým buňkám náleží startovní a cílový bod. Zadruhé najít v grafu sousednosti posloupnost buněk pro průchod od startu k cíli.

Mezi nejznámější metody dělení volného prostoru patří takzvaná **lichoběžníková dekompozice**. Prostor je dělen na lichoběžníky odtud lichoběžníková. Bohužel tento způsob je

možno aplikovat pouze na polygonální volný prostor. Obecnějším způsobem je například **Morseova dekompozice,** která ve své nejzákladnější verzi je dekompozicí lichoběžníkovou upravenou pro nepolygonální prostory.

2.2.1 Lichoběžníková dekompozice

Jak již název napovídá, jedná se o rozdělení prostoru na lichoběžníky. Lichoběžníky byly vybrány, protože pro polygonální prostor existuje velice jednoduchá metoda jak jej rozdělit na lichoběžníky. Tímto způsobem rozdělíme volný prostor svislými čarami na lichoběžníky. Na obrázku vidíte mapu s překážkami.



Ilustrace 6: Prostor rozdělený lichoběžníkovou dekompozicí výsledný graf sousednosti

2.2.1.1 Algoritmus Lichoběžníkové dekompozice

Vstupem tohoto algoritmu je množina polygonálních překážek reprezentovaných seznamem svých vrcholů (viz Algoritmus 2).

Procházíme vrcholy zleva doprava. Podle polohy hran polygonální překážky, na které daný vrchol leží. Určíme, které Lichoběžníky tento vrchol vytvoří a které pouze začne vytvářet. Platí následující pravidla:

1. Hrana, která je **vlevo** od aktuálně testovaného vrcholu **ukončuje** vytváření lichoběžníku.
2. Hrana, která je **vpravo** od aktuálně testovaného vrcholu **začíná** vytváření lichoběžníku.
3. **Předchozí** hrana, **začíná** lichoběžník **pod** sebou a **ukončuje** lichoběžník **nad** sebou.
4. **Následující** hrana, **začíná** lichoběžník **nad** sebou a **ukončuje** lichoběžník **pod** sebou.

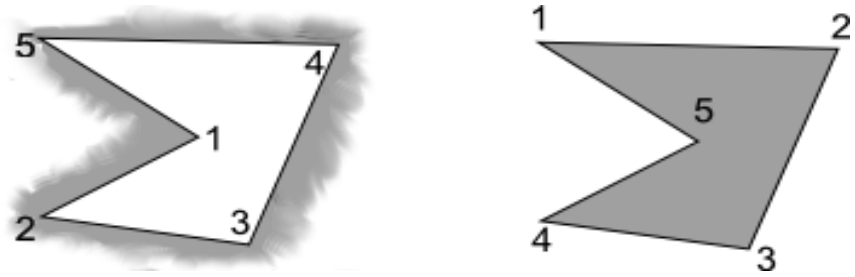
```
Seznam V = seznam vrcholů vstupních polygonů seřazený podle jejich x-ové
      souřadnice
Inicializujeme seznam L jako prázdný.
      Seznam L je seznam obrysových hran vstupních polygonů, které protíná
      zametací přímka. Hrany jsou zde seřazeny vzestupně podle y-ové
      souřadnice průsečíku hrany a zametací přímky.
foreach (V){
      if ( hrana1 při vrcholu V je v seznamu L ){
            odeber hranu1 ze seznamu L
      }else{
            přidej tuto hrana1 do seznamu L
      }
      if ( hrana2 při vrcholu V je v seznamu L )
            ukonči buňku omezenou hrana2
      }else{
            přidej hrana2 do seznamu L
      }
      if ( do L byly přidány 2 hrany ){
            if (předchozí hrana je nad následující hranou){
                  začni lichoběžník mezi předchozí a následující hranou
            }else{
                  ukonči lichoběžník před předchozí a následující hranou
                  začni lichoběžník nad následující hranou
                  začni lichoběžník pod předchozí hranou
            }
      }else if ( z L byly odebrány 2 hrany ){
            if (předchozí hrana je pod následující hranou){
                  ukonči lichoběžník mezi předchozí a následující hranou
            }else{
                  ukonči lichoběžník omezený následující hranou
                  ukonči lichoběžník omezený předchozí hranou
                  začni lichoběžník za předchozí a následující hranou
            }
      }else if ( Přidána předchozí hrana ){//=>odebrána následující
            ukonči lichoběžník omezený následující hranou
            začni lichoběžník pod předchozí hranou
      }else if ( Přidána následující hrana ){//=>odebrána předchozí
            ukonči lichoběžník omezený předchozí hranou
            začni lichoběžník nad následující hranou
      }
}
```

Algoritmus 2: algoritmus Lichoběžníkové dekompozice

Jak je vidět vyskytují se zde pojmy **předchozí** a **následující** hrana. Předchozí hrana spojuje aktuálně prověřovaný vrchol s vrcholem předchozím. Následující hrana spojuje aktuálně prověřovaný vrchol s vrcholem následujícím. Vrcholy musejí být seřazeny ve směru hodinových ručiček okolo oblasti, kterou ohraničují (Viz Ilustrace 7).

Dalším pravidla, která logicky vyplynula z předchozích čtyř, jsou pravidla o začínání nebo ukončování pouze jednoho lichoběžníku místo dvou. Pokud obě hrany začínají vytvářet lichoběžník a hrana začínající vytvářet lichoběžník **nad** sebou je pod hranou začínající vytvářet lichoběžník

pod sebou, tak se začne vytvářet pouze jeden lichoběžník mezi těmito hranami. Totéž platí pro ukončování vytváření lichoběžníku.



Ilustrace 7: polygony s vrcholy po směru hodinových ručiček

2.2.1.2 Vlastnosti

Algoritmus pro lichoběžníkovou dekompozici je velice rychlý díky svojí jednoduchosti. Celá dekompozice je jeden průchod přes seřazené pole polygonálních překážek. Algoritmus je aplikovatelný ovšem pouze na polygonální překážky. A doplnit jej o práci s nepolygonálními překážkami není úplně triviální. Je to typický představitel buněčné dekompozice.

2.2.2 Morseova dekompozice

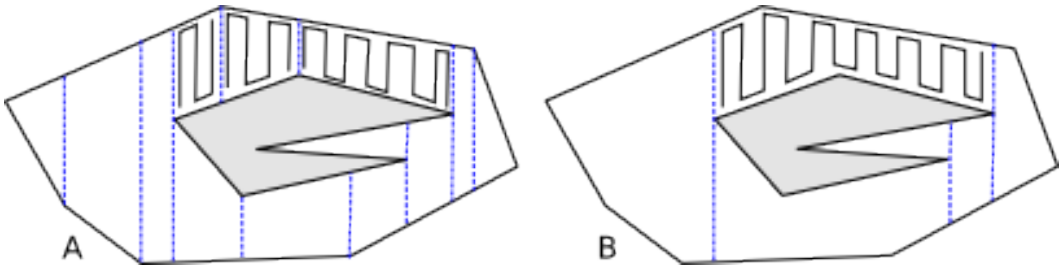
Ne vždy je potřeba hledat cestu spojující dva body, jako jsme to dělali doteď. Morseova dekompozice slouží k hledání cesty, která vykryje celou prohledávanou plochu. Samozřejmě si řeknete, že to by šlo i s Lichoběžníkovou dekompozicí, ovšem Morseova dekompozice najde efektivnější cestu a tím docílíme žádaného výsledku.

Využití tohoto je široké v různých automatických robotech na zametání, kosení trávy nebo mytí podlahy.

2.2.2.1 Algoritmus

Algoritmus vyplňování prostoru je založen na vyplňování podprostorů pohybem vpřed a vzad[1].

Jednou z možností pro toto je takzvaný „Boustrophedon Decomposition“, který rozdělí prostor svislými čárami podobně jako u Lichoběžníkové dekompozice, ale využije pouze vrcholy, u nichž vedou obě hrany buď doleva, nebo doprava. Tímto dojde k optimalizaci a vyplňování je potom



Ilustrace 8: vyplňování prostoru A) rozděleného na lichoběžníky B) rozděleného Boustrophedon Decomposition

efektivnější. Na obrázku (viz Ilustrace 8) je vidět, že vyplňování pomocí tohoto algoritmu je lepší, protože není potřeba provádět tolik dokročení na koncích oblastí.

Morseovu dekompozici můžeme provádět mnoha dalšími způsoby, dekomponovat pomocí soustředných kružnic a hledat tečná místa jako kritické body. Cesta vyplňování je spirálovitá. Mezi další možnosti patří například dekomprese pomocí soustředných čtverců či pomocí tečných přímek a mnoho dalších.

2.2.2.2 Vlastnosti algoritmu Morseovy dekompozice

Algoritmus Boustrophedon Decomposition jakožto hlavní a typický představitel Morseovy dekompozice je určen pro polygonální prostory. Není těžké tento algoritmus upravit, aby pracoval i v prostoru s překážkami obecných tvarů a přesto zůstala zachována jeho výkonnost. Algoritmus je velice rychlý a překvapivě jednoduchý i pro nepolygonální prostory.

2.2.3 Další typy dekompozic

Existuje mnoho dalších typů dekompozice založených například na **grafu viditelnosti** optimalizované pro všechny možné účely.

3 Implementace Java Appletů

Nejprve je nutné rozhodnout se, které dva algoritmy budu chtít demonstrovat. Bylo by vhodné vzít jednoho zástupce Road map a jednoho zástupce buněčné dekompozice. Z algoritmů pro Road mapy je typickým představitelem obecný Voronoiův diagram, ovšem algoritmy pro jeho sestavení a používání mi připadají příliš zřejmé, alespoň co se týče 2D prostoru. Jako zástupce **Road map** jsem se rozhodl implementovat vizualizaci algoritmu pro sestavení **grafu viditelnosti**.

Z algoritmů pro dekompozici prostoru jsem si vybral algoritmus lichoběžníkové dekompozice, který je typickým představitelem. A navíc má podobné potřeby, co se vstupů týče jako předchozí vybraný algoritmus, což je vhodné a může to ušetřit mnoho práce při návrhu a programování.

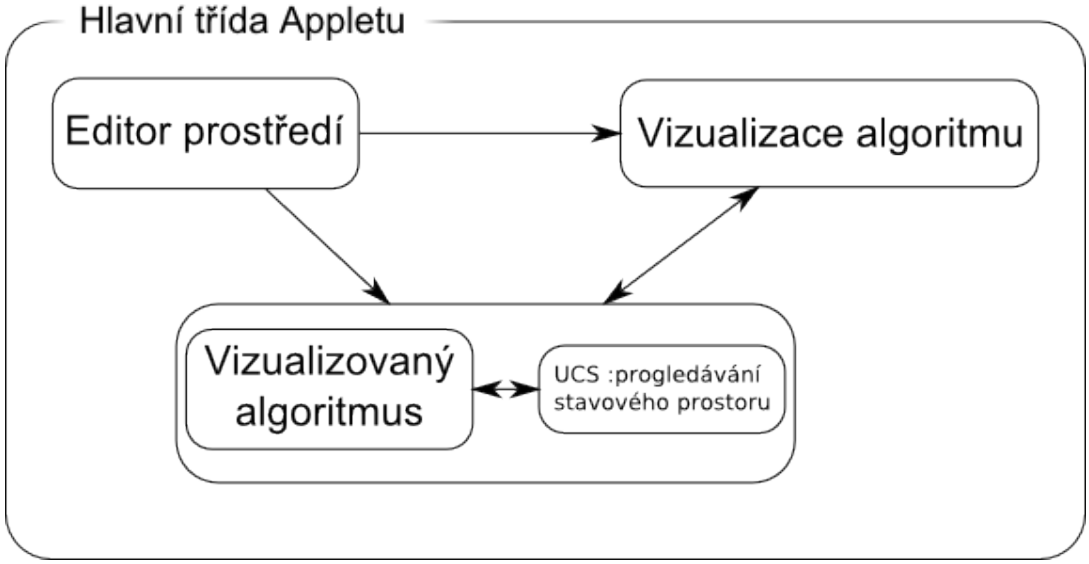
3.1 Analýza

Je potřeba udělat dva Java applety pro vizualizaci algoritmů. Oba algoritmy mají jako vstup polygonální prostor a slouží k vyhledávání cesty mezi 2 body *Start* a *Cíl*. Ačkoli to v zadání není přímo specifikováno, je třeba udělat nějaké prostředí pro sestavení prostoru pro simulaci. Asi by bylo zbytečné dělat Java applet prezentující metodu na nějakém pevně daném prostředí, protože to už by mohl zvládnout klidně i nějaký gif obrázek a navíc tento způsob řešení by neumožňoval experimentovat s danými algoritmy.

Další, co je potřeba udělat, je samozřejmě naprogramovat vybrané algoritmy tak, aby bylo možno je nějak vizualizovat. Samotná vizualizace by měla být jednoduchá a jasně pochopitelná, aby pozorný uživatel byl schopen tyto algoritmy vyzkoušet a pochopit jejich funkci a následně je byl schopen implementovat.

S malou dávkou abstrakce mám udělat applet, který bude provádět simulaci běhu algoritmu. A následně bude schopen postupně zobrazit simulovaný algoritmus, tak aby byl snadno pochopitelný.

3.2 Návrh řešení



Ilustrace 9: Návrh řešení Appletu

Applet se bude skládat z tří samostatných částí:

- **Editační část** bude sloužit k návrhu prostoru pro robota, přijatelné uživatelské rozhraní.
- **Vizualizace** algoritmu bude jednoduše, ale názorně zobrazovat průběh algoritmu s přijatelným uživatelským rozhraním.
- Samotný **algoritmus** pro sestavení mapy bude obsahovat prohledávání stavového prostoru pro možnost hledání výsledné cesty.

První dvě části budou dělány dostatečně obecně, aby zvládly simulaci libovolného algoritmu na 2D prostoru a poslední část bude vytvořená dvakrát: jednou pro algoritmus na sestavení grafu viditelnosti a podruhé pro algoritmus lichoběžníkové dekompozice.

Díky tomu, že program bude tvořen v jazyce Java, který je objektový, bude dělán tak, aby byl applet použitelný i pro další algoritmy. A bylo jej možno využít i na simulaci algoritmů, které popisuji v kapitole 2.

Následuje podrobnější návrh jednotlivých části appletu.

3.2.1 Editační část

V editační části by měl uživatel být schopen vytvořit jakékoliv běžné prostředí pro pohyb robota, například půdorys skladiště, budovy, místnosti s nábytkem a překážkami nebo exteriéru. Protože oba mnou implementované algoritmy mají jako vstup množinu polygonálních překážek, celá editační část bude schopná uživatelem vytvořený prostor převést na množinu polygonů. Tyto polygony jsou odtud předány, jak vizualizační části, tak samotnému demonstrovanému algoritmu.

Celá editační část bude vytvořena jako jeden objekt. Jako vzor pro tvorbu tohoto prvku by bylo vhodné použít nějaký vektorový editor, aby nebylo náročné pro uživatele s ním pracovat. Nyní jde o to, které útvary je potřeba tvořit - minimálně tedy libovolný polygon. Dalším by měl být asi obdélník, protože většina půdorysů staveb je tvořena z pravoúhlých zdí. Mezi další tvary, které by mohly být k dispozici, patří například úsečky, čtverce, pravidelné mnohoúhelníky pro abstrakci kruhových útvarů. Popřípadě by mohly být k dispozici další tvary běžné ve vektorových editorech, ale nejsou podmínkou. Tyto tvary omezíme na tvary, které jsou převoditelné na polygony.

3.2.1.1 Ukládání a načítání vytvořeného prostředí

Po rozvaze a prozkoumání, jak se ukládá obraz ve formátu SVG, což je publikační formát pro ukládání vektorové grafiky ve formátu XML, jsem se rozhodl také používat pro ukládání prostoru do XML, ovšem složitost formátu SVG mě přesvědčila pro použití vlastní DTD.

Problém ovšem nastává v možnostech Java appletů. Java applet ze zásady nesmí přistupovat k souborovému systému, nelze uložit ani načíst soubor z disku. Jedna varianta řešení je naprogramovat na straně serveru další aplikaci pro ukládání souborů (stačí malý skript) a při ukládání soubor poslat po sítí a uložit na serveru, poté by bylo možno tento soubor stáhnout a uložit. Je zde ovšem problém se jmény souborů a také nechat uživatelé ukládat soubory na server je bezpečnostní díra, které by se jistě dalo zneužít. Mnohem jednodušším řešením je při uložení zobrazit XML dokument, který si uživatel pomocí copy & paste uloží, kam je mu libo, a načítat konfiguraci tak, že načítá textový řetězec a uživatel zde vloží dříve vytvořenou konfiguraci prostoru. Tímto způsobem nevzniká bezpečnostní riziko ani problém se jmény souborů ani s postupným zaplněním serveru. Pouze je to nezvyklé řešení, a proto bude uživateli asi posuzováno jako ne příliš uživatelsky příjemné.

3.2.2 Vizualizace algoritmu

Uživatel zde bude schopen krokovat algoritmus několika způsoby nebo nechat algoritmus, aby se krokoval sám. Tento applet bude nejspíš převážně využíván lidmi z oblasti informačních technologií, kteří se budou chtít dozvědět, jak to funguje nebo prozkoumat chování algoritmu

v určitém prostoru, popřípadě prozkoumat vlastnosti samotného algoritmu. Proto jako vzor pro vizualizační část použiji ovládací prvky z ladících programů a umožním krokování algoritmu i v opačném pořadí pro případ, kdy uživatel projde některou částí algoritmu příliš rychle na to, aby ji stihl pochopit, a tak touto částí bude moci projít znova, aniž by musel projít celý algoritmus od začátků, jak tomu je u ladících programů.

Vizualizace algoritmu bude rozdělena na dvě části. **Tabulka instrukcí**, která bude vytvářená ze strany vizualizovaného algoritmu a interpretována, bude v prostředí vizualizace podle toho, jak bude uživatel chtít krokovat. Vzor pro tento návrh jsem nalezl v programech pro simulaci, kde probíhá simulace, jejíž výsledky se nějakým způsobem zaznamenávají. Po skončení simulace dochází k interpretaci výsledků simulace. Podobně tomu bude i zde. Vizualizovaný algoritmus proběhne a vytvoří **tabulku instrukcí**, která již neobsahuje podmínky a cykly, ale pouze lineární pořadí konkrétních instrukcí pro dané prostředí. Interpretací těchto výsledků bude docházet k vizualizaci daného algoritmu.

Možná množina instrukcí by mohla být například: zobrazit prvek, skrýt prvek, zvýraznit prvek a zrušit zvýraznění prvku. Vidíme, že navržené instrukce jsou v komplementárních párech, aby se dalo krokovat pozpátku algoritmem. Pokud krokujeme v opačném pořadí, provádí se opačná instrukce komplementárního páru. Například narazíme-li na instrukci Skrýt prvek, provedeme jeho zobrazení, protože před touto instrukcí byl prvek jistě viditelný, pokud se má skrývat.

Chceme-li, aby algoritmus a výkonné jádro programu byly úplně nezávislé na grafickém uživatelském rozhraní, tak vizualizační část bude implementovat rozhraní, díky kterému se instrukce budou moct zadávat do tabulky a taky vytvářet jednotlivé zobrazitelné tvary, které se mohou pro algoritmy lišit. Mezi základní **tvary vizualizace** patří:

- **úsečka,** která slouží buď jako vysvětlující úsečka, nebo jako hrana sestavovaného grafu reprezentujícího mapu prostředí pro vyhledávání
- **bod**, který slouží jako orientační bod vysvětlující algoritmus nebo jako uzel grafu

Ale mohou sem patřit i další podle potřeby vizualizovaných algoritmů a požadované názornosti vizualizace.

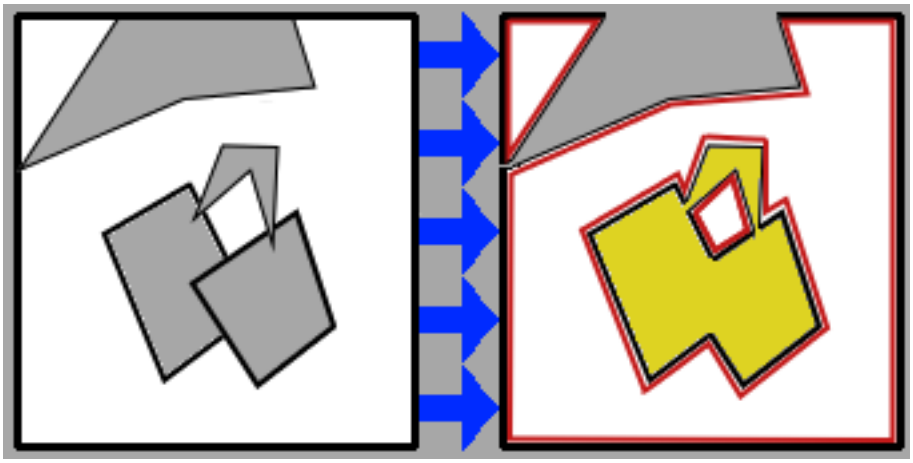
3.2.3 Vizualizovaný algoritmus

Algoritmus naprogramuji tak, aby správně fungoval, a doplním jej o příkazy pro vizualizaci. Normálně proběhne a bude generovat do vizualizační části tabulku instrukcí pro následnou vizualizaci. V této části také bude docházet ke spojování polygonů.

3.2.3.1 Spojování polygonů

Spojováním polygonů je myšleno množinu polygonů převést na množinu polygonů, z nichž žádné dva polygony se neprotínají. Každý polygon má vrcholy seřazeny po směru hodinových ručiček okolo prostoru, který omezuje (Ilustrace 7).

Na následujícím obrázku (Ilustrace 10) vidíme, jaký je požadovaný výsledek spojování polygonů, spojením čtyř polygonů vznikne jeden polygon, který má prvky po směru hodinových ručiček (ten žlutý) a čtyři polygony opačné (červeně označené), které omezují 3 prostředí (souvislé bíle plochy, ve kterých se robot může pohybovat).



Ilustrace 10: Sloučení 4 polygonů

```
přidejPolygon(Množina kam , Polygon přidávaný){
      if(přidávaný neprotíná nic v množině kam){
            kam.add(přidávaný)
      }else{
            Množina protnuto=vše z kam co protíná přidávaný
            kam.removeAll(protnuto)
            Množina spojeno
            foreach(polygon z množiny protnuto){
                  spojeno.add(spojeni(polygon,přidávaný))
            }
            foreach(polygon z množiny spojeno){
                  přidejPolygon(kam , polygon)
            }
      }
}
```

*Algoritmus 3: slouží k přidávání polygonu do výsledné množiny tak, aby v množině „**kam**“ byly po skončení této funkce požadované polygony.*

Kýženého výsledku dosáhneme, pokud pro každý polygon provedeme rekurzivní funkci (viz. Algoritmus 3). Tato funkce zkontroluje, zda se s něčím kříží a pokud ano, tak zavolá funkci pro spojení 2 polygonů, kterou jsem udělal podobně, jak to dělá Weiler-Atherton algoritmus.

3.2.3.2 Geometrické operace

Pro vizualizované algoritmy je potřeba mnoho geometrických funkcí, například pro hledání průsečíku 2 přímek nebo úseček, určování směrnice, výpočet úhlu 2 přímek, ...

Pro tento účel jsem se rozhodl používat knihovnu Geometry.jar, která mnoho z tohoto umí a jistě to má vyřešeno optimálně. Avšak některé věci neumí, a proto si je budu muset dodělat.
Problém je zjistit, zda jsou body polygonu po směru hodinových ručiček vzhledem k vnitřní části polygonu. Řešením je funkce, která sečte pravé uhly mezi následujícími obvodovými úsečkami polygonu a toto číslo porovná s přímým uhlem vynásobeným počtem vrcholů polygonů. Pokud je tento součet menší, pak vrcholy polygonu jsou po směru hodinových ručiček a naopak.

3.2.3.3 Graf viditelnosti

Pro sestrojení grafu viditelnosti budeme muset vytvořit všechny funkce popsané v kapitole 2.1.4. O těchto funkcích si budete moci více přečíst v programové dokumentaci. Z grafických objektů si plně vystačíme s úsečkami a uzly grafu.

3.2.3.4 Lichoběžníková dekompozice

Pro lichoběžníkovou dekompozici bude potřeba nějak rychle a správně nacházet průsečíky přímek. Dále bude potřeba při vizualizaci grafická reprezentace lichoběžníku. Kvůli obecnosti a možnosti opakovaného použití nebudu dělat grafický prvek lichoběžník ale polygon, se kterým se v jazyce Java lépe pracuje a je pro něj již mnoho předpřipravených funkcí.

3.3 Popis řešení

Řešení se ve valné části drží návrhu a výsledkem jsou tři samostatné části, z nichž každá plní svůj účel, pro který byla navržená. Protože algoritmus pro sestavení grafu viditelnosti i přes veškeré snahy optimalizovat rychlost výpočtu trvá neúměrně dlouho, rozhodl jsem se, že tento algoritmus bude probíhat ve vlastním threadu. Je jasné, že toto řešení nebylo úplně triviální, bylo nutno vyřešit všechny problémy, které sebou multithreadové programování přináší. Naštěstí jazyk Java je pro multithreadové programování připraven výborně.

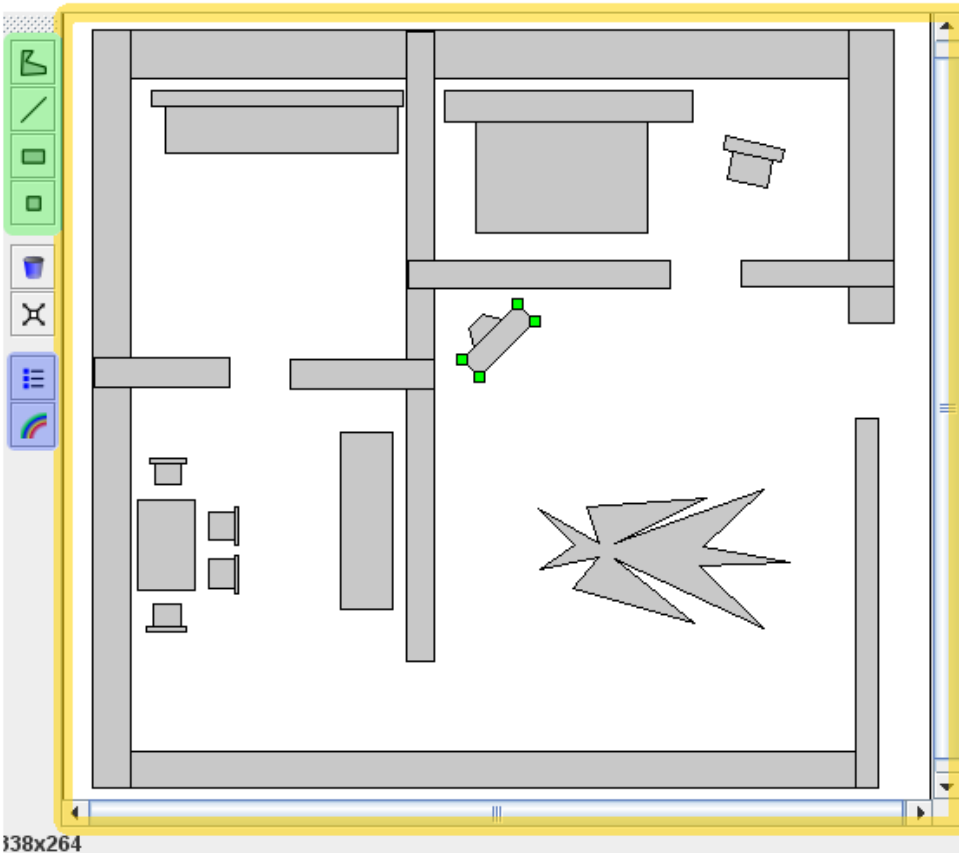
3.3.1 Komunikace jednotlivých částí programu

Komunikuje se přes rozhraní, což zajišťuje možnou rozšiřitelnost programu vytvořením nové třídy implementující toto rozhraní a použitím této třídy místo stávající třídy. Tím například můžeme dosáhnout toho, aby editační část běžela jako konzolová aplikace místo jako vektorový editor a jako editor použít nějaký externí program.

Ačkoli byla navržena komunikace mezi editační a vizualizační částí, tak tuto část zajišťuje hlavní třída appletu (třída `IBPApplet`), tato třída zároveň ustavuje vazby mezi třemi základními částmi appletu: editor prostředí (třída `Editor`), vizualizací algoritmu (třída `Vizualizer`) a samotný algoritmus (třídy `VisibilityGraphs`, `TrapezoidalDecompozition` odvozené od `Robot-MoverAlgoritm`).

3.3.2 Editor prostředí

Editor prostředí je reprezentován třídou `Editor`. Editor je prvkem grafického uživatelského rozhraní.



Ilustrace 11: Editor prostředí pro následnou vizualizaci algoritmu

Na Obrázku je vidět klíčová část appletu a tou je mapa (viz Ilustrace 11 označená žlutým rámečkem a v programu reprezentovaná třídou Mapa), ve které jsou vytvářeny a editovány jednotlivé

tvary. Množinu tvarů, které lze vykreslit, jsem omezil na polygon, přímku, obdélník a čtverec (zelenou barvou označená tlačítka). Stále lze vykreslit celkem zajímavé prostředí s velkou členitostí a vcelku podobné něčemu reálnému, například půdorysu domku nebo jinému místu, popisovanému v dřívějších kapitolách.

V editoru lze také například nastavit barvy, což je děláno pomocí statické třídy, jejíž proměnné lze měnit přes připravený dialog. Tyto proměnné jsou používány pro vykreslování v mapě a to jak při editaci, tak i při vizualizaci. Dále lze měnit velikost mapy, parametry simulace. Ukládání je implementováno podle návrhu pomocí copy & paste.

Tato část programu napodobuje vektorový editor, ovšem ovládání je omezeno pouze na ty prvky, na které jsem v podobných aplikacích narazil, takže to asi nebude úplně uživatelsky přívětivá oblast.

3.3.3 Vizualizovaný algoritmus

Každý vizualizovaný algoritmus je implementován v samostatné třídě a tyto třídy jsou odvozeny od abstraktní třídy `RobotMoverAlgoritm`, která obsahuje metody používané v obou vizualizovaných algoritmech a provádí přípravu prostoru spojováním polygonu. Při spojování polygonů zároveň doplňuje informaci, zda obsah překážky omezené tímto polygonem je uvnitř nebo vně tohoto polygonu.

Oba algoritmy jsou proloženy instrukcemi pro vizualizaci, a proto jejich průběh není tak efektivní, jak by měl být u vyladěného algoritmu. Příklad zdrojového kódu, který to prakticky ukazuje (viz Kód 1). Všechny řádky začínající „gui“ v Kód 1 jsou pouze pro generování tabulky instrukcí. Místy se objevují i cykly, které slouží pouze pro generování těchto instrukcí, a proto jsou tyto algoritmy v mém podání neefektivní, ale dobře vizualizované.

```
gui.start("for pro obě hrany z uzlu v");
      for(int i=0;i<2;i++){
      index=L.indexOf(Vedge[i]);
      gui.start("if hrana je v seznamu L");
      if ((index!=-1)){//jiz je v seznamu
            gui.start("hrana je v seznamu => odebrat");
            currentEdge=L.get(index);
            gui.hide(currentEdge.guiEdge);
            add[i]=false;
            L.remove(index);
            gui.end("hrana je v seznamu => odebrat");
      }else{
            gui.start("hrana není v seznamu=>přidat");
            Vedge[i].guiEdge=gui.createLine(Vedge[i].from.x,
                                            Vedge[i].from.y,
            Vedge[i].to.x, Vedge[i].to.y);
            sortedAdd(V.x,L,Vedge[i]);
            add[i]=true;
            up[i]=!isStartVertex(V, Vedge[i]);
            gui.end("hrana není v seznamu=>přidat");
      }
      gui.end("if hrana je v seznamu L");
      }
gui.end("FOR pro obě hrany z uzlu v");
```

Kód 1: část zdrojového kódu z Lichoběžníkové dekompozice pro ilustraci množství příkazu pro Vizualizaci

3.3.3.1 Implementace Grafu viditelnosti

Nejprve je vytvořena množina všech uzlů grafu tj. vrcholů polygonu a ty jsou zobrazeny. Poté se podle Algoritmus 1 naleznou všechny hrany grafu viditelnosti a to buď rovnou minimalizované, nebo ne podle parametrů simulace. Potom se čeká, než uživatel zadá odkud, kam chce cestu nalézt. Následně je tento graf prohledán pomocí třídy UCS.java (**Uniform-cost-search** informované prohledávání stavového prostoru) a nejkratší cesta je nalezena.

Klíčový bod je zde algoritmus nalezení **viditelných hran**. Tento algoritmus je popisován již dříve v této práci a také na mých webových stránkách věnovaných této problematice.

3.3.3.2 Implementace Lichoběžníková dekompozice

Prvním krokem lichoběžníkové dekompozice je seřazení prvků vrcholů všech polygonů vzestupně podle x-ové souřadnice. Následuje cyklus přes všechny tyto vrcholy v pořadí daném v prvním kroku. U každého vrcholu jsou vytvořeny nové lichoběžníky (jeden až dva) a předpřipraveny jeden až dva další lichoběžníky pro dokončení u jiných vrcholů.

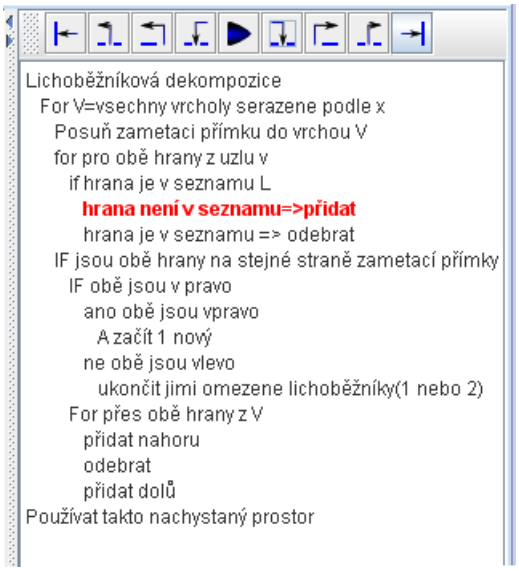
Pro tento algoritmus využívám seznamu vrcholů, množiny hotových lichoběžníků a množiny předpřipravených lichoběžníků.

3.3.4 Vizualizace algoritmu

Vizualizace algoritmu je reprezentována třídou `Vizualizer`. Ta je prvkem grafického uživatelského rozhraní. Při vizualizaci algoritmu je také klíčovou a dominantní částí mapa prostředí. Je to stejná komponenta jako v editoru prostředí, ovšem je v jiném módu. Není možno do ní vytvářet nové překážky. Pouze je možno vytvářet objekty vizualizace, které jsou popsány již v návrhu. Všechny překážky zde jsou vykreslovány najednou jako jediný tvar a tedy velice efektivně.

Neméně důležitou částí okna při vizualizaci algoritmů je pravý panel, kde nalezneme ovládací tlačítka simulace, a pod ním je vypsaný simulovaný algoritmus, kterým lze krokovat, jak již bylo popsáno dříve.

Jak je vidět struktura algoritmu je daná odsazením, nikoli závorkami. Pro krokování máme



Ilustrace 12: vypsaný algoritmus a ovládací prvky vizualizace algoritmu

mnoho možností, to jsou ta tlačítka nahoře, funkce tlačítek je následující:

 Vrací se do počátečního stavu (na řádek Lichoběžníková dekompozice).

 Vrací se zpět, dokud se nevynoří na vyšší úroveň (např. vynoření z cyklu).

 Vrací se zpět bez zanořování, to znamená skok na nejbližší předchozí řádek stejné nebo vyšší úrovni. Z Pozice na Ilustrace 12 skočí na řádek 5, ale z řádku 14 by skočil na řádek 9).

 Vrátí se zpět o jeden řádek včetně zanořování.

 Spustí automatické krokování a změní se na znak pauzy, kterým se automatické krokování vypíná. Pokud krokujeme automaticky, lze pomoci kolečka myši nad tímto tlačítkem měnit rychlost krokování.

 O jeden řádek vpřed, nejjemnější krokování.

 Vpřed na další řádek na stejné nebo vyšší úrovni.

 Vpřed až do vynoření lze používat k dokončení nějakého dlouhého cyklu.

 Nechá algoritmus dojít v mžiku až dokonce a zobrazí výsledek.

Při krokování algoritmu je uživatel občas vyzván k zadání bodu podle toho, jaký bod je potřeba, buď cílový bod, nebo startovní bod. Toto je řešeno tak, že uspí vlákno výpočtu a jakmile uživatel zadá bod, tak je probuzeno a pokračuje se ve výpočtu.

4 Závěr

Tato práce měla za úkol prezentovat algoritmy pro hledání cesty robotem, konkrétně skupiny algoritmů **Road mapy** a **Buněčná dekompozice** jak jsou popsány v knize „Principles of robot motion: theory, algorithms and implementations“

4.1 Výsledek

Byly vytvořeny webové stránky prezentující tyto algoritmy, tak aby čtenář na těchto stránkách získal ucelený přehled o tom, které algoritmy se řadí mezi **Road mapy** a **Buněčná dekompozice**, jak tyto algoritmy fungují, a jaké mají vlastnosti ve smyslu rozsahu použitelnosti, rychlosti, …

Dále byly vytvořeny dva demonstrační Java applety umožňující prozkoumání a experimentování s vybraným algoritmem. První Java applet představuje algoritmus na **sestavení grafu viditelnosti** jako představitele Road map. Druhý Java applet představuje algoritmus na rozdělení prostoru pomocí **Lichoběžníkové dekompozice,** jakožto představitele Buněčné dekompozice.

Při tvorbě těchto appletů vzniklo také prostředí na simulaci podobných algoritmů, ve kterém by v budoucnu mohly být simulovány, jak doufám další algoritmy nejen z oblasti navigace robotů, ale obecně z oblasti geometrie. Vnitřní struktura pro rozšiřování tohoto je popsána v **API dokumentaci** vygenerované ze zdrojových kódů.

4.2 Další vývoj

Mohly by se udělat další algoritmy pro vizualizaci ostatních metod s využitím stejného vizualizačního prostředí a doplnit je do webové prezentace. Také by se mohly důkladně prozkoumat algoritmy, o kterých zde byla jenom zmínka. Popřípadě doplnit o další algoritmy, které jsem zde vůbec nebral v úvahu či další skupiny algoritmů.

Jistě by bylo vhodné spojit tuto práci s dalšími, které se zabývají skupinami algoritmů, jako jsou Bug algoritmy, Potenciálová pole a další, a udělat jim jednotné webové rozhraní a jednotný způsob vizualizace. V takovém případě by se dalo uvažovat o rozšíření mnou navrženého vizualizačního rozhraní o další vizualizační prvky.

Editor prostředí v mých appletech by zasloužil vylepšit popřípadě vyměnit za nějaký jiný, lepší uživatelky příjemnější.

Nejlepší další rozvoj, který by mi udělal radost, by bylo prakticky ukázat na reálném robotovi, že tyto algoritmy jsou skutečně funkční a připraveny pro nasazení na řešení reálných úkolů.

Literatura

[1] Choset, Howie a kol., Principles of robot motion :theory, algorithms and implementations / Massachusetts : MIT Press, 2005. xix, 603 s. : il. ISBN 0-262-03327-5

[2] Herout, Pavel. Java :grafické uživatelské prostředí a čeština / 1. vyd. České Budějovice : Kopp, c2001. 316 s. ISBN 80-7232-237-0

[3] Herout, Pavel. Učebnice jazyka JAVA / 1. vyd. České Budějovice : KOOP, 2001. 349 s. ISBN 80-7232-115-3

[4] Wikipedia, The Free Encyclopedia [online], url: http://en.wikipedia.org(6.5.2008)

[5] Wikipedie, otevřená encyklopedie [online], url: http://cs.wikipedia.org(6.5.2008)

[6] Java Tutorials [online], url: http://java.sun.com/docs/books/tutorial/ (5.5.2008)

[7] Java API dokumentace [online], url:http://java.sun.com/j2se/1.4.2/docs/api/ (6.5.2008)

Seznam příloh

Příloha 1. CD obsahující webovou prezentaci vybraných algoritmů Java applety včetně jejich zdrojových souborů a API dokumentace.