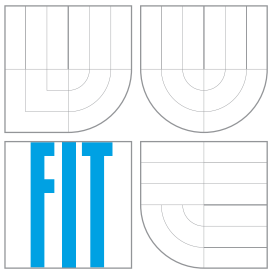
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VIZUALIZÁCIA PLÁNOVANIA CESTY PRE LIETADLÁ

VISUALISATION OF PATH-PLANNING FOR AIRCRAFTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MILOŠ VOLNÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAROSLAV ROZMAN, Ph.D.

BRNO 2016

Zadání bakalářské práce/14516/2015/xvolny02

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Volný Miloš**
Obor: Informační technologie
Téma: **Vizualizace plánování cesty pro letouny**
Visualisation of Path-Planning for Aircrafts
Kategorie: Umělá inteligence

Pokyny:

1. Seznamte se s metodami pro plánování cesty pro letadla.
2. Proveďte rešerši používaných metod a dostupných implementovaných algoritmů pro plánování cesty pro letadla.
3. Navrhněte applet, ve kterém demonstrujete funkčnost vybraných metod.
4. Applet implementujte jako součást webových stránek. Webové stránky doplňte teorií k jednotlivým metodám. Navrhněte případná rozšíření a vylepšení.

Literatura:

- Howie Choset et al., Principles of Robot Motion, 2005, ISN 0-262-03327-5

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Rozman Jaroslav, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Táto práca sa zaoberá preskúmaním základných prístupov k plánovaniu ciest a ich aplikácii na problémy plánovania cesty pre lietadlá. Práca diskutuje výber plánovacích algoritmov vhodných na uvedenie čitateľa do problematiky plánovania ciest pre lietadlá a ich následnú implementáciu. Cieľom práce je vytvorenie appletu vizualizujúceho fungovanie vybraných algoritmov a jeho umiestnenie na web stránky poskytujúce teoretický popis týchto algoritmov.

Abstract

This thesis compiles basic methods of path planning and their application to problems involving path planning for aircraft. Work discusses selection of algorithms suitable for introduction of aircraft-oriented path planning to the reader and their subsequent implementation. Main objective of the thesis is to create an applet capable of visualisation of chosen algorithms and its placement onto a website, which provides theoretical explanation of said planning algorithms.

Kľúčové slová

plánovanie cesty, vizualizácia, lietadlo, RRT, Rapidly-Exploring Random Trees, Naivná bunková dekompozícia

Keywords

path planning, visualisation, airplane, RRT, Rapidly-Exploring Random Trees, Naive cell decomposition

Citácia

VOLNÝ, Miloš. *Vizualizácia plánovania cesty pre lietadlá.* Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Rozman Jaroslav.

Vizualizácia plánovania cesty pre lietadlá

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Ing. Jaroslava Rozmana, Ph.D. a uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Miloš Volný
17. mája 2016

Poďakovanie

Chcel by som sa poďakovať Ing. Jaroslavovi Rozmanovi, Ph.D. za jeho rady, postrehy a vedenie mojej bakalárskej práce.

Obsah

Kapitola 1

Úvod

Táto práca sa zaoberá základnými metódami plánovania cesty pre lietadlá v 3D priestore a ich vizualizáciou. Práca skúma použitie známych algoritmov plánovania cesty pre roboty v 2D a ich použiteľnosť v 3D neholonomickom plánovaní pre lietadlá. Hlavným cieľom tejto práce je vytvorenie appletu vizualizujúceho vybrané algoritmy plánovania ciest pre lietadlá. Applet bude súčasťou web stránky, kde bude pomocou vizualizácie demonštrovať fungovanie vybraných algoritmov, ktoré budú na tejto web stránke aj vysvetlené. Zámerom tejto stránky a appletu je priblížiť filozofiu plánovania ciest pre lietadlá a fungovanie vybraných algoritmov.

Obsahom tejto práce sú kapitoly pojednávajúce o teórii plánovacích algoritmov, reprezentácii pohybu robota a reprezentácii prostredia, v ktorom plánovanie prebieha. Práca je štruktúrovaná následovne:

V prvej kapitole sú predstavené základné pojmy z problematiky plánovania ciest, ako aj spôsob zachytenia pohybových možností robotov, konkrétne lietadiel.

Druhá kapitola poskytuje čitateľovi prehľad základných metód plánovania ciest rozdelených na metódy so znalosťou prostredia a na vzorkovacie metódy. V tejto kapitole sú tiež uvedené príklady prác, v ktorých boli dané metódy, alebo metódy z nich odvodené, použité na plánovanie ciest pre lietadlá. Taktiež sú tieto metódy v krátkosti diskutované z hľadiska použitia v tejto práci.

V poslednej kapitole je popísaná implementácia vybraných algoritmov a ich vizualizácie, ako aj použitie samotného appletu. Na záver sú v tejto kapitole obsiahnuté výsledky testovania appletu.

Kapitola 2

Plánovanie cesty

V úvodnej kapitole budú v krátkosti predstavené základné pojmy z problematiky plánovania ciest. Ide hlavne o pojmy týkajúce sa pracovného prostredia robota, jeho pohybu a rôznych typov plánovania cesty. Tieto definície sú parafrázované z prác Choset [2] a LaValle [11].

2.1 Pohyb robota

Predpokladáme, že robot je pevné teleso definované ako podmnožina dvoj alebo trojrozmerného priestoru. Potom pohyb takéhoto robota reprezentujeme pomocou transformácií, ktoré sú funkciami $h : A \to W$, ktoré mapujú všetky body robota do pracovného priestoru, pričom zachovávajú vzdialenosť medzi každým párom bodov ako aj ich orientáciu.

2.1.1 Transformácie v 2D priestore

Základnými transformáciami v 2D priestore sú **translácia** a **rotácia**.

Translácia je funkcia h posunu v rovine s dvoma parametrami $x_t, y_t \in \mathbb{R}$ definovaná ako

$$h(x, y) = (x + x_t, y + y_t) \tag{2.1}$$

Každá translácia sa dá interpretovať dvoma spôsobmi - ako translácia robota alebo ako translácia sústavy koordinátov. V ďalších kapitolách budeme pre maximálny počet nezávislých parametrov potrebných pre úplnú charakteristiku aplikovanej transformácie požívať termín **stupne voľnosti**.

Rotácia robota $\mathbb{A}$ o uhol $\theta \in (0, 2\pi)$ [1] je namapovanie všetkých bodov $(x, y) \in \mathbb{A}$ ako

$$(x, y) \to (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta) \tag{2.2}$$

Alebo zapísané s použitím 2x2 rotačnej matice

$$R(\theta) = \begin{pmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{pmatrix} \tag{2.3}$$

[1] V tejto práci budeme všetky rotácie uvažovať v protismere hodinových ručičiek

2.1.2 Transformácie v 3D priestore

Koncept transformácii v 3D priestore je podobný tomu v 2D, avšak rotácie v sú 3D komplikovanejšie.

3D translácia Posun v 3D priestore je analogický k 2D translácii. Je definovaný parametrami $x_t, y_t, z_t \in \mathbb{R}$ ako

$$(x, y, z) \rightarrow (x + x_t, y + y_t, z + z_t) \tag{2.4}$$

3D rotácie 3D objekt môže rotovať okolo troch ortogonálnych ôs. Pre potreby tejto práce je vhodné si zapožičať leteckú terminológiu a tieto rotácie nazývať *klopenie, zatáčanie* a *klonenie* .

1. Klonenie (pitch) je rotácia o α stupňov okolo bočnej horizontálnej osi lietadla. Môžeme ju zapísať pomocou matice nasledovne:

$$R_z(\alpha) = \begin{pmatrix} cos\alpha & -sin\alpha & 0 \\ sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.5}$$

2. Zatáčanie (yaw) je rotácia o β stupňov okolo vertikálnej osi lietadla. Môžeme ju zapísať pomocou matice nasledovne:

$$R_z(\beta) = \begin{pmatrix} cos\beta & 0 & sin\beta \\ 0 & 1 & 0 \\ -sin\beta & 0 & cos\beta \end{pmatrix} \tag{2.6}$$

3. Klopenie (roll) je rotácia o γ stupňov okolo pozdĺžnej osi lietadla. Môžeme ju zapísať pomocou matice nasledovne:

$$R_z(\gamma) = \begin{pmatrix} 1 & 0 & 0 & \\ 0 & cos\gamma & -sin\gamma & \\ 0 & sin\gamma & 0 & cos\gamma \end{pmatrix} \tag{2.7}$$

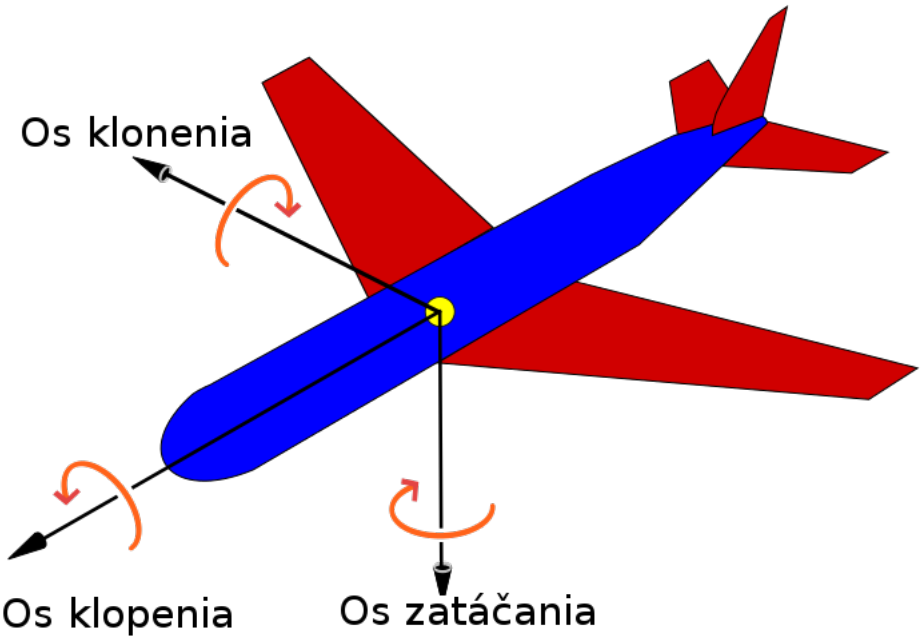
2.2 Prostredie

Pre potreby plánovania cesty je dôležité definovať stavový priestor, v ktorom sa bude robot pohybovať. N-rozmerný Euklidovský priestor, v ktorom sa nachádza robot a možné prekážky, označujeme ako **pracovný priestor**. Potom množinu všetkých konfigurácií [2] daného systému v pracovnom priestore nazývame **konfiguračný priestor**.

2.2.1 Konfiguračný priestor

Konfiguračný priestor slúži na matematický popis a namapovanie pozície a orientácie robota a prekážok do pracovného priestoru. Pre potreby detekcie kolízií rozlišujeme tri podmnožiny konfiguračného priestoru.

[2] konfigurácia je kompletná špecifikácia pozície všetkých bodov systému



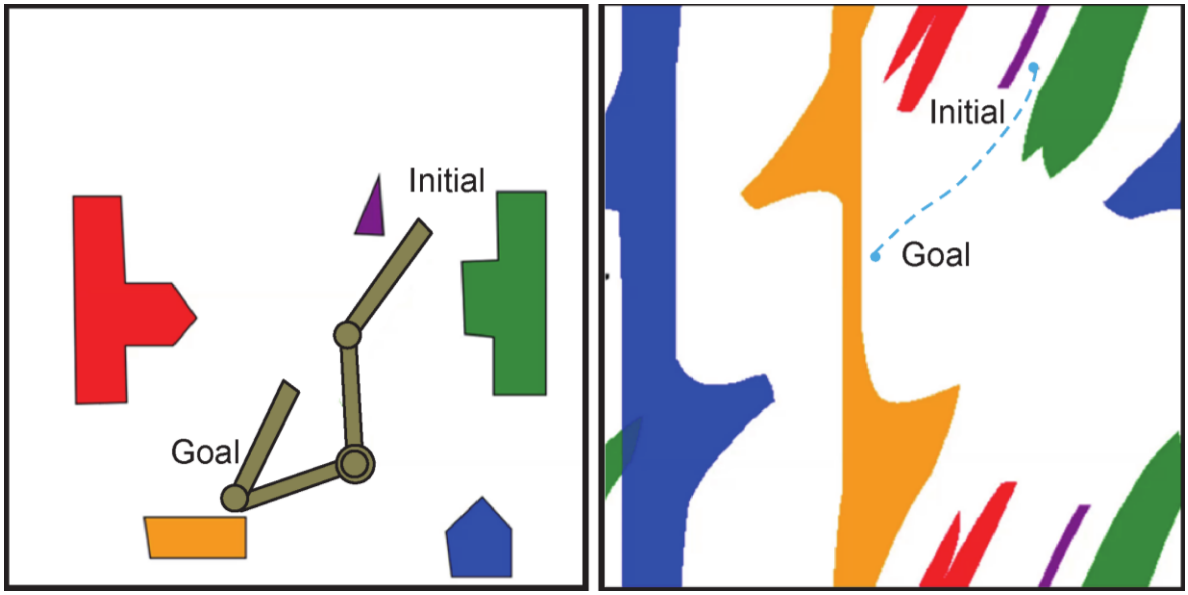
Obr. 2.1: Obrázok modelu lietadla s vyznačenými osami možných rotácií, zdroj: Wikipédia [18]

1. C_{obst} - množina konfigurácií, v ktorých dôjde ku kolízii medzi robotom a prekážkou
2. C_{free} - množina konfigurácií, v ktorých nedôjde ku kolízii medzi robotom a prekážkou, táto množina je komplementárna k C_{obst}
3. $C_{semi-free}$ - množina konfigurácií, ktorá povoľuje kontakt medzi robotom a hranicou prekážky

2.2.2 Reprezentácia pohybu robota

Pozícia robota v konfiguračnom priestore je úplne daná n hodnotami, kde n je počet stupňov voľnosti robota. V prípade lietadla v 3D priestore je počet stupňov voľnosti 6 - 3 rotačné a 3 translačné. Medzi často používané reprezentácie pozície v 3D priesotre patria Eulerova metóda, metóda quarterniónov a metóda smerových kosínov. Podrobnejší prehľad týchto metód je diskutovaný v Cooke [3].
Vlastnosti robota, pre ktorého je cesta plánovaná, zásadne ovplyvňujú náročnosť plánovacieho problému. Z hľadiska obmedzení robota sa preto delí plánovanie ciest na:

- Holonomické plánovanie - holonomické obmedzenia sa dajú vyjadriť ako funkcie konfiguračných premenných a prípadne času. Každé lineárne nezávislé holonomické obmedzenie znižuje dimenziu problému o jeden stupeň.
- Neholonomické plánovanie - neholonomické obmedzenia sú obmedzenia týkajúce sa rýchlosti robota, tieto neznižujú dimenziu problému



Obr. 2.2: Naľavo: pracovný priestor robota typu otočné rameno s vyznačenou počiatočnou a koncovou konfiguráciou.
Napravo: ten istý priestor zobrazený s ohľadom na stupne voľnosti robota. Farebne vyznačené objekty zodpovedajú prekážkam rovnakej farby, výsledná cesta je zobrazená modrou prerušovanou čiarou Zdroj: [9]

- Plánovanie s kinodynamickými obmedzeniami - kinodynamické obmedzenia obsahujú okrem neholonomických obmedzení aj obmedzenia zmeny rýchlosti robota

V rámci tejto práce budeme uvažovať o neholonomickom plánovaní vzhľadom na kinematický model lietadiel. Na zachytenie kinematiky lietadiel používajú Cheng [1] a Yang [20] tzv.*pohybové rovnice*. Kinematika lietadiel je nimi popísaná nasledovne:

$$\begin{aligned} \frac{dx}{dt} &= V\cos\gamma\cos\psi \\ \frac{dy}{dt} &= V\cos\gamma\sin\psi \\ \frac{dz}{dt} &= V\sin\gamma \\ \frac{d\psi}{dt} &= g\tan\frac{\phi}{V} \end{aligned} \tag{2.8}$$

V tomto kontexte sú x, y, z koordináty lietadla, V jeho rýchlosť, γ je uhol o ktorý zatáča cesta, ψ uhol smeru lietadla, ϕ uhol klopenia lietadla a g gravitačné zrýchlenie.

Dynamické vlastnosti lietadiel sú vo Weitz [17] popísané rovnicami:

$$\begin{aligned} \frac{dV}{dt} &= \frac{T-D}{m} + g\sin\gamma \\ \frac{d\gamma}{dt} &= -\frac{L\cos\phi}{mV} + \frac{g\cos\gamma}{V} \\ \frac{d\psi}{dt} &= -\frac{L\sin\phi}{mV\cos\gamma} \end{aligned} \tag{2.9}$$

Tu T značí ťah lietadla, D je odpor vzduchu, L je vztlak a m je hmotnosť lietadla. Ostatné značenie je rovnaké ako v kinematických rovniciach 2.8. Môžeme si všimnúť, že v prípade kinematických vlastností lietadiel sa počíta iba so zjednodušenou rovnicou pre zmenu uhlu smeru lietadla ϕ.

Kapitola 3

Prehľad prístupov k plánovaniu ciest

Algoritmy plánovania cesty pre jednoduchých mobilných robotov sú v zásade použiteľné aj pre plánovanie ciest pre lietadlá. Je však potrebné vziať na zreteľ spomínané obmedzenia, ktorým lietadlá podliehajú, a aj zámer tejto práce plánovať a vizualizovať cestu v 3D. Nasledujúca kapitola popisuje vybrané prístupy k plánovaniu ciest, ktoré autor považuje za vhodné zohľadniť pre použitie v tejto práci. Hlavnými kritériami pre výber boli jednoduchosť, frekvencia použitia a dobrá možnosť vizualizácie. Kapitola napríklad nepojednáva o genetických algoritmoch, nakoľko ich vizualizácia by bola problematická a pre používateľa pravdepodobne ťažko zrozumiteľná, aj keď sú pomerne používané, napr. Őzalp [21]. Nižšie uvedené podkapitoly čerpajú z LaValle [11], Choset [2] a článku Goerzen [7], ktoré poskytujú všebecný prehľad o problematike plánovania cesty, ako aj z odborných článkov pojednávajúcich o konkrétnych použitiach algoritmov, zväčša pre potreby plánovania cesty pre bezpilotné lietadlá. Citácie k týmto článkom sú uvedené v relevantných odstavcoch.

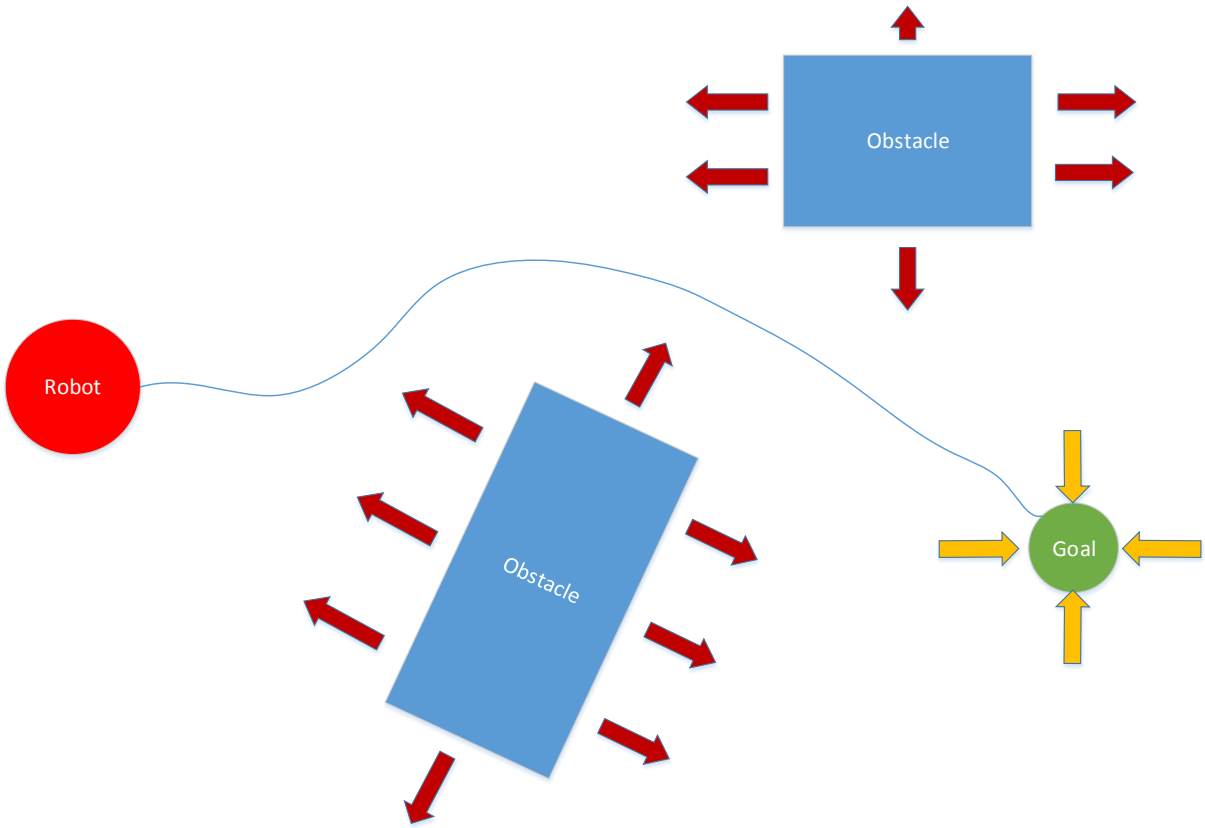
3.1 Metódy so znalosťou prostredia

Nasledujúce metódy vyžadujú znalosť rozmiestnenia prekážok, ešte pred zahájením plánovania cesty, pre konštrukciu reprezentácie prostredia, v ktorej následne hľadajú cestu pre robota.

3.1.1 Potenciálne funkcie

Potenciálna funkcia je diferencovateľná funkcia $U : \mathbb{R}^m \to \mathbb{R}$, s ktorej hodnotou môžeme pracovať ako s energiou a teda jej gradientom potenciálu je sila. V tomto prístupe sa používa gradient na definovanie vektorového poľa gradientov, ktoré priraďuje každému bodu priestoru vektor gradientu nejakej funkcie. Následne potenciálna funkcia navádza robota, ako by bol časticou pohybujúcou sa vo vektorovom poli gradientov. Robot je priťahovaný k cieľu a odpudzovaný prekážkami. Kombináciou týchto príťažlivých a odpudivých síl je robot navigovaný okolo prekážok k cieľu. V podstate ide o sledovanie obráteného gradientu potenciálnej funkcie, čo nazývame aj *gradientovým zostupom*. K zastaveniu gradientového zostupu dochádza pri dosiahnutí tzv. *kritického bodu* - bod kde gradient zaniká. Kritický bod je zpravidla maximom, minimom alebo sedlovým bodom potenciálnej funkcie. Keďže robot postupuje gradientovým zostupom, nemôže sa dostať do maxima, pokiaľ v ňom nezačína (v takom prípade je potrebné ho akýmkoľvek pohybom vyslobodiť). Podobne aj sedlové

body sú nestabilné, a teda robot zpravidla zastaví iba v minime, ktoré je stabilné (po ľubovoľnom pohybe vráti robota gradientový zostup späť do lokálneho minima). Problémom tohto prístupu je možný výskyt lokálnych miním, ktoré nie sú cieľovým bodom, čo môže spôsobiť predčasné ukončenie algoritmu. V Choset [2] sú navrhnuté dva prístupy, ktorými sa tento problém dá riešiť: prvý prístup vylepší potenciálne pole o plánovač založený na vyhľadávaní, ten druhý definuje potenciálnu funkciu s jedným lokálnym minimom, ktorá sa nazýva *navigačná funkcia*.



Obr. 3.1: Metóda potenciálnych funkcií: Na obrázku môžeme vidieť červeného robota, ktorý hľadá cestu okolo modrých prekážok k zelenému cieľu pomocou metódy potenciálnych polí. Odpudivé sily prekážok sú vyznačené červenými šípkami a priťažlivá sila cieľu žltými šípkami. Na začiatku smeruje cesta priamo k cieľu, kým sa nedostane do vplyvu odpudivých síl, potom sleduje cestu najnižšieho potenciálu pomedzi prekážky až do cieľa.

Riešenie problému lokálnych miním

Plánovač Choset [2] napríklad navrhuje použite Wave-Front plánovača, ktorý je plánovačom pracujúcim s mriežkovou reprezentáciou prostredia. Pre jednoduchosť vysvetlenia budeme uvažovať 2D priestor. Tento plánovač začína s binárnou mriežkou núl reprezentujúcich voľný priestor, jednotiek reprezentujúcich prekážky a lokáciou štartu a cieľu v mriežke. Následne pridelí cieľovej bunke číslo dva a v prvom kroku algoritmu pridelí všetkým bunkám s hodnotou nula a susediacim s cieľovou bunkou číslo tri. Potom všetkým

nulovým bunkám susediacim s trojkou pridelí štvorku, atď. Prideľovanie hodnôt bunkám končí, keď algoritmus narazí na štartovaciu bunku. Takto v podstate vytvorí potenciálnu funkciu na mriežke s jedným lokálnym minimom (číslom dva). V ďalšej fázi plánovač postupne nájde cestu pomocou gradientového zostupu. Ak začínal na bunke s hodnotou X, tak hľadá bunku s hodnotou X-1, v prípade viacerých buniek s rovnakou hodnotou je môžné zvoliť ľubovoľnú bunku. Wave-Front Planner je možné zovšeobecniť aj do vyšších dimenzií (pomocou viac-dimenzionálnej mriežky, postup je potom analogický), avšak nakoľko potrebuje tento plánovač za účelom nájdenia cesty prehľadať celý priestor, narastá jeho pamäťová a časová náročnosť exponenciálne s nárastom dimenzií.

Navigačné funkcie Navigačné funkcie sú Morseove funkcie $\varphi : \mathbb{Q}_{free} \to [0, 1]$, ktoré sú spojito diferencovateľné (alebo aspoň dvakrát), majú iba jedno unikátne minimum a sú uniformne maximálne na hranici voľného priestoru. *Morseova funkcia* je funkcia, ktorá má iba nedegenerované (izolované) kritické body, teda akékoľvek narušenie destabilizuje maximá a sedlové body. V týchto funkciách je potenciál funkciou vzdialenosti od prekážok.
Metóda potenciálnych funkcií je pomerne rozšírená a aplikovateľná na problémy plánovania cesty, ako je demonštrované napr. v Khuswendi [10], ktorý využíva tento prístup spolu s algoritmom A* na plánovanie cesty pre bezpilotné lietadlá. Táto práca však neberie do úvahy plne 3D priestor a podobný trend sa dá pozorovať aj v iných prácach zaoberajúcich sa potencálnymi funkciami. Ostáva teda otázkou, ako efektívne je možné pomocou potenciálných funkcií využiť plné pohybové možnosti lietadiel v 3D priestore, napríklad pri prelietavaní ponad vysoké prekážky.

3.1.2 Dekompozícia do buniek

Metódy dekompozície do buniek sa spoliehajú na reprezentácii voľného konfiguračného prostredia ako zjednotenia jednoduchých regiónov - buniek. Zdieľané hranice buniek sú často abstrakciou zmeny niektorej metriky bunky, napríklad jej najbližšej prekážky. Bunky zdieľajúce hranice nazývame *priľahlé*. Na tomto princípe priľahlosti môžeme následne zkonštruovať graf priľahlosti, kde sú bunky vrcholmi a hrany spojujú vrcholy priľahlých buniek. Plánovanie cesty prebieha v dvoch fázach: najprv plánovač zistí, ktoré bunky obsahujú štart a cieľ a následne hľadá cestu v grafe priľahlosti.
Dekompozícia do buniek tiež poskytuje možnosť dosiahnutia tzv. pokrytia. Pokrytie je nájdenie takej cesty, ktorá robota naviguje cez všetky body vo voľnom priestore. V prípade dekompozície do buniek je dosiahnuteľná vyčerpávajúcim prechodom hrán grafu priľahlosti.

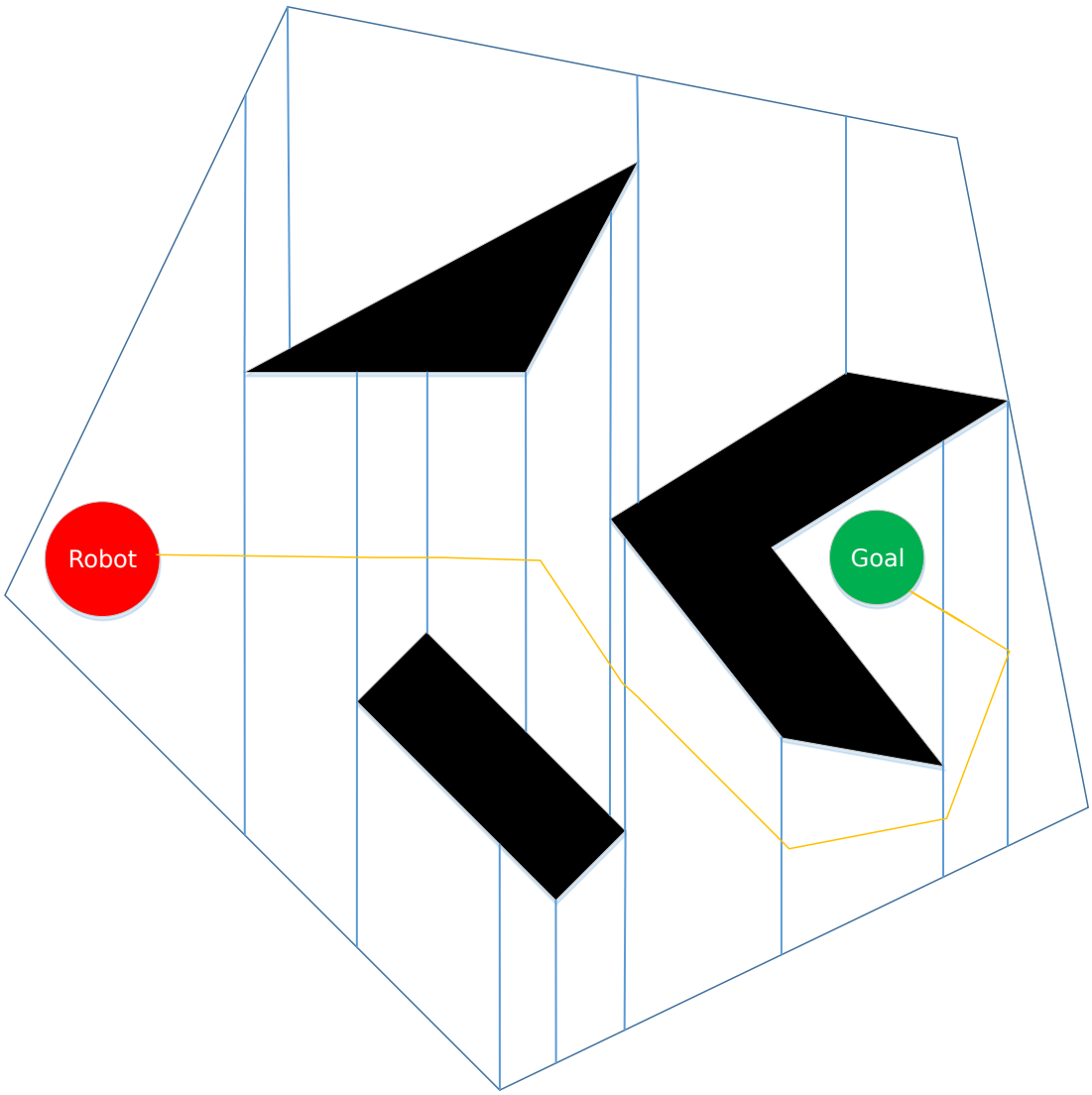
Lichobežníková dekompozícia

Tento spôsob dekompozície rozdeľuje rovinný konfiguračný priestor do lichobežníkov, prípadne trojuholníkov, o ktorých môžme uvažovať ako o lichobežníkoch s nulovou dĺžkou jednej z rovnobežných strán. Vo všeobecnom mnohouholníkovom konfiguračnom priestore, ktorý obsahuje mnohouholníkové prekážky, prebieha lichobežníková dekompozícia nasledovne:

1. Z každého vrcholu mnohouholníkov (vrátane toho, čo reprezentuje hranice konfiguračného priestoru) nakreslíme dve úsečky - horné a dolné vertikálne predĺženie. Tie začínajú v bode vrcholu a končia hneď ako narazia na hranu niektorého ďalšieho mnohouholníku. Tým rozdelíme priestor do buniek.

2. Následne zkonštruujeme graf priľahlosti a vyhľadáme v ňom cestu, čím získame postupnosť vrcholov, ktoré vedú zo štartu do cieľa.

3. V poslednom kroku vytvoríme cestu, ktorú môže robot sledovať. Keďže lichobežníky sú konvexné, tak ľubovoľné body ležiace na hranách lichobežníkov sa dajú spojiť priamou čiarou, ktorá nepretína žiadnu prekážku. Teda vytvoriť cestu môžeme napríklad spojením stredov strán (vertikálnych predĺžení) lichobežníkov.



Obr. 3.2: Lichobežníková dekompozícia: Robot sa snaží nájsť cestu k cieľu pomedzi čierne prekážky. Vertikálne predĺženia sú zobrazené ako zvislé modré čiary, deliace priestor do buniek. Cesta, vyznačená žltou, sa skonštruuje spojením stredov strán buniek.

3D dekompozícia do buniek

Najjednoduchšou metódou 3D bunkovej dekompozície je *naivná* dekompozícia, kedy je celý konfiguračný priestor rovnomerne rozdelený do 3D mriežky, ktorej bunky majú tvar kvádra.

Túto metódu úspešne použili v Hwangbo [8], kde vo fázi globálneho plánovania prostredie rozdelili do 3D mriežky, v ktorej následne hľadali cestu algoritmom A*. Taktiež je možné do vyšších dimenzií rozšíriť lichobežníkovú dekompozíciu, prípadne použiť Collinsovu dekompozícu alebo prístupy založené na algebraickej geometrii. Použitie 3D lichobežníkovej dekompozície je ale obmedzené iba na na problémy, v ktorých sa vyskytujú iba po častiach lineárne prekážky a Collinsova dekompozícia a algebraická geometria sú pre potreby tejto práce zbytočne zložité.
Grafy, ktoré vznikajú pri použití metódy dekompozície do buniek sa dajú využiť aj ako roadmapy. Niektoré práce, ako napr. LaValle [11] ich dokonca považujú za, v podstate, rovnaký prístup.
Algoritmy dekompozície do buniek sú populárne hlavne pre 2D problémy, ale je možné ich použiť aj v 3D. Príkladom môže byť už spomínaná práca Hwangbo [8], ale aj Wang [15]. Metódy dekompozície do buniek sú z hľadiska tejto práce zaujímavé napriek ich vyššej náročnosti v 3D pre ich názornosť a jednoduchú myšlienku. Je na nich dobre vidieť ich jednotlivé kroky a sú tiež veľmi flexibilné.

3.1.3 Roadmapy

Roadmapy sú široko rozšírenou metódou plánovania ciest ako v 2D, tak aj v 3D. Z pohľadu použita pre lietadlá však nie je ich schopnosť vyhodnotiť viacero plánovacích problémov v jednom prostredí zásadná, nakoľko sa nepredpokladá nutnosť navigovania jedným nezmeneným prostredím viackrát. Taktiež fakt, že sa v mnohých prípadoch jedná iba o rozšírenie algoritmov dekompozície do buniek, znamená, že z pohľadu tejto práce nie je metóda roadmáp zásadne zaujímavá, je však obsiahnutá pre úplnosť.
Roadmap môžeme definovať ako topologický graf $\mathcal{G}$ ležiaci vo voľnom konfiguračnom priestore C_{free}, ktorý spĺňa nasledujúce podmienky:

1. **Dostupnosť:** Z ľubovoľného $q \in C_{free}$ je možné nájsť jednoduchú a efektívnu cestu $\tau : [0,1] \rightarrow C_{free}$ takú, že $\tau(0) = q$ a $\tau(1) = s$, $s \in S$, kde S je množina bodov obsiahnutých v grafe $\mathcal{G}$.

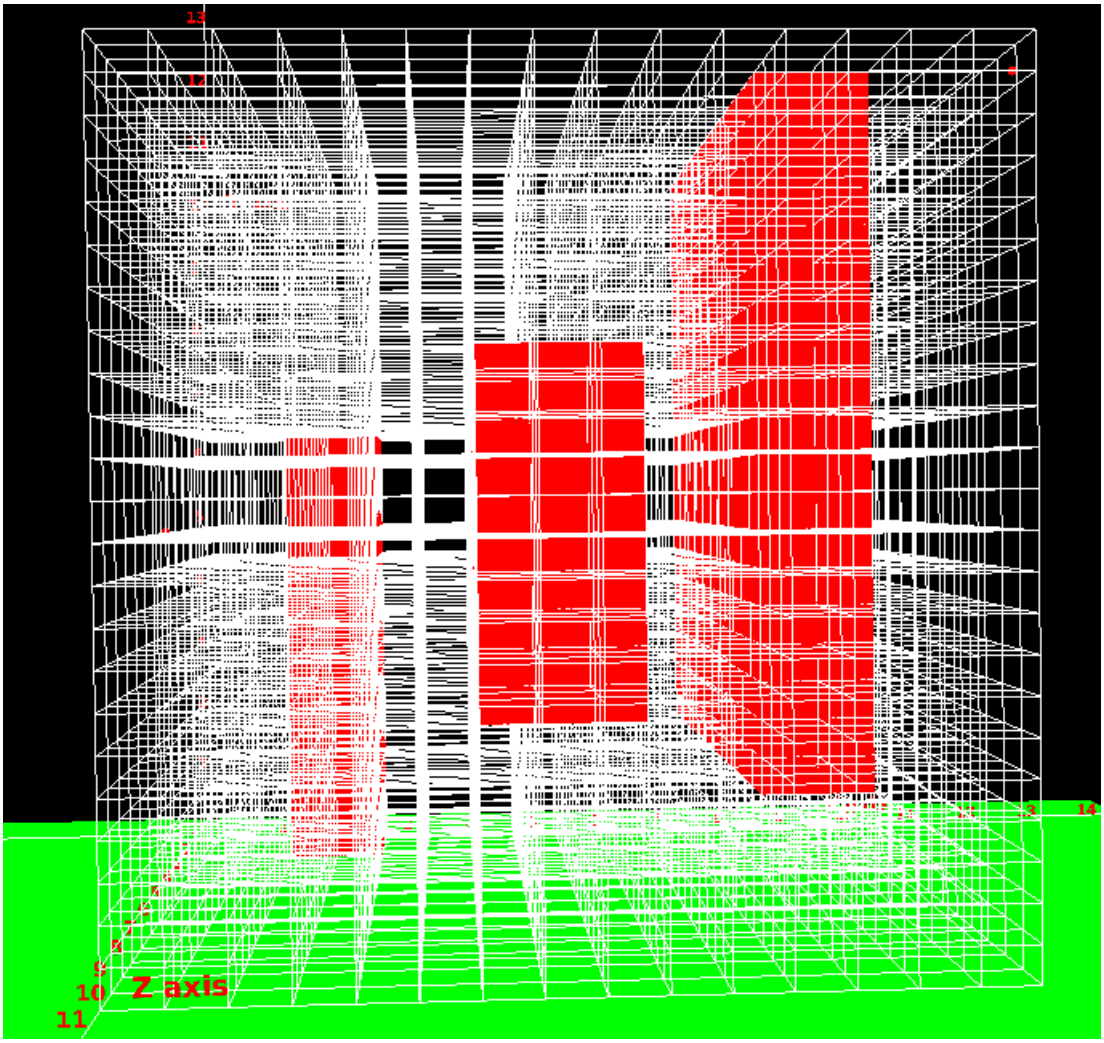
2. **Zachovávanie prepojenosti:** Ak existuje cesta $\tau : [0,1] \rightarrow \{C_{free}$, kde $\tau(0) = q_1$ a $\tau(1) = q_G$, tak tiež musí existovať taká cesta $\tau' : [0,1] \rightarrow \mathcal{S}$, kde $\tau'(0) = s_1$ a $\tau'(1) = s_2$. To zaisťuje, že $\mathcal{G}$ zachytáva prepojenosť C_{free}.

Požiadavka q_1, q_G je vyriešená najprv spojením začiatočného bodu s roadmapou a následne prehľadaním grafu $\mathcal{G}$.

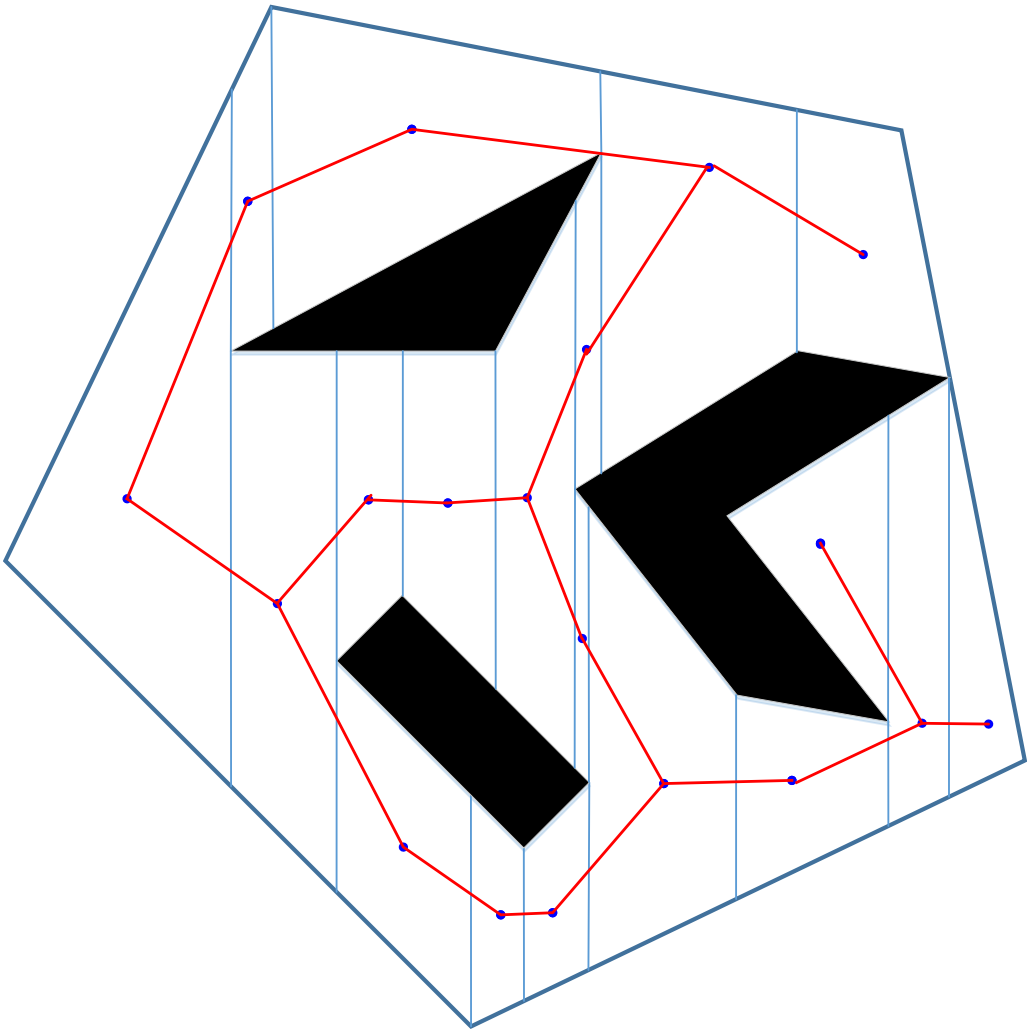
Konštrukcia roadmáp

V tejto podsekcii predstavíme dva typy roadmáp a spôsob ich konštrukcie. Detailnejší prehľad je dostupný v Choset [2].

Mapy viditeľnosti Mapy viditeľnosti sú charakterizované tým, že ich vrcholy zdieľajú hranu vtedy, keď sú vo vzájomných líniach viditeľnosti a ešte tým, že všetky body voľného priestoru C_{free} sú viditeľné z aspoň jedného vrcholu mapy. Najjednoduchší typ mapy viditeľnosti je *graf viditeľnosti*.



Obr. 3.3: Mriežka zobrazujúca dekompozíciu 3D priestoru do buniek rovnakej veľkosti, s prekážkami vyznačenými červenou farbou. Zdroj: Obrázok vyhotovený z appletu vytvoreného v rámci tejto práce.



Obr. 3.4: Roadmapa, vyznačená červenou, je skonštruovaná na základe zachytenia prepojenosti buniek vytvorených v rámci lichobežníkovej dekompozície, viď obrázok **??**.

Graf viditeľnosti Je definovaný v 2D mnohouholníkovom konfiguračnom priestore ako graf, ktorého vrcholmi sú počiatočný a cieľový bod spolu so všetkými vrcholmi prekážok v pracovnom priestore. Hrany spájajú vrcholy v líniách viditeľnosti a zahŕňajú aj hrany prekážok. Toto však často vedie k priveľkému počtu nepotrebných hrán, preto sa pre zníženie počtu hrán používajú koncepty *podporných* a *rozdeľujúcich* hrán. Podporné hrany sú také, ktoré sú dotyčnicami dvoch prekážok tak, že obe prekážky ležia na jednej jej strane. Rozdeľujúce hrany sú dotyčnicami dvoch prekážok tak, že ležia na rozdielnych stranách hrany. Potom *redukovaný graf viditeľnosti* je grafom tvoreným iba z podporných a rozdeľujúcich hrán.
Nech $V = v_1, \ldots, v_n$ je množina všetkých vrcholov prekážok v konfiguračnom priestore ako aj počiatočnej a cieľovej konfigurácie. Potom na konštrukciu grafu viditeľnosti musíme pre každé v určiť, ktoré vrcholy sú mu viditeľné. Najjednoduchším spôsobom je otestovať všetky polpriamky $vv_i, v \neq v_i$, či pretínajú niektorú z hrán prekážok. Na ich základe následne môžme zostrojiť graf viditeľnosti.

Všeobecné Voroného diagramy

Všeobecné Voroného diagramy (OVD) sú založené na množinách bodov, ktorých vzdialenosti d_i a d_j od dvoch množín bodov $\mathcal{QO}_\rangle$ a $\mathcal{QO}_|$ sú rovnaké, tieto množiny nazývame dvoj-ekvidištantné [1] povrchy $S_{ij} = x \in \mathcal{Q}|(d_i(q) - d_j(q)) = 0$. Tieto povrchy však môžu zasahovať do prekážok, preto sa zavádza aj pojem dvoj-ekvidištantné strany, ktoré sú definované ako $\mathcal{F}_{ij} = q \in \mathcal{S}_{ij}|d_i(q) \leq d_h(q) \forall h$. Potom v 2D prípade sú hranami OVD práve strany $\mathcal{F}_{ij}$, ktoré končia buď v *stretových bodoch* (body rovnako vzdialené od troch alebo viacerých prekážok) alebo *hraničných bodoch*, ktorých vzdialenosť od prekážok je nulová (body, kde sa pretínajú hrany OVD s hranicou konfiguračného priestoru).
OVD sa dajú konštruovať niekoľkými spôsobmi:

1. Ak je robot vybavený senzormi, tak je možné zostrojiť OVD dynamicky počas skúmania priestoru robotom.

2. V priestoroch, ktoré obsahujú iba mnohouholníkové objekty zložené z vrcholov a hrán je jednoduché definovať rovnako vzdialené body: v prípade hrany sú reprezentované priamkou, v prípade vrcholu parabolou.

3. V priestore reprezentovanom diskrétnou mriežkou je možné použiť tzv. *Brushfire* algoritmus, ktorého vstupom je mriežka, kde všetky bunky zabrané prekážkami majú hodnotu 1 a voľné bunky 0. Výstupom tohto algoritmu je diskrétna mriežka, kde každá bunka má hodnotu odpovedajúcu jej vzdialenosti od najbližšieho bodu najbližšej prekážky. Brushfire si môžme prestaviť ako vlnu, ktorá sa šíri smerom od prekážok a priraďuje bunkám hodnotu rovnú prejdenej vzdialenosti. Bunky, na ktorých sa dve vlny zrazia majú rovnakú vzdialenosť od daných prekážok a teda budú patriť do OVD.

Príkladom použitia Voroného diagramu je aj práca spomínaná v sekcii o dekompozícii do buniek, Wang [15].

3.2 Prístupy založené na vzorkovaní

Hlavnou myšlienkou vzorkovacích prístupov je postupné prehľadávanie konfiguračného priestoru. Algoritmy tohto typu sa nesnažia skonštruovať C_{obs}, miesto toho hľadajú cestu po-

[1] po anglicky *two-equidistant*

mocou tzv. modulu detekcie kolízii, ktorý určuje či je navzorkovaná konfigurácia v kolízií. Modul detekcie je nezávislým komponentom, vďaka čomu je možné vyvinúť plánovacie algoritmy, ktoré sú nezávislé od geometrickej reprezentácie pracovného prostredia.
Táto sekcia je organizovaná naslednovne: v prvej podsekcii sú popísané dôležité pojmy vzťahujúce sa na vzorkovacie prístupy plánovania cesty, v nasledujúcich podsekciách sú popísané konkrétne algoritmy plánovania cesty založené na vzorkovaní.

3.2.1 Pojmy

Vzorkovanie Množstvo prvkov v pracovnom priestore plánovaču je nekonečné, avšak plánovací algoritmus založený na vzorkovaní môže zobrať do úvahy iba konečný počet vzoriek. Keďže tieto algoritmy sú často ukončené skôr poradie v ktorom sú vzorky vyberané, je kľúčové. Z toho dôvodu rozlišujeme *množinu* vzoriek a *sekvenciu* vzoriek. Zo sekvencie vzoriek vieme vždy skonštruovať unikátnu množinu vzoriek, ale z jednej množiny vzoriek vieme skonštruovať viacero sekvencií.

Hustota vzorkovania Ak hľadáme nekonečnú sekvenciu vzoriek nad C, tak by sme ideálne chceli, aby sekvencia nakoniec pokryla všetky body v C. Nakoľko je C nespočítateľne nekonečné, nie je možné pokryť všetky jeho body, ale môžeme nájsť sekveniciu, ktorá sa k nim dostatočne blíži. Toto v topológii nazývame *hustota*. Uvažujme U a V, ktoré sú podmnožinami topologického priestoru, potom U je husté ak platí, že uzáver $U = V$. Vytváranie hustých sekvencií je základná vlastnosť vzorkovacích metód pre plánovanie ciest.

Kompletnosť Algoritmus je považovaný za *kompletný*, ak pre ľubovoľný vstup správne rozhodne, či existuje riešenie v konečnom čase. Ak riešenie existuje, tak ho vypočíta v konečnom čase. Vzorkovacie algoritmy nedosahujú kompletnosť, avšak za predpokladu, že vzorkovanie je dostatočne husté, môžme deterministický algoritmus vyhlásiť za *rezolúčne kompletný*. To znamená, že ak existuje riešenie, tak ho algoritmus nájde v konečnom čase, avšak ak neexistuje, môže sa algoritmus zacykliť. Algoritmy založené na náhodnom vzorkovaní vedú k *pravdepodobnostne kompletným* algoritmom, nakoľko náhodne vygenerovaná sekvencia je hustá s pravdepodobnosťou rovnej 1 (LaValle [11]). Pravdepodobnosť nájdenia existujúceho riešenia potom konverguje k 1 s narastajúcim počtom bodov.

Detekcia kolízií Ďalším krokom po navzorkovaní priestoru je detekcia kolízií. Väčšina algoritmov spotrebuje najviac výpočetného času práve pri tejto činnosti. Detekciu kolízií môžeme reprezentovať ako logický výraz $\phi : C \to \{\texttt{TRUE, FALSE}\}$. Potom ak $q \in C_{obs}$, tak $\phi(q) = \texttt{TRUE}$, inak $\phi(q) = \texttt{FALSE}$. V niektorých príkladoch používajú algoritmy detekcie kolízií dvojfázový prístup.

1. Hrubá fáza - V tejto fázi sa algoritmus snaží vyhnúť náročným výpočtom pre telesá, ktoré sú ďaleko od seba. Napríklad pomocou ohraničenia nekonvexných telies boxami a následne zisťovaním, či došlo ku kolízii iba v prípade, že sa niektoré boxy prekrývajú.

2. Úzka fáza - V úzkej fázi sa páry telies individuálne a dôsledne skontrolujú na kolízie, ak sa v hrubej fázi zistila ich kolízia.

V LaValle [11] sú rozlišované dve metodológie detekcie kolízií: *hierarchická* a *inkrementálná*.

Hierarchické metódy detekcie kolízií V hierarchických metódach sa predpokladajú dve komplikované nekonvexné telesá E a F, u ktorých chceme skontrolovať, či sú v kolízii. Hierarchické metódy rozložia každé teleso do stromu. Každý vertex stromu zastupuje hraničná oblasť, ktorá obsahuje nejakú časť telesa. Hraničná oblasť koreňového vertexu obsahuje celé teleso. Výber typu hraničnej oblasti sa zvyčajne riadi jedným z týchto kritérii:

1. Oblasť by mala čo najužšie priliehať na vybrané body telesa
2. Test prieniku dvoch oblastí by mal byť čo najefektívnejší

Inkrementálne metódy detekcie kolízií Tieto metódy sú založene na tzv. inkrementálnom výpočte vzdialenosti, ktorý predpokladá, že medzi za sebou nasledujúcimi volaniami algoritmu detekcie kolízií sa telesá posunú iba o malú vzdialenosť. Podmienkou použitia inkrementálnej metódy je spojitosť modelov. Tieto metódy počítajú s množinami bodov telies rozdelených do Voroného regiónov (regiónov, z ktorých je daný bod najbližším bodom telesa), spomedzi ktorých vždy vyberú najbližšie dvojice bodov a ich vzdialenosť je rovná vzdialenosti medzi telesami. Predpoklad malého posunu medzi meraniami zvyšuje pravdepodobnosť, že vybraná dvojica bodov bude najbližšou dvojicou aj v ďalšom kroku a teda nebude nutné znovu ju určiť.

3.2.2 Pravdepodobnostné roadmapy

Pravdepodobnostné roadmapy (PRM) sú roadmapy, ktoré konštruujeme na základe navzorkovaných bodov. Fungujú v dvoch fázach:

Učiaca fáza Tu dochádza ku konštrukcii neorientovaného grafu $G = (V, E)$, kde uzly vo V sú množinou konfigurácií robota vybraných určitou metódou v Q_{free}. Hrany (q_1, q_2) v E korešpondujú s lokálnymi bezkolíznymi cestami spájajúcimi q_1 a q_2, vybranými lokálnym plánovačom. Na začiatku je graf G prázdny, následne dochádza opakovane k vzorkovaniu konfigurácii z $\mathcal{Q}$ podľa vybranej vzorkovacej metódy. Pokiaľ nie je vybraná konfigurácia v kolízii, tak je pridaná do roadmapy. Po nájdení n bezkolíznych konfigurácií je pre každé $q \in V$ vybraná množina N_q jej k najbližších konfigurácií podľa nejakej vzdialenostnej metriky. Potom sa lokálny plánovač pokúsi prepojiť q s každým uzlom $q' \in N_q$, ak sa prepojenie podarí, tak je hrana (q, q') pridaná do roadmapy.

Dotazovacia fáza V druhej fázi sa roadmapa G používa na hľadanie ciest. Pre počiatočnú konfiguráciu q_{start} a cieľovú q_{goal} sa ich lokálny plánovač pokúsi spojiť s k najbližšími $q, \ldots, q_k \in V$. Ak úspešne prepojí aspoň jeden pár (q_{start}, q') a (q_{goal}, q''), tak v grafe G vyhľadá najkratšia postupnosť hrán spájajúcich q' a q''. Nakoniec je táto postupnosť transformovaná na uskutočniteľnú cestu prepočítaním a pospájaním príslušných lokálnych ciest.

Rýchlosť a úspešnosť PRM úzko súvisí s vybranými metódami vypočtu vzdialeností, vzorkovania, spájania konfigurácií, lokálneho plánovania, ako aj prípadných heuristík výslednej cesty. V nasledujúcich podsekciách priblížime vybrané metódy vzorkovania prostredia a spájania konfigurácií. Širší prehľad, ako aj dopad metód výpočtu vzdialeností, lokálneho plánovania a heuristík na kvalitu roadmapy je možné nájsť v Choset [2] a v Geraerts [5]. Podobne ako klasické roadmapy, aj PRM algoritmy sú veľmi populárne. Príkladom použitia PRM pre plánovanie cesty pre lietadlá môžu byť práce Yan [19] a Pettersson [13].

Vzorkovacie metódy

Uniformná náhodná Vzorkovanie prebieha náhodne podľa uniformného rozloženia. Nevýhodou tejto metódy je, že je niekedy pomalá, čoho príčinou je často tzv. problém úzkeho prechodu, kedy úzky prechod existujúci v Q_{free} je potrebný na nájdenie cesty. Spomalenie nastáva, pretože je potrebné vybrať vzorku z veľmi malej množiny konfigurácií nachádzajúcich sa práve v tomto úzkom prechode.

Vzorkovanie blízko prekážok Filozofiou za vzorkovaním blízko hraníc prekážok je to, že úzke prechody môžme považovať aj za koridory obklopené prekážkami. Príkladom tejto metódy je OBPRM, ktoré najprv náhodne vygeneruje konfigurácie podľa uniformného rozloženia. Pre každú konfiguráciu q_{in}, ktorá je v kolízii, následne náhodne zvolí smer v a hľadá bezkolíznu konfiguráciu q_{out} v tomto smere. Následne pomocou binárneho vyhľadávania nájde konfiguráciu q, ktorá je najbližšie hranici prekážky a pridá ju do roadmapy, konfigurácie q_{in} a q_{out} sa zahodia.

Vzorkovanie na základe viditeľnosti Viditeľnosť konfigúracie q je množina všetkých konfigurácií, s ktorými vie lokálny plánovač q prepojiť. Táto metóda neakceptuje všetky bezkolízne konfiguácie, ale do roadmapy pridáva iba konfigurácie, ktoré sa buď nedajú pripojiť k žiadnemu existujúcemu uzlu roadmapy, alebo aspoň ku dvom.

Vzorkovanie založené na mriežke Táto metóda využíva bunky mriežky ako vzorky, často s hrubším rozlíšením, aby naplno využila modul detekcie kolízii. V dotazovacej fázi sa potom roadmap pokúša spojiť q_{start} a q_{goal} k priľahlým bunkám mriežky.

Metódy spájania konfigurácií

Najbižší susedia Lokálny plánovač sa tu pokúsi spojiť každú konfiguráciu s jej k najbližšími susedmi. Myšlienkou za týmto prístupom je to, že nakoľko sú susedia blízko k vybranej konfigurácii, tak je vyššia pravdepodobnosť, že bude spojenie bezkolízne. Túto metódu sme použili pri predstavení pravdepodobnostných roadmáp v tejto podsekcii.

Riedke spájanie Použitím riedkeho spájania je možné urýchliť konštrukčnú fázu roadmapy, napríklad vynechaním počítania hrán, ktoré sú sučasťou rovnakého, už prepojeného komponentu. Pridávanie týchto hrán nezlepší prepojenosť roadmapy, nakoľko už existuje cesta medzi ľubovoľnými dvoma konfiguráciami v jednom prepojenom komponente. Najjednoduchšie je prepojiť konfiguráciu iba s najbližšími uzlami priľahlých komponentov. Nevýhodou je, že plánovač môže niekedy kvôli absencii týchto hrán vypočítať zbytočne dlhú cestu. To sa však dá napraviť rôznym postprocessingom, alebo konštrukciou iba niektorých redundantných hrán.

Lenivé vyhodnocovanie Tento prístup sa dá aplikovať na všetky metódy spájania konfigurácií. Jeho cieľom je zlepšiť výkon algoritmu oddialením kontroly kolízií, až kým to nebude absotútne nutné. V takomto prípade bude PRM pracovať s roadmapou G, ktorej hrany ešte neboli plne skontrolované, algoritmus teda zprvu považuje všetky hrany a uzly za bezkolízne. Keď príde požiadavka na vyhľadanie cesty, a konfigurácie q_{start} a q_{goal} sa podarí spojiť s G, tak sa PRM pokúsi vyhľadať najkratšiu cestu grafom G. Vtedy dôjde aj ku kontrole kolízií, a to nasledovne:

Najprv sa skontrolujú uzly ležiace na nájdenej ceste grafom, pokiaľ sa zistí, že uzol je v kolízii, tak je spolu so všetkými jeho hranami z grafu vylúčený. Potom, čo sa nájde cesta, na ktorej sú všetky uzly bezkolízne, sa skontrolujú hrany tejto cesty. Prvá kontrola je spravená s hrubým merítkom, ale následne sa opakuje so stále jemnejším, až kým sa nedosiahne chcenej diskretizácie. Ak je hrana v kolízii s prekážkou, tak je podobne ako uzol vyradená z grafu G. Medzi jednotlivými požiadavkami sa informácie a vyradených uzloch a hranách ukladajú, aby sa predišlo duplicitným výpočtom.

3.2.3 Rapidly-exploring random trees (RRT)

RRT je pravdepodobnostne kompletný plánovací algoritmus pôvodne určený pre kinodynamické plánovanie cesty. Narozdiel od PRM vie RRT odpovedať iba na jeden dotaz, nakoľko konštruuje iba cestu pre daný problém a nie mapu celého Q_{free}. Vzhľadom na náhodnosť vzorkovaných bodov je tiež pravdepodobné, že pri niekoľkých nazávislých behoch algoritmu v rovnakom prostredí nájde zakaždým inú cestu. RRT sa skladá z dvoch striedajúcich sa krokov.

Konštrukcia stromov

Najprv RRT postupne konštruuje dva stromy, T_{start} má koreň v q_{start} a T_{goal} v q_{goal}. Oba stromy sú postupne rozširované pridávaním náhodnej konfigurácie q_{rand} uniformne vzorkovanej z Q_{free}. Algoritmus nájde najbližšiu konfiguráciu q_{near} ku q_{rand} už existujúcu v jednom zo stromov a pokúsi sa o priblíženie z q_{near} ku q_{rand}. Väčšinou sa jedná o posunutie q_{near} o krok `step size` po priamke z q_{near} do q_{rand}. Ak je táto nová konfigurácia q_{new} bezkolízna, tak je pridaná do vrcholov T spolu s jej príslušnou hranou. Veľkosť kroku `step size` sa často vyberá dynamicky, v závislosti od vzdialenosti medzi q_{new} a q_{rand}, ak sú od seba ďaleko, tak je aj krok dlhší a naopak.

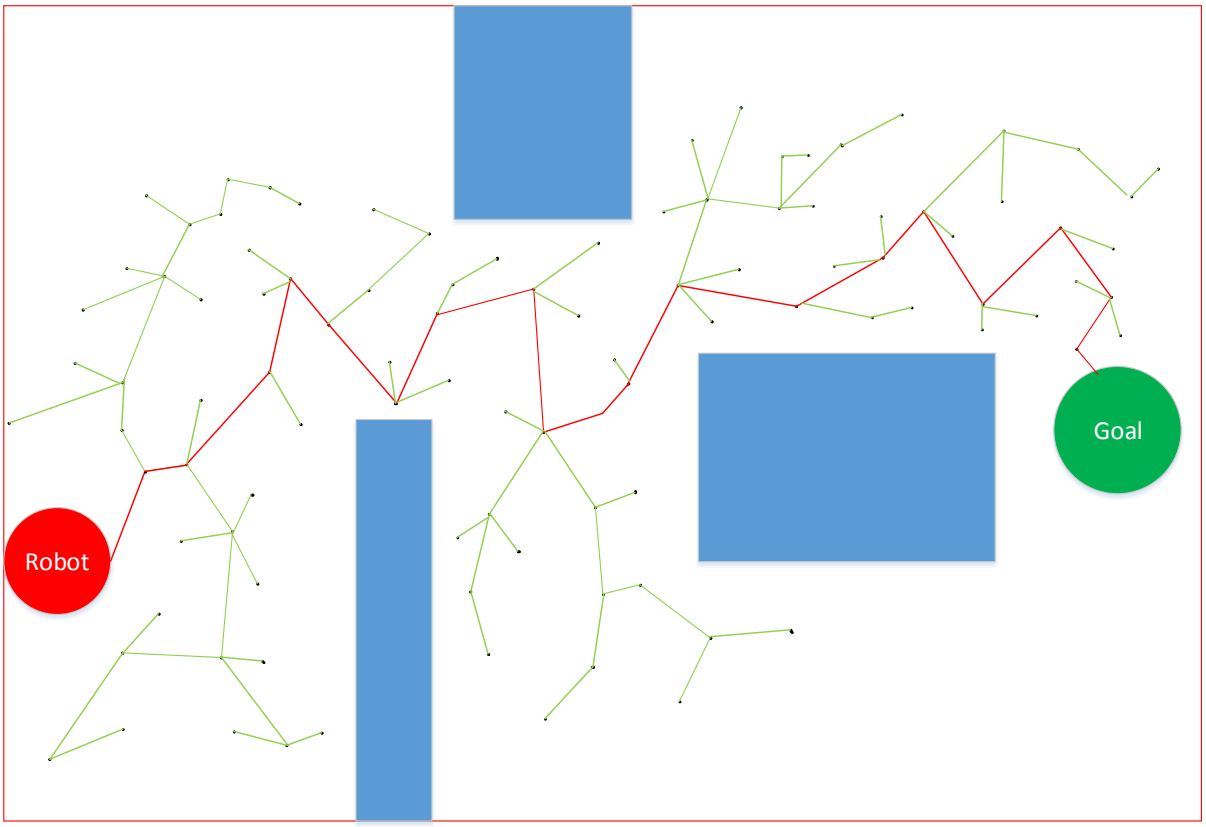
Spájanie stromov

Druhým krokom RRT algoritmu je spojenie oboch stromov T_{start} a T_{goal}. Najprv sa podobne ako pri konštrukcii jednotlivých stromov vygeneruje q_{rand}, následne RRT predĺži jeden zo stromov ku q_{rand} a získa tak q_{new}, ku ktorému sa pokúsi predĺžiť druhý strom. Ak sa preĺženie podarí a stromy sa spoja, tak dôjde k ukončeniu algoritmu. V opačnom prípade sa vymení poradie rozširovania stromov a proces sa opakuje.

Vzorkovanie

Okrem uniformného náhodného vzorkovania sa často používajú aj zaujaté metódy, ako napríklad za q_{rand} v každom kroku zvoliť q_{goal} pre T_{start} a q_{start} pre T_{goal}. Tento prístup teoreticky zrýchľuje konvergenciu stromov ku chceným konfiguráciám, no môže aj spôsobiť zaseknutie algoritmu v lokálnych minimách. Preto je lepšou možnosťou striedať uniformné a zaujaté vzorkovanie s určitou pravdepodobnosťou, čo poskytuje rýchlejšiu konvergenciu a zároveň sa vyhýba problémom lokálnych miním.

V poslednom čase sú RRT algoritmy jedny z najpoužívanejších plánovacích algoritmov v oblasti bezpilotných lietadiel. Je možné ich rozšíriť do 3D, sú pomerne rýchle a tiež vynikajú v zachytávaní kinematických možností lietadla. Ich použitie je demonštrované v



Obr. 3.5: Rapidly-exploring random trees: Malé čierne body značia náhodne zvolené vzorky. Modrou sú vyznačené prekážky, zelenou strom konštruovaný algoritmom a červenou nájdená cesta.

Cheng [1], kde je použitý algoritmus RRT spolu s B-Spline krivkami vyhladzujúcimi výslednú cestu. Metóda RRT sa používa aj pri plánovaní pre quatrokoptéry, napr. v Richtera [14], Pettersson [13] a v upravenej podobe RRT* v práci Webb [16].

3.2.4 Sampling-based roadmap of trees (SRT)

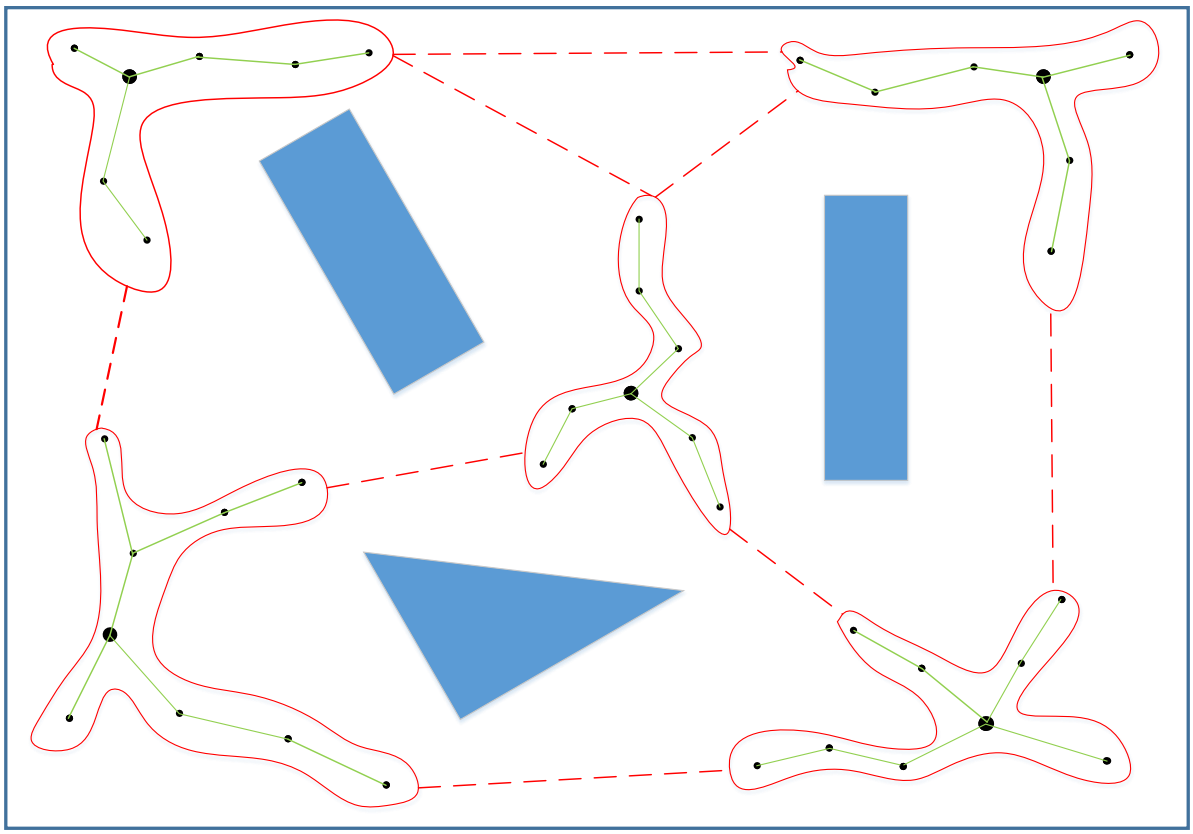
SRT je kombináciou PRM a vzorkovacích metód založených na stromoch, ako napríklad RRT. RRT v tomto algoritme nahradí lokálny plánovač PRM, výsledná roadmapa zachytávajúca prepojenosť Q_{free} má namiesto uzlov stromy, hrany sú vypočítané pomocou obojsmerného stromového algoritmu (napríklad RRT). Matematicky môžme výslednú roadmapu zapísať ako neorientovaných graf $G_T = (V_T, E_T)$, kde $V_T = T_1, \ldots, T_n$ a $(T_i, T_j) \in E_T$ ak existujú také konfigurácie $q_i \in T_i$ a $q_j \in T_j$, ktoré boli spojené lokálnou cestou.

Pridávanie uzlov

Uzly sú pridávané náhodným uniformným vzorkovaním Q_{free}. Z týchto vzorkov algoritmus následne spraví korene stromov, ktoré sú rozširované pomocou RRT. Je možné aplikovať aj iné metódy vzorkovania pre PRM.

Pridávanie hrán

Spájanie stromov (uzlov) roadmapy prebieha podobne ako v PRM, pre každý strom T_i sa nájde množina N_{T_i} najbližších stromov. Následne sa algoritmus snaží spojiť T_i s každým stromov z N_{T_i}. Na vypočítanie susedov daného stromu je pre všetky stromy vypočítaná tzv. reprezentatívna konfigurácia q_{T_i} ako agregát konfigurácií v danom strome. Susedné stromy sú potom určené vzdialenosťou medzi týmito konfiguráciami. Podobne ako pri vzorkovaní uzlov aj pri spájaní hrán sa v SRT dajú použiť rôzne metódy známe z konštrukcie PRM. Algoritmus SRT dosahuje dobrých výsledkov pre širokú škálu problémov, ale jeho náročnosť z hľadiska kombinácie viacerých plánovacích algoritmov ho robí menej ako ideálnym pre potreby tejto práce.



Obr. 3.6: Sampling-based roadmap of trees: Okolo náhodne zvoleného vzroku, vyznačeného čiernym bodom, sa ďalším vzorkovaním utvorí strom, značený zelenou farbou. Vytvorené stromy nakoniec slúžia ako uzly roadmapy, na obrázku vyznačenej červenou farbou.

3.3 Zhrnutie

Z algoritmov popísaných v tejto prehľadovej kapitole som sa rozhodol vizualizovať v applete po jednom zástupcovi z oboch prístupov k plánovaniu ciest. Z metód so znalosťou prostredia som vybral metódu naivnej 3D dekompozície do buniek pre jej už spomínané prednosti - názornosť, jednoduchosť a flexibilnosť. Tiež má výhodu, že sa dá použiť aj na konštrukciu roadmapy, takže s prípadným budúcim rozšírením, ktoré by v danej dekompozícii zostrojilo roadmapu môžeme používateľovi predstaviť aj tento prístup. Medzi vzorkovacími metódami je najviac zaujímavý algoritmus RRT, ktorý je ideálny pre problematiku plánovania ciest pre lietadlá, ako aj quatrokoptéry, je často používaný a dobre odráža filozofiu vzorkovacích metód.

Kapitola 4

Implementácia

V tejto kapitole je priblížená implementácia vizualizácie vybraných algoritmov vo forme appletu, ktorý bol cieľom tejto práce. Applet je implementovaný v jazyku Java s pomocou knižnice Java3D, použitej na implementáciu 3D grafického prostredia. Applet sa skladá z tried implementujúcich: grafické uživateľské prostredie, algoritmus RRT, algoritmus naivnej bunkovej dekompozície, vizualizáciu týchto algoritmov, algoritmus Runge-Kutta 4. rádu, modul výpočtu pohybu lietadla podľa pohybových rovníc a niekoľko pomocných modulov poskytujúcich dátové štruktúry a rozhrania. Popis implementácie je rozdelený do nasledujúcich sekcii podľa príslušnosti modulov k daným častiam výslednej aplikácie.

4.1 Algoritmy plánovania cesty

Plánovacie algoritmy sú implementované v moduloch RRT a CellDecomposition v rovnomenných triedach. Triedy plánovacích algoritmov dedia z abstraktnej generickej triedy `Planner`, ktorá triedy zaväzuje k implementácii metód slúžiacich na komunikáciu s vizualizačnou triedou `VisControl`. Trieda `Planner` je parametrizovateľná všetkými potomkami abstraktnej triedy `MotionData`.
Po implementácii algoritmu RRT sa ukázalo, že výpočet plánovacieho algoritmu je potrebné presunúť do nového vlákna, nakoľko pri výpočte na jednom vlákne dochádzalo k strate responzivity GUI a k zamrznutiu vizualizácie, až do času, kedy sa výpočet cesty dokončil. Preto `Planner` rozširuje triedu `SwingWorker` z balíčku `java.swing`, pomocou ktorej je dosiahnuté vytvorenie samostatného vlákna pre vykonanie výpočtu plánovacieho algoritmu. Toto rozšírenie následne zaväzuje potomkov triedy `Planner` k implementácii metód `doInBackground, process` a `done`, ktoré slúžia, v tomto poradí, na: vykonanie práce vo vlákne, spracovanie možných medzivýsledkov za behu vlákna a vykonanie akcií po ukončení behu vlákna. Vo všeobecnom kontexte práce modulov plánovania cesty je ešte potrebné spomenúť triedu `MotionData` a triedu `VisContainer` z modulu `Container`. Trieda `MotionData` je abstraktná a neimplementuje žiadne metódy, slúži iba na zastrešenie tried reprezentujúcich stav pomyselného lietadla v danej iterácii plánovacieho algoritmu. Konkrétne triedy budú bližšie diskutované v sekciách príslušných algoritmov. Trieda `VisContainer` sa využíva na predávanie dát z plánovacích algoritmov triede zaisťujúcej vizualizáciu. Je možné parametrizovať túto triedu všetkými potomkami triedy `MotionData`. V inštancii sa predáva premenná typu `retTypes` z modulu VisControl indikujúca úspešnosť jedného kroku algoritmu, prípadne jeho ukončenie a zoznam objektov triedy `MotionData`, ktoré je potrebné vizualizovať.

V nasledujúcich podsekciách je priblížená implementácia plánovacích algoritmov a ich pomocných tried.

4.1.1 Rapidly-exploring Random Trees (RRT)

Na výpočte algoritmu RRT sa podieľajú 3 hlavné triedy `RRT`, `UAVMotionEquations`, `RungeKutta` a niekoľko pomocných tried definujúcich dátové štruktúry a rozhrania. V nasledujúcich podsekciách sú čitateľovi priblížené jednotlivé časti implementovaného algoritmu RRT.

Algoritmus RRT

Samotný algoritmus RRT je veľmi flexibilný a vhodný na plánovanie cesty alebo pohybu pre takmer ľubovoľného robota. RRT má niekoľko variánt líšiacich sa počtom stromov a ich smerom expanzie (z cieľa do počiatku, alebo opačne). V applete je implementovaná varianta s jediným stromom, ktorého koreňom je počiatočný bod nastavený cez GUI, a teda expanduje smerom ku konečnému bodu. Pseudokód algoritmu RRT s jedným stromom, uvedený v LaValle [12]:

```
1  RRT(xRoot)
2  {
3    Tree.init(xRoot)
4    for k := 1 to K do
5    {
6      xRand := RANDOM_STATE()
7      EXTEND(Tree, xRand)
8    }
9    return Tree
10 }
```

```
1  EXTEND(Tree, xRand)
2  {
3    xNear := NEAREST_NEIGHBOR(xRand, Tree)
4    if NEW_STATE(xRand, xNear, xNew, uNew) then
5      Tree.add_vertex(xNew)
6      Tree.add_edge(xNear, xNew, uNew)
7      if xNew = xRand
8        return Reached
9      else
10       return Advanced
11   return Trapped
12 }
```

V pseudokóde môžeme vidieť, že algoritmus RRT beží v konečnom cykle, kým nevyčerpá počet povolených krokov `K`. V kontexte tejto práce je počet pokusov obmedzený na 200 000. RRT v applete implementované tak, aby sa rozširovanie stromu vykonávalo v nekonečnom cykle až kým sa strom nerozrastie do dostatočnej blízkosti cieľového bodu, alebo sa nevyčerpajú pokusy. V prípade, že by chcel používateľ algoritmus ukončiť predčasne, môže použiť tlačidlo Stop dostupné z GUI appletu.

Implementácia stromu Nakoľko autor nenašiel v balíčku `java.util` implementáciu dátovej štruktúry typu strom, implementoval pre použitie v applete jednoduchú generickú triedu `Tree`, ktorá podporuje iba vkladanie prvkov a vrátenie postupnosti uzlov od koreňa po daný uzol. Keďže priechod stromom bol potrebný iba pre nájdenie najbližšieho uzlu stromu k náhodne zvolenej vzorke, bola funkcionalita prechodu stromom presunutá do triedy `RRT`. Nájdenie najbližšieho uzlu prehľadáva strom spôsobom `pre-order` a ako vzdialenosť medzi uzlom a vzorkom používa Euklidovskú vzdialenosť ich súradníc.

Vzorkovanie Vzorkovanie priestoru je implementované pomocou triedy `Random` z balíčku `java.util`. Postupne sa vygeneruje náhodné číslo od 0 do X pre každú súradnicu náhodného bodu, kde X je rozmer relevantnej dimenzie pracovného priestoru. Trieda `Random` zaručuje približne uniformné rozloženie vygenerovaných čísiel. Pokiaľ je vygenerovaný bod v kolízii s niektorou z prekážok, tak je zahodený a prejde sa ku generovaniu novej vzorky. Za účelom zrýchlenia konvergencie RRT ku koncovému bodu je za každú desiatu vzorku zvolený samotný koncový bod. Dôvodom pre urýchlenie konvergencie je prevencia pred výpočtom príliš veľkého počtu krokov a s tým spojenému zaplneniu plátna vizualizácie. To vedie k nadmernému zaťaženiu systémových prostriedkov, najmä pamäte, a spomalenému a trhanému zobrazovaniu ďalších geometrických tvarov na plátne.

Rozširovanie stromu Nové uzly stromu vznikajú použitím tzv. *pohybových rovníc*, ktoré zachytávajú kinematické možnosti lietadiel. Pri pridávaní nového uzlu sa najprv nájde uzol s najnižšou Euklidovskou vzdialenosťou od vzorky a následne sa pomocou týchto rovníc spraví 5 manévrov. Manéver, ktorý skončí najbližšie vzorke a nie je v kolízii so žiadnou prekážkou, je následne pridaný do stromu.

Pohybové rovnice

V rámci appletu boli implementované pohybové rovnice zachytávajúce kinematické možnosti lietadiel. Ide o rovnice 2.8 spomínané v podsekcii 2.2.2. Vstupom týchto rovníc je rýchlosť lietadla, uhol ktorý zviera vektor rýchlosti lietadla a horizont, uhol náklonu lietadla a gravitačné zrýchlenie. Hodnoty uhlov na vstupe sú obmedzené na maximálne 45°, čo sú hodnoty približne odrážajúce možnosti bezpilotných lietadel. Výstupom sú súradnice lietadla a uhol smeru letu lietadla meraný od severu, prípade tohto appletu uhol od osi X. Na aproximáciu hodnôt rovníc je použitá numerická metóda Runge-Kutta 4. rádu. Pohybové rovnice sú využité pri expandovaní stromu RRT, čo zaisťuje použiteľnosť výslednej cesty z kinematického hľadiska. Expanzia stromu prebieha cez spomínaných 5 manévrov, inak nazývaných aj *pohybovými primitívami*. Ide o manévre: stúpanie, klesanie, zmena smeru letu v smere a protismeru hodinových ručičiek a let rovno. Ukončenie aproximácie pohybových rovníc pri expanzii stromu nezáleží od času, ale od dosiahnutia očakávanej hodnoty jedného z výstupov rovníc. V prípade manévrov meniacich výšku beží aproximácia kým nedôjde k zmene o 1 jednotku na ose Y, pri manévroch zmeny smeru letu ide o zmenu uhlu smerovania o 90°a pri pohybe dopredu o preletenie 1 jednotky v smere letu. Ako sa aproximácia blíži k očakávanej hodnote prichádza k skracovaniu aproximačného kroku na jednu tretinu kroku predošlého až po minimálnu hodnotu kroku. Výstupom aproximácie pohybových rovníc je kontajner typu `MotionResultContainer` implementovaný pre tento applet v module `Container`. Obsahuje pole stavov, ktorými pri aproximácii model lietadla prešiel a stav, v ktorom aproximácia zastavila. Z dôvodu zníženia nárokov na pamäť sa takto ukladá každý

tretí stav do maxima 10 stavov. Stav je základnou dátovou štruktúrou v tejto implementácii algoritmu RRT, ktorá uchováva informácie o lietadle v danom momente výpočtu. Je implementovaný triedou `State` a obsahuje informácie o vstupoch a výstupoch pohybových rovníc, ako aj súradnice v priestore z daného kroku aproximácie pohybu lietadla. Trieda `State` rozširuje všeobecnú triedu `MotionData`.

Detekcia kolízií Detekcia kolízie aproximovaného pohybu je založená na výstupe rovníc, kedy sa každý uložený stav otestuje na kolíziu so všetkými prekážkami. Nakoľko sa neukladá každý stav, je možné, že dôjde ku kolízii, ktorú algoritmus nezaregistruje. V dôsledku tejto nedokonalosti vznikajú kolízie hlavne pri pohybe veľmi blízkom prekážke alebo pri oblietaní rohu prekážky. Z hľadiska použiteľnosti appletu však toto autor nepovažuje za kritický problém, nakoľko nedetekované kolízie nie sú veľmi početné. Naviac existuje možnosť, že ich odstránenie by bolo výpočetne náročnejšie na pamäť a procesor, čo by mohlo viesť k zníženiu responzivity appletu, čo autor považuje za kritickejšie z hľadiska jeho použitia.

Runge-Kutta 4. rádu

Algoritmus Runge-Kutta 4. rádu je implementovaný v triede `RungeKutta`, a ako bolo spomenuté, je v applete používaný na aproximáciu hodnôt pohybových rovníc. Aproximácia hodnoty diferenciálnej rovnice pomocou Runge-Kutta 4. rádu je definovaná ako:

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \\ k_3 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2) \\ k_4 &= hf(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 \end{aligned} \tag{4.1}$$

kde h je krok, x_n je vstup a y_n výstup diferenciálnej rovnice danej funkciou f. Pri implementácii sa používa rozhranie `DiffEquation` poskytujúce pole pre prípad, keď majú diferenciálne rovnice viacero vstupných hodnôt.

Vyhladenie nájdenej cesty

Algoritmus RRT zriedka vytvára optimálne cesty kvôli náhodnému charakteru vyberania vzoriek, preto je vhodné nájdenú cestu istým spôsobom vylepšiť. V publikácii Geraerts [6] sú tri spôsoby vyhladzovania: *Path pruning*, *Shortcuts Partial Shortcuts*. V applete je implementovaná metóda *Shortcuts*, ktorá sa pokúša nájsť lepšiu cestu medzi dvoma uzlami x a y, kde y je aspoň o dva uzly ďalej ako x. Cestu hľadá pomocou lokálneho plánovania. V kontexte tejto práce je lokálne plánovanie vykonávané rovnako pomocou pohybových rovníc. Narozdiel od plánovania pôvodnej cesty sa tu nepoužívajú pohybové primitíva, ale algoritmus sa pokúša postupne meniť vstupy pohybových rovníc tak, aby sa mu podarilo vybrané uzly prepojiť. V prípade úspechu lokálneho plánovaču sa úsek cesty medzi uzlami nahradí novo vypočítanou cestou, v opačnom prípade sa lokálna cesta zahodí.

4.1.2 Naivná bunková dekompozícia

Algoritmus naivnej bunkovej dekompozície je implementovaný v dvoch triedach: v triede `CellDecomposition`, ktorá obsahuje samotnú implementáciu algoritmu a v triede `Cell`, ktorá reprezentuje objekt bunky.

Dekompozícia

Prvým krokom dekompozície je určenie veľkosti jednotlivých buniek. Rozmery buniek sú vypočítané tak, aby zachytávali kinematické možnosti lietadiel. Podľa Hwangbo [8]:

$$\begin{aligned} cellX &= \frac{minTurnRadius}{1.5} \\ cellY &= cellX * \arctan(maxClimbAngle) \\ cellZ &= cellX \end{aligned} \tag{4.2}$$

Hodnota `maxClimbAngle` je daná konštantou `fpAscent` z triedy `UAVMotionEquations` a `minTurnRadius` je podľa leteckej príručky [4] definovaný ako:

$$minTurnRadius = \frac{velocity}{\tan(maxBankAngle) * g} \tag{4.3}$$

všetky hodnoty *velocity*, *maxBankAngle* a *g* sú prevzaté z triedy `UAVMotionEquations`. Samotné bunky sú v applete reprezentované triedou `Cell`, ktorá rozširuje spomínanú všeobecnú triedu `MotionData` a implementuje rozhranie `Comparable`. Dodržanie tohto rozhrania je nutné pre možnosť radiť a kontrolovať inštancie triedy `Cell` na duplikáty. Duplicitné bunky sa odhalujú na základe ich koordinátov a radenie buniek prebieha podľa hodnoty vybranej heuristiky pre algoritmus A*. Každá inštancia triedy `Cell` v sebe uchováva informácie o pozícii bunky v konfiguračnom priestore, hodnoty heuristík pre použitie v algoritme A*, index bunky v poli `cells`, predošlú bunku, smer pohybu a premennú typu `boolean` indikujúcu, či sa bunka nachádza v kolízii s niektorou z prekážok v priestore. Naviac každá inštancia bunky obsahuje aj objekt typu `Box` z knižnice Java3D, ktorý určuje rozmery bunky a jeho vlastnosti sa používajú aj na detekciu kolízii. Smer pohybu lietadla je zakódovaný v enumerácii, ktorá definuje 24 možných smerov letu - všetky svetové strany, svetové strany spolu s naberaním výšky a svetové strany spolu s klesaním.
Po určení rozmerov buniek je spravená dekompozícia priestoru. Pri tomto procese sa postupne inicializujú jednotlivé bunky a ukladajú sa do spomínaného trojrozmerného poľa `cells`, ktorého rozmery sa vypočítajú ako:

$$width = \frac{xDim}{cellX} \qquad height = \frac{yDim}{cellY} \tag{4.4}$$

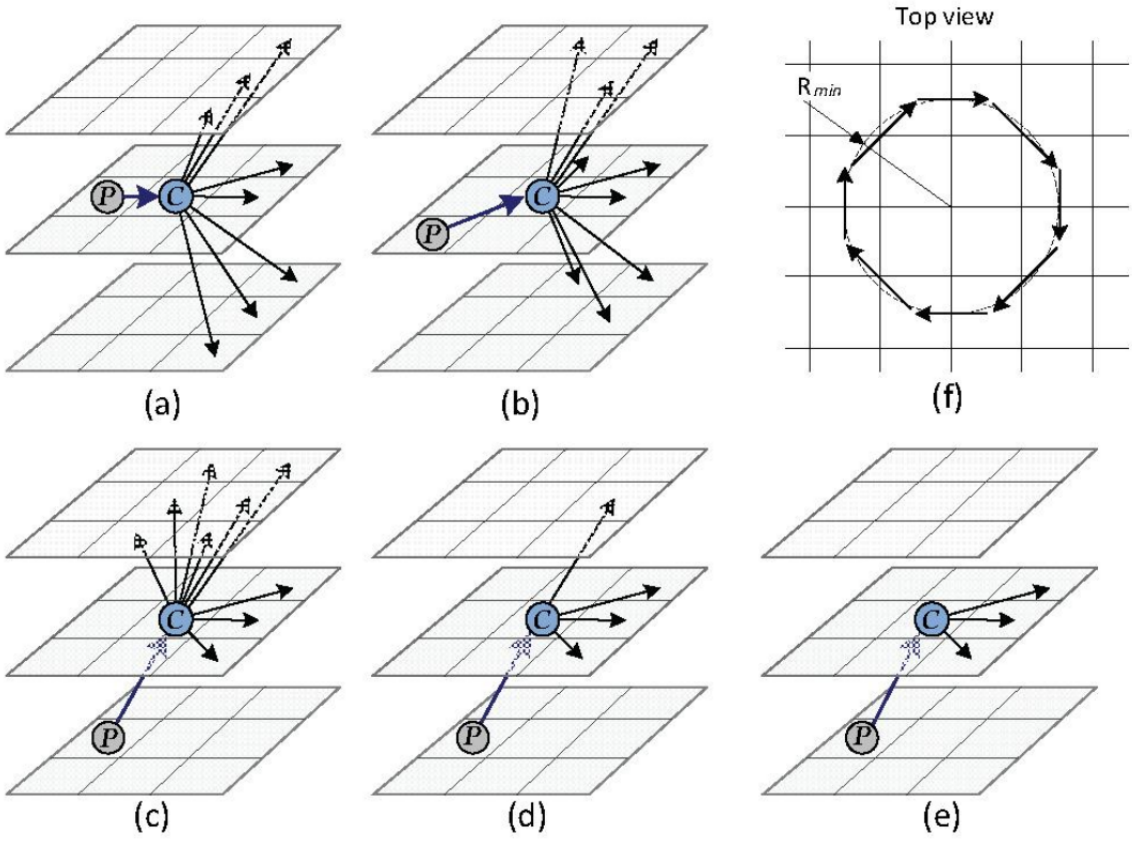
kde *xDim* a *yDim* sú rozmery pracovného priestoru a hodnoty *height* a *widht* sú zaokrúhlené nahor na najbližšie celé číslo. Bunky sú navrhnuté tak, aby sa ich X a Z rozmery rovnali, preto nie je nutné vypočítavať hĺbku poľa. Zároveň sa testuje či sa bunky nenachádzajú v kolízii s prekážkami. Následne sa prejde k samotnému hľadaniu cesty implementovanému algoritmom A*.

Algoritmus A*

A* je v algoritme naivnej bunkovej dekompozície použitý na nájdenie cesty v priestore diskretizovanom do buniek. Zoznamy `open` a `closed` sú implementované typom `HashSet` z

balíčku `java.util`, ktorý ukladá prvky na základe ich hash kódu. Táto štruktúra nepovoľuje duplikáty. Objekty v štruktúre typu `HashSet` však nie je možné radiť, čo je nutné pre správne fungovanie algoritmu. Preto je implementovaný ešte zoznam objektov `openSorted` typu `ArrayList`, ktorý je presnou kópiou `open` ale je zoradený vzostupne podľa danej heuristiky. Heuristiky `startScore` a `goalScore` sú dané ako vzdialenosť bunky od počiatočnej, prípadne cieľovej bunky. Ako heuristická funkcia heuristiky `goalScore` je použitá Euklidovská vzdialenosť medzi koordinátmi danej a cieľovej bunky. Hodnota `startScore` je daná akumulovanou cenou pohybov z počiatočného bodu algoritmu do danej bunky. Cena pohybu medzi susednými bunkami záleží na smere pohybu. Menenie výšky, zatáčanie a kombinácia týchto pohybov sú penalizované vyššou cenou, aby sa minimalizovali možné nepotrebné pohyby.

Expanzia buniek Susedné bunky sú definované ako bunky, do ktorých sa lietadlo môže z danej východzej bunky dostať. Tieto pohybové možnosti sú dané expanziou buniek. Expanzia buniek prebieha na základe smeru letu lietadla. V prípade, že susedná bunka sa už na-



Obr. 4.1: Expanzia buniek: Tento obrázok zobrazuje niekoľko možností expanzie buniek v závislosti od smeru pohybu lietadla. V kontexte tejto práce boli zvolené expanzie (a), (b) a (d), ktoré najlepšie zachytávajú kinematické možnosti lietadiel. Obrázok (f) ilustruje zatáčanie lietadla s minimálnym polomerom. Zdroj: Prevzaté z článku Hwangbo [8].

chádza v `openSet`, je potrebné skontrolovať, či má aktuálna cesta lepšiu cenu. Pokiaľ áno, tak dôjde k upraveniu záznamu v zoznamoch `open` a `openSorted` a k zoradeniu objektov

`openSorted`. Ak je susedná bunka už v zozname `closed`, tak sa preskočí. Po vykonaní expanzie bunky sa vráti zoznam susedných buniek v objekte triedy `visContainer` spolu s návratovým kódom a pošle sa na vizualizáciu.

Detekcia kolízií Detekcia kolízií prebieha použitím objektov triedy `Box` uložených v bunkách. Každému z týchto objektov je pridelená inštancia triedy `BoundingBox` z balíčku `javax.media.j3d`, ktorá poskytuje metódu `intersect`. Práve táto metóda potom slúži na detekciu kolízií. Detekcia prebieha v algoritme naivnej bunkovej dekompozície iba pri diskretizácii priestoru, nakoľko je už vtedy známe, ktoré bunky sú v kolízii. Za behu algoritmu A^* potom tieto kolízne bunky nie sú brané do úvahy.

Beh algoritmu Krokovanie algoritmu naivnej bunkovej dekompozície sa nevykonáva v samostatnom vlákne, nakoľko pri krokovaní nepríchádza k postrehnuteľnému blokovaniu GUI. Pri krokovaní tlačidlom Step sa najprv vykoná dekompozícia konfiguračného priestoru a inicializácia štruktúr pre algoritmus A^*. Následne prebehne vizualizácie diskretizovaného priestoru. Potom dochádza opakovane k výpočtu jedného kroku algoritmu A^* a jeho vizualizácii až do doby, kedy návratový kód signalizuje úspešné dokončenie algoritmu. Potom sa zvýraznia bunky tvoriace nájdenú cestu a odomkne sa tlačidlo vizualizácie cesty pomocou Béziérových kriviek, ako aj ostatné prvky GUI. Vykonanie celého algoritmu tlačidlom Run je v podstate identické, iba sa spúšťanie ďalších krokov robí automaticky a celý algoritmus prebieha v samostatnom vlákne.

4.2 Vizualizácia

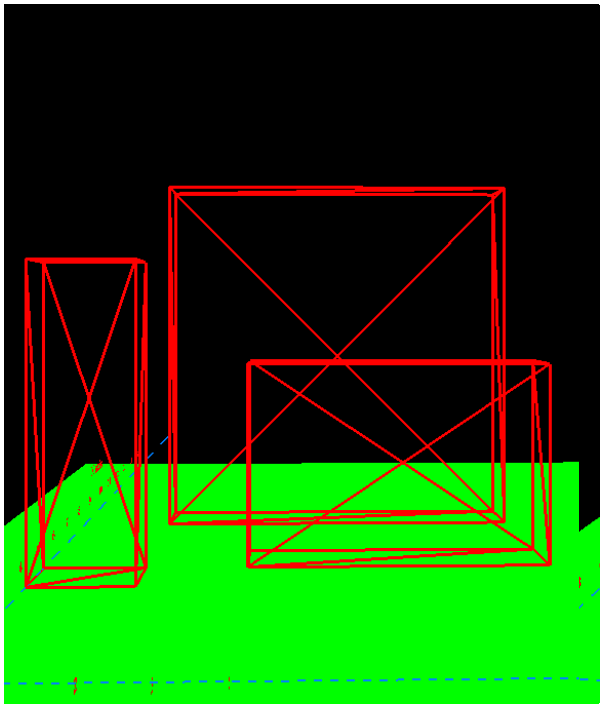
Vizualizácia prebieha v triede `VisControl`, ktorá zároveň slúži aj na komunikáciu medzi prvkami GUI a triedami plánovacích algoritmov. Na vizualizáciu sa v práci používa knižnica Java3D distribuovaná pod licenciou GPL. Základná štruktúra jej fungovania je načrtnutá v podskecii 4.2.1. V rámci triedy VisControl je implementovaná aj privátna trieda `DrawWorker`, ktorá je podobne ako triedy plánovacích algoritmov potomkom `SwingWorker` a umožňuje popri vykonávaní plánovacieho algoritmu vizualizovať jeho jednotlivé kroky v samostatnom vlákne, aby sa predišlo blokovaniu GUI. VisControl ďalej poskytuje enumeráciu `retTypes` slúžiacu na indikovanie úspešnosti kroku algoritmu, prípadne jeho dokončenie. Možnými hodnotami enumerácie sú SUCCESS, FAILURE, END.

4.2.1 Java3D

Princípom vykresľovania scény v Java3D je tzv. *scénový graf*, ktorý reprezentuje objekty určené na vykreslenie. Scénový graf je štruktúrovaný ako orientovaný acyklický graf skladajúci sa z uzlov, ktoré môžu byť buď skupinami alebo listami. Skupina združuje jednu alebo viac uzlov potomkov, ale vždy má maximálne jednen rodičovský uzol. List potom obsahuje už konkrétne definície geometrických tvarov, svetiel atď. Na vrchole grafu je superštruktúra `VirtualUniverse`, ktorá sa skladá zo zoznamu objektov `Locale`. Locale slúži ako kontajner podgrafov, ktoré sú reprezentované stromami s koreňom typu `BranchGroup`. Najčastejšími uzlami v podgrafe bývajú objekty typu `TransformGroup`, ktoré upravujú pozíciu, orientáciu a škálu im podradených objektov.

4.2.2 Scéna appletu

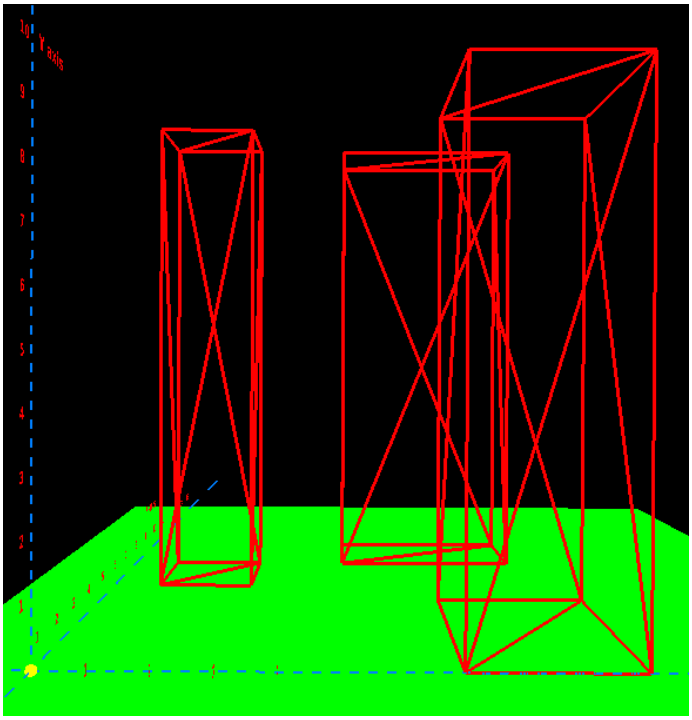
Pri spustení appletu pozostáva scéna z iba z plochy predstavujúcej zem, niekoľkých kvádrov predstavujúcich prekážky, troch ôs slúžiacich na indikáciu pracovného prostredia a dvoch gúľ predstavujúcich štart a cieľ. Ovládanie kamery v scéne je zabezpečené myšou. Nie je možné kameru v rámci scény presúvať, iba meniť uhol pohľadu okolo pevne daného stredu rotácie a zvyšovať, prípadne znižovať priblíženie kamery. Pôvodne scéna knižnice Java3D poskytuje možnosť rotácie okolo ľubovoľnej osi, o ľubovoľný uhol. Nakoľko táto možnosť mohla viesť k scéne obrátenej dole hlavou a iným, pre používateľa potenciálne mätúcim situáciam bolo autorovou snahou obmedziť možné rotácie scény. S týmto zámerom autor prebral z knižnice Java3D triedu `OrbitBehavior` definujúcu správanie kamery a pokúsil sa o obmedzenie rotácie okolo osi X iba na 90°. Toto obmedzenie síce v applete funguje, avšak sa nepodarilo obmedzenie rozšíriť aj na diagonálnu rotáciu okolo osi XY, nakoľko vtedy prichádza ku kombinácii rotačných matíc pre os X aj pre os Y a teda je v tomto smere možné dostať kameru do nevyhovujúceho uhlu. Applet ponúka aj tlačidlo Reset, ktoré reinicializuje scénu do predvoleného stavu a nastaví kameru do pôvodnej pozície. Na výber sú používateľovi dané tri prostredia s rôzne rozmiestnenými prekážkami, medzi ktorými môže prepínať pomocou tlačidiel Preset1 – Preset3.



Obr. 4.2: Prostredie 1.

Obr. 4.3: Prostredie 2.

Prekážky Prekážky v applete sú dané ako staticky umiestnené kvádre, ktorých zoznam sa predáva medzi triedami vizualizácie a plánovacích algoritmov za účelom detekcie kolízií. Tento zoznam je implementovaný dostatočne všeobecne, aby zvládol ukladať ľubovoľný geometrický tvar z knižnice Java3D, takže by bolo možné tvar prekážok zmeniť nie len pre všetky, ale aj pre každú prekážku individuálne. Možným rozšírením tohto appletu by mohla byť možnosť výberu tvaru a umiestnenia prekážok používateľom.



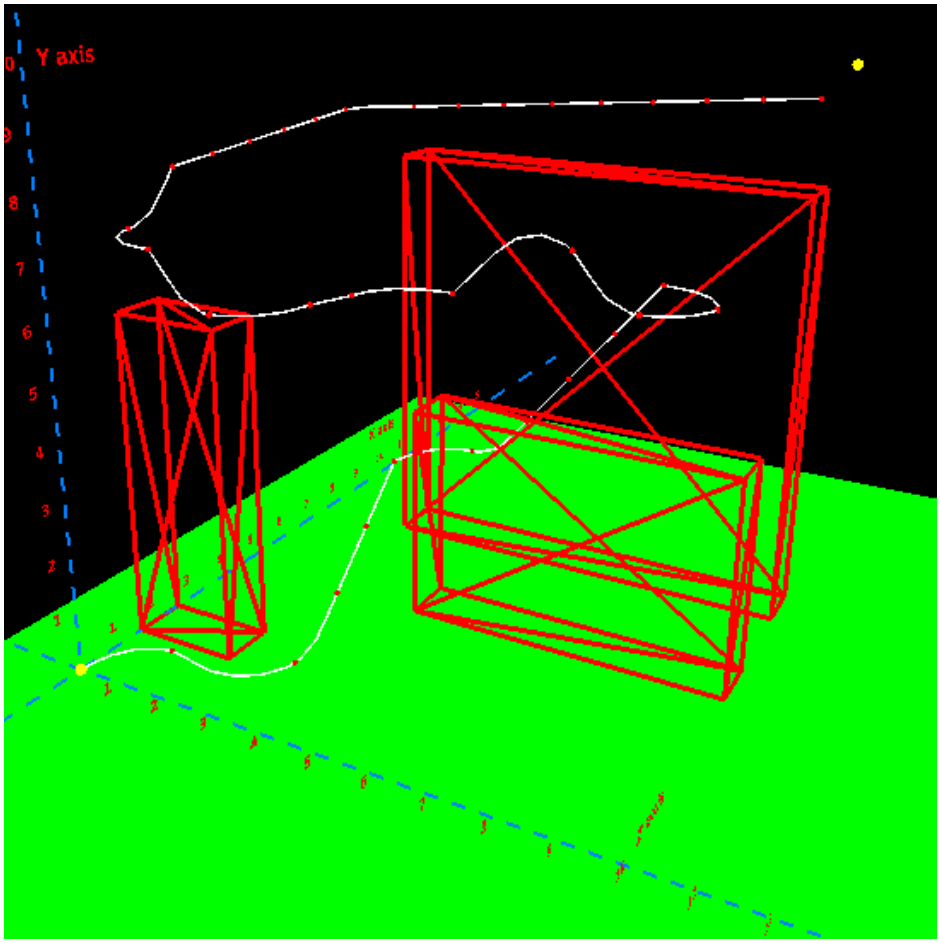
Obr. 4.4: Prostredie 3.

4.2.3 Vizualizácia algoritmov

Vizualizácia algoritmov prebieha v samostatnom vlákne appletu. Vizualizácia prebieha na základe kontajneru typu `visContainer`, ktorý obsahuje buď objekty buniek `Cell`, alebo stavov `State` v závislosti od práve prebiehajúceho plánovacieho algoritmu. Vykresľovanie sa po každom vykreslenom kontajneri uspí na čas `delay` milisekúnd, ktoré sa nastavuje na posuvníku Delay v GUI appletu. Toto správanie je implementované pre prípad, že vykonanie celého plánovacieho algoritmu prebehne príliš rýchlo a používateľ by tak nestihol zaregistrovať jednotlivé kroky algoritmu. Uspaním vykresľovania na určitý krátky čas sa tak umožní používateľovi lepšie si prezrieť kroky algoritmu. Algoritmy sú vizualizované osobitnými spôsobmi popísanými v nasledujúcich podsekciách.

Vizualizácia RRT

Na vizualizáciu cesty a pohybových primitív v algoritme RRT mali pôvodne byť použite kubické Béziérove krivky, avšak zvolenie vhodných kontrolných bodov sa ukázalo byť pomerne zložitým. Problematickými boli konkrétne body určujúce zakrivenie Béziérovej krivky, nakoľko by museli byť nejakým spôsobom vypočítané z počiatočného a koncového bodu krivky. Riešením tohto problému sa ukázala byť trieda `LineStripArray` knižnice Java3D. Táto trieda vykresľuje ľubovolnú čiaru jej rozdelením na množstvo menších vzájomne spojených úsečok. Pre úspešné vykreslenie je potrebné objektu triedy `LineStripArray` poskytnúť pole celých čísel, ktoré určujú počet vertexov X potrebných na vykreslenie jednej úsečky a samotné pole vertexov, v ktorom každých X záznamov určuje jednu úsečku. Body použiteľné ako vertexy pre `LineStripArray` je možné získať v rámci aproximácie letu lietadla pohybovými rovnicami, zaznamenaný je každý tretí až desiaty bod v závislosti od kroku aproximácie.

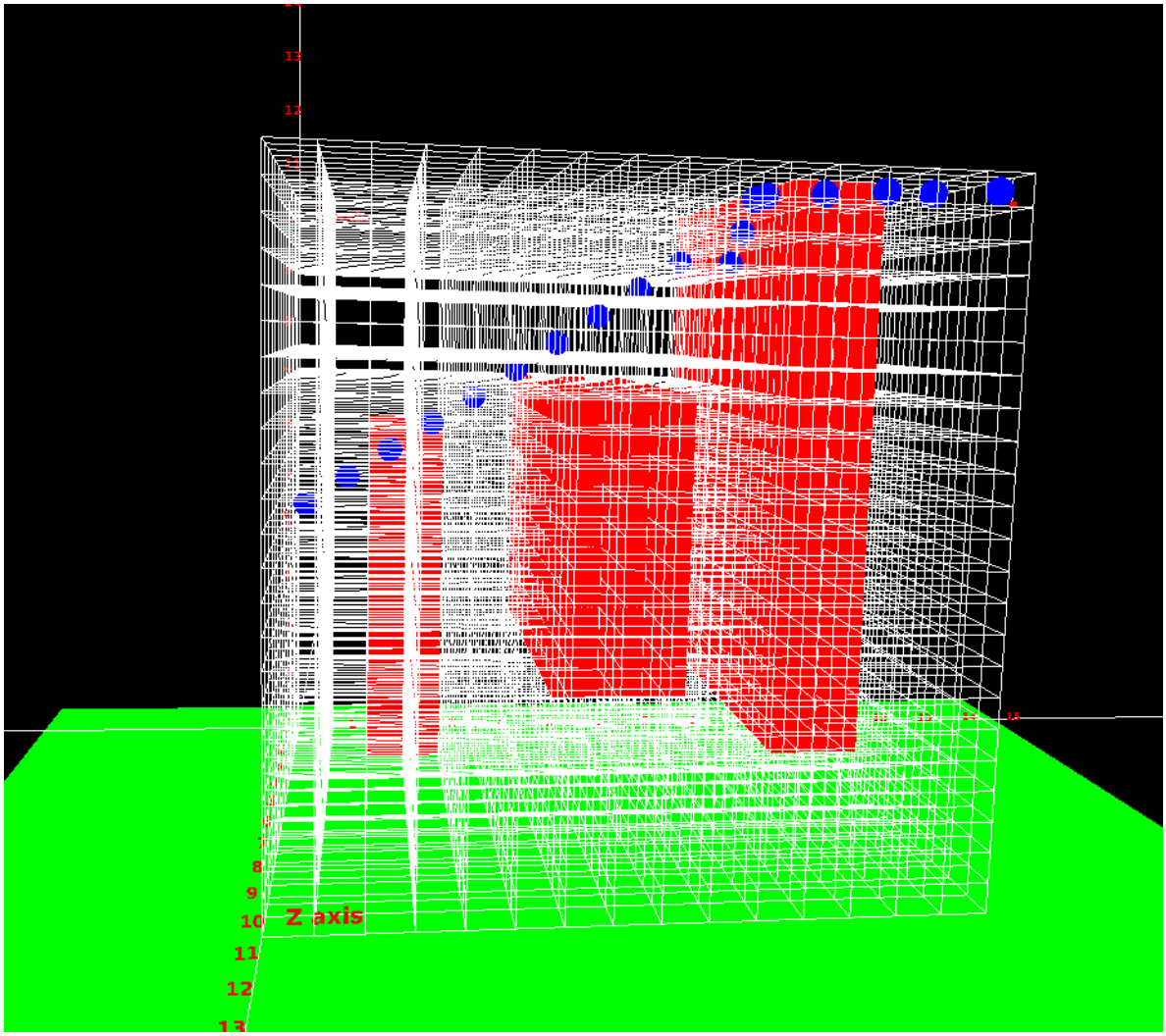


Obr. 4.5: Vizualizácia cesty nájdenej algoritmom RRT.

Vizualizácia naivnej bunkovej dekompozície

Vizualizácia bunkovej dekompozície sa ukázala byť podstatne jednoduchšia, nakoľko pozícia bunky je v priestore pevne daná. V princípe ide iba o vybranie buniek zo zoznamu a následné vykreslenie objektu typu `Sphere` z knižnice Java3D na súradnice bunky. Samotná dekompozícia je vizualizovaná ako 3D mriežka, kde kolízie buniek s prekážkami sú zobrazené ako červené kvádre s rozmermi buniek. Na vizualizáciu buniek spracovávaných v rámci hľadania cesty bola vybraná guľa menších rozmerov ako samotná bunka, nakoľko pri vykreslení v rozmeroch bunky dochádzalo k zakrytiu buniek vo vnútri diskretizovaného priestoru. Farba vykreslovaných objektov je modifikovaná triedami `Appearance` a `Material` knižnice Java3D s cieľom farebne odlíšiť bunky s lepšiou cenou v rámci algoritmu A*.

Po nájdení cesty je možné tlačidlom Special vizualizovať nájdenú cestu pomocou kvadratických Béziérových krivek. Táto vizualizácia nie je úplne presná, slúži skôr na približné zobrazenie. Orientácia a zakrivenie kriviek sú dané smerom pohybu lietadla potrebným na presun do nasledujúcej bunky. V rámci tejto vizualizácie Béziérovými krivkami sa nerozlišuje pozícia počiatočného bodu, ale iba súradnice počiatočnej bunky. Preto vykresľovanie krivky vždy začína v ľavom dolnom rohu tejto bunky.



Obr. 4.6: Vizualizácia cesty nájdenej algoritmom naivnej bunkovej dekompozície.

Kvadratické Béziérove krivky Kvadratické Béziérove krivky sú dané troma kontrolnými bodmi, z ktorých prvý a posledný bod udávajú počiatok a koniec krivky a sú interpolované, zatiaľ čo prostredný bod určuje zakrivenie krivky a je aproximovaný. Výpočet bodov, ktorými Béziérova krivka prechádza, je daný vzťahom:

$$B(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2 \tag{4.5}$$

kde $P_{0..2}$ sú kontrolné body a parameter $t \in < 0, 1 >$. Váhy kontrolných bodov na jednotlivé body krivky sa dajú vypočítať ako:

$$\begin{aligned} b_0(t) &= (1-t)^2 \\ b_1(t) &= 2t(1-t) \\ b_2(t) &= t^2 \end{aligned} \tag{4.6}$$

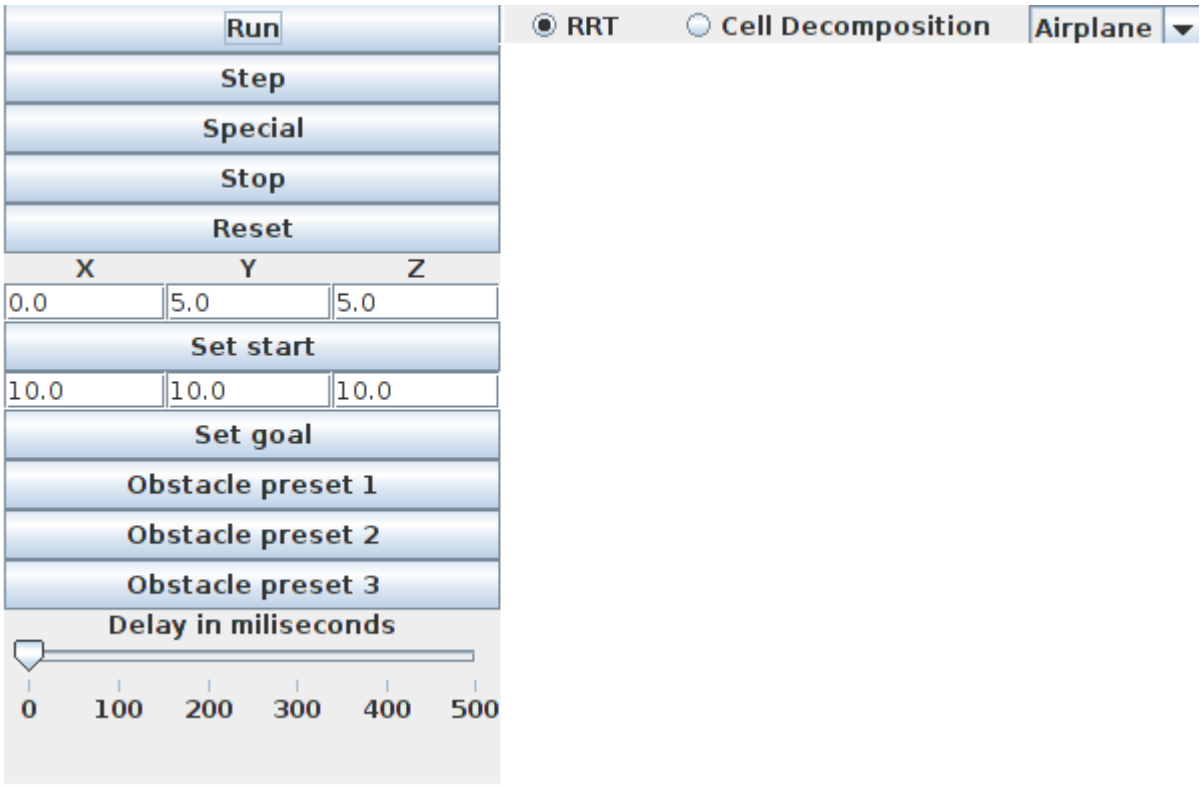
Vykresľovanie kvadratickej Béziérovej krivky je implementované v metóde `renderBezierCurve`, algoritmus vypočíta 16 bodov krivky pomocou a následne s použitím objektu typu `LineStripArray`, z knižnice Java3D, krivku vykreslí.

4.3 Grafické uživateľské rozhranie

Grafické uživateľské rozhranie je implementované v triede APPLETGUI, ktorá rozširuje triedu `JApplet`, čo aplikácii prepožičiava vlastnosti java appletu. Jednotlivé komponenty GUI sú implementované pomocou balíčku `java.swing` a ich umiestnenie je definované cez triedu `GridBagLayout`. Tá vytvorí pomyselnú mriežku, do ktorej buniek je potom možné umiestniť dané komponenty. Konkrétna pozícia komponentu a vlastnosti bunky, v ktorej sa nachádza, ako napr. vyplnenie bunky a zarovnanie textu, sú potom určené inštanciou triedy `GridBagConstraints` pri definícii komponenty.

Interakcie s inými modulmi Grafické uživateľské rozhranie komunikuje iba s triedou `VisControl`. Pri použití niektorého z prvkov GUI sa zavolá relevantná metóda `VisControl`, kde sa následne vykoná požadovaná akcia alebo je tento požiadavok preposlaný do jednej z ostatných tried. Po spustení vizualizácie vybraného algoritmu dôjde k zablokovaniu určitých prvkov GUI na trvanie behu vizualizácie, čím sa zabráni manipulácii s nastaveniami. Tie je potrebné nastaviť ešte pred spustením vybraného algoritmu. Po ukončení alebo prerušení vizualizácie trieda `VisControl` zabezpečí opätovné odomknutie elementov GUI.

4.3.1 Prvky grafického uživateľského rozhrania



Obr. 4.7: Ovládacie prvky grafického uživateľského prostredia.

Komponenty GUI môžme rozdeliť na nastavenia a na ovládacie prvky. Na ovládanie appletu slúžia tlačidlá Run, Step, Reset, Stop a Special, ktorými sa kontroluje spúšťanie vizualizácie algoritmov. Ostatné komponenty, okrem plátna, slúžia na nastavenie vstupných

parametrov vizualizácie a vizualizovaných algoritmov. Plátno slúži zobrazenie vizualizácie a ovládanie kamery v scéne.

Ovládanie Applet umožňuje dva spôsoby vizualizácie algoritmov, a to normálnym spustením celého algoritmu alebo jeho krokovaním. Na to slúžia tlačidlá Run a Step. Po spustení vizualizácie sa zablokujú všetky prvky GUI okrem tlačidla Stop a v prípade krokovania tlačidla Step, aby sa zabránilo meneniu vstupných parametrov za behu a iným nepodporovaným použitiam appletu. Prvky GUI sa po dokončení vizualizácie automaticky odblokujú. Tlačidlo Stop slúži na zastavenie vykonávania vizualizácie. Tlačidlo Reset uvedie plátno do počiatočného stavu a nastaví uhol pohľadu na východzí. Tlačidlo Special má osobitnú funkciu pre každý z implementovaných algortimov. Po ukončení vykonávania algoritmu RRT toto tlačidlo spustí vyhladenie nájdenej cesty a v prípade algoritmu naivnej bunkovej dekompozície vykreslí z nájdenej postupnosti buniek približnú cestu pomocou Béziérových kriviek. Tlačidlá Preset1, Preset2 a Preset3 prepínajú medi implementovanými konfiguráciami prekážok. Uhol kamery na plátne sa ováda myšou.

Nastavenia V ľavom hornom rohu appletu sú situované dve navzájom sa vylučujúce rádiové tlačidlá, ktoré označujú práve vybraný algoritmus plánovania cesty. Po spustení vizualizácie sa bude demonštrovať práve tu vybraný algoritmus. Menu v pravom hornom rohu slúži na výber pohybového modelu, ktorý definuje pohybové možnosti, s ktorými použité algoritmy plánovania cesty počítajú. V rámci tejto práce bol implementovaný iba pohybový model podobný bezpilotným lietadlám. Hodnota nastavená na posuvníku Delay určuje čas v milisekundách, ktorý vizualizačný modul čaká pri automatickej vizualizácii plánovacieho algoritmu medzi vykreslením jednotlivých krokov. Polia nad tlačidlami Set start a Set goal určujú súradnice počiatočného a koncového bodu v postupnosti x, y, z, medzi ktorými má vybraný algoritmus nájsť cestu.

4.4 Testovanie

Implementované algoritmy boli testované na základe množstva prehľadaných buniek/vzorkov, počtu uzlov v nájdenej ceste a približnej dĺžke nájdenej cesty. Dĺžka nájdenej cesty bola v tomto testovaní počítaná ako Euklidovská vzdialenosť medzi uzlami cesty. Nakoľko je algortimus RRT do určitej miery náhodný, bolo jeho testovanie opakované vždy trikrát s cieľom získať lepší prehľad o jeho výsledkoch. Cesty nájdené algoritmom RRT boli zmerané pred aj po ich vyhladení, aby bolo možné zmerať úspešnosť implementovaného vyhladzovacieho algoritmu. V rámci testovania boli merané nájdené cesty medzi bodmi $P_{S1}(0,0,0)$[1], $P_{G1}(10,10,10)$; $P_{S2}(0,5,5), P_{G2}(10,5,5)$ a $P_{S3}(0,0,5), P_{G3}(10,0,8)$. Merania boli uskutočnené pre každé implementované prostredie. Výsledky testovania sú usporiadané do tabuliek 4.1, 4.2, 4.3 podľa jednotlivých prostredí.

Z tabuliek môžme vidieť, že algoritmus RRT prehľadáva niekoľkonásobne viac bodov v priestore ako algoritmus naivnej bunkovej dekompozície. Tiež môžme pozorovať, že cesty nájdené pomocou RRT sa často značne odlišujú. Toto je dané vzorkovacou povahou RRT. Na druhú stranu, ako môžme vidieť v prípade priestoru č. 2, bodov $P_{S1}(0,0,0), P_{G1}(10,10,10)$ a $P_{S3}(0,0,5), P_{G3}(10,0,8)$, je RRT úspešnejšie pri riešení problémov vyžadujúcich vyššiu precíznosť. V prípade bodu $P_{S3}(0,0,5), P_{G3}(10,0,8)$, v priestore č. 2, dokonca algoritmus naivnej bunkovej dekompozície zlyhá pri hľadaní cesty. Toto zlyhanie je predikované samotnou dekompozíciou priestoru, kedy potenciálne voľný priestor je považovaný za kolízny kvôli tomu, že existuje prekážka, ktorá malou časťou zasahuje do danej bunky. Takto ignorovaný priestor potom môže chýbať pri zmenách smeru alebo výšky, ako tomu je aj v tomto prípade. V tabuľke sú zobrazené výsledky naivnej bunkovej dekompozície, ktorá pre vierohodnosť vyžadovala istú minimálnu výšku expandovaných buniek. Bola testovaná aj verzia, ktorá nevyžadovala minimálnu výšku, avšak jej výsledky boli až na jeden testovaný prípad totožné. Dekompozícia nevyžadujúca minimálnu výšku našla kratšiu cestu iba v prípade priestoru č. 1, bodu $P_{S1}(0,0,0), P_{G1}(10,10,10)$.

Pri testovaní sme ďalej mohli odpozorovať, že implementované vyhladzovanie cesty nájdenej algortimom RRT nebolo konzistentne úspešné. Najväčším úspechom vyhladzovania bolo skrátenie cesty z 50.25 na 27.21 a z 36.70 na 20.15, avšak všeobecne medzi ostatnými testovanými prípadmi neprichádzalo k výraznému skráteniu cesty, naopak v niektorých prípadoch došlo k marginálnemu predĺženiu.

V nasledujúcich tabuľkách značí stĺpec *Searched* počet prehľadaných uzlov, *Nodes* počet uzlov v nájdenej ceste, *Length* približnú dĺžku cesty a v stĺpcoch *Nodes** a *Length** sú hodnoty po vyhladení cesty nájdenej algoritmom RRT.

[1] Algortimus Naivnej bunkovej dekompoozície štartoval v prostredí 1 z bodu (0,0,1.5), nakoľko inak sa dostal do okamžitej kolízie s prekážkou

Konfigurácia štart/cieľ	Plánovač	Merané hodnoty				
		Searched	Nodes	Length	Nodes*	Length*
start(0,0,0) – goal(10,10,10)	RRT Run 1	11055	30	39.81	21	40.60
	RRT Run 2	20407	28	37.18	8	34.98
	RRT Run 3	4907	26	36.7	6	20.15
	Naive CD	562	26	32.26	—	—
start(0,5,5) – goal(10,5,5)	RRT-Run 1	590	14	17.53	9	17.63
	RRT-Run 2	138842	40	50.25	17	27.21
	RRT-Run 3	1985	13	18.07	10	18.55
	Naive CD	590	13	15.41	—	—
start(0,0,5) – goal(10,0,8)	RRT Run 1	11246	19	25.37	11	21.46
	RRT Run 2	7393	22	27.42	12	27.42
	RRT Run 3	3210	14	20.76	11	21.72
	Naive CD	139	16	18.83	—	—

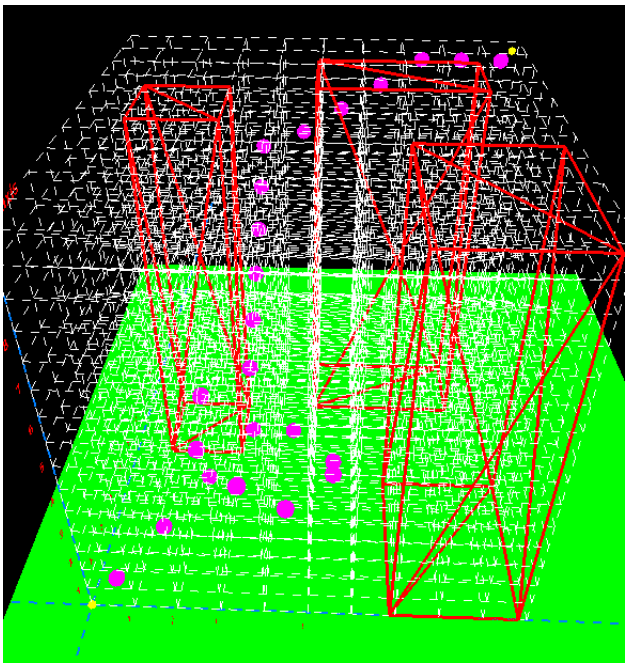
Tabuľka 4.1: Výsledky testov v prostredí 1.

Konfigurácia štart/cieľ	Plánovač	Merané hodnoty				
		Searched	Nodes	Length	Nodes*	Length*
start(0,0,0) – goal(10,10,10)	RRT Run 1	6373	15	20.61	11	21.08
	RRT Run 2	1753	17	21.27	11	20.95
	RRT Run 3	17080	31	41.31	21	41.91
	Naive CD	771	26	34.95	—	—
start(0,5,5) – goal(10,5,5)	RRT-Run 1	1179	11	13.98	9	14.59
	RRT-Run 2	936	11	15.90	7	13.39
	RRT-Run 3	13277	19	26.39	15	26.73
	Naive CD	82	10	9.71	—	—
start(0,0,5) – goal(10,0,8)	RRT Run 1	2354	19	25.89	10	25.69
	RRT Run 2	3186	20	25.56	18	26.35
	RRT Run 3	895	17	23.21	17	23.21
	Naive CD	n/a	n/a	n/a	—	—

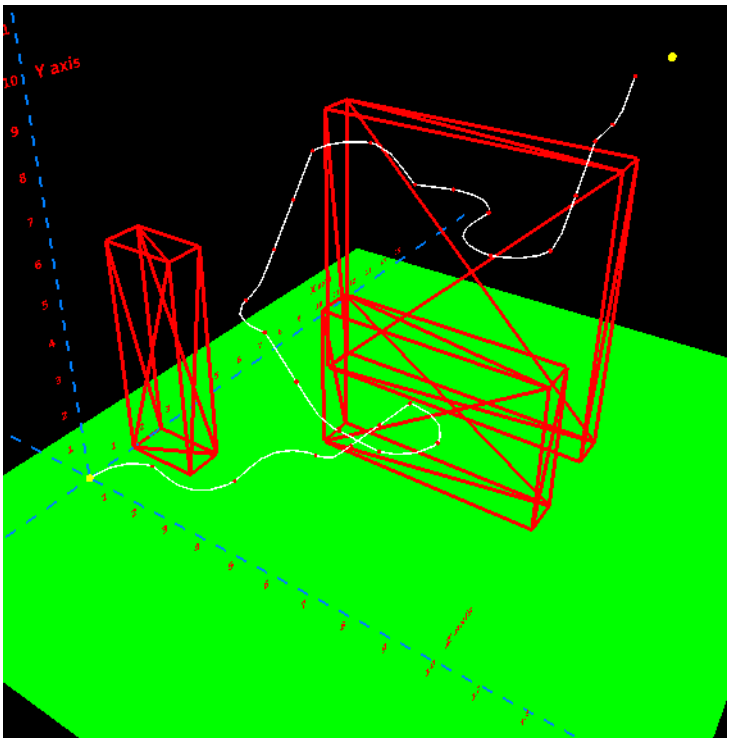
Tabuľka 4.2: Výsledky testov v prostredí 2.

Konfigurácia štart/cieľ	Plánovač	Merané hodnoty				
		Searched	Nodes	Length	Nodes*	Length*
start(0,0,0) – goal(10,10,10)	RRT Run 1	2760	15	20.36	8	20.17
	RRT Run 2	15807	28	35.81	8	22.19
	RRT Run 3	15383	44	58.43	6	20.39
	Naive CD	416	23	28.52	—	—
start(0,5,5) – goal(10,5,5)	RRT-Run 1	1204	16	20.92	16	20.92
	RRT-Run 2	13453	13	16.41	13	16.41
	RRT-Run 3	582	17	23.03	10	22.42
	Naive CD	186	10	10.38	—	—
start(0,0,5) – goal(10,0,8)	RRT Run 1	4002	13	15.89	10	15.68
	RRT Run 2	3364	23	31.1	16	25
	RRT Run 3	146242	19	24.35	8	15.38
	Naive CD	114	13	14.72	—	—

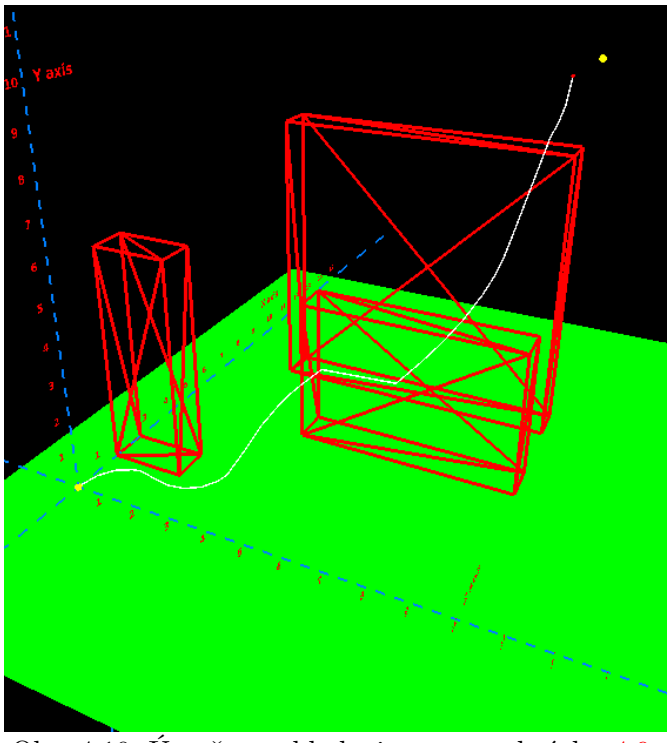
Tabuľka 4.3: Výsledky testov v prostredí 3.



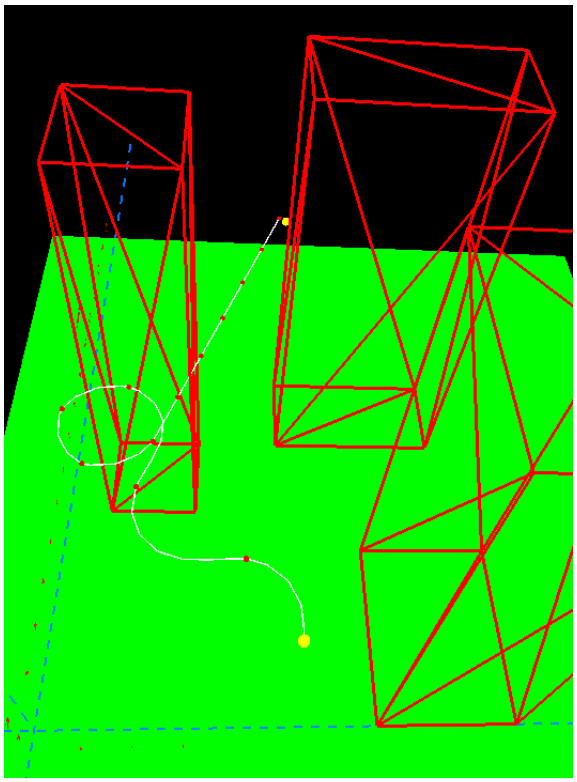
Obr. 4.8: Obrázok cesty nájdenej naivnou bunkovou dekompozíciou v rámci testovania. Prostredie 3, štart(0,0,0) – cieľ(10,10,10).



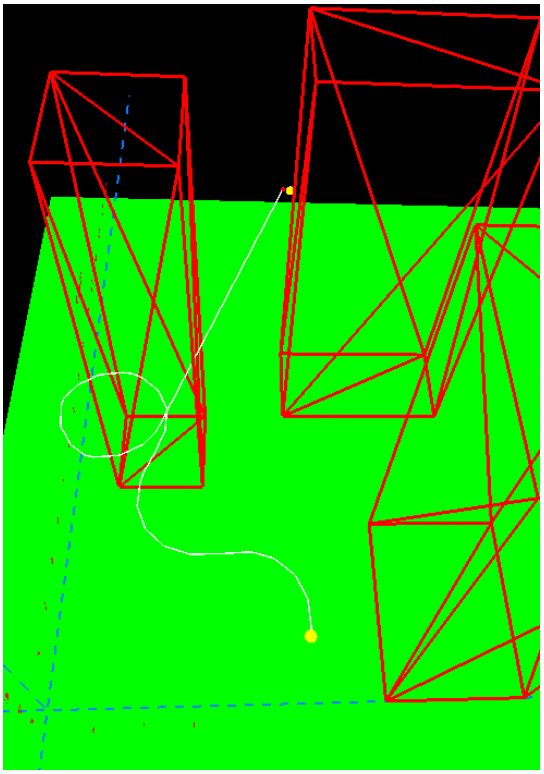
Obr. 4.9: Obrázok cesty nájdenej algoritmom RRT v rámci testovania. Prostredie 1, štart(0,0,0) - cieľ(10,10,10) beh 3.



Obr. 4.10: Úspešne vyhladenie cesty z obrázku 4.9.



Obr. 4.11: Obrázok cesty nájdenej algoritmom RRT v rámci testovania. Prostredie 3, štart(0,5,5) – cieľ(10,5,5) beh 2.



Obr. 4.12: Neúspešné vyhladenie cesty

Kapitola 5

Záver

V rámci jednotlivých kapitol a sekcií tejto práce postupne popisujeme základné prístupy k plánovaniu ciest, spôsob reprezentácie pohybových možností lietadiel a implementáciu appletu vizualizujúceho algoritmus RRT a algoritmus Naivnej bunkovej dekompozície pomocou knižnice Java3D.

Dôležitý bol výber algoritmov plánovania cesty, ktoré by boli vhodné ako na demonštráciu zachytenia pohybových možností lietadiel, tak aj na vizualizáciu. Vybral som algoritmus Naivnej bunkovej dekompozície pre jeho jednoduchý a názorný spôsob, akým sa vysporiadava so základnými problémami plánovania ciest: reprezentáciou prostredia, detekciou kolízií, zachytením pohybových možností robota a hľadaním cesty. Algoritmus Rapidly-Exploring Random Trees som zvolil na základe úspešnosti jeho aplikácie pri riešení problémov plánovania ciest pre lietadlá a pre jeho široké možnosti z hľadiska zachytenia pohybových možností robotov.

Na vytvorenej aplikácii by bolo možné zlepšiť implementáciu algoritmu vyhladzovania ciest úpravou logiky manévrovania lietadla, prípadne rozšíriť algoritmus RRT o použitie pohybových rovníc popisujúcich dynamiku lietadiel. V budúcnosti by mohlo byť zaujímavé umožniť používateľom vytvárať vlastné prostredie s ľubovoľne rozmiestnenými prekážkami, alebo implementovať iné pohybové modely, napr. helikoptéru/quadrokoptéru, pre použitie s už implementovanými algoritmami.

Z testovania naplánovaných ciest môžeme konštatovať, že applet často nedosahuje optimálne výsledky z hľadiska kvality nájdenej cesty. Nakoľko je však táto práca a výsledný applet zameraný skôr na priblíženie princípov a fungovania implementovaných algoritmov, nepovažujem kvalitu nájdených ciest za kľúčovú. Verím, že z hľadiska vizualizácie sú jednotlivé kroky algoritmov názorne zobrazené a spolu s vysvetlením na web stránkach, na ktorých je applet situovaný, poskytujú zrozumiteľnú pomôcku pre pochopenie plánovania ciest pre lietadlá.

Literatúra

[1] Baofeng Shi, P. C.; Cheng, N.: 3D flight path planning based on RRTs for RNP requirements. Technická zpráva, Department of Automation, Tsinghua University, Beijing, China, 2012.

[2] Choset, H.; Lynch, K.; Hutchinson, S.; aj.: Principles of Robot Motion (Theory, Algorithms and Implementations). Massachusetts Institute of Technology. Technická zpráva, ISN 0-262-03327-5, 2005.

[3] Cooke, J. M.; Zyda, M. J.; Pratt, D. R.; aj.: NPSNET: Flight simulation dynamic modeling using quaternions. *Presence: Teleoperators & Virtual Environments*, ročník 1, č. 4, 1992: s. 404–420.

[4] Federal Aviation Administration: Pilot's Handbook of Aeronautical Knowledge, Department of Transportation, Federal Aviation Administration, Washington, 2008. *Newcastle, Washington: Aviation Supplies and Academics, Inc., 2008.*, 2008.

[5] Geraerts, R.; Overmars, M. H.: A comparative study of probabilistic roadmap planners. In *Algorithmic Foundations of Robotics V*, Springer, 2004, s. 43–57.

[6] Geraerts, R.; Overmars, M. H.: Creating high-quality paths for motion planning. *The International Journal of Robotics Research*, ročník 26, č. 8, 2007: s. 845–863.

[7] Goerzen, C.; Kong, Z.; Mettler, B.: A survey of motion planning algorithms from the perspective of autonomous UAV guidance. *Journal of Intelligent and Robotic Systems*, ročník 57, č. 1-4, 2010: s. 65–100.

[8] Hwangbo, M.; Kuffner, J.; Kanade, T.: Efficient two-phase 3d motion planning for small fixed-wing uavs. In *Robotics and Automation, 2007 IEEE International Conference on*, IEEE, 2007, s. 1035–1041.

[9] Jia Pan, D. M.: Efficient Configuration Space Construction and Optimization for Motion Planning. *Engineering*, ročník 1, č. 1, 2015: 46, doi:10.15302/J-ENG-2015009, [Online; navštívené dňa 24.4.2016].
URL http://engineering.org.cn/EN/abstract/article_12149.shtml

[10] Khuswendi, T.; Hindersah, H.; Adiprawita, W.: UAV path planning using potential field and modified receding horizon A* 3D algorithm. In *Electrical Engineering and Informatics (ICEEI), 2011 International Conference on*, IEEE, 2011, s. 1–6.

[11] Lavalle, S. M.: *Planning algorithms*. Cambridge University Press, 2006.

[12] LaValle, S. M.; Kuffner Jr, J. J.: Rapidly-exploring random trees: Progress and prospects. 2000.

[13] Pettersson, P. O.; Doherty, P.: Probabilistic roadmap based path planning for an autonomous unmanned helicopter. *Journal of Intelligent & Fuzzy Systems*, ročník 17, č. 4, 2006: s. 395–405.

[14] Richter, C.; Bry, A.; Roy, N.: Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments. In *Proceedings of the International Symposium on Robotics Research (ISRR)*, 2013.

[15] Wang, X.; Yang, C.; Wang, J.; aj.: Hierarchical Voronoi diagram-based path planning among polygonal obstacles for 3D virtual worlds. In *VR Innovation (ISVRI), 2011 IEEE International Symposium on*, IEEE, 2011, s. 175–181.

[16] Webb, D. J.; Berg, J. v. d.: Kinodynamic RRT*: Optimal motion planning for systems with linear differential constraints. *arXiv preprint arXiv:1205.5088*, 2012.

[17] Weitz, L. A.: Derivation of a Point-Mass Aircraft Model used for Fast-Time Simulation. 2015.

[18] WWW stránky: Wikipédia. [Online; navštívené dňa 14.2.2016].
URL https://commons.wikimedia.org/wiki/File:Yaw_Axis_Corrected.svg

[19] Yan, F.; Zhuang, Y.; Xiao, J.: 3D PRM based real-time path planning for UAV in complex environment. In *Robotics and Biomimetics (ROBIO), 2012 IEEE International Conference on*, IEEE, 2012, s. 1135–1140.

[20] Yang, J.; Qu, Z.; Wang, J.; aj.: A real-time optimized path planning for a fixed wing vehicle flying in a dynamic and uncertain environment. In *Advanced Robotics, 2005. ICAR'05. Proceedings., 12th International Conference on*, IEEE, 2005, s. 96–102.

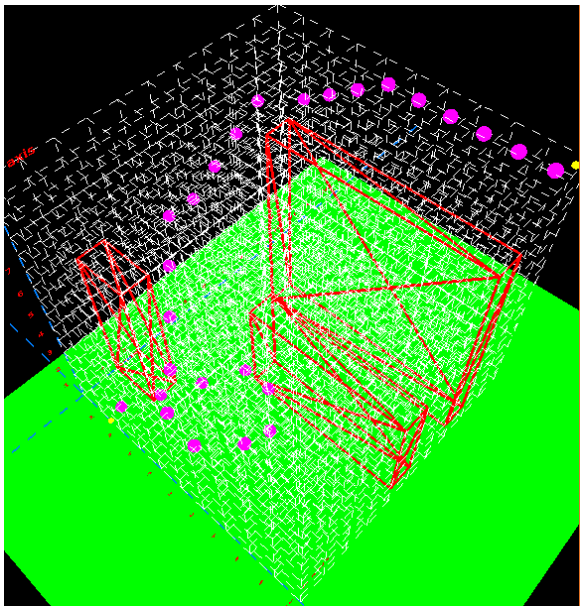
[21] Őzalp, N.; Sahingoz, O. K.: Optimal UAV path planning in a 3D threat environment by using parallel evolutionary algorithms. In *Unmanned Aircraft Systems (ICUAS), 2013 International Conference on*, IEEE, 2013, s. 308–317.

Prílohy

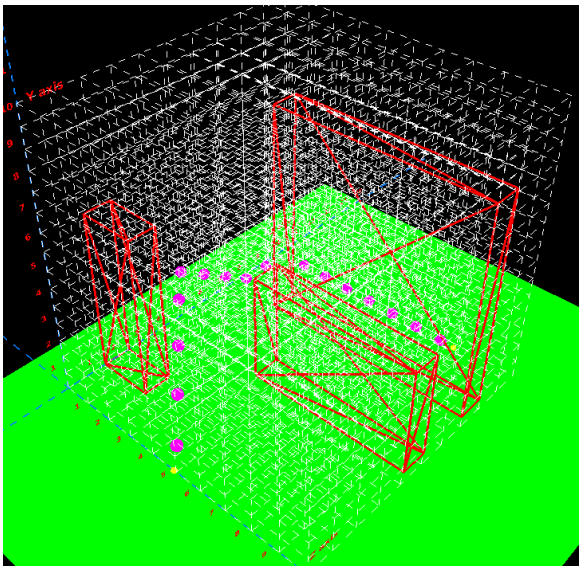
Príloha A

Obrázky z testovania

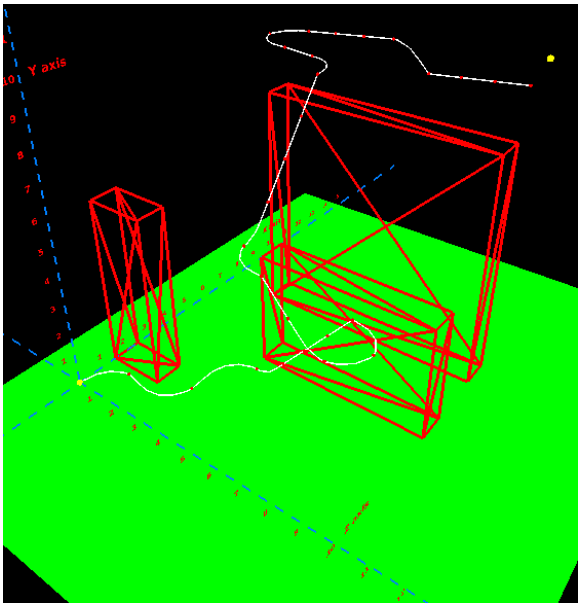
Na nasledujúcich stránkach sú priložené vybrané obrázky ciest nájdených implementovanými algoritmami v rámci ich testovania. Na obrázkoch A.1(d), A.1(j), A.1(o), A.1(q) môžeme vidieť úspešne vyhladené cesty, na A.1(g) je naopak vidieť, že implementované vyhladenie v tomto prípade zlyhalo, a cesta sa citeľne nezmenila. Na obrázku A.1(i) môžeme pozorovať výpadok v ceste, spôsobený chybou vo vykresľovaní, v nájdenej ceste, ako vidieť na A.1(j), nechýba. Cesty zachytené na týchto obrázkoch priamo odpovedajú cestám s rovnakými parametrami prostredia a štartu/cieľu v tabuľkách prezentovaných v sekcii Testovanie, okrem A.1(b), kde je vyobrazená bunková dekompozícia nevyžadujúca minimálnu výšku letu.



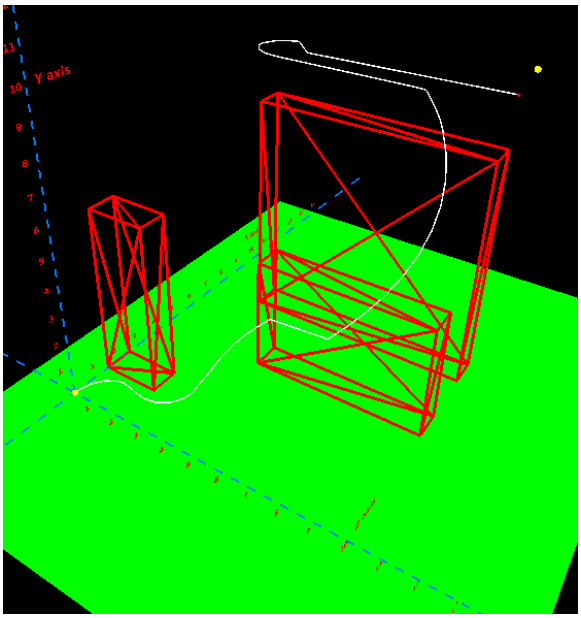
(a) CD, Preset 1, štart(0,0,1.5), cieľ(10,10,10)



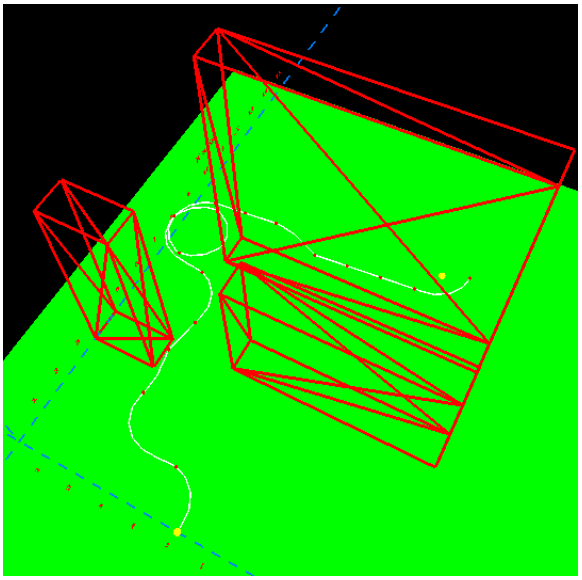
(b) CD, Preset 1, štart(0,0,5), cieľ(10,0,8)



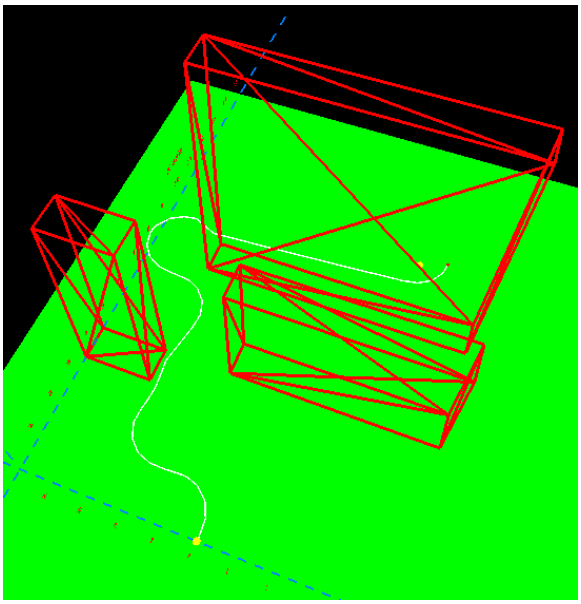
(c) RRT, Preset 1, štart(0,0,0), cieľ(10,10,10)



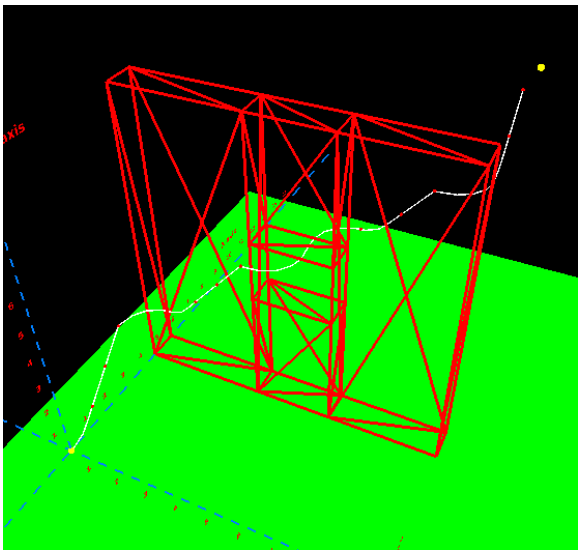
(d) Vyhladená cesta z A.1(c)



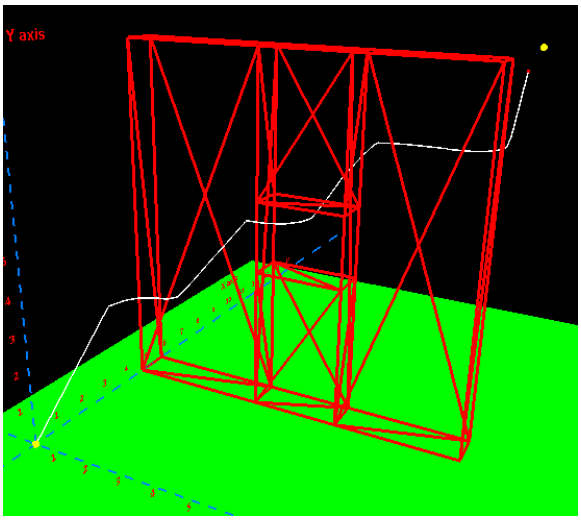
(e) RRT, Preset 1, štart(0,0,5), cieľ(10,0,5)



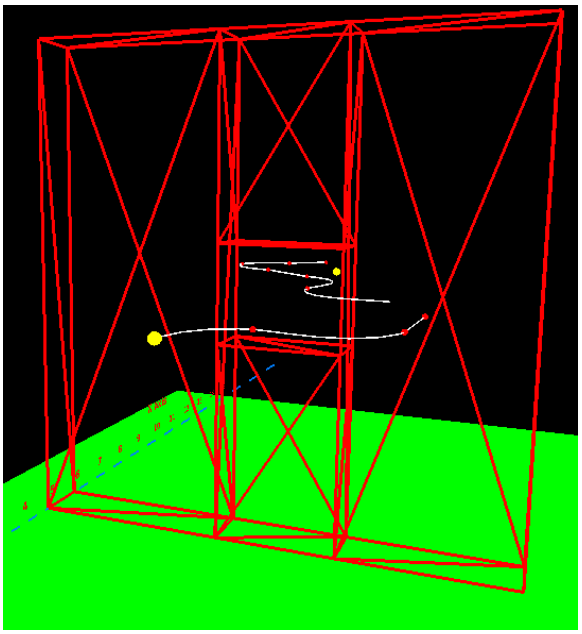
(f) Vyhladená cesta z A.1(e)



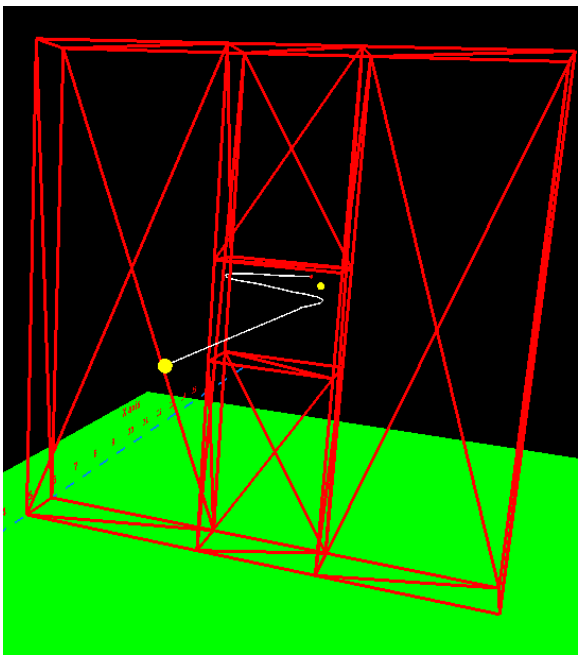
(g) RRT, Preset 2, štart(0,0,0), cieľ(10,10,10)



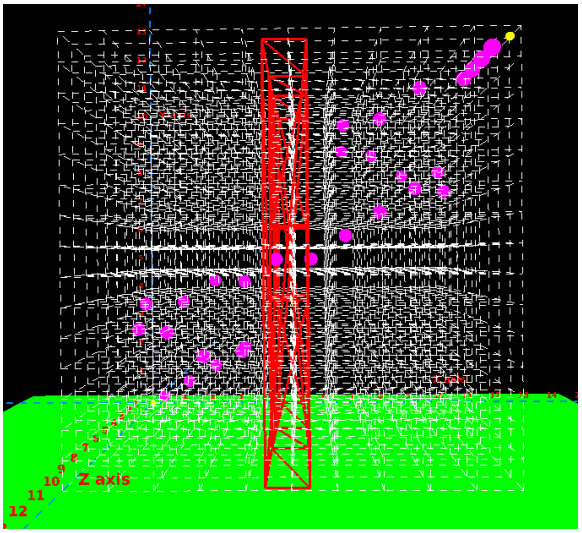
(h) Vyhladená cesta z A.1(g)



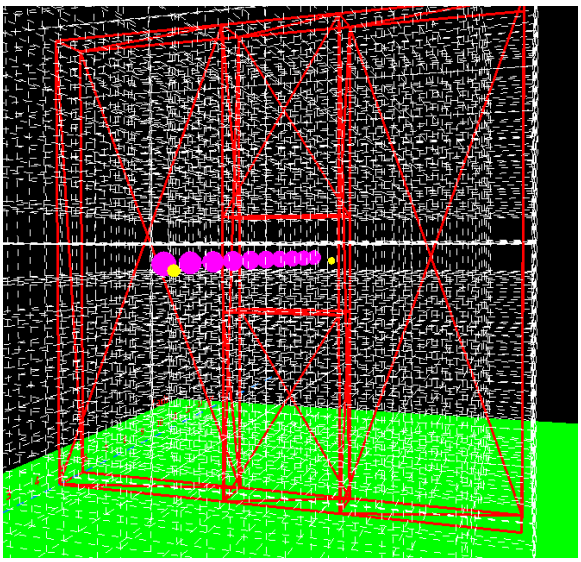
(i) RRT, Preset 2, štart(0,5,5), cieľ(10,5,5)



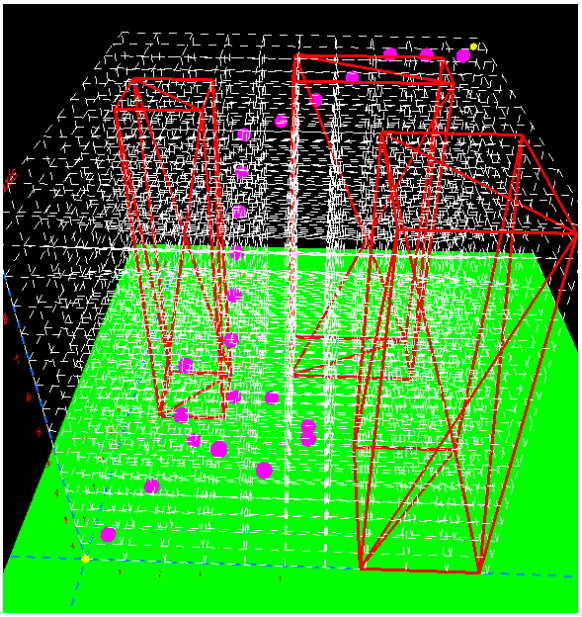
(j) Vyhladená cesta z A.1(i)



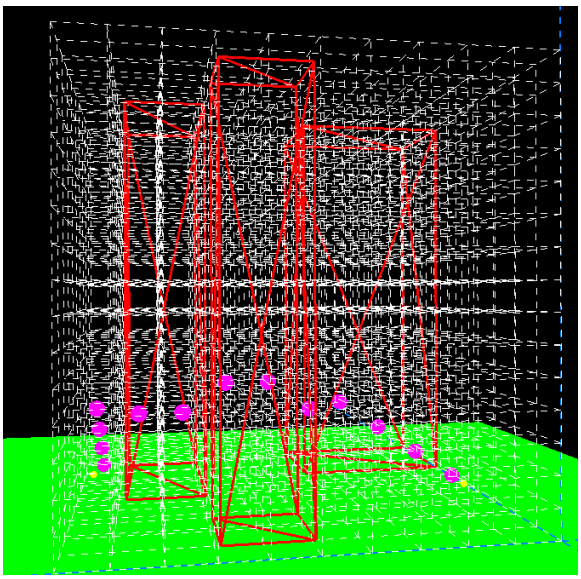
(k) CD, Preset 2, štart(0,0,0), cieľ(10,10,10)



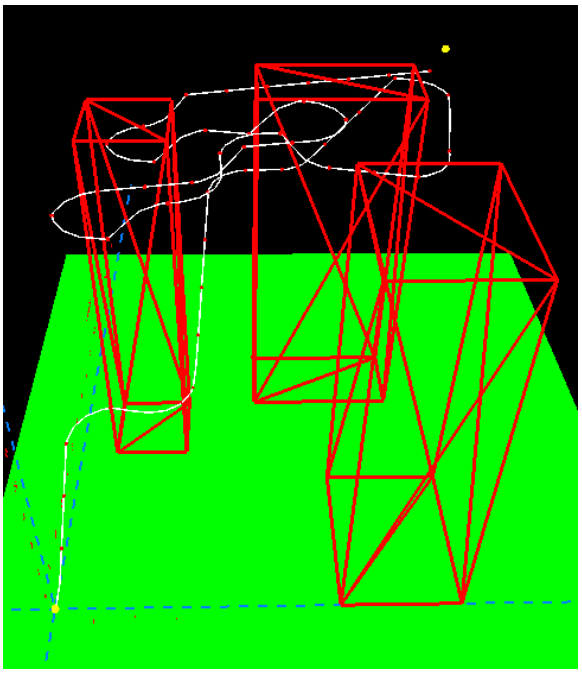
(l) CD, Preset 2, štart(0,5,5), cieľ(10,5,5)



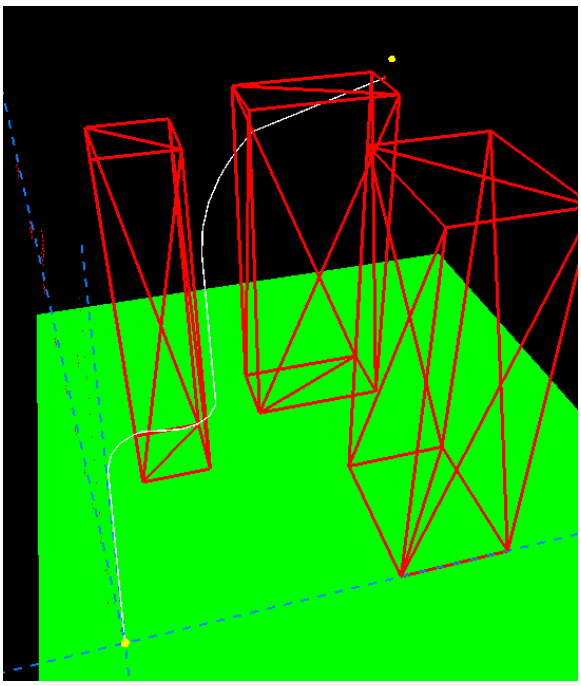
(m) CD, Preset 3, štart(0,0,0), cieľ(10,10,10)



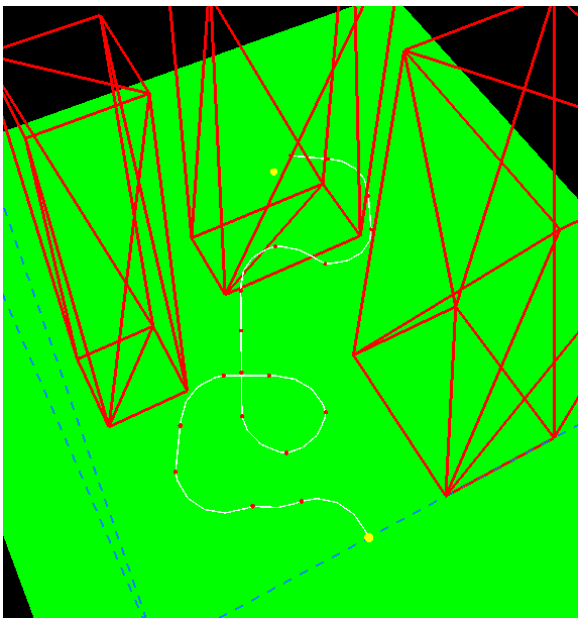
(n) CD, Preset 3, štart(0,0,5), cieľ(10,0,8)



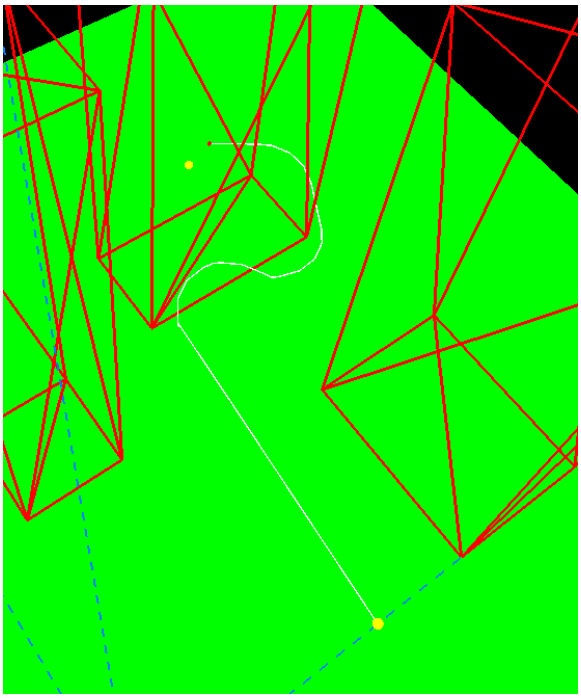
(o) RRT, Preset 3, štart(0,0,0), cieľ(10,10,10)



(p) Vyhladená cesta z A.1(o)



(q) RRT, Preset 3, štart(0,5,5), cieľ(10,0,8)



(r) Vyhladená cesta z A.1(q)