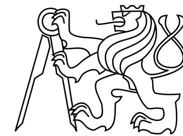
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



# ZADÁNÍ DIPLOMOVÉ PRÁCE

| | |
|---|---|
| **Název:** | Software pro grafickou demonstraci plánování pohybu kvadrokoptér |
| **Student:** | Bc. Tomáš Mahr |
| **Vedoucí:** | doc. Dipl.-Ing. Dr. techn. Stefan Ratschan |
| **Studijní program:** | Informatika |
| **Studijní obor:** | Webové a softwarové inženýrství |
| **Katedra:** | Katedra softwarového inženýrství |
| **Platnost zadání:** | Do konce zimního semestru 2017/18 |

## Pokyny pro vypracování

Cílem práce je návrh a implementace softwarového systému, který m že sloužit ke grafické demonstraci ur itého algoritmu pro plánování pohybu kvadrokoptéry, k experimentování s tímto algoritmem a k porovnávání s jinými algoritmy.

1) Vypracujte rešerši softwarových balík pro simulaci, vizualizaci a plánování pohybu robot a kvadrokoptér.
2) Ve spolupráci s vedoucím práce prove te detailní specifikaci a navrhn te uživatelské rozhraní výsledného softwarového systému.
3) Navrhn te softwarovou architekturu, která je modulární ve smyslu, že jednotlivé komponenty (simulace, vizualizace, plánování pohybu) se dají snadno vym nit.
4) P izp sobte algoritmus pro falzifikaci hybridních systém [1] problému plánování pohybu kvadrokoptéry a implementujte jej jako modul systému.
5) Systém implementujte, otestujte na vhodných p íkladech, zdokumentujte a zhodno te jeho užite nost.

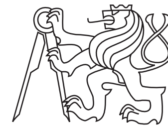
## Seznam odborné literatury


[1] Ku átko, Jan, and Stefan Ratschan. "Combined global and local search for the falsification of hybrid systems." Formal Modeling and Analysis of Timed Systems. Springer International Publishing, 2014. 146-160.


| | |
|---|---|
| Ing. Michal Valenta, Ph.D.<br>vedoucí katedry | prof. Ing. Pavel Tvrdík, CSc.<br>d kan |

V Praze dne 23. zá í 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství



Diplomová práce

# Software pro grafickou demonstraci plánování pohybu kvadrokoptér

***Bc. Tomáš Mahr***

Vedoucí práce: doc. Dipl.-Ing. Dr. techn. Stefan Ratschan

5. května 2017

# Poděkování

Rád bych na tomto místě poděkoval panu Stefanu Ratschanovi za pomoc při vedení mé diplomové práce. Dále bych chtěl poděkovat své rodině za podporu během studia.

# Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 5. května 2017 . . . . . . . . . . . . . . . . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Tomáš Mahr. Všechna práva vyhrazena.
*Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.*

## Odkaz na tuto práci

Mahr, Tomáš. *Software pro grafickou demonstraci plánování pohybu kvadrokoptér*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

# Abstract

The goal of this thesis is to design and implement modular software for the graphical demonstration of quadrocopter motion planning. The thesis also explains basic theory of quadrocopter motion planning. One of the existing algorithms is implemented as a module.

**Keywords** Quadrocopter, motion planning, graphical demonstration, modular software.

# Abstrakt

Práce se zabývá návrhem a implementací aplikace pro grafickou demonstraci plánování pohybu kvadrokoptér. Je kladen důraz na modulárnost softwaru. Dále je vyložena základní teorie plánování pohybu kvadrokoptér a je implementován jeden z existujících algoritmů pro plánování pohybu.

**Klíčová slova** Kvadrokoptéra, plánování pohybu, grafická demonstrace, modulární software.

# Obsah

**Úvod** **1**

**1 Plánování pohybu kvadrokoptér** **3**
1.1 Základní pojmy . . . 3
1.2 Ovládání pohybu kvadrokoptéry . . . 4
1.3 Rešerše softwarových balíků . . . 5

**2 Analýza** **9**
2.1 Funkční požadavky . . . 9
2.2 Nefunkční požadavky . . . 9
2.3 Scénáře užití . . . 9
2.4 Výběr technologií . . . 10

**3 Návrh** **13**
3.1 Architektura aplikace . . . 13
3.2 Modul plánovač pohybu . . . 16
3.3 Modul simulátor pohybu . . . 17
3.4 Modul zobrazovač . . . 17
3.5 Parametrizace modulů . . . 18
3.6 Rozšířitelnost pomocí pluginů . . . 18
3.7 Uživatelské rozhraní . . . 19

**4 Implementace** **23**
4.1 Načítání scény . . . 23
4.2 Detekce kolizí . . . 24
4.3 Plánovač pohybu . . . 24
4.4 Simulátor pohybu . . . 33

4.5 3D zobrazovač . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.6 Textový zobrazovač . . . . . . . . . . . . . . . . . . . . . . . . 39
4.7 Vyhodnocení úspěšnosti plánovačů . . . . . . . . . . . . . . . 40

**5 Testování a nasazení** **41**
5.1 Vyhodnocení implementovaného plánovače . . . . . . . . . . 41
5.2 Struktura projektu . . . . . . . . . . . . . . . . . . . . . . . . 42
5.3 Jednotkové testy . . . . . . . . . . . . . . . . . . . . . . . . . 42
5.4 Nasazení aplikace . . . . . . . . . . . . . . . . . . . . . . . . . 43

**Závěr** **45**

**Literatura** **47**

**A Seznam použitých zkratek** **49**

**B Obsah přiloženého CD** **51**

# Seznam obrázků

1.1 Ovládání pohybu kvadrokoptéry . . . . . . . . . . . . . . . . . . . . 5
1.2 GUI pro OMPL . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
1.3 V-REP IDE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.1 Scénáře užití . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3.1 Diagram tříd 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.2 Diagram tříd 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.3 Sekvenční diagram . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.4 Uživatelské rozhraní - zadání definice problému . . . . . . . . . . 20
3.5 Uživatelské rozhraní - zadání parametrů simulátoru . . . . . . . 21
3.6 Uživatelské rozhraní - zadání fyzikálních vlastností . . . . . . . 21
3.7 Uživatelské rozhraní - spuštění plánovače . . . . . . . . . . . . . 21
3.8 Uživatelské rozhraní - export plánu pohybu . . . . . . . . . . . 22

4.1 Algoritmus plánování pohybu . . . . . . . . . . . . . . . . . . . . 25
4.2 3D zobrazovač před načtením scény . . . . . . . . . . . . . . . . 38
4.3 3D zobrazovač po načtení scény . . . . . . . . . . . . . . . . . . 38

# Úvod

Kvadrokoptéra, nebo také dron, je bezpilotní vrtulník se čtyřmi rotory (aktuátory). Ovládá se na dálku nebo pomocí naprogramovaného mikropočítače. Pohyb kvadrokopéry se řídí nastavením otáček rotorů. Mezi typické vlastnosti kvadrokoptéry patří nízká hmotnost, malé rozměry a dobré manipulační schopnosti. V současné době se drony stávají velmi populárními. Používají se nejen pro zábavné účely (focení, nahrávání videa), své uplatnění mají také v průmyslu a používá je policie, hasiči i armáda.

V této práci se zabývám automatickým ovládáním kvadrokoptér. Konkrétně se jedná o situaci, kdy známe matematický model dronu a popis okolí, ve kterém se pohybuje. Dále máme zadaný počáteční a koncový stav dronu. Cílem je najít plán pohybu, tedy časový vývoj hodnot aktuátorů, který dovede drona z počátečního do koncového stavu, aniž by došlo ke kolizi s okolím. Tato práce se zabývá pouze softwarovou simulací a vizualizací, nikoliv ovládáním reálného dronu. Cílem práce je vytvořit modulární software pro plánování pohybu kvadrokoptér s grafickou vizualizací. Modularita softwaru spočívá v možnosti snadno měnit a přidávat plánovač, simulátor a vizualizi pohybu. Existuje velké množství algoritmů pro plánování pohybu. V této práci je jeden ([2]) z nich popsán a implementován.

První část této práce obsahuje definice základních pojmů týkajících se plánování pohybu kvadrokoptér a rešerši existujících softwarových balíků. Následující dvě kapitoly se zabývají analýzou a návrhem vlastního modulárního softwaru pro plánování pohybu kvadrokoptér. Dále je popsána implementace, včetně všech tří modulů. Plánovač pohybu je implementací algoritmu [2]. Poslední kapitola je věnována testování a nasazení softwaru.

Kapitola 1

# Plánování pohybu kvadrokoptér

## 1.1 Základní pojmy

Tato sekce vysvětluje několik základních pojmů a definuje problém, kterým se práce zabývá.

- **Aktuátor**
  Pojmem aktuátor označujeme rotor kvadrokoptéry. Hodnota aktuátoru pro nás znamená rychlost otáčení vrtule rotoru.

- **Stav kvadrokoptéry**
  Jedná se o hodnoty stavových proměnných kvadrokoptéry, mezi které patří:

  - pozice
  - rotace
  - rychlost
  - rychlost otáčení kolem tří os

  Jelikož uvažujeme trojrozměrný prostor, jsou všechny tři parametry reprezentovány trojrozměrným vektorem reálných čísel.

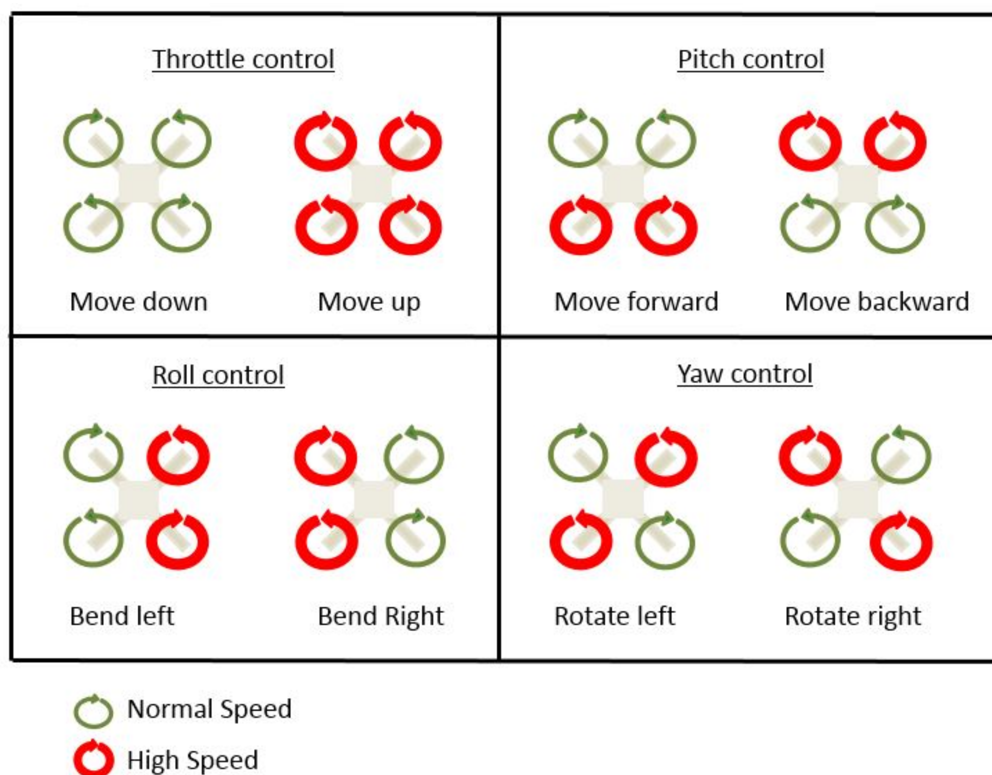
- **Fyzikální vlastnosti kvadrokoptéry**
  V této práci budeme uvažovat hmotnost a rozměry kvadrokoptéry.

- **Popis okolí**
  Jedná se o geometrický popis prostředí, ve kterém se kvadrokoptéra pohybuje (např. budova). V této práci uvažujeme trojrozměrné prostředí.

- **Plánovací úloha**
  Vstupními parametry plánovací úlohy jsou:
  - fyzikální vlastnosti kvadrokoptéry
  - popis 3D okolí
  - počáteční stav kvadrokoptéry
  - cílový stav kvadrokoptéry
- **Plán pohybu**
  Plán pohybu je časový vývoj hodnot aktuátorů. Pro zjednodušení budeme uvažovat diskrétní hodnoty na časové ose. Plán splňuje plánovací úlohu, pokud vede kvadrokoptéru z počátečního do cílového stavu, aniž by došlo ke kolizi s okolím.

## 1.2 Ovládání pohybu kvadrokoptéry

Pohyb kvadrokoptéry se řídí výhradně nastavením aktuátorů. Následující obrázek zobrazuje základní pohyby kvadrokoptéry a příslušná nastavení aktuátorů.




Obrázek 1.1: Ovládání pohybu kvadrokoptéry

Zdroj: http://www.socialledge.com/sjsu/index.php?title=File:CmpE244_S14_Quadcopter_Quad_motion1.JPG

Dron se vznese přímo vzhůru, pokud jsou hodnoty všech čtyř aktuátorů stejné a pokud je vztlaková síla vytvořená aktuátory větší než gravitační síla. Nastavením vyšších hodnot nějakých dvou aktuátorů oproti jiné dvojici lze dosáhnout různých pohybů. Tato teorie je využita při implementaci plánovače pohybu.

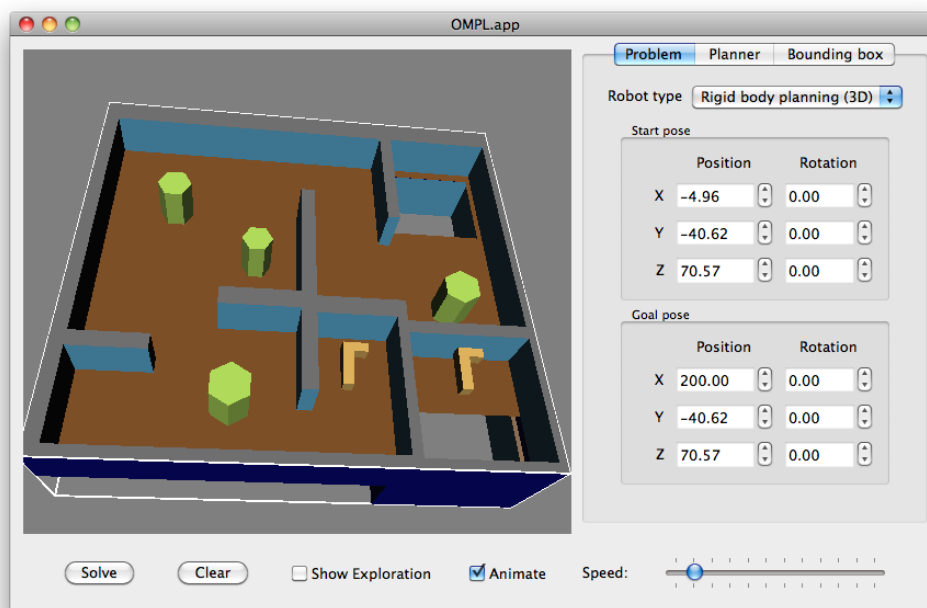
## 1.3 Rešerše softwarových balíků

Tato část obsahuje rešerši existujících softwarů pro plánování pohybu autonomních robotů.

### 1.3.1 OMPL

Open Motion Planning Library (OMPL) [9] je open source knihovna pro výpočet plánů pohybu. Obsahuje pouze sampling-based algoritmy pro plánování pohybu, nikoliv popis prostředí, detektor kolizí, vizualizaci, atd. Knihovna je navržena tak, aby ji bylo možné jednoduše integrovat s jinými systémy a frameworky (např. ROS nebo MoveIt!). Implementace je tedy dostatečně abstraktní, aby bylo možné potřebné komponenty doimplementovat. Knihovna OMPL je napsána v C++ s možností napojení na Python. K dispozici je také grafická aplikace OMPL.app napsaná v Pythonu, která slouží jako front-end k OMPL knihovně. K integraci Pythonu a C++ slouží tzv. Python bindings. Cílem aplikace je demonstrovat integraci s OMPL a poskytnout jednoduchý nástroj pro vyzkoušení plánovacích algoritmů. Plánovat pohyb lze buď pro volná tělesa, která nevyžadují vstupní kontrolní signály, nebo pro roboty vyžadující ovládání. OMPL obsahuje několik předdefinovaných typů robotů, mezi které patří i kvadrokoptéra. Je zde implemetován jednoduchý, blíže nespecifikovaný matematický model kvadrokoptéry.

OMPL i grafická aplikace se distribují ve formě zdrojových kódů. Kompilaci lze provést na OS Linux nebo Mac OS. Kompilace je poměrně náročná - vyžaduje alespoň 4 GB RAM a trvá několik hodin. I přes dodržení přesného postupu se mi nepodařilo grafickou aplikaci zprovoznit.




Obrázek 1.2: GUI pro OMPL

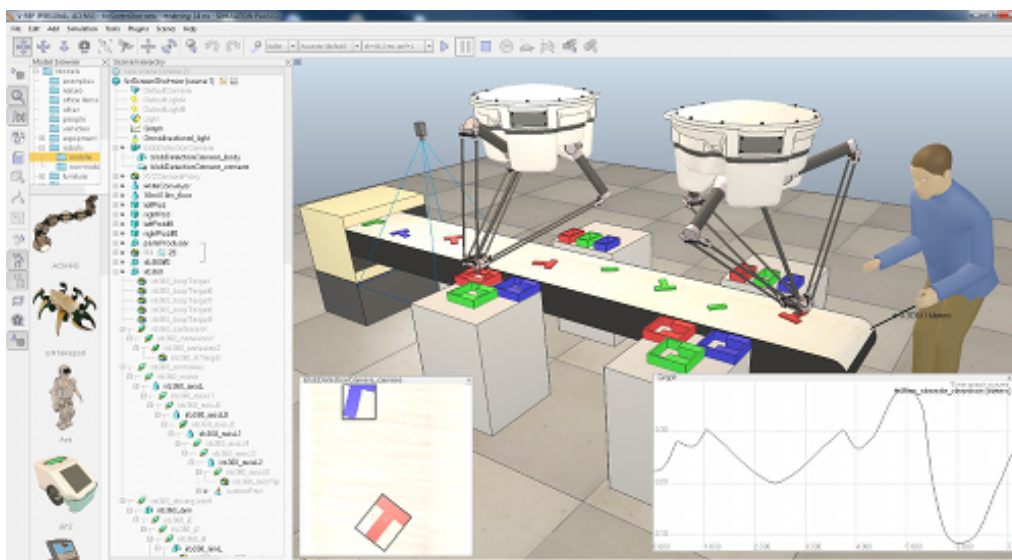
Zdroj: `http://ompl.kavrakilab.org/images/gui_define.png`

### 1.3.2 MoveIt!

MoveIt! [10] je framework pro plánování pohybu, který funguje nad ROS (Robot Operating System). ROS je open-source operační systém pro roboty, který poskytuje nízkoúrovňovou funkcionalitu, jako například ovladače hardwaru. MoveIt! poskytuje funkcionalitu pro kinematiku, plánování pohybu, detekci kolizí, a další. O plánování pohybu se typicky starají externí knihovny, např. OMPL.

### 1.3.3 V-REP

V-REP [11] je simulátor robotů, který obsahuje integrované vývojové prostředí (IDE). Plánování pohybu opět zajišťují externí knihovny.



Obrázek 1.3: V-REP IDE

Zdroj: https://www.ode-wiki.org/wiki/images/e/e5/V-rep.png

### 1.3.4 Závěr rešerše

Během hledání softwarových balíků pro plánování pohybu jsem nenašel žádný, který by se zaměřoval pouze na kvadrokoptéry. Výše zmíněné balíky jsou velmi komplexní. Mým cílem je vytvořit software, který bude jednoduchý nejen pro uživatele, ale i pro vývojáře.

Kapitola 2

# Analýza

## 2.1 Funkční požadavky

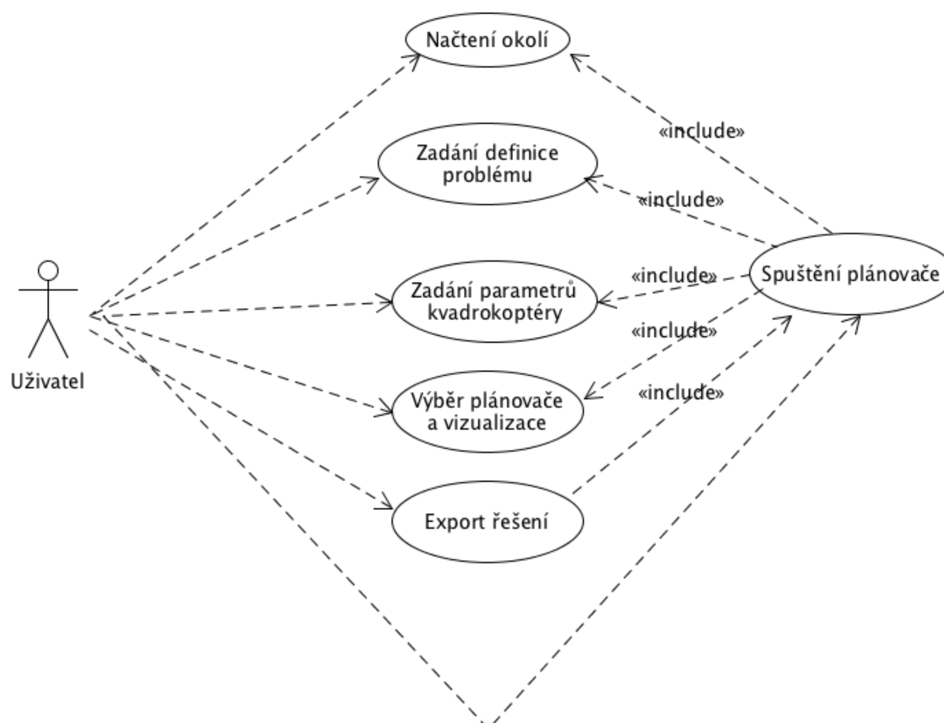
Na základě zadání byly stanoveny následující funkční požadavky, které budou implementovány.

- F1: Uživatel bude mít možnost nahrát popis okolí ze souboru.
- F2: Uživatel bude mít možnost zvolit algoritmus plánování, modul pro simulaci pohybu a modul pro vizualizaci.
- F3: Aplikace bude obsahovat defaultní simulátor pohybu a zobrazovač. Dále bude implementován plánovač pohybu dle [2].
- F4: Uživatel bude mít možnost zadat definici problému a fyzikální vlastnosti kvadrokoptéry.
- F5: Uživatel bude mít možnost měnit rychlost simulace pohybu kvadrokoptéry.
- F6: Aplikace umožní export výsledného řešení (plánu) do souboru ve formátu CSV.

## 2.2 Nefunkční požadavky

- N1: Aplikace poběží alespoň pod operačním systémem Linux.

## 2.3 Scénáře užití




Obrázek 2.1: Scénáře užití

## 2.4 Výběr technologií

Rozhodoval jsem se mezi dvěma programovacími jazyky: Java a C++. Jelikož má aplikace umožňovat grafickou demonstraci plánování pohybu, bylo potřeba zvolit nějakou grafickou knihovnu nebo framework s podporou 3D grafiky. Jednou z možností bylo vyvinout celou aplikaci na platformě Java s využitím Java 3D API [5]. Druhou možností bylo zvolit jazyk C++ spolu s nějakým frameworkem, kterých existuje celá řada. Nejvíce mě zaujal framework Qt [4]. Oba přístupy (C++ a Java) nabízejí v podstatě stejnou funkcionalitu. Z důvodu osobních preferencí a lepší podpory 3D grafiky byla zvolena druhá možnost, tedy jazyk C++ a framework Qt.

Qt je multiplatformní framework pro vytváření aplikací s grafickým uživatelským rozhraním (GUI). Podporuje nejen desktopové, ale i mobilní platformy. Tento framework nabízí vše, co potřebujeme, tedy tvorbu GUI, podporu 3D grafiky, načítání 3D formátů a tvorbu modulárních aplikací. Qt nabízí své IDE s názvem Qt Creator, které jsem pro vývoj použil. Použil

jsem Qt ve verzi 5.7.1. Abych měl jistotu, že aplikace poběží správně pod OS Linux, probíhal vývoj na tomto systému (Ubuntu 14.04.5).

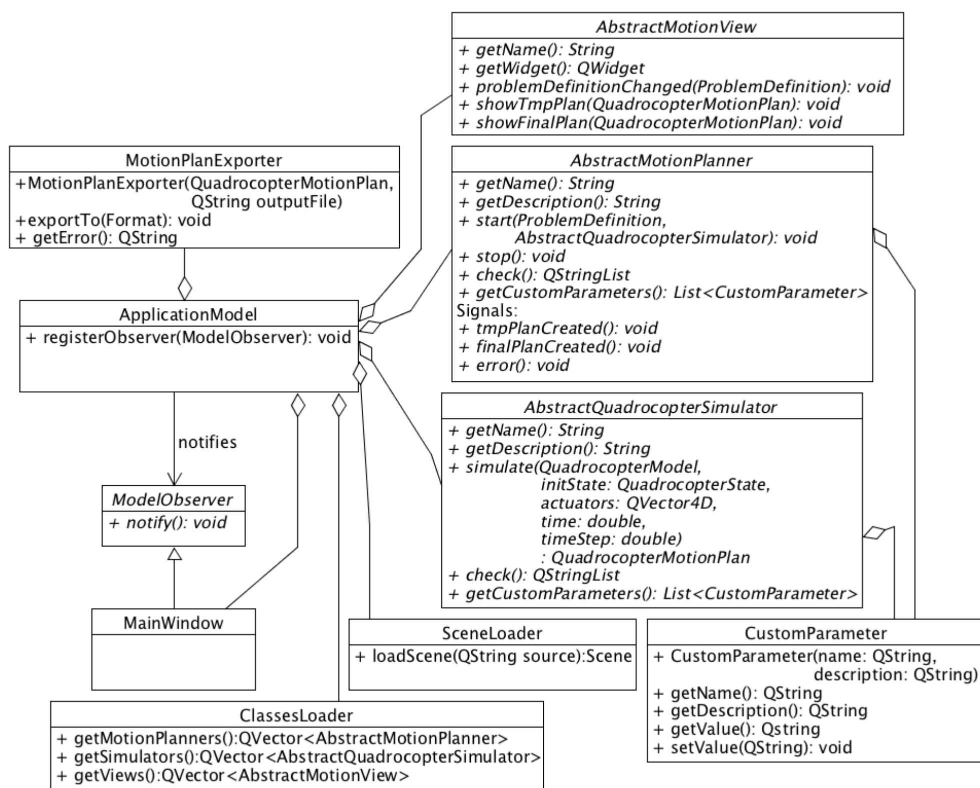
Kapitola 3

# Návrh

## 3.1 Architektura aplikace

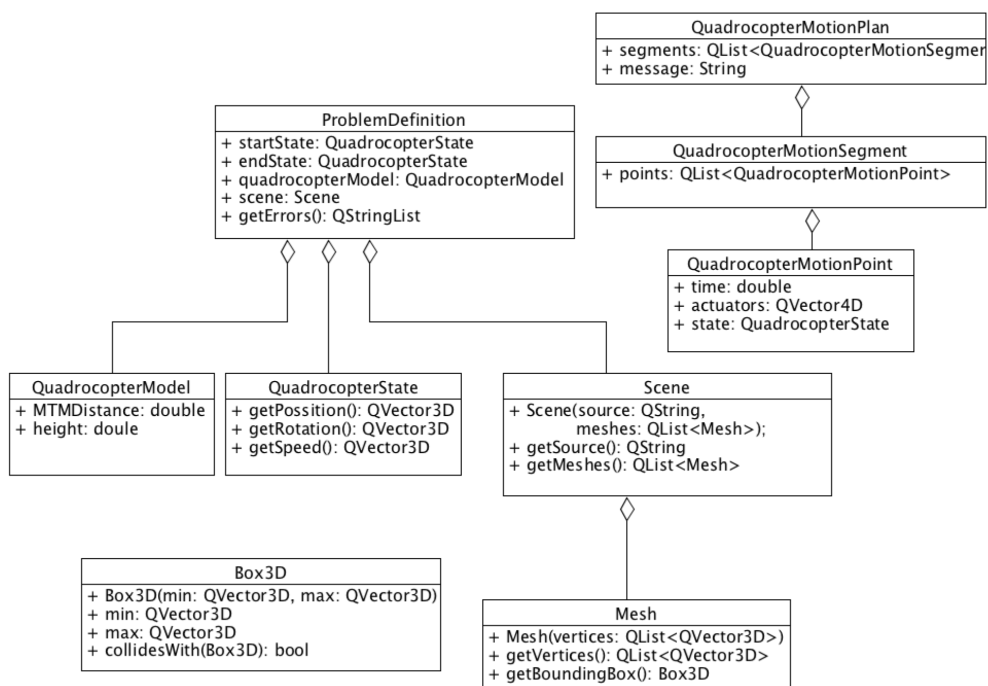
Hlavním návrhovým vzorem v této aplikaci je Model-View-Controller (MVC), který odděluje business logiku od zobrazování dat. Není zde samostatná komponenta controller, události z view jsou mapovány přímo na model. Hlavní dvě komponenty tedy jsou model a view. Konkrétně se jedná o třídy `ApplicationModel` a `MainWindow`. Model slouží jako fasáda pro view, které pomocí ní přistupuje k datům. View pouze zobrazuje data a přijímá vstupy (signály) od uživatele, které následně deleguje do modelu. Dalším použitým návrhovým vzorem je observer. Třída `MainWindow`, která dědí z abstraktní třídy `ModelObserver`, je zaregistrována jako observer v modelu. Model v případě potřeby (např. při změně stavu plánovače) notifikuje své observery.

Základní třídy lze vidět v následujícím diagramu.



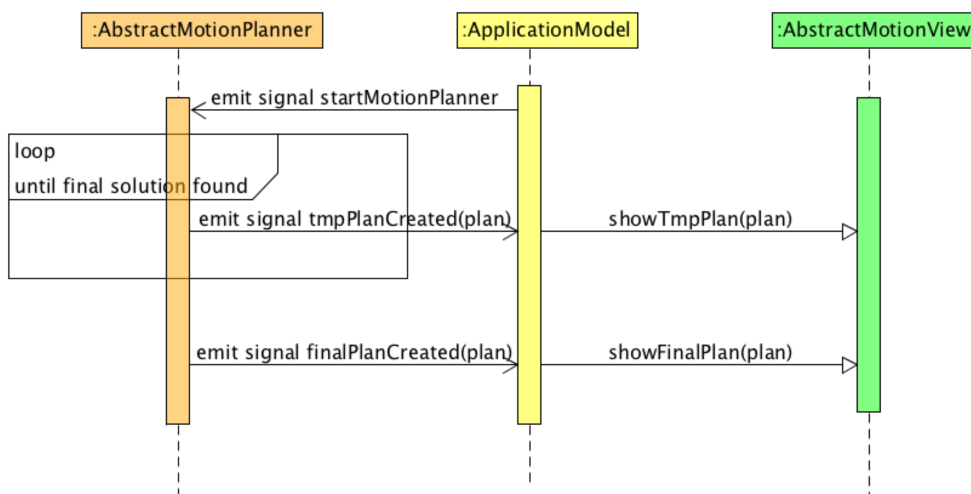

Obrázek 3.1: Diagram tříd 1

Další třídy, především ty které souvisí s definicí problému, lze vidět v následujícím diagramu.




Obrázek 3.2: Diagram tříd 2

Následuje sekvenční diagram, který zobrazuje spolupráci tříd při spuštění plánovače.




Obrázek 3.3: Sekvenční diagram

### 3.1.1 Signály a sloty

Qt používá mechanismus signálů a slotů. Signál je vyvolán objektem na základě nějaké události (např. kliknutí na tlačítko, pohyb myši). Slot je metoda, která na signál reaguje. Více informací o použití v signálů a slotů v této práci se nachází v následujících odstavcích.

## 3.2 Modul plánovač pohybu

Jedním z modulů této aplikace je plánovač pohybu, který je popsán třídou `AbstractMotionPlanner`. Cílem plánovače je vyřešit problém plánování, tedy vrátit takový plán, který splňuje uživatelem zadanou úlohu. K tomu slouží metoda `start`, která přijímá dva parametry: defininci problému a simulátor pohybu, který se má při řešení použít. `ApplicationModel` má seznam plánovačů. Vybraný plánovač je modelem přesunut na nové vlákno a spuštěn pomocí metody `start`. Třída `AbstractMotionPlanner` definuje následující signály:

- `void tmpPlanCreated(QuadrocopterMotionPlan)` Vyvolán při vytvoření dočasného plánu.
- `void finalPlanCreated(QuadrocopterMotionPlan)` Vyvolán při vytvoření závěrečného plánu.
- `void error(QString)` Vyvolán při výskytu chyby (např. když úloha nemá řešení).

Model tyto signály přijímá pomocí svých slotů a předává vytvořený plán vizualizéru.

## 3.3 Modul simulátor pohybu

Simulátor pohybu kvadrokoptéry je popsán třídou `AbstractQuadrocopterSimulator`. Její klíčová metoda je `simulate`, která přijímá následující parametry:

- fyzikální vlastnosti kvadrokoptéry (`QuadrocopterModel`)
- počáteční stav kvadrokoptéry (`QuadrocopterState`)
- požadované nastavení jednotlivých aktuátorů (úhlové rychlosti)
- požadovanou délku simulace v sekundách
- požadovanou velikost kroku v sekundách

Cílem metody, která vrací plán pohybu, je provést simulaci pohybu kvadrokoptéry při daném počátečním stavu a daném nastavení aktuátorů. Tato třída by v sobě měla mít zakódovaný matematický model kvadrokoptéry, což je typicky soustava diferenciálních rovnic. Vyřešením soustavy rovnic pro dané počáteční podmínky lze získat časový vývoj stavu kvadrokoptéry.

## 3.4 Modul zobrazovač

Abstraktní třída `AbstractMotionView` reprezentuje zobrazovač plánu pohybu. Jednou z čistě virtuálních metod je `getWidget`. Pomocí ní třída vrací ukazatel na `QWidget`, což je obecný grafický prvek v Qt. Tento prvek bude zobrazen v pravé části okna aplikace. Metody `showTmpPlan` a `showFinalPlan` jsou volány v okamžiku, kdy vznikne dočasný, resp. nový plán pohybu a je potřeba ho zobrazit. Plán pohybu musí být zobrazen pomocí instance `QWidget`, kterou vrací metoda `getWidget`. Zobrazovač je

povinen plán odstranit pomocí `delete` po dokončení zobrazování. Metoda `problemDefinitionChanged` je volána, kdykoliv dojde ke změně definice problému. Lze tak například zobrazit počáteční a cílový stav, vykreslit scénu, atd.

## 3.5 Parametrizace modulů

Moduly plánovač a simulátor je možné jednoduše parametrizovat. Chce-li modul definovat vlastní parametry, využije k tomu metodu `getCustomParameters`. Pomocí ní vrátí seznam obsahující instance třídy `CustomParameter`, které se skládají ze jména, popisu a hodnoty. Parametry jsou zobrazeny uživateli v záložkách v levé části okna. Uživatel může měnit hodnoty parametrů pomocí textových polí.

Kontrolu parametrů lze provést v metodě `check`, která se volá před spuštěním plánovače. Modul tak má možnost zkontrolovat uživatelem zadané parametry a případně vrátit seznam chyb. Plánovač je spuštěn pouze při nulovém počtu chyb.

## 3.6 Rozšířitelnost pomocí pluginů

Jedním z požadavků na aplikaci byla modularita, tedy snadná výměna a možnost přidávání plánovače, simulátoru a zobrazovače. Tento požadavek bude realizován pomocí pluginů. Pluginy jsou v Qt implementovány jako sdílené knihovny, které lze načítat za běhu aplikace. Máme zde 3 druhy modulů, které jsou popsány rozhraními: `AbstractMotionPlanner`, `AbstractQuadrocopterSimulator` a `AbstractMotionView`. Pro vytvoření nového modulu je potřeba vytvořit nový plugin (sdílenou knihovnu), který musí obsahovat třídu implementující jedno ze tří rozhraní. Třída musí také dědit ze třídy `QObject` a musí obsahovat makra `Q_OBJECT`, `Q_PLUGIN_METADATA` a `Q_INTERFACES`. Níže je ukázka třídy, která implementuje zobrazovač.

```
class TextViewPlugin : public QObject, AbstractMotionView
{
    Q_OBJECT
    Q_PLUGIN_METADATA(IID "org.qt-project.Qt.TextViewPlugin")
    Q_INTERFACES(AbstractMotionView)

public:
    explicit TextViewPlugin(QObject *parent = 0);
```

```
    virtual QString getName();
    virtual QWidget* getWidget();
    virtual void problemDefinitionChanged(ProblemDefinition &problemDef);
    virtual void showTmpPlan(QuadrocopterMotionPlan *plan);
    virtual void showFinalPlan(QuadrocopterMotionPlan *plan);

private:
    QTextEdit *mWidget;
};
```

Makro `Q_PLUGIN_METADATA` slouží ke specifikaci unikátního identifikátoru pluginu. Makro `Q_INTERFACES` slouží pro specifikaci rozhraní, které je implementováno. Sestavený plugin má koncovku `.so` na OS Unix/Linux a koncovku `.dll` na OS Windows. Plugin je třeba umístit do složky `plugins`, která se nachází v instalační složce aplikace.

O načítání pluginů se stará třída `ClassesLoader`. Načítání probíhá tak, že je prováděna iterace přes všechny soubory ve složce plugins. Každý soubor je předán třídě `QPluginLoader`, která se z něj snaží získat třídu, kterou plugin exportuje, a dále přetypovat na jedno z rozhraní. Načítání pluginů inicializuje instance třídy `ApplicationModel` během jejího vytváření.

## 3.7 Uživatelské rozhraní

Jelikož je cílem vytvořit aplikaci pro grafickou demonstraci plánování pohybu, je celé uživatelské rozhraní grafické. Při návrhu uživatelského rozhraní jsem se snažil vycházet z desatera bodů použitelnosti podle Jakoba Nielsena [1]. Jedná se o soubor pravidel, které by měly být dodrženy, aby bylo rozhraní uživatelsky přívětivé a dobře ovladatelné.

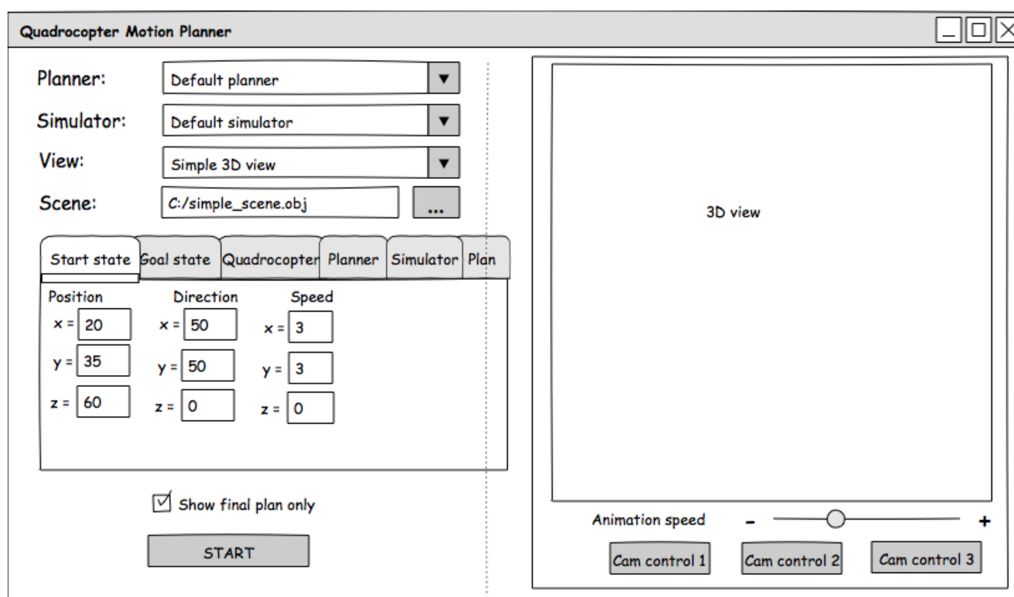
Aplikace se skládá z jediného okna, které je rozděleno na levou a pravou část. Levá část slouží k zadání definice problému, parametrů, atd. Konkrétně tato část obsahuje následujcí ovládací prvky:

- 3 rozbalovací menu pro výběr plánovače, simulátoru a zobrazovače
- tlačítko pro výběr scény
- několik záložek pro definování problému
    - počíteční a cílový stav kvadrokoptéry
    - fyzikální vlastnosti kvadrokoptéry

  - parametry simulátoru
  - parametry plánovače

- záložka zobrazující informace o stavu aplikace
- tlačítko pro export plánu do souboru
- zaškrtávací políčko umožňující zvolit, zda mají být zobrazeny i dočasné plány, nebo pouze finální plán
- tlačítko Start pro spuštění plánovače (po stisknutí tlačítka se z něj stane tlačítko Stop pro ukončení plánovače)

Celá pravá část okna je vyhrazena pro zobrazování plánu pohybu. Vzhled a chování této části je plně v kompetenci konkrétního zobrazovače. Vizualizace nemusí být jen grafická, může být např. textová. Níže se nachází navržený prototyp uživatelského rozhraní s ukázkovým 3D zobrazovačem.




Obrázek 3.4: Uživatelské rozhraní - zadání definice problému

Následující obrázky zachycují pouze výřez uživatelského rozhraní, jelikož zbytek ovládacích prvků se nemění. Pro zadání parametrů plánovače a simulátoru jsou určeny dvě záložky, jak je vidět na následujícím obrázku:

Obrázek 3.5: Uživatelské rozhraní - zadání parametrů simulátoru

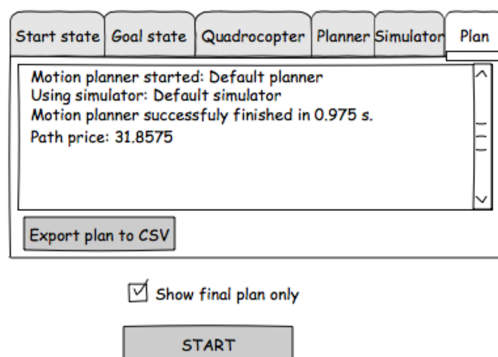
Další záložka je vyhrazena pro zadání fyzikálních vlastností kvadrokoptéry.

Obrázek 3.6: Uživatelské rozhraní - zadání fyzikálních vlastností

Kliknutím na tlačítko START dojde ke spuštění plánovače. Tlačítko se změní na STOP a aktivuje se záložka Plan, kde se vypisuje stav běhu plánovače.

Obrázek 3.7: Uživatelské rozhraní - spuštění plánovače

Když plánovač skončí, je tlačítko opět změněno na START a zároveň je aktivováno tlačítko Export plan to CSV.




Obrázek 3.8: Uživatelské rozhraní - export plánu pohybu

Kapitola 4

# Implementace

## 4.1 Načítání scény

Jednou z částí definice problému je scéna, neboli okolí, ve kterém se kvadrokoptéra pohybuje. Definice scény je řešena pomocí 3D grafických formátů. O načítání scény se stará třída `SceneLoader`, která vnitřně využívá knihovnu Assimp (Open Asset Import Library) [6]. Tato knihovna, která je součástí Qt, podporuje širokou škálu 3D formátů, například Collada, Blender 3D, Wavefront Object a mnoho dalších. Kompletní seznam lze najít na `http://www.assimp.org/main_features_formats.html`.

Třída `SceneLoader` má metodu `loadScene`, která pro daný soubor ve 3D formátu použije knihovnu Assimp pro jeho načtení a vrátí instanci třídy `Scene`. Knihovna Assimp vrací strukturu `aiScene`, která popisuje celou scénu. Scéna se skládá mimo jiné z částí označených jako mesh. Mesh popisuje geometrii dané části - tedy souřadnice vrcholů, jejich propojení, atd. Aby naše třída `Scene` nebyla závislá na knihovně Assimp, překopírujeme potřebná data ze struktury `aiScene`. Třída `Scene` by měla popisovat především geometrii. V našem případě bude scéna popisovat interiér budovy, případně venkovní oblast, kde můžeme jednotlivé objekty pro zjednodušení vnímat jako kvádry. Kopírovat budeme tedy jen meshe a vrcholy, propojením vrcholů se nebudeme zabývat. Toto rozhodnutí souvisí i s detektorem kolizí, který je popsán v následující sekci.

Byla vytvořena třída `Mesh`, která obsahuje seznam vrcholů a metodu `getBoundingBox` pro získání ohraničujícího kvádru. Třída `Scene` tedy obsahuje zdrojový soubor a seznam instancí třídy `Mesh`. Třídu `Scene` je možné v budoucnu vylepšit tak, aby detailněji popisovala geometrii scény.

I přes to, že by Qt 3D mělo podporovat všechny formáty dostupné z

`http://www.assimp.org/main_features_formats.html`, jsem měl s některými formáty problém. Scéna se nevykreslila vůbec, případně se vykreslila jen její část. Velmi záleželo i na programu, pomocí kterého byly 3D scény vytvářeny. Nakonec se osvědčil program FreeCAD [13] a formát Collada (.dae). Tento program a formát jsou tedy doporučeny pro tvorbu scén.

## 4.2 Detekce kolizí

V průběhu řešení plánovací úlohy je potřeba detekovat kolize kvadrokoptéry s okolím. Použití detektoru kolizí plně závisí na plánovači. V rámci této práce byl implementován jednoduchý detektor kolizí `DefaultCollisionChecker`, který přijímá model kvadrokoptéry, její aktuální stav a scénu. Tento detektor funguje tak, že projde všechny `Mesh` objekty, které scéna obsahuje, a pro každý z nich zjistí, zda koliduje s kvadrokoptérou. Pro zjednodušení se porovnávají ohraničující boxy rovnoběžné s osami souřadnic (axis-aligned bounding box). Třída `Mesh` má metodu `getBoundingBox`. Ohraničující box kvadrokoptéry je pro zjednodušení krychle, jejíž hrana má délu rovnou nejdelší hraně kvadrokoptéry vynásobené $\sqrt{2}$. To zaručí, že se kvadrokoptéra do boxu vejde při jakékoliv rotaci.

Způsob získání ohraničujícího boxu je snadný. Projdeme všechny body meshe a pamatujeme si nejmenší souřadnice ($x_{min}$, $y_{min}$, $z_{min}$) a největší souřadnice ($x_{max}$, $y_{max}$, $z_{max}$). Ohraničující box je potom definován vrcholy $[x_{min}, y_{min}, z_{min}]$ a $[x_{max}, y_{max}, z_{max}]$.

Dva ohraničující boxy $A$ a $B$ spolu kolidují, pokud platí všechny následující podmínky:

- $A.x_{min} <= B.x_{max} \wedge A.x_{max} >= B.x_{min}$
- $A.y_{min} <= B.y_{max} \wedge A.y_{max} >= B.y_{min}$
- $A.z_{min} <= B.z_{max} \wedge A.z_{max} >= B.z_{min}$

## 4.3 Plánovač pohybu

V rámci této aplikace byl naimplementován algoritmus pro plánování pohybu podle dokumentu [2]. Algoritmus byl naimplementován jako modul aplikace. Zmíněný dokument popisuje algoritmus pro falsifikaci hybridního systému, neboli nalezení takové trajektorie, která vede systém z počátečního

do nebezpečného stavu. V tomto algoritmu provedeme drobnou úpravu: místo nebezpečného stavu budeme uvažovat cílový stav kvadrokoptéry.

### 4.3.1 Parametry plánovače

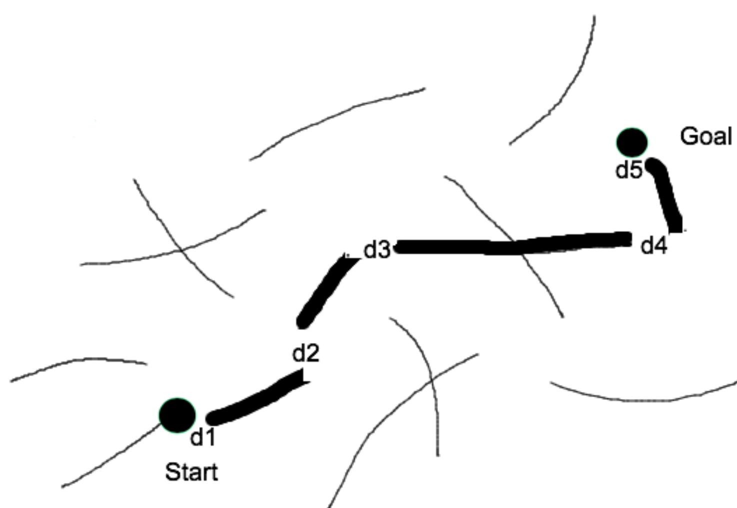
Níže je seznam parametrů, které tento plánovač přijímá.

- Délka jedné simulace (segmentu) v sekundách
- Počet segmentů na metr krychlový (pro počáteční generování)
- Maximální počet iterací hlavního algoritmu plánovače
- Maximální akceptovatelná cena trajektorie (plánovač skončí při nalezení nižší nebo stejné ceny)
- Počet iterací lokální optimalizace

### 4.3.2 Algoritmus pro plánování pohybu

Tato sekce popisuje algoritmus převzaný z [2]. Nejprve je uvedena základní myšlenka algoritmu a jsou definovány potřebné pojmy. Následně je algoritmus popsán formou pseudokódu.

Základní myšlenku algoritmu znázorňuje následující obrázek:




Obrázek 4.1: Algoritmus plánování pohybu

Nejprve je vygenerováno velké množství krátkých trajektorií. Z nich je vybrána nejkratší cesta ze startu do cíle (tučně znázorněné úseky). Mezery d1 - d5 značí vzdálenosti mezi krajními body trajektorií. Jejich součet udává cenu trajektorie. Následně se opakují dvě fáze: lokální optimalizace a globální prohledání prostoru. Na nejkratší cestu je aplikována některá z metod pro lokální optimalizaci s cílem vytvořit novou trajektorii s nižší cenou. Globální prohledání prostoru spočívá v přidání nové trajektorie.

**Definice 1** *Stav kvadrokoptéry je čtveřice trojrozměrných vektorů reálných čísel:*

- *pozice*
- *rotace*
- *rychlost*
- *úhlové rychlosti otáčení kolem os*

**Definice 2** *Segment řešení je dvojice (x, t), kde x je stav kvadrokoptéry a t je nezáporné reálné číslo. Bod x se nazývá startovní bod segmentu řešení. Číslo t značí délku segmentu řešení, která je měřena v časových jednotkách (např. sekundy).*

**Definice 3** *Stav kvadrokoptéry y nazýváme koncovým bodem segmentu (x, t), pokud je y koncovým bodem simulace o délce t z bodu x.*

Abchom mohli označit dva stavy kvadrokoptéry jako stejné, je nutné, aby se rovnaly nejen pozice, ale i rotace, rychlosti a úhlové rychlosti otáčení kolem tří os. Definujme proto vzdálenost dvou stavů.

**Definice 4** *Vzdálenost stateDist dvou stavů x a y označme jako stateDist(x, y). Platí, že dist(x, y) je rovno součtu položek:*

- *Euklidova vzdálenost vektorů pozic*
- *Euklidova vzdálenost vektorů rotací*
- *Euklidova vzdálenost vektorů rychlostí*
- *Euklidova vzdálenost vektorů úhlových rychlostí otáčení kolem os*

Množinu segmentů řešení $S$ si lze představit jako ohodnocený graf, jehož vrcholy jsou:

- startovní body segmentů řešení
- koncové body segmentů řešení
- počáteční stav kvadrokoptéry
- cílový stav kvadrokoptéry

Váhu hrany mezi vrcholy $v_1$ a $v_2$ označme $w(v_1, v_2)$. Platí, že $w(v_1, v_2) = 0$, pokud $v_2$ je koncový bod segmentu řešení se startovním bodem $v_1$. V opačném případě platí $w(v_1, v_2) = stateDist(v_1, v_2)$.

**Definice 5** *Kandidátní trajektorie je posloupnost segmentů ($x_1$, t), ... ($x_n$, t).*

Chceme najít trajektorii z počátečního stavu do cílové stavu. Kandidátní trajektorie musí splňovat následující podmínky:

- startovní bod prvního segmentu je roven počátečnímu stavu kvadrokoptéry
- koncový bod segmentu je roven startovnímu bodu následujícího segmentu
- koncový bod posledního segmentu je roven cílovému stavu kvadrokoptéry

**Definice 6** *Cena kandidátní trajektorie ($x_1$, t), ..., ($x_n$, t) je součet následujících položek*

- *vzdálenost stateDist mezi bodem x1 a počátečním stavem kvadrokoptéry,*
- *součet vzdáleností stateDist mezi koncovým bodem segmentu ($x_i$, t) a startovním bodem $x_{i+1}$*
- *vzdálenost stateDist mezi koncovým bodem segmentu ($x_n$, t) a cílovým stavem kvadrokoptéry*

Obecný algoritmus pro plánování pohybu autonomního robota lze popsat následujícím pseudokódem:

**Algorithm 1** Plánování pohybu

1: $S \leftarrow \{\}$
2: Vygeneruj velké množství sekvencí a přidej je do množiny $S$.
3: $cand \leftarrow$ trajektorie s nejnižší cenou z $S$
4: **while** optimalizace trajektorie $cand$ pomocí minimalizace cenové funkce neprodukuje dostatečně dobré řešení **do**
5: přidej nový segment do množiny $S$
6: $cand \leftarrow$ trajektorie s nejnižší cenou z $S$
7: **end while**

### 4.3.3 Implementace algoritmu pro plánování pohybu

V této sekci jsou popsány implementační detaily výše zmíněného algoritmu, který byl upraven pro potřeby plánování pohybu kvadrokoptér. Algoritmus je nejprve představen ve formě pseudokódu. Následně jsou jeho zásadní kroky slovně vysvětleny.

**Algorithm 2** Plánování pohybu kvadrokoptér

1: $maxIter \leftarrow$ uživatelem zadaná konstanta reprezentující maximální počet iterací
2: $maxPrice \leftarrow$ uživatelem zadaná konstanta reprezentující maximální akceptovatelnou cenu řešení
3: $optimalActuators \leftarrow$ najdi hodnoty aktuátorů, které způsobí vzlet kvadrokoptéry přímo vzhůru optimální rychlostí
4: $S \leftarrow \{\}$
5: Vygeneruj velké množství sekvencí a přidej je do množiny $S$.
6: Odstraň segmenty kolidující s okolím.
7: $cand \leftarrow$ trajektorie s nejnižší cenou z $S$
8: $price \leftarrow$ cena trajektorie $cand$
9: $iter \leftarrow 0$
10: **while** $iter < maxIter$ and $price > maxPrice$ **do**
11: optimalizuj $cand$ pomocí minimalizace cenové funkce
12: $price \leftarrow$ cena $cand$
13: **if** $price <= maxPrice$ **then**
14: return $cand$
15: **end if**
16: přidej nový segment do množiny $S$
17: $cand \leftarrow$ trajektorie s nejnižší cenou z $S$
18: $iter \leftarrow iter + 1$
19: return $cand$
20: **end while**

#### 4.3.3.1 Zjištění optimálního nastavení aktuátorů

Aby byl let kvadrokoptéry realistický, je potřeba zjistit optimální nastavení otáček rotorů. To závisí nejen na matematickém modelu (který je zakódován v simulátoru), ale také na fyzikálních parametrech kvadrokoptéry (hmotnost, rozměry, ...). Cílem je tedy zjistit, jaké nastavení aktuátorů vznese kvadrokoptéru přímo vzhůru rozumnou rychlostí. Jako rozumná rychlost bylo zvoleno nejvýše 2 m/s. Aby se kvadrokoptéra vznesla směrem přímo vzhůru, je potřeba nastavit všechny 4 aktuátory na stejnou hodnotu. Hledání optimální hodnoty probíhá pomocí binárního půlení, kde počáteční spodní hranice je 0 a horní hranice $10^6$. To, jestli se kvadrokoptéra vznese a jakou rychlostí, je testováno pomocí modulu simulátor. Jakmile je nalezeno nastavení aktuátorů, které kvadrokoptéru vznese nejvýše požadovanou rychlostí, je algoritmus binárního půlení ukončen. Plánovač si hodnotu a zapamatuje a následně ji využívá během dalších nastavení aktuátorů.

#### 4.3.3.2 Počáteční generování segmentů

Plánovač definuje parametry, pomocí kterých lze nastavit počet segmentů a délku segmentu v sekundách. Generování segmentů probíhá systematicky tak, aby byla pokryta co největší část scény. Plánovač nejprve zjistí ohraničující box scény, který následně rozdělí na kvádry stejné velikosti. Z každého kvádru je provedeno 8 simulací, jejichž počáteční pozice jsou generovány náhodně v rámci kvádru. Každá z osmi simulací by měla ideálně směřovat jiným směrem. Simulace by ideálně měly odpovídat pohybům popsaným v sekci 1.2. Je tedy potřeba zajistit, aby kvadrokoptéra nejen stoupala přímo vzhůru a klesala přímo dolu, ale aby se pohybovala i jinými směry. K tomu je potřeba změnit hodnoty některých aktuátorů. Jako základ je použita hodnota *optimalActuators*. Dále byla experimentálně zvolena multiplikativní konstanta 0.75.

Nechť $OPT$ je rovno *optimalActuators*. Nechť $low = OPT \cdot 0.75$. Simulace používají následující nastavení aktuátorů (aktuátory jsou v pořadí: přední, pravý, zadní, levý):

- $(OPT, OPT, OPT, OPT)$
- $(0, 0, 0, 0)$
- $(low, low, OPT, OPT)$
- $(OPT, low, low, OPT)$
- $(OPT, OPT, low, low)$
- $(low, OPT, OPT, low)$
- $(low, OPT, low, OPT)$
- $(OPT, low, OPT, low)$

#### 4.3.3.3 Odstranění segmentů kolidujících s okolím

Segment je reprezentován třídou `QuadrocopterMotionSegment` a skládá se z jednotlivých bodů (`QuadrocopterMotionPoint`). Jedním z atributů každého bodu je stav kvadrokoptéry (`QuadrocopterState`). Plánovač iteruje přes všechny stavy segmentu a pokud některý z nich koliduje s okolím, je takový segment odstraněn. Pro kontrolu kolizí je použit `DefaultCollisionChecker`.

#### 4.3.3.4 Nalezení kandidátní trajektorie s nejnižší cenou

Problém nalezení trajektorie s nejnižší cenou je převeden na problém hledání nejkratší cesty v ohodnoceném grafu (graf je popsán v sekci 4.3.2). Pro implementaci byl zvolen Dijkstrův algoritmus.

#### 4.3.3.5 Optimalizace trajektorie

Vstupem funkce pro optimalizaci trajektorie je seznam segmentů, které tvoří kandidátní trajektorii. Cílem je provést minimalizaci cenové funkce, tedy upravit segmenty tak, aby cena výsledné trajektorie byla co nejmenší, pokud možno nulová. Problém, který zde řešíme, se obecně nazývá matematická optimalizace a zabývá se minimalizací (maximalizací) cílové funkce. V našem případě se jedná o minimalizaci cenové funkce kandidátní trajektorie. Nejprve definujme cenu mezi dvěma stavy kvadrokoptéry:

**Algorithm 3** statesPrice: Cena mezi dvěma stavy kvadrokoptéry

**Input:** stavy kvadrokoptéry ($a$, $b$)
**Output:** cena (reálné číslo)

1: $price \leftarrow 0$
2: $price \leftarrow price +$ euclideanDistance($a.position$, $b.position$)
3: $price \leftarrow price +$ euclideanDistance($a.rotation$, $b.rotation$)
4: $price \leftarrow price +$ euclideanDistance($a.speed$, $b.speed$)
5: **return** $price$

Nyní můžeme uvést pseudokód pro výpočet ceny kandidátní trajektorie:

**Algorithm 4** trajectoryPrice: Cena kandidátní trajektorie

**Input:** počáteční a cílový stav kvadrokoptéry ($startState$, $goalState$), pole segmentů $segments$ velikosti $segmentsCnt$
**Output:** cena (reálné číslo)

1: $price \leftarrow 0$
2: $price \leftarrow price +$ statesPrice($startState$, $segments[0].firstState$)
3: **for** $i$ from 0 to $segmentsCnt - 1$ **do**
4: $price \leftarrow price +$ statesPrice($segments[i].lastState$, $segments[i+1].firstState$)
5: **end for**
6: $price \leftarrow price +$ statesPrice($segments[segmentsCnt-1].lastState$, $goalState$)
7: **return** $price$

Výše uvedený algoritmus reprezentuje pouze funkci pro výpočet ceny trajektorie, nikoliv cenovou funkci. Cenová funkce by měla mít následující signaturu:

$$\mathbb{R}^n \rightarrow \mathbb{R}.$$

Uvažujme, že chceme optimalizovat kandidátní trajektorii tvořenou $k$ segmenty. Známe nastavení aktuátorů pro každý segment. Zafixujme počáteční bod (stav) v každém segmentu a snažme se změnit nastavení aktuátorů tak, aby nová kandidátní trajektorie měla menší cenu. Naše cenová funkce tedy bude mít na vstupu $k$ čtveřic hodnot aktuátorů (jedna čtveřice pro každý segment) a na výstupu cenu nové kandidátní trajektorie.

Nyní tedy můžeme definovat algoritmus cenové funkce, kterou se následně budeme snažit minimalizovat:

**Algorithm 5** Cenová funkce pro danou kandidátní trajektorii tvořenou $n$ segmenty

**Input:** $4 \cdot n$ hodnot aktuátorů (reálná čísla)
**Output:** cena nové kandidátní trajektorie (reálné číslo)

1: **for all** segment $s$ **do**
2:     *state* ← počáteční stav segmentu $s$
3:     *sNew* ← proveď simulaci ze stavu *state* s novými hodnotami aktuátorů
4:     nahraď segment $s$ segmentem *sNew*
5: **end for**
6: *price* ← cena nově vytvořené kandidátní trajektorie
7: **return** *price*

Postupy pro minimalizaci funkcí více proměnných můžeme rozdělit do dvou základních skupin:

- **Bez využití derivací**
  Na funkci se díváme jako na black box. Známe pouze výstupy pro námi zadané vstupy. Do této skupiny patří například Nelder-Meadova metoda simplexů.

- **S využitím derivací**
  Do této skupiny patří metody, které využívají derivace pro odhad klesání/stoupání cílové funkce. Typicky se využívá gradient, což je vektor obsahující parciální derivace cílové funkce.

Rozhodl jsem se pro použití některé metody z první skupiny, tedy bez využití derivací. Druhý způsob nelze využít, protože neznáme rovnice popisující pohyb kvadrokoptéry (moduly plánovač a simulátor jsou na sobě nezávislé) Pro tyto účely existuje velké množství knihoven, z nichž mě nejvíce zaujala knihovna GSL (GNU Scientific Library) [12]. GSL poskytuje velké množství funkcí z různých oblastí matematiky, což se může hodit pro další potenciální vylepšení plánovače.

Hledání minima funkce probíhá iterativně. Nejprve je potřeba zvolit tzv. minimizer, neboli algoritmus minimalizace. Zvolil jsem minimizer s názvem `gsl_multimin_fminimizer_nmsimplex2`, který používá výše zmíněnou Nelder-Meadovu metodu simplexů. Dále je potřeba nastavit počáteční hodnoty parametrů funkce (v našem případě tedy počáteční hodnoty aktuátorů). Zvolil jsem hodnoty hodnoty aktuátorů, které odpovídají jednotlivým segmentům. Dalším krokem vyplnění struktury `gsl_multimin_function`, která má následujcí položky:

- `n`: počet dimenzí
- `f`: ukazatel na cenovou funkci
- `params`: volitelné parametry cenové funkce (`void *`)

Poté už následuje iterativní volání funkce `gsl_multimin_fminimizer_iterate`, která vrací status. Ten lze použít jako ukončovací kritérium. Jako další ukončovací kritérium jsem se rozhodl použít maximální počet iterací, který definuje plánovač jako jeden ze svých parametrů.

#### 4.3.3.6 Přidání nového segmentu

V případě, že optimalizací kandidátní trajektorie nevznikne dostatečně dobré řešení, přichází na řadu rozšíření množiny segmentů o nový segment. Máme spoustu možností, jak přidat nový segment. Jednou z možností je vygenerovat nový stav a z něj provést náhodnou simulaci. Já jsem zvolil jiné řešení, a sice prodloužení stávajícího segmentu. Simulace se tedy provádí z koncového bodu náhodně vybraného segmentu.

## 4.4 Simulátor pohybu

Byl naimplementován simulátor pohybu kvadrokoptéry, který je založen na matematickém modelu [3]. Model pracuje s následujícími dvanácti stavovými proměnnými:

| **Stavová proměnná** | **Popis** |
|---|---|
| $p_n$ | pozice kvadrokoptéry na ose X |
| $p_e$ | pozice kvadrokoptéry na ose Z |
| $h$ | výška kvadrokoptéry (osa Y) |
| $u$ | rychlost kvadrokoptéry (osa X) |
| $v$ | rychlost kvadrokoptéry (osa Z) |
| $w$ | rychlost kvadrokoptéry (osa Y) |
| $\phi$ | roll (úhel rotace kolem osy X) |
| $\theta$ | pitch (úhel rotace kolem osy Z) |
| $\psi$ | yaw (úhel rotace kolem osy Y) |
| $p$ | úhlová rychlost otáčení kolem osy X |
| $q$ | úhlová rychlost otáčení kolem osy Z |
| $r$ | úhlová rychlost otáčení kolem osy Y |

Pohyb kvadrokoptéry je pak popsán soustavou diferenciálních rovnic (ODE systém), uvedenou v dokumentu [3] v kapitole 3, rovnice 16 - 19. Pro každou stavovou proměnnou je zde jedna rovnice. Vyřešením ODE systému pro nějaké počáteční hodnoty proměnných získáme časový vývoj těchto proměnných.

Simulátor definuje následující parametry, které matematický model vyžaduje:

- poloměr středu kvadrokoptéry
- hmotnost středu kvadrokoptéry
- hmostnost rotoru

- $k_1$:
  koeficient pro převod hodnoty aktuátoru ($\delta$) na sílu ($F$) produkovanou rotorem:
  $F = k_1 \cdot \delta$

- $k_2$:
  koeficient pro převod hodnoty aktuátoru ($\delta$) na točivý moment ($\tau$) produkovaný rotorem:
  $\tau = k_2 \cdot \delta$

Pro vyřešení ODE systému byla zvolena C++ knihovna ODEINT [7], která je součástí Boost C++ knihoven. ODEINT s pracuje s polem stavových proměnných (náš matematický model se skládá z 12 proměnných, pole má tedy délku 12), které musí být na začátku nainicializovány. Dále je potřeba reprezentovat ODE systém a předat ho řešiči. To lze provést implementováním funkce se signaturou

```
void ode_system(const state_type &x,
                state_type &dxdt,
                 const double t).
```

Datový typ `state_type` je v našem případě pole délky 12. Proměnná `t` reprezentuje čas. Tělo funkce musí obsahovat přepis daného ODE systému. Jako příklad uveďme rovnici

$$s'(t) = f(s, t).$$

Její přepis bude vypadat následovně:

```
dxdt[i] = f(x[i], t),
```

kde `i` je index proměnné `s`.

Pro získání výsledků je potřeba naimplementovat funkci se signaturou

```
void observer(const state_type &x , const double t).
```

Funkce je volána během integrování, kdykoliv jsou k dispozici nové hodnoty. Řešic lze spustit pomocí funkce `integrate`, které je potřeba předat funkci reprezentující ODE systém, počáteční hodnoty proměnných, počáteční a koncový čas a observer.

## 4.5 3D zobrazovač

V rámci této práce byla implemetována třída `Simple3DView`, jako jeden ze zobrazovačů. Kód tohoto zobrazovače je naimplementován v rámci aplikace, nejedná se o plugin. Třída dědí z `AbstractMotionView` a slouží pro 3D vizualizaci plánů pohybu. Samotné zobrazování má na starosti pomocná třída `Simple3DViewWidget`, na kterou jsou delegovány veškeré požadavky na zobrazení plánu.

Pro práci s 3D grafikou byly použity Qt 3D moduly (3dcore, 3drender, 3dinput a 3dextras). Základním elementem v Qt 3D je `Qt3DCore::QEntity`. Při volání `problemDefinitionChanged` je pomocí třídy `Qt3DRender::QSceneLoader` načtena 3D scéna. `Qt3DRender::QSceneLoader` vnitřně používá knohovnu Assimp, takže stačí předat cestu k nějakému souboru ve 3D formátu (.dae, .obj, atd.). Dále je vykreslen počáteční a cílový stav kvadrokoptéry. Kvadrokoptéra je pro jednoduchost reprezentována jako kvádr. Uživatel tak může vidět, jak vypadá definice problému před spuštěním plánovače. Kdykoliv je definice problému změněna (např. posunutí startovní pozice), je tato změna okamžitě reflektována ve 3D zobrazovači.

Ovládání 3D scéno je řešeno pomocí `QOrbitCameraController`. Instance této třídy je součástí 3D scény a umožňuje uživateli ovládat kameru pomocí kláves.

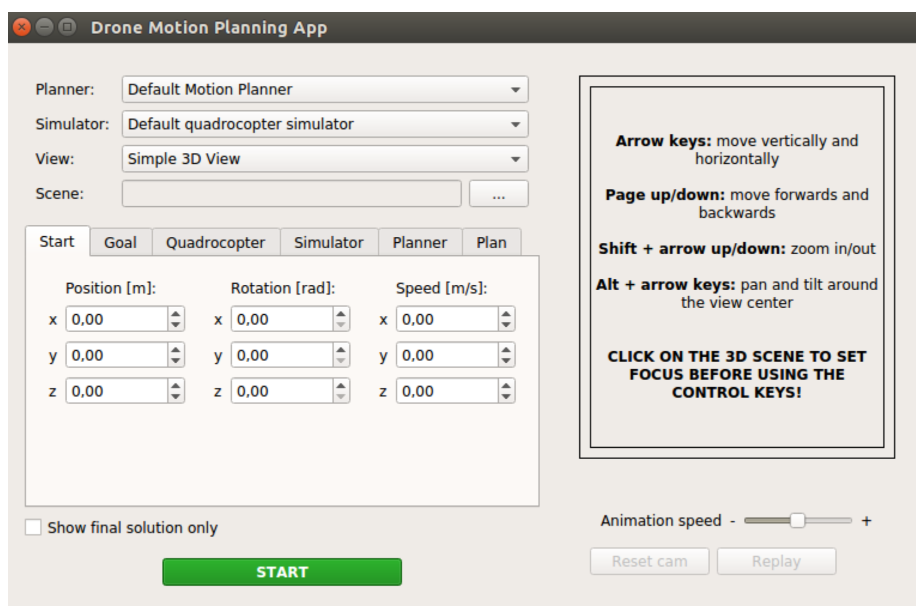
Zobrazování plánů pohybu je řešeno pomocí fronty. Na volání metod `showTmpPlan` a `showFinalPlan` tato třída reaguje tak, že si daný plán uloží do fronty. Třída `Simple3DViewWidget` obsahuje časovač (`Qtimer`), který vybírá plány z fronty a postupně je zobrazuje. Jelikož se přidávání i odebírání plánů odehrává na hlavním vlákně, nejsou potřeba žádné prostředky synchronizace. Následuje ukázka přidání plánu do fronty:

```
void Simple3DViewWidget::addPlan(QuadrocopterMotionPlan *plan)
{
    mPlansQueue.enqueue(plan);
    if (!mTimer.isActive())
    {
        pickNextPlanToPlay();
        mTimer.start();
    }
}
```

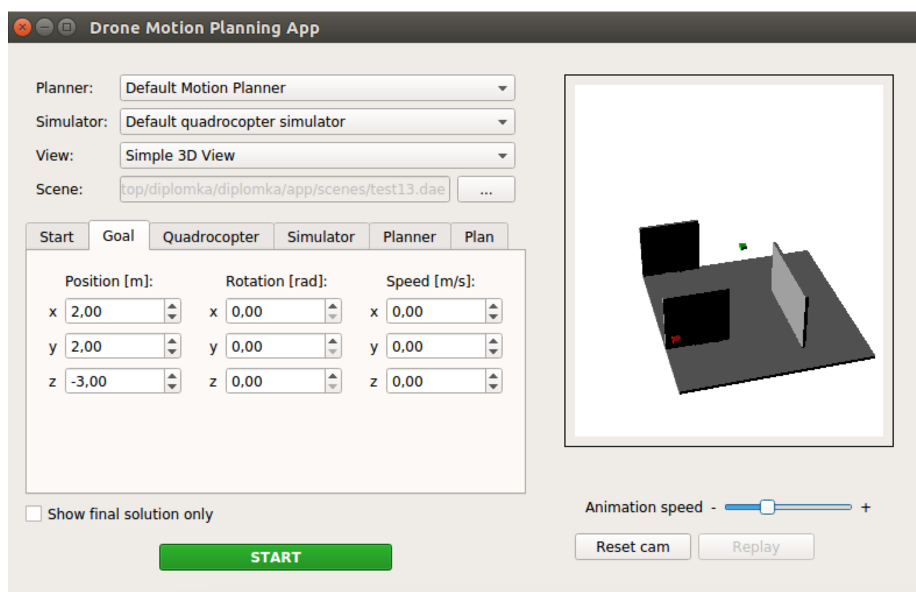
Segmenty aktuálního plánu jsou ve druhé frontě. Časovač iteruje přes stavy aktuálního segmentu a postupně je zobrazuje. V případě, že už není k dispozici žádný segment ani plán, je časovač zastaven. Spuštěn může být opět voláním metody addPlan. Chování časovače popisuje následující metoda:

```
void Simple3DViewWidget::timeout()
{
    if (nextStateInSegment())
    {
        QuadrocopterState nextState = nextStateToShow();
        moveQuadrocopter(nextState);
    }
    else
    {
        // No more states in current segment
        if (mSegmentsQueue.size() >= 1)
        {
            mCurrentSegment = mSegmentsQueue.dequeue();
        }
        else if (mPlansQueue.size() >= 1)
        {
            pickNextPlanToPlay();
        }
        else
        {
            // No more plans
            mTimer.stop();
        }
        mPointIdx = 0;
    }
}
```

Grafické prostředí zobrazovače je rozděleno na dvě části. Horní plocha slouží k samotnému zobrazování 3D scény. Těsně pod ní je zobrazována textová zpráva související s právě zobrazovaným plánem. Ve spodní části se nacházejí ovládací prvky. Uživatel má možnost měnit rychlost animace pomocí posuvníku. Ovládání rychlosti je realizováno nastavením intervalu časovače. Dále jsou k dispozici tlačítka pro resetování pohledu kamery a pro přehrání závěrečného plánu. Kamera se ovládá pomocí kláves, jejichž popis je zobrazen před načtením scény.




Obrázek 4.2: 3D zobrazovač před načtením scény




Obrázek 4.3: 3D zobrazovač po načtení scény

## 4.6 Textový zobrazovač

Pro demonstraci modularity aplikace byl naimplementován další zobrazovač. Tento zobrazovač byl vytvořen jako zvláštní plugin a slouží k textové prezentaci plánů pohybu. Je zamýšlen pouze jako ukázka tvorby pluginu, proto je jeho funkcionalita velmi strohá. Skládá se z jediného widgetu - `QTextEdit`, tedy textového pole. Prezentace plánu pohybu probíhá jednoduše vypsáním jednotlivých stavů kvadrokoptéry.

Plugin byl vytvořen pomocí Qt Creator IDE jako sdílená knihovna (shared library). Celý projekt se nachází na přiloženém CD. Při vytváření pluginu je potřeba nastavit cestu k hlavní aplikaci. To lze provést v projektovém souboru (.pro) například takto

```
INCLUDEPATH += ../DronMotionPlanningApp/src
```

Následuje ukázka metody, která se stará o prezentaci plánu pohybu.

```
void TextViewPlugin::showPlan(QuadrocopterMotionPlan *plan)
{
    QString str;
    QTextStream stream(&str);
    stream << "\nPlan:\n";
    for (int i=0; i<plan->mSegments.size(); i++)
    {
        QuadrocopterMotionSegment segment = plan->mSegments.at(i);
        for (int j = 0; j < segment.mPoints.size(); j++)
        {
            QuadrocopterMotionPoint point = segment.mPoints.at(j);
            stream << "t = " << point.mTime;
            stream << " Actuators = [";
            stream << point.mActuators.w();
            stream << ", " << point.mActuators.x();
            stream << ", " << point.mActuators.y();
            stream << ", " << point.mActuators.z();
            stream << "]\n";
        }
    }
    mWidget->insertPlainText(str);
}
```

## 4.7 Vyhodnocení úspěšnosti plánovačů

Dle zadání práce by aplikace měla umožňovat porovnávat jednotlivé plánovací algoritmy. Pro tyto účely jsem vytvořil třídu `MotionPlanEvaluator`, jejíž úkolem je nezávisle vyhodnotit kvalitu výsledného plánu pohybu. Implementoval jsem dvě metriky:

- **Délka trajektorie**
  Počítá se jako součet Euklidovských vzdáleností mezi sousedními pozicemi. Do délky trajektorie se počítá i vzdálenost počáteční pozice od prvního bodu a vzdálenost posledního bodu k cílové pozici.

- **Cena trajektorie**
  Tato hodnota je spočtena podle definice ceny kandidátní trajektorie [6].

Kapitola 5

# Testování a nasazení

## 5.1 Vyhodnocení implementovaného plánovače

Implementovaný plánovač je size funkční, nicméně vytváří pouze přibližné plány pohybu. Výstupem plánovače je plán, který se skládá z několika nespojených segmentů. Plán tedy není úplný a proto není příliš využitelný v praxi. Změny parametrů příliš neovlivnily kvalitu plánu. Hlavní příčiny vidím v následujících položkách

- **Počáteční generování segmentů**
  Je třeba více pokrýt scénu. To souvisí s porozuměním dynamiky a kinematiky kvadrokoptéry, díky čemuž lze lépe pracovat s nastavováním aktuátorů. Plánovač na začátku provádí zjištění optimálního nastavení aktuátorů pro přímý vzlet. To není dostatečné, je potřeba spočítat další klíčové hodnoty aktuátorů, aby bylo možné s kvadrokoptérou lépe a kontrolovaně manipulovat.

- **Optimalizace**
  Zde je prostor pro vyzkoušení jiných optimalizačních metod.

- **Přidání nového segmentu**
  Je třeba prozkoumat i jiné způsoby a experimentovat s nimi. Tento bod souvisí i s

## 5.2 Struktura projektu

- `DronMotionPlanningApp.pro` ........ konfigurační soubor Qt projektu
- `src` ........
  - `gui` ........ adresář obsahující zdrojové kódy pro GUI
  - `model` ........ adresář obsahující zdrojové kódy pro business logiku
  - `test` ........ adresář obsahující zdrojové kódy unit testů
- `doc` ........
  - `doxygenconfig` ........ konfigurační soubor pro Doxygen
  - `html` ........ výstup programu Doxygen obsahující HTML dokumentaci
- `distribution` ........ adresář sloužící jako šablona pro distribuci aplikace
  - `lib` ........ sdílené knihovny potřebné pro běh aplikace
  - `plugins` ........ sdílené Qt knihovny + moduly aplikace
  - `resources` ........
    - `scenes` ........ ukázkové scény
  - `README.txt` ........ Instrukce ke spuštění aplikace
  - `run.sh` ........ Skript ke spuštění aplikace.

## 5.3 Jednotkové testy

Pro ověření správné funkčnosti jednotlivých tříd byly napsány unit testy (jednotkové testy). Zdrojové kódy testů jsou součástí hlavní aplikace a nachází se v adresáři `src/test`. Framework Qt zajišťuje podporu pro unit testy, čehož jsem využil. Z časových důvodů testy nepokrývají všechny třídy, ale ty nejdůležitější třídy pokryty jsou. Pro každou testovanou třídu byla vytvořena nová třída, jejíž název má příponu `Test`. Pro každou testovanou metodu byla vytvořena nová metoda s předponou `test`. Vstupním bodem pro spuštění testů je funkce `main` v souboru `test_main.cpp`, která postupně spouští všechny testové třídy pomocí `QTest::qExec`.

Projektový soubor `DronMotionPlanningApp.pro` obsahuje sekci pro unit testy:

```
test {
    message(Test build)
    QT += testlib
    TARGET = UnitTests
    SOURCES -= src/main.cpp
    HEADERS += <vycet_hlavickovych_souboru>
    SOURCES += <vycet_implementacnich_souboru>
```

```
} else {
    message(Normal build)
}
```

Projekt lze sestavit dvěma způsoby:

- **Build hlavní aplikace**: V tomto případě není potřeba upravovat projektový soubor. Výsledkem bude spustitelná hlavní aplikace. Testy v ní nebudou obsaženy.
- **Build testů**: V tomto případě je potřeba přidat řádek `CONFIG += test`. Výsledkem bude spustitelná aplikace obsahující pouze testy.

## 5.4 Nasazení aplikace

Výsledkem kompilace je spustitelná aplikace, která ovšem ke svému běhu potřebuje některé sdílené knihovny z framkeworku Qt. Aplikace je proto distribuovaná spolu s těmito knihovnami. Pro zjištění závislostí aplikace byl použit nástroj `ldd`, který je běžně dostupný na Linuxových systémech.

Aplikaci je potřeba spouštět prostřednictvím skriptu `run.sh`, který nejprve nastaví proměnné prostředí `LD_LIBRARY_PATH`, `QT_PLUGIN_PATH` a `QT_QPA_PLATFORM_PLUGIN_PATH` na správné adresáře obsahující potřebné knihovny.

Aplikace byla úspěště otestována na čisté instalaci 64 bitové verze Ubuntu 16.04.2.

# Závěr

Cílem diplomové práce bylo vytvořit software pro grafickou demonstraci plánování pohybu kvadrokoptér. Nejprve jsem vysvětlil záklaní terminologii v oblasti plánování pohybu, nadefinoval potřebné pojmy a provedl rešerši existujících řešení.

Dále jsem provedl návrh software pro plánování pohybu kvadrokoptér, který je dostatečně modulární a udržovatelný. Jeho architektura tedy splňuje zadání - moduly plánovač, simulátor a vizualizaci lze snadno měnit a přidávat. Díky distribuci jednotlivých modulů ve formě sdílených knihoven (pluginů) je možné moduly přidávat a odebírat za běhu aplikace. Jako programovací jazyk jsem zvolil C++, a to mimo jiné z důvodu lepší podpory 3D grafiky. Pro usnadnění implementace GUI, pluginů a 3D grafiky jsem vybral framework Qt, jehož výhodou je mimo jiné podpora více platforem (Windows, Linux, Mac OS). Pomocí zmíněných technologií jsem vytvořil aplikaci s grafickým rozhraním, která může sloužit pro běh a porovnávání plánovačů pohybu kvadrokoptér. Mnou vytvořený modul pro simulaci pohybu je plně funkční a použitelný, stejně jako modul pro vizualizaci. Uživatel může vytvářet vlastní 3D scény a používat je v aplikaci.

Provedl jsem základní implementaci plánovače pohybu dle [2]. Moje implementace je stále ve fázi vývoje a není plně použitelná v praxi, protože plánovač vytváří pouze přibližné plány. Výsledné plány pohybu se skládají z několika nespojených segmentů. Je zde tedy velký prostor pro vylepšení v oblasti optimalizace trajektorie. Dále je vhodné více analyzovat dynamiku a kinematiku kvadrokoptéry a negenerovat segmenty příliš náhodně.

# Literatura


[1] Nielsen, J. *10 Usability Heuristics for User Interface Design.* 30.4.2017. Dostupné z http://www.designprinciplesftw.com/collections/10-usability-heuristics-for-user-interface-design

[2] Kuřátko, J., Ratschan, S. *Combined global and local search for the falsification of hybrid systems. Formal Modeling and Analysis of Timed Systems.* Springer International Publishing, 2014. 146-160.

[3] Beard, Randal *Quadrotor Dynamics and Control Rev 0.1.* 2008. All Faculty Publications. Paper 1325. http://scholarsarchive.byu.edu/facpub/1325 146-160.

[4] *Qt framework.* https://www.qt.io

[5] *Java 3D API.* http://www.oracle.com/technetwork/java/javase/overview/index-jsp-138252.html

[6] *Open Asset Import Library.* http://www.assimp.org

[7] *ODEINT - řešič diferenciálních rovnic pro C++.* http://headmyshoulder.github.io/odeint-v2/

[8] Gamma, E. *Design patterns : elements of reusable object-oriented software.* 1995. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA

[9] *The Open Motion Planning Library.* http://ompl.kavrakilab.org

[10] *MoveIt!.* http://moveit.ros.org

[11] *V-REP*. http://www.coppeliarobotics.com

[12] *GSL - GNU Scientific Library*. https://www.gnu.org/software/gsl/

[13] *FreeCAD*. https://www.freecadweb.org

Příloha A

# Seznam použitých zkratek

**GUI** Graphical user interface

**ODE** Ordinary Differential Equations

**GSL** GNU Scientific Library

**OMPL** Open Motion Planning Library

**IDE** Integrated Development Environment

**MVC** Model-View-Controller

Příloha B

# Obsah přiloženého CD

```
readme.txt ............ stručný popis obsahu CD
exe ................... adresář se spustitelnou formou implementace
src
  impl ................ zdrojové kódy implementace
  thesis .............. zdrojová forma práce ve formátu LaTeX
text .................. text práce
  thesis.pdf .......... text práce ve formátu PDF
```