Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

# Simulácia správania UAV v roji

Dokumentácia k inžinierskemu dielu

**Vedúci práce:** Ing. Viktor Šulák
**Členovia tímu:** Bc. Ondrej Antl, Bc. Adam Bacho , Bc. Marián Čarnoký,
Bc. Michal Grznár, Bc. Jakub Mrocek, Bc. Tomáš Rychvalský,
Bc. Katarína Szakszová
**Akademický rok:** 2015/2016

# Obsah

**1 Úvod** **1**

**2 Globálne ciele na letný semester** **2**
2.1 Zhodnotenie výsledkov zimného semestra . . . . . . . . . . . . . 2
2.2 Ciele na letný semester . . . . . . . . . . . . . . . . . . . . . . . . . 2

**3 Celkový pohľad na systém** **3**
3.1 Architektúra . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
3.2 Diagram tried . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

**4 Moduly systému** **15**
4.1 Modul inicializácie systému . . . . . . . . . . . . . . . . . . . . . . 15
4.2 Modul komunikácie . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
4.3 Modul simulácie . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
4.4 Modul vykresľovania simulácie . . . . . . . . . . . . . . . . . . . . 24

**Prílohy**

**A Spracované analýzy** **A-1**
A.1 Prehľad databázových technológií . . . . . . . . . . . . . . . . . . A-1
A.2 Analýza programových riešení . . . . . . . . . . . . . . . . . . . . A-3
A.3 Analýza komunikácie UAV zariadení . . . . . . . . . . . . . . . . . A-5
A.4 Algoritmy pre plánovanie trajektórie UAV zariadení . . . . . . . . A-9

**B Používateľská dokumentácia** **B-1**
B.1 Spustenie programu . . . . . . . . . . . . . . . . . . . . . . . . . . B-1
B.2 Ukončenie programu . . . . . . . . . . . . . . . . . . . . . . . . . . B-4

**C Technická dokumentácia** **C-1**
C.1 Prepojenie logiky simulátora a vykresľovania simulácie . . . . . . C-1
C.2 Opis skriptu správania sa hlavného drona . . . . . . . . . . . . . . C-5

# 1 Úvod

UAV (*Unmanned aerial vehicles*) zariadenia, často nesprávne nazývané „drony", sú v súčasnosti jeden z najrýchlejšie sa rozvíjajúcich smerov v robotike. Ide o zariadenia so schopnosťou letu bez ľudskej posádky. Väčšina ľudí vníma drony ako lietajúce hračky na ovládanie, čo je pri komerčnom pohľade na vec pomerne správny úsudok. V skutočnosti je však ich účel a potenciál omnoho väčší. V súčasnosti sa začínajú dostávať do povedomia ako zariadenia uľahčujúce prácu v teréne, v nedostupných či nebezpečných oblastiach, slúžia ako oko na oblohe. Aktuálne však žiadne voľne pohybujúce sa zariadenie nie je samostatné. Vždy ho riadi človek prostredníctvom vysielačky.

Novým prínosom v tejto oblasti by malo byť programovanie takýchto zariadení pre vykonávanie určitých špecifických úloh a samostatné riadenie. Exituje viacero projektov, venujúcim sa podobnej problematike. Ďalšou limitáciou použiteľnosti takýchto zariadení je kontrola nad jedným zariadením v jednom momente. Pri použití viacerých je možné skvalitniť či dosiahnuť zrýchlenie vykonávania špecifickej úlohy. V takomto prípade je nutné aby UAV dokázali spolu navzájom komunikovať ale zároveň aby ostali nezávislými a samostatnými jednotkami.

Keďže UAV nie sú lacnými hračkami na ovládanie ako si mnohí predstavujú ale robustnými, komplexnými zariadeniami vysokej hodnoty, vývoj typu pokus-omyl nie je prípustný. Vznikla teda potreba vytvorenia simulátora pre správanie sa takýchto zariadení s možnosťou širokej konfigurácie prostredia a taktiež samotných zariadení. Simulátor správania UAV je prostriedok k zrýchleniu výskumu kooperácie medzi takýmito zariadeniami nie len na našej fakulte.

# 2 Globálne ciele na letný semester

Práca v letnom semestri priamo naväzuje na výstupy prác počas semestra zimného. Preto najprv uvádzame výsledky, ktoré sme dosiahli počas práce v zimnom semestri. Neskôr v tejto kapitole uvádzame ciele na letný semester.

## 2.1 Zhodnotenie výsledkov zimného semestra

V zimnom semestri sme narazili na viacero problémov, ktore viedli k opakovanym zmenám pri návrhu systému. Museli sme opakovane a stále podrobnejšie analyzovať problémy, ktoré tvorba simulátora obnáša, aby sme vedeli čo najlepšie navrhnúť spôsob jeho fungovania. Keďže bolo veľké úsilie venované práve analýze a návrhu systému, k implementácii sme sa dostali iba okrajovo, napriek tomu, že sme mali vyššie plány za začiatku semestra. Výhodou je zas dostatočne detailne navrhnutý systém. Počas návrhu sme mysleli na viaceré problémy, ktoré sme sa snažili eliminovať a veríme, že sa nám to aj podarilo.

## 2.2 Ciele na letný semester

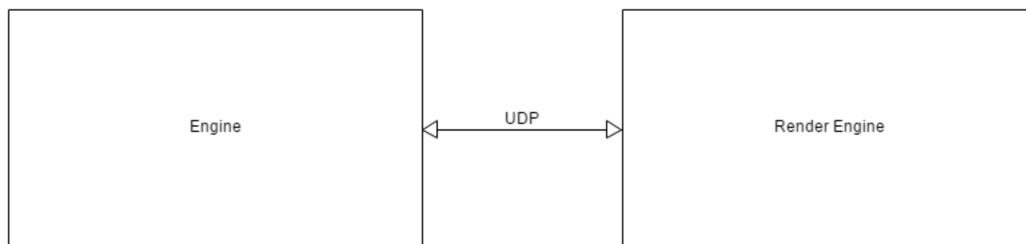
- Zabezpečiť rýchlu a spoľahlivú komunikáciu medzi simulačnýcm enginom a vykresľovacím enginom.
- Zabezpečiť pravidelné obnovovanie polohy jednotlivých UAV zariadení na základe ich pohybu a vplyvu prostredia.
- Umožniť UAV zariadenie pravidelné a časté vykonávanie vlastnej logiky definovanej používateľským skriptom.
- Umožniť posielať UAV zariadeniam správy, pričom používateľ bude vedieť udávať spôsob šírenia signálu v priestore.
- Ošetriť zrážky UAV zariadení s prostredím aj navzájom
- Vytvoriť ukážkový skrpit logiky UAV zariadenia

# 3 Celkový pohľad na systém

V tejto kapitole prinášame podrobný prehľad navrhnutého a implementovaného systému, jeho architektúry, tried a modulov. Ďalej sa v kapitole venujeme opisu vlastností systému.

## 3.1 Architektúra

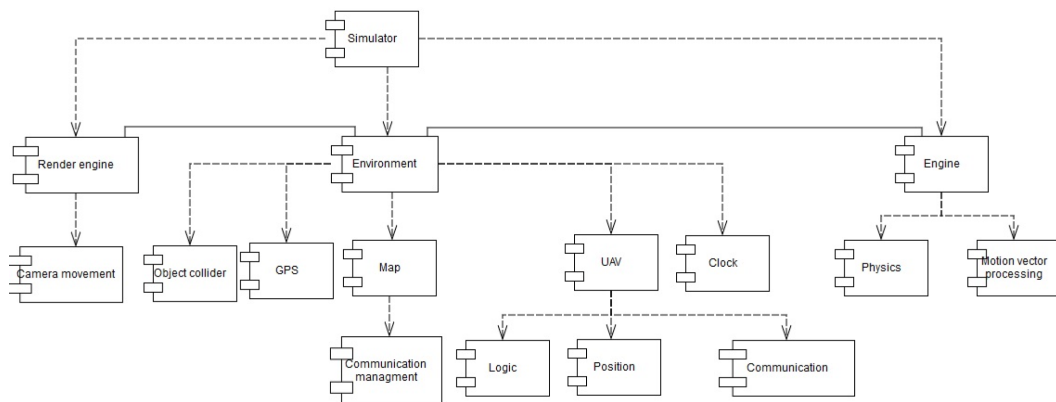
Celý systém sa rozdeľuje na 2 základné časti. *Engine* a *RenderEngine*. Komunikáci medzi nimi je zabezpečená pomocou UDP protokolu. Oba hlavné komponenty môžu bežať nezávisle na rôznych strojoch, potrebujú však byť spojené.




Obr. 1: Deployment diagram

### 3.1.1 Komponenty simulátora

Keďže architektúra systému uvedená vyššie je veľmi strohá (čo ale zodpovedá reálnemu návrhu systému typu simulátora), identifikovali sme všetky kľúčové komponenty, aby bolo viditeľné, z akých hlavných častí náš prototyp bude pozostávať.




Obr. 2: Component diagram

Ako je na diagrame vidieť, troma hlavnými komponentami sú: Render engine, Environment a Engine. V nasledujúcej časti opíšeme ich úlohu.

### Engine

Riadiaci komponent v architektúre simulátora, ktorý inicializuje a riadi beh simulácie. Prebiehajú v ňom výpočty pohybov objektov v prostredí mapy, s využitím vplyvov poveternostných podmienok a fyzikálnych zákonov. Komponent *Engine* je logicky spojený s komponentom *Environment*, s ktorým si navzájom vymieňajú dáta nevyhutne potrebné na tieto výpočty.

### Environment

Tento komponent pokrýva celú oblasť mapy, objektov obsiahnutých v tejto mape, ako aj ich vzájomnú interakciu. Uchováva si informácie s umiestnením jednotlivých UAV v prostredí, umožňuje im prijímať a odosielať správy, pomocou ktorých je vykonávaná všetka komunikácia medzi UAV zariadeniami. Logika zabezpečujúca šírenie správ (resp. logiku celej komunikácie) medzi dronmi v priestore vzhľadom na špecifiká jednotlivých typov prenosov a vlastností jednotlivých objektov prostredia je obsiahnutá v komponente *Map*. Komponent *Environment* vďaka komponentu *GPS* poskytuje rozhranie, ktoré simuluje GPS systém a tak UAV zariadenia dokážu nezávisle na iných komponentoch zistiť, kde sa práve nachádzajú, tak ako je to aj v reálnom svete. Komponent *Clock* posiela v pravidelných intervaloch signály, na základe ktorých sú vyzvané UAV zariadenia, aby poslali informácie týkajúce sa

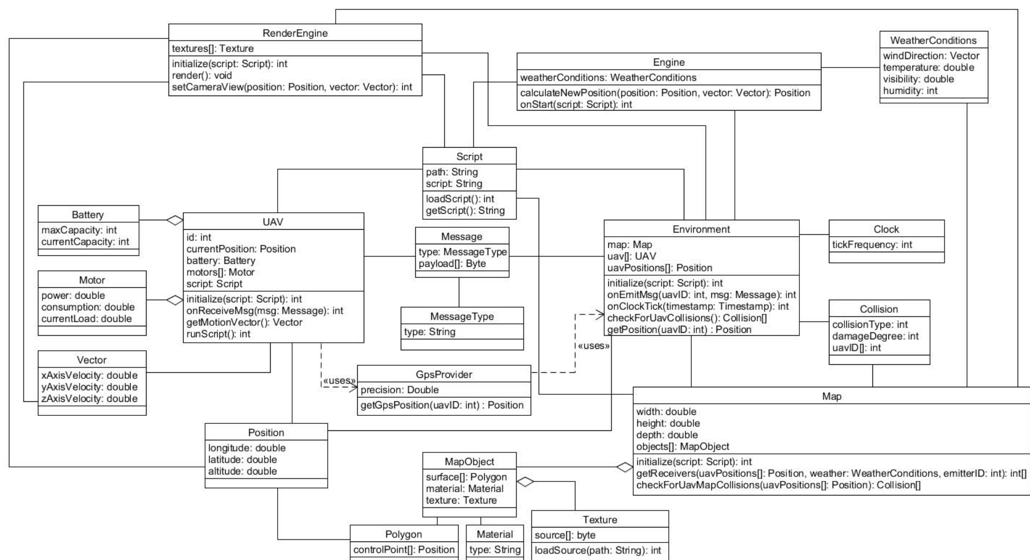
smeru a rýchlosti pohybu vykonaných od posledného signálu hodín. Tieto informácie sú následne v komponente *Engine* transformované do aktuálnej polohy UAV zariadení. V momente, keď sú informácie o nových polohách UAV zariadení poslané naspäť komponentu *Environment*, tento ešte následne pomocou komponentu *Object collider* vypočíta kolízie, ktoré nastali medzi UAV zariadeniami a objektami mapy, prípadne kolízie medzi jednotlivými UAV zariadeniami.

#### Render engine

Komponent zabezpečujúci grafické vykresľovanie mapy, jej objektov a UAV zariadení, aby bolo možné v reálnom čase sledovať prebiehajúcu simuláciu a ľahko tak identifikovať problémy, ktoré nastanú počas jej behu. Vlastnosti mapy (rozmery) a jej objektov (materiál, textúra) je možné konfigurovať pomocou inicializačného skriptu pri spustení simulátora. Pomocou komponentu *Camera movement* je zabezpečená funkcionalita pohybu kamery ovládanej používateľom v priestore.

## 3.2 Diagram tried

V tejto časti prinášame podrobný pohľad na navrhnutý prototyp z hľadiska tried. Za ním nasleduje opis tried, ich atribútov a metód, ktoré odhalia princípy ich vzájomnej interakcie.




Obr. 3: Class diagram

### Engine

Je spúšťačom celej simulácie. Pri štarte sa volá metóda `onStart(script: Script): int`, ktorá spustí konfiguráciu celého systému zo skriptu na vstupe. Návratová hodnota metódy hovorí o úspešnosti inicializácie. Ak je rovná nule, inicializácia prebehla v poriadku. Hodnota väčšia ako nula špecifikuje typ chybového stav. Vstupný skript obsahuje tieto základné konfiguračné časti nastavení:

- Hodín
- Počasia
- Prostredia
- UAV zariadení
- Mapy

Po úspešnej inicializácií sa spúšťa simulácia. Trieda *Engine* poskytuje tiež metódu `calculateNewPosition(position: Position, vector: Vector): Position`, ktorej hlavnou úlohou je vypočítať novú aktuálnu pozíciu na základe údajov o súčasnej pozícií UAV zariadenia a údajoch o smere a veľkosti

pohybu. Táto metóda je pre každé UAV zariadenie volaná triedou *Environment* vždy, keď príde signál hodín z triedy *Clock* a UAV zariadenia vrátia pomocou poslania správy informáciu o vykonanom pohybe.

### WeatherConditions

Trieda uchovávajúca informácie o stave počasia v simulátore, ktoré sú využívané pri výpočtoch pozícií UAV zariadení triedou *Engine* a tiež pri zisťovaní dosahu signálu daného typu komunikácie triedou *Map*. Vlastnosti počasia sú konfigurovateľné pomocou inicializačného skriptu na začiatku behu simulácie.

### Script

Trieda určená na načítavanie a uchovávanie konfiguračného skriptu. Metóda `loadScript(): int` načíta skript zo vstupného súboru určeného atribútom `path`. Výstup metódy `getScript(): String` je používaný pri inicializácií viacerých tried:

- *RenderEngine*
- *Engine*
- *Environment*
- *Map*
- *UAV*

### Environment

Trieda simulujúca prostredie reálneho sveta. Obsahuje inštanciu triedy *Map*, pole inštancií triedy *UAV*, pre každé UAV zariadenie v simulátore jednu. Ako posledný atribút obsahuje pole inštancií triedy *Position*, v ktorom sú uchovávané reálne pozície UAV zariadení. Tieto sú pravidelne aktualizované pri prijatí signálu z triedy *Clock*. Trieda *Environment* je zodpovedná za veľkú časť logiky, ktorá je v simulátore potrebná. Túto logiku zabezpečujú jej nasledovné metódy:

- **`initialize(script: Script): int`**
  Metóda je volaná pri spustení simulátora triedou *Engine*, ktorá poskytne príslušný skript, ktorý obsahuje funkcie, ktoré následne preťažia metódy triedy *Environment*, čím používateľ môže definovať vlastné správanie prostredia.

- **`onEmitMsg(uavID: int, msg: Message): int`**
  Metóda je volaná triedou *UAV*, keď chce UAV zariadenie poslať správu ostatným zariadeniam alebo systémovú správu prostrediu *Environemt*. Spolu s metódou `onReceiveMsg(msg: Message): int` v triede *UAV* zabezpečuje hlavnú komunikáciu v simulátore. Parameter `uavID` definuje odosielateľa správy (UAV zariadenie).

- **`onClockTick(timestamp: Timestamp): int`**
  Metóda obsahuje logiku vykonanú pri každom signále hodín z triedy *Clock*. Súčasťou tejto logiky je rozposlanie systémových správ všetkým UAV zariadeniam aby aktualizovali informácie o svojom pohybe. Metóda tiež spúšťa udalosti pre jednotlivé UAV zariadenia, ktoré boli definované v inicializačnom skripte.

- **`checkForUavCollisions(): Collistion[ ]`**
  Metóda sa spúšťa pri vrátení novej pozicíe UAV zariadenia z triedy *Engine* a jej hlavnou úlohou je overiť, či daný pohyb UAV zariadenia nespôsobil kolíziu tohto zariadenia s prostredím, prípadne s inými UAV zariadeniami. Aby bolo možné overiť kolízie s mapou a jej objektami, volá sa metóda `checkForUavCollisions(uavPositions[ ]: Position): Collision[ ]` triedy *Map*.

- **`getPosition(uavID: int): Position`**
  Metóda na získanie reálnej pozície UAV zariadenia v prostredí. UAV zariadenia k nej nevedia pristupovať priamo, ale iba cez triedu *GpsProvider*, ktorá môže do výpočtu pozície zaniesť istú mieru nepresnosti, čím sa simuluje GPS systém z reálneho sveta.

### Clock

Trieda, ktorá v pravidelných intervaloch volá metódu `onClock(timestamp: Timestamp): int` triedy *Environment*, čím jej posiela časové signály, na základe ktorých potom táto trieda vykonáva potrebné operácie. Frekvencia posielania signálov je uložená v atribúte v `tickFrequency` v milisekundách.

### Collision

Trieda predstavujúca rôzne typy kolízií UAV zariadení. Atribút `collisionType` určuje, či sa jedná o kolíziu dvoch UAV zariadení, alebo kolíziu UAV zariadenia s objektom prostredia. Všetky UAV zariadenia, ktoré sa dostali do kolízie sú špecifikované pomocou identifikátorov uložených v atribúte `uavID[ ]`. Atribút `dameDegree` určuje mieru poškodenia, ktorá má následný dopad na ďalší priebeh simulácie.

### Message

Inštancie triedy *Message* sú používané na posielanie správ UAV zariadeniami a prostredím. Uchovávajú v sebe informácie o type správy v atribúte `type` a samotný obsah správy v atribúte `payload`.

### MessageType

Pomocou triedy *MessageType* je možné rozlišovať viacero typov správ, ktoré si vzájomne posielajú UAV zariadenia a prostredie. Každá správa môže byť jedného z nasledujúcich typov:

- **Start** - Špeciálny typ správy, ktorý je odoslaný každému UAV zariadeniu na začiatku simulácie. Umožňuje tým vykonať špecifické kroky potrebné iba pro štarte simulácie.
- **Clock tick** - Typ správy, posielaný triedou *Environment* vždy, keď chce požiadať UAV zariadenie o zaslanie informácie o vykonanom pohybe, potrebnej na aktualizáciu jeho súčasnej pozície.
- **Environment** - Riadiaci typ správy posielaný medzi prostredím a UAV zariadeniami. Tento typ správy môže slúžiť na zaslanie udalosti špecifikovanej

používateľom v inicializačnom skripte.

- **UAV** - Typ správy určený na komunikáciu medzi UAV zariadeniami. Keď trieda *Environment* dostane takýto typ správy, zistí, ktoré zariadenia sú v komunikačnom dosahu vysielajúceho UAV zariadenia a týmto správu doručí.

### GpsProvider

Trieda simulujúca GPS systém reálneho sveta. Keď chce UAV zariadenie získať svoju aktuálnu polohu, zavolá metódu `getGpsPosition(uavID: int): Position`, ktorá zavolá metódu `getPosition(uavID: int): Position` triedy *Environment*. K reálnej pozícií vrátenej z triedy *Environment* je následne pridaná nepresnosť. Jej veľkosť je konfigurovateľná a uchovávaná v atribúte `precision`. Takto znepresnená poloha je vrátená UAV zariadeniu. Z pohľadu UAV zariadenia je poskytnutá rovnaká funkcionalita ako v prípade využitia reálneho GPS systému pre reálne UAV zariadenie.

### Position

Trieda používaná na ukladanie pozícií UAV zariadení a objektov prostredia mapy. V troch atribútoch (`longitude`, `latitude`, `altitude`) sú uložené geografické koordináty, ktoré jednoznačne určujú ich polohu.

### Map

Trieda reprezentujúca mapu. Atribúty `width`, `height` a `depth` určujú jej rozmer v trojdimenzionálnom priestore. V atribúte `objects[ ]` sú uložené objekty mapy (terén, stromy, kamene, a pod.). Trieda *Map* disponuje týmito metódami zabezpečujúcimi jej hlavnú funkcionalitu:

- **`initialize(script: Script): int`**
  Metóda je volaná pri spustení simulátora triedou *Engine*, pričom sa načítajú všetky objekty mapy, ich materiály a textúry. Súčasne je preťažená metóda `getReceivers(uavPositions[ ]: Position, weather: WeatherConditions, emitterID: int): int[ ]`, ktorá špecifikuje šírenie signálu v mape.

- **`getReceivers(uavPositions[ ]: Position, weather: WeatherConditions, emitterID: int): int[ ]`**
  Metóda je volaná triedou *Environment*, keď táto potrebuje zistiť ku ktorým UAV zariadeniam sa dostane komunikačný signál vysielajúceho zariedenia. Táto metóda je plne konfigurovateľná používateľom, čo necháva otvorený priestor pre širokú škálu spôsobov komunikácie (WiFi, optický prenos informácií, akustický prenos informácií, a pod.). Metóda na výstupe vracia pole identifikátorov jednotlivých UAV zariadení, ktoré sa aktuálne nachádzajú v dosahu signálu vysielajúceho zariadenia.
- **`checkForUavMapCollisions(uavPositions[ ]: Position): Collision[ ]`**
  Metóda, ktorá zisťuje kolízie UAV zariadení s objektami mapy. Je volaná triedou *Environment* vždy, keď je vypočítaná nová poloha UAV zariadení.

#### MapObject

Trieda predstavujúca jednotlivé objekty v mape. Je špecifikovaná povrchom v atribúte `surface`, ktorý sa skladá z poľa polygónov. Materiál objektu je definovaný v atribúte `material` a môže byť využitý pri výpočte šírenia signálu v simulačnom prostredí. Atribút `texture` definuje textúru daného objektu, ktorá ma byť použitá pri jeho vykreslení pomocou triedy *RenderEngine*.

#### Polygon

Trieda na reprezentáciu polygónov opisujúcich povrchy objektov mapy. Polygón sa skladá z viacerých bodov, uložených v atribúte `controlPoint[ ]`. Množina riadiacich bodov určuje dvojdimenzionálnu plochu v trojdimenzionálnom priestore. Súbor polygón teda umožňuje opísať povrch akéhokoľvek trojdimenzionálnom priestore.

#### Material

Trieda opisujúca typ materiálu objektu mapy. Tento typ je uložený v atribúte `type`.

#### Texture

Trieda obsahujúca textúru objektu mapy, ktorá sa načítava pomocou metódy `loadSource(path: String): int`. Táto metóda je volaná pri spustení simulátora. Načítaná textúra je potom uložená v atribúte `source[ ]`.

#### UAV

Trieda reprezentujúca jednotlivé UAV zariadenie ako objekty v prostredí. Každé UAV zariadenie sa riadi vlastnou logikou. V simulátore k nemu pristupujeme ako k autonómnemu objektu. Jediný spôsob, akým UAV zariadenie dokáže s inými objektami komunikovať sú správy (trieda *Message*). Každému UAV zariadeniu je pri spustení pridelený jednoznačný identifikátor, pod ktorým UAV zariadenie následne počas behu celej simulácie vystupuje. Tento identifikátor je uložený v atribúte `id`. Do atribútu `currentPosition` sa ukladá posledná známa pozícia získaná od triedy *GpsProvider*, ktorá simuluje GPS systém. Každé UAV zariadenie sa skladá z jednej batérie (atribút `battery`) a z ľubovoľného počtu motorov (atribút `motors[ ]`). Trieda *UAV* má definované nasledovné metódy:

- **`initialize(script: Script): int`**
  Metóda je volaná pri spustení simulátora triedou *Engine*. Pri tomto sú preťažené metódy `onReceiveMsg(msg: Message): int` a `runScript(): int`, ktoré definujú správanie UAV zariadenia. Tiež slúži na definovanie typu UAV zariadenia podľa inicializačného skriptu (počet motorov, kapacita batérie, a pod.).

- **`onReceiveMsg(msg: Message): int`**
  Metóda, ktorú volá trieda *Environment*, keď chce doručiť UAV zariadeniu správu. UAV zariadenie následne na základe typu prijatej správy rozhodne o svojom ďalšom správaní. Toto správanie je na začiatku simulácie načítané zo vstupných skriptov, ktoré si definuje používateľ.

- **`getMotionVector(): Vector`**
  Metóda generuje vektor pohybu na základe informácií o stave aktuálneho zaťaženia a výkonu motorov UAV zariadenia.

- **`runScript(): int`**
  Metóda, ktorá sa zavolá bezprostredne po ukončení metódy `initialize(script: Script): int`. Metóda spustí skript s logikou UAV zariadenia, ktorý beží počas celej doby simulácie, prípadne počas existencie UAV zariadenia v prostredí.

### Battery

Reprezentuje vlastnosti batérie UAV zariadenia a jej stav. V atribúte `maxCapacity` je uložená hodnota reprezentujúca kapacitu plne nabitej batérie v *mAh*. Atribút `currentCapacity` obsahuje hodnotu reprezentujúcu zostávajúcu kapacitu batérie v *mAh*.

### Motor

Predstavuje motor UAV zariadenia. V atribúte `power` je uložená hodnota maximálneho výkonu motora, zatiaľ čo v atribúte `currentLoad` je jeho aktuálne vyťaženie. Atribút `consumption` udáva spotrebu motora pri maximálnom výkone.

### Vector

Trieda používaná na vyjadrenie zmeny pohybu UAV zariadenia. Je reprezentovaná troma atribútmi `xAxisVelocity`, `yAxisVelocity` a `zAxisVelocity`, ktoré uchovávajú rýchlosť pohybu po danej osi.

### RenderEngine

Zabezpečuje vykresľovanie simulácie, vďaka čomu má používateľ možnosť sledovať simuláciu v reálnom čase. Táto trieda obsahuje nasledovné metódy:

- **`initialize(script: Script): int`**
  Konfiguruje inštanciu tejto triedy podľa skriptu zadaného používateľom.

- **`render(): void`**
  Vykreslí aktuálny stav mapy spolu s objektami a UAV zariadeniami na obrazovku.

- **`setCameraView(position: Position, vector: Vector): int`**

  Nastaví kameru do prostredia simulátora.

# 4 Moduly systému

## 4.1 Modul inicializácie systému

Úlohou umodul inicializácie systému je konfigurácia aplikácie, reprezentácia základných vlastností systému. Tento modul sa používa vždy pri štarte simulátora a je spúšťaný prostredníctvom metódy `onStart(script: Script): int` komponentu *Engine*. Počas analýzy sme dospeli k záveru, že simulátor by mal umožňovať čo najvyššiu možnú konfigurovateľnosť. Ak by používateľ takúto možnosť nemal, pravdepodobne by ani nemal záujem simulátor využívať pretože jeho funkcionalita by bola príliš obmedzená našim pohľadom na problém.

### 4.1.1 Návrh

Pri návrhu riešenia počiatočnej inicializácie sme sa rozhodli pre využitie dokumentu vo formáte JSON (*JavaScript Object Notation*). Je to najmä z dôvodu jednoduchosti, je ľahko čitateľný a porozumiteľný. Je tiež s obľubou používaný a rozšírený. Na ukážke (1) sa nachádza náš návrh inicializačného dokumentu formátu JSON. Tento dokument bude slúžiť pre nastavenie hodín, simulácie počasia, logiky prostredia a jednotlivých UAV a tiež vlastností mapy. Nasledujúce body opisujú JSON dokument priložený nižšie.

- **clock** - Objekt predstavujúci pravidelnú časovú signalizáciu. Ide o pomerne prostý objekt systému. Jeho úlohou je rozdeľovanie času behu simulátora na pravidelné intervaly. Vždy na hranici dvoch intervalov (pri tiku hodín) spustí komponent *Environment* metódu `onClockTick(timestamp: Timestamp)`. Atribútmi objektu `clock` sú frekvencia "tikov" a jednotka času.
- **weather** : Atribútmi tohto objektu sú atribúty triedy *WeatherConditions*. Teda ide o počiatočné nastavenie počasia, ktoré by mal simulovať komponent *Engine*.
- **environment** - V prípade objektu environment nenastavujeme žiadne špeciálne atribúty. Dokument však nastavuje atribúty komponentov, ktoré *Environment* spravuje. Zároveň pri inicializácii načítavame inicializačný skript

zo súboru definovaným používateľom. Tento skript je volaný metódou `initialize(script: Script)`, ktorá vyvolá a spustí logiku naprogramovanú v zdrojovom kóde. Týmto spôsobom je umožnené používateľovi nakonfigurovať správanie prostredia podľa vlastných požiadaviek. To je realizované pomocou preťaženia metód komponentu *Environment* v inicializačnom skripte.

- **UAVs** - Objekt UAVs obsahuje pole UAV pričom má používateľ možnosť pre každé vytvoriť špecifickú konfiguráciu. Každý objekt UAV je definovaný rôznymi atribútmi, ktoré vždy závisia od modelu. Samozrejme ide o simulátor a to znamená, že používateľ môže definovať UAV s vlastnosťami, ktoré nezodpovedajú realite (napr.: dron s nekonečnou kapacitou batérie a pod.). Okrem objektov opisujúcich vlastnosti UAV je znovu možné a pravdepodobne aj žiaduce načítať skript definujúci správanie konkrétneho zariadenia. Ide o rovnaký princíp ako pri komponente *Environment* kedy používateľ chce mať kontrolu nad tým čo a ako sa v simulátore deje. Každé zariadenie tiež môže obsahovať pole udalostí opísaných atribútmi `triggerTime`, `message_type` a `message_payload`. Tieto udalosti sa vyvolajú vždy v presný stanovený čas. Vzhľadom na to, že UAV v simulátore vykonávajú iba skript im určený a ich kontakt s ostatnými komponentami je obmedzený len na komunikáciu prostredníctvom správ, sú udalosti reprezentované správami. Tie budú v určitom "tiku" hodín vyvolané a odoslané zariadeniu, ktoré ich spracuje na základe logiky implementovanej v skripte špecifického UAV.

- **map** - Posledným objektom dokumentu je mapa. Tá je definovaná svojimi rozmermi a objektami nachádzajúcimi sa v nej. Jednotlivé objekty budú načítané zo štruktúrovaného dokumentu, ktorý bude obsahovať ich umiestnenie, tvar, textúru. Medzi takéto objekty patrí napríklad aj terén mapy. Zároveň mapa obsahuje definíciu logiky komunikácie medzi UAV. Ide o simulovanie odosielania a prijímania správ, pričom komunikácia je definovaná typom použitej technológie a logika sa nachádza v skripte napísanom používateľom. V zásade by malo byť možné definovať ľubovoľný typ komunikácie (wifi, optická či akustická signalizácia a pod.). Všetko závisí od požiadaviek používateľa. Táto logika je zahrnutá v mape z dôvodu existencie rôznych objektov, ktoré môžu brániť zariadeniam v komunikácii. K účelu detegovania rušivých objek-

tov pri komunikácii slúži metóda `getReceivers(uavPositions[ ]: Position, weather: WeatherConditions, emitterID: int)` komponentu *Map*, ktorá je volaná komponentom *Environment* v prípade, že je potrebné odoslať nejakú správu zariadeniam UAV.

### 4.1.2 Implementácia

Modul bol rozširovaný postupne ako sme potrebovali inicializovať nové časti systému a ich vlasnosti. Hlavný inicalizačný súbor obsahuje odkazy na inicializačné súbory nižšej úrovne, ktoré opsiujú vlastnosti niektorých konkrétnych častí systému. Pre všetky inicializačné súbory sme sa držali striktne jednotného formátu JSON. Modul sa využíva iba raz - na začiatku. Keď je raz systém inicializovaný a spustený, nepotrebuje už prístup k inicializačným súborom.

### 4.1.3 Testovanie

Tento modul si nevyžadoval žiadne špeciálne testovanie. Princíp fungovania je veľmi jednoduchý. Potrebné hodnoty sú uložené v dohodnutom formáte. Pri spustení programu sa načítajú a uložia do príslušných premenných. Testovanie prebiehalo postupne, vždy s novou časťou sa otestovalo aj či je správne inicializovaná z inicializačných súborov.

```
{
  "clock": {
    "frequency": INTEGER,
    "units": UNIT [ms | s | m | h]
  },
  "weather": {
    "windDirection": VECTOR [x, y, z],
    "temperature": DOUBLE,
    "visbility": DOUBLE,
    "humidity": INTEGER
  },
  "environment": {
    "scriptSource": PATH
  },
  "UAVs": {
    "count": INTEGER,
    "arrayOfUAVs": [  {
      "motors": [   {
          "power": DOUBLE,
          "consumption": DOUBLE
        },
        ...
      ]
      "batteryCapacity": INTEGER,
      "scriptSource": PATH,
      "startingPosition": POSITION [lat, long, alt],
      "events":[  {
          "triggerTime": TIMESTAMP,
          "message_type": STRING,
          "message_payload": DATA
        },  {
          "triggerTime": TIMESTAMP,
          "message_type": STRING,
          "message_payload": DATA
        },
        ...
      ]
    },
      ...
  },
  "map": {
  "dimensions": {
    "width": DOUBLE,
    "height": DOUBLE,
    "depth": DOUBLE
  }
  "objects": {
    "count": INTEGER,
    "source": PATH
  },
  "communication": {
      "type": STRING,
      "scriptSource": PATH
  }
  }
}
```

Ukážka 1: Návrh JSON dokumentu inicializácie systému.

## 4.2 Modul komunikácie

Modul pre realizáciu komunikácie medzi UAV zariadeniami sme identifikovali hneď v počiatkoch analýzy systému. Je jasné, že pri kooperatívnom vykonávaní úloh, čo je aj prípad spolupráce UAV v roji, je komunikácia základným problémom, ktorý treba riešiť. V našom prípade bude nutné využiť niektorý z bezdrôtových prístupov komunikácie, keďže je prakticky nemožné využívať fyzické spojenie medzi zariadeniami. Bezdrôtová komunikácia môže byť realizovaná rôznymi spôsobmi. Či už sa bude využívať niektorý zo známych komunikačných štandardov ako IEEE 802.11 (WI-FI), či Bluetooth alebo neštandardný prístup ako napríklad optická signalizácia, vždy narazíme na niekoľko nedostatkov, ktoré daná technológia so sebou prináša. Pri simulácii však pre nás vôbec nie je podstatné o aký typ komunikácie pôjde. Našim cieľom je vytvoriť aplikáciu, ktorá bude umožňovať jej používateľom použiť ľubovoľnú technológiu. Jej využitie bude následne možné otestovať v simulácii a overiť tak jej vhodnosť, či nevhodnosť pre danú úlohu.

### 4.2.1 Návrh

Pri návrhu sa stále držíme maximálnej konfigurovateľnosti simulátora a jeho komponentov. V prípade simulácie druhu použitej technológie prenosu dát sme obmedzení len použitým programovacím jazykom. Komunikácia medzi UAV navzájom či UAV a komponentom *Environment* bude riešená pomocou odosielania správ. Ide o dve metódy a to `onEmitMsg(uavID: int, msg: Message)` triedy *Environment* a `onReceiveMsg(msg: Message)` triedy *UAV*. Princíp simulácie výmeny správ sme navrhli tak, že každé zariadenie ak chce poslať správu iným zariadeniam, zavolá metódu `onEmitMsg()` do ktorej ako argument vloží správu na odoslanie. Správa bude určitého typu a bude obsahovať informáciu pre adresáta. Komponent *Environment* následne zavolá metódu `getReceivers(uavPositions[ ]: Position, weather: WeatherConditions, emitterID: int)` komponentu *Map*, ktorá vráti zoznam UAV v dosahu použitej komunikačnej technológie. Nakoniec *Environment* vyvolá metódu `onReceiveMsg(msg: Message)` pre každé zo zariadení v zozname. Jednotlivé UAV spracujú správu na základe postupu implementovaného v skripte pre dané zariadenie. To aký spôsobom je vygenerovaný zoznam dostupných zariadení pre prijatie správy je implementované v skripte pre komunikáciu.

### 4.2.2 Implementácia

Komunikácia sa šíri prostredím podľa vlastností určených používateľským skriptom. Je teda možné implmentovať akýkoľvek spôsob komunikácie dronov (laser, dymové signály, zvukové signály...). Skript, ktorý zabezpečuje určovanie UAV zariadení, ku ktorým sa správa dostane len musí byť schopný určiť, ako sa signál v prostredí šíri a ku ktorým UAV zariadeniam sa dostane.

### 4.2.3 Testovanie

Pre overenie korektneho fungovania a zistenia ktorým UAV zariadeniam príde vyslaná správa sme implmentovali podrobné logovanie udalostí v simulátore a najmä odosielania prijímania správ. Každá správa, ktorá je vyslaná je zalogovaná. Loguje sa pri tom odosielateľ, požadovaní príjemcovia (komu odosielateľ chcel aby správa prišla) a reálni príjemcovia (komu správa naozaj prišla). Vďaka tomu sme vedli overiť, či správy chodia tým UAV zariadeniam, ktorým by vzhľadom na podmienky šírenia signálu chodiť mali.

## 4.3 Modul simulácie

Tento modul bol identifikovaný zo zrejmého dôvodu. Simulátor potrebuje simulovať správanie zariadení a taktiež prostredia. To sú dve hlavné časti, ktoré by mal poskytovať modul simulácie. Tie navzájom spolu súvisia pretože jednotlivé UAV sú zasadené v prostredí s určitými vlastnosťami a teda priamo súvisia s jeho konfiguráciou.

### 4.3.1 Návrh

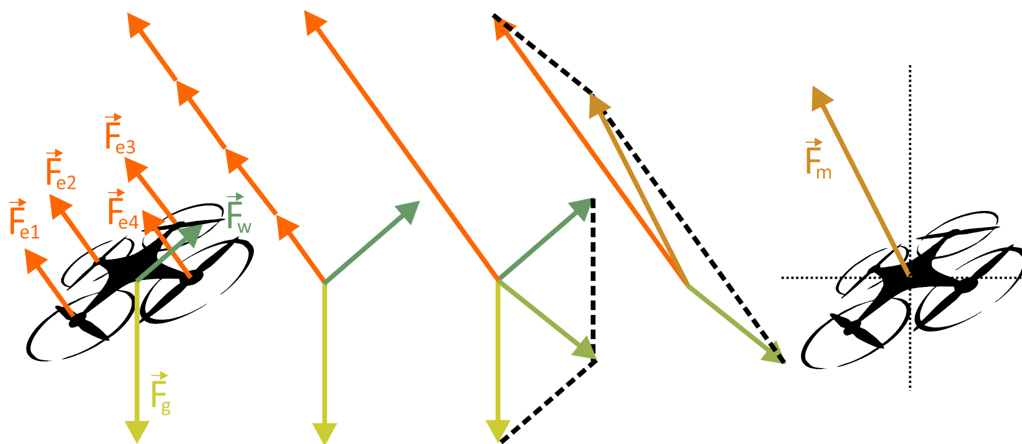
#### Simulácia prostredia

Simulácia vlastností prostredia bude vykonávaná v komponente *Engine*. Ide o výpočtový komponent, ktorý spolupracuje s komponentami *Environment* a *Render engine*. Z triedy *WeatherConditions* získava údaje o simulovaných meteorologických javoch ako je vietor, teplota a pod. Tieto javy bezprostredne ovplyvňujú UAV pohybujúce sa v prostredí, ale taktiež napríklad aj komunikáciu medzi nimi. Napríklad pri slabej viditeľnosti je neefektívne použiť komunikáciu prostredníctvom optického

signalizovania. Vlastnosti prostredia by malo byť možné meniť v čase, respektíve malo by byť možné nastaviť zmenu poveternostných podmienok skriptom ak to bude používateľ vyžadovať.

### Simulácia zariadení

Ako už bolo spomenuté, zariadenia sú priamo závislé od prostredia a od jeho vlastností. Preto bude pohyb zariadení v prostredí počítaný v závislosti od poveternostných podmienok a iných bežných fyzikálnych javov (gravitácia a pod.). Princíp je založený na sčítavaní vektorov. Každá vektorová fyzikálna veličina sa dá reprezentovať ako vektor, teda má veľkosť a smer. Po sčítaní všetkých vektorov pôsobiacich na zariadenie vieme vypočítať výsledný vektor pohybu UAV. Z vektoru a aktuálnej polohy sa vypočíta nová poloha zariadenia v priestore. Tieto informácie sú v návratovej hodnote metódy `calculateNewPosition(position: Position, vector: Vector): Position` poskytnuté komponentu *Environment*, ktorý túto metódu zavolá vždy keď UAV zariadenia odošlú správu o zmene polohy. Na nasledujúcom obrázku je jednoduchý príklad výpočtu vektorového súčtu pre 4 - motorový dron, na ktorého pôsobí odpor vzduchu (sila vetra $\vec{F_w}$) a gravitačná sila $\vec{F_g}$. Výsledný vektor pohybu je označený ako $\vec{F_m}$.




Obr. 4: Príklad vektorového súčtu

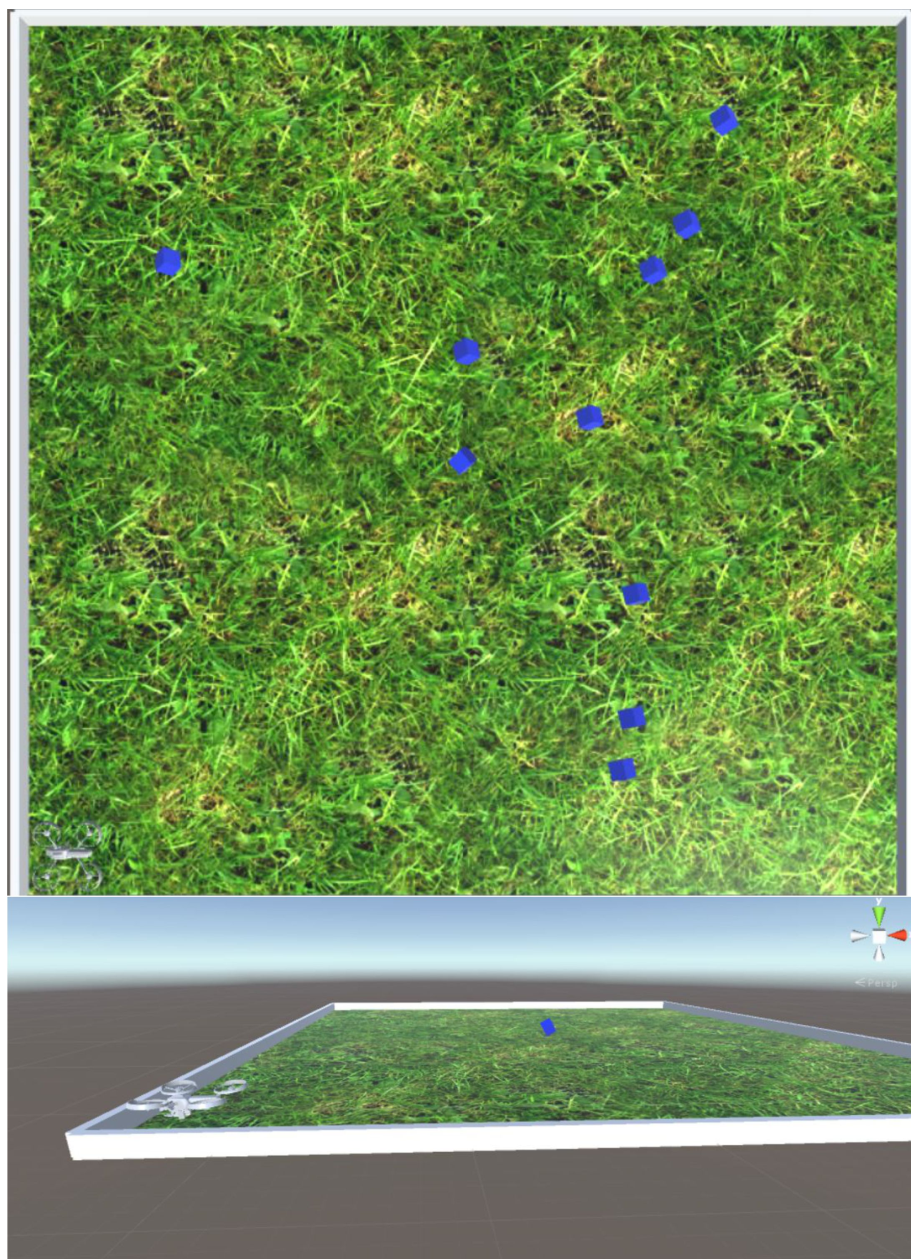
Podstatné je, že zariadenia nemajú informáciu o tom, na akej polohe sa nachádzajú, respektíve nemajú možnosť zmeniť svoju polohu z bodu A do bodu B inak než vykonaním nasledujúceho cyklu.

1. Zavolanie metódy `getGpsPosition(auvID: int) : Position` triedy *GpsProvider* ,ktorá zavolá metódu `getPosition(uavID: int) : Position` triedy *Environment*, ktorá vráti približnú fyzickú pozíciu zariadenia (bod $A$).
2. Výpočet výkonov motorov UAV pravdepodobne potrebného k presunu z bodu $A$ do bodu $B$.
3. Výpočet vektoru pohybu zavolaním metody `getMotionVector() : Vector`.
4. Odoslanie správy triede *Environment* o vykonanom pohybe.
5. Zavolanie metódy `getPosition(uavID: int): Position`.
6. Výpočet novej polohy UAV v priestore triedou *Engine*.
7. Zaznamenanie novej polohy zariadenia v triede *Map*.
8. Zavolanie metódy `getGpsPosition()` zariadením pre zistenie aktuálnej polohy (aktuálny bod $A$).
9. Ak sa zariadenie nenachádza v bode $B$ opakuje sa proces od bodu 2.

Týmto spôsobom je možné simulovať pohyb zariadení v prostredí, ktorý je možné prirovnať k realite. To akým spôsobom budú jednotlivé zariadenia vyhodnocovať svoj postup pri pokuse dostať sa z bodu $A$ do bodu $B$ závisí na logike zariadenia implementovanej v skripte pre každé UAV. Využitie fyzikálnych javov bude realizované pomocou 3D Engine-u Unity 3D.

### 4.3.2 Implementácia

Modul sme postavili na vektoroch: každé UAV zariadenie udáva prostrediu, ktorým smerom a ako rýchlo sa chce pohybovať, teda ako by zaplo v reálnom svete motory. Prostredie túto informáciu vyhodnotí na základe jeho polohy a vplyvu prostredia. Ak je teda nastavený nenulový vietor, odfukne UAV zariadenie niektorým smerom - tým ktorým fúka.



Obr. 5: Zjednodušený model simulácie. UAV zariadenia oznamujú svoj želaný pohyb prostrediu. To aplikuje vplyv prostredia (vietor) a vypočíta novú pozíciu.

### 4.3.3 Testovanie

Počas testovania simulácia sme menili vektory smeru pohybu, rovnako aj vektory fúkania vetra. Následne sme pomocou logov zo simulátora overovali, či sa UAV

zariadenie v určitom čase nachádzalo na správnom mieste, vzhľadom na vektor jeho pohybu a nastavený vplyv prostredia.

## 4.4 Modul vykresľovania simulácie

Ako doplnkom simulátora je modul vykresľovania simulácie - všeobecne ho označujeme ako **Render Engine**. Simulátor je samozrejme možné spustiť aj bez tohto doplnku. Výhodou zobrazenia je lepšie predstavenie si priebehu simulácie.

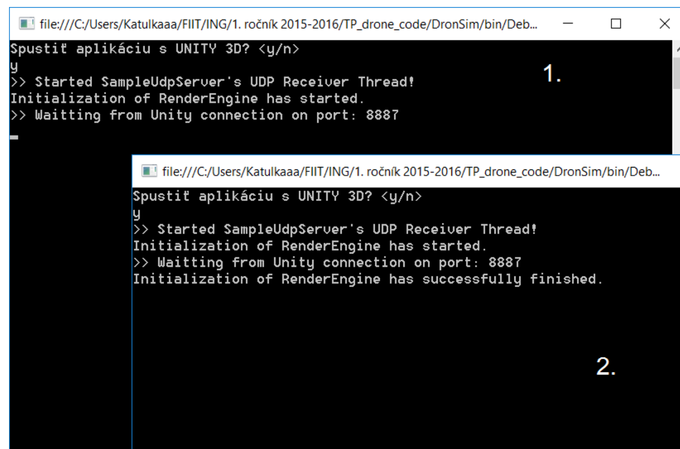
Na vykresľovanie simulácie sme si zvolil nástroj Unity 3D. Bolo potrebné však prepojiť logiku simulácie, ktorá sa nachádza v konzolovej c# aplikácii a aplikáciu v Unity 3D.

Na základe zadefinovania inicializačných JSON súborov sa zhodnotilo, že aplikácie si budú medzi sebou vymieňať len tento týp dát - resp. Render Engine prijme jeden inicializačný JSON súbor, na základe ktorého sa nastaví veľkosť mapy, počet dronov - ich veľkosť, počiatočná pozícia, prípadne farebné rozlíšenie a potom už len bude očakávať JSON súbory, v ktorých sa budú nachádzať aktualizované súradnice dronov. Na tento typ komunikácie bol zvolený protokol UDP. Aj keď ide o tzv. nespoľahlivý protokol, pretože príchod dát nie je potvrdzovaný a tým pádom nie je zaručené ich správne doručenie, na druhej strane je vďaka tomu časovo rýchlejší a efektívnejší.

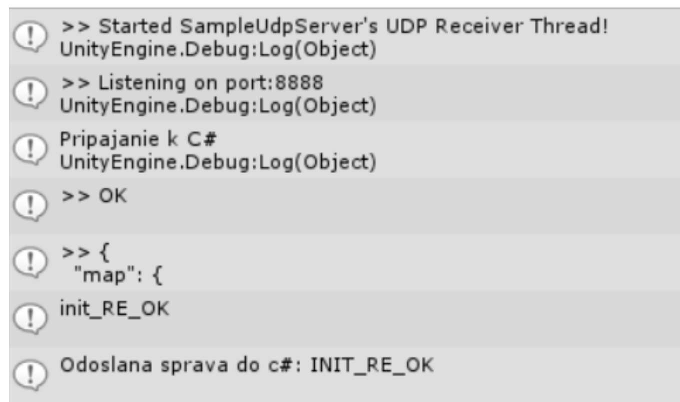
To znamená, že v oboch čatiach simulátora sa nachádza UDP server (c# - *Server.cs*, Unity 3D - *NetworkController*), ktorý neustále čaká na príjem správ (je realizovaný prostredníctvom samostatného vlákna, nakoľko ide o blokujúci proces). Postup výmeny údajov medzi aplikáciami je nasledovný:

1. Konzolová aplikácia c# po spustení spustí vlákno, v ktorom beží UDP server a počúva na zvolenom porte (obr. 6 1.).
2. Aplikácia v Unity 3D po spustení pošle správu „unity_connection“ na adresu a port c# aplikácie (momentálne IP adresa localhost-127.0.0.1 a port 8887).
3. Aplikácia c# si z tejto správy vytiahne IP adresu a port Unity (momentálne IP adresa localhost-127.0.0.1 a port 8888) a odpovie na ňu správou „OK“. Následne posiela inicializačný JSON súbor.

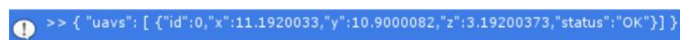
4. Unity po prijatí inicializačného JSON súboru (ukážka 2) a po jeho spracovaní posiela do c# správu „INIT_RE_OK“, ktorou hovorí že je pripravené na prijímanie aktualizačných JSON súborov (obr. 6 2., obr. 7). Spustí sa simulácia a v určitých (pravidelných) intervaloch sú do Unity posielane aktualizácie o pozíciách dronov. (obr. 8)




Obr. 6: Ukážka čakajúcej operácie na pripojenie Unity




Obr. 7: Ukážka výmeny správ pri inicializácii simulácie.



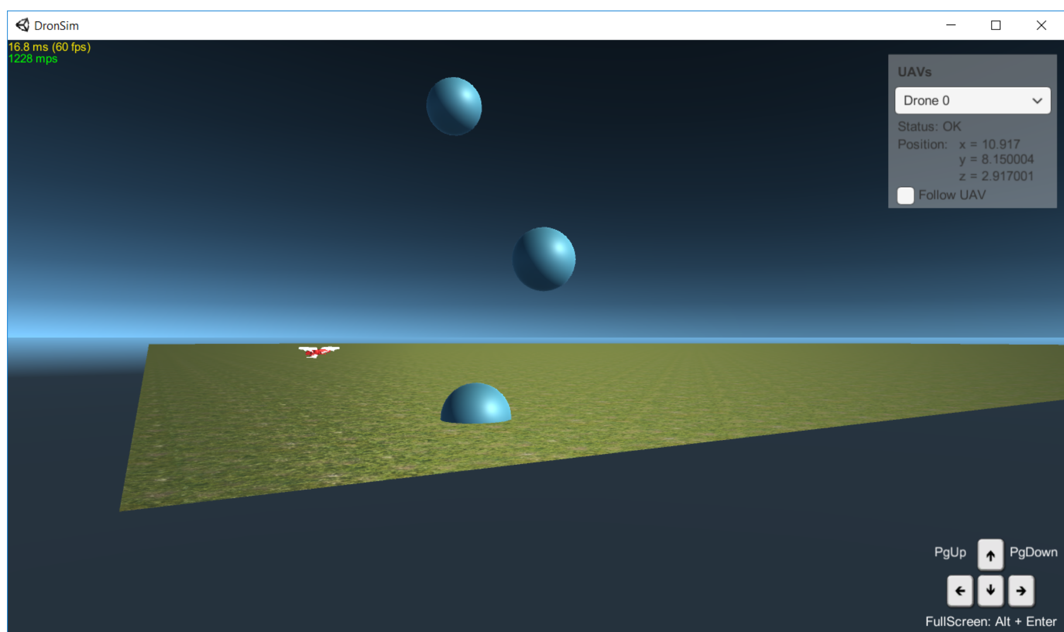

Obr. 8: Ukážka aktualizačného JSON-u.

```
{
  "map": {
    "h": 1000,
    "w": 1000,
    "l": 1000,
    "obj": ""
  },
  "uavs": [
    {
      "id": 0,
      "x": 10,
      "y": 0,
      "z": 2,
      "r": 2,
      "battery": 100,
      "consumption": 1,
      "speed": 10,
      "source": ".\\..\\..\\PythonScripts\\master.py",
      "obj": "red"
    }
  ],
  "objects": [
    {
      "id": 0,
      "x": 14,
      "y": 50,
      "z": 13,
      "r": 10,
      "obj": ""
    },
    {
      "id": 1,
      "x": 25,
      "y": 30,
      "z": 13,
      "r": 5,
      "obj": ""
    },
      ...
  ]
}
```

Ukážka 2: Inicializačný JSON súbor objektov simulácie.

Na obrázkoch 9 a 10 je zobrazená scéna pred a po inicializácii objektov.

Obr. 9: Scéna simulácie v Unity pred inicializáciou objektov.




Obr. 10: Scéna simulácie v Unity po inicializácii objektov.

# Prílohy

---

Príloha A: Spracované analýzy
Príloha B: Používateľská dokumentácia

---

# A Spracované analýzy

V tejto prílohe sa nachádzajú časti analýzy, ktoré sme vykonali v úvodných fázach projektu. V tom čase nebolo úplne jasné akým smerom sa bude projekt uberať. Po niekoľkých stretnutiach s externým konzultantom sme si ujasnili ciele a vytvorili architektúru systému. Následne sme pochopili že väčšinu z doposiaľ vyvorenej architektúry potrebovať nebudeme. Rozhodli sme sa však, že ju vložíme ako prílohu k dokumentu keďže bola súčasťou projektu.

## A.1 Prehľad databázových technológií

Databázové systémy sú rozdelené na dve hlavné kategórie a to SQL a NoSQL. V nasledujúcej časti opíšeme ich vlastnosti, prednosti, nedostatky a ich využitie.

**SQL**

Táto skratka vyjadruje Structured Query Language, teda štruktúrovaný dopytovací jazyk. Ide o programovací jazyk špeciálneho účelu navrhnutého pre správu dát uložených v systéme správy relačných databáz. Je založený na relačnej algebre, skladá sa z jazyka definície, manipulácie a riadenia dát.

**PostgreSQL**

Open source objektovo-relačný databázový systém. Na trhu od roku 1996. Aktuálna stabilná verzia je PostgreSQL 9.4.5 z októbra 2015. Počas svojej existencie si vybudovalo povesť spoľahlivého databázového systému. Pracuje na väčšine operačných systémov postavených na Unix-e tiež na Mac OS X a Windows-e. Dopytovacím jazykom je SQL. Podporuje väčšinu známych, používaných programovacích jazykov (C/C++, C#, Java, Perl, Python, Ruby,...). Podporuje správu geografických objektov pomocou PostGIS.

**MySQL**

Open source systém správy relačných databáz. Je populárny vo webových aplikáciách. Je napísaný v C/C++, multiplatformový (Windows, Linux, Solaris,...). Je často porovnávaný s PostgreSQL. V minulosti bolo jeho výhodou najmä rýchlosť avšak mal menej funkcionalít. V ďalších verziách sa vývojári snažili tieto rozdiely

postupne vyrovnať a zdokonaľovať systém v oblasti v ktorej mal nedostatky.

**SqLite**

Tento databázový systém je na rozdiel od iných obsiahnutý priamo v knižnici jazyka C. Jeho architektúra nie je klient-server ale naopak sa snaží byť zasadený do koncovej aplikácie. Implementuje väčšinu SQL štandardov a používa slabo typový SQL syntax, ktorý však negarantuje doménovú integritu. Ako aj vyššie spomínané databázové systémy, podporuje veľmi veľké množstvo programovacích jazykov. Je veľmi obľúbený a používaný najmä v aplikáciách využívajúcich lokálne (klientské) úložisko dát.

**NoSQL**

NoSQL databázy poskytujú mechanizmus ukladania a prístupu k dátam, ktoré sú ukladané iným spôsobom ako v relačných databázach. Motiváciou tohto prístupu zahŕňa jednoduchý dizajn, jednoduchšie škálovanie klastrov, ktoré sú pre relačné databázy problémom a majú lepšiu kontrolu dostupnosti. Dátové štruktúry použité NoSQL databázami (key-value, graf, dokument) sú trochu odlišné od tých v relačných databázach, čo vedie k tomu, že niektoré operácie sú rýchlejšie v relačných a iné v NoSQL databázach. Konkrétna vhodnosť použitej NoSQL databázy závisí od problému, ktorý chceme riešiť. Niekedy sú dátové štruktúry v NoSQL databázach považované za flexibilnejšie ako tabuľky relačných databáz.

**Apache Cassandra**

Je open source distribuovaný databázový systém navrhnutý pre správu veľkého množstva dát v rámci mnohých serverov, poskytujúci veľkú dostupnosť a spoľahlivosť. Dátový model Cassandry poskytuje vymoženosť stĺpcových indexov so silnou podporou denormalizácie a výkonným vstavaným kešovaním.

**Neo4j**

Súčasne najpoužívanejšia grafová databáza implementovaná v Jave. Vývojári opisujú Neo4j ako vstavaný, diskový, plne transakčný Java engine, ktorý ukladá dáta v grafoch namiesto tabuliek.

**Záver**

Na základe informácií, ktoré sme nadobudli počas analýzy sme sa v projekte rozhodli použiť tradičnú relačnú databázu PostgreSQL najmä z dôvodu veľkého rozšírenia a použiteľnosti a tiež z dôvodu podpory reprezentácie geografických údajov o objektoch.

## A.2 Analýza programových riešení

**Unity**

Unity3D je medzi platformový herný engine vyvinutý spoločnosťou Unity Technologies, ktorý slúži na vývoj videohier pre rôzne platformy, ako sú PC, konzoly, mobilné zariadenia, webové stránky. Unity sa zameriava na nasledujúce API:

- Direct3D v systéme Windows a konzolách Xobox 360
- OpenGL v systéme Windows a počítačoch MAC
- OpenGL ES pre operačný systém Andriod a iOS
- proprietárne API pre ostatné herné konzoly

Unity taktiež slúži na špecifikáciu kompresie textúr a rozlíšenia pre každú platformu, ktorú herný engine podporuje, pričom poskytuje podporu aj dump mapovanie, mapovanie reflexie, dynamice tiene s použitím tieňovej mapy a mnoho ďalšieho.

Skriptovanie v tomto hernom engine je postavené na Mono (voľný open sorce projekt, kompatibilný s .NET frameworkom, ktorý obsahuje rôzne nástroje vrátane C# kompilátora) a implementácia sa uskutočňuje na frameowku .NET. Programátori si môžu vybrať medzi UnityScript (syntax inšpirovaná JavaScriptom), C# alebo Boo (Python).

Samotné Unity bolo oznámené v roku 2005 a prvé spustenie sa konalo 8. Júna 2015, pričom prvé vydanie bolo obmedzené len pre OS X, kedy počas nasledujúcich verzií pribúdali aj ďalšie funkcionality a to hlavne spustenie pod operačným systémom Windows. Ďalšie verzie vznikali postupne a nadobúdali nové funkcie a možnosti, ako napríklad podporu pre platformy Andriod a iOS. Aktuálna verzia Unity je 5.0 ktorá bola spustená 3. Marca 2015 ako voľne dostupná. Nová verzia

priniesla veľké zmeny napríklad vo využívanej fyzike. Aktuálne Unity 5.0 podporuje veľké množstvo platforiem a to napríklad: Windows, OS X, Andriod, iOS, PlayStation, Xbox, Oculus Rift a mnoho ďalších.

**Unreal Engine**

Unreal Engine je herný engine vytvorený spoločnosťou Epic Games, ktorý bol predstavený už v roku 1998. Jeho zdrojový kód je napísaný v jazyku C++, čo využíva veľké množstvo vývojárov hier. Na začiatku bol tento engine navrhnutý iba pre určitý typ hier a grafiky (pohľad z prvej osoby) ale postupne sa začal využívať aj iných žánroch hier kde sa už nevyužíval pohľad z prvej osoby.

Unreal Engine je tiež multiplatformový a má podporu v operačnom systéme Windows, OS X, Linux, iOS, Andriod, pričom sa využívajú OpenGL, JavaScript a tiež WebGL pre potreby webových prehliadačov (HTML 5).

Aktuálne ma Unreal Engine 5 verzií. Prvá verzia s názvom Unreal Engine 1 bola predstavená verejnosti v roku 1998, ako prvá generácia. Neskôr k nej pribudli aj ďalšie a to Unreal Engine 2 v roku 2002 a Unreal Engine 3 v roku 2004, ktoré oproti prvej verzii už podporovali väčšie množstvo platforiem vrátane Xboxu a PlayStation. Aktuálne najnovšia verzia je Unreal Engine 5 na ktorom sa pracovalo pomerne dlhý čas ale verejnosť ho mohla vidieť až vo februári 2012, kde sú už podporované aj zariadenia s operačným systémom Android. Veľkou výhodou novej verzie je aj aktualizácia zdrojového kódu napísaného v C++ počas samotného behu aplikácie čo v predchádzajúcich verziách nebolo možné.

Keďže každý kto chce pracovať s Unreal Enginom si potrebuje zakúpiť licenciu, bola vyvinutá aj verzia s názvom Unreal Development Kit, ktorá je voľne dostupná a podporuje vytváranie hier a aplikácií aj pre operačný systém iOS.

**Ogre**

Ogre je 3D renderovací engine napísaný v jazyku C++, a je navrhnutý tak aby vývojári dokázali ľahšie písať programy, ktoré využívajú 3D grafiku. Má multiplatformovú podporu a v súčasnosti podporuje operačný systém Windows, Linux, OS X, iOS a Android. Keďže ide o multiplatformový engine môže sa používať spolu s OpenGL a Direct3D knižnicami čo zaručuje, že rovnaký obsah bude funkčný aj na iných platformách.

Hlavnou úlohou Ogre je poskytnúť vykresľovanie grafiky to znamená, že ide iba o vykresľovanie rôznych 3D scén. Ide o objektovo orientovaný dizajn kde sa využívajú hlavne rôzne pluginy čo umožňuje pridávanie ďalších funkcií a preto je veľmi modulárny. Ogre ako 3D renderovací engine neposkytuje zvukové záznamy a taktiež fyziku čo je veľkou nevýhodou.

**Záver**

Aj keď každý s analyzovaných 3D enginov má svoje výhody a nevýhody vybrali sme si Unity3D. Jeho hlavnou výhodou prečo sme si ho vybrali je možnosť vytvárať projekty v rôznych programovacích jazykoch ako je C# a Python, ktoré sú aktuálne veľmi populárne vývojové jazyky. Taktiež nám poskytuje všetky potrebné nástroje na vývoj a hlavne fyziku. Nenáročnosť na hardvérové prostriedky a veľká podpora zo strany komunity, ktorá vytvára projekty práve v Unity3D, sú pre nás tiež veľmi dôležité faktory.

## A.3 Analýza komunikácie UAV zariadení

**Bezdrôtové spojenie**

Je prenos informácií cez diaľku bez nutnosti použitia vodičov pre prepojenie komunikujúcich objektov. Vzdialenosť medzi komunikujúcimi objektami môže byť malá alebo veľká.

Bezdrôtový prenos dát môže ľahko a lacnejšie prekonať vzdialenosť prenášaných dát ako spojenie vytvorené pomocou vodičov. Pracovať s bezdrôtovými technológiami je už v dnešnej dobe pohodlnejšie ako využívať káblové technológie. Komunikácia v bezdrôtovej sieti je taktiež flexibilná a sieť dokáže byť rozdelená a izolovaná. Bezdrôtové technológie poskytujú možnosť použiť záložný komunikačný kanál pri zlyhaní hlavného kanálu.

Keďže sa UAV zariadenia pohybujú rýchlo a flexibilne, je nemožné využívať káblové technológie pre komunikáciu medzi týmito zariadeniami. Taktiež aj pri používaní bezdrôtových technológií treba brať ohľad na nadmorskú výšku a rýchlosť UAV zariadenia.

Existujú taktiež rôzne výhody používania bezdrôtových technológií pri UAV zariadeniach :

- UAV vie ponúknuť kvalitnú komunikáciu v dôsledku komunikácie line of sight,
- dynamické snímanie dát v regióne,
- UAV môžu modifikovať svoju dráhu letu pre zvýšenie kvality bezdrôtového signálu,
- zariadenia dokážu niesť obrovské množstvá dát.

**Faktory ovplyvňujúce komunikáciu medzi UAV**
Jedným z faktorov je šírka využívaného pásma. To sa odkazuje na rýchlosť prenášanej informácie, čo zaručuje prenos väčšieho množstva dát za kratšiu jednotku času.

Ďalším dôležitým faktorom je energia, ktorú UAV zariadenia potrebujú pre svoju letovú prevádzku. Energia pre let je obmedzená veľkosťou batérie. Rýchlosť prenosu dát je taktiež závislá na dostupnom množstve uchovanej energie.

**Požiadavky a špecifikácie**
Rozsah komunikácie pre mini UAV stroje je 10 km a nadmorská výška menšia ako 300 metrov. Batéria by mala vydržať aspoň 2 hodiny a UAV zariadenie by malo byť schopné uniesť 30 kg nákladu.

Prenosové dáta by mali byť v textovej forme ktoré by mali reprezentovať napr. letové príkazy. UAV zariadenia by mali komunikovať pomocou spätných potvrdzovacích správ. Maximálna rýchlosť UAV zariadenia by mala byť okolo 10 m/s. Je známe, že vzdialenosť medzi komunikujúcimi zariadeniami ovplyvňuje množstvo prenesených dát. Pre správnu komunikáciu zariadení by prenosová rýchlosť mala dosahovať aspoň 100 Kps.

**Bezdrôtové zariadenia a protokoly**
Existuje niekoľko komponentov, ktoré by mali byť zahrnuté pri komunikácii:

- Element ktorý prenáša správu,
- prijímacie zariadenie,
- prostredie v ktorom prebieha komunikácia,
- anténa.

Úlohou vysielača bude vysielať signál cez anténu. Rádiový vysielač kóduje dáta do RF vĺn a s určitým výkonom vysiela signál príjemcovi. Prijímač prijíma dáta a dekóduje ich. Prijímač vykoná prijatie signálu, potvrdenie a dekóduje navrhnuté RF vlny zatiaľ čo odmieta nechcené alebo redundantné dáta. Prostredie medzi UAV zariadeniami dokáže ovplyvniť komunikáciu zariadení. Elementy ako stromy, vietor, budovy dokážu výrazne ovplyvniť šírku prenášaných dát.

Pre správne nájdenie najlepšej cesty pre prenos signálu medzi dvoma zariadeniami sa využíva pojem "Freshnelova zona" . Podstata tejto metódy je dodržanie čistej komunikačnej cesty medzi UAV zariadeniami. Pre modifikovanie tejto cesty stačí meniť výšku antény buď na jednej strane komunikácie, alebo na oboch.

**IEEE 802.11 (WI-FI)**

IEEE 802.11 je sada štandardov pre WLAN komunikáciu vo frekvenčných pásmach 2.4 , 3.6 a 5 GHz.

Štandard 802.11 využíva rovnaký dátovo-linkovo-vrstvový protokol a rámcový formát ako originálny štandard. Kvôli veľkému využívaniu 2.4 GHz frekvenčnému pásmu je výhodou využívať protokol 802.11a, ktorý pracuje v pásme 5 GHz. Maximálny dátový prenos v tomto protokole je 54Mbit/s ale reálny tok je približne 20 Mbit/s.

Štandard 802.11b je využíva rovnaký prístup k médiám ako originálny štandard. Maximálny prenos hrubých dát je 11 Mbit/s. 802.11b zariadenia trpia rušením od iných zariadení pracujúcich na frekvencií 2.4 GHz.

Štandard označený 802.11g je tretím modulačným štandardom, ktorý taktiež pracuje vo frekvenčnom pásme 2.4 GHz a pracuje na fyzickej vrstve. Dátový prenos môže dosahovať rýchlosť 54 Mbit/s ale reálna rýchlosť opäť nepresahuje hranicu 22 Mbit/s.

802.11n je pozmeňujúci štandard pre 802.11 a pridáva viac vstupno-výstupných rámcových agregácií a pracuje s viacerými anténami.

**IEEE 802.11 (Bluetooth)**

Je štandardný komunikačný protokol, primárne dizajnovaný pre nízky odber energie, krátku prenosovú vzdialenosť. Keďže zariadenia využívajú broadcastovú komunikáciu, tak zariadenia nemusia byť medzi sebou v priamom bezprekážkovom priestore. Na základe dosahu signálu môžu byť tieto zariadenia rozdelené do troch

skupín.

- Prvá skupina má najväčší dosah v rozsahu niekoľkých stoviek metrov.
- Druhá skupina zariadení dokáže komunikovať v rozsahu niekoľkých desiatok metrov.
- Posledná skupina dokáže komunikovať len na niekoľko metrov.

Tieto zariadenia sa dajú ďalej rozdeliť podľa rýchlosti prenosu.

- Prvá generácia zariadení s označením 1.2 dokáže prenášať informácie iba v rýchlosťou 1 Mbit/s.
- Druhá generácia 2.0 + EDR zvýšila prenosovú rýchlosť na 3 Mbit/s.
- Ešte novšia generácia 3.0 + HS vie dosiahnúť prenosovú rýchlosť až 24 Mbit/s.

Výhody Bluetooth technológie:

- Je bezdrôtová technológia,
- spracováva dáta aj hlas,
- signály sú všesmerové a dokážu prechádzať stenami a inými prekážkami,
- využíva frekvenčné skákanie (Frequency hopping).

Bluetooth sa využíva pri:

- Prenose súborov,
- Ad-hoc networkingu,
- synchronizácii zariadení,
- prepojenií periférnych zariadení,
- mobilných platbách.

**IEEE 802.15.4(ZigBee)**

Je to bezdrôtová mesh sieťová norma, ktorá je nízko nákladová a energeticky šetrná. Nízke náklady umožňujú široké využitie v kontrolných a monitorovacích aplikáciách. Nízka spotreba energie umožňuje dlhšie využívanie služieb a tým pádom zabezpečuje dlhšiu životnosť batérie.

ZigBee je nasedený v rôznych odvetviach priemyslu. Jeho frekvenčný rozsah sa pohybuje v rozsahu 868 MHz - 2,4 GHz . Pri pásme 2,4 Ghz sa nachádza 16 kánalov a každý kanál vyžaduje 5 MHz frekvenčné pásmo.

Rýchlosť bezdrôtového prenosu je 250 kbit/s pre kánal pri 2.4 GHz, 40 kbit/s pre kánal pri 915 MHz a 20 kbit/s pri frekvencií 868 MHz.

Dosah zariadení tohto protokolu je v rozmedzí od 10m do 100m pri reálnom použití. Vo priestoroch budovy je dosah od 10-20m a pri použití vo vonkajších priestoroch bez bariér medzi dvoma zariadeniami môže byť až 1500m. Táto hodnota ale závisí od hodnoty dodanej energie batériou a charakteristiky vonkajšieho prostredia.

#### Porovnanie štandardov

Pri opisovaní jednotlivých protokolov v predchádzajúcich kapitolách, boli nájdené rozdielne vlastnosti týchto protokolov. Protokol IEEE 802.11b pracuje na frekvencií 2.4 GHz, ktorá je rovnaká aj pri Bluetooth a ZigBee. 802.11b ponúka vysokú prenosovú rýchlosť a jeho dosah je od niekoľkých desiatok metrov po niekoľko kilometrov.

Bluetooth na druhej strane má nižšiu prenosovú rýchlosť a jeho dosah vo vonkajších priestoroch je niekoľko stoviek metrov.

ZigBee má najnižšiu prenosovú rýchlosť a zložitosť z porovnávaných protokolov. Naopak zas ponúka najdlhšiu životnosť batérie.

Finančne, komplexnosťou a spotrebou energie najnáročnejšie je využívanie zariadení protokolu 802.11b. Tento protokol ale ponúka najrýchlejší dátový tok a dosah týchto zariadení je najväčší z porovnávaných typov.

Najlacnejší, najjednoduchší a energeticky najšetrnejšie sú zariadenia protokolu ZigBee. Tieto zariadenia ale majú najnižšiu rýchlosť prenosu a majú najmenší dosah.

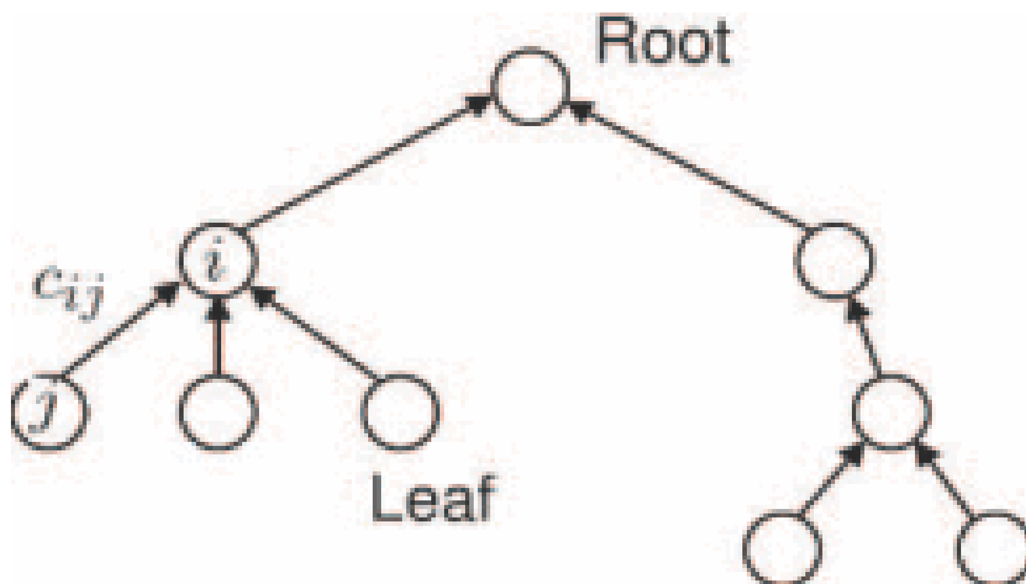
## A.4 Algoritmy pre plánovanie trajektórie UAV zariadení

V tejto kapitole budú opísané tri algoritmy pre plánovanie trajektórie UAV zariadení.

#### RRT algoritmus (Rapidly-exploring Random Tree)

Tento algoritmus je vytvorený pre efektívne a rýchle preskúmavanie veľkých plôch

v teréne, pričom tieto plochy nie sú konvenčne konvexné. Hlavnou myšlienkou RRT algoritmu je náhodný prieskum priestoru bez obmedzenia na ploche. Používa dáta prostredie pre vytvorenie stromu. Hrany v strome sú orientované z podriadených uzlov smerom k rodičovi. Každý uzol má jedného rodiča, okrem koreňa stromu. V tomto strome jednotlivé body predstavujú fyzické stavy a hrany označujú cestu medzi týmito stavmi . Jednotnosť je dosiahnutá tým, že sa náhodne vzorkuje z rovnomerného rozdelenia pravdepodobnosti.




Obr. A.1: RRT strom

Na obrázku A.1 sa nachádza hodnota cij, ktorá vyjadruje cenu cesty medzi dvoma bodmi. Počas narastania veľkosti stromu sa môže list strom stať rodičom.

Nevýhodou RRT je, že nevie nájsť optimálne riešenie pre nájdenie najkratšej a najefektívnejšej cesty. Z praktického pohľadu má tento algoritmus zopár výhod:

- Vie nájsť rýchle riešenie aj medzi zložitými prekážkami v teréne.
- Vie byť ľahko upraviteľný pre splnenie výpočtových obmedzení.
- Vie vyriešiť problémy v nekonvexnom prostredí.

Postup výpočtu RRT algoritmu :

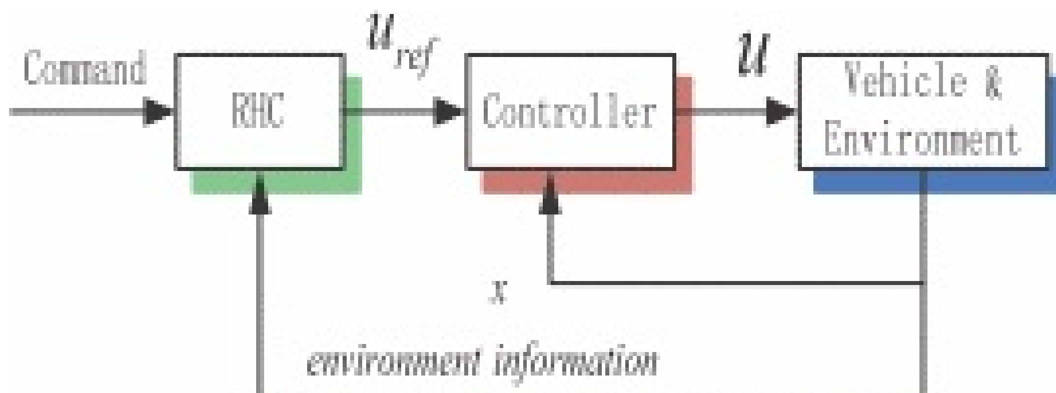
1. Náhodný bod je vybraný počas každej iterácie. Bod môže byť vybraný pomocou generátora náhodných čísiel.

2. Vybraný bod je porovnávaný oproti aktuálnemu stromu dosiahnuteľných bodov, vyberie sa bod, ktorý je najbližšie.
3. Ak sa nedá nájsť cesta zo zvoleného bodu, tak sa bod vymaže, inak sa pridá hrana dĺžky trasy od najbližšieho bodu k vygenerovanému.
4. Skontrolovať či nevznikla nová cesta. Inak opakovať 1. - 4. krok.

Algoritmus sa dá vylepšiť tak, že sa budú generovať dva stromy. Jeden strom sa generuje od štartu a druhý od cieľa.

**RHC algoritmus (Receding Horizon Control)**

RHC algoritmus je taktiež označovaný ako MPC (Model Predictive Control). Základnou myšlienkou je využitie modelu systému pre predpovedanie budúceho správania.




Obr. A.2: RHC algoritmus

Z obrázku A.2 je vidieť, že stavy a informácie prostredia budú spätná väzba pre RHC a ovládač , vďaka ktorým sa vytvoria kontrolné akcie pre uspokojenie rôznych obmedzení a optimalizáciu výkonu. Všetky neisté sú zahrnuté v informáciách prostredia.

Z matematického hľadiska sa dá vyjadriť problém, ktorý rieši algoritmus RHC. Pomocou X0 označíme stav v ktorom sa aktuálne UAV nachádza. Xfinal sa označuje cieľový stav.

$$x_{k+1} = Ax_k + Bu_k$$
$$y_k = Cx_k + Du_k$$
$$u_i \in \{u : u_{i\min} \le u_i \le u_{i\max}, i = 1, 2, \cdots\}$$
$$J = \sum_{i=0}^{N} h(x_i, u_i)$$

Výhody RHC:

- Jednoduchá formulácia na základe dobre pochopených konceptov.
- Explicitne spracováva obmedzenia.
- Explicitné použitie modelu.
- Dobre rozumie parametrom pre ladenie.
- Krátka doba vývoja.
- Ľahká úprava aj počas chodu systému.

Nevýhody RHC:

- Robustnosť,
- online výpočet je náročný,
- výsledok nemusí byť optimálny .

**Voronoiove diagramy**

Voronoivé diagramy sú vhodné pre scenáre, v ktorých prekážky sú relatívne malé a môžu byť modelované ako body .

Typickým znakom Voronoiového grafu je , že okraje grafe sú kolmé bisektory a sledovanie okrajov Voronoiových diagramov potenciálne vytvára cesty pre UAV zariadenia . Voronoi graf je vhodný pre plánovanie statických ciest, to znamená, že prekážky alebo hrozby sú známe vopred pred plánovaním letu . Plánovaná cesta pomocou Voronoiovho diagramu nie je úplne vhodný pre UAV zariadenia.

Klasický algoritmus využívajúci Voronoive diagramy:

1. Vstupné body prekážok $Q$ , začiatočný bod $P_s$ a konečný bod $P_e$

2. $(V, E) = constructVoronoiGraph(Q)$
3. $V^+ = V \cup \{P_s\} \cup \{P_e\}$
4. pre nájdenie $\{v_{1s}, v_{2s}, v_{3s}\}$, tri najbližšie body vo $V$ pre $P_s$, a $\{v_{1e}, v_{2e}, v_{3e}\}$, tri najbližšie body vo $V$ pre $P_e$
5. $E^+ = E \bigcup_{i=1,2,3}(v_{is}, P_s) \bigcup_{i=1,2,3}(v_{ie}, P_e)$
6. pre každý element $(v_a, v_b) \in E$ vykonaj:
7. Priradiť cenu hrán $J_{ab} = J(v_a, v_b)$
8. Ukonči for cyklus
9. $W = DijkstraSearch(V^+, E^+, J)$
10. Vráť $W$

# B Používateľská dokumentácia

Táto používateľská príručka je návodom popisujúcim ako spustiť program pre simuláciu pohybu roja UAV zariadení. Používatelia sa dajú rozdeliť na dve skupiny, a to klasický používatelia, spúšťajúci si program za tým účelom, aby sledovali pohyb roja UAV zariadení a používatelia, ktorí by chceli dopĺňať implementáciu tohto projektu.

Projekt simulácie pohybu roja UAV zariadení (dronov) bol implementovaný v týchto vývojových prostrediach:

- Microsoft Visual Studio 2013 s podporou .NET 4,5 – implementovaná celková logika programu (správanie sronov, poveternostné podmienky...)
- Unity 3D 5.3.4 - implementácia, vyobrazenie prostredia (sveta), v ktorom sa UAV zariadenia pohybujú

Preto používatelia, ktorí chcú meniť implementáciu, prípadne spúšťať program priamo z vývojového prostredia, musia použiť verzie vyššie spomenutých vývojových nástrojov, prípadne vyššie. Vyššie spomenuté nástroje teda predstavujú minimálne požiadavky pre možnosť implementácie zmien do projektu. Program je spustiteľný na strojoch bežiacich na Operačnom systéme Windows.

## B.1 Spustenie programu

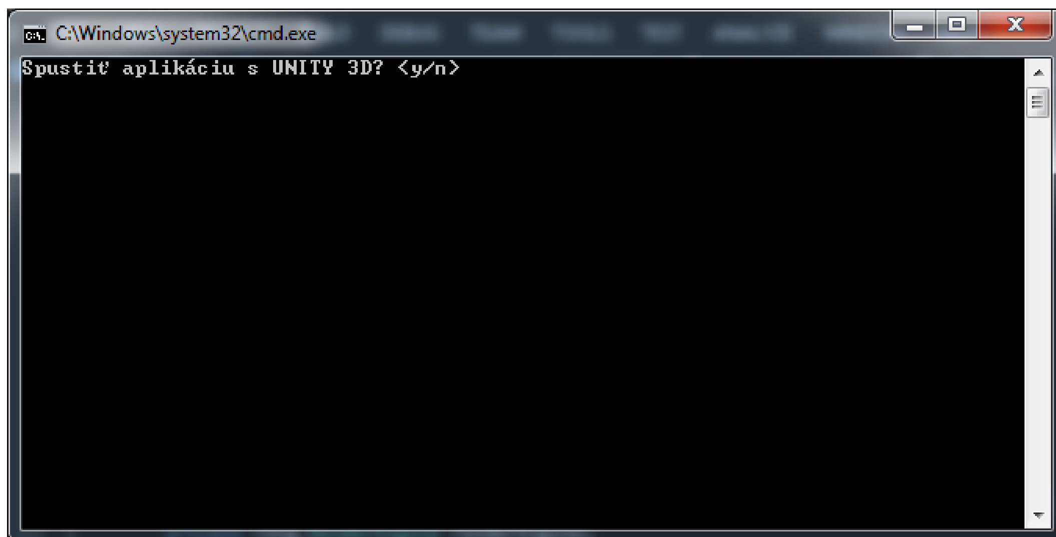
Program je možné spustiť dvomi spôsobmi:

- otvorením spustiteľného *.exe* súboru,
- spustením priamo v Microsoft Visual Studio pomocou klávesovej skratky *F5*.

Po spustení programu sa používateľovi zobrazí úvodná konzolová obrazovka, v ktorej si používateľ vyberie, ako chce program spustiť. Či ho chce spustiť iba ako konzolovú aplikáciu, kde pomocou LOG výpisov môže sledovať správanie sa, stav, polohu UAV zariadení alebo sa program spustí cez Unity 3D a používateľ má možnosť aj obrazovo vidieť správanie sa UAV zariadení a takisto má možnosť vidieť, v akom prostredí sa UAV zariadenia pohybujú.

Po spustení programu sa zobrazí výpis: *Spustiť aplikáciu s UNITY 3D? <y/n>*

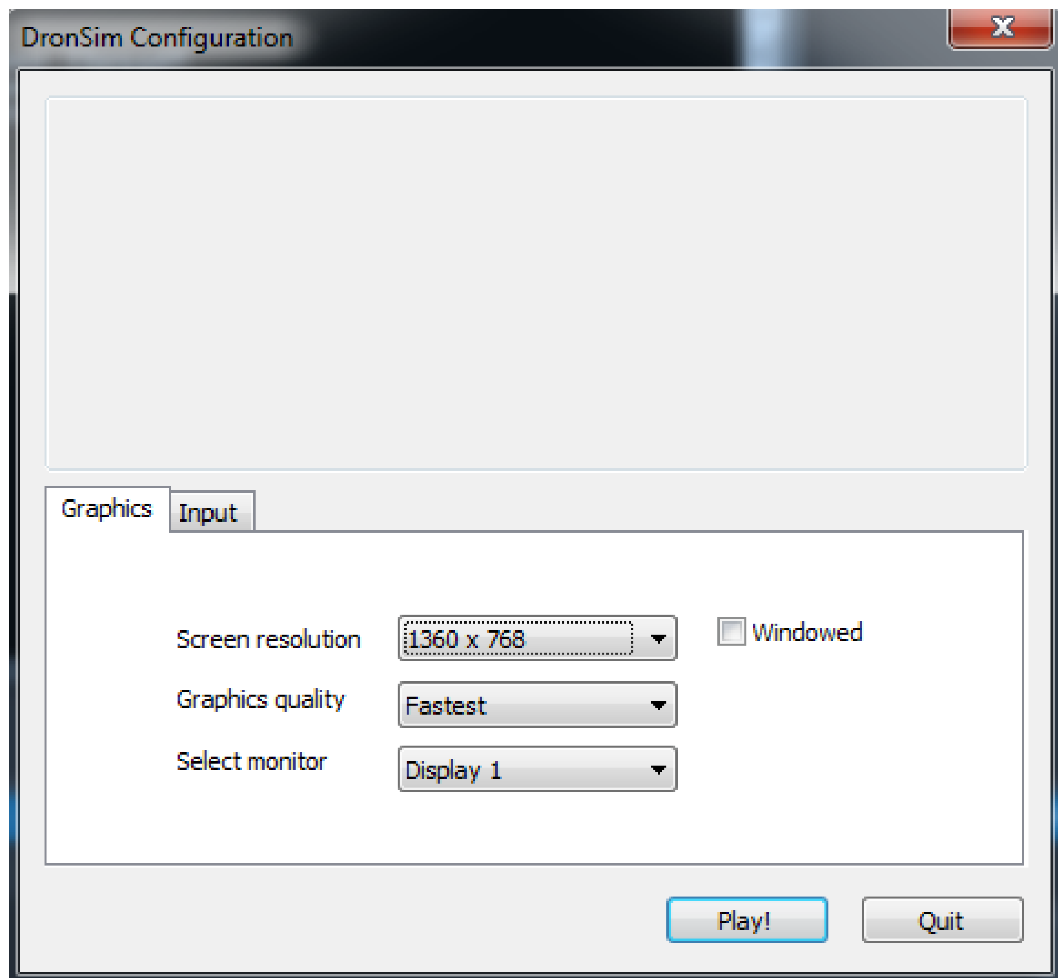
Stlačením „y + enter“ sa aplikácia spustí s Unity 3D, stlačením voľby „n + enter“ sa spustí iba konzolová aplikácia.




Obr. B.1: Úvodná obrazovka.

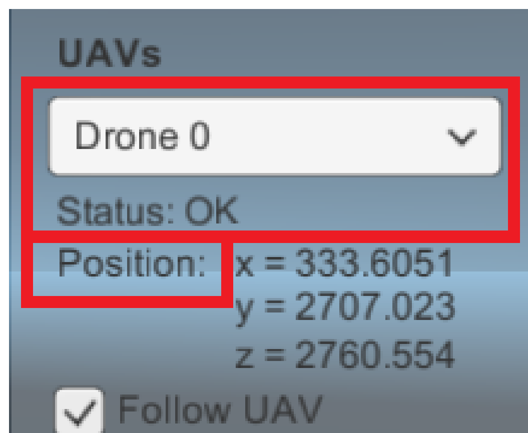
Vrámci projektového adresára sa nachádza podadresár *Logs*, kde sa pri každom spustení vytvorí *.log* súbor, v ktorom sú ukladané informácie z komunikácie medzi UAV zariadeniami a najmä pri spustení programu ako konzolovej aplikácie si používateľ môže pozrieť loggovacie výpisy, ktoré popisujú správanie UAV zariadení počas simulácie. Tieto súbory sú vytvárané aj pri spustení programu spolu s Unity 3D, a tak má používateľ po ukončení simulácie aj vtedy možnosť prezretia si loggovacích výpisov, kde detailnejšie vidí aj komunikáciu medzi UAV zariadeniami a informácie o ich stave (kedy havarovali, ak havarovali a podobne).

Pri spustení aplikácie s Unity 3D sa spustí program v Unity 3D, kde si používateľ najskôr zadefinuje základné vlastnosti pre zobrazenie simulácie, ako napríklad rozlíšenie okna, kvalitu vykreslovania a podobne.



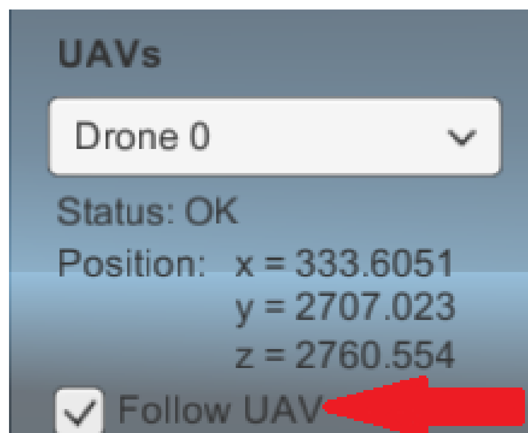

Obr. B.2: Nastavenie základných vlastnosti pri spustení s Unity 3D.

Po stlačení tlačidla *„Play!“* sa spustí vykreslovanie prostredia (sveta) spolu s pohybujúcimi sa UAV zariadeniami. Tu má používateľ možnosť pomocou myšky a šípiek, pohybovať sa po celej „mape“ a tak sledovať simuláciu pohybu UAV zariadení. Používateľ si môže vybrať zo zoznamu UAV zariadení (umiestnenie vpravo hore) a sledovať informácie o ich základných údajoch, ako sú poloha, stav UAV zariadenia (obr. B.3).




Obr. B.3: Výber konkrétneho UAV zariadenia zo zoznamu.

Používateľ má taktiež možnosť, okrem všeobecného sledovania simulácie pohybu UAV zariadení, aj sledovať vrámci tejto simulácie konkrétne UAV zariadenie, ktoré si vyberie zo spomínaného zoznamu, musí však zakliknúť tlačidlo *„Follow UAV"* (obr. B.4). Takto získa aj obrazovo detailný pohľad na pozíciu konkrétneho UAV zariadenia vrámci simulácie.




Obr. B.4: Sledovanie konkrétneho UAV zariadenia.

## B.2 Ukončenie programu

Ak program bežal ako konzolová aplikácia, jeho ukončenie používateľ vykoná stlačením *„ctrl + c"* a následným zatvorením konzolového okna (možné ukončenie aj priamym zatvorením okna).

Ak bol program spustený spolu s Unity 3D, tak používateľ ukončí program zatvorením programu Unity 3D, v prípade potreby je taktiež možné ukončenie z príkazového riadku v konzole.

# C Technická dokumentácia

## C.1 Prepojenie logiky simulátora a vykresľovania simulácie

Na tento typ komunikácie bol zvolený protokol UDP. Aj keď ide o tzv. nespoľahlivý protokol, pretože príchod dát nie je potvrdzovaný a tým pádom nie je zaručené ich správne doručenie, na druhej strane je vďaka tomu časovo rýchlejší a efektívnejší.

Pre umožnenie tejto komunikácie beží v oboch aplikáciach samostatné vlákno - tzv. UDP server. V c# aplikácii simulátora je zastrešené pod triedou *Server.cs* (`Thread sampleUdpThread`) a prostredníctvom UDP klienta sú správy prijímané (`UdpClient listener`) vo fukncii `StartReceiveFrom()` - ukážka 3.

Na strane Unity beží vlákno (`ActionThread myThread`) takmer identické ako v c# a taktiež využíva na príjem UDP správ UDP klienta (`UdpClient client`). Vlákno vykonáva svoju činnosť vo funkcii `DoThreadWork()` - ukážka 4. Na začiatku bolo dosť problematické komunikovať z tohto vlákna s hlavným vláknom Unity. Podarilo sa to použitím hotového projektu UnityThreading[1], čo umožňuje zavolaním `UnityThreadHelper.Dispatcher.Dispatch(() => ( ...)` vykonať operáciu nad objektami hlavného Unity vlákna.

Unity projekt sa skladá zo c# skriptov. Delené sú do viacerých častí:

- Data - skript *Enviroment.cs* - pomocou SimpleJSON sú prijaté JSON-y rozobrané do dátových štruktúr - listov pre drony, objekty v mape a mapu. Na základe informácií z nich sú objekty inicializované do scény.
- Init - obsahuje skripty, ktoré sú inicializované hneď pri štarte - nazvané sú ako kontrolery - *CammeraController.cs*, *DropDownController.cs*, *FPSDisplay.cs*, *NetworkController.cs* a *Send.cs.*
- Render - skript *Render.cs* sa stará o vykresľovanie objektov v scéne na základe údajov z *Enviroment.cs.*
- SimpleJSON[2]a UnityThreading.

[1]`http://forum.unity3d.com/threads/unity-threading-helper.90128/`

*CammeraController.cs* - slúži na ovládanie kamery, kamerou je možné pohybovať pocmou šípok na klávesnici alebo otáčať pomocu ľavého kliku myšky. Zároveň je implementovná funkcionalita *Follow* - nasledovanie označeného drona.

*DropDownController.cs* - slúži na výber drona - zobrazia sa informácie o pozícii a stave a zároveň je možné aktivovať funkciu *Follow*. Pred inicializáciou objektov z JSON súboru je zoznam prázdny.

*FPSDisplay.cs* - zobrazuje obrazovú frekvenciu na používateľskom rozhraní - počet zobrazených obrázkov za sekundu.

*NetworkController.cs* - reprezentuje UDP server na prijímanie správ.

*Send.cs* - pomocou skriptu sa odosielajú UDP správy do simulátora.

*Enviroment.cs* - dátové štruktúry objektov.
Funkciou `init_json_parse(string json_string)` sa rozoberie inicializačná JSON správa.
Funkcia `update_json_parse(string json_string)` slúži na analýzu aktualizovacích JSON správ, v ktorých sa už nachádzajú len pozície a stavy dronov. Ak sa pri aktualizácii zistí stav drona „DESTROYED“, pomocou funkcie Unity `OnGUI()` sa vypíše informácia o jeho zlyhaní do používateľského rozhrania pod FPS.

*Render.cs* - funkcia `init_render()` je volaná po rozobratí inicializačného JSON-u a v nej sa vykoná inicializácia objektov do scény, ná základe údajov z *Enviroment.cs*. Funkcia `Update()` (ukážka 5) sa vykonáva tak často aké je FPS. V nej sa prechádza list dronov z *Enviroment.cs* a objektom sa priraďujú na základe neho súradnice pozície.

[2] `http://wiki.unity3d.com/index.php/SimpleJSON`

```
public void StartReceiveFrom()
{
    UdpClient listener = new UdpClient(port);
    plistener = listener;
    IPEndPoint groupEP = new IPEndPoint(IPAddress.Any, port);
    Console.WriteLine(">> Waitting from Unity connection on port: " + »
       » port);

    try
    {
        while (true)
        {
            byte[] bytes = listener.Receive(ref groupEP);
            byte[] message = bytes;
            string sprava = Encoding.ASCII.GetString(message, 0, message.»
               » Length);

            if (connected == true)
            {
                if (string.Compare(sprava, "INIT_RE_OK") == 0)
                {
                    jsonCreator.send_update(UDP_client);
                }
            }
            else
            if (connected == false && string.Compare(sprava, "»
               » unity_connection") == 0)
            {
                UDP_client.IP = groupEP.Address.ToString();
                UDP_client.port = groupEP.Port;
                UDP_client.SendCmd("OK");
                connected = true;
            }
        }
    }
    catch (SocketException se)
    {
        Console.WriteLine("A Socket Exception has occurred!" + se.»
           » ToString());
    }
```

Ukážka 3: Funkcia UDP servera na príjem správ.

```
void DoThreadWork()
{
    while (true && quit == false)
    {
        counter++;
        // receive bytes
        anyIP = new IPEndPoint(IPAddress.Any, 0);
        data = client.Receive(ref anyIP);

        text = Encoding.UTF8.GetString(data);

        Debug.Log(">>_" + text);

        if (text == "OK")
        {
            connected = true;
        }

        if (text == "INIT_OK")
        {
            init = true;
            render.init = true;
        }

        if (text.Trim().StartsWith("{") && init == false)
        {
            env.init_json_parse(text);
            Debug.Log("init_RE_OK");
            render.init_render();
            sender.send_cmd("INIT_RE_OK");
        }
        else
            if (text.Trim().StartsWith("{") && init == true)
            {
                env.update_json_parse(text);
            }
    }
}
```

Ukážka 4: Funkcia UDP servera Unity na príjem správ.

```
// Update is called once per frame
void Update () {
    if (init == true)
    {
        for (int i = 0; i < env.drone_param.Count; i++)
        {
            index = uavs.FindIndex(item => item.id == env.drone_param[i].id);
            uavs[index].uav.transform.position = new Vector3(env.drone_param[»
               » index].x, env.drone_param[index].y, env.drone_param[index].z»
               » );
        }
    }
}
```

Ukážka 5: Funkcia Update v Render.cs na zmenu pozície dronov.

## C.2 Opis skriptu správania sa hlavného drona

Master skript opisuje správanie hlavného drona, ktorý odosiela správy o svojej polohe podriadeným dronom. Podradení droni sa snažia nasledovať master drona podľa prijatých správ. Poloha dronov je získávaná z gps providera, ktorý sa nachádza v enviroment časti aplikácie. Master dron má preddefinované vektory po ktorých sa pohybuje. Po týchto vektoroch sa pohybuje počas nastavenej časovej dĺžky.

Konštruktor `__init_` sa stará o inicializáciu skriptu. Vstupnými parametrami tejto funkcie sú atribúty potrebné na využívanie gps providera, ostatných funkcií *enviroment* časti aplikácie a obsahuje taktiež atribúty definujúce vlastnosti drona. Atribúty drona boli získané z *enviroment.json* súboru, ktorý definuje vstupné parametre dronov a objektov. V metóde konštruktora sa nastavuje aj pole vektorov, ktorými sa master dron bude riadiť. V tomto konštruktore sa získavá aj počiatočná gps poloha od *gps providera*.

Vo funkcii `run` sa spustí nekonečná slučka, ktorá sa stará o kontinuálny posun drona po nastavených vektoroch a čase. V tomto cykle sa odosielajú aj správy podriadeným dronom o aktuálnej polohe master drona.

Funkcia `sendMessage()` sa stará o odosielanie správ iným dronom.

Funkcia `onReceiveMessage` spracuje prijatú správu a podľa jej typu ju ďalej pošle na spracovanie.

`Calucalate()` je možné využiť ak chceme vypočítať nasledujúcu pozíciu drona. Táto funkcia sa momentálne nepoužíva, pretože sa využíva vektorový posun drona a nie posun podľa súradníc.

```
def __init__(self, id, gpsProvider, communicationProvider, queue, objectLock,»
  »  enviroment, battery, consumption, speed) :
    ...

def run(self) : # hlavna funkcia uav - spusta sa s threadom v c#
    # odoslanie spravy vsetkym dronom v liste rec_list, typ komunikacia, telo»
      »  spravy je prazdne (testovanie),
    # 1 je sposob komunikacie. mame spravene zatial len wifi, prakticky sa »
      » nikde toto cislo zatial nezohladnuje
    while self.Thread : # nekonecny cyklus zivota drona
    ...
    ..
    .

def sendMessage(self, receiversId, msgType, msgBody, communicationType) :
    self.communicationProvider.SendMessage(self.id, receiversId, msgType, »
      » msgBody, communicationType)

def onReceiveMessage(self, msg) :
    if msg.Type == MessageType.TYPE_SIMULATION:
        self.handleSimulationMessage(msg)
    else :
        self.handleCommunicationMessage(msg)

def calculate(self):
    diffTime=int(time.time() * 1000)-self.time
    self.time = int(time.time() * 1000)
    stepLength = (diffTime/1000) * self.speed
    for pos in ['X', 'Y', 'Z']:
        pos_value = getattr(self.currentPosition, pos)
        if pos_value > self.positions[self.index][pos]:
            if pos_value - stepLength <= self.positions[self.index][pos]:
                setattr(self.currentPosition, pos, self.positions[self.index»
                  » ][pos])
            else:
                setattr(self.currentPosition, pos, pos_value - stepLength)
        else:
            if pos_value + stepLength >= self.positions[self.index][pos]:
                setattr(self.currentPosition, pos, self.positions[self.index»
                  » ][pos])
            else:
                setattr(self.currentPosition, pos, pos_value + stepLength)
    current_position = {'X': self.currentPosition.X, 'Y': self.»
      » currentPosition.Y, 'Z': self.currentPosition.Z}
    if current_position == self.positions[self.index] and self.index+1 != len»
      » (self.positions):
        self.index += 1
    self.battery -= self.consumption
```

Ukážka 6: Ukážka funkcií zo skriptu správania sa hlavného drona.