Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matej Hrlec

Brezžični protokol in podpora za navidezno resničnost za upravljanje brezpilotnih zrakoplovov

magistrsko delo
študijski program druge stopnje
računalništvo in informatika

Mentor: asst. prof. Friedrich Fraundorfer, PhD
Somentor: izr. prof. dr. Matija Marolt

Ljubljana, 2019

University of Ljubljana
Faculty of Computer and Information Science

Matej Hrlec

Wireless protocol and virtual reality support for controlling unmanned aerial vehicles

MASTER'S THESIS

THE 2ND CYCLE MASTER'S STUDY PROGRAMME
COMPUTER AND INFORMATION SCIENCE

Supervisor: Asst. Prof. Friedrich Fraundorfer, PhD

Co-supervisor: Assoc. Prof. dr. Matija Marolt

Ljubljana, 2019

[1]V dogovorju z mentorjem lahko kandidat magistrsko delo s pripadajočo izvorno kodo izda tudi pod drugo licenco, ki ponuja določen del pravic vsem: npr. Creative Commons, GNU GPL. V tem primeru na to mesto vstavite opis licence, na primer tekst [8].

I would start by extending my thanks to Assist. Prof. Dr. Friedrich Fraundorfer for being my mentor during my stay in Graz, providing a topic to work on and finding time to discuss ideas for the thesis. This would be followed by thanks to Assoc. Prof. Dr. Matija Marolt, who not only mentored me for my Bachelor's Thesis, but again did so for my Master's and also helped me with the registration process.

First, I would thank Dr. Jesús Pestana Puerta for helping me get thorough the hurdles of setting up ROS. Through his tutelage I learned a lot about drone development and had quite a programming exercise with concurrent work on Android while at the same time battling time to get our test flight going.

Second, I am grateful to MSc (soon-to-be Dr.) Werner Alexander Isop for instructions on drone development during Camera Drones lecture and later bringing me up on his project of meshing together VR and drones with a plethora of innovative ideas, which I am sure will be soon concluded with his PhD.

Since I spent much time there, I would like to express gratitude to the Institute of Computer Graphics and Vision for providing me a nice and cosy place to conduct my work and commend all the staff there for their friendliness.

Special thanks to my faraway friend Dartimos, who proof-read the document for nothing but gratitude in return.

My families contributions for bringing me up to this point and being by my side all throughout my life had also an unquantifiable beneficial impact that could never be repaid. And last, I am thankful for my partner Manuela's support and love through more or less all of our joint (now finishing) student life.

Contents

List of used acronyms

acronym	**meaning**
API	Application Programming Interface
AOI	Area of Interest
GCS	Ground Control System
GPS	Global Positioning System
GUI	Graphical User Interface
HMD	Head-Mounted Display
IDE	Integrated Development Environment
RC	Remote Controller
ROS	Robot Operation System
SfM	Structure from Motion
UAV	Unmanned Aerial Vehicle
VR	Virtual Reality

Povzetek

Naslov: Brezžični protokol in podpora za navidezno resničnost za upravljanje brezpilotnih zrakoplovov

Naraščajoča priljubljenost potrošniških brezpilotnih letalnikov je posledica bistvenega napredka pri avtonomnosti avtopilotov in enostavnosti uporabe. V našem magistrskem delu smo zato želeli razviti nova orodja za brezpilotnike, ki bi lahko bile v pomoč pri prihodnjih raziskavah.

Zagotovili smo robustno in prilagodljivo rešitev za komunikacijo, ki temelji na protokolu MAVLink. Telekomunikacijska povezava je vzpostavljena preko daljinskega upravljalnika med DJI brezpilotnikom, računalnikom in tabličnim računalnikom Android. Doseg in zanesljivost naše rešitve smo preizkusili v simuliranem in zunanjem okolju, kjer se je izkazalo, da je sposobna krmiliti brezpilotnika pri letu. Prikazali smo tudi koncept za integracijo z obstoječo zemeljsko kontrolno postajo. Da bi omogočili dodatne metode vizualizacije in manipulacije, smo razvili popolnoma integrirano orodje za ROS, ki temelji na OpenVR API-ju. Slednji zagotavlja vmesnike za uporabo večine očal za navidezno resničnost in krmilnikov preko vizualizatorja RViz. Hkrati smo preučili uporabo omenjenega orodja pri prilagodljivem upravljanju pogleda v raziskovalnem scenariju, kjer je operaterju letalnika zagotovljen pregled nad interesnim območjem preko sekundarne navidezne kamere ali letalnika.

Ključne besede: brezpilotnik, letalnik, zrakoplov, DJI, ROS, MAVLink, Android, C++, pluginlib, avtonomni let, navidezna resničnost, RViz.

Abstract

Title: Wireless protocol and virtual reality support for controlling unmanned aerial vehicles

The growing popularity and affordability of consumer drones is driven by continuous advances on ease of use and autonomy. In our work we aspired to provide more development options for UAVs, by looking into tools to help with future research.

We provided a robust and scalable solution for communication based on MAVLink protocol between DJI drones, its onboard computer, and an Android tablet computer, where the telecommunication link is established over the remote controller. We tested the range and reliability of our framework in simulated and real-life environments where it proved fully capable of steering a drone on flight missions. We also did a proof of concept integration with an existing ground control station. To allow for additional methods of visualization and manipulation, we also developed a fully ROS-integrated tool based on the OpenVR API that provides support for most VR headsets and controllers through RViz visualizer. We furthermore looked into using that tool for adaptive view management in an exploration scenario, where the drone operator is provided an overview of the area of interest through a secondary virtual camera or drone.

Keywords: drone, unmanned aerial vehicle, DJI, ROS, MAVLink, Android, C++, pluginlib, autonomous flight, virtual reality, RViz.

Razširjeni povzetek

I Uvod

S pojavom cenovno ugodnih brezpilotnikov je začela njihova prodaja strmo rasti. Medtem ko so jih včasih uporabljali predvsem v vojski, za letalsko fotografijo ali kot majhne igrače za zabavo, sodobni civilni letalniki izpolnjujejo najrazličnejše naloge tako za osebne kot tudi za komercialne namene. Uporabljajo se lahko npr. za celovit pregled industrijskih objektov, fotografiranje in snemanje posebnih dogodkov, kot npr. porok in odprtja olimpijskih iger, v kmetijstvu za pastirstvo in pregledovanje pridelkov, v iskalnih in reševalnih misijah na težko dostopnih krajih, za pomoč pri prodaji nepremičnin s kakovostnimi slikami nepremičnin, za dostavo manjših paketov in pregledovanje velikih površin.

Največji izdelovalec brezpilotnikov je DJI, ki za svoje produkte razvija lastno programsko opremo. API-ji za letalnike zagotavljajo hitre in učinkovite klice funkcij, na primer za prenos videa, osnovne podatke o letu ali daljinsko krmiljenje. Težava izhaja iz zelo omejenih možnosti ustvarjanja aplikacij po meri in prenosa poljubnih podatkov. Trenutno uporabljena rešitev za zagon programov in sprejemanje podatkov je uporaba Wi-Fi povezave med zemeljsko nadzorno postajo in računalnikom, vgrajenim v brezpilotnik. Slabost tovrstne povezave so omejitve v dosegu komunikacije.

V tem stoletju je kot posledica preboja računalniške tehnologije, zlasti majhnih in močnih mobilnih tehnologij, prišlo do velikega napredka v razvoju navidezne resničnosti (NR) [27]. Zagon v veliki meri sloni na razvoju pametnih telefonov z visokoločljivimi zasloni in zmogljivo podporo za 3D grafiko, obenem pa je zniževanje cen njihovih strojnih komponent omogočilo razvoj nove generacije lahkih NR naprav. Industrija video iger je razvoj potrošniške NR še dodatno pospešila. Krmilniki gibanja, prepoznavanje potez, senzorji kamer v globini in naravni človeški vmesniki so že del trenutne izbire potrošnikov. Večina cenovno dostopnih očal se ponaša z visokoločljivimi zasloni, sledenjem do 6 stopinj svobode in običajno tudi možnostjo uporabe ročnih krmilnikov.

V prvem delu magistrskega dela se osredotočamo na zagotavljanje robustne in razširljive rešitve za komunikacijo med brezpilotnim letalom DJI M100 in tabličnim računalnikom z operacijskim sistemom Android, ki je priložen daljinskemu krmilniku drona (RC). Telekomunikacijska povezava je dosežena prek RC-ja. Ta preskusna povezava je v skupni rabi z običajno komunikacijsko povezavo RC to drone, le majhna pasovna širina je rezervirana za komunikacijsko povezavo tabličnega računalnika. Na vgrajenem računalniku drona je medprocesna komunikacija dosežena s programsko opremo Robot Operating System (ROS).

Ker je pasovna širina komunikacijske povezave med tablico in dronom zelo omejena, mora biti protokol sporočanja lahek in jedrnat, kar nas je pripeljalo do izbire komunikacijskega protokola Micro Air Vehicle Link (MAVLink). Določili smo arhitekturo programske opreme, nov sklop sporočil, podobnih ROS-u, in protokol za prenos datotek, ki nam nudi okvir za doseganje učinkovite in prilagodljive komunikacije med brezpilotnimi letali serije DJI M in pametnim vmesnikom App. Z zasnovo ogrodja, ki uporablja MAVLink, zagotovimo učinkovito uporabo zelo omejene pasovne širine brezžične povezave in omogočimo daljinsko upravljanje brezpilotnika brez uporabe stacionarnih zemeljskih nadzornih postaj, kot so namizni računalniki.

Izvedli smo nabor eksperimentalnih preizkusov letenja v zunanjem okolju, s katerim smo testirali zanesljivost sistema, in niz preskusov za merjenje pretočnosti prenosa. Nadalje smo si ogledali učinke razdalje na hitrost prenosa, pasovno širino, latenco in doseg.

Odločili smo se za uporabo tako očal kot krmilnikov in jih vključili v rešitev za letenje z brezpilotnimi letali z NR. Za prikaz prizorišča pilotu smo uporabili eno najbolj priljubljenih naprav NR v času pisanja, HTC Vive s krmilniki Vive, ki se uporabljajo za manipulacijo, in bazne postaje Vive, ki omogočajo sledenje tem komponentam. Osnovno ogrodje je ROS Indigo za Ubuntu 14.04, pri čemer paketi med seboj komunicirajo prek ROS sporočil. Navidezna scena, ki smo jo uporabili pri testiranju, je narejena v Googlovi programski opremi za 3D modeliranje Sketchup. Fizikalna simulacija zrakoplova Parrot Bebop 2 in sveta je bila narejena v RotorS, ki je MAVLinkov simulator za Gazebo. Vizualizacija je bila izvedena v RVizu, ki je namenski 3D vizualizator za ROS. 3D prizor se je nato uporabil v vtičniku, kjer se z gonilnikom OGRE 3D izrišejo ločeni pogledi za zaslona na napravi Vive. Ti pogledi se pošljejo v kompozitor, ki ga zagotovita OpenVR in SteamVR.

II Pregled podobnih del

Ker so mobilne naprave v nasprotju z računalniki prenosljive, so raziskovalci [7] zasnovali zemeljsko nadzorno postajo (angl. ground control station - GCS) na operacijskemu sistemu Android. Z UAV-jem komunicira s pomočjo protokola TCP/IP, preko katerega se periodično pošiljajo paketi s podatki o koordinatah GPS, pospeških in hitrosti. Pomanjkljivosti so, da se UAV še vedno upravlja s standardnim daljinskim upravljalnikom ter da ni prenosa slike in avtonomnih zmogljivosti.

V raziskavi [3] so naredili pregled različnih možnih arhitektur za komunikacijo med posameznimi ali skupinami brezpilotnikov ter njihovo uporabo za vodenje podvodnih brezpilotnih vozil. Obravnavali so jih znotraj domene letečih ad-hoc omrežij (FANET), ki so ob nizki hitrosti primerna za uporabo v klasičnih mobilnih omrežjih. Poleg tega predlagajo še uporabo omrežij na zahtevo in lokacijske protokole.

Nedavno je bil objavljen podoben prispevek [21] o integraciji nadzorne postaje QGroundControl z brezpilotniki DJI z uporabo funkcij, ki jih ponuja

SDK. Ogrodje prestreže ukaze, poslane iz GCS-ja na dron, in pokliče ustrezne funkcije iz SDK-ja. Slabost je, da je QGroundControl omejen le na nabor funkcij, ki jih ponuja DJI, in ne omogoča pošiljanja poljubnih podatkov preko kanala Data Transparent Transmission. Tudi povezava je omogočena le preko TCP/IP.

Integracijo sistema ROS in 3D pogonov za igre so raziskali za uporabo v Unity [5] in Unreal Engine [17]. Skupna jima je uporaba spletnih vtičnic za povezavo med odjemalcem in strežnikom, kjer se 3D objekti prenašajo preko predpisanih tipov. Sicer pa je uporaba informacij in ukazov, ki bi jih lahko VR oprema posredovala ostalim ROS vozliščem, otežena, saj vtičnika ne omogočata direktne integracije s sistemom za izmenjavo sporočil, ki je ključni del ROS-a. Njuna prednost je v možnosti uporabe vseh naprednih senčilnikov, vizualizacijskih tehnik in ostalih orodij, ki jih ta dva moderna pogona za igre omogočata. Vizualizacijsko orodje RViz [11] je po drugi strani grafično manj dovršeno, vendar je že integrirano v ekosistem ROS in posledično bolj dostopno razvijalcem, ki so ga navajeni. Vizualizator uporablja grafični pogon OGRE, ki skozi abstrakcijo omogoča uporabo mnogih sistemskih knjižnic, npr. OpenGL, WebGL in Direct3D.

III MAVLink podpora za DJI letalnike

III.I Možnosti povezav

Razvili smo dva načina za prenos sporočil MAVLink med vgrajenim računalnikom in sistemom GCS, in sicer preko daljinskega upravljalnika ali Wi-Fi-ja. Osredotočili smo se na prvega, saj je bolj uporaben v primerih, ko je potrebna stabilna povezava na dolgih razdaljah in žrtvovanje pasovne širine ni težava, npr. pošiljanje meritev in podatkov o letu. Mobile SDK ponuja tudi ločen kanal za sprejemanje video prenosa iz kamere. S tem in z možnostjo prenosa paketov podatkov po meri s sporočili MAVLink je okvir uporaben v večini primerov, zlasti za lete z daljšim dosegom v zunanjih okoljih.

Kadar je potrebna večja pasovna širina, na primer pri prenosu velikih paketov podatkov za obdelavo na zemeljski nadzorni postaji, se lahko komunikacija vzpostavi tudi s TCP/IP, najpogosteje prek povezave Wi-Fi. To je mogoče storiti z usmerjevalnikom kot posrednikom med mobilno napravo in brezpilotnikom, lahko pa jih neposredno povežemo tudi z mobilno dostopno točko. V obeh primerih letalnik potrebuje modul za Wi-Fi. Ta način je uporaben predvsem za lete v zaprtih prostorih, kjer je za povečanje dosega v več prostorih mogoče uporabiti številne usmerjevalnike z istim imenom omrežja (SSID).

III.II Ustvarjanje MAVLink narečja

Ustvarili smo nov nabor definicij sporočil, imenovan MAVLink narečje. Sporočila so po strukturi podobna sporočilom ROS, ki jih bomo potrebovali za izvajanje misije leta, in uporabljajo časovne žige ROS, ki so pomembni za sinhronizacijo izvajanja tem. Omogočajo nam pošiljanje ukazov za spremembo položaja in naklona, preklapljanje med različnimi načini leta in prenos podatkov o letu na mobilno napravo. Ustvarili smo tudi protokol za prenos datotek, ki zagotavlja mehanizme za nadomestitev, s katerimi poskušamo zagotoviti garancijo za prenos. Upoštevati moramo tudi razliko med paketi in sporočili v našem kontekstu. Velikosti paketov so omejene na 100 B zaradi DJI jevega protokola za prenos podatkov Data Transparent Transmission, a ker so sporočila MAVLink običajno večja, so v tem primeru razdeljena na več paketov in poslana po kosih. Zato moramo imeti v mislih, da lahko pri ponovnem sestavljanju na strani prejemnika dobimo pokvarjene/neurejene pakete različnih sporočil.

Za ustvarjanje izvorne kode za knjižnico MAVLink smo uporabili [15] orodje za ustvarjanje kode, ustvarjeno v Pythonu, ki ponuja ukazne vrstice in pripomočke na grafičnem vmesniku. Za generiranje kode v specifičnih programskih jezikih uporablja definicije MAVLink narečij. Po spremembi definicije sporočila je treba to ponoviti tako za C++ kot za Javo, da zagotovimo sinhronizirano razčlenitev sporočil. Izvorne datoteke C++ se samodejno ustvarijo pri sestavljanju knjižnice MAVLink.

III.III Android projekt

Androidna aplikacija, ki deluje na tabličnem računalniku, uporablja mobilni SDK DJI za interakcijo z brezpilotnikom s povezavo preko daljinskega upravljalnika. Za dostop do avtopilota se je potrebno najprej registrirati kot razvijalec DJI in določiti namen svoje aplikacije, sledi pridobitev kode za avtorizacijo. Najlažji način za integracijo SDK-ja je uporaba skripte Gradle build za uvoz knjižnic preko Mavena. Aplikacija se ob zagonu poskusi povezati s kakršnimkoli izdelkom DJI, v tem primeru z daljinskim upravljalnikom, ki letalniku pošlje poizvedbo o specifikacijah.

Zanimata nas predvsem dve funkciji povratnega klica SDK-ja, ena za sprejemanje podatkov in druga za pošiljanje podatkov. Obe kot vhodni argument zahtevata funkcijo, ki se sproži, ko je prenos uspešno opravljen. Ker lahko obstaja samo ena funkcija, ki se v obeh primerih imenuje funkcija povratnega klica, je treba posebno pozornost nameniti organiziranju ukazov v čakalno vrsto. Za pridobitev izvorne kode in razčlenjevalca sporočil MAVLink uporabljamo generator kod, ki ustvari kodo za branje sporočil v jeziku Java. Poenostavljena predstavitev celotnega delovnega toka je razvidna iz tega diagrama 3.3.

Povratni klic za prejemanje podatkov se sproži, ko je prejet kateri koli paket, ki ima pravilen kontrolni seštevek (angl. checksum). Parameter funkcije nato poda paket kot niz bajtov, ki jih je treba razčleniti, da bi prejeli kakršnekoli koristne podatke. Da bi to dosegli, uporabimo pomožni razred, ki ga ponuja generator kode MAVLink Java, ki razčleni bajt po bajt in vrača trenutni napredek sestave sporočila, kar nam tudi pove, katero polje sporočila je bilo nazadnje prejeto. Prva vrednost, ki se jo pričakuje, je čarobna številka, ki nam pove različico protokola MAVLink. Če se zgodi napaka, se razčlenilec resetira in čaka na prihod novega sporočila, medtem ko ignorira preostale bajte. Napaka se lahko zgodi, če so paketi nepravilno urejeni, če je sporočilo večje od 100 B, če kontrolni seštevek ni pravilen ali če overitveni ključ ne more dešifrirati sporočila. Ko je sporočilo končno sestavljeno, uporabimo objekta, ki odda vsakemu naročenemu ukazu najnovejši element. Te ulovijo naročniki in izvedejo ustrezne akcije.

Pri odločanju, katero sporočilo naj bo poslano ob določenem času, uporabimo čakalno vrsto prvi noter in prvi ven oz. sklad (FIFO) na niti v ozadju. Ta čakalna vrsta vsebuje ukaze, katerih naloga je izvesti operacijo, ki vključuje tudi pošiljanje podatkov letalniku. Če v določenem trenutku ni nobenih ukazov na voljo, nit spi. V nasprotnem primeru se vzame prvega v vrsti, sestavi se ustrezno MAVLink sporočilo in se ga kodira v niz bajtov. Ta se nato razdeli na pakete v velikosti do 100 B in se jih pošlje drug za drugim s funkcijo za pošiljanje povratnih klicev. Če RC uspešno pošlje vse pakete, ukaz kliče svojo povratno funkcijo za uspeh in zapusti sklad.

Uspeli smo razviti odzivno aplikacijo, ki v nobenem trenutku ne blokira uporabniškega vmesnika in uporabniku ne preprečuje opravljanje novih nalog. Na primer pri pošiljanju velike datoteke letalniku sistem ne bo visel in bo zmogel vmes izvajati druge ukaze, saj se novo sporočilo za pošiljanje naslednjega dela datoteke ustvari šele po tem, ko se je prejšnji sam razrešil, kar omogoča, da nov uporabniški vhod stopi v čakalno vrsto prej. To bi se lahko izkazalo kot koristno, saj omogoča letenje brezpilotnega letala tudi med prenosom novih parametrov misije. Vse posodobitve uporabniškega vmesnika se izvedejo tudi v ločeni niti, spet kot čakalna vrsta. Zato ukazu, ki izda zahtevo za risanje, ni treba čakati, da se razreši, ampak lahko nadaljuje z njegovo izvedbo. Dodajanje preprostih novih sporočil je precej enostavno, saj zahteva le določitev funkcije preslikave vhodnih vrednosti.

III.IV ROS projekt

Naš ROS paket je bil inicializiran znotraj standardnega catkin delovnega prostora. Ustvarili smo vozlišče (angl. node) ROS. Ta si ime skupine deli z drugimi vozlišči, ki delujejo na vgrajenem računalniku, poskrbi za inicializacijo upravljavca vozlišč in ustvari primerek razreda, ki smo mu rekli TransparentTransmissionAPI (TTApi). Ta TTApi omogoča uporabo MAVLink mosta preko daljinskega upravljalnika z Data Transparent Transmission ali TCP/IP, ustvari razčlenjevalnik sporočil MAVLink in inicializira dva razreda, ki skrbita za sprejemanje in pošiljanje sporočil. Njegova delovna zanka spi v skladu s hitrostjo zanke, katere različne vrednosti smo kasneje raziskali 3.9.

Kaj se zgodi, ko je sporočilo MAVLink ali sporočilo ROS prejeto, je določeno v naših vtičnikih. Njihova naloga je inicializirati naročnike in funkcije za pošiljanje sporočil. Da se jih prosto zamenjati, da se zagotovi različne nabore funkcij ali za obdelavo različnih MAVLink narečij. Upravljajo jih z ROS paketom pluginlib, ki dinamično nalaga vtičnike s pomočjo infrastrukture za gradnjo ROS.

Obdelovalci sporočil so ustvarjeni tako, da jih povežejo z edinstvenim ID-jem sporočila MAVLink, medtem ko so naročniki običajne ROS vrste. To je koristno za spreminjanje in razširitev vedenja ogrodja, ne da bi bilo treba vedeti, kako deluje preostala koda.

Običajno pošiljamo sporočila MAVLink, ko prejmemo vhod iz sistema ROS, na primer telemetrične podatke z letalnika. Naročnine na teme ROS so opredeljene v vtičnikih in vsaka ima povezan povratni klic. Odgovorni so za inicializiranje ustreznega sporočila MAVLink in izpolnjevanje polj. To se nato serijsko pretvori v splošno sporočilo MAVLink in poda razčlenjevalniku, ki to razčleni v polje bajtov, kar se nadalje razdeli na pakete v velikosti do 100 B, ki se uporabljajo kot parameter za odjemalca storitev ROS, ki ga nudi Onboard ROS SDK. Nato avtopilot pošlje pakete aplikaciji Android preko sprejemnega modula. Za razliko od mobilnega SDK-ja ne dobimo povratnih informacij, ali je bil prenos uspešen ali ne.

Za prejemanje sporočil imamo naročnino na temo, ki jo ponuja SDK in nam da polje bajtov, kadar pride nov paket. To se razčleni bajt po bajt, dokler ni mogoče pridobiti celotnega sporočila MAVLink ali se pojavi napaka ali mora preprosto priti več paketov, preden je mogoče sestaviti celotno sporočilo. Sledi klic funkcije iz vtičnikov, ki uporabljajo to vrsto sporočila, označeno z msgid. Povratni klic nato izvede potrebna dejanja, kot je letenje na položaj ali pristanek ali v primeru prenosa datoteke počasno zbiranje podatkov iz več sporočil. Potencialno lahko imamo tudi več funkcij, ki prejmejo isto sporočilo, in vsaka opravi nalogo, določeno v svojem vtičniku.

III.V Integracija s QGroundControl

Ko smo uspeli prikazati zanesljiv koncept za naše komunikacijsko ogrodje za MAVLink, smo preučili tudi možnost, da bi ga vključili v obstoječo zemeljsko nadzorno postajo. Izbrali smo QGroundControl, saj tudi ta deluje preko MAVLink protokola, je odprtokoden in deluje na večini operacijskih sistemov. QGroundControl podpira samo TCP/IP, zato za povezavo z letalnikom 3.5 potrebuje UDP ali TCP most.

Kot del protokola MAVLink naj bi vsako vozilo vsako sekundo pošiljalo sporočilo Heartbeat, s čimer sporoča svoje zmožnosti in potrdi stabilno povezavo. S tem je vgrajeni računalnik identificiral naš brezpilotni zrakoplov kot generični MAVLink avtopilot. Poslali smo preproste podatke o telemetriji, ki so prikazani na desni strani in omogočajo sledenje statusu brezpilotnika. Aplikacija omogoča načrtovanje kakršnih koli letalskih nalog in ukaze pošlje kot zaporedje sporočil MAVLink. Ta so zakodirana kot sporočila MAVLink Command, ki se izvajajo korak za korakom in zahtevajo posebne potrditvene odgovore obeh strani. Podatki so prav tako strukturirani na način, ki se močno razlikuje od strukture v DJI ROS. MAVROS ponuja nekaj vtičnikov za ravnanje s pretvorbo iz ROS-a v slog MAVLink, vendar v obratni smeri kot v naši situaciji, kar bi pomenilo veliko delovno obremenitev in bi bilo izven obsega projekta. Kljub temu smo pokazali, da je okvir sposoben zagotoviti most med ekosistemom ROS in odprtokodno MAVLink zemeljsko nadzorno postajo.

III.VI Test prenosa

V okviru naloge smo testirali tudi učinek razdalje in velikosti poslanih paketov na hitrost pretoka podatkov. Prva slika prikazuje število izgubljenih sporočil ob pošiljanju na letalnik 3.7. Do 200 metrov razdalje se je izgubilo zelo malo paketov, nato pa je zanesljivost nenadoma padla. Zdi se, da velikost sporočila ne vpliva nujno na verjetnost izgube. Kot je najbrž pričakovano, se je pretok 3.8 zmanjšal z razdaljo, čeprav se paketi najprej niso začeli izgubljati. Opazimo lahko, da oddaljenost močno vpliva na količino poslanih podatkov. Razlika med najmanjšo in največjo razdaljo je približno štirikratna.

Ko smo ustvarili grafe za pretok podatkov, prejetih z letalnika, smo rezultate strnili v najmanjšo in največjo razdaljo. Posebej smo še prikazali učinek hitrosti zanke. Na 3.7 lahko opazimo, da hitrost zanke vozlišča, ki ustreza številu poslanih paketov na sekundo, močno vpliva na število izgubljenih paketov. Meritve so zelo nestabilne, vendar lahko vidimo, kako višje hitrosti zanke povečajo izgubo paketov. Pri najhitrejšem paketnem prenosu 100 Hz lahko vidimo, da se približno tretjina izgubi pri največji velikosti paketa. Opazimo pa tudi 3.8, da lahko s povečanimi hitrostmi zanke še vedno dosežemo veliko boljši pretok.

IV ROS vtičnik za navidezno resničnost

IV.I Uporaba vtičnika za RViz

Naše ogrodje za podporo NR je zasnovano tako, da ga uporabljamo kot vtičnik, ki ga uvozimo v RViz, in nam omogoča interakcijo s sceno s slušalkami in krmilniki, ki podpirajo OpenVR. Upravlja se s paketom ROS pluginlib, ki dinamično nalaga vtičnike s pomočjo orodja za gradnjo ROS-a. Upravljalec vtičnikov ga registrira in inicializira glede na vsebino datoteke *package.xml.* To omogoči uporabo vtičnika v kateri koli instanci aplikacije RViz in ga je mogoče preprosto dodati z grafičnega vmesnika za izbiro vizualizacij, saj podeduje osnovni razred zaslona RViz. Dodatno prilagajanje izvajanja je zagotovljeno z uporabo polj, ki jih je mogoče uporabiti za preklapljanje med upodabljanjem navidezne scene ali prikazovanjem video toka stereo kamer na očalih. To omogoča krmiljenje navidezne kamere s premikanjem HMD-ja ali prostim ogledom prizora in spreminjanjem naročnikov na teme (angl. topic) video pretoka. Pregled delovanja zagona vtičnika in zanke upodabljanja je prikazan na tej sliki 4.2.

IV.II Zagon vtičnika

Ob vključitvi našega vtičnika v grafičnem vmesniku RViz se najprej sproži poskus inicializacije OpenVR in poskus povezovanja s katerokoli vključeno strojno opremo za NR. Če so očala za NR najdena, zahtevamo ekskluziven dostop do vizualizacije na njegovih zaslonih z zagonom API-ja v aplikacijskem načinu. Kadar želimo brati le lokacijske podatke, lahko zahtevamo samo pravico za branje, kar bo kasneje koristno za uporabo krmilnikov.

Ob uspehu API naloži vse podatke za specifično strojno opremo, ki se uporabijo za nastavitev stvari v OGRE. Ustvarimo tudi dinamične spremenljivke, ki jih lahko spreminjamo med izvajanjem v Rvizu, in njihove ustrezne povratne klice. Z njimi spreminjamo orientacijo scene zaradi gibanja slušalk, preklop na stereo-video prenos in spreminjanje tf poslušalcev in oddajnikov. Nato nadaljujemo s sklicevanjem na delujoči primerek zaslonskega konteksta RViz, glavno povezavo, ki jo ima zaslon s preostankom RViz, in nam omogoča dostop do upraviteljev okvirjev in izbire. Dobimo tudi referenco na upravitelja scen OGRE, ki upravlja hierarhijo scene, in upravitelja tekstur, ki pozneje ustvari teksture, ki bodo korelirale z očmi. Nato imamo dve različni nastavitvi za OGRE, odvisno od izbire ciljev upodabljanja: navidezni prizor ali prikazovanje video toka s stereo kamerami.

IV.III Nastavitev navidezne scene

Začnemo z ustvarjanjem vozlišča za uprizoritev prizora, ki predstavlja položaj glave in je relativen glede na svetovni koordinatni okvir, nanj pa pritrdimo dva na novo ustvarjena objekta za kameri. Za pravilno projekcijo v OpenVR potem pridobimo matrike za prevajanje iz očesnega prostora v prostor scene, ki zagotovijo stereo dispariteto. Uporabimo jih na kamerah, ki predstavljajo oči.

Če se uporabnik odloči za preklop na stereo-video pretok, pridobimo objekte za upodabljanje obeh tekstur in odstranimo njihova navidezna vidna polja (angl. viewport), da preprečimo odvečno upodabljanje. Prav tako spremenimo teksture OpenVR, da sedaj referencirajo teksturi stereo kamer.

IV.IV Zanka za uprizoritev

Začetek iteracije upodabljanja začne RViz, ki najprej obdela svoj uporabniški vmesnik in glavni zaslon v ozadju. Po tem se pokliče povratni klic za posodobitev v našem vtičniku, kjer se obravnavajo preobrazbe, upodabljanje in oddajanje okvirja v kompozitor OpenVR. Slike, ki so bile v virtualnih prizorih pritrjene na teksture OGRE, so že izdelane in pripravljene za uporabo. To pa pomeni, da je treba pred premikanjem kamere brezpilotnika in njegove kamere upoštevati en okvir zamika, čeprav to v resnici ni problem.

Prva stvar, ki jo naredimo, je zahteva za transformacijo tf iz sveta v pozo navidezne kamere. Vozlišče prizorišča, ki predstavlja postavitev glave, je nastavljeno na kardansko glavo, da simulira gledanje skozi kamero.

Sledi klic blokade v kompozitor OpenVR, kar nam omogoči dostop do položaja opreme za NR in je potrebno za poznejši klic oddaje. To se reši le približno 2–3 ms pred tem, ko ga mora kompozitor pripraviti na branje, zato so očala Vive omejena na 90 Hz. V tem času je potrebno izračunati potrebne transformacije in pripraviti teksture za izris slike v očalih. Gledišče v navidezni sceni je odvisno od postavitve baznih postaj Vive, lokacije uporabnika znotraj prostora, rotacije očal, kompenzacije vrtenja glede na nastavitve dinamičnih spremenljivk vtičnika in tf transformacije (ki v našem primeru predstavlja lokacijo kamere na kardanski glavi letalnika). Po vseh teh operacijah dobimo končno transformacijo kamere v navidezni sceni, ki jo s tf oddamo za kasnejši izračun lokacije krmilnikov.

V zadnjem koraku lokacijo kamere rahlo zamaknemo za razdaljo med očmi in skupaj s teksturami ter opredelitvijo obsega UV koordinat posredujemo v OpenVR kompozitor. Če je bilo to storjeno znotraj časovnega okvirja, bo uporabnik končno dobil izrisan prizor v očalih.

IV.V Branje ocenjenega položaja

Za uporabo očal in krmilnikov za manipulacijo in interakcijo s sceno smo ustvarili vozlišče ROS za branje ocenjenega položaja. Tako kot vtičnik se vozlišče začne z inicializacijo API OpenVR, čeprav tokrat zahtevamo le dostop za branje. Če je uspešen, nam posreduje ID-je, ki jih kasneje uporabimo za dostop do podatkov.

Glavna zanka torej ne uporablja blokirnega klica za prejem posodobljenih pozicij kot prej, saj ne potrebujemo sinhronizacije upodabljanja. Vedno zahtevamo nove podatke o položaju in jih razčlenimo v tf preobrazbo. Nato jih objavimo v drevesu tf ločeno 4.5od preostalih tf okvirjev, ki jih uporabljamo v drugih paketih in vtičniku. To je zato, ker Vive uporablja poljubno koordinatni izhodišče, ustvarjeno med Steam's Room Setup, ki je uporaben samo za določanje relativnih transformacij iz položaja očal v krmilnike v resničnem svetu.

Poleg premika in usmeritve krmilnikov nas zanima tudi branje pritiskov na gumbe. To beremo kot analogne vrednosti z 2D sledilne ploščice in sprožilca ter dvojišče vrednosti z ostalih gumbov. Te podatke objavljamo kot sporočilo ROS Joy Sensor, ki lahko nosi poljubno število pritiskov gumbov in vrednosti osi. To pomeni, da vsako sprejemno vozlišče potrebuje posebno razčlenitev, da lahko pravilno interpretira podatke. Imamo tudi par naročnikov ROS, za katere lahko objavimo zahtevo po vibracijah.

IV.VI Simulacija v Gazebu

Za zaznavanje trkov in fizikalno simulacijo smo vključili Gazebo kot sklop paketov ROS. Z njim smo v navidezno upodobitev uvozili urbano območje, brezpilotne letalnike in interaktivne krmilnike kot fizične objekte. Če bi na tej točki uporabljali krmilnike kot navidezne roke v sceni, bi bili omejeni le na interakcijo z okolico, ki je v dosegu roke.

Za razširitev možnosti interakcije s sceno smo uporabili metodo dolgega dosega rok. Položaj dobimo kot transformacijo med svetom in navideznimi krmilniki. Tako pridemo do položaja, do katerega bi se roke želele premikati v simuliranem prizoru.

Uporabljamo tudi vhod sprožilnega gumba. Oddaljenost roke se linearno pomakne od telesa, sorazmerno s tem, kolikor je pritiska na sprožilec. Usmerjeni vektor je določen s položajem krmilnika in pripadajočo ramo 4.6. Za začetek interakcije s prizorom lahko uporabimo gumb za prijem (angl. grip).

IV.VII Upravljanje prilagodljivega pogleda

Nadaljevali smo s prilagajanjem navideznega okolja s teleoperacijo, ki bi uporabniku nudila okolju prilagojeno gledišče, ki bi se samodejno prilagajalo za izboljšanje varnosti in nemoteno delovanje uporabnika.

Naš načrt je bil zagotoviti dva vmesnika za vizualizacijo okolja. Eden izmed njih je upodabljanje navideznega prizora, kar je uporabno za izvajanje simuliranih letov ali letenje na prostem v svetu, kjer ne bi pričakovali veliko dinamičnih sprememb. Druga možnost je ogled sveta s stereo kamero na krovu drona, ki bi omogočila takojšen pogled na trenutni prizor, resničen ali simuliran. Za povečanje razumljivosti scene v teh dveh različnih primerih uporabljamo navidezni pogled preko očal za NR, kar bi lahko tudi zmanjšalo zapletenost naloge zaradi razumevanja globine.

Naš prototip postavi gledišče v orbito brezpilotnika, kjer ostane in sledi na določeni razdalji. Uporabnik lahko intuitivno prilagodi kot gledanja z gibi očal za NR 4.7. Če na primer pogledamo levo, se bo navidezna kamera preusmerila okoli navpične osi zrakoplova, s pogledom navzdol pa jo bo usmerila okoli vodoravne osi. Z uporabo naravnih gibov glave kot metode vnosa bi moral uporabnik imeti možnost, da hitro najde primerno gledišče.

V Sklep

V magistrskem delu smo raziskali možnosti razširitve orodij za razvijalce brezpilotnikov, ki podpirajo ROS. Predlagali in implementirali smo ogrodje, ki omogoča dodajanje novih ukazov za letalnike DJI na večjih razdaljah z uporabo daljinskega upravljalnika in integracijo podpore za NR za večino sodobnih očal v RViz.

Začeli smo z iskanjem različnih orodij, ki se uporabljajo za komunikacijo med letalnikom in zemeljsko nadzorno postajo. Izbrali smo protokol za sporočanje MAVLink, deloma tudi zato, ker se uporablja v številnih odprtokodnih avtopilotih in zaradi prilagodljivosti definicij sporočil. Daljinski upravljalnik ima zelo majhno pasovno širino, vendar ima zaradi svojega dometa veliko prednost pred Wi-Fi povezavo. Uporablja se lahko v odprtih ter zaprtih prostorih, česar mobilna omrežna povezava ne bi omogočala. Osredotočili smo se na zagotavljanje ogrodja za dodajanje novih sporočil, ki bi jih lahko uporabili za pošiljanje ukazov, podatkov dnevnika, stanja leta in podatkov zunanjih senzorjev. Uvedli smo tudi protokol za zanesljiv prenos datotek med napravami, ki lahko vsebujejo konfiguracijske parametre za misije letenja ali računalniške rezultate. Skratka, zagotovili smo povezavo med mobilno platformo in vgrajenim računalnikom ROS, kar bi razvijalcem omogočilo večjo prožnost pri ustvarjanju lastnih API-jev za letenje.

V nadaljevanju smo se lotili neposredne integracije podpore NR v ekosistemu ROS, da bi omogočili lažji razvoj in oblikovanje prototipov. Ustvarili smo vtičnik, ki je neposredno na voljo v RViz in ga uprizori njegov pogon OGRE. Vmesnik do očal za NR je izveden z uporabo OpenVR. Stereo slike lahko prikažemo tako, da jih generiramo iz virtualne scene, kjer od poslušalca tf prejmemo položaj kamere in nato uporabimo stereo projekcijo. Tudi video tok lahko prejmemo s stereo kamere in ga neposredno prikažemo v očalih. Predstavili smo primer uporabe upravljanja pogleda, kjer glavni letalnik opravlja svojo nalogo. Sledi mu navidezna kamera ali sekundarni letalnik, ki je tam, da bi zagotovil boljši pregled nad prizorom. Uporabnik spreminja gledišče s premikanjem položaja očal. Ideja je, da lahko operater pridobi alternativno gledišče na območju operacij in ga prilagodi z intuitivnimi gibi glave, ne da bi potreboval dodatni krmilnik. Da bi omogočili nadaljnjo interakcijo z okoljem in omogočili na primer prilagajanje poti leta, smo uvedli tudi podporo za odčitavanje položaja krmilnikov in pritiskov na gumbe.

Chapter 1

Introduction

With the advent of affordable and easy to use drones we have seen a steep rise in their sales for personal and commercial use. Even though they used to be commonly associated with aerial photography, small toys for entertainment and military use, the modern civilian drones fill a variety of roles. They can be used for inspecting the integrity of industrial facilities, taking pictures and videos of special events like the opening of the Olympic Games or a wedding, in agriculture for shepherding and surveying of crops, search and rescue missions in hard-to-reach places, help in the sales of real estate by making wholesome pictures of the properties, for performing deliveries of smaller packages, and surveying vast stretches of land.

This demand is showing an exponential growth of commercial drone sales [25]. This provides a strong incentive for active research to provide new capabilities, enhance existing capabilities and find new use-cases especially for Unmanned Aerial Vehicles (UAV). To this end, robust and lightweight components have been created and competent autopilot boards have been developed. By using different sensors, computer vision algorithms, and various artificial intelligence, autonomous flight is currently a research topic receiving a significant amount of attention.

Out of many competing companies Dà-Jiāng Innovations (DJI) has come atop as the market share leader in civilian drone sales, industrial use and almost every software category [1]. It has 72% of the global market share and in the medium price range of $1000-$2000 it even managed to capture 87% of the market share. Its market share has been steadily increasing for many years with no signs of slowing down. This is both troubling and a good reason to work with their brand of drones. As part of their effort on providing their own custom tailored software, the APIs they provide quick and efficient function calls, for things such as transmitting video, basic flight data or remote steering. The issue stems from very limited options in creating custom applications and transmitting arbitrary data. Currently, a WiFi connection to a ground control station (GCS) is used and the onboard computer on the drone runs the programs and receives the data. Unfortunately, this limits the range of the communication and portability.

Furthermore, this century has seen rapid advancement in the development of virtual reality (VR), driven by breakthroughs in computer technology, especially small and powerful mobile technologies [27]. Carried on the back of the rise of smartphones with high resolution displays and 3D graphics capabilities, while also driving down the prices of their hardware components, a new generation of lightweight VR devices have been developed. The video game industry has led the drive in the development of consumer VR. Motion controllers, gesture recognition, depth-sensing cameras sensors and natural human interfaces are already a part of most common VR sets available to consumers.

The recent years have seen the release of quite a few digital VR headsets in the price range of around 200-500 Euro; namely Samsung Odyssey VR, Oculus Rift, Sony Playstation VR and HTC Vive. All of them have different quirks to cater to different consumers, but what they share are high resolution displays, tracking of up to 6 degrees of freedom (DOF) and most also provide the option of using hand-held controllers. There was also the release of a few interim VR products such as Google Cardboard, a do-it-yourself headset that uses your smartphone as a screen. Companies like Samsung have taken this concept further with products such as the Galaxy Gear, which is mass produced and contains “smart” features such as gesture control.

In the first part of our work, we focused on providing a robust and scalable solution for communications between a DJI M100 drone and an Android tablet computer, which was attached to the drone’s Remote Controller (RC). The telecommunication link was achieved through the RC. This data link is shared with the usual RC-to-drone communication link, with only a small bandwidth reserved for the tablet-to-drone communication link. On the on-board computer of the drone, the inter-process communication is achieved using the Robot Operating System (ROS) middleware.

Since the bandwidth of the tablet-to-drone communication link was very limited, the messaging protocol had to be lightweight and concise, which led us to select the Micro Air Vehicle Link (MAVLink) communication protocol. We defined a software architecture, a new set of ROS-like messages, and a file transmission protocol which, when compiled into appropriate source code, provided us with a framework for achieving an efficient and adaptable communication between DJI M Series drones and a smartphone App Interface. By using MAVLink, the designed framework provided efficient usage of limited bandwidth within the wireless link and allowed us to remotely control the drone without the use of a stationary GCS.

We performed a set of experimental flight tests in an outdoor environment to try out the system’s reliability and a set of static tests to measure the packet transmission throughput as well as to get further insight into the effects of distance on the packet transmission by measuring data bandwidth, latency and range.

We chose to use both a headset and controllers by combining them into a VR solution for drone flight. We used one of the most popular VR devices at the time of writing, HTC Vive, for displaying our scene to the pilot, with Vive controllers used for manipulation and Vive base stations providing tracking for those components. The underlying framework we used was ROS Indigo for Ubuntu 14.04, with the heap of packets communicating with each other via ROS messages. The virtual scene used for testing was built up in

Google's 3D modeling software Sketchup. The physics simulation of a Parrot Bebop 2 and the world was done in RotorS, which was a MAVLink based Gazebo simulator. The visualization was then done in RViz, which was a dedicated 3D visualizer for ROS. The 3D scene was then used in a plugin, where the OGRE 3D engine was used to render separate views for the Vive head-mounted display (HMD) and then they were sent to the compositor provided by OpenVR and SteamVR.

1.1 Motivation

We started by focusing on the communication protocol between a mobile platform and the DJI M100 drone. DJI autopilots provide few options for easy development unlike other open-source autopilots. While this is not an issue for an average user, since the software DJI provides for its drones is of high quality, it does not give many tools for developers to work with. Our main motivation was creating a prototype of a GCS, which would provide an open-source wireless interface between an Android device and the onboard computer, which could later be integrated into an existing GCS application. The communication link is based on the link between the RC and the autopilot board (or flight controller board). The main reason for choosing the use of the shared communication link is that the RC to Drone communication link has been engineered for reliability, and should provide a safe, although of limited bandwidth, means of communication between our smartphone App interface and the on-board computer of the drone.

Our framework should provide a scalable software for adding new message definitions in XML, which are then compiled into appropriate Java and C++ files. The systems had to be then designed in such a way, so that adding new messages required only code, which told which task needed to be executed. This would provide an easier way to test new algorithms made at the ICG for drones and provide a very mobile Android platform instead of the traditional GCS based on a laptop computer. We also needed a way to transmit any file, which would allow us to, for example, send and execute a specific mission configuration. Another very benefit of using an open-source messaging protocol was that we avoided being limited to the functionalities provided by the SDKs. This ability to quickly reconfigure was important for research purposes.

When working on the VR interface, our goal in mind was providing an alternative means of visualization and interaction with the world for a drone pilot in a search and rescue scenario. We used the virtual viewpoint of the HMD interface in this setup, to follow the Bebop 2 drone respectively. The drone performed the task of searching and flying on its precomputed path followed by the virtual viewpoint of the HMD, with which we planned to interact with by using Vive controllers. We would be able to provide an optimal view vantage position to the pilot, so he can get a better overview of the area in interest and reduce mental load.

To provide VR capabilities for use in the most popular ROS 3D visualizer RViz, we used the ROS' package pluginlib to create a plugin. This would then use the built-in 3D graphics engine OGRE and Valve's OpenVR API that provided access to the VR hardware from multiple vendors. We would provide two options for visualizations, First, stereo video streams from ROS topics and, second, rendering the virtual scene in RViz for both eyes. The Vive controller's relative position and orientation to the HMD would then be used for interacting with the world from another package. The headsets movements would also be used to control the movement of the virtual camera by adapting the view-port.

1.2 Contributions

The main contribution of both the communication protocol and the VR interface is to act as tools that can be used to conduct further research and develop of emerging solutions.

Our MAVLink based communication protocol provides better portability by only requiring an Android device, while also allowing us to use the great range of an RC. The interface also allows use of a normal TCP/IP connection. Another important point is this framework helps with opening up the DJI drones for further open-source development. The drawback is that it requires implementing new message definitions when one wishes to add new capabilities. Although, we eased some of these difficulties through clever use of programming patterns.

The VR interface on the other hand allowed the use of SteamVR compliant devices with along with the OpenVR SDK directly in ROS. While there were up-to-date options for using VR equipment, they required interfaces to be used through game engines, so we designed the plugin primarily for its use in the ecosystem's most popular visualizer, RViz. It could then either visualize the virtual world or stream video from stereo cameras, while also capturing the input of corresponding controllers. We showed a simple adaptive view management use-case with a pair of drones, where one was controlled with an HTC Vive, demonstrating the usage of our plugin as a tool for VR supported robot interaction.

1.3 Framework specifications

The main goal of the communication framework developed during this project was to achieve an efficient and adaptable communication between DJI M Series drones and an Android device as well as providing an ROS integrated VR interface for use in drone flight scenarios, with RViz as the visualization tool.

The following is a list of the main characteristics fulfilled by the developed framework:

- The exchange of data is achieved using the DJI's *Data Transparent Transmission*, which shares the communication link with the RC-to-drone communication.

- The framework should be able to deal with disconnections of the communication link, which may happen intermediately due to it being wireless.
- Responsive user interface, for instance the transmission of big files, which should not render the communication link unusable for other types of messages.
- Usage or design of a communication protocol with as little package overhead as possible.
- Capacity of modifying and designing new message definitions, so that new sensors and features can easily be included into a drone application.
- Setup of the framework to use a reasonable amount of the bandwidth, so that a good balance between packet loss and package transmission speed can be achieved. This setup should be supported on experiments that measure the actual achievable bandwidth of the DJI's *Data Transparent Transmission.*
- Possibility of transmitting big files, for example allowing to send a specific mission configuration for the drone.
- Easy to maintain code-base. The Android project uses a command pattern, where each new message defines a new command, with inheritance providing shared functionalities. The ROS package uses a plugin system, where each new functionality is defined in a separate plugin, which can be ported between different projects.
- Integrated with the ROS Indigo and RViz visualization tool.
- Uses modern VR headset HTC Vive for viewing the scene.
- Uses an abstraction of VR hardware layer by using up-to-date OpenVR and SteamVR without requiring specific version as was previously required.

- Is able to support any other SteamVR enabled device.
- Enables the use of Vive controllers as intuitive tools to manipulate the world, by modeling them as hand extensions
- Allows viewing of the virtual scene or through a stereo camera video stream, or a mix of two.
- Achieves a stable frame rate and has no stuttering.
- Using the virtual viewpoint of the HMD to facilitate better overview over the environment by the pilot.

Chapter 2

Tools

2.1 Robot Operating System

The Robot Operating System [18] (ROS) is an open source middleware software, which provides a management system for robotic components. The services provided include hardware abstraction, low-level device control, message transmission between different processes and package management. In practice such a system is based on a number of independent nodes, which communicate by using a publish/subscribe messaging system, potentially running on any number of computers. Programming solutions in this framework are usually provided as packages, which you can then overlay and import in any other project [22]. The version that we used is called Indigo, which runs only on Ubuntu 14.04 LTS. We also considered using [13] as a substitute to ROS, since it promises better performance and less latency, and provides interoperability with ROS, but in the end decided against it since the latter framework has wider usage and we are more knowledgeable in its use.

Development was first done on my laptop, which worked out fine for work on the communication protocol. We later switched to PCs in the Deskotheque - the drone space for ICG - because we struggled with graphics drivers problems due to issues with integrated graphics card. Programming was mostly done in Java and C++ language, with some Python on scripts from other

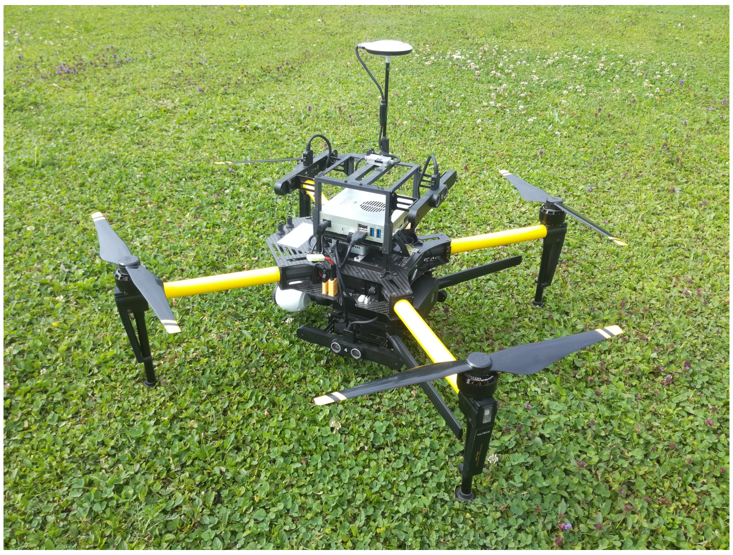


Figure 2.1: An image of the DJI M100 from the ICG during our flight test in the field.

packages. Android Studio and QTCreator were my IDEs of choice.

2.2 DJI M100

This quadcopter is DJI's first attempt at providing a fully-integrated flight platform specifically targeted at developers. Its hardware expansion bays can carry any configuration of sensors and devices up to 1 kg 2.1. It also provides ultrasonic sensors and depth cameras for automatic obstacle detection.

The two new DJI's SDKs released together with the drone provide a possibility of creating advanced flight controls and custom mobile applications. On the Linux computer, mounted on the drone, we use the Onboard SDK ROS, which provides an interface to the autopilot that our ROS nodes use for guidance. For sending commands to the drone we use the Mobile SDK for Android, which provides support for accessing common sensor data, video streams and other functionalities. Most importantly to us it provides a channel for custom data transmission. We use a USB connection between the RC and the Android device to connect to the drone 2.2.

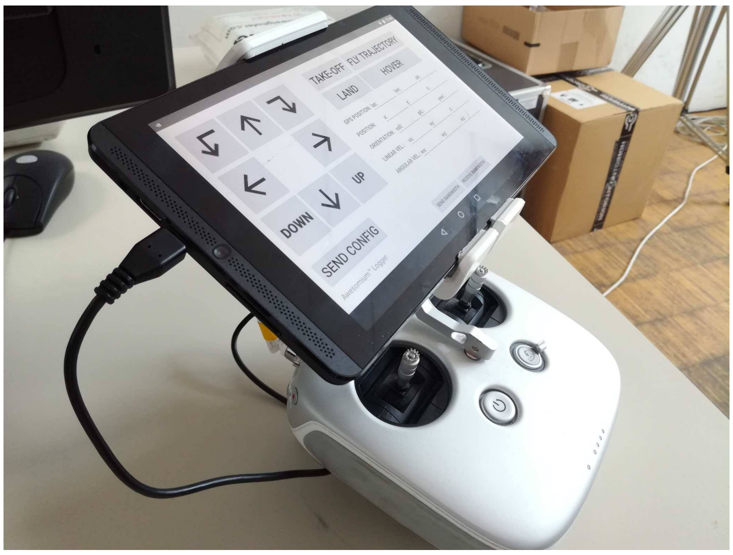


Figure 2.2: An image showing the RC connected to the tablet via a USB cable. On the screen we can see our demo application for MAVLink communication.

The link for sending arbitrary data is called Data Transparent Transmission and is part of the DJI Mobile SDK for Android and iOS. The specified upstream bandwidth is 1 KB/s and downstream is 8KB/s. The message size is also limited to 100B at one time. As is apparent the link is very limited in the amount of traffic that can be handled. It does not provide support for dropped and corrupted packets, which have to be provided by the protocols using it if needed.

2.3 MAVLink

This is the messaging protocol we chose for our binary telemetry. Is is a lightweight, header-only message protocol for communicating with small unmanned vehicles. It is used mostly for communication between UAVs and GCSs, and for intercommunication between the subsystems. It is used by many autopilots, such as Parrot, Ardupilot, PX4 and more. It is not supported in any way on the DJI drones, since they mostly provide only propri-

Table 2.1: MAVLink 2 packet format. Bold text denotes required fields.

Default name	Bytes	Description
magic	1	Protocol magic marker
len	1	Length of payload
incompat_flags	1	Flags that must be understood
compat_flags	1	Flags that can be ignored, if not understood
seq	1	Sequence of the packet
sysid	1	ID of the sender drone
compid	1	ID of the sender component
msgid	3	ID of the message
target_system	1	Optional field for point-to-point messages
target_component	1	Optional field for point-to-point messages
payload	max 253	A maximum of 253 payload bytes
checksum	2	X.25 CRC
signature	13	Signature which ensures, that the link is tamper-proof

etary software for operating the systems and communication on their vehicles.

We are using the newer MAVLink 2 version, which uses a message format that can be represented in the following way 2.1. The MAVLink message parser recognizes the protocol version by the *magic* field, although all messages are still backwards compatible by ignoring unrecognized fields. Every message needs its own unique message ID and the definition of its other payload fields in XML files, which can then be generated into appropriate source code for the supported languages, of which C++ and Java are interesting to us. The fields *sysid* and *compid* are set by the sender drone for identification purposes. Many common messages are point-to-point, therefore requiring target drone specification, otherwise the messages are sent in broadcast mode. Checksum *X.25* is used to verify messages integrity and ignoring corrupted chunks. Other fields are then generated automatically by the MAVLink code generator with the same values across different programming languages.

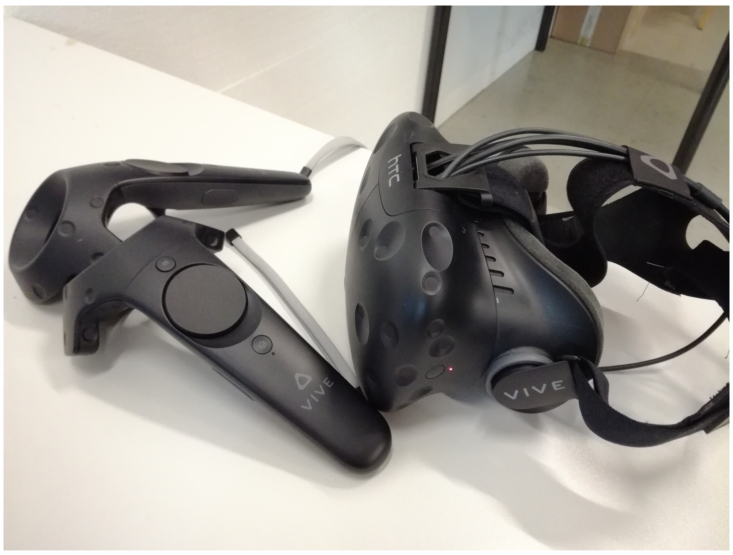


Figure 2.3: An image of the HTC Vive headset and controllers used during the project.

2.4 HTC Vive

Developed and released by HTC and Valve in 2016 this virtual reality headset [26] is one of the most popular currently. We chose this one because the ICG had the full equipment for it already and we wanted to be able to use the controllers.

The main hardware piece is the HMD itself, which is capable of a refresh rate of 90 Hz and a 110 degree field of view. The display is made out of a pair of organic light-emitting diode panels, each at a a display resolution of 1080x1200 pixels per eye, which puts it at 2160x1200 total pixels, requiring quite heavy processing power to run at the recommended refresh rate. See-through display is possible with the use of the front facing camera and allows the user to observe their surroundings without removing their headset. The HMD's position in space is tracked by the infrared (IR) sensors in the casing's divots, which track the IR pulses from the base stations. Additional sensor input for orientation correction is given by a G-Sensor and gyroscope.

The Vive Controller also has IR sensors to detect the base stations, pro-

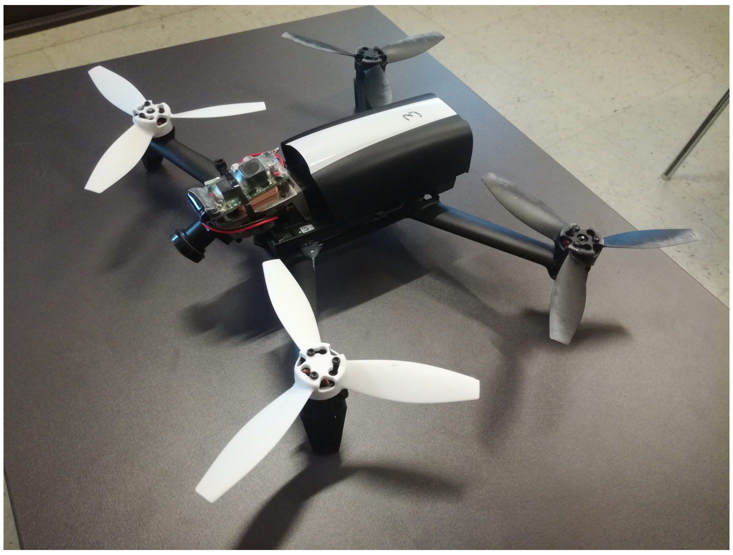


Figure 2.4: An image of the Bebop2 used in the Deskotheque.

viding them with full 6 DOF movements. It hosts an array of addition input methods with grip grip buttons, a dual-stage trigger and a 2D track pad. The approximate update rate given by the SteamVR Tracking system is at about 250Hz. They are connected to the HMD via a wireless link.

The Base Stations, also called Lighthouses, are a tracking system that can cover a field of about $20m^2$. They emit 60 pulses of IR rays per second, that are then detected by the headset and controllers. They are supposed to provide sub-millimeter precision, although there were issues of tracking if one of the base stations rays were occluded.

2.5 Parrot Bebop 2 and RotorS

The drone we use is Parrot Bebop 2, both in the RotorS simulation or for a possible real-world flight, although any MAVLink enabled autopilot could also be used. This particular model is especially lightweight at 0.5kg and up to 25min of flight time, so quite useful for surveying a scene, although it can't carry much other equipment.

Gazebo is an open-source simulator [12] that expedites algorithm testing, robot design and flight simulations using realistic scenarios. It provides the functionalities to accurately and efficiently simulate robots and drones in complex indoor and outdoor environments. It comes packaged with a robust physics engine, customizable graphics, and convenient programmatic and graphical interfaces. Different simulation models for specific robot platforms are then provided by the community. Integration with ROS is provided by using the package gazebo_ros_pkgs [10], which provides ROS messages, services and dynamic reconfiguration of parameters.

The specific variant of simulator we use is called RotorS, a MAVLink gazebo simulator [9]. It provides some drone models from the AscTec company, but the simulator is not limited for the use with these. There are simulated sensors coming with the simulator such as an inertial measurement unit, a generic odometry sensor, and the visual-inertial sensor, which can be mounted on the drone. The geometry of the world is used for collision detection, and the scene can also have additional extra forces like wind to make the simulated drone unstable. Any new interactions with the scene could be provided by writing additional plugins. The positions and other information about objects of interests are then published as ROS topics for use in RViz.

2.6 RViz

ROS visualization (RViz) is a 3D visualizer [11] for displaying sensor data, state information, camera data, infrared distance measurements, etc. from ROS. It is the most widely used tool for displaying data because of deep integration with the ROS ecosystem and provides direct interaction methods with the scene. Like Gazebo it also uses plugins to extend and add new functionalities.

We use it to display the models, included in our simulated test scene, in their correct position and the rest of the world as a static model. We also created two video streams from where the drone would have the stereo setup

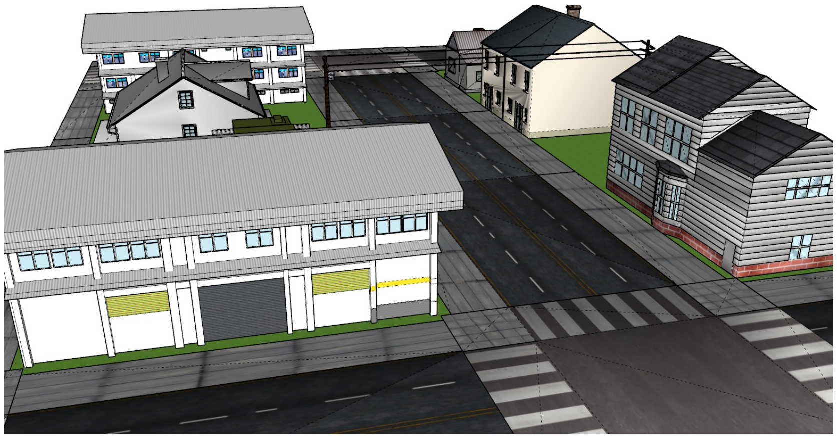


Figure 2.5: An image of the model of the world, created in SketchUp. It is used both for physics simulation in Gazebo and for visualization in RViz.

of cameras, which we later use as one the possible VR display inputs. We created the scene in Google's Sketchup, a 3D modeling software with the use of publicly available assets. We very quickly then encountered problems with low frames per second (FPS) in RVIZ. This is a problem, because 90 FPS is necessary in Vive to prevent stuttering, which is very jarring on the eyes. The tool uses an old version of OGRE graphics engine, which does more computation on the CPU then newer graphics engines would. It also uses only one core for the render loop of the whole scene, which makes the problem even more pronounced and the power of the graphics card does not have much impact. We tackled this problem by reducing the complexity of the scene, cleaning up the geometry and removing excess polygons created during designing of a model.

To handle translations and rotations between different components in the scene we use a ROS package called tf. Our system uses many 3D coordinate frames that change over time, such as a world frame, drone frame, Vive HMD frame, etc. tf keeps track of all these frames over time, and allows querying for the current cumulative transformation from one frame to another. This is useful for example getting the relative position of controllers in relation to your shoulders, which allows for an arm-extension technique.

2.7 OGRE

The 3D graphics engine used in RVIZ and our Vive plugin is called Object-Oriented Graphics Rendering Engine (OGRE) [14]. It provides an abstraction of nuances between different implementations of 3D APIs like Direct3D and OpenGL, and operating systems. The design follows the principles of encapsulation and polymorphism to ease state management and provide context-specific actions to be done in a graphics engine.

It is split into two version, with the newer one supporting abstractions of much newer render systems. Because ROS follows a design policy of long software support, many of its components then also use older frameworks, RViz is limited to older CPU bound OGRE version. All plugins created in RViz inherit all instances of OGRE's manager classes, which provide abstractions for programmatically creating new objects in the scene, viewports and textures needed for rendering to the HMD. It also provides us a with a full complement of support classes for matrix and vector multiplication, helpful for camera and controllers manipulation.

2.8 OpenVR and SteamVR

OpenVR is a cross-platform open-source VR API for use with many headsets, and is developed by Valve to support their SteamVR gaming application. Due to its openness and support from its creators, OpenVR actually allows the use of almost any of the major high-end headsets by providing access to pose estimates and triggers of HMDs and their accessories. The API is implemented as a set of C++ interface classes, full of pure virtual functions, whose implementations by headset's drivers. When an application initializes the system it will return the interface that matches the header in the SDK used by that application. The rendering to the displays is then handled by a compositor provided by SteamVR. By choosing to develop our RViz plugin with this API, we will be able to run it all of those platforms with very little modifications, although all our testing has been done only with the Vive

HMD. The API also provides support for many rendering systems, although RViz's OGRE instance limits us to use of OpenGL.

Chapter 3

DJI drone MAVLink support

3.1 Related work

Due to the great advantage in portability when using mobile devices compared to computers, there are existing solutions [7] on designing a ground control station based on Android Operating System. It communicates with UAV via TCP/IP by using an Ethernet connection, through which periodic packets are sent with GPS coordinates, accelerations and speed data. This data was then displayed on the Android device with as Compass, text fields and an Open Street Map interface. The drawbacks are that the UAV is still managed with a standard remote control, since the data link only transmits telemetry data and no commands. The transmission also only operates on a single statically defined message definition, since the goal of this study was showing the possibility of using a mobile device as a GCS. They concluded that this kind of system will become a potential instrument in UAV-related systems.

In the [3] study an overview of the various possible architectures for communication between individual or groups of unmanned vehicles and their use for the management of underwater drones has been done. They were dealt with within the domain of Flying Ad-Hoc Networks (FANET), which are suitable for use on classic mobile networks at low speeds and proposed the use of on-demand networks and location protocols. The use of UAVs in a network allows them to perform their tasks with better awareness that could be helpful in the decision making in missions, and such are expected to be used in many more important applications.

We have looked through open-source solutions involving MAVLink as the messaging medium, with MAVROS [16] standing out because of its semi-official support and active community development. The aim of this ROS package is to provide support for MAVLink extendable communication between MAVLink enabled autopilots and GCSs, and a computer running ROS. Its goal is to support all autopilots whose message definitions can be found in the standard MAVLink dialect set by being a bridge between ROS and MAVLink. We were first looking into using this package to parse messages on the drone, since on the first look it seems to fit our problem area and the scant documentation did not make the purpose obvious. The problem is that it is made to support already MAVLink enabled autopilots by parsing their messages to their ROS equivalents and vice versa. They do take care of conversion between different reference frames used by ROS and MAVLink, and provide many utility functions and a plugin system, which we later adopted for our purposes. The nodes provided do also support communication with a GCS but only by forwarding the MAVLink messages directly from the autopilot, hence basically behaving as a proxy. Therefore the shortcoming is they do not cover communication between a GCS and a ROS node directly, with MAVLink behaving only as a medium of exchange. This is a problem, because DJI's drones use their own SDK to send commands to the autopilot and as previously stated, have no innate support for MAVLink. What we required is the mobile device being able to send orders to the drone and receiving telemetry data back from it. This necessitated creation of a standalone project described in the following subsection.

A popular open-source GCS we looked at is QGroundControl, which provides the operator a simulation environment for flight control and configuration of multiple UAVs, although it's made to work only with ArduPilot or PX4 Pro powered vehicles. To provide the possibility of fully integrating our application with it as a plugin, we designed our application to be built with a similar coding style as used in MAVROS, which already has an implementation of a bridge to it. The mission designer allows building of complex missions with multitude of task, saving them and repeating them on will. Extending the existing framework with an automated mission planning as shown in [20] is very useful for increasing the autonomy of UAVs and reducing operator workload. It shows design and development of a system for replanning and integration of an automated mission planner inside QGroundControl by extending on its interface for automated mission planning. This work shows a way of adding new functionalities to this actively developed GCS. Future work could be done on integrating our communication protocol for DJI drones by using the Wi-Fi and RC connection interface used in our Android application described in later sections, which would be a great boon to future of open-source development on DJI drones. Recently there has been a similar contribution [21] integrating QGroundControl with DJI drones, by use of functionalities provided by the SDK. The wrapper unpacks commands sent from the GCS to the drone and calls appropriate functions from the SDK. The limitations are that it's only limited to the feature set provided by DJI and provides no way of sending custom data over the Data Transparent Transmission. The connection is also only possible over TCP/IP.

DJI has provided their official solution for connecting an onboard embedded system with a GCS using mavlink protocol [6]. Dji2mav package is designed as a library and can be included in various platforms. It is implemented in C++ and depended on the MAVLink library. It connects the onboard computer with GCS over UDP network protocol, where it decodes and encodes MAVLink messages. It has access to the autopilot via DJI's Onboard SDK ROS, which allows it to get data from the drone's sensors, execute missions and follow waypoints. The Onboard SDK has since got a complete rewrite, and we could have only used this package by downgrading autopilot version, since DJI announced they won't be updating the project anymore. It is also written in a way that wouldn't allow new message definitions or adding functionalities easily, and so we choose to build upon the better structured MAVROS codebase.

3.2 Connection options

We provided two ways to transmit MAVLink messages between the onboard computer and the GCS. The one we focused on was transmission via the RC, which is more useful in cases where a stable long-range connection is required, and sacrificing bandwidth is not an issue e.g. sending measurements and flight data. The Mobile SDK also provides a separate channel for receiving a video stream from the gimbal camera. With that and the ability to transmit custom data packets with MAVLink messages, the framework is useful in most use-cases, especially for longer range flights in outdoor environments.

When high bandwidth is required, such as when needing to transmit big packets of data for processing on a GCS, the connection can be instead established over TCP/IP, most commonly over a Wi-Fi connection. This can be done with a router as an intermediary between the mobile device and the drone, with higher grade ones still being able to offer decent range. Alternatively we can directly connect them with a mobile hotspot. In both cases the drone also requires a Wi-Fi module to be able to use that functionality. The common use-case would be for indoor flights, where a number of routers with a common service set identifier (SSID) can be used to extend range over multiple rooms.

While testing we also used a setup, where my laptop hosts a hotspot, which the mobile device connects to. The laptop is meanwhile also connected directly to the onboard computer with a Ethernet cable. While this obviously can't be used for field flights, it serves its use for testing the Android and ROS projects. Android Studio IDE supports directly debugging applications on a mobile device over a TCP/IP connection. Since QTCreator does not support a similar feature for onboard computers, debugging was done by logs accessed over Secure Shell (SSH).

3.3 Creating a MAVLink dialect

We created a new message definition set called a MAVLink dialect. The messages there are similar in structure to ROS messages we will need to run the flight mission and they all use ROS' timestamps, which are important for synchronizing the execution of topics. They allow us to send position and rotation commands, switch between different flight modes and transmit flight data to the hand-held device. We also created a file transmission protocol, which provides fallback mechanisms to try to provide transfer guarantee. We should also note the difference between packets and messages in our context. The packet sizes are limited to 100B because of the DJI's Data Transparent Transmission, but the MAVLink messages are usually bigger than that, which are in such case split into multiple packets and sent piecemeal. Therefore we have to keep in mind we can get out-of-order/corrupted packets of different messages when reassembling them on the receiver's side.

One example of a message definition and its enum is provided here 3.3. This one corresponds to a common ROS message PoseStamped for setting a setpoint or receiving a drone's position [23]. We tried to provide a similar structure to messages in our dialect to allow much easier parsing and forwarding of MAVLink messages to ROS topics/services and vice versa. It also does not require conversion of data from NED coordinate system to ENU or any such similar challenges, which is a common problem when trying to work with the common MAVLink message set.

```
<enum name="DJI_TOPIC_NAME">
  <description>
    An enum holding the combinations of topic names and header frames.
  </description>
  <entry value="0" name="DJI_TOPIC_NAME_SETPOINT">
```

```
        <description>Setpoint.</description>
  </entry>
  <entry value="1" name="DJI_TOPIC_NAME_DRONE_POSITION">
        <description>Drone position.</description>
  </entry>
</enum>
<message id="50004" name="DJI_POSE_STAMPED">
  <description>Set position in pose stamped format.</description>
  <field type="uint8_t" name="target_system">System ID</field>
  <field type="uint32_t" name="sec">
    Timestamp - Seconds since epoch
  </field>
  <field type="uint32_t" name="nsec">
    Timestamp since epoch = sec + nsec * 10^-9
  </field>
  <field type="uint8_t" name="topic_name" enum="DJI_TOPIC_NAME">
    See DJI_TOPIC_NAME enum
  </field>
  <field type="double[3]" name="position">Position</field>
  <field type="double[4]" name="orientation">Orientation</field>
</message>
```

As for the message definition itself, it first defines its *id*, which has to be unique, and the name of the message. The *description* is there to describe the purpose of the message. A *field* can have many different types and to have it lightweight, a developer should try and select one that takes the least necessary bytes. Currently it is not possible to use inheritance or composition to combine messages together. Arrays can be of any size, as long as they do not surpass the maximum message size, but variable length is not possible. As a kind of compromise all trailing zeros in a message, which usually come from not filling in the last array to its fullest, are trimmed. *Enums* can be also separately defined and shared between multiple message definitions and the value transmitted is a char. Most messages also define a field with an

ID of the targeted, and sometimes also of the sending, drone. This is used to ignore messages not meant for a particular system. A message without a target ID is used for broadcasting.

To create source code for the MAVLink library we used [15] this code generation tool created in Python, which provides command line and GUI utilities. It uses the message definition files to provide code for your projects in the specified languages. After any message definition change this has to be rerun for both C++ and Java to provide synchronized message parsing. The C++ source files are also automatically generated when compiling the MAVLink library by providing the XML file in the same directory as all other message sets. To have it setup for usage as a package in the ROS system the compilation is done with catkin. Since Java is currently not officially supported for MAVLink 2, we have found two outstanding bugs that prevented successful message parsing when receiving on the drone, but we have managed to narrow down the problem and report the solution on the code generator repository. With those fixes we could run the Python generator, which creates appropriate Java classes for all message definitions.

3.4 File transfer over the link

We defined a sub-protocol for sending files across the link, supporting guaranteed delivery of messages by resending lost pieces. To perform this action the user first has to select a file he wants to transfer over the link on the mobile application. The request for transmission is then sent to the drone, where it is accepted unless there are problems creating a new file. When the approval is received, the Android application starts sending file parts, packaged into MAVLink messages, which are then reassembled on the drone's ROS node 3.1. When the last piece has been received on the drone or a timeout happens while waiting for a new piece, we try to find lost pieces. If there any missing, we request the pieces in intervals 3.2 instead of each separately to decrease the number of necessary messages. This procedure repeats

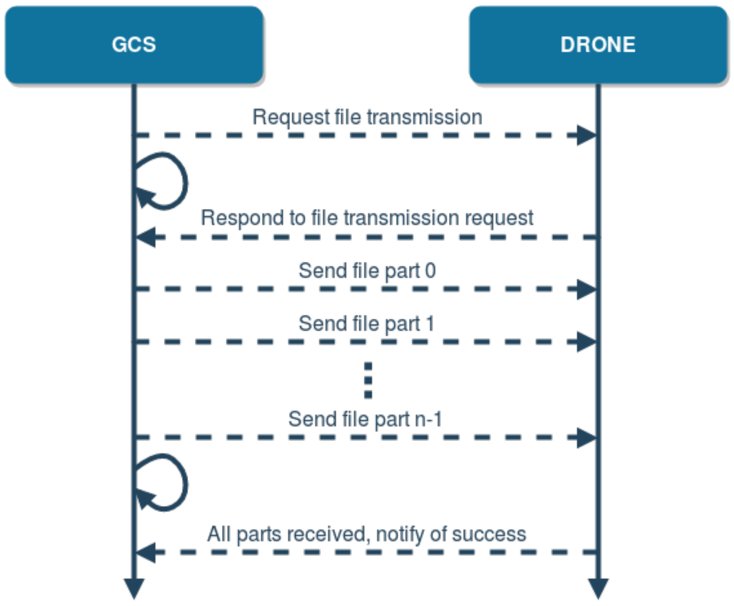

GCS
DRONE
Request file transmission
Respond to file transmission request
Send file part 0
Send file part 1
Send file part n-1
All parts received, notify of success

Figure 3.1: A diagram of file transmission when no messages get lost.

itself, until we have received everything. At that point the file is saved on the onboard computer's disk and we send a message, which tells the Android application that it may cease waiting for new requests.

One limitation of this file transfer protocol is the very limited bandwidth available, which means a 1.5MB file takes about an hour to be sent. This still makes it useful for sending mission parameters files, which are usually saved in YAML format in ROS, because such text files are usually a few KB in size. With our lost parts retrieval we provide a transmission guarantee unless the link is lost for a longer time.

3.5 Android project

The Android application running on the tablet uses the DJI's Mobile SDK to interact with drone using the RC. To be allowed to access the drone you must first register as a DJI Developer and specify the intentions of your application to get an authorization code. The easiest way to integrate the SDK is to use the Gradle build script to import it over Maven. To get a reference to the

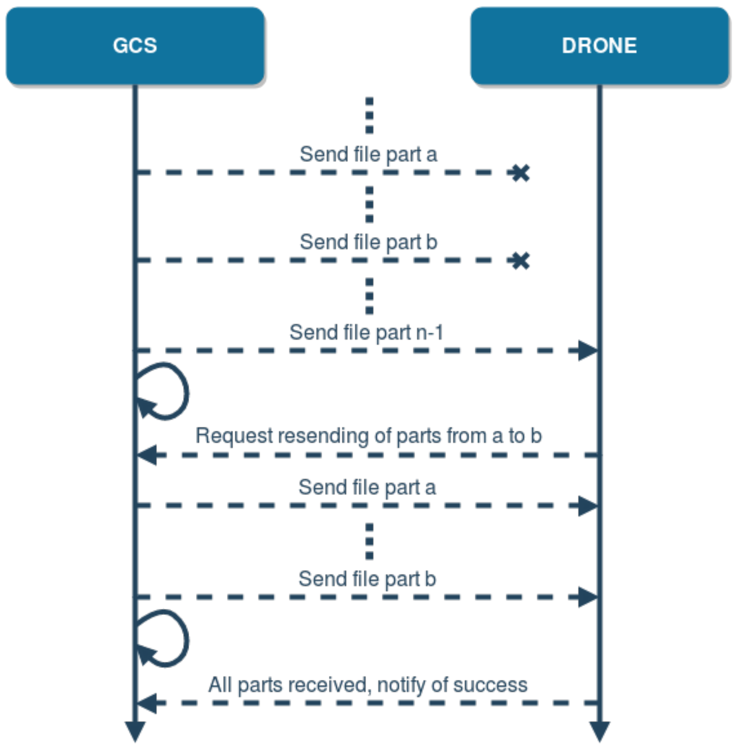


Figure 3.2: A diagram of file transmission, where parts of the file from number a to b are lost. To complete the file the drone first has to make a request for those missing parts.

drone, we use the DJI SDK manager to register the application. If approved, the application tries to connect to the product, in this case the RC, which sends an inquiry about the specifications to the drone. On success we get a reference to an object representing the correct model of the drone, in our case the DJI M100. If the connection is lost at any point, the application waits until it can be reestablished. We are mostly interested in two callback functions provided by the SDK, one for receiving data and the other for sending it. Both of them require you to provide a function, which fires when a transmission goes through successfully. Because there can only ever exist one function, which is referenced to as the callback function, in both cases special care had to be taken to create some kind of a queuing system. To get source code and the parser for MAVLink messages we use the code generator 3.3 and copy the generated Java code to our project. A simplified representation of the whole workflow can be seen in this diagram 3.3.

To allow field testing of the framework we created a simple graphical user interface (GUI) 3.4. It provides buttons for changing the drone's setpoint by changing its position in the 3D space and changing its orientation. There are also commands for the drone to take-off, hover on place, fly on the previously plotted trajectory and land on the spot. At the same time we receive drone's GPS and local position, and odometry data. We can also start a transmission of any file from the Android device on the fly while also showing the progress of that action.

3.5.1 Receiving data

The callback for receiving data triggers when any packet, that passes the DJI's checksum test, is received. The function's parameter then provides the payload as an array of bytes, which have to be parsed to receive any meaningful data. To achieve that we use the helper class provided by the MAVLink Java code generator, which takes a byte at a time and returns the current progress of message regeneration, which tells us which message's field was received last. The first value this objects expects is the magic number,

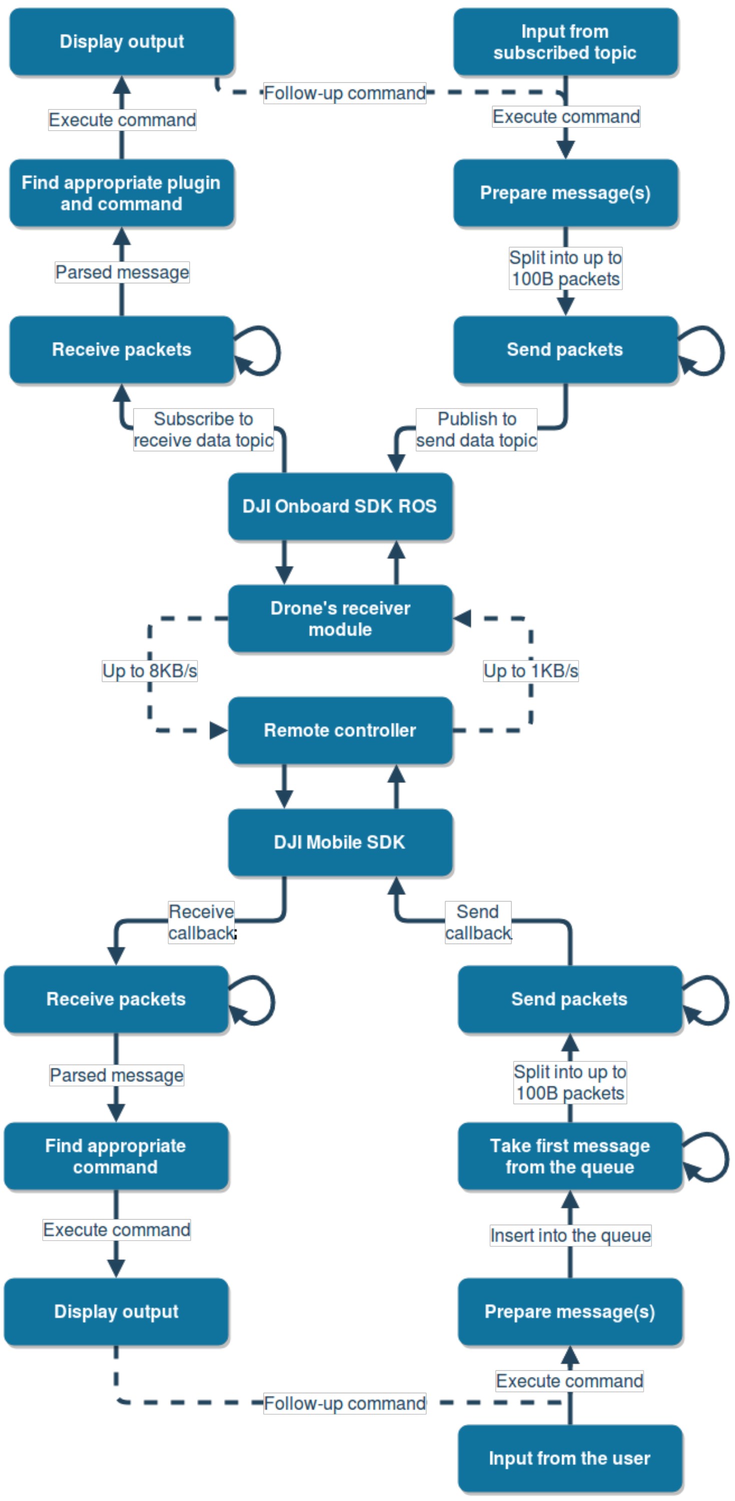
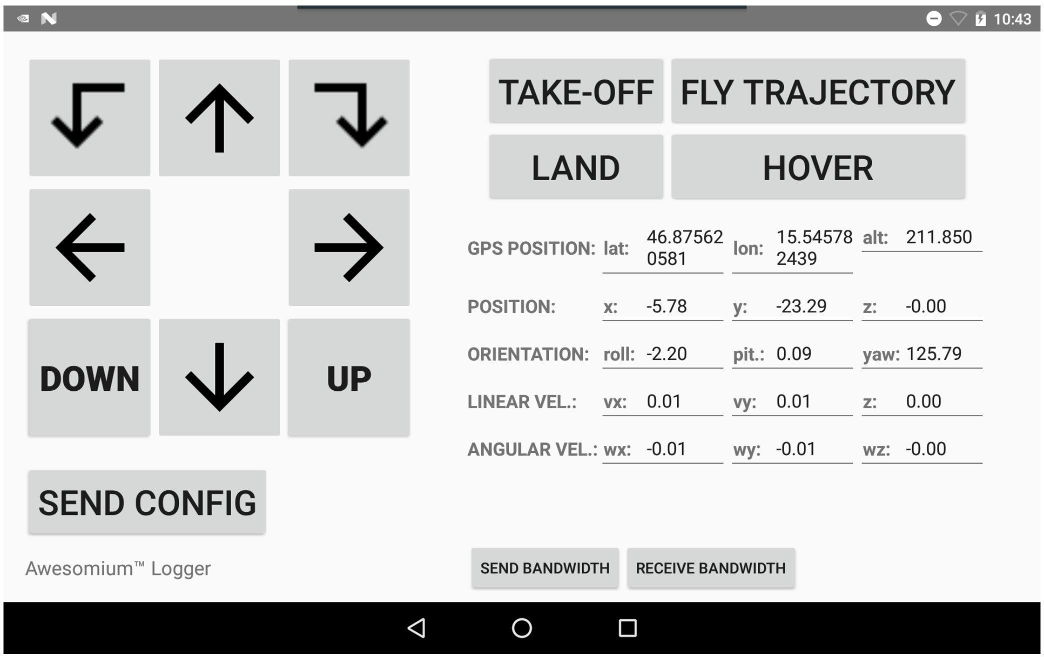


Figure 3.3: A diagram of the workflow on the drone and the Android application.



GPS POSITION: lat: 46.875620581 lon: 15.545782439 alt: 211.850
POSITION: x: -5.78 y: -23.29 z: -0.00
ORIENTATION: roll: -2.20 pit.: 0.09 yaw: 125.79
LINEAR VEL.: vx: 0.01 vy: 0.01 z: 0.00
ANGULAR VEL.: wx: -0.01 wy: -0.01 wz: -0.00

Figure 3.4: A screenshot of the Android application running on the tablet used for testing.

which tells us the MAVLink protocol version. If an error happens, the parser resets and waits for an arrival of a new message while ignoring any leftover bytes. The error can happen if the packets are received out of order, the message is are bigger than 100B in size, if the MAVLink's checksum fails or the authentication key can't decrypt the message. When a message is finally reassembled, we use a type of an object called a Subject, that emits the most recent item it has observed to each subscribed command, which use an Observer to receive that.

3.5.2 Sending data

To decide which message should be sent at a given time we use a first in, first out (FIFO) queue on a background thread. This queue holds commands, whose task is to do an operation that also involves sending data to the drone. If there are none available at a particular moment the thread sleeps. Otherwise the first is taken out, which then generates its corresponding MAVLink message and encodes it into a byte array. This is then split into packets of size up to 100B, and sent one after the other with the help of a send callback function. If the RC successfully transmits out all packets, the command performs its on-success callback and leaves the queue.

3.5.3 Command pattern

The code that reacts to and creates MAVLink messages is structured in a command pattern, which is used so as to facilitate easy addition of new message types and their corresponding commands. The abstract class that is the parent to all commands is called a CommandMessage. It requires a generic class that extends the MAVLinkMessage, which in of itself is an abstract class for all specific MAVLink messages, and handles encoding of said messages. The most important abstract function is execute, whose implementations are provided by abstract classes Send- and ReceiveCommandMessage, which are children of the aforementioned parent class. It also provides wrappers for calling the execution of the command asynchronously and/or with a timeout. When a command successfully finishes it will also trigger a callback function if provided, but should it fail a fallback function will be called if provided. All child classes, which provide implementations for commands, have all message fields in their constructor, because messages may require multiple repackaging should their transmission fail. If a message requires a follow-up command it has to provide the necessary code in the on-success function, and fallback mechanisms in the on-failure function.

The SendCommandMessage's execute function adds the current command to the FIFO queue, which runs in the background. If the command is unsuccessfully resolved, it requeues itself up to four times, but should this not be sufficient its rollback function is called. This class also takes care of generating and encoding the correct MAVLink message, that is filled with the correct sequence number and all its payload data. All fields have to be defined in the child implementations. All children then have to provide an implementation for the function which creates and fills a MAVLink message with all the specified fields.

ReceiveCommandMessage on the other hand subscribes to the RC in its execute function. This generates an Observable instance that waits to receive the correct message. Before a subscription callback could trigger, we filter the message by the msgid and additional predicates that a command can individually define. These predicates usually use sysid and compid to verify if a particular message was really meant for it. Children then have to only provide a function which fills in the fields of the command.

What we managed to achieve with all of this is a very responsive application, that does not block the UI at any point and prevent the user from performing new tasks. As an example lets say we performed an action to send a big file to the drone. The system won't hang and be unable to perform any other commands, because we generate a new command to send a new part of the file only after the previous one resolved itself, allowing a new user's input to come into the queue before it. This could prove useful because it allows flying the drone around even while transmitting new mission parameters. Any UI updates are also done on a separate thread, again as a queue. Consequently the command that issues a draw request does not have to wait for it to be resolved, but can continue with its execution. Adding simple new messages is also quite trivial, since it only requires defining a mapping function for the input values.

3.6 ROS project

The prerequisite is compiling the MAVLink framework as has been described in this subsection 3.3. This allows us to use it as a ROS package, which we overlayed over our own to get the necessary header files. Additionally we use MAVROS, since it provides some utility features for dealing with MAVLink messages easier, although that was not strictly necessary. To not have dependency issues, the packages have to be on very specific versions, with further issues of them being prepared to work with the newer ROS Kinetic, which required updating a number of other dependencies by hand.

Our package was initialized inside a standard catkin workspace. We created a ROS node, which shares the group name with other nodes running on the onboard computer. It takes care of node handler initialization and creates an instance of a class we called TransparentTransmissionAPI (TTApi). This TTApi instance provides a way to use a MAVLink bridge over the RC with Data Transparent Transmission or TCP/IP as needed, creates a MAVLink message parser and initializes two classes that take care of receiving and sending messages. Its work loop sleeps according to the loop rate, whose different values we researched later on 3.9.

3.6.1 Plugin system

What happens, when a MAVLink message or a ROS message is received, is defined in our plugins. Their job is to initialize subscribers and message sending functions. They can be freely swapped around to provide different feature sets or to handle different MAVLink dialects. They are managed with a ROS package pluginlib, which dynamically loads plugins using the ROS build infrastructure. They are registered and initialized by a plugin provider according to the contents of the *package.xml* file in our TTApi package, which imports the specified MAVLink dialects. All plugins have to extend a parent plugin class, which handles initialization, MAVLink message subscription management and handler registrations for the decoded messages. The handlers for messages are created by binding them with with the MAVLink message's unique id, while subscribers are a usual ROS one. This is useful for modifying and extending the framework's behavior without the need to know, how the rest of the code operates.

3.6.2 Adding new functionalities

By using the plugin system and the subscription pattern, we allow a very straightforward way of adding new functionalities. We do so by creating a new C++ source file, whose name we add to the *icg_plugins.xml* file, which registers the new plugin in our TTApi. After that we can use any preexisting plugin to replicate the class structure. A developer can then easily add new subscriptions to topics the usual ROS way or add handler functions, which respond to newly added MAVLink message definitions. Since the rest of the system, according to our testing, seems to be robust, we remove the need to consider the operation of the rest of the code when adding new functionalities. This should allow an ease of adding code for more field testing of new algorithms and solutions developed at ICG or elsewhere with the help of the onboard computer and the tablet.

3.6.3 Sending data

We usually send out MAVLink messages, when we receive input from the ROS system, such as telemetry data from the drone. The subscriptions to ROS topics are defined in the plugins, and each one has a callback function associated with it. They are responsible for initializing an appropriate MAVLink message and using the ROS message to fill in the fields.

This is then forwarded to a sender class which serializes it into a generic MAVLink message and gives to a parser, which packs the payload into a byte array. This array is further split into packets of size up to 100 B, which are used as a parameter for a ROS service client provided by the Onboard ROS SDK, on a topic called send_data_to_mobile. The autopilot then handles the sending of the packets to the Android application over the receiver module. Unlike with the Mobile SDK we do not get any feedback if the transmission was successful or not, since they made the function just always return true. If we were able to know, how fast the autopilot can handle packet transmission it would probably make sense to create a queue system similar to the ones used in the Android application.

3.6.4 Receiving data

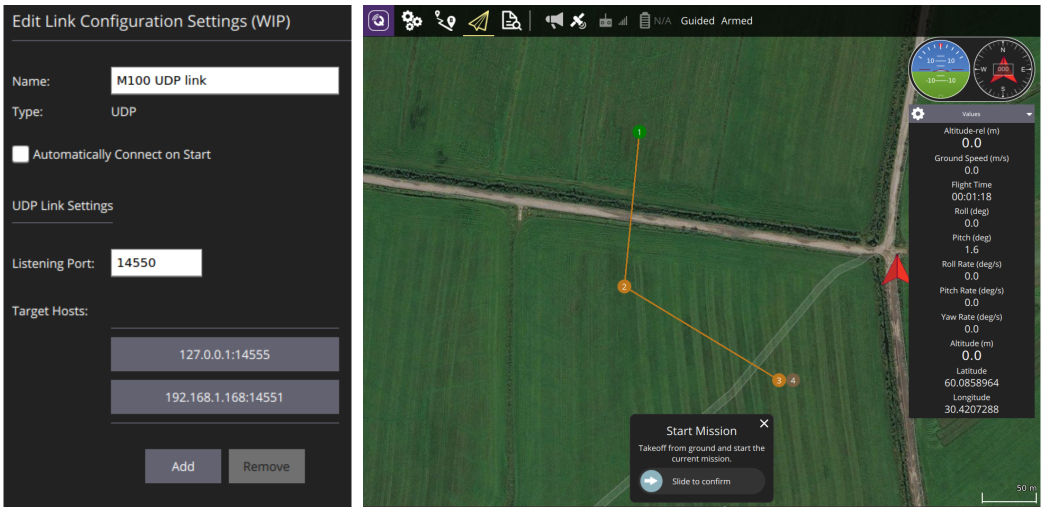
We created a class that has a subscription to a topic provided by the SDK, which gives us byte arrays whenever a new packet arrives. This packet is then fed into the parser byte by byte, until a complete MAVLink message can be retrieved, a parse error occurs or simply more packets have to arrive before a full message can be assembled.

What follows is a call to a handler function that uses this message type, denoted by the msgid. This is done by the subscription management of MAVLink messages provided in the base plugin class. Each a handler function uses a template, where the first argument is a pointer to a raw message and the second is the MAVLink message we receive from the parser. The callback function then performs the necessary actions such as flying to a position or landing, or in the case of file transmission slowly assembles the data over many messages. We can potentially also have multiple handlers that receive the same message, and each one performs the task defined in their plugin.

3.7 Integration with QGroundControl

As we managed to provide a proof-of-concept for our MAVLink framework, we also looked into a possibility of integrating it into an existing GCS. The advantages that would provide is using regularly updated software, already used by the community, with all necessary features for drone flying implemented. Our choice was QGroundControl, since it already supports MAVLink, and is uniquely cross-platform.

It works by giving the running QGroundControl application a MAVLink System ID, which it then uses as an identifier in MAVLink messages it sends out. This identifier should be unique in case of using multiple GCSs. QGroundControl only supports TCP/IP, requiring a UDP or TCP bridge


Edit Link Configuration Settings (WIP)

Name:	M100 UDP link
Type:	UDP

Automatically Connect on Start

UDP Link Settings

Listening Port:	14550
Target Hosts:	127.0.0.1:14555
	192.168.1.168:14551

Altitude-rel (m)	0.0
Ground Speed (m/s)	0.0
Flight Time	00:01:18
Roll (deg)	0.0
Pitch (deg)	1.6
Roll Rate (deg/s)	0.0
Pitch Rate (deg/s)	0.0
Yaw Rate (deg/s)	0.0
Altitude (m)	0.0
Latitude	60.0858964
Longitude	30.4207288

Start Mission

Takeoff from ground and start the current mission.

Figure 3.5: A sample configuration we used to connect the drone over TCP/IP to QGroundControl and a map view with a mission inside the application.

to connect to a drone 3.5. For our proof-of-concept we simply connected my laptop with the onboard computer with a Ethernet cable, and used it's network IP.

As part of the MAVLink protocol, every vehicle is supposed to send out a Heartbeat message every second, which identifies it's capabilities by an enum and shows a continuous stable connection. By having the onboard computer send one out in a loop, the QGroundControl recognized our drone as MAVLink enabled generic autopilot with full feature set. We submitted simple telemetry data, which shows on the right hand side and would allow to track the status of the drone. The application allows us to plan any kind of flight mission, and sends out the commands as a series of MAVLink messages upon execution. Those are encoded as MAVLink Command messages, which are executed step by step and require specific acknowledge answers from both sides. The data is also structured in a way that differs a lot from the way DJI ROS handles them. MAVROS provides some plugins for handling conversion from ROS to MAVLink style, although we would have to prepare our own to do it the other way around, which would be a big workload and out of the

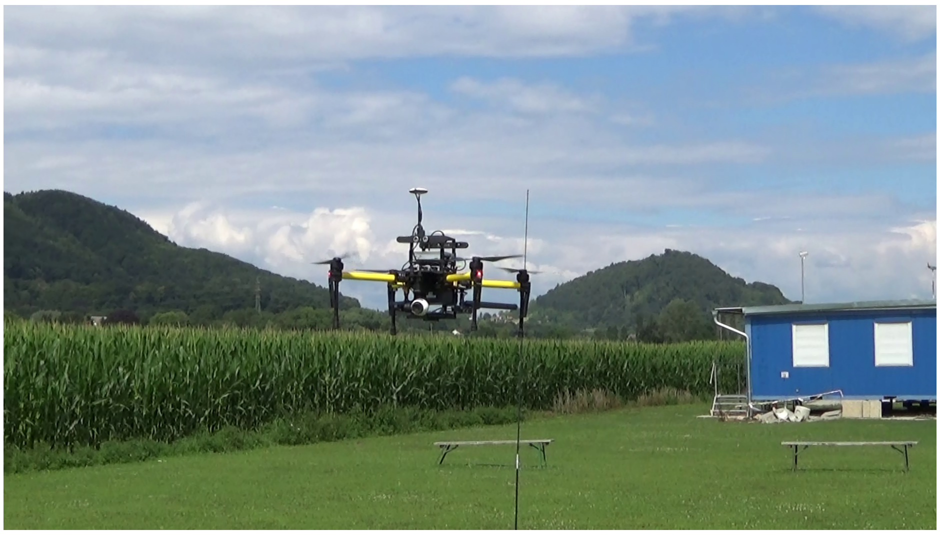


Figure 3.6: An image of the drone flying during our field test.

scope of the project. Still, we showed the framework is capable of providing a bridge between the ROS ecosystem and the open-source enabled MAVLink GCS.

3.8 Flight test

Since the start of the project we have planned to try out our solution in an outside environment. When we were satisfied with our solution and tested it in a simulated environment, we went to a flight-field Modellflugplatz UMFC Stocking, reserved for model aircraft flying. We tried out all the different commands for movement, rotation, landing, taking-off, canceling, ordering new actions and changing between RC modes to do a kind of a stress test. We concluded it with ordering the drone to do a flight on a preprogrammed flight path. We had no problems with any of the commands not being accepted or the drone reacting with latency, and were able to constantly receive flight status data. With that field test, we have shown a real-life integration of MAVLink and ROS for DJI drones, where the light-weight Android GCS performs as expected of any other desktop application.

3.9 Throughput test

To perform the throughput test, we prepared an algorithm on the Android application, which counts the number of packets successfully sent/received and the necessary time to do so. It was done on an outside flat environment, between the buildings of the campus, on distances of 10, 50, 100, 150 and 200 metres from the drone. The packets sent ranged in size from 1 to 100 B with steps of 10 B in size. Packets with a payload of 0 B cannot be sent because the DJI SDK will simply ignore them. Every step involved sending 100 packets under the same parameters and calculating averages. Sending packets from the Android application required less measurements, because you always get a callback when a transmission is resolved. On the other hand we get no feedback on the ROS side of things, since the SDK there provides no such functionality. Because of that we tested the throughput by increasing the loop rate of the ROS node from 10 to 100 Hz in steps of 10 Hz. That then basically directly correlates to the number of packets we try to send. The problem is that the SDK starts dropping the packets, without trying to send them, if we are request their transmission to quickly, since it does have a queue to manage that.

The first figure shows the number of lost messages, when sending them to the drone 3.7. Very little packets are dropped until we reached the distance of 200 m, where the reliability suddenly plummeted. Size of the message doesn't seem to necessarily have an impact on likelihood of losing it.

As probably expected, the throughput 3.8 was decreasing by distance, even though the packets weren't dropped at first. We can notice that the distance has great effect on the amount of data transmitted. Difference between the smallest and largest distance is about fourfold.

The results suggest that the throughput would get very low if we increased the distance further, which would make the link very slow at transmitting data files with our protocol. Commands should mostly go through though, since our application retries sending the message multiple times. Furthermore, even when sending largest messages of 100 B at the smallest distance, the upload speed is only half of the advertised.

When we created the graphs for throughput of the data received from the drone we congested the results into the smallest and largest distance, and showed loop rates in bigger steps. We can observe 3.7 that the loop rate of the node, which correlates to the number of packets sent per second, has a strong influence on the number of lost packets. The measurements are very unstable, but we can see how higher loop rates have a bad impact on packet loss. With the quickest packet transmission of 100 Hz we can see around a third of them get lost on the largest packet size.

But we can see 3.8 that we can still get much better throughput with increased loop rates. The benefit of using the higher transmission rate depends a lot on the type of the message. We can afford to lose position and odometry updates from the drone, since having a higher loss rate but a higher update rate is preferable. On the other hand losing mission commands and file parts would probably slow it down, due to waiting for a repetition or recovery. Optimally, DJI would in the future introduce a way to queue ROS messages, which would eliminate most problems with the update rate. We would still need to consider not overstacking the queue with odometry or similar messages, since that would make the mission important ones be delayed.

3.10 Result

We successfully developed a robust and scalable framework for Drone to Tablet message transmission over a very low throughput RC link. Additionally, the interface was also made to support a normal TCP/IP link over Wi-Fi or Ethernet. The application was tested with a flight in an outside environment on a DJI M100 drone, where it performed with no issues.

The applications running on the Android tablet and on the onboard computer are able to reliably handle communication between each other, and both provide easy ways to add new functionalities, which should be very use-

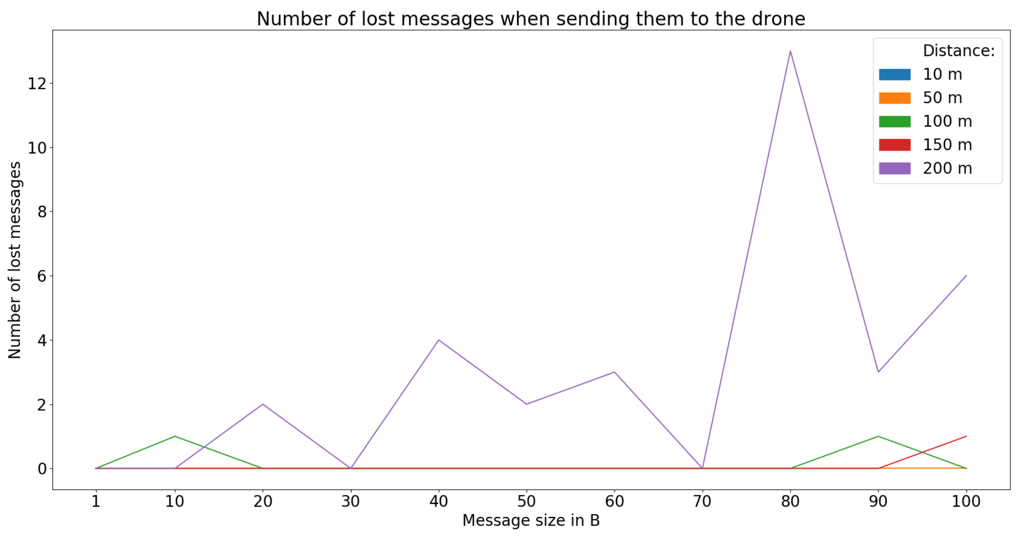


Figure 3.7: A graph of the number of messages lost out of 100, when sent from the Android application to the ROS node on the drone.

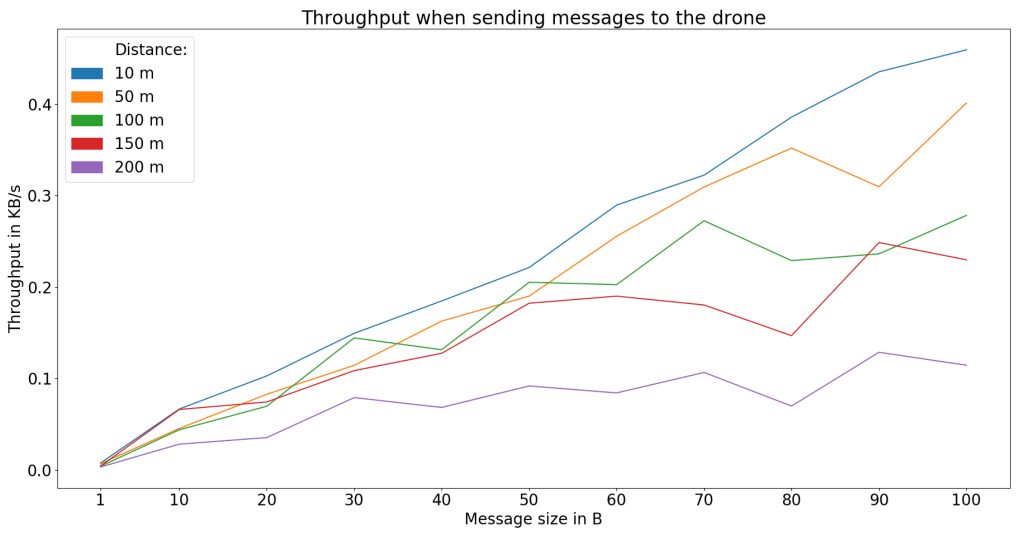


Figure 3.8: A graph of the average throughput during our field test, when sending messages from the Android application to the ROS node on the drone.

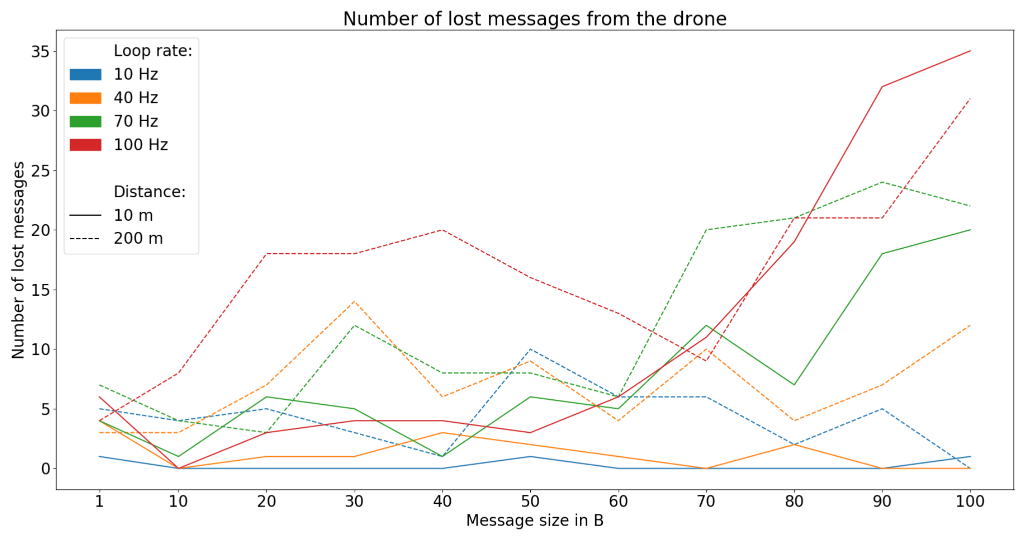


Figure 3.9: A graph of the number of messages lost out of 100, when receiving messages from the the ROS node on the drone on our Android application.

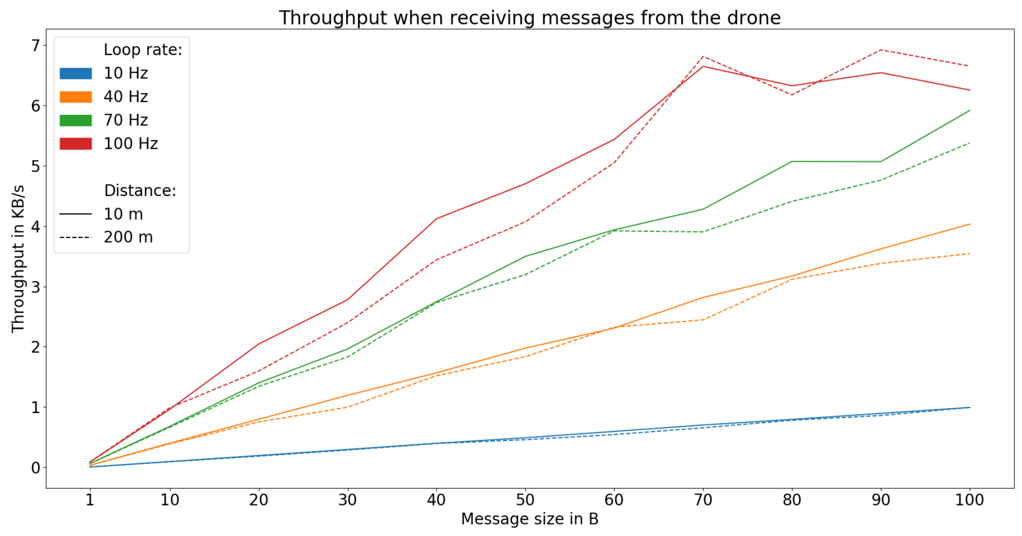


Figure 3.10: A graph of the average throughput during our field test, when receiving messages from the the ROS node on the drone on our Android application.

ful for any future testing of algorithms running on DJI drones with a ROS on-board computer. We have provided a convenient way to extend the capabilities of our Smartphone App, on top of the capabilities already provided by the DJI Mobile SDK, by the use of the command pattern. This allows easy addition of new message types, since most of them require only defining, how the data fields are filled-in. Features of the software run on the on-board computer is extendable in a very ROS-like way. To receive a new MAVLink message, you need to only add a subscription, where the argument is your desired class. To send them, you again need to only define, how the data fields are filled-in.

MAVLink as a messaging protocol turned out to be a good choice, because it offers an easy way of adding new definitions to the drone application at hand, and a very efficient payload size. With it's use in many other drone and robotics projects we are confident it will receive continuous development. MAVLink did seem to have a disadvantage of being somewhat poorly documented, especially for the changes from version one to two. Although that issue shouldn't be a problem for any later work on our applications, since there shouldn't be a need to tweak the base code for message transmission anymore.

The throughput measurements also give us insight into the capabilities in terms of data transmission speed over longer distances with varying sizes of packets. It did unfortunately show that the actual possible speed is at most half of the one advertised by DJI. Interestingly, distance did not seem to have much effect on throughput when receiving messages, but sending them slowed down by a factor of four. At the time we did not have a way to test our link at even greater distance, but it seems the Transparent Transmission Link would work

All in all, we are satisfied with the results of the project and we believe, they can be further used to at least help ICG with testing drone algorithms on a portable platform.

3.11 Challenges

The main challenges we faced during development of the framework were lackluster and outdated MAVLink and MAVROS documentation. The former also had two critical bugs that made MAVLink messages encoded by the Java interface provided by Parrot developers unreadable due to checksum mismatch, when optimization is done on removing trailing zeros. Only after days of digging through source code of three projects did we narrow it down to this fault, and we took care to report it afterwards. DJI's RC link also had undocumented behaviour on packet transmission to or from the onboard computer. This was also overcome by looking through source code and testing. Another layer of challenge came from working with different systems with different programming languages.

Chapter 4

VR-HMD ROS interface

4.1 Related work

As interaction with robots covers many disciplines with contributions from computer scientists, robotics, design and others it encouraged development of common visualization and interaction tools. This paper [4] shows an approach to animating robots and programming interaction methods for users. As they put it forward, one should look to existing animation and programming tools of a game engine to give interaction designers the tools they need. The result of their approach was an interactive tool that uses Unity for interaction and serial communication to access hardware. This does not require a middleware solutions like ROS and programming solutions by hand, and is therefore useful for people less knowledgeable in C++ or Python. On the other side we have communication interfaces implemented over a ROS bridge to game engines, such as ROS# [5] and UnrealCV [17]. Their commonality is use of Rosbridge, a package providing a JSON API to ROS functionality for prescribed 3D object for non-ROS programs via a web socket. The use of this bridge does preclude direct integration with the messaging system, a key part of ROS system. Information and commands from VR equipment therefore needs handling over proxy ROS nodes. Their advantage is the ability to use all advanced shading, visualization techniques and other tools these two modern game drives provide.

The visualization tool RViz [11], on the other hand, is graphically less complete, but already integrated into the ROS ecosystem. It is the default visualizer packaged with ROS distributions, actively developed by the community and consequently more accessible to the developers who are accustomed to it. Adding new display objects representing sensor data and state information is made easy and configurable with its GUI. It can display video streams from cameras, depth images, measurements and other data, with the possibility of extending its functionalities with plugins, all of which provides for quick prototyping.

There was already work on directly integrating HTC Vive as a RViz plugin [2]. This work itself is based on a similar plugin for Oculus Rift and Vrui VR Toolkit. Both of them are tightly knit to only work with those two headsets. It allows RVIZ to render to the HTC Vive using OGRE by providing an additional render output in RVIZ. It does that by setting up custom projection matrices for lens correction and two cameras as viewports. The textures also have to be rendered to for every color channel and eye separately, and are then applied to a material and used in shaders to combine into the final rendered image, which require extra rendering passes. It also uses a custom written driver to access headset's pose and coordinate system correction allowing the user to immerse himself in the simulated world. To display the rendered scene it requires the HMD to behave as an extended desktop, where it then creates a new desktop window with a mirror view. The issues stem from the fact that this work is based on SteamVR, OpenVR and Vive drivers from two years ago, so it already requires a careful setup by downgrading to the specified version since Linux distributions were not yet officially supported. The Linux kernel prevents newer graphics cards from using the HMD as an extended desktop, which this project relied on, but allows access to the displays only by submitting rendered frames through dedicated software, in our case SteamVR.

There had been few publications on the topic of adaptive view manage-

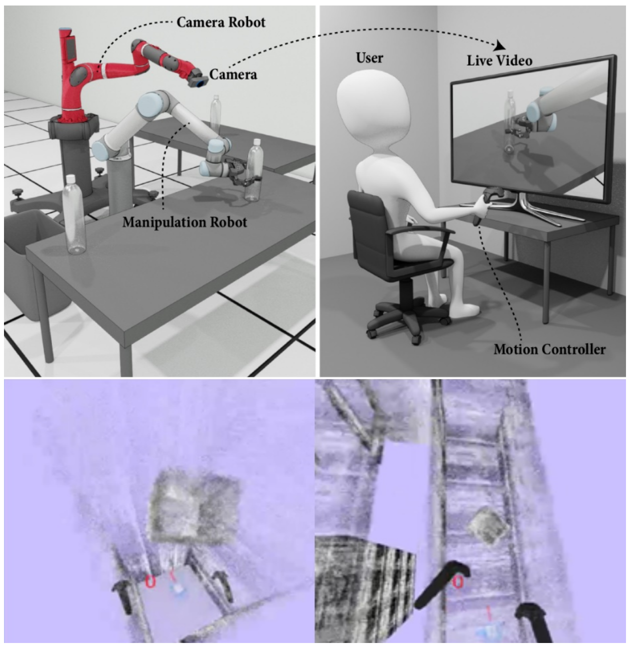


Figure 4.1: Snippets of the representation of the work done in the two papers. The upper image relates to the camera control done by an additional robotic arm [19] and the bottom to the virtual camera in an indoors drone flight [24].

ment, and event then most of them dealing with ground based robots. The older techniques of using static cameras or full 360° views seemed to be much better researched.

In a paper on steering in complex 3D structures [24], researchers tried dealing with the issue of providing a user an automatic viewpoint inside the confined spaces. The drone flight was done manually by using the Vive controllers, with people that had no prior experience. They used a virtual environment, built by reconstructing a real one as a point cloud and then visualized in the Vive headset. The adaptive viewpoint works in such a way, that during "safe" flight the camera would be following behind the drone, but when an obstacle got nearer to the drone, it would start shifting sideways to try to provide the best viewing angle to help get around it. The obstacles are detected by simple raycasting and the result then averaged over a small time span to get smooth movement. In the user study they conducted, part of the users was limited to using only the first person view, visualized as a static 2D plane in space. The second was also provided the possibility of looking at the 3D model, which they moved manually with the controllers to try and get a good perception of the scene. And the third had the possibility of switching their view to an adaptive one, which would automatically move the camera to a point calculated as optimal. The results were encouraging, with the users using the adaptive view having the least collisions and best times, since they did not have to spend as much time orienting themselves in the world.

We also looked into an adaptive view provided by a robotic hand holding a camera [19]. The idea here was that a robot operator has to perform a task in a limited space via teleoperation by using the robotic arm. The typical issue is view obstruction, especially when having to do more detailed work, because the geometry of the arm would get in the way. This is usually solved by using many static cameras, both in the environment and on the hand. The researchers here tried to provide a solution, where an additional robotic arm would be placed in the space. Their method avoids occlusions with the manipulation arm to improve visibility. Context and detailed views of the work area is provided by varying the distance of the camera to the target. They achieve that by zooming when arm movements are slow and the user is therefore probably performing a sensitive task, and vice versa. By utilizing motion prediction they follow the user's subsequent manipulation actions, and actively correct the view to avoid disorienting the user as the camera moves. The arm manipulation is done with a gamepad, where the coordinate of the arm remains static even as the camera moves. They also achieved better results than using static cameras or by having the camera be attached to the manipulator arm. They suggested the results suffered from the lack of depth cues, and recommended use of an VR headset in an reconstructed virtual environment.

4.2 Plugin system

Our VR framework is structured to be used as plugin, which we import into RViz, and allows us to interact with the scene with a OpenVR supported headset and controllers. It is managed with a ROS package pluginlib, which dynamically loads plugins using the ROS build infrastructure. It is registered and initialized by a plugin provider according to the contents of the *package.xml* file. It can then be used in any running instance of RViz by simply adding it through the visualizations selection screen, because it inherits RViz's Display base class. Extra runtime customization is provided by the use of properties that can be used to change between rendering of the virtual scene or showing the stereo camera video stream in the displays, enabling steering of the virtual camera by HMD movement or free viewing of the scene and changing the video stream topics. An overview of both the plugin startup and the render loop are provided in this figure 4.2.

4.3 Plugin startup

Including our plugin in a running instance of RViz first triggers an attempt to initialize OpenVR and trying to connect to any attached VR hardware. If a virtual reality headset is found, we request an exclusive access to rendering to its screens by trying to run the API in application mode. Alternatively we could only request read-only rights for reading out positional data, as will be useful later for controllers. If successful, the API loads all vendor specific data needed for setting up things in OGRE. We also create dynamic variables, that can be changed during runtime in RViz, and their relevant callback functions. They modify scene orientation due to headset movement, switching to a stereo video stream and changing tf listeners and broadcasters. We then continue by getting a reference to the running instance of RViz's display context, which is the main connection a Display has into the rest of RViz, providing us with access to frame and selection managers. We also get a reference to OGRE's scene manager, which handles the scene hierarchy,

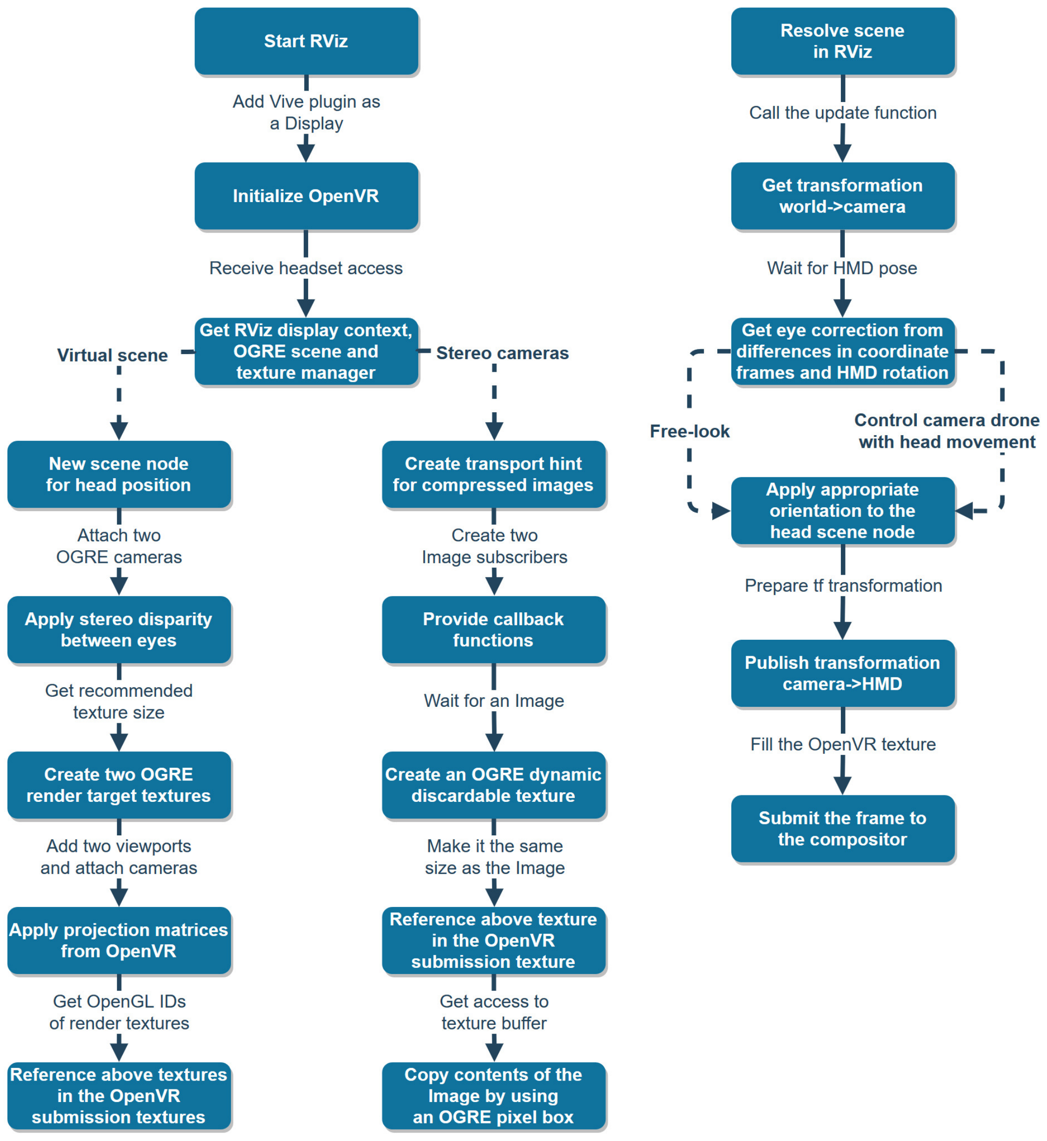

Start RViz
Add Vive plugin as a Display
Initialize OpenVR
Receive headset access
Get RViz display context, OGRE scene and texture manager
Virtual scene
Stereo cameras
New scene node for head position
Attach two OGRE cameras
Apply stereo disparity between eyes
Get recommended texture size
Create two OGRE render target textures
Add two viewports and attach cameras
Apply projection matrices from OpenVR
Get OpenGL IDs of render textures
Reference above textures in the OpenVR submission textures
Create transport hint for compressed images
Create two Image subscribers
Provide callback functions
Wait for an Image
Create an OGRE dynamic discardable texture
Make it the same size as the Image
Reference above texture in the OpenVR submission texture
Get access to texture buffer
Copy contents of the Image by using an OGRE pixel box
Resolve scene in RViz
Call the update function
Get transformation world->camera
Wait for HMD pose
Get eye correction from differences in coordinate frames and HMD rotation
Free-look
Control camera drone with head movement
Apply appropriate orientation to the head scene node
Prepare tf transformation
Publish transformation camera->HMD
Fill the OpenVR texture
Submit the frame to the compositor

Figure 4.2: Simplified diagrams of the workings of the plugin setup on the left and the render loop on the right. Both them should provide an overview over the procedure necessary to run the Vive headset with RViz.

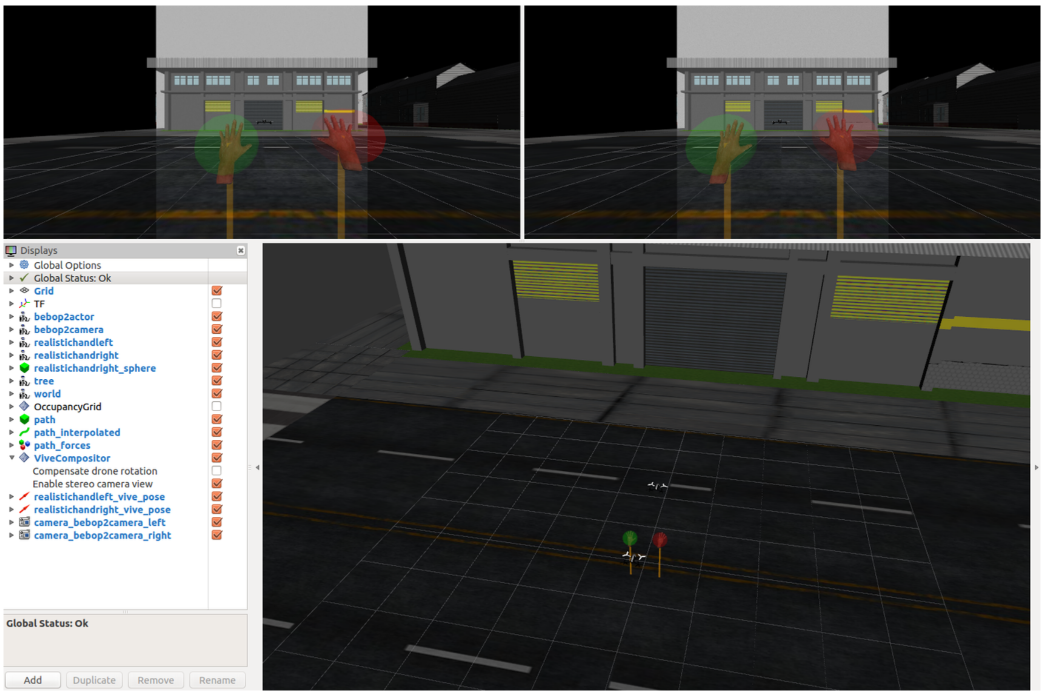
and texture manager, to later create textures that will correlate to eyes. We then have two different setups for OGRE, depending on our choice of render targets: a virtual scene or displaying a video stream from stereo cameras.

4.4 Virtual scene setup

We create a scene node at the root of the scene, which will represent the head position, and is relative to the world's coordinate frame, and we attach to it two newly created camera objects. We get the translation matrices from eye space to head space from OpenVR, which provide stereo disparity, and apply them to the eye cameras.

The API then provides us with the recommended size for the intermediate render target that the distortion pulls from. We use that width and height to create two 2D OpenGL textures with the texture manager, which act as a render target for the cameras. We follow that up by creating a viewport for each of the texture's render targets, and attach the cameras to them. These viewports represent virtual eyes in the scene, and be rendered to at every RViz's render loop. They still require their appropriate projection matrices to achieve the correct stereo distortions, and those are again given by the API. At the end of the procedure we also create two OpenVR textures, which hold a pointer by providing the OpenGL ID of the two OpenGL textures we created previously, and will be later submitted to the HMD. We also set the UV coordinates range for the textures to default values of 0 to 1.

If the user decides to switch to the stereo video stream, we get the render targets of the two texture and remove their viewports, to prevent excess rendering. We also tell the OpenVR textures to now point to the stereo cameras textures.



```
Displays
▸ Global Options
▸ ✓ Global Status: Ok
▸ Grid                              ☑
▸ TF                                ☐
▸ bebop2actor                       ☑
▸ bebop2camera                      ☑
▸ realistichandleft                 ☑
▸ realistichandright                ☑
▸ realistichandright_sphere         ☑
▸ tree                              ☑
▸ world                             ☑
▸ OccupancyGrid                     ☐
▸ path                              ☑
▸ path_interpolated                 ☑
▸ path_forces                       ☑
▾ ViveCompositor                    ☑
    Compensate drone rotation       ☐
    Enable stereo camera view       ☑
▸ realistichandleft_vive_pose       ☑
▸ realistichandright_vive_pose      ☑
▸ camera_bebop2camera_left          ☑
▸ camera_bebop2camera_right         ☑

Global Status: Ok
```

Figure 4.3: On the bottom left there is the usual Displays setup in RViz. On the top we can see video stream given by the stereo cameras, overlayed over the virtual world in the background. On the right bottom we can see the scene in the default view provided by RViz.

4.5 Stereo cameras setup

To visualize views of the drones stereo camera we first need to receive their video stream. We start by creating a transport hint, which basically tells ROS to deliver compressed images to save on the necessary bandwidth. This is then used to make two subscriptions to ROS topics, which use the same callback function with only the differentiation between the eye used.

When an image is received, we read out the pixel format and use that information to convert the image from its compressed form to bitmap with the help of an OpenCV bridge. The image is also converted from BGR color format to RGB.

If this is the first image we received for that eye, we must first create a texture for it. We read out the width and height of the image and make an equally sized texture. To get the best performance out of frequent reloading of images, we need to set some buffer settings. We make it dynamic, signalling that CPU will often modify this buffer, which then retains it in Accelerated Graphics Port (AGP) memory instead on the video memory. The write only flag tells it that we will only be writing, which is not true of course, although it makes the buffer always create a pointer to new memory, basically allowing rewriting the contents while the previous one is read out by OpenVR compositor and submitted to the HMD. Last thing we need is that it is discardable, which tells the buffer we will be often recreating the contents, and does therefore not care about losing the previous contents. Such setup is needed, otherwise it prove to be a big bottleneck because of bandwidth limitations in buffer access between the CPU and GPU. This is followed by getting the OpenGL ID of the newly created resource and setting it in OpenVR texture as a pointer.

In any case, the next step is copying the bitmap image to the buffer. This is achieved by creating an OGRE pixel box, which is basically a 2D panel. We set it to the same size and pixel format as the texture, and provide it a pointer to the byte array of the bitmap image. We then request direct access to stereo camera's texture and copy the pixel box to it. We set the UV coordinates range for the textures to values of -0.2 to 1.2, to stretch the received image over a smaller surface. This was necessary, because the result is otherwise too zoomed in and we also do not get any strange warping effects like we would when rendering the virtual scene.

If the user switches from stereo camera view to the virtual scene, we have only need to stop the subscribers to the ROS topics.

4.6 Render loop

The start of the render iteration starts by RViz first handling its GUI and main display in the background. After that the update callback function in our plugin is called, where the transformations, rendering and frame submission to the OpenVR compositor is handled. The viewports, that were attached to the OGRE textures in the virtual scenes, are already rendered to and ready to be used. This does mean there is a one frame delay before the movement of the drone and its camera is taken into account, although that was not really an issue.

The first thing we do is request for a tf transformation from the world to the pose of the virtual camera. If we are not able to get that, all other operations are meaningless and so we would skip the render frame. Although that never happens unless the tf tree is somehow broken. The scene node, representing the head, is set to the gimbal's location to simulate looking through the camera. We set that as the position of the RViz scene node representing the camera, while rotation will still require a few more steps to compute.

We follow this up with a blocking call to the OpenVR compositor, to provide us with the position of the VR hardware, which is required for the later submission function call to work. This is only resolved about 2-3ms before the frame needs to be prepared for read-out by the compositor, and is therefore limited to 90Hz with the Vive headset. We transform HMD's OpenVR 3x4 matrix into an Ogre 4x4 one by just extract its orientation as a quaternions x to z.

From this point on we had some issues with taking correct coordinate system transformations into account. All of them are right-handed, with OGRE and OpenVR using Y as the up axis, and ROS Z 4.4. When getting the rotational matrix for our headset from OpenVR, we also have to account that the orientation of pitching is inverted, to reflect the way it's used in real-life flying. To get our eye correction we therefore need to rotate around X axis by $90°$ and around Z by $-90°$.

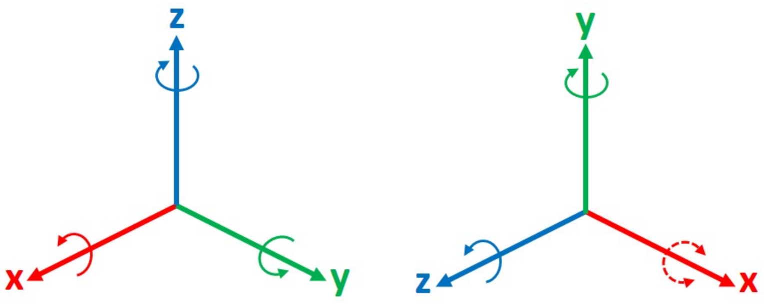


Figure 4.4: A representation of the two coordinate systems, former being used in ROS and latter in OGRE and OpenVR.

Quaternions have the following format: [x y z w]. Here we take in the rotation of the gimbal camera in our scene. We already change the direction of rotation and inverting the yaw due to the way OpenVR represents HMD's pose:

$$OgreCameraRot = \begin{bmatrix} GimbalRotX \\ GimbalRotY \\ GimbalRotZ \\ GimbalRotW \end{bmatrix} * \begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \end{bmatrix} * \begin{bmatrix} 0 \\ 0 \\ -1 \\ 1 \end{bmatrix}$$

Next we take into account plugin's parameter for switching between free-look and virtual camera movement with HMD rotation. In the former case we multiply the orientations of the camera, the eye correction and the HMD. In the latter we have to inverse the cameras yaw and multiply that with the eye orientation.

We are correcting rotations due to differences in OGRE's and OpenVR's coordinate systems. We do that by rotating around the X and Z axis by Π /2:

$$EyeRotCorrection = \begin{bmatrix} OgreCameraRotX \\ OgreCameraRotY \\ OgreCameraRotZ \\ OgreCameraRotW \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \\ 0 \\ \frac{\pi}{2} \end{bmatrix} * \begin{bmatrix} 0 \\ 1 \\ 0 \\ \frac{\pi}{2} \end{bmatrix}$$

When calculating the final rotation quaternion, which will be used in the RViz scene node for the virtual camera, we can ignore the roll in the HMD's orientation. When we are working only with a simulation, and we are using a secondary drone with the camera, we can compensate the tilt that happens due to acceleration and deceleration. This makes the flight feel smoother and causes less nausea due to shaking. This could be achieved in a real-life flight by having a 6-DOF gimbal, which would try to rotate the camera to keep the view horizontal.

We calculate the final rotation, which will be set to RViz node representing the virtual camera. . This makes the flight feel smoother and causes less nausea due to shaking:

$$RVizNodeRot = \begin{bmatrix} EyeRotCorrectionX \\ EyeRotCorrectionY \\ EyeRotCorrectionZ \\ EyeRotCorrectionW \end{bmatrix} * \begin{bmatrix} HmdRotX \\ 0 \\ HmdRotZ \\ HmdRotW \end{bmatrix}$$

We follow that up by preparing a tf frame, that will represent the HMD's position relative to the camera. We will use that later to put the controllers in the scene. That means the translation is simply zero, with orientation again needing special care. The issue here is that the controllers do not receive their position relative to the base stations, but relative to the HMD, plus again taking into account differences between OpenVR and ROS coordinate systems. We negated the values of x and y. We then shifted x, y and z by one position, because of the difference in which axis is up. So we ended up with a quaternion like this: [-z, -x, y, w]. We can then finally publish the transform to our tf tree.

At this point we are ready to display the frames in the headset. We submit them to the compositor by getting the enum value of the eye, OpenVR texture and the definition of UV coordinates range. When both have been sent to SteamVR, and if they have been sent fast enough, we will now see the result in the HMD.

4.7 Reading out pose estimates

To use the HMD and controllers to manipulate and interact with the scene, we created a ROS node to read out pose estimates. Like the plugin also starts by initializing the OpenVR API, although this time it request only read-only access. If successful, it provides us with IDs, that we later use to access data. It will fail, if there is no other instance of OpenVR running in application mode on the computer, so we repeat the initialization until the RViz plugin is running. We need to do it as such, since it is not possible to run multiple processes with direct access to the VR hardware. Although in this case this is not a problem, since we are only interested in reading out positions and button presses.

The main loop is then not using a blocking call to receive the updated positions like before, since we do not need render synchronization. We always request for new positional data, which is not necessarily updated yet, and parse it to a tf transform. We then publish them to a separate tf tree 4.5 from the rest of all the tf frames we use in other packages and the plugin. This is because Vive uses an arbitrary coordinate frame created during Steam's Room Setup, which is useful only to us to determine relative transformations from the HMD to the controllers in the real world.

Besides the translation and orientation of the controllers we are also inter-

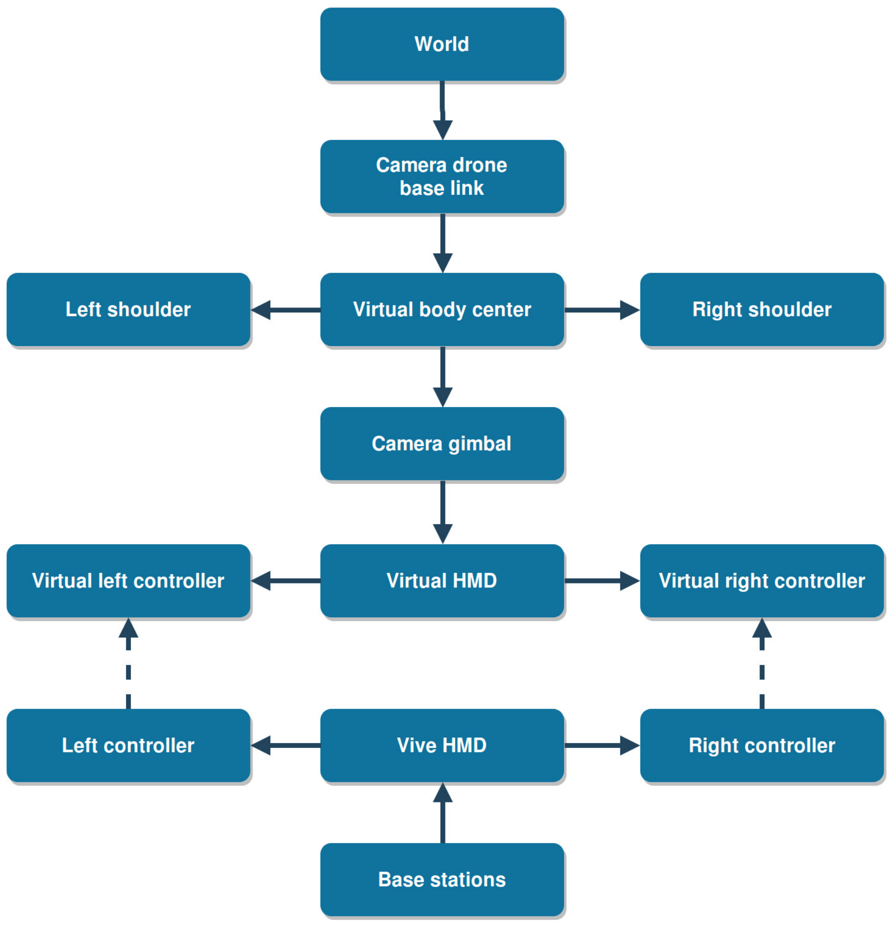


Figure 4.5: An overview of the relative transformations related to the virtual camera and the Vive headset.

ested in reading out the button presses. The 2D trackpad can take any value between -1 and 1 on the X and Y axes. The trigger button similarly goes from 0 to 1, but all other buttons just output binary values. We publish this data as a ROS Joy Sensor message, which can take an arbitrary number of button presses and axes values as input. This does mean that any receiving node needs special parsing to interpret the data correctly.

We also have a pair of ROS subscribers, to which we can publish a request for vibration. That could be useful as haptic feedback to inform the user of problems or potential failures in performing a task. To trigger it the message needs to provide only a float between the values 0 and 3999, with the highest number causing the strongest response.

4.8 Vive controllers integration

Vive controllers were an additional VR accessory that we wanted to use for interaction inside the virtual environment. They let you wirelessly interact with the scene, providing very responsive and accurate localization. It has a multi-function analog trackpad, a dual-stage trigger, grip button and haptic feedback. The sensor data for them is provided by SteamVR at the speed of 250-1000Hz. We read out and publish the data in a node, separate from our visualization plugin. This allows us to use pose and triggers in any other ROS node, without creating bottlenecks by running too many calculations inside the render loop, which would inevitably slow it down.

A separate node then handles the transformation to the correct HMD tf frame in the virtual scene. The main loop also runs at a high frequency, so it is able to very quickly update the translation and orientation of the controllers in the real world, although synchronization with the virtual scene is still limited by the speed of the render loop in the plugin, which is at most 90-times per second with the Vive HMD. We start by requesting the newest tf transformation between the headset and its controllers, where both are relative to the base stations. This is followed up by sending a pose message, where we set its header frame ID to the HMD transformation from the virtual scene. Alternatively we also offer the same information by publishing it as a tf transform. Both of them have an effect of positioning the controllers in the virtual scene relative to the virtual view. This gives then the impression of simply having virtual, highly responsive hands.

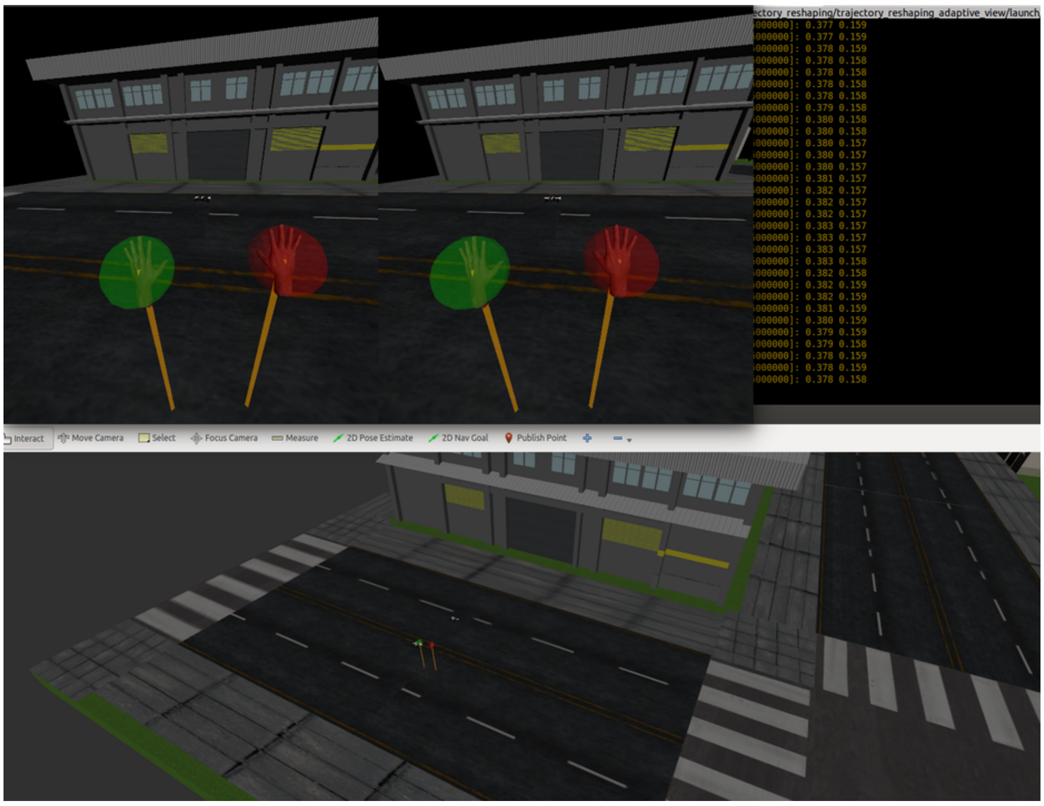
4.9 Simulation in Gazebo

To provide collision detection and physics simulation we included Gazebo as a set of ROS packages. With that, we imported the virtual scene with the urban setting, drones and interactive controllers as physical objects. If we just used the controllers as the virtual hands in the scene now, we would be only limited to interacting with vicinity at the reach of the hand. Additionally, it would also ignore any collisions with the virtual scene.

To extend scene interaction options, we used a method of extending the reach of the hands. The pose is looked up as a transformation between the world and the virtual controllers. This gives us a position to which the hands would like to move in the simulated scene. After one tick we republish these new positions, which can be the same as the original ones if nothing was interacted with, so that we can visualize it in RViz.

We also use the input of the trigger button as given by the Joy sensor message. The distance of the hand is then scaled linearly away from the body proportional to how much the trigger was depressed. The directional vector is given by the position of the controller and its corresponding shoulder 4.6. Although for that to feel as intuitive movement the user has to hold the hand up, usually at the same height as the shoulder. To start interaction with the scene we can then also use the grip button.

When we were testing out this solution we quickly realized, that even though we have stereo vision by using VR hardware, it is sometimes still problematic to judge the depth and the distance to the points of interest. Therefore we plan to add straight lines, perpendicular to the world plane, to



```
ectory_reshaping/trajectory_reshaping_adaptive_view/launch
5000000]: 0.377 0.159
5000000]: 0.377 0.159
5000000]: 0.378 0.159
5000000]: 0.378 0.158
5000000]: 0.378 0.158
5000000]: 0.378 0.158
5000000]: 0.378 0.158
5000000]: 0.379 0.158
5000000]: 0.380 0.158
5000000]: 0.380 0.158
5000000]: 0.380 0.157
5000000]: 0.380 0.157
5000000]: 0.380 0.157
5000000]: 0.381 0.157
5000000]: 0.382 0.157
5000000]: 0.382 0.157
5000000]: 0.382 0.157
5000000]: 0.383 0.157
5000000]: 0.383 0.157
5000000]: 0.383 0.157
5000000]: 0.383 0.158
5000000]: 0.382 0.158
5000000]: 0.382 0.159
5000000]: 0.382 0.159
5000000]: 0.381 0.159
5000000]: 0.380 0.159
5000000]: 0.379 0.159
5000000]: 0.379 0.158
5000000]: 0.378 0.159
5000000]: 0.378 0.159
5000000]: 0.378 0.158
```

Figure 4.6: On the top we can see the mirror view of the current display in the headset. Here we are moving the virtual camera around the drone, in a kind of an orbit by rotating the HMD. The visualized hands correspond to the controllers position in the real world, and can be used to interact with the scene. On the right bottom we can see the scene in the default view provided by RViz.

act as visual guides.

4.10 Adaptive view management

We continued the adaption of the immersive virtual environment with teleoperation that would provide a user with environment-adaptive viewpoints, which would be automatically adjusted to improve safety and smooth user operation. The idea is that the method tries to avoid occlusions with the surrounding environment to improve visibility, provide context and detailed views of the area of interest (AOI) by changing the virtual cameras distance to the drone, utilizes motion prediction to cover the space of the user's next manipulation actions, and actively corrects views to avoid disorienting the user as the camera moves.

This approach comes from the fact that visual support is necessary for smooth teleoperation, as even quality control interfaces would not help if we are dealing with an obfuscated or unclear view of the manipulation space. A default solution would be to simply use the drone's camera to observe the world in first person. The problem of that is that you always have to interrupt an ongoing task, if you wish to look around the environment to get an overview and more context, by manually moving the drone to a vantage point, which is time-consuming and an additional mental load on the user, since he would also have to memorize the layout of the world in the meantime. This is partially solved by using a virtualized world and looking it up in a visualizer, since you could look at the 3D world reconstruction in real-time, although that would again mean switching between the cameras views and manual manipulation of the view in the virtual scene. One solution for that is using a collection of static cameras arranged in a workspace, although a lot of them would probably not be able to see the AOI and are not that useful in an outdoor scenario because of the possible size of the area.

4.11 Our approach

Our plan was to provide two interfaces for visualization of the environment. One of them is rendering a virtual scene, which is useful for performing simulated flights or flying outdoors in a world, where we would not expect much dynamic changes. Such a scene could be built before the assisted flight by an auxiliary drone going over the scene, which would take a set of pictures, which would be used for reconstructing a 3D environment of the area. The second option is viewing the world through the drone's onboard stereo camera, which would provide an immediate view of the current scene, real or simulated. To increase the comprehensibility of handling the view in those two different cases we use the HMD's virtual view for both, which would potentially also reduce the complexity of the task due to depth cues.

The virtual camera's job is to perform similar view management like the virtual camera in [24]. It needs to avoid any collisions with the world and the Bebop 2 drone, while also providing a view over the interest area. Our prototype places the viewpoint in orbit of the drone, where it then stays and follows at a fixed distance. The user can intuitively adjust the viewing angle by looking around with the HMD 4.7. If we for example look left, the virtual camera will shift around the drones yaw axis, and by looking down, it will orbit it around the pitch axis. By using the natural movements of the head as an input method, the user should be able to quickly to find a viewpoint he deems necessary.

4.12 Result

We have successfully developed a VR-HMD interface for visualization in RViz. To test the solution we extensively used HTC Vive, although by abstracting the VR hardware layer through the use of OpenVR most other SteamVR enabled devices should be usable with minimal or no tweaking required. The plugin will remain usable for the foreseeable future with the newest software, since we managed to remove specific version requirements

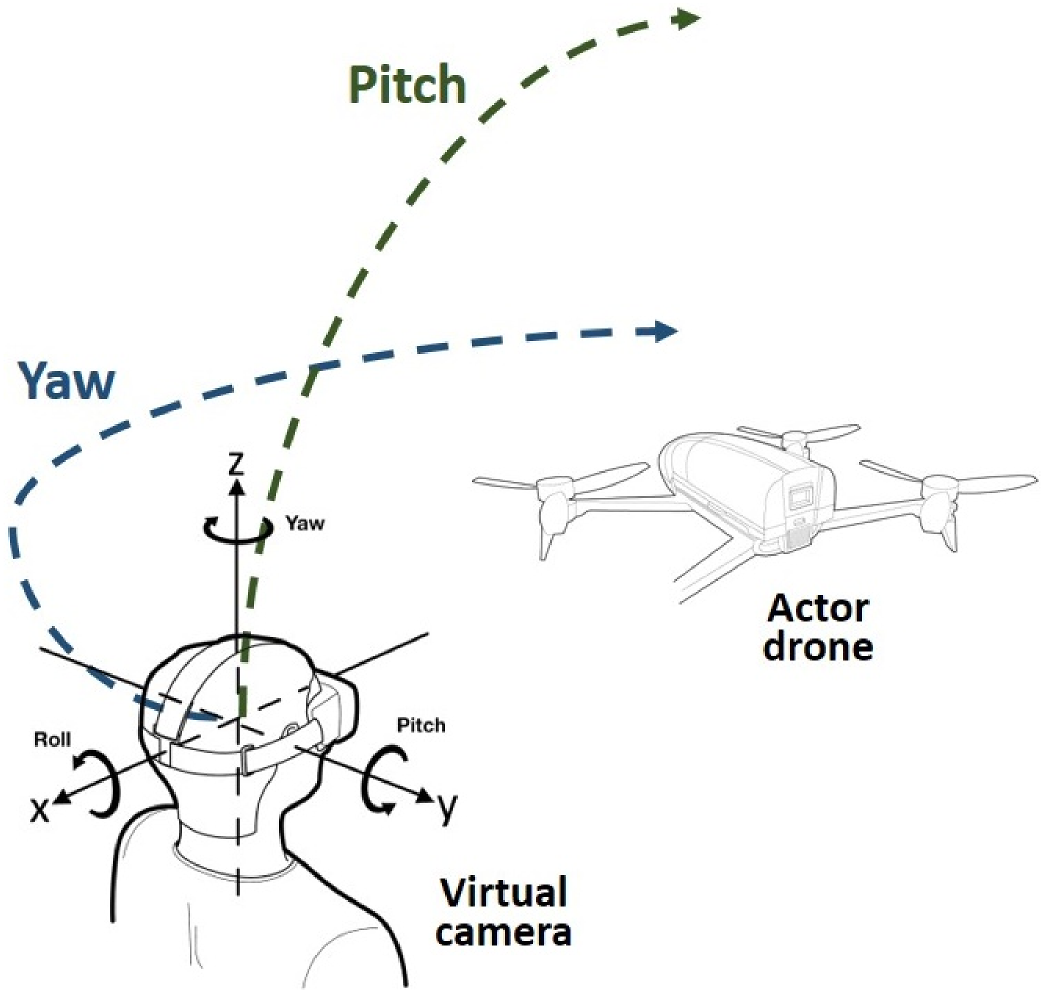


Figure 4.7: Representation of radial movements around the actor drone done by the virtual camera in relation to the user's yaw and pitch movements.

in a similar previous plugin. The performance and rendering speed manages to achieve stable recommended frame rate. Since we can visualize the virtual scene or the stereo camera video streams, it is usable both in simulated and real-world scenarios. The integration of Vive controllers in the virtual scene also allows their use for manipulation in different HRI scenarios. As a proof-of-concept we setup a VR prototype, where VR controllers and the headset is used to manipulate artificial hands and move a drone. This shows a possible use in future UAV exploration missions, where the virtual viewpoint would follow a drone in the most efficient way providing optimal viewpoints to the drones pilot.

An extension of that method is to use the drone's waypoints on its current mission and measuring the speed, which we could use to adjust the distance of the virtual camera, thereby changing the zoom level. By moving slower, that probably means the drone is approaching a critical point, where a user's attention would be then needed to perform or supervise a necessary task. During faster transit around the scene it would zoom out to provide a better overview and context of the whole environment. The next step is taking into account the hand controllers movement to allow the user to have some control over what the interest area. By moving the controllers towards the edge of the viewing frustum a user can influence the calculated interest area by shifting it towards something a user needs to manipulate to fulfill his task. The aforementioned methods are planned to be used and tested as part of a concept for a natural UAV exploration interface, introduced by my advisor W. A. Isop, replacing the virtual camera by a second physical drone and interacting with their flight paths.

4.13 Challenges

Because VR is mainly used in Windows environments, especially in gaming, there were a lot of compatibility issues and constant problems with freezing and unresponsiveness with HDM drivers on Ubuntu - usually solved by simply unplugging the headset. SteamVR's configuration interface for developers is also mostly unresponsive in Ubuntu, so everything had to be done manually via configuration-file editing. Another problem was also that the Vive's display worked as extended desktop on older graphics-mode on newer graphics cards. To conform to that we had to completely rewrite the VR plugin to use newer graphics drivers, SteamVR and OpenVR, which coincidentally made this the focus of this project. We had some issues also with specific versions of packages you have to use in tandem with Indigo, although troubleshooting such things was usually sufficiently covered in their description. Similarly to the MAVLink interface, there were issues with lack of documentation on OGRE and RViz integration, and no description on how render loops and background threads are handled. This meant surfing through the source code and blind testing was again necessary.

Chapter 5

Conclusion and future work

5.1 Concluding remarks

In this thesis we explored the possibilities of extending developer tools for ROS supported drones. We proposed and implemented a framework that allows use of custom commands for DJI drones over greater distances by using the RC and integration of VR support for most modern headsets in RViz.

We started by looking into different frameworks, used for UAV to GCS communication. We choose the MAVLink messaging protocol, partially since it is used in a number of open-source autopilots and due to its message definitions customizability. While there were similar solutions already existing, none of them tried implementing a communication link over the RC. While its main drawback is the very low bandwidth, it has a great range advantage to a Wi-Fi connection. It can be used in enclosed spaces, something that a mobile network connection would not guarantee. We focused on providing a framework which would need minimal work to add new messages, which could be used for transmitting commands, log data, flight status and external sensor data. To that end we implemented an Android GCS and a pluginlib-supported ROS package. We also implemented a protocol to reliably transmit any files between devices, which could contain configuration parameters for flight missions or computing results. Image, video, telemetry and sensor data would still preferably be transmitted over functions provided by the Mobile SDK, as they offer greater bandwidth than the Data Transparent Transmission. In short, we provided a linkage between a mobile platform and the onboard ROS computer, granting developers a better flexibility in creating their own flight APIs.

We used our demo application to demonstrate it is possible to fully control the drone over the custom data link. It provides basic flight controls for directly steering the drone, in a similar fashion as can be done with the RC sticks. We display telemetry data, which is received from the drone in regular intervals. To demonstrate the file transmission protocol, we can send a configuration file for a flight trajectory and then execute it. We started our tests on the framework by first measuring the throughput of messages over varying distances and message payloads. This was followed up by trying out our GCS in a DJI flight simulator, and once we were satisfied any unfortunate accidents would not occur, we went to a a flight-field. The real-life experiment, where we completed a number of flights, went through without any issues, demonstrating the feasibility of our approach.

In the next part we explored ways of directly integrating VR support inside the ROS ecosystem, with the goal of providing a way to visualize and interact with the environment without needing outside tools. The idea was to allow easier development and prototyping by closely coupling our solution with the ROS-native RViz visualizer. To that end we again based it on the pluginlib system, where the plugin is then directly available in RViz and drawn by its OGRE engine. The interface to an HMD is implemented by using OpenVR, a SDK that abstracts the hardware layer. The stereo images can be displayed in two different ways. First is by rendering them from an virtual environment, where we receive the position of the camera from a tf listener and then apply stereo projection. Second is by streaming images from a stereo camera, which we directly display in the headset.

Since our thesis rests on support for UAV missions, our use case presents view management for a drone on an exploration mission, where the operator is required to perform some input to route around obstacles and finding targets. It is followed by a virtual camera or a secondary drone, which are there to provide a greater overview of the scene. The viewpoint is controlled by user's yaw and pitch VR-HMD rotations. The idea is the operator can get a secondary viewing angle on the area of operations and adjust it with intuitive HMD movements, without needing an additional controller. To allow further interaction with the environment and to provide a way to for example adjust the flight path, we also implemented support for reading out VR controllers position and button presses. Our solution was only tested in a virtual environment on an HTC Vive, although any other SteamVR-supported device should work as well.

5.2 Future work

The next sensible step for our MAVLink framework would be to integrate it to any of the supported GCS, of which we think QGroundControl would be the best choice due to its cross-platform compatibility. By emulating this project [21], we could provide a way of allowing integration of custom new messages and commands, that can be transmitted not just over TCP/IP, but over the RC as well. We have already tested this out by simply transmitting a MAVLink Hearbeat message, and QGroundControl was able to identify the connection as coming from a drone. If the integration would be completed, it would provide a fully functioning and actively developed GCS as a testing platform for further research and real-life flight missions.

As for our VR development, it could be worth exploring on further options of providing adaptive view management with the secondary drone/virtual camera. Actor drone, headset and controllers movement could be jointly used to calculate and predict an optimal viewing angle and height. Similar to [24] automatic view adjustment might achieve better mission performance than in cases, where only the actor drone's view is available or manual adjustment of the secondary is necessary. The challenge would be to make it work in unknown environments, where your would also have to take care for collision avoidance of a potential secondary drone.

Bibliography

[1] Skylogic Research Drone Analyst. 2017 drone market sector report, 2017. [Online; accessed 2018-May-09]. 18

[2] Andre Gilerson. Rviz plugin for the htc vive, 2016. [Online; accessed 27-August-2018]. 62

[3] Catalina Aranzazu Suescun and Mihaela Cardei. Unmanned aerial vehicle networking protocols. 01 2016. iii, 35

[4] C. Bartneck, M. Soucy, K. Fleuret, and E. B. Sandoval. The robot engine — making the unity 3d game engine work for hri. In *2015 24th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 431–437, Aug 2015. 61

[5] R. Codd-Downey, P. M. Forooshani, A. Speers, H. Wang, and M. Jenkin. From ros to unity: Leveraging robot and virtual environment middleware for immersive teleoperation. In *2014 IEEE International Conference on Information and Automation (ICIA)*, pages 932–936, July 2014. iv, 61

[6] DJI developers. Dji sdk dji2mav, 2016. [Online; accessed 2018-February-11]. 37

[7] Thach D. Do, Juhum Kwon, and Chang-Joo Moon. Ground system software for unmanned aerial vehicles on android device. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 7(2):250 – 255, 2013. iii, 35

[8] Faculty of Computer and Information Science. Kr nekam tole kaže, 2018. [Online; accessed 21-September-2018].

[9] Fadri Furrer, Michael Burri, Markus Achtelik, and Roland Siegwart. *Robot Operating System (ROS): The Complete Reference (Volume 1)*, chapter RotorS—A Modular Gazebo MAV Simulator Framework, pages 595–625. Springer International Publishing, Cham, 2016. 31

[10] Gazebo and ROS community contributors. Ros package gazebo_ros_pkgs, 2018. [Online; accessed 20-August-2018]. 31

[11] Hyeong Ryeol Kam, Sung-Ho Lee, Taejung Park, and Chang-Hun Kim. Rviz: A toolkit for real domain data visualization. *Telecommun. Syst.*, 60(2):337–345, October 2015. iv, 31, 62

[12] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, Sept 2004. 30

[13] L. Meier, D. Honegger, and M. Pollefeys. Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6235–6240, May 2015. 25

[14] OGRE contributors. About, ogre - open source 3d engine, 2018. [Online; accessed 24-August-2018]. 33

[15] PARROT developers. Mavlink code generator, 2018. [Online; accessed 2018-June-11]. v, 41

[16] PARROT developers. Mavros overview and installation instructions, 2018. [Online; accessed 2018-June-15]. 36

[17] Weichao Qiu and Alan Yuille. Unrealcv: Connecting computer vision to unreal engine. In Gang Hua and Hervé Jégou, editors, *Computer*

Vision – ECCV 2016 Workshops, pages 909–916, Cham, 2016. Springer International Publishing. iv, 61

[18] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009. 25

[19] Daniel Rakita, Bilge Mutlu, and Michael Gleicher. An autonomous dynamic camera method for effective remote teleoperation. In *Proceedings of the 2018 ACM/IEEE International Conference on Human-Robot Interaction*, HRI '18, pages 325–333, New York, NY, USA, 2018. ACM. 63, 64

[20] Camacho David Ramirez-Atencia Cristian. Extending qgroundcontrol for automated mission planning of uavs. *Sensors*, 18(7):2339, 2018. 37

[21] Rogue Squadron, Defense Innovation Unit Experimental. Android-based mavlink wrapper for dji drones, 2018. [Online; accessed 2019-January-12]. iii, 37, 85

[22] ROS community contributors. Ros concepts, 2018. [Online; accessed 23-August-2018]. 25

[23] ROS community contributors. Ros posestamped message, 2018. [Online; accessed 2018-May-25]. 39

[24] John Thomason, Photchara Ratsamee, Kiyoshi Kiyokawa, Pakpoom Kriangkomol, Jason Orlosky, Tomohiro Mashita, Yuki Uranishi, and Haruo Takemura. Adaptive view management for drone teleoperation in complex 3d structures. In *Proceedings of the 22Nd International Conference on Intelligent User Interfaces*, IUI '17, pages 419–426, New York, NY, USA, 2017. ACM. 63, 79, 85

[25] Tractica community contributors. Commercial drone shipments to surpass 2.6 million units annually by 2025, 2018. [Online; accessed 10-May-2018]. 17

[26] Virtual Reality and Augmented Reality Wiki community contributors. Htc vive, 2018. [Online; accessed 19-August-2018]. 29

[27] Virtual Reality Society. What is virtual reality?, 2018. [Online; accessed 25-August-2018]. i, 18