



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG



Simuleringssystem för decentraliserade UAV-svärmar i tredimensionella miljöer

Kandidatarbete för Data- och Informationsteknik

Marcus Forslund

Odin Hufnagl

Emil Lindblom

Axel Olsson

Viktor Strandrot Frid

Melker Östling

Institutionen för Data- och Informationsteknik

CHALMERS TEKNISKA HÖGSKOLA

GÖTEBORGS UNIVERSITET

Göteborg, Sverige 2026

KANDIDATARBETE 2026

Simuleringssystem för decentraliserade UAV-svärmar i tredimensionella miljöer

Marcus Forslund
Odin Hufnagl
Emil Lindblom
Axel Olsson
Viktor Strandrot Frid
Melker Östling



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2026

Simuleringssystem för decentraliserade UAV-svärmar
Marcus Forslund Odin Hufnagl Emil Lindblom Axel Olsson Viktor Strandrot
Frid Melker Östling

© Marcus Forslund, Odin Hufnagl, Emil Lindblom, Axel Olsson, Viktor Strandrot
Frid, Melker Östling 2026.

Handledare: Arne Linde, Institutionen för Data- och Informationsteknik

Examinator: Patrik Jansson och Arne Linde, Institutionen för Data- och Informationsteknik

Rättande lärare: Örjan Askerdal, Institutionen för Data- och Informationsteknik

Kandidatarbete 2026
Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola och Göteborgs Universitet
SE-412 96 Göteborg
Telefon +46 31 772 1000

Cover: Renderad figur över tredimensionell värld med autonoma drönare som avsöker miljön.

Typeset in L^AT_EX
Göteborg, Sverige 2026

Simuleringssystem för decentraliserade UAV-svärmar

Marcus Forslund, Odin Hufnagl, Emil Lindblom, Axel Olsson, Viktor Strandrot Frid, Melker Östling

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola och Göteborgs Universitet

Abstract

Unmanned aerial vehicles (UAVs) have great potential to help increase effectiveness in search and rescue operations. Testing of these systems with real hardware is costly and time-consuming, therefore this thesis develops a system to run the tests in a simulation environment. The simulations are run with Gazebo together with the flight controller ArduPilot and robotics middleware ROS 2. The system is used to evaluate four search algorithms BMUTA, UBS, PSO and RAS in forest, city and sea environments. The algorithms are evaluated using several metrics, including coverage rate, overlap, and time to target detection. The proposed system enables simulations in varying environments with relevant data collected for analysis and evaluation of the algorithms. The results indicate that swarm size and start formation had the clearest impact on mission time, while world type showed a less consistent effect in the tested scenarios. Among the evaluated algorithms, RAS achieved the shortest median mission times across all swarm sizes and the lowest variation between replications, indicating the most stable overall performance.

Sammandrag

Obemannade luftfarkoster (UAV:er) har stor potential att bidra till ett effektivare sökarbete efter försvunna personer. Att utföra testerna med riktig hårdvara är kostsamt och tidskrävande, därför tar detta arbete fram ett system för att kunna utföra dessa tester i simuleringssmiljöer som speglar verkligheten. Simuleringarna körs i Gazebo tillsammans med styrsystemet ArduPilot och robotikramverket ROS 2. Systemet används för att utvärdera prestandan av fyra sökalgoritmer BMUTA, Uncertainty-Based Baseline Search, PSO och RAS i skogsmiljö, stadsmiljö och havsmiljö. Algoritmerna utvärderas på bland annat täckningsgrad, överlappning och tid att hitta målen. Det framtagna systemet möjliggör simulering i varierande miljöer där relevant data samlas in för analys och utvärdering av algoritmerna. Resultaten visar att svärmstorlek och startformation hade tydligast påverkan på uppdragstiden, medan världstypen hade en mindre konsekvent effekt i de genomförda experimenten. Av de utvärderade algoritmerna uppnådde RAS kortast medianuppdragstid för samtliga svärmstorlekar och lägst variation mellan replikeringar, vilket tyder på den mest stabila övergripande prestandan.

Nyckelord: Sök- och räddningsuppdrag, UAV:er, Simulering, BMUTA, PSO, RAS

Erkännanden

Vi vill uttrycka vår tacksamhet till vår handledare, Arne Linde, för hans stöd och vägledning under projektets gång.

Marcus Forslund, Odin Hufnagl, Emil Lindblom,
Axel Olsson, Viktor Strandrot Frid, Melker Östling
Göteborg, juni 2026

Akronymer

AWS Amazon Web Services
BMUTA Bee-inspired multi-UAV teaming algorithm
ENU East-North-Up
IQR Interquartile Range
NED North-East-Down
PSO Particle Swarm Optimization
RAS Recontamination Aware Search
ROS 2 Robot Operating System 2
SITL Software In The Loop
UAV Unmanned Aerial Vehicle
UBS Uncertainty-Based Baseline Search

Innehåll

Akronymer	viii
Figurer	xi
Tabeller	xiv
1 Introduktion	1
1.1 Bakgrund	1
1.2 Problembeskrivning	1
1.2.1 Syfte	2
1.3 Metod	2
1.4 Avgränsningar	3
2 Teoretisk bakgrund	4
2.1 Multi-agent system	4
2.2 ROS 2	4
2.3 ArduPilot	5
2.4 Gazebo	5
2.5 Amazon Web Services	5
2.6 RViz2	6
3 System	7
3.1 Systemöverblick	7
3.2 Scenarion och experiment	7
3.2.1 Scenario	8
3.2.2 Experiment	8
3.3 Telemetri och datainsamling	8
3.3.1 Telemetrimodell	8
3.3.2 Datainsamling	9
3.3.3 Visualisering och analys	9
3.4 Drönarstyrning	9
3.4.1 Brain-noden	10
3.4.2 Integration med flygkontroll	10
3.4.3 Uppstarts- och startsekvens	11
3.5 Simuleringsvärld	11
3.5.1 Generering av 3D-världar	11
3.5.2 Tredimensionella modeller av träd och byggnader	14

3.5.3	Automatisk exportering av världar och texturer	14
3.5.4	Generering av voxeliserade varianter av världar	15
3.6	Gemensam sök- och perceptionsinfrastruktur	16
3.6.1	Inläsning och publicering av världen	16
3.6.2	Konbaserad raycasting och voxelintersection	17
3.6.3	Sökregistrering och tillståndsspårning	17
3.6.4	Måldetektion	19
3.6.5	Kollisionsundvikande	19
3.7	Målens representation och rörelse	20
3.7.1	Representation och placering	21
3.7.2	Rörelsemodell	21
3.7.3	Tröghet i rörelsen	21
3.7.4	Tillståndspubliserad och detektion	22
3.8	Användning av systemet	22
3.8.1	Systemstöd för beteenden	22
3.8.2	Gränssnitt och krav	23
3.8.3	Integration i systemet	24
3.8.4	Exekvering	24
3.8.5	Köra experiment	25
3.8.6	Analys av experiment	26
4	Experimentell utvärdering	27
4.1	Testade algoritmer	27
4.1.1	Bee-inspired multi-UAV teaming algorithm	27
4.1.2	Uncertainty-Based Baseline Search	30
4.1.3	Particle Swarm Optimization	31
4.1.4	Recontamination Aware Search	33
4.2	Experimentuppställning	37
4.2.1	Antal drönare	37
4.2.2	Antal mål	38
4.2.3	Världstyper	38
4.2.4	Startformationer	38
4.2.5	Totalt antal simuleringar	38
4.3	Genomförande av simuleringar	39
5	Resultat	41
5.1	Experimentets genomförande och datakvalitet	41
5.2	Hur påverkas uppdragstiden av experimentparametrarna?	42
5.3	Hur skiljer sig algoritmernas sökbeteende?	47
5.4	Hur skiljer sig algoritmernas rörelse- och koordinationsbeteende?	49
5.5	Hur robusta är algoritmerna?	50
5.6	Sammanfattande jämförelse mellan algoritmer	51
6	Slutsats	52
6.1	Modulär systemarkitektur	52
6.2	Decentraliserade algoritmer	52
6.3	Simulerade uppdrag och experiment	53

6.4	Utvärdering och analys	53
7	Framtida arbete	55
8	Etiska aspekter	56
	Referenser	57
A	Blender	I
A.1	Skogsgeneratorns parametrar	I
A.2	Skogsgeneratorns nodbaserade implementation	I
A.2.1	Terränggenerering	I
A.2.2	Vägggenerering	III
A.2.3	Skogsmask och trädplacering	III
A.3	Stadsgeneratorns nodbaserade implementation	IV
A.4	Havsgeneratorns nodbaserade implementation	VI
A.5	Förberedelse av träd- och byggnadsmodeller	VI
A.6	Automatisk exportering av världar och texturer	VII
B	Voxelisering	IX
B.1	Voxeliseringskedja	IX
B.2	CSV- och BT-representationerna	X
B.3	Val av voxelupplösning	X
B.4	Användning under simulering	X
C	Sök- och perceptionsinfrastruktur	XII
C.1	Inläsning och publicering av världen	XII
C.2	Konbaserad raycasting och voxelintersection	XIII
C.3	Sökregistrering och tillståndsspårning	XIII
C.4	Kollisionsundvikande	XIV
D	Målens representation och rörelse	XVI
D.1	Representation och placering	XVI
D.2	Rörelsemodell	XVI
D.3	Tröghet i rörelsen	XVII
D.4	Tillståndspublicering och detektion	XVIII
E	Pseudokod	XIX
E.1	Experiment-fil för analys	XXII

Figurer

3.1	Översikt av separationen mellan beteende och centraliserad systemdel.	10
3.2	Jämförelse mellan två genererade skogsmiljöer med olika världsstorlek och terrängupplösning. Den mindre miljön använder <i>World size</i> = 80 och <i>World resolution</i> = 200, medan den större miljön använder <i>World size</i> = 150 och <i>World resolution</i> = 375.	12
3.3	Jämförelse mellan två genererade stadsmiljöer med olika världsstorlek och terrängupplösning. Den mindre miljön använder <i>World size</i> = 80 och <i>World resolution</i> = 200, medan den större miljön använder <i>World size</i> = 150 och <i>World resolution</i> = 375.	13
3.4	Jämförelse mellan två genererade vattenmiljöer med olika världsstorlek och terrängupplösning. Den mindre miljön använder <i>World size</i> = 80 och <i>World resolution</i> = 200, medan den större miljön använder <i>World size</i> = 150 och <i>World resolution</i> = 375.	13
3.5	Jämförelse mellan olika voxelupplösningar. En större voxelstorlek ger en grövre och mer kompakt representation av miljön, medan en mindre voxelstorlek bevarar fler geometriska detaljer men ökar antalet voxelceller och därmed minnes- och beräkningskostnaden.	16
3.6	Visualisering av drönarens raycast-baserade genomsökning av den voxeliserade terrängen. De blåmarkerade voxlarna representerar det område som klassificerats som genomsökt efter en raycast-träff. Detta omfattar inte enbart de direkt träffade voxlarna, utan även de voxelceller som ligger inom radien <code>hit_radius_m</code> kring varje träffpunkt. I den aktuella systemkonfigurationen är denna radie satt till 7,0 m. De grönmärkade voxlarna representerar områden som ännu inte har genomsökts.	18
3.7	GUI för visualisering av insamlad telemetri under uppdragets gång. .	24
3.8	GUI för visualisering av Bee-inspired multi-UAV teaming algorithm (BMUTA) under uppdragets gång.	25
3.9	Skärmdump från GUI för visualisering av uppdraget via RVIZ2. . . .	25
4.1	Visuell beskrivning av regionval.	29
4.2	Avtagande för olika α	30

4.3	Schematisk illustration av Recontamination Aware Search (RAS) i en horisontell skiva av voxelvärlden. De röda voxelcellerna med gas (1) representerar områden med hög osäkerhet, medan vita voxelceller utan gas (2) representerar områden som nyligen har avsökts. Frontcellerna (3) bildar gränsen mellan dessa regioner och används som kandidater för waypoint-planering. Waypoints placerade på fronten (4) kopplas samman till waypoint-segment (5), där ett aktivt segment även används för att beräkna den projicerade look-ahead-distansen (6). Den resulterande look-ahead-vektorn (7), och dess skalade variant (8), styr drönarens (11) rörelse längs fronten. Den streckade orange kurvan (9) visar en approximativ flygbana, medan de röda streckade pilarna (10) illustrerar hur gasen diffunderar tillbaka in mot tidigare avsökta områden.	36
4.4	GUI för visualisering av experimentets progression under distribuerad exekvering av simuleringar.	40
5.1	Andel lyckade simuleringar i experimentet.	42
5.2	Jämförelse av total uppdragstid mellan olika världstyper och startformationer för svärmar med tre drönare.	43
5.3	Jämförelse av total uppdragstid mellan olika världstyper och startformationer för svärmar med åtta drönare.	43
5.4	Procentuell skillnad i medianuppdragstid mellan startformationerna border och grid, grupperad per beteende och svärmstorlek.	44
5.5	Skalning med svärmstorlek.	45
5.6	Relativ uppdragstid per algoritm och världstyp, uttryckt som procentuell avvikelse från medelvärdet för respektive världstyp. Ett positivt värde innebär att algoritmen presterar bättre än genomsnittet i den världstypen. Världstyperna är ordnade från mest till minst komplex tredimensionell miljö.	46
5.7	Täckningsgrad över tid för olika algoritmer.	47
5.8	Andel redundant genomsökning för olika algoritmer.	48
5.9	Genomsnittlig distans mellan drönare och dess målposition.	49
5.10	Genomsnittlig hastighet för olika algoritmer.	50
5.11	Replikeringsvarians för olika världstyper.	50
A.1	Jämförelse mellan tre olika brusnivåer på ett rutnät samt en kombinerad version. Stort brus ger större övergripande variationer, medium brus skapar mellanstora förändringar och litet brus bidrar med finare detaljer.	II
A.2	De olika stegen i genereringen av vägnätet. Först skapas ett slumpmässigt slingrande vägnät med Perlin-brus, därefter projiceras vägarna mot terrängytan med en shrinkwrap-liknande operation, och slutligen genereras en avståndsbaserad mask som används för att representera vägbredd och materialblandning.	IV

A.3	Exempel på tre skogsmasker genererade med samma procedur men olika <i>World seed</i> . Den fjärde dimensionen i Perlin-bruset styrs av seed-värdet, vilket gör att varje seed ger ett distinkt men reproducerbart täthetsfält. Maskerna används för att styra var trädinstanser sprids över terrängen, där ljusare områden motsvarar högre trädtäthet och mörkare områden motsvarar öppnare ytor.	V
-----	---	---

Tabeller

3.1	Antal genererade världar per typ och storlek. <i>World size</i> anger sidlängd i Blender-enheter och <i>World resolution</i> antalet hörn längs varje sida av rutnätsmeshen.	15
5.1	Medianuppdragstid för olika svärmstorlekar samt procentuell minskning i uppdragstid mellan svärmstorlekar. $\tilde{t}_N$ anger medianuppdragstiden för svärmstorlek N , medan kolumnerna $3 \rightarrow 5$, $5 \rightarrow 8$ och $3 \rightarrow 8$ anger den procentuella minskningen i uppdragstid.	45

1

Introduktion

1.1 Bakgrund

Varje år får Polisen in runt 27 000 ärenden om försvunna personer runt om i landet (Polismyndigheten, 2023). Många ärenden hanteras snabbt, men en mindre del tar lång tid att upplära eller uppläras inte alls. Dessa längre ärenden tas bland annat upp av organisationer som Missing People, som under 2024 hade 490 ärenden (Missing People Sweden, 2024). Sökandet efter försvunna personer sker ofta i svårtillgänglig terräng, exempelvis tät skog eller berg (DesMarais, 2017). I sådan terräng begränsas markbaserade sökningar av låg framkomlighet, vilket gör luftburna system till ett viktigt komplement. Autonoma drönarsystem, även kallade Unmanned Aerial Vehicle (UAV), skulle därför kunna bidra till effektivare sökinsatser och öka chansen att hitta den försvunne vid liv.

Under senare år har UAV:er fått ökad betydelse inom områden som övervakning, kartläggning och säkerhetskritiska insatser. Samtidigt blir begränsningarna hos manuellt styrda och enskilda UAV-system tydliga i komplexa uppdrag, särskilt när sökområden är stora, svåröverskådliga eller dynamiska (Zhao m. fl., 2025). För att hantera dessa begränsningar har intresset ökat för autonoma multi-UAV-system, där flera drönare samarbetar för att lösa gemensamma uppgifter. Genom samverkan kan arbetsbelastningen fördelas och större områden täckas mer effektivt, vilket är särskilt relevant i miljöer där sensorer begränsas av exempelvis vegetation, byggnader eller terräng (Zhao m. fl., 2025).

För att kunna utveckla och utvärdera sådana system krävs dock mer än bara effektiva sökalgoritmer. Det behövs också en stabil och duplicerbar simuleringsmiljö där algoritmer kan testas i realistiska världar innan de används i verkliga uppdrag. En sådan miljö behöver möjliggöra kommunikation mellan autopilot, simulator och styrlogik, samtidigt som olika miljöparametrar, startpositioner och sökförhållanden kan varieras på ett kontrollerat sätt. Detta är extra viktigt eftersom implementering, validering och utvärdering av multi-UAV-algoritmer kräver pålitliga simuleringsscenarier där flera typer av miljöer kan analyseras (Cho m. fl., 2022).

1.2 Problembeskrivning

Det övergripande problemet i detta arbete är hur en stabil och reproducerbar simuleringsmiljö kan utformas för att möjliggöra utveckling, testning och utvärdering av

samverkande multi-UAV-system inom sök- och räddningsuppdrag. Sådana system bygger ofta på distribuerade beslutsalgoritmer, där flera autonoma enheter behöver samordna sina handlingar baserat på lokal information och begränsad kommunikation. För att kunna analysera dessa algoritmer på ett bra sätt krävs därför inte enbart en teoretisk jämförelse, utan även en teknisk miljö där flera UAV:er kan operera parallellt i realistiska tredimensionella scenarier.

1.2.1 Syfte

Syftet med detta kandidatarbete är att undersöka hur decentraliserade samarbetsalgoritmer för autonoma UAV:er påverkar systemets prestanda i sök och räddningsinspirerade uppdrag inom simulerade miljöer. Arbetet fokuserar på att möjliggöra en jämförbar utvärdering av olika beslutsstrategier i distribuerade multi-UAV system, där varje enhet fattar lokala beslut baserat på begränsad information och kommunikation.

För att uppnå detta har en systemstruktur utformats som möjliggör implementering och testning av samverkande algoritmer i varierande miljöer, med målet att analysera hur faktorer som uppdragskaraktär och val av algoritm påverkar systemets effektivitet vid områdessökning och informationsinsamling i sök och räddningsrelaterade scenarier.

Det övergripande syftet konkretiseras genom följande mål:

1. Utforma en modulär systemarkitektur som separerar simulering, kommunikation, beslutslogik och analys för att möjliggöra implementering och jämförbar testning av olika samarbetsalgoritmer.
2. Implementera decentraliserade samverkansstrategier där varje UAV fattar beslut baserat på lokal information och informationsutbyte med närliggande enheter.
3. Definiera representativa sök och räddningsinspirerade uppdrag med varierande miljömässiga förutsättningar.
4. Utvärdera algoritmernas prestanda med avseende på relevanta multi-UAV måttetal, exempelvis täckningsgrad, tid till måluppfyllelse och överlappning.

1.3 Metod

Arbetet genomfördes i olika delar där varje gruppmedlem fick ett område att ansvara för. De olika områdena omfattade algoritmutveckling, miljögenerering och struktursturuppbbyggande.

Under utvecklingsfasen användes ett Git-baserat versionshanteringssystem för att dela projektets olika delar inom gruppen och för att säkerhetskopiera systemet vid eventuell dataförlust. Gitmiljön bidrog även till ett effektivare arbetsflöde eftersom den möjliggjorde sammanfogning av gruppmedlemmarnas arbete via en plattform.

I systemet skickar sökalgoritmerna signaler som anger hur drönarna ska agera i simuleringen. Robot Operating System 2 (ROS 2) användes som robotikramverk för att strukturera kommunikationen mellan systemets komponenter, vilket bland annat möjliggjorde att varje drönare representerades av en separat nod där data kunde delas effektivt. Som flygkontrollsystem för drönarna användes ArduPilot, vilket valdes för dess omfattande navigeringsstöd och förmåga att hantera autonom flygstyrning. Simuleringarna genomfördes i Gazebo och detta valdes baserat på dess realistiska simuleringar samt att programmet fungerade väl ihop med ROS 2 och ArduPilot. Världarna som drönarna befann sig i skapades med Blender och detta valdes då gruppen har erfarenhet inom detta program samt att det tillät anpassning av tredimensionella världar.

För att jämföra algoritmer under olika förhållanden byggdes det även in stöd i systemet för att exekvera flera simuleringar inom ett kontrollerat experiment, spara ner simuleringarnas telemetri, och sedan kunna använda utvecklade analysverktyg för att utvärdera telemetrin och dra slutsatser. Det byggdes även stöd för att köra dessa simuleringar i Amazon Web Services (AWS) för att påskynda tiden det tar att köra simuleringarna.

1.4 Avgränsningar

Enligt Zhao m. fl. (2025) är drönare känsliga för kollisioner, och verkliga tester kan medföra ekonomiska risker. Arbetet har därför avgränsats till simulering för att minska kostnader och undvika skador på fysisk utrustning. Detta innebär att resultaten bör tolkas utifrån de antaganden och förenklingar som följer av en simulerad miljö, och att systemets prestanda inte har verifierats genom verkliga flygtester.

Arbetet har huvudsakligen genomförts i Python, med fokus på att utveckla och utvärdera ett fungerande koncept snarare än ett produktionsfärdigt system. Därför behandlas inte realtidskrav, hårdvarunära optimering eller integration med specifika fysiska sensorer och kommunikationsmoduler. Simuleringen har även begränsats i detaljgrad för att möjliggöra upprepade tester inom rimlig tid. Enligt Xiong m. fl. (2023) kan komplexa simuleringsmiljöer kräva hög beräkningskraft och längre simuleringstider, vilket motiverar att terräng, hinder och fysikmodeller hållits på en nivå som är tillräcklig för att analysera koordinering och informationsutbyte.

Vidare har arbetet avgränsats till ett begränsat antal världar, scenarier och algoritmer. Miljöerna är avsedda att efterlikna sök- och räddningsinspirerade situationer, men omfattar inte hela komplexiteten i verkliga insatser, såsom mänskligt beteende, juridiska flygrestriktioner eller samverkan med externa aktörer. Kommunikationen mellan drönare modelleras inom det simulerade systemet och inkluderar inte en fullständig modell av verklig radiokommunikation med exempelvis paketförlust, signalstörningar eller begränsad bandbredd. Fokus ligger därför på att jämföra decentraliserade beslutsstrategier utifrån valda mätvärden, såsom täckningsgrad och tid till måluppfyllelse.

2

Teoretisk bakgrund

Detta kapitel introducerar de teoretiska och tekniska begrepp som ligger till grund för arbetet. Kapitlet behandlar bland annat multi-agent system samt de verktyg och ramverk som används för simulering, styrning och visualisering av drönarsystem.

2.1 Multi-agent system

Ett multi-agent system består av flera autonoma enheter, så kallade agenter, som interagerar inom en gemensam miljö för att uppnå individuella eller kollektiva mål. I kontexten av multi-UAV system representeras varje drönare som en agent med förmåga att fatta egna beslut baserat på lokalt tillgänglig information samt information mottagen från andra enheter i (Ordoukhanian & Madni, 2017).

Till skillnad från centraliserade system, där en enskild enhet ansvarar för planering och koordinering, bygger distribuerade multi-agent system på att varje agent agerar självständigt samtidigt som samarbete uppnås genom informationsutbyte. Detta möjliggör ökad robusthet vid bortfall av enskilda enheter eller kommunikationslänkar, vilket är särskilt relevant i sök- och räddnings-scenarier där miljön kan vara dynamisk och infrastruktur saknas eller är instabil. I sådana uppdrag behöver UAV:er exempelvis kunna fördela sökområden, undvika överlappande täckning och anpassa sitt beteende vid förändrade förutsättningar. Detta kräver beslutsstrategier som fungerar trots begränsad kommunikation och frånvaro av global överblick, vilket utgör en central utmaning inom distribuerade multi-agent system.

2.2 ROS 2

ROS 2 är ett robotikramverk och en mellanprogramvara för robotprogramvara som används för att strukturera distribuerade robotsystem. ROS 2 är uppbyggt av fyra baskomponenter noder, gränssnitt, parametrar och klientbibliotek (Open Robotics, 2022a).

- **Noder** är alla delar av systemet som skickar och/eller tar emot data via meddelande. Det är till exempel drönarna själva, sensorer och kameror.
- **Gränssnittet** inkluderar tre delar: topics, services och actions. Topics som används för distribution av data. Noderna kan publicera data under topics eller prenumerera på ett topic för att få tillgång till all information som publiceras

över detta topic. Services används för att noder ska kunna fråga varandra om information. Actions används för att utföra längre uppgifter som kräver återkoppling under utförandet.

- **Parametrar** används som inställningar för noderna och kan konfigureras vid start eller under körtid.
- **Klientbibliotek** är programmeringsgränssnitt som gör att ROS 2 kan användas och ger möjligheten att välja det programmeringsspråk som passar användaren och syftet bäst.

2.3 ArduPilot

ArduPilot är ett mjukvaruprogram som möjliggör styrning av virtuella farkoster från programkod. Programmet fungerar på det sättet att operatören beskriver i koden vad drönarna ska göra med hjälp av olika förprogrammerade kommandon. Dessa kommandon tolkas sedan av MAVLink, vilket i sin tur fördelar information till en Software In The Loop (SITL) port. Varje drönare har en egen SITL port, vilket gör att drönarna kan agera på olika sätt i simuleringen. ArduPilot är en form av central som talar om för farkosterna hur och vart de ska röra sig. (Coombes m.fl., 2012).

2.4 Gazebo

Gazebo är ett simuleringsprogram skapat för robotsimuleringar i realistiska miljöer. Programmet låter en implementera till exempel sensorer, kameror och egna världar för att skapa de förutsättningarna ens system utsätts för i verkligheten och därav testa lösningar utan att behöva spendera pengar på riktiga hårdvarukomponenter innan man vet att det fungerar vilket beskrivs av (Chen m.fl., 2023).

2.5 Amazon Web Services

Amazon Web Services (AWS) är en molnbaserad plattform som tillhandahåller distribuerade infrastrukturtjänster, bland annat virtuella servrar, datalagring och nätverksresurser.

Elastic Compute Cloud (EC2) är en tjänst inom AWS för att skapa och hantera virtuella instanser som kan användas för distribuerad exekvering av applikationer och beräkningar (Amazon Web Services, 2026a).

Simple Storage Service (S3) är en objektbaserad lagringstjänst inom AWS som används för att lagra och hämta filer och data i molnet (Amazon Web Services, 2026b).

2.6 RViz2

RViz2 är ett visualiseringsverktyg för ROS 2 som används för att visualisera data och tillstånd i robotiksystem (Open Robotics, 2022b). Verktöget stödjer visualisering av exempelvis koordinatsystem, sensordata, punktmoln, banor och tredimensionella modeller i realtid. I detta arbete används RViz2 främst för övervakning, felsökning och analys av systemets tillstånd under simulering och exekvering.

3

System

Detta kapitel beskriver hur systemet fungerar. Det börjar med en överblick av system, följt av beskrivningar av viktiga delar och en förklaring av hur systemet används.

3.1 Systemöverblick

Det utvecklade systemet är ett ramverk för att implementera, testa och utvärdera sökbaserade beteenden för svärmar av drönare i simulerade tredimensionella miljöer. Systemet är utformat för att möjliggöra jämförelse av olika beteenden under kontrollerade och reproducerbara förhållanden, där fokus ligger på att separera beslutslogik från exekvering och datainsamling.

På en övergripande nivå består systemet av tre huvudsakliga delar. Den första delen utgörs av beteenden, vilka definierar hur en enskild drönare fattar beslut baserat på tillgänglig information. Den andra delen är den centraliserade systemkomponenten som ansvarar för exekvering, tillgång till tillståndsinformation, interaktion med den tredimensionella simuleringen samt insamling av telemetridata. Den tredje delen utgörs av experiment- och analysverktyg, vilka används för att konfigurera simuleringar, genomföra upprepade körningar samt analysera resultaten.

Interaktionerna mellan dessa delar är utformade så att beteenden enbart behöver fokusera på beslutslogik och beteende-specifik kommunikation mellan drönare, medan den centraliserade delen hanterar centraliserad kommunikation oberoende av beteende, koordinatsystem, rörelsekommandon och datainsamling. Detta möjliggör en modulär struktur där nya beteenden kan utvecklas och testas utan att påverka övriga delar av systemet.

Genom att kombinera denna uppdelning med ett flexibelt experimentsystem kan olika parametrar, exempelvis antal drönare, miljöer och mål, varieras systematiskt. Detta skapar en grund för att analysera beteendens prestanda och robusthet under olika förutsättningar.

3.2 Scenarion och experiment

Som introducerades i avsnitt 3.1 utgörs systemets tredje del av de verktyg som hanterar experiment och analys. För att styra hur olika simuleringar definieras,

organiseras och exekveras inom systemet används en scenariohanterare.

3.2.1 Scenario

Ett scenario utgör den minsta exekverbara enheten inom systemet. Varje scenario definieras av en tillhörande JSON-fil som specificerar samtliga initialparametrar för simuleringen. Dessa parametrar omfattar bland annat val av beteende, simuleringsvärld, drönarnas startpositioner, målens initiala placeringar samt eventuella beteendespecifika parametrar. Ett exempel på scenariokonfiguration ges i Listning E.3.

Exekveringen av ett scenario orkestreras av en dedikerad scenariehanterare, vars process följer en sekventiell uppstartsfas. Den börjar med att konvertera scenario-filen till ett scenario-objekt för att sedan starta en gazebo-server. Sedan startar Ardupilot SITL upp en instans för varje drönare definierade i scenario-filen, i syfte att hantera flygdynamik och upprätta kommunikationsportar med både gazebo och ROS 2. Därefter initieras ROS 2-ramverket. Denna delen av uppstarten beskrivs mer i detalj i avsnitt 3.4.3. Under hela simuleringen loggas telemetridata, vilken vid körningens slut sammanställs i en lokal databas för efterföljande analys.

3.2.2 Experiment

Ett experiment definierar ett parametersvep över flera experimentdimensioner, exempelvis beteende, världstyp, svärmstorlek och startformation. Konfigurationen anges i en tillhörande YAML-fil. Pseudokod för experimentkonfigurationen finns i Appendix E.2.

Exekveringen av ett experiment orkestreras av en dedikerad experimenthanterare, som läser in en experiment-fil och genererar ett scenario-objekt för varje kombination av de parametrar som ingår i parametersvepet. Dessa objekt exekveras sedan via scenariohanteraren. Efter att samtliga scenarier i ett experiment har slutförts, sammanställs den genererade telemetridatan från alla exekveringar i en gemensam katalogstruktur.

3.3 Telemetri och datainsamling

För att möjliggöra en systematisk utvärdering av de implementerade beteendena samlar systemet löpande in telemetridata under varje simulering. Insamlingen hanteras centralt och är frikopplad från drönarnas specifika beslutslogik, vilket garanterar ett objektivt och jämförbart dataunderlag.

3.3.1 Telemetrimodell

För att standardisera informationsutbytet och loggningen är telemetrimodellen uppbyggd kring två primära meddelandetyper i ROS 2, vilka tillsammans definierar systemets tillstånd och händelser.

- **DRONE_STATE**: Utgör en kontinuerlig dataström som loggar varje enskild drönarens aktuella tillstånd under körningen. Detta inkluderar:
 - Drönarens unika identifierare
 - Tredimensionella position
 - Linjära och angulära hastighet
 - Aktuell målposition
- **EVENT**: En händelsestyrd dataström som publiceras vid specifika händelser. Modellen stöder tre unika händelsetyper:
 - **MISSION_STARTED**: När en drönare startas
 - **TARGET_DETECTED**: När ett mål upptäcks
 - **MISSION_COMPLETE**: När alla mål är funna

För att hantera händelsespecifik information på ett flexibelt sätt bifogas relevant data som en JSON-sträng.

Samtliga meddelandetyper innehåller `drone_id` för den drönare som publicerade meddelandet, samt vid vilken tidpunkt detta skedde. **TARGET_DETECTED** innehåller även ett `target_id` på vilket mål som blev hittat och positionen målet hittades.

3.3.2 Datainsamling

Insamlingen av telemetridata sker i en dedikerad ROS 2-nod, `bag_recorder_node`, vilken ansvarar för att starta och hantera inspelningen av de förutbestämda dataströmmarna med hjälp av verktyget `rosbag2`. Datan skrivs till en lokal databas och sparas i en utdatamapp strukturerad efter experimentets unika körnings-ID.

3.3.3 Visualisering och analys

För att möjliggöra både realtidsövervakning och retrospektiv utvärdering är systemet utrustat med dedikerade verktyg för visualisering och dataextraktion. Under exekvering kan en grafisk gränssnittsnod, `telemetry_gui_node`, användas för realtidsvisualisering. Denna nod använder Matplotlib för att rendera en live-vy över svärmens agerande, där en karta kontinuerligt uppdateras med data från telemetrin, exempelvis drönarnas positioner, mål och upptäckta målobjekt. Parallellt ritas prestandagrafer upp som följer drönarnas tillryggalagda distans och hastighet över tid.

I systemet finns det tillgång till analys-verktyg som tar telemetri-data från ett experiment som indata och genererar tabeller och grafer för en djupare analys av datan.

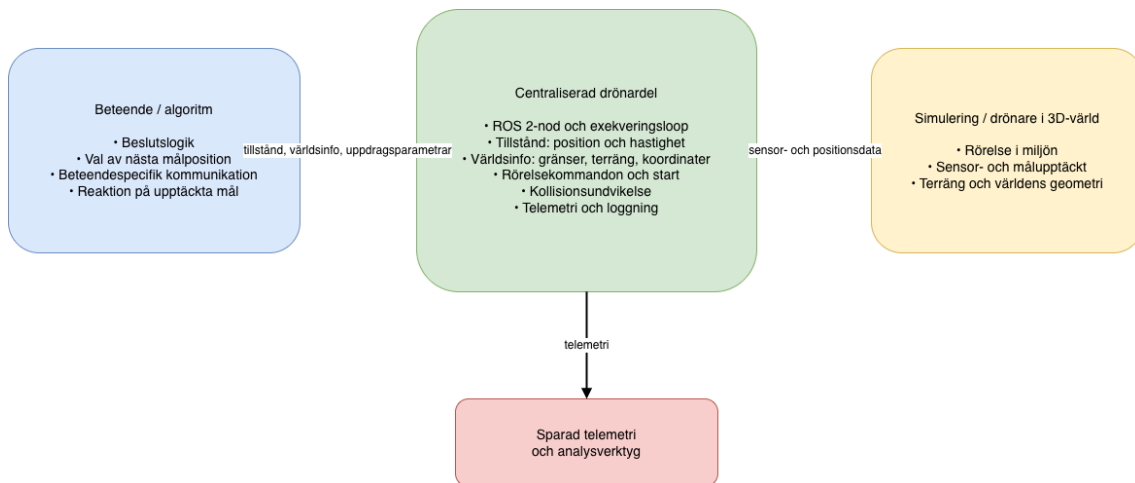
3.4 Drönarstyrning

Systemet är utformat för att separera den kognitiva beslutslogiken från den lågnivå-flygstyrning som krävs för fysisk förflyttning. Denna separation möjliggör en modular arkitektur där olika sökbeteenden kan implementeras och utvärderas på en enhetlig plattform.

3.4.1 Brain-noden

Varje drönare i svärmen kör en instans av **brain_node**, en dedikerad ROS 2-nod som utgör länken mellan det överliggande beteendet och flygkontrollen. Noden ansvarar för att, bland annat anropa beteendets tick-funktion med en konfigurerbar frekvens, assemblera rörelsekommandon från beteendets waypoint och kollisionsundvikandets motstående vektor, samt emittera telemetri. Brain-noden fungerar därmed som ett standardiserat gränssnitt mellan beteendekoden och övriga systemet. Beteendet behöver inte själv hantera interaktionen med ArduPilot, koordinattransformationer eller telemetriloggning, utan returnerar endast beslutsinformation i form av exempelvis nästa målposition eller önskad rörelse. Detta gör att olika beteenden kan implementeras med samma externa gränssnitt, trots att deras interna beslutslogik skiljer sig åt.

Figur 3.1 visar hur beteendekoden, den centraliserade drönarfunktionaliteten och simuleringen är separerade från varandra.



Figur 3.1: Översikt av separationen mellan beteende och centraliserad systemdel.

3.4.2 Integration med flygkontroll

Drönarsimuleringen bygger på ArduPilot SITL, som hanterar flygdynamik och attitydreglering. Integrationen mellan ROS 2 och ArduPilot sker via en MAVLink-adapter som hanterar den dubbelriktade kommunikationen samt nödvändiga transformationer mellan koordinatsystemen East-North-Up (ENU) och North-East-Down (NED). Genom att försätta ArduPilot i flygläget *GUIDED* kan svärmens centrala kontrollnod **brain_node** kontinuerligt strömma hastighetskommandon som omsätts till fysiska motorpådrag, samtidigt som aktuell telemetri exempelvis position och hastighet återkopplas till systemet.

3.4.3 Uppstarts- och startsekvens

När ett scenario initieras och har genomgått sin uppstartsfas av simuleringsmiljön, beskrivet i avsnitt 3.2.1 orkestreras uppstarten av svärmen i ROS 2. Detta sker via ett antal launch-skript. Dessa skript ansvarar för att starta och knyta samman ArduPilot SITL, MAVLink-adaptrar och respektive drönarens `brain_node`. För att säkerställa ett säkert och kontrollerat förlopp innan de autonoma beteendena tar över, hanteras startsekvensen av en övervakande systemnod `supervisor_node`. Denna process innebär att systemet först inväntar stabil telemetri och GPS-låsning, varefter drönarna armeras och ett automatiserat startkommando *TAKEOFF* utfärdas. Först när samtliga drönare har nått sin angivna operationella höjd överlämnas kontrollen slutligen till sökalgoritmerna, vilket förhindrar instabilitet under själva uppstartsfasen.

3.5 Simuleringsvärld

Enligt Strohhecker m. fl. (2008) ger en simuleringsvärld en uppfattning om hur en modell presterar i ett verkligt scenario. Olika parametrar och väderförhållanden kan styras i syfte att minimera oförutsägbara avvikelser som kan inträffa i en verklig miljö.

3.5.1 Generering av 3D-världar

I arbetet valdes tredimensionella världar framför tvådimensionella för att öka realismen och möjliggöra algoritmer som tar hänsyn till fysiska hinder, vilket är en förutsättning för eventuell tillämpning i verkliga scenarier. För att uppnå detta krävs ett tillräckligt brett urval av världar samt möjligheten att generera flera varianter av samma världstyp med olika parametrar. Av denna anledning valdes Blender, ett verktyg med öppen källkod för 3D-modellering som innehåller funktionen `geometry nodes`. Denna funktion gör det möjligt att proceduralt definiera geometri, relationer mellan objekt, partikelsystem och texturer. Tre typer av världsgeneratorer utvecklades: skogsvärldar, stadsvärldar och havsvärldar (Soni m. fl., 2023).

Urvalet av världstyper är gjort för att täcka tre distinkta scenarier som är representativa för verkliga sök- och räddningsuppdrag samtidigt som de spänner över ett spektrum av geometriska och kommunikationsmässiga förutsättningar. Skogsmiljön (se Figur 3.2) representerar ett organiskt naturlandskap där vegetationen utgör många små, oregelbundet utspridda hinder. Detta är en typ av miljö där människor kan gå vilse, vilket gör miljön direkt relevant för sök- och räddningstillämpningar (Schedl m. fl., 2021). Vegetationen ger samtidigt partiell sikt och en gradvis dämpning av kommunikationssignaler mellan enheter, vilket testar algoritmernas förmåga att hantera ofullständig information.

Stadsmiljön (se Figur 3.3) representerar en artificiell, människoskapad terräng där bebyggelsen utgör täta klusterstrukturer med mark mellan höga byggnader. I sådana miljöer blir kollisionsundvikande en högre prioritet, samtidigt som målobjekt

enkelt kan döljas bakom byggnadshörn vilket ökar betydelsen av hur drönarna positionerar sig och fördelar synfält mellan sig. Den vertikala dimensionen blir därmed avgörande för algoritmernas effektivitet på ett sätt som inte kan utvärderas i en plan miljö. Stadsmiljön är även relevant för urbana sök- och räddningsscenarier, exempelvis räddningsinsatser efter naturkatastrofer eller bränder, och kan samtidigt skapa utmanande kommunikationsförhållanden genom flervägsutbredning och varierande signaldämpning i urbana radiomiljöer (Longley, 1978).

Havsmiljön (se Figur 3.4) representerar slutligen en öppen miljö i princip helt fri från fysiska hinder. Den fungerar dels som ett baslinjefall där algoritmernas grundläggande sökeffektivitet kan utvärderas utan kollisionsbegränsningar, dels som en kontrastpunkt som möjliggör jämförelser mellan algoritmer som är optimerade för platta tvådimensionella sökytor och sådana som drar nytta av tredimensionell geometri. Sjöräddning utgör en tidskritisk och resurskrävande sök- och räddningsmiljö, där förseningar kan minska möjligheten att rädda personer i nöd och där insatser ofta påverkas av faktorer som väder, sjötilstånd och begränsad sikt (Martinez-Esteso, Castellanos & Gallego, 2025).

Tillsammans täcker de tre miljöerna ett brett spektrum av förutsättningar med avseende på hindertäthet, vertikal komplexitet och kommunikationsbegränsningar, vilket möjliggör en jämförande analys av samarbetsalgoritmernas robusthet och anpassningsförmåga över varierande scenarioklasser. Beskrivning av processen att generera världarna för de tre miljöerna ges i Appendix A.



Mindre skogsmiljö. Genererad skogsmiljö med världsstorlek 80 och upplösning 200. Den mindre storleken ger en mer kompakt testmiljö med relativt hög detaljgrad.

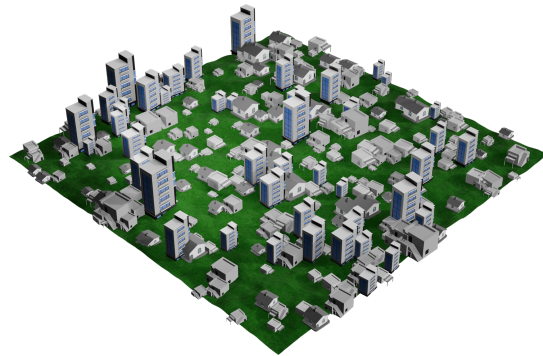


Större skogsmiljö. Genererad skogsmiljö med världsstorlek 150 och upplösning 375. Den större miljön täcker ett större område samtidigt som upplösningen skalas upp för att behålla en jämförbar detaljnivå.

Figur 3.2: Jämförelse mellan två genererade skogsmiljöer med olika världsstorlek och terrängupplösning. Den mindre miljön använder *World size* = 80 och *World resolution* = 200, medan den större miljön använder *World size* = 150 och *World resolution* = 375.

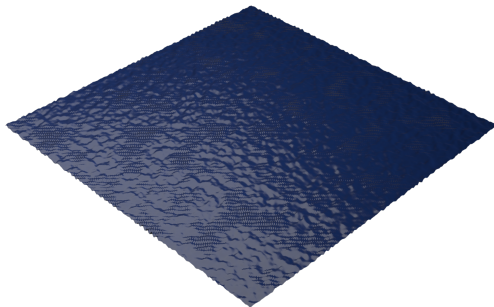


Mindre stadsmiljö. Genererad stadsmiljö med världsstorlek 80 och upplösning 200. Den mindre storleken ger en mer kompakt testmiljö med relativt hög detaljgrad.

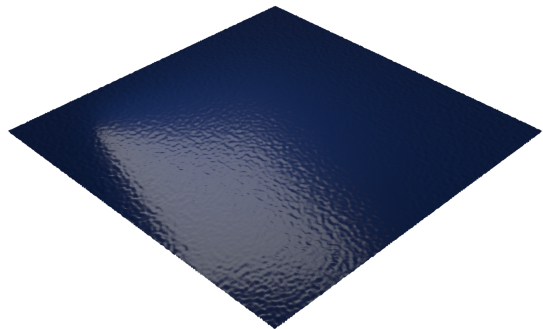


Större stadsmiljö. Genererad stadsmiljö med världsstorlek 150 och upplösning 375. Den större miljön täcker ett större område samtidigt som upplösningen skapas upp för att behålla en jämförbar detaljnivå.

Figur 3.3: Jämförelse mellan två genererade stadsmiljöer med olika världsstorlek och terrängupplösning. Den mindre miljön använder *World size* = 80 och *World resolution* = 200, medan den större miljön använder *World size* = 150 och *World resolution* = 375.



Mindre vattenmiljö. Genererad vattenmiljö med världsstorlek 80 och upplösning 200. Den mindre storleken ger en mer kompakt testmiljö med relativt hög detaljgrad.



Större vattenmiljö. Genererad vattenmiljö med världsstorlek 150 och upplösning 375. Den större miljön täcker ett större område samtidigt som upplösningen skapas upp för att behålla en jämförbar detaljnivå.

Figur 3.4: Jämförelse mellan två genererade vattenmiljöer med olika världsstorlek och terrängupplösning. Den mindre miljön använder *World size* = 80 och *World resolution* = 200, medan den större miljön använder *World size* = 150 och *World resolution* = 375.

3.5.2 Tredimensionella modeller av träd och byggnader

Trädmodellerna är hämtade från en fritt tillgänglig samling skapad av Xee (2022) distribuerad under royaltyfri licens. Sammanlagt sex lågpolymodeller används, bestående av flera typer av träd. Eftersom skogsvärldarna kan innehålla ett stort antal instanser, och hela geometrin senare voxeliseras till ett octree för drönarnas sökning, reducerades modellernas geometriska komplexitet innan de placerades i generatorns modellkollektion. Voxelisering är en process där man överför en geometri till en annan geometri där endast tredimensionella kuber används (se figur 3.5).

Byggnadsmodellerna är hämtade från fritt tillgängliga samlingar skapade av David (2025) samt Babia (2026), båda distribuerade under royaltyfria licenser. Samlingen består av fem distinkta lågpolymodeller representerande olika typer av urbana byggnader. Även dessa modeller förenklades och sparades i en **Collection** som refereras av stadsgeneratorns instansspridning. Den mer detaljerade asset-förberedelsen, inklusive användningen av Blenders **Decimate**-modifierare, beskrivs i Appendix A.5.

3.5.3 Automatisk exportering av världar och texturer

För att utvärderingen av samarbetsalgoritmerna ska vila på ett statistiskt meningsfullt urval krävs ett stort antal distinkta simuleringsvärldar. Att manuellt justera parametrar i Blender, applicera modifierare och exportera resultatet för varje enskild värld vore opraktiskt och tidskrävande. Av denna anledning utvecklades ett Python-skript som körs internt i Blender och automatiserar kedjan från parametersättning till färdig exporterad fil.

Skriptet utför fem huvudsakliga steg för varje genererad värld. Först dupliceras det procedurella källobjektet till en arbetskollektion, så att originalgeneratoren kan återanvändas. Därefter randomiseras nodgruppens ingångsparametrar inom fördefinierade intervall, inklusive *World seed*, vilket gör varje värld unik men reproducerbar. I nästa steg appliceras **Geometry Nodes**-modifieraren, eftersom den exporterade världen måste bestå av faktisk mesh-geometri snarare än en procedurell noddefinition. Mer detaljerad information om textureringsprocessen beskrivs i Appendix A.6.

Med detta ramverk genererades sammanlagt 62 världar fördelade enligt Tabell 3.1. Skogs- och stadsvärldar producerades i två storleksvarianter (small respektive large), 15 stycken vardera, motsvarande totalt 60 unika landbaserade världar. För havsvärldarna genererades endast en av varje storlek, eftersom havets visuella variation huvudsakligen utgörs av små amplitudvariationer i vågmönstret som inte är tillräckligt distinkta mellan instanser för att motivera ett större urval ur ett utvärderingsperspektiv. Den valda fördelningen reflekterar därmed en avvägning mellan beräknings- och lagringsåtgång och den variationsbredd som krävs för att täcka algoritmernas relevanta scenariotyper.

Tabell 3.1: Antal genererade världar per typ och storlek. *World size* anger sidlängd i Blender-enheter och *World resolution* antalet hörn längs varje sida av rutnätsmeshen.

Världstyp	Liten (80×80 , res 200)	Stor (150×150 , res 375)	Totalt
Skog	15	15	30
Stad	15	15	30
Hav	1	1	2
Summa	31	31	62

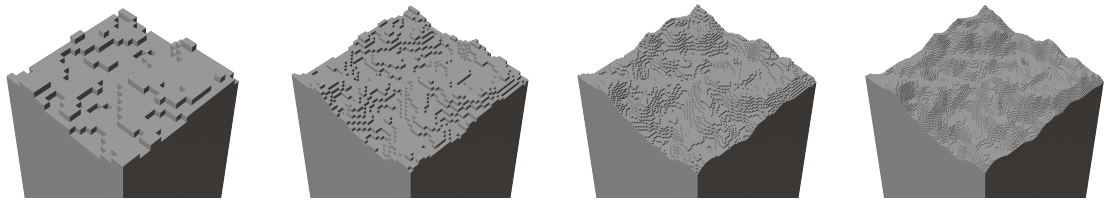
3.5.4 Generering av voxeliserade varianter av världar

Den exporterade GLB-representationen av varje värld utgör en rent visuell beskrivning lämplig för Gazebo-simulatorns renderingssteg, men den är inte direkt användbar för drönarnas sökalgoritmer. För att en drönare ska kunna avgöra om ett område är upptaget av ett hinder, kastas strålar mot omgivningen som markerar utforskade celler i en delad lägesbild. Den geometriska representationen möjliggör snabba spatiala uppslagningar. Av denna anledning konverteras varje exporterad värld i ett tvåstegsförfarande till ett octree, vilket är den datastruktur som faktiskt används av algoritmerna under simulering. Djupare beskrivning av hur de olika filtyperna används beskrivs i Appendix B.4.

Steg 1: Ytvoxelisering av mesh. I det första steget körs ett Python-skript baserat på `trimesh`-biblioteket som läser in världens GLB-fil och konverterar dess geometri till en uppsättning kubiska voxlar med konfigurerbar sidlängd. Voxeliseringen sker som en ren *ytvoxelisering* snarare än en fylld voxelisering, vilket innebär att endast de voxlar som skär meshens yta markeras som ockuperade. Detta är tillräckligt för kollisionsdetektion och raycast-baserad sensorsimulering, eftersom drönarna aldrig behöver representera objektens inre volymer. I samma steg transformeras koordinaterna mellan Blender-, glTF-, ROS- och Gazebo-konventionerna, och resultatet skrivs till en CSV-fil med centrumkoordinaten för varje ockuperad voxel.

Steg 2: Konvertering till OctoMap-binärformat. I det andra steget läses CSV-filen in och konverteras till en `octomap::OcTree`, som därefter serialiseras till en kompakt binär `.bt`-fil. Octree-strukturen gör det möjligt att utföra punktuppslagningar och raycasting betydligt effektivare än om voxeldata enbart hade lagrats som en linjär lista. Den detaljerade voxeliseringskedjan, inklusive koordinattransformationer och OctoMap-konvertering, beskrivs i Appendix B.1.

Val av upplösning. Voxlarnas sidlängd är en central parameter som styr avvägningen mellan rumslig precision och beräkningskostnad. Inledningsvis utvärderades en upplösning på 0,2 m, men detta gav för stora octree-filer för den tillgängliga hårdvaran. Den slutliga upplösningen sattes därför till 1,0 m, vilket skapade totalt 70 000 celler och höll minnesförbrukningen på en hanterbar nivå (se figur 3.5). Bakgrunden till upplösningvalet beskrivs mer detaljerat i Appendix B.3.



Voxelupplösning 4 m. Voxaliserad miljö med en voxelstorlek på 4 m, vilket ger en grov representation med låg detaljgrad men färre voxelceller.	Voxelupplösning 2 m. Voxaliserad miljö med en voxelstorlek på 2 m, vilket ger en bättre balans mellan detaljgrad och beräkningskostnad.	Voxelupplösning 1 m. Voxaliserad miljö med en voxelstorlek på 1 m, där mindre objekt och terrängdetaljer bevaras tydligare.	Voxelupplösning 0,5 m. Voxaliserad miljö med en voxelstorlek på 0,5 m, vilket ger högst detaljgrad men också flest voxelceller och högst beräkningskostnad.
---	--	--	--

Figur 3.5: Jämförelse mellan olika voxelupplösningar. En större voxelstorlek ger en grövre och mer kompakt representation av miljön, medan en mindre voxelstorlek bevarar fler geometriska detaljer men ökar antalet voxelceller och därmed minnes- och beräkningskostnaden.

3.6 Gemensam sök- och perceptionsinfrastruktur

Samtliga utvärderade algoritmer bygger på en gemensam runtime i form av en dedikerad ROS-2-nod, `octree_search_node`, vars uppgift är att förse drönarna med en konsekvent perceptions- och söklogik oavsett vilken beslutsstrategi som körs ovanpå. Noden ansvarar för att läsa in den voxeliserade världen, ta emot drönarnas position från flygkontrollen, utföra konbaserad raycasting nedåt mot omgivningen, registrera vilka delar av världen som har avsökts, detektera målobjekt samt larma vid risk för kollision. Genom att samla dessa funktioner i en enda delad nod undviks att samma logik måste återimplementeras i varje algoritm, vilket dels förenklar implementationen och dels säkerställer att de fyra testade algoritmerna utvärderas mot exakt samma sensor- och kollisionsmodell. Algoritmernas roll reduceras därigenom till att konsumera de händelser som noden publicerar (sökfeedback, måldetektion, kollisionslarm) och utifrån dessa fatta beslut om nästa rörelseriktning eller sökmål.

3.6.1 Inläsning och publicering av världen

Vid uppstart läser `octree_search_node` in de två voxelrepresentationerna som beskrevs i föregående avsnitt. CSV-filen används för att skapa en intern per-voxelstruktur där varje cell kan bära söktillstånd, medan den binära `.bt`-filen avserialliseras till en `octomap::OcTree` för effektiv raycasting. Kombinationen gör att noden både kan genomföra snabba spatiala frågor och samtidigt uppdatera varje voxels sökstatus under simulering.

Efter inläsningen skapar noden de ROS 2-publicerare och prenumeranter som krävs för visualisering, sökfeedback, kollisionslarm och måldetektering. Visualisering pub-

liceras som `MarkerArray`-meddelanden till RViz, medan algoritmerna tar emot sammanfattad sökfeedback, måldetektioner och kollisionssvarningar via separata topics. Noden prenumererar samtidigt på drönarnas positioner och aktuella målpositioner som skickas från `brain_node` för varje drönare, vilket gör att perceptionen kan uppdateras kontinuerligt när systemet körs.

3.6.2 Konbaserad raycasting och voxelintersection

Den centrala perceptionsmekanismen i noden är en konformad raycasting som körs varje gång en drönare publicerar sitt tillstånd. Strålnippet konstrueras vid uppstart och styrs främst av antalet strålar, `cone_rays`, samt konens halva öppningsvinkel, `cone_half_angle_deg`. I den aktuella konfigurationen används 500 strålar och en halv öppningsvinkel på 30°, riktade längs en nedåtriktad lokal sensoraxel. Detta modellerar en kamera- eller djupsensor monterad under drönaren.

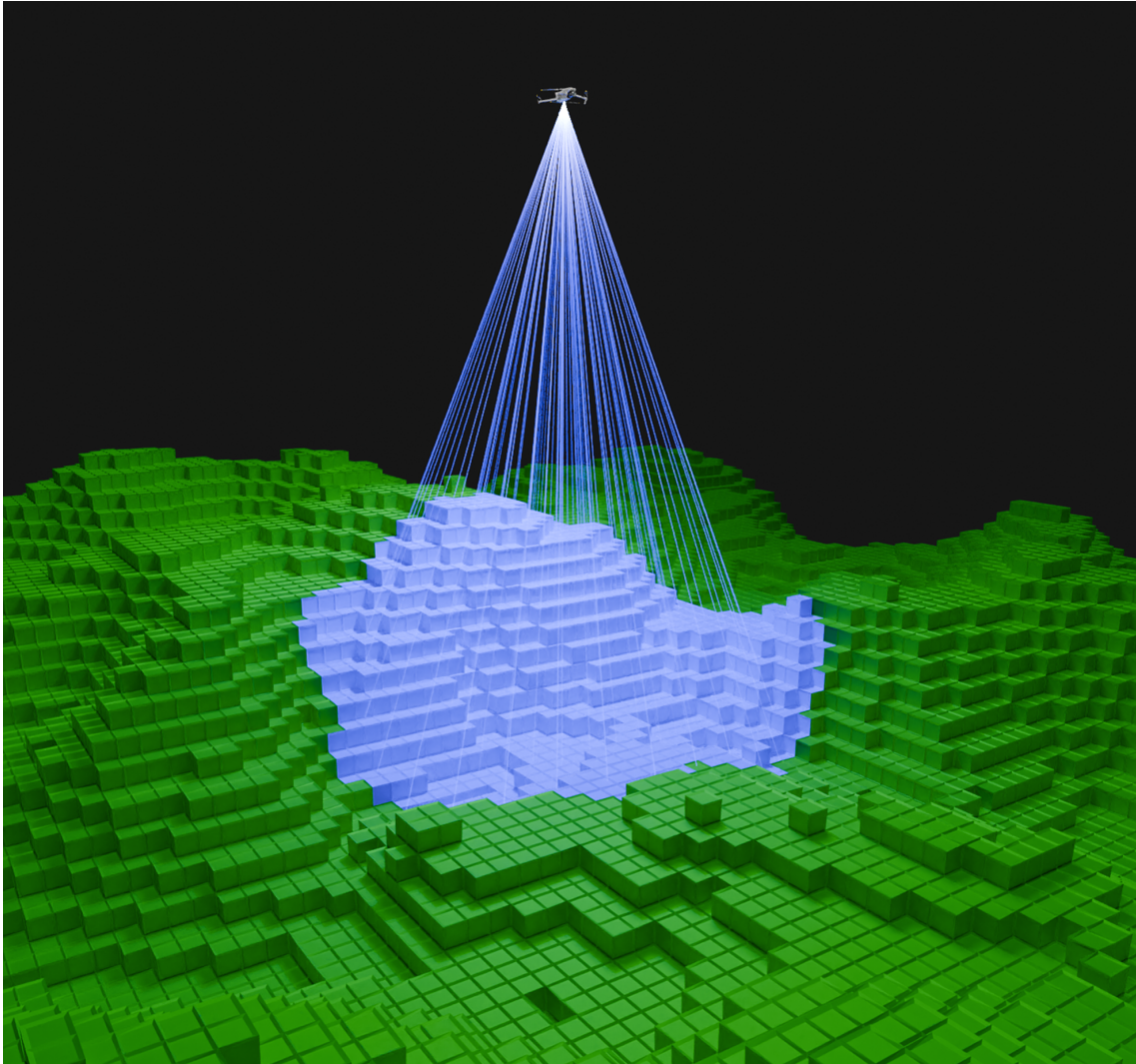
När drönarens position tas emot omvandlas den till simuleringens koordinatsystem och anpassas sedan till octree-strukturens lokala koordinater. Varje stråle kastas sedan mot den voxeliserade världen med `OctTree::castRay`, begränsad till `ray_max_range_m`, satt till 80 m. Resultatet blir en lista av strålsegment per drönare, där varje segment klassificeras som träff eller miss och vid träff innehåller motsvarande träffpunkt och avstånd. Den exakta strålkonstruktionen och koordinathanteringen beskrivs i Appendix C.2.

Octree-strukturen är central för att denna operation ska vara realtidsmässigt genomförbar. Eftersom `OctTree::castRay` traverserar trädet rekursivt och hoppar över stora rymdvolymmer på höga nivåer, blir kostnaden per stråle logaritmisk i antalet ockuperade voxlar snarare än linjär. Detta är en avgörande egenskap eftersom strålnippet med 500 strålar per drönare innebär en hög sammanlagd raycast-belastning per tick i en svärm med flera enheter, och en linjär kostnad per stråle skulle göra realtidsuppdateringen omöjlig vid simuleringens publiceringsfrekvenser.

3.6.3 Sökregistrering och tillståndsspårning

Varje raycast-träff översätts inte enbart till en visuell markör utan utlöser även en uppdatering av världens persistenta söktillstånd. Mekanismen expanderar varje träffpunkt till en sfärisk omgivning med radie `hit_radius_m`, satt till 7,0 m i den aktuella systemkonfigurationen. Alla voxelceller inom denna omgivning markeras som avsökta genom att deras `search_score` uppdateras, vilket gör att söktillståndet representerar ett område kring sensorträffen snarare än en enstaka voxel.

Sökpoängen utgör ett mått på hur väl en given voxel anses ha avsökts och varierar i intervallet [0.05, 1.0]. Implementationen stödjer en distansberoende poängsättning där närmare observationer kan ge högre poäng, men i den aktuella experimentkonfigurationen tilldelas samtliga praktiska träffar värdet 1,0. Detta val gjordes för att hålla utvärderingen hanterbar: en striktare avståndsviktning hade krävt betydligt lägre flyghöjd och därmed längre körtider. Den maximala flyghöjden är satt till 70 m vilket är för att förhindra att endast den algoritm som flyger högst alltid får



Figur 3.6: Visualisering av drönarens raycast-baserade genomsökning av den voxeliserade terrängen. De blåmarkerade voxelerna representerar det område som klassificerats som genomsökt efter en raycast-träff. Detta omfattar inte enbart de direkt träffade voxelerna, utan även de voxelceller som ligger inom radien `hit_radius_m` kring varje träffpunkt. I den aktuella systemkonfigurationen är denna radie satt till 7,0 m. De grönmärkade voxelerna representerar områden som ännu inte har genomsökts.

kortast genomsökningstid. Detaljerna bakom `search_score`-uppdateringen och avståndsmodellen beskrivs i Appendix C.3.

För att undvika onödig beräkning och kommunikationslast republiceras voxelvisualiseringen endast när söktillståndet faktiskt förändrats. Visualiseringsfärgen tolkar sökpoängen genom att gradvis blanda från grön (osökt) till blå (fullt avsökt), vilket gör det möjligt att följa svärmens framsteg över världen i realtid.

Parallellt med visualiseringen publicerar noden ett `SearchFeedback`-meddelande efter varje raycast-tick. Meddelandet innehåller den observerande drönarens identifierare, nuvarande position, direkta träffpunkter samt aggregerade fält för hur stor del av världen som har avsökts. Detta meddelande tas emot av `brain_node` för varje drönare och betyder att `brain_node` själv inte behöver beräkna täckningsgraden från voxeldata.

3.6.4 Måldetektion

Måldetekteringen utnyttjar samma raycast-resultat som sökregistreringen och kräver därmed ingen separat sensoroperation. Målpositionerna uppdateras via topicen `/targets/state`. När ett nytt `TargetState`-meddelande tas emot uppdateras den interna listan med mål, där varje mål kopplas till sitt unika ID och sin position.

I varje tick av `OnDroneState` jämförs samtliga raycast-träffpunkter med de aktuella målpositionerna. Om avståndet mellan en träffpunkt och ett mål understiger `hit_radius_m` klassificeras händelsen som en lyckad detektion. För att undvika dubbelrapportering inom samma tick samlas träffarna först i en lokal map som garanterar högst en detektion per (drönare, mål)-par. Därefter publiceras ett `TargetHit`-meddelande per unik detektion på topic `/octree/target_hits`. Meddelandet innehåller den detekterande drönarens identifierare, målets identifierare och träffpunkten i världskoordinater. Detta meddelanden skickas till `brain_node` där den registrerar att målet har hittats.

Värt att notera är att samma parameter `hit_radius_m` används som tröskel både för sökregistreringens voxelmarkering och för måldetekterings avståndstest. De två operationerna delar därmed samma effektiva sensoryta, vilket innebär att en justering av parametern påverkar båda funktionerna samtidigt. Detta är en medveten implementationskoppling som upprätthåller en konsekvent definition av vad som räknas som "inom sensorräckhåll" oavsett om det gäller en geografisk cell eller ett målobjekt.

3.6.5 Kollisionsundvikande

Utöver den nedåtriktade sökkonen utför noden även en separat omnidirektionell raycast vars syfte är att detektera närliggande hinder och tillhandahålla ett underlag för kollisionsundvikande. Riktningarna konstrueras vid uppstart från ett normaliserat $5 \times 5 \times 5$ -rutnät runt drönaren, vilket ger 124 riktningar som täcker rummet utan

separat sfärisk samplingslogik.

I varje tick utförs raycasts längs dessa riktningar med maximal räckvidd `collision_range_m`, satt till 20,0 m. Träffar omvandlas till en sammantvägd motstående vektor där närliggande hinder får större vikt än avlägsna hinder. Om minst ett hinder detekteras publiceras ett `CollisionAlert`-meddelande med den resulterande undvikanderiktningen, närmaste hinderavstånd och de observerade träffpunkterna. Den detaljerade beräkningen av kollisionsvektorn finns i Appendix C.4.

Själva sammanslagningen av hinderunderlaget med drönarens nuvarande sökmål sker inte i `octree_search_node` utan i den centrala `brain_node` som varje drönare kör. `brain_node` blandar den aktuella målriktningen från sökalgoritmen med undvikanderiktningen från kollisionslarmet, där vikten ökar när hindret är närmare. Den resulterande riktningen omvandlas därefter till ett hastighetskommando och skickas till flygkontrollen.

Utöver terränghinder hanterar `brain_node` även separation mellan drönarna i svärmen som en additiv komponent i samma rörelseberäkning. Om en annan drönare befinner sig inom en konfigurerbar separationsradie läggs en horisontell evasionsvektor till den slutliga rörelseriktningen. På så sätt hanteras både terrängkollisioner och drönare-drönare-separation på samma nivå i rörelselogiken.

Genom denna centraliserade hantering behöver de enskilda sökalgoritmerna varken implementera kollisionslogik mot terräng eller separationslogik mot andra drönare, utan kan begränsa sig till att producera waypoints utifrån sin respektive strategi. Detta upprätthåller den arkitektoniska uppdelningen mellan perception, beslutsfattande och rörelseutförande, och säkerställer att samtliga utvärderade algoritmer hanterar identisk kollisionsundvikande och drönare-drönare-separation utan möjlighet till oavsiktliga implementationsskillnader.

3.7 Målens representation och rörelse

För att efterlikna verkliga sök- och räddningsscenarier där det som söks är rörliga personer eller objekt modelleras målen i simuleringen inte som statiska markeringar utan som autonoma entiteter som kan förflytta sig över tid enligt en sannolikhetsbaserad rörelsemodell. Hanteringen av målen sköts av en dedikerad ROS 2-nod, `targets_manager_node`, som ansvarar för deras initiala placering, periodiska förflyttning, tillståndspubliserad samt registrering av när en drönare har detekterat ett mål. Genom att samla all mållogik i en enda nod säkerställs att samtliga drönare ser exakt samma målmängd och att målen rörelse är frikopplad från de testade sökalgoritmernas implementation.

3.7.1 Representation och placering

Varje mål representeras som en logisk ROS 2-entitet med identifierare, aktuell position, detektionsstatus och den senaste rörelseriktningen. Målen modelleras alltså inte som fysiska Gazebo-objekt med egna kollisionskroppar. Detta är tillräckligt eftersom detektionen sker genom raycast-baserad geometrisk uppslagning i `octree_search_node`, samtidigt som simuleringskostnaden hålls lägre.

Vid uppstart placeras målen antingen genom sampling av ockuperade voxlar från världens `voxels_csv`-fil eller, om sådan information saknas, från explicit angivna målpositioner i scenariot. Voxelbaserad placering säkerställer att målen hamnar på en navigerbar yta eller struktur snarare än i tomrum. Den fullständiga representationen och placeringslogiken beskrivs i Appendix D.1.

3.7.2 Rörelsemodell

Målen förflyttas i diskreta tidssteg via en timer som triggas vid frekvensen `update_rate_hz`, satt till 1,0 Hz i den aktuella konfigurationen. Rörelsen sker som en sannolikhetsbaserad slumpvandring på samma tredimensionella voxelnät som används av övriga delar av systemet. För varje aktivt mål undersöks dess $3 \times 3 \times 3$ -omgivning, och endast grannpositioner som motsvarar existerande ockuperade voxlar tillåts som rörelsealternativ. Därmed kan målen inte röra sig ut i tomrum eller passera genom föremål utan en sammanhängande yta att färdas på.

Sannolikhetsfördelningen för rörelseval definieras i `target_movement.json` via tre 3×3 -matriser, en för nivån under, samma nivå och nivån ovanför målet. Värdena är valda för att ge en sök- och räddningsrelevant rörelseprofil där målen oftast står still eller rör sig kort horisontellt, medan vertikala rörelser är ovanligare. Den detaljerade sannolikhetsmodellen finns i Appendix D.2.

3.7.3 Tröghet i rörelsen

För att undvika att målens rörelse uppfattas som en slumpvandring där riktningen ändras kraftigt varje tick används en tröghetskomponent baserad på senaste rörelseriktningen. Kandidater som ligger i samma riktning som föregående rörelse får högre sannolikhet, medan kandidater i motsatt riktning får lägre sannolikhet. Detta ger mjukare rörelsebanor där målen tenderar att fortsätta i samma riktning under flera tick innan de byter riktning.

I implementationen används parametern `heading_persist_gain`, satt till 0,8, för att styra hur starkt tidigare rörelseriktning påverkar nästa val. Efter att sannolikheterna justerats normaliseras de över de tillåtna grannpositionerna och en ny målposition väljs. Den exakta beräkningen av tröghetstermen beskrivs i Appendix D.3.

3.7.4 Tillståndspublicering och detektion

Efter varje rörelsetick publicerar noden ett `TargetState`-meddelande på topic `/targets/state`. Meddelandet innehåller samtliga målidentifierare och deras aktuella världspositioner, vilket används av `octree_search_node` för måldetektion och av drönarnas `brain_node` för loggning och utvärdering. Publiceringen sker med `transient_local` QoS, vilket gör att sent anslutande noder direkt får den senast publicerade målmängden.

När `octree_search_node` detekterar att en drönares raycast passerat tillräckligt nära ett mål publiceras ett `TargetHit`-meddelande på `/octree/target_hits`. `targets_manager_node` markerar då målet som upptäckt, vilket gör att det slutar förflyttas i efterföljande tick men ligger kvar i mållistan för utvärdering. Detaljer om publicering och detektionshantering finns i Appendix D.4.

3.8 Användning av systemet

Systemet är utformat för att stödja ett strukturerat arbetsflöde där en utvecklare implementerar egna beteenden enligt ett givet gränssnitt, genomför simuleringar baserade på fördefinierade konfigurationer, och slutligen analyserar resultaten med hjälp av tillhörande analysverktyg.

Denna uppdelning möjliggör en tydlig separation mellan utveckling, exekvering och utvärdering. Genom att dessa delar hanteras oberoende av varandra underlättas både iterativ utveckling och jämförelse mellan olika beteenden. Samtidigt bidrar upplägget till ett reproducerbart arbetssätt där experiment kan återskapas och analyseras under kontrollerade förhållanden.

Att utveckla ett nytt beteende kräver viss förståelse för ROS 2, men systemet är utformat för att minimera behovet av djupgående kunskap om dess interna funktionalitet. Gemensam funktionalitet, exempelvis kommunikation och grundläggande beteendestruktur, är centraliserad i systemet. Utvecklaren behöver därför främst implementera den specifika logiken för sitt beteende, vilket förenklar processen att skapa och integrera nya algoritmer.

3.8.1 Systemstöd för beteenden

Systemet tillhandahåller en centraliserad funktionalitet som delas mellan alla beteenden, vilket innebär att ett beteende har direkt tillgång till relevant information och styrfunktioner utan att någon ytterligare integration krävs. Informationen och styrfunktionerna kommer från den `brain_node` som drönaren använder.

Ett beteende har tillgång till information om drönarens identitet och dess kontext inom svärmen, bland annat ett unikt ID, svärmens storlek samt parametrar definierade i experimentkonfigurationen. Vidare tillhandahålls information om den simulerade miljön, inklusive sökområdets utbredning, terränghöjd samt funktioner

för att omvandla mellan olika koordinatsystem.

Under körning kan beteendet kontinuerligt hämta aktuell tillståndsinformation, exempelvis position och hastighet, samt information om upptäckta mål. För att styra drönaren tillhandahåller systemet högre nivåns kommandon, exempelvis att flyga till en position, sätta hastighet eller initiera start. Både datan och kommandon är tillgängliga genom integreringen av `brain_node` och beteendekoden.

3.8.2 Gränssnitt och krav

Varje beteende måste implementera ett gemensamt gränssnitt som definierar dess livscykel och interaktion med systemet. Följande funktioner utgör kärnan i detta gränssnitt.

Förberedelser: Denna funktion används för att utföra initialisering innan simuleringen startar. Detta kan inkludera att etablera kommunikation eller att flyga till en fördefinierad startposition. Funktionen anropas av den centraliserade delen innan drönaren anses vara redo att delta i simuleringen.

Tick: Denna funktion anropas kontinuerligt av drönarens ROS 2-nod med en given frekvens. Här implementeras beteendets huvudsakliga logik, där beslut fattas baserat på aktuell tillståndsinformation, exempelvis val av nästa målposition.

Målposition: Varje beteende ansvarar för att kontinuerligt uppdatera sin aktuella målposition, vilket i detta fall beskriver positionen drönaren är på väg till. Denna information används av den centraliserade delen för att inkludera målpositionen i den telemetry som samlas in under simuleringen, vilket möjliggör analys av beteendets beslutsprocess.

Granninformation: Varje beteende ansvarar för hur information om andra drönare hanteras och delas. Detta möjliggör flexibilitet, då olika beteenden kan implementera olika strategier för kommunikation och samordning. Samtidigt används denna information av den centraliserade delen för funktioner som kollisionsundvikelse.

Mål hittat: När ett mål upptäcks vidarebefordras denna information till beteendet. Detta möjliggör att beteenden kan anpassa sitt agerande, exempelvis genom att ändra strategi.

Avaktivering: När simuleringen avslutas meddelas beteendet, vilket möjliggör eventuella avslutande operationer. De flesta kritiska åtgärder hanteras dock av den centraliserade delen.

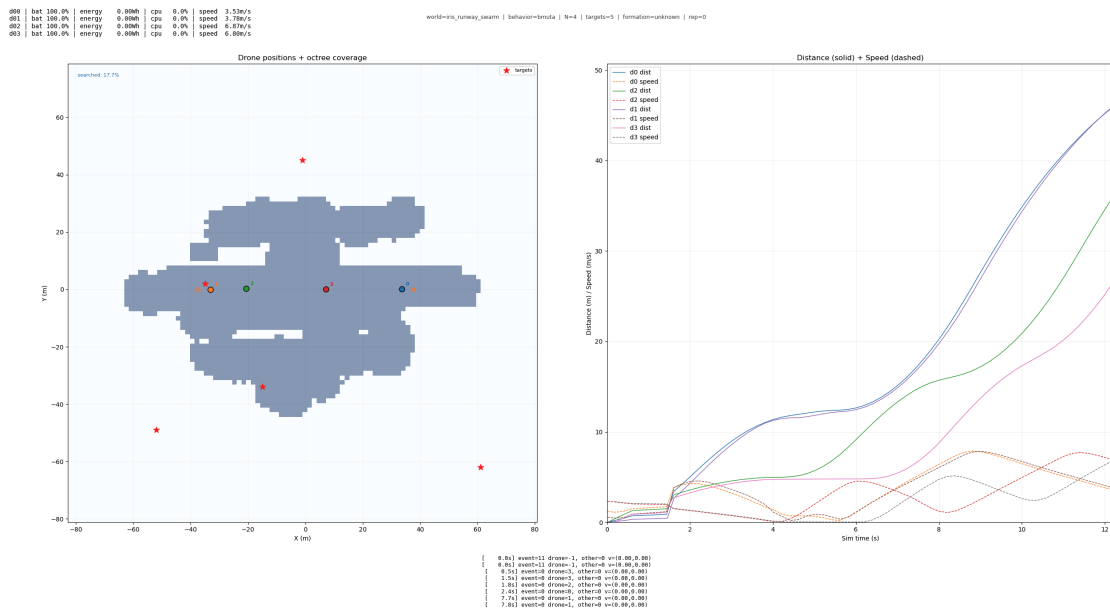
Övrig funktionalitet: Utöver det gemensamma gränssnittet kan varje beteende implementera egen kommunikation mellan drönare. Detta är avsiktligt inte centraliserat för att möjliggöra jämförelse mellan olika kommunikationsstrategier.

3.8.3 Integration i systemet

För varje nytt beteende ska en beteendemapp skapas och läggas i beteendekatalogen. Denna separering är kritisk för att ge varje beteende sin egen miljö i filsystemet där utvecklaren av beteendet har fria händer att lägga till filer och strukturera inom mappen som passar bäst för det implementerade beteendet. Systemet är utformat för att automatiskt identifiera och ladda dessa, vilket möjliggör enkel integration utan att befintlig kod behöver modifieras.

3.8.4 Exekvering

Exekvering av ett beteende görs enklast med hjälp av en scenario-fil. I scenario-filen specificeras in-parametrar till simuleringen, som till exempel vilket beteende som ska exekveras, drönarens start-positioner, målens utgångspositioner, rörelse-modeller och vilken värld som simuleringen ska använda. För att underlätta debugging och få ut mer information kring beteenden under exekveringen, öppnas ett gui-fönster upp under exekveringen som visar visualiserar telemetridatan som sparas in under simuleringens gång.



Figur 3.7: GUI för visualisering av insamlad telemetri under uppdragets gång.

Det finns även möjlighet att sätta upp en beteende-specifik gui som även den kommer öppnas upp automatiskt när man kör ett scenario. Detta är hjälpsfullt då många beteenden har data som är intressant för visualiseringen, men som inte är en del av den centraliserade telemetrin som sparas oberoende av beteende.

Utöver dessa gränssnitt går det även att följa drönarna genom en RVIZ2-visualisering. Under uppdragets gång publiceras data som RVIZ2 prenumererar på, och presenterar sedan denna data i ett visuellt gränssnitt som ger en tredimensionell överblick som man inte får från de andra gränssnitten.

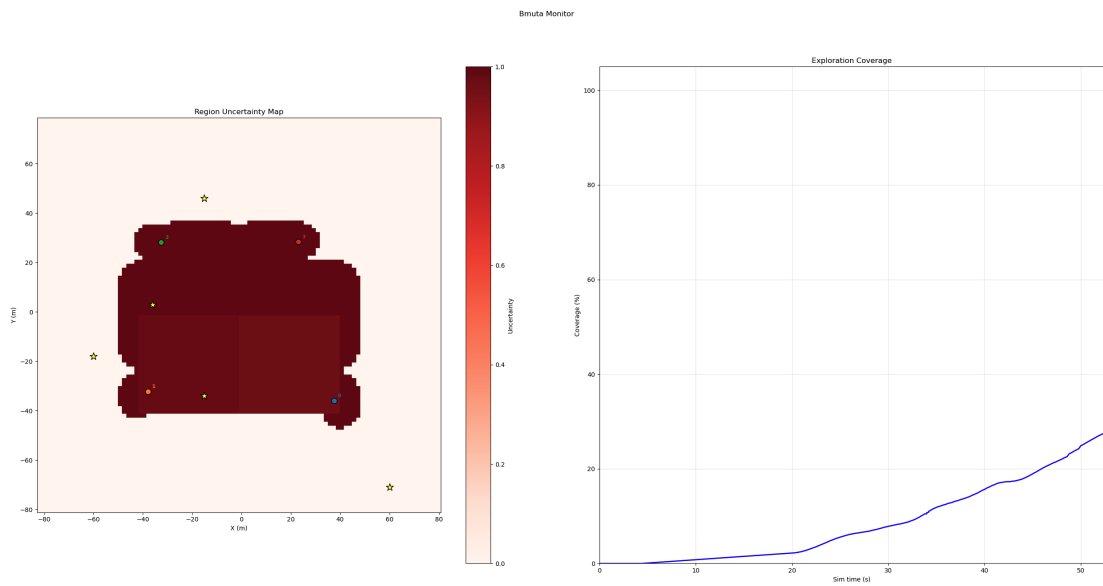


Figure 3.8: GUI för visualisering av BMUTA under uppdragets gång.

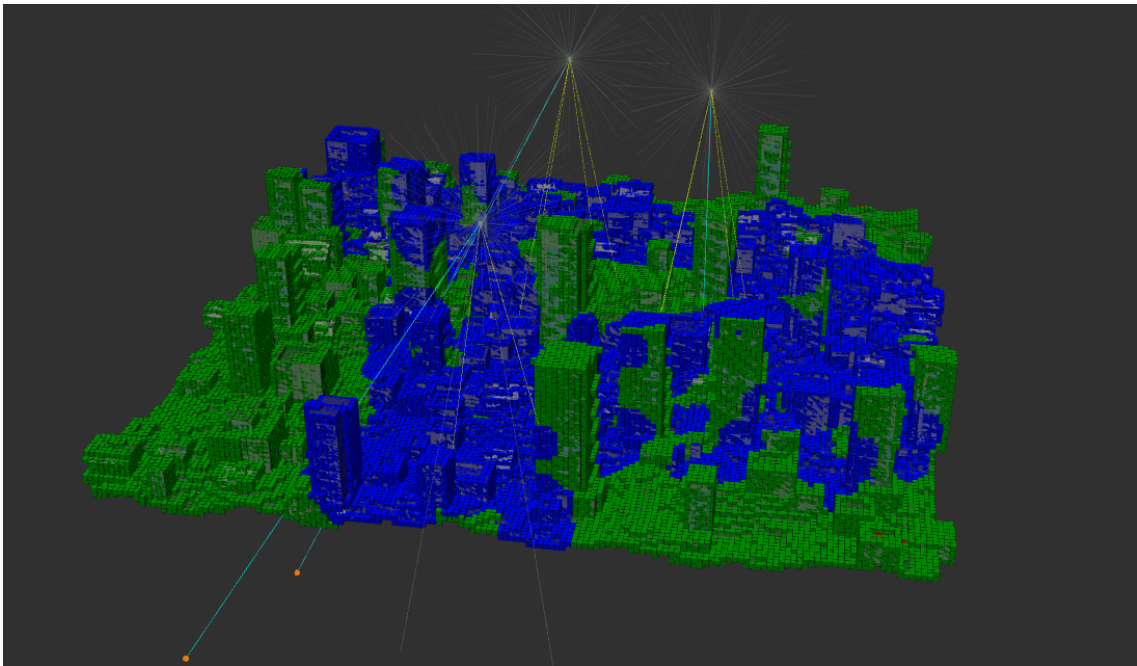


Figure 3.9: Skärmdump från GUI för visualisering av uppdraget via RVIZ2.

3.8.5 Köra experiment

En central funktion i systemet är möjligheten att genomföra experiment bestående av flera simuleringar. Medan enskilda körningar främst används för utveckling och debugging, möjliggör experiment en systematisk utvärdering av beteenden under varierande förutsättningar. Ett experiment definieras i en experimentfil och består av en uppsättning parametrar, eller dimensioner, som varierar mellan körningar. Till

skillnad från en scenariofil, som beskriver en enskild simulering, specificerar experimentfilen hur flera simuleringar ska genereras och genomföras.

Följande dimensioner kan varieras inom ett experiment:

- **Antal drönare:** Antalet drönare i simuleringen.
- **Antal mål:** Hur många mål som ska lokaliseras innan simuleringen avslutas.
- **Målrörelse:** En separat modell som beskriver hur målen rör sig över tid.
- **Beteenden:** Vilka algoritmer som ska utvärderas.
- **Världstyper:** Typ av miljö, exempelvis stad eller skog.
- **Random seeds:** Används för att säkerställa reproducerbarhet mellan körningar.
- **Startformation:** Hur drönarna initialt placeras i miljön.

Denna struktur möjliggör systematiska jämförelser mellan beteenden och andra dimensioner under kontrollerade och reproducerbara förhållanden.

3.8.6 Analys av experiment

Efter att ett experiment har genomförts lagras telemetridata från varje simulering. Denna data kan analyseras med hjälp av systemets inbyggda analysverktyg, vilket genererar grafer och tabeller som underlättar tolkning av resultatet. Det finns även möjlighet att utveckla egna analysverktyg eller visualiseringar baserade på den insamlade datan. Detta kräver dock en mer djupgående förståelse av datastrukturen och behandlas därför inte vidare i denna rapport.

4

Experimentell utvärdering

Med syfte att utvärdera det utvecklade systemets funktionalitet och tillämpbarhet har fyra olika beteenden implementerats oberoende av varandra. Dessa har därefter simulerats inom ramen för ett definierat experiment och analyserats med hjälp av systemets analysverktyg utifrån ett antal utvärderingsmått.

4.1 Testade algoritmer

Fyra olika beteenden har implementerats och utvärderats för att demonstrera systemets funktionalitet och flexibilitet. Dessa representerar olika strategier för sökning inom ett sök och räddnings-uppdrag och har valts för att ge en varierad bild av hur systemet kan användas för att jämföra algoritmer med olika egenskaper.

Samtliga beteenden har utvecklats oberoende av varandra enligt det gränssnitt som systemet tillhandahåller, vilket möjliggör en konsekvent och rättvis jämförelse mellan dem inom samma experimentella ramverk.

4.1.1 Bee-inspired multi-UAV teaming algorithm

En av de valda metoderna är BMUTA, som bygger på bioinspirerade principer för samverkan mellan autonoma enheter och är relevant som representant för strategier där samarbete uppstår genom lokala interaktioner och begränsat informationsutbyte. Enligt Kurdi m. fl. (2020) fungerar BMUTA på det sättet att varje drönare kan ha två olika roller: sökare och räddare, och detta ändras dynamiskt under arbetets gång. Sökaren har till uppgift att söka efter ett mål, och räddaren har som uppgift att rädda de mål som sökaren hittar. Detta system använder sig av pelare för att markera områdets status till de andra drönarna.

Systemet fungerar på det sätt genom att alla drönare börjar som sökare och sökningsområdet fördelas bland de befintliga drönarna. Sökningsområdet delas in i regioner där varje region innehåller ett visst antal block och detta initieras vid programstart. Varje region har tre olika lägen:

Utforskad: Antalet block i regionen är utforskade

Pågående: Ett av blocken i regionen har tillståndet tom eller godkänd

Sökt: Alla block i regionen har tillståndet tom eller godkänd

Varje block kommer att som mest ha ett mål och kan anta fyra olika lägen:

Outforskad: Blocket har inte blivit genomsökt

Pågående: Blocket har blivit genomsökt och ett mål har hittats

Godkänd: Blocket har blivit genomsökt och målet har räddats

Tom: Blocket har blivit genomsökt och inget mål har hittats

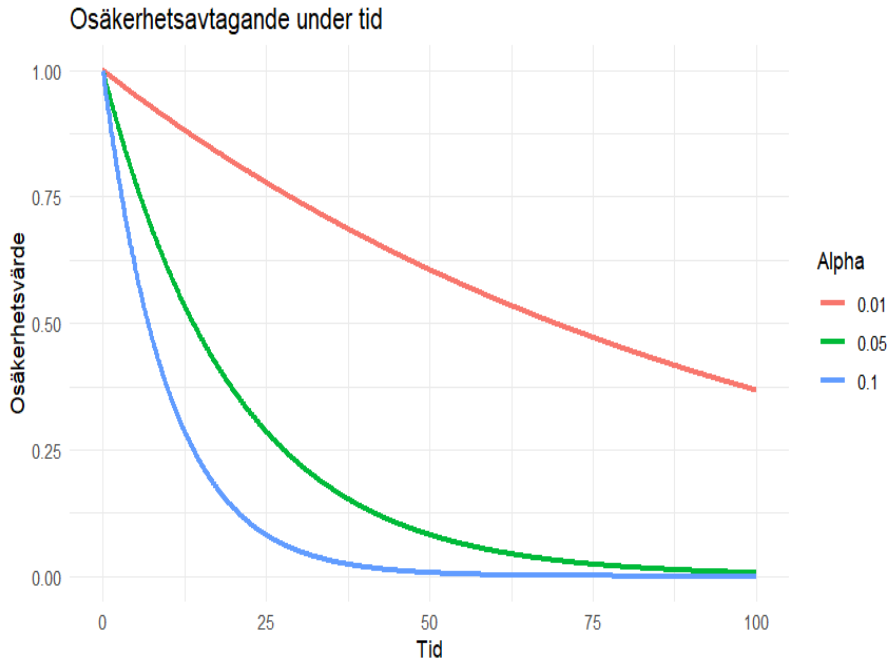
Sökaren söker efter de regioner som har läget outforskad. När sökaren hittar ett sådant område kommer den att uppdatera det området till pågående och börja söka efter mål. Sökaren genomsöker hela området oavsett om den hittar ett mål eller inte. Hittar sökaren ett mål, kommer den att placera en pelare där för att markera målets position. Räddaren har som uppgift att leta efter de pelare som sökaren har placerat för att rädda målet.

För att anpassa algoritmen till projektets antaganden om ett känt antal mål har vissa delar från BMUTA-algoritmen ändrats. Enligt Kurdi m. fl. (2020) kan högst ett mål finnas i varje region, medan detta arbete tillåter flera mål per region.

Detta arbete fokuserar på sökdelen av algoritmen och inkluderar därför inte räddardelen. Arbetet kommer även att exkludera blocket och kommer endast att använda sig av regioner, då denna del av arbetet enbart fokuserar på hur drönarna prioriterar valet av områden. Detta medför också att pelarsystemet inte ingår i programmeringen.

Drönarnas val av nästkommande sökregion baseras på flera kriterier som testas i olika steg, vid start av programmet väljer drönarna den region närmast till sig själv och om några skulle välja samma lämnar drönaren med högst drönar ID och väljer en ny region. Efter detta väljs regioner utifrån dessa steg som även beskrivs visuellt i Figur 4.1:

1. Först kollas statusen på alla regioner och de med status *outforskad* eller status *sökt* och ett entropivärde under 0,5 filtreras fram som möjliga kandidater.
2. Avståndet till centrum beräknas för samtliga kvarvarande kandidatregioner.
3. De regioner med status *sökt* tilldelas ett straffvärde som adderas till det beräknade avståndet, proportionerligt mot regionens entropivärde.
4. Ett ytterligare straffvärde adderas för de regioner där en annan drönare befinner sig närmare och är bättre placerad att söka av regionen efter att ha avslutat sin nuvarande uppgift.
5. Baserat på ovanstående väljer drönaren den region med det lägsta sammanlagda värdet.



Figur 4.2: Avtagande för olika α .

4.1.2 Uncertainty-Based Baseline Search

Uncertainty-Based Baseline Search (UBS) är den minst komplexa algoritmen som har valts att testas och används som ett typ av basfall för hur en algoritm kan utvecklas inom systemet. Algoritmen bygger på en delad osäkerhetskarta (*uncertainty map*) över sökområdet. Kartan är ett 100×100 rutnät där varje cell initialt tilldelas osäkerhetsvärdet $U(c) = 1,0$ (utforskad). När en drönare passerar inom sin observationsradie markeras berörda celler som utforskade ($U(c) = 0,0$). Denna karta är i två dimensioner och det betyder att observationsradien är en estimering.

Målval

Varje drönare väljer kontinuerligt nästa målposition genom att maximera en poängfunktion över alla utforskade celler:

$$s(c) = U(c) - \alpha \cdot d(c) - \beta \cdot R(c) + \varepsilon \quad (4.3)$$

där $d(c)$ är det euklidiska avståndet från drönaren till cell c , och $R(c)$ är repulsionskraften från övriga drönare:

$$R(c) = \sum_{j \neq i} \exp\left(-\frac{\|c - p_j\|^2}{\sigma^2}\right) \quad (4.4)$$

Termen $\varepsilon \sim \mathcal{U}(0, 0,07)$ är ett litet slumpmässigt brus som bryter symmetri vid likvärdiga celler. Parametrarna är satta till $\alpha = 0,01$, $\beta = 20,0$ och $\sigma = 5,0$ m. Parametrarna har valts genom enklare manuell testning på ett urval scenarier och representerar värden som gav rimlig prestanda — någon systematisk optimering av

parametrarna har inte genomförts. Drönaren planerar om till ett nytt mål när den befinner sig inom 5 m från det nuvarande.

Kommunikation

Drönarna delar två typer av information med varandra: **(1)** position, som används för att beräkna repulsionskraften $R(c)$, samt **(2)** osäkerhetskarta, där varje drönare vid mottagning tillämpar ett elementvist minimum mot sin egen karta — ett område utforskat av en drönare betraktas därmed som utforskat av hela svärmen.

Ifall hela *uncertainty map* har utforskats (alla celler har värdet $U(c) = 0,0$) och alla mål inte har upptäckts, sätts alla celler till initialvärdet $U(c) = 1,0$ och sökningen fortsätter därför tills alla mål har upptäckts.

4.1.3 Particle Swarm Optimization

En av de valda metoderna är Particle Swarm Optimization (PSO), som i detta arbete används som ett decentraliserat sökbeteende för en svärm av drönare. PSO är en populationsbaserad optimeringsalgoritm som inspireras av naturliga svärmbeteenden, till exempel flockar av fåglar eller stim av fiskar, där individer anpassar sin rörelse utifrån både egen erfarenhet och information från omgivningen. I denna metod representeras varje drönare som en partikel i sökområdet, och rörelsen styrs inte av en central planerare utan istället av en kombination av den egna rörelsehistoriken, det egna bästa identifierade området, samt information från närliggande drönare. PSO är därför högst relevant för kollaborativ intelligens, eftersom varje drönare fattar egna beslut utifrån lokal information samtidigt som svärmen påverkas av begränsad informationsdelning mellan agenterna.

Sökområdet delas in i rutnät där varje cell representerar en del av området. Varje cell har ett osäkerhetsvärde som säger om området fortfarande behöver genomsökas. Vid start betraktas hela området som outforskat, vilket innebär att alla celler har en hög osäkerhet. När en drönare flyger över ett område markeras cellerna inom dess observationsradie som genomsökta. Denna radie påverkas av drönarens höjd i förhållande till underlaget, vilket innebär att den förväntade täckningen från en position varierar beroende på miljön.

Utöver täckningskartan används även en sökbarhetskarta som beskriver hur lämpliga olika delar av området är att söka i. Områden där marken förväntas kunna observeras tydligt och effektivt, exempelvis öppna ytor eller vägar och fält, ges högre värde. Områden med skog, täta träd, kraftiga höjdskillnader eller hustak i stadsmiljö är mindre lämpliga. Om terräng finns tillgänglig används den därför för att identifiera celler som är mindre sökbara, exempelvis på grund av höjdskillnader eller låg öppenhet. Celler som bedöms som otillgängliga väljs inte som mål, medan övriga celler prioriteras utifrån sin sökbarhet.

Observationen modelleras som en förenklad täckningsmodell snarare än en detaljerad sensorsimulering. Drönarens position är känd i simuleringen, och celler inom observationsradien markeras som genomsökta. Modellen använder däremot inte

raycasting eller kamerabaserad siktkontroll och kontrollerar därför inte om en cell skymms av exempelvis träd, byggnader eller andra objekt. Detta gör modellen lämplig för att studera och jämföra sökstrategier på en högre nivå, men mindre lämplig för att dra slutsatser om faktisk sensorprestanda i en verklig miljö.

Varje drönare underhåller en egen täckningskarta över området och skickar regelbundet denna till andra drönare. När en drönare tar emot en annan drönares karta kombineras den med den egna kartan genom att redan genomsökta celler behålls som genomsökta. På så sätt kan drönarna undvika att prioritera områden som redan har täckts av andra. Informationsutbytet är därmed begränsat till sammanfattad sökstatus, positioner, målpositioner och bästa hittade sökområden, snarare än fullständig central styrning.

När en drönare ska välja nästa mål beräknas ett värde för möjliga positioner i rutnätet. Detta värde baseras främst på hur många utforskade celler som kan observeras från positionen, men påverkas även av sökbarheten, om området nyligen har besökts och om andra drönare eller deras planerade mål redan ligger nära samma område. Därmed skapas en spridningseffekt där drönarna tenderar att fördela sig över sökområdet utan central tilldelning.

Den PSO-baserade målberäkningen består av flera delar: en tröghetskomponent, där drönaren tenderar att fortsätta i samma rörelseriktning som tidigare. En kognitiv komponent, där drönaren påverkas av det bästa område som den själv har identifierat. En social komponent, där drönaren påverkas av andra drönares bästa identifierade områden, samt en frontier-komponent, som pekar mot den mest lovande utforskade cellen enligt den aktuella täcknings- och sökbarhetsinformationen. I implementationen används inte ett globalt bästa område för hela svärmen, utan ett lokalt bästa område baserat på drönare inom ett visst avstånd, vilket gör samarbetet lokalt och decentraliserat.

Dessa komponenter kombineras till en ny rörelseriktning för drönaren. Steglängden begränsas så att målet inte väljs alltför långt bort vid varje uppdatering, och målet begränsas även till sökområdets gränser samt hålls på den aktuella flyghöjden. Om det beräknade målet ligger för nära en annan drönare eller en annan drönares mål väljs ett säkrare alternativ.

Varje drönare uppdaterar kontinuerligt sitt *personal best*, alltså den bästa position eller region som den själv har hittat utifrån den aktuella värderingsfunktionen. Detta bästa område skickas till andra drönare tillsammans med ett värde som beskriver hur lovande området är. När en drönare tar emot sådana värden från andra drönare följs de bara om de bedöms vara tillräckligt bättre än det egna alternativet och om området inte redan är för trångt med andra drönares mål i närheten. Detta motverkar att hela svärmen samlas kring samma punkt.

I detta arbete har PSO anpassats till ett täckningsproblem snarare än ett klassiskt optimeringsproblem med en fast global målfunktion. Målet är inte att hitta

en enskild optimal punkt, utan att effektivt minska osäkerheten över hela sökområdet. Därför kombineras PSO-principerna med en täckningskarta, sökbarhetsvärden, besöksstraff och separationsstraff. Algoritmen avslutas när en tillräckligt stor andel av området har observerats. I implementationen används en gräns där uppdraget betraktas som färdigt när ungefär 99% av täckningskartan är genomsökt.

Sammanfattningsvis fungerar PSO-beteendet som en decentraliserad samarbetsstrategi där varje drönare själv väljer mål, men där besluten påverkas av information från resten av svärmen. Detta gör att svärmen kan sprida ut sig, undvika onödigt överlappning och samtidigt utnyttja lovande områden som andra drönare har identifierat.

4.1.4 Recontamination Aware Search

Den fjärde metoden, RAS, utvecklades inom detta arbete. Till skillnad från de övriga metoderna arbetar RAS direkt på den tredimensionella voxelrepresentationen och modellerar uttryckligen att mål i ett sök och räddning-scenario kan röra sig. Ett område som nyligen avsökts kan därför inte betraktas som permanent säkert, eftersom mål senare kan röra sig in från angränsande celler. RAS representerar denna osäkerhet som ett spatialt fält som diffunderar över tid enligt samma rörelsesannolikheter som målen antas följa.

Gasfältmodell. Centralt i algoritmen är ett *gasfält*, implementerat i klassen `GasField`, som tillordnar varje ockuperad voxel i världen ett skalärt osäkerhetsvärde. Fältet initieras med en uniform fördelning över samtliga ockuperade celler så att summan av fältet är ett, vilket konceptuellt motsvarar ett antagande att ett mål kan finnas var som helst i världen med samma sannolikhet vid uppdragets start. Den semantiska tolkningen är att fältets värde i en cell är proportionellt mot sannolikheten att en godtycklig instans av en oändligt stor population av rörliga mål befinner sig i just den cellen vid den aktuella tidpunkten.

För att modellera målens rörelse använder algoritmen de tre lokala rörelsematriser som beskrivs i Avsnitt 3.7. Matriserna motsvarar förflyttningar till voxel i nivån under, på samma nivå och nivån över den aktuella cellen, och bildar tillsammans en tredimensionell rörelsekärna av storleken $3 \times 3 \times 3$. Utifrån denna kärna byggs en gles diffusionsmatris där varje element anger sannolikheten att osäkerhet förflyttas mellan två angränsande celler under ett diffusionssteg. Varje rad normaliseras till summan ett, vilket bevarar fältets totala massa. Diffusionen kan därmed utföras som en gles matris-vektor-multiplikation $\mathbf{g}_{t+1} = \mathbf{D}\mathbf{g}_t$.

När en drönare flyger över en del av världen och raycast-träffar registreras av `octree_search_node` mottar gasfältet ett `SearchFeedback`-meddelande och tillämpar en *absorption* av osäkerheten i en kubisk omgivning kring varje träffpunkt. Absorptionen sker multiplikativt, cellens osäkerhetsvärde multipliceras med $(1 - s)$, där s är samma distansviktade observationspoäng som beskrivs i avsnittet om sökregistrering. Eftersom `search_score_far_m` i den aktuella konfigurationen är satt till

ett mycket högt värde resulterar varje träff i att osäkerheten i den lokala omgivningens rensas helt. Effekten är att drönarens sökoperation ses som att den *suger upp gas*, området under drönaren töms på osäkerhet, men denna kommer gradvis tillbaka allt eftersom diffusionen sprider osäkerhet från omgivande icke-avsökta celler. Det är denna återinträngningseffekt som gett algoritmen dess namn: *recontamination* avser hur ett tidigare avsökt område successivt återförorenas av osäkerhet från sin omgivning.

Frontdetektering. En central observation är att den ackumulerade diffusionseffekten domineras av cellerna vid *fronten* mellan avsökta och icke-avsökta områden, eftersom det är längs denna gräns som osäkerhet diffunderar in i de redan avsökta regionerna. Idén att använda gränsområden som navigeringsmål förekommer även inom frontier-baserad utforskning (Yamauchi, 1997). Att direkt minska den totala mängden gas i världen är därmed ett mindre effektivt mål än att minska antalet frontceller, eftersom en kortare front innebär att färre celler bidrar till återförorening per tidsenhet. Algoritmen identifierar därför aktivt frontcellerna via metoden `extract_frontier_cells`, som inspekterar samtliga grannparrelationer i diffusionsmatrisen och klassificerar en cell som frontcell om dess gasvärde står i ett tillräckligt skarpt förhållande till någon av sina grannar. Konkret klassificeras paret som frontkandidat om förhållandet $\min(g_i, g_j) / \max(g_i, g_j) < 0,5$, vilket innebär att en cell räknas som front om någon angränsande cell har minst dubbelt så hög gasnivå.

För att undvika att vertikala gaskontraster mellan tomma luftvolymmer och tätt byggda markvolymmer felaktigt klassificeras som relevanta frontmål filtreras grannparen tidigt i sökfasen så att enbart par med skild horisontell position betraktas. Algoritmen använder en tvådimensionell frontrepresentation där alla frontceller längs samma vertikala kolumn kollapsas till en enskild representant. När den horisontella täckningsgraden överstiger tröskeln `vertical_frontier_coverage_threshold` (satt till 0,8) övergår algoritmen till en fullt tredimensionell frontdetektering där även vertikala kontraster beaktas, vilket är nödvändigt för att avsluta sökningen kring höga objekt såsom byggnader i stadsmiljöerna.

Köbaserad waypoint-planering. Drönarens rörelse styrs av en framåtblickande kö av waypoints, hanterad av klassen `RecontaminationPlanner`. Köns längd är konfigurerbar via parametern `number_of_waypoints_predicted` (satt till 5) och fylls på dynamiskt vid frekvensen `planner_rate_hz` (2,0 Hz) eller när drönaren har avverkat tillräckligt mycket av den befintliga banan. Endast cellpositioner som identifierats som frontceller får utgöra waypoint-kandidater, vilket säkerställer att drönaren kontinuerligt rör sig längs gränsen av sitt avsökta område snarare än upprepar besök i redan rensade regioner.

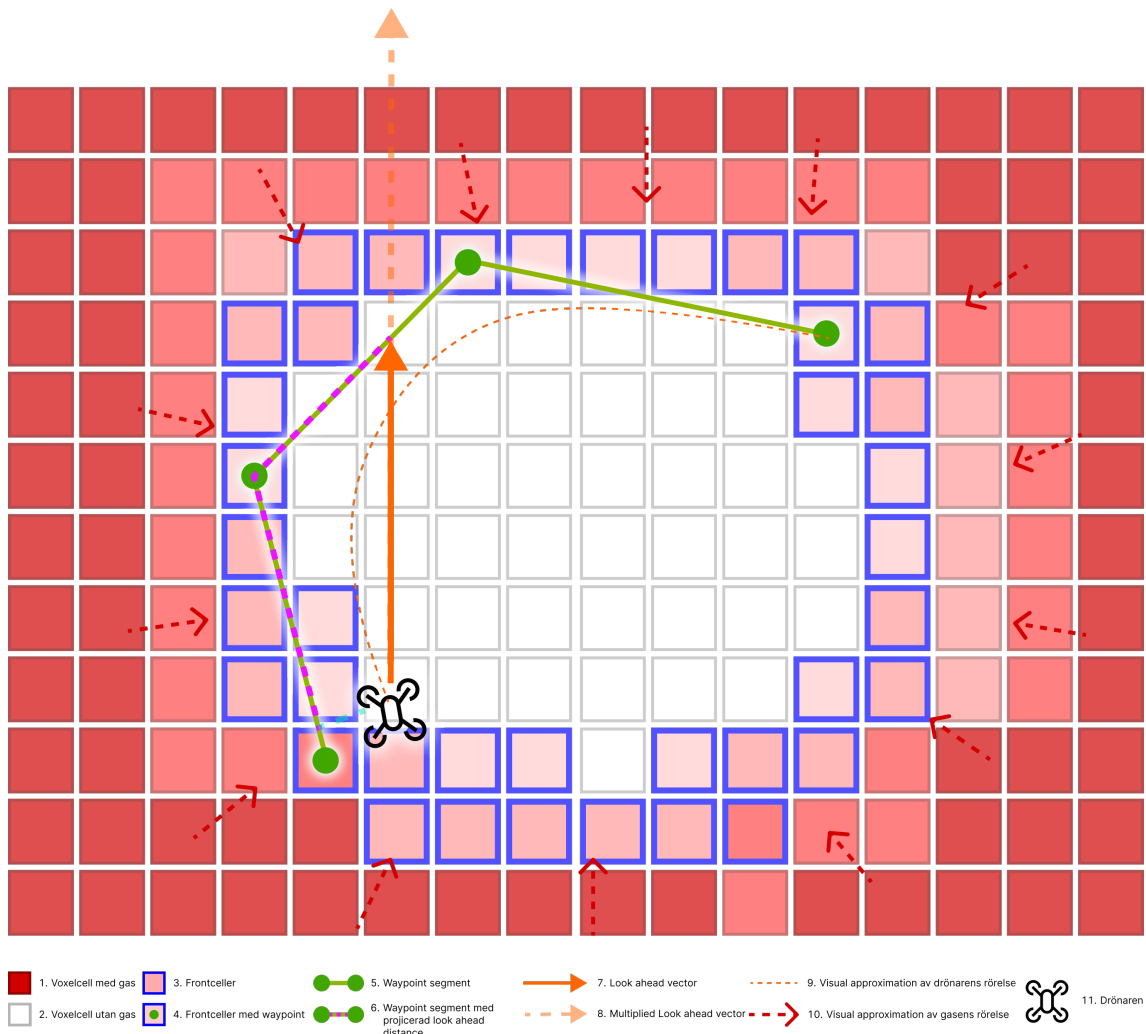
För varje ny waypoint byggs kandidatmängder upp i prioritetsordning: först frontceller i den senaste waypointens omedelbara grannskap, därefter geometriskt närliggande frontceller och slutligen samtliga återstående frontceller som fallback. Varje kandidat poängsätts utifrån en effektivitetspoäng, definierad som uppskattad frontreduktion längs sträckan till kandidaten dividerad med sträckans längd. Frontreduk-

tionen beräknas genom att summera bidrag från aktiva frontceller inom radien `frontier_reduction_radius_m`, satt till 20 m, kring den planerade sträckan. En riktningsbonus premierar dessutom kandidater som fortsätter i den senaste rörelseriktningen, medan kandidater för nära redan reserverade waypoints förkastas.

Den sammanlagda effekten av detta poängsystem är att drönaren rör sig längs frontens omkrets i en mjukt kurvande bana snarare än att hoppa fram och tillbaka mellan avlägsna frontsegment. I praktiken leder det till att drönaren kontinuerligt cirklar sin egen avsökta region samtidigt som regionen växer utåt, vilket både maximerar frontreduktionen per flugen meter och minimerar antalet frontceller som genererar diffusion in i det avsökta området.

Look-ahead-baserad navigering. Eftersom flygkontrollens (ArduPilot) inbyggda waypoint-följning bromsar in inför varje waypoint på ett sätt som påminner om en Bézier-kurva, skulle ett naivt val att skicka waypoint-kön rakt av till flygkontrollen leda till att drönaren tappar fart vid varje punkt. För att undvika detta beräknar planeraren istället en *look-ahead-position* längs den polyline som utgörs av den aktiva kön. Denna typ av framåtblickande banföljning är besläktad med pure pursuit, där ett fordon styrs mot en punkt längre fram längs den planerade banan (Coulter, 1992). Drönarens nuvarande position projiceras först ortogonalt på det segment den befinner sig på, och en initial look-ahead-punkt placeras därefter `look_ahead_distance_in_segment` (25 m) framåt längs banan. Punkten kan dock förflyttas ytterligare framåt, algoritmen testas successivt varje efterföljande hörn i polylinen och ersätter look-ahead-punkten med detta hörn under förutsättning att samtliga mellanliggande hörn ligger inom det maximala vinkelrätta avståndet `max_lookahead_deviation_m` (3,5 m) från den raka linjen mellan drönaren och kandidatpunkten, samt att totalavståndet inte överskrider `max_lookahead_distance_m` (80 m).

Effekten av denna mekanism är att look-ahead-punkten kan sträcka sig betydligt längre fram när banan är nära rätlinjig, eftersom flera på varandra följande hörn då kommer att ligga inom avvikelsekorridoren. Drönaren ser då en avlägsen målpunkt i samma riktning som den färdas och kan därmed uppnå sin maximala marschfart utan att flygkontrollens inbromsningslogik aktiveras. Genom kurvigare segment kontraherar look-ahead-punkten automatiskt till en närmare position, vilket bevarar säkerhetsmarginalen mot att drönaren genar utanför banan. För att ytterligare motverka inbromsning multipliceras den slutgiltiga look-ahead-vektorn med faktorn `carrot_distance_multiplier` (2,0), vilket effektivt placerar målpunkten dubbelt så långt bort sett från flygkontrollens perspektiv och därmed får den att begära konstant hög hastighet även när drönaren faktiskt närmar sig den verkliga look-ahead-punkten (se figur 4.3).



Figur 4.3: Schematisk illustration av RAS i en horisontell skiva av voxelvärlden. De röda voxelcellerna med gas (1) representerar områden med hög osäkerhet, medan vita voxelceller utan gas (2) representerar områden som nyligen har avsökts. Frontcellerna (3) bildar gränsen mellan dessa regioner och används som kandidater för waypoint-planering. Waypoints placerade på fronten (4) kopplas samman till waypoint-segment (5), där ett aktivt segment även används för att beräkna den projicerade look-ahead-distansen (6). Den resulterande look-ahead-vektorn (7), och dess skalade variant (8), styr drönarens (11) rörelse längs fronten. Den streckade orange kurvan (9) visar en approximativ flygbana, medan de röda streckade pilarna (10) illustrerar hur gasen diffunderar tillbaka in mot tidigare avsökta områden.

Svärmkoordination via virtuell frontreduktion. För att förhindra att flera drönare oberoende av varandra planerar mot samma frontområde delar drönarna sina aktuella waypoint-köer på en gemensam topic `/swarm/waypoint_queues` som varje agent prenumererar på. När en drönare planerar nya waypoints behandlar den de delar av frontens som ligger nära någon annan drönarens planerade segment som om de redan vore avsökta, vilket implementeras genom att tillfälligt exkludera dessa frontceller från kandidatmängden under poängsättningen. Mekanismen är rent passiv, ingen central tilldelningsalgorithm beslutar vem som ska söka var, utan ko-

ordinationen uppstår spontant ur att varje drönare lokalt undviker att planera mot områden som redan är reserverade av en granne. Resultatet är en implicit spatial uppdelning av sökarbetet utan explicit kommunikationsöverenskommelse, vilket bevarar algoritmens decentraliserade karaktär.

4.2 Experimentuppställning

För att validera systemet och jämföra de utvecklade algoritmerna, både mot varandra och över olika dimensioner, genomfördes ett experiment baserat på det experimentssystem som beskrivs i avsnitt 3.8.5. Fokus för experimentet var att generera tillräckligt många simuleringar för att kunna utvärdera algoritmernas prestanda med minskad påverkan av slumpmässiga faktorer, exempelvis målens startpositioner och rörelsemönster. Se appendix E.1 för experiment-filen som användes.

För att minska påverkan från slumpmässiga variationer användes fem replikationer för varje unik kombination av experimentdimensioner. En replikation innebär i detta sammanhang att samma uppsättning parametrar körs flera gånger med olika random seeds, vilket påverkar exempelvis målens initiala positioner och deras rörelsemönster. Genom att utföra flera replikationer reduceras risken att resultaten påverkas oproportionerligt mycket av enskilda gynnsamma eller ogynnsamma simuleringar, vilket är särskilt viktigt när simuleringar innehåller stokastiska inslag (Banks m. fl., 2010).

Utöver detta användes fem olika instanser av varje världstyp. Även om två världar tillhör samma kategori, exempelvis skog eller stad, kan deras interna struktur skilja sig åt beroende på placering av hinder, terrängvariationer och framkomlighet. Genom att använda flera instanser av varje världstyp minskar risken att resultaten domineras av egenskaper hos en specifik världskonfiguration. Detta följer samma övergripande experimentella princip som användningen av flera replikationer, där variation mellan körningar används för att minska påverkan från enskilda utfall (Banks m. fl., 2010).

Härnäst beskrivs de parametrar och dimensioner som användes för experimentet.

4.2.1 Antal drönare

Antalet drönare valdes till 3, 5 och 8 för att undersöka hur algoritmernas prestanda förändras när svärmstorleken ökar och eftersom skalbarhet, tillsammans med robusthet och flexibilitet, ofta lyfts fram som en central egenskap inom svärmrobotik (Brambilla m. fl., 2013). Valet av dessa storlekar motiveras enligt följande:

- **3:** Tre drönare valdes som minsta svärmstorlek eftersom det möjliggör interaktion mellan flera agenter samtidigt som experimentet tydligt skiljer sig från ett en- eller tvåagentsfall.
- **5:** Fem drönare valdes som en mellannivå mellan den minsta och största svärmstorleken, vilket möjliggjorde analys av hur prestandan förändrades mellan små och större svärmar.

- **8:** Simuleringar med fler än åtta drönare kräver betydligt större beräkningsresurser. Eftersom de simulerade världarna är begränsade till områden om 150×150 meter bedömdes åtta drönare vara en rimlig övre gräns för att studera skalning utan att göra experimentkampanjen oproportionerligt stor.

4.2.2 Antal mål

Antalet mål fixerades till 10 för samtliga simuleringar, vilket innebar att uppdrags-tiden avgjordes av hur lång tid det tog för en simulering att hitta samtliga 10 mål. Antalet mål varierades inte, då fokus i experimentet låg på att jämföra beteendernas förmåga att täcka av sökområdet snarare än att analysera påverkan från olika målantalar. Att inkludera flera målantalar hade dessutom kraftigt ökat antalet simuleringar utan att nödvändigtvis bidra med proportionellt mer jämförbar data.

4.2.3 Världstyper

Tre olika världstyper användes i experimentet:

- Stad
- Skog
- Hav

Dessa miljöer representerar olika typer av terräng där sök- och räddningsuppdrag kan förekomma, vilket motiveras mer utförligt i avsnitt 3.5.1. Genom att inkludera flera miljötyper möjliggörs analys av hur robusta algoritmerna är under varierande förhållanden.

4.2.4 Startformationer

Två olika startformationer användes i experimenten. I analysgraferna benämns dessa som *grid* respektive *border*.

- **Grid:** Drönarna startar centrerat i mitten av sökområdet, placerade i en samlad rutnätsliknande formation.
- **Border:** Drönarna startar utspridda längs sökområdets kanter med jämna mellanrum.

Dessa formationer representerar två tydligt skilda initiala förutsättningar för svärmen och används för att undersöka hur känsliga olika algoritmer är för startkonfigurationen. Det är även relevant att analysera om vissa algoritmer presterar bättre under specifika formationer.

4.2.5 Totalt antal simuleringar

Kombinationen av de valda experimentdimensionerna resulterade i ett stort antal simuleringar. För varje unik kombination av parametrar genomfördes fem replikationer samt fem olika instanser av respektive världstyp.

Det totala antalet simuleringar beräknades enligt följande:

$$3 \cdot 3 \cdot 2 \cdot 4 \cdot 5 \cdot 5 = 1800$$

- 3 representerar antalet drönarkonfigurationer (3, 5, 8)
- 3 representerar antalet världstyper (*stad*, *skog*, *hav*)
- 2 representerar antalet startformationer (*grid*, *border*)
- 4 representerar antalet testade beteenden (RAS, BMUTA, PSO, UBS)
- 5 representerar antalet världsinstanter per världstyp
- 5 representerar antalet replikationer per dimensionskombination

Detta resulterade i totalt 1800 simuleringar, vilket gav ett större underlag än en enskild körning per konfiguration och möjliggjorde jämförelser baserade på flera replikationer och världsinstanter.

4.3 Genomförande av simuleringar

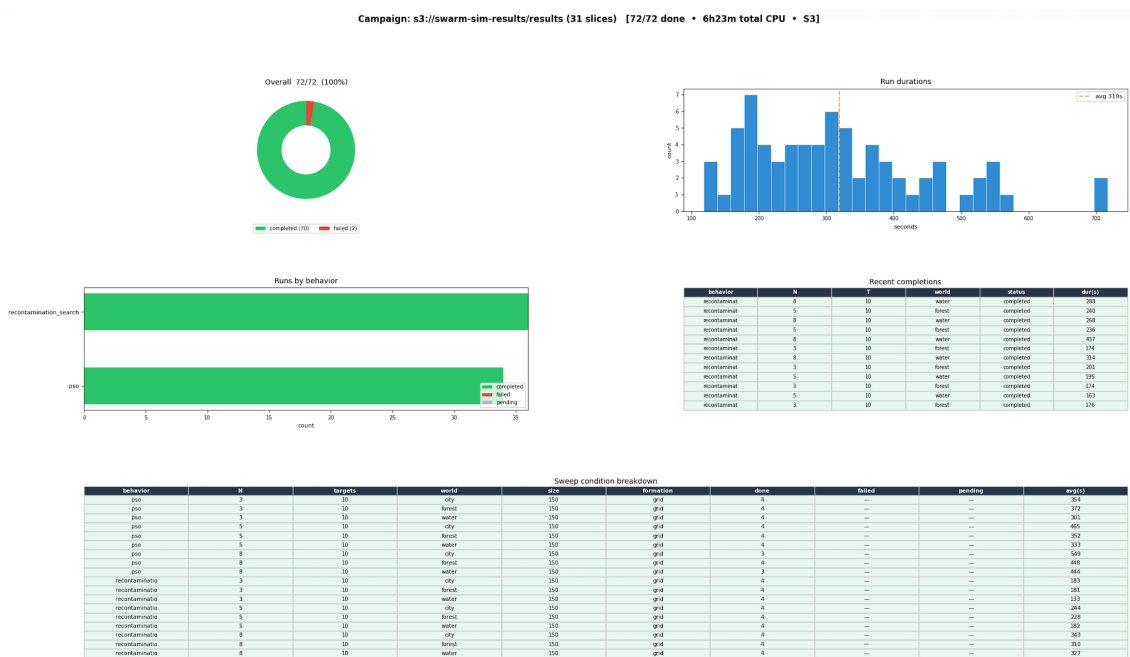
Experimentet genomfördes distribuerat över 30 instanser i AWS EC2 under 5 timmar. Även om experimentet kunde ha körts lokalt på en enskild dator hade exekveringstiden blivit mycket lång. Genom att distribuera simuleringarna över flera instanser kunde exekveringstiden reduceras från dagar till endast några timmar.

Den genomsnittliga exekveringstiden för en enskild simulering uppgick till ungefär 4 minuter. Totalt genererades ungefär 17 GB telemetridata bestående av positioner, målstatus, målpositioner och annan tillståndsinformation för samtliga drönare under simuleringarnas gång.

Experimentkonfigurationen definierades lokalt och genererade de scenarion som behövde exekveras. Dessa distribuerades senare över de olika instanserna där varje instans exekverade 60 scenarion. Efter en instans var klar med sina simuleringar laddades den genererade telemetridatan upp till en gemensam Amazon S3-bucket. För att följa statusen för simuleringarna i instanserna utvecklades ett enkelt gränssnitt som laddar ner data från S3-bucketen och visualiserar den. Se figur 4.4.

När samtliga simuleringar hade slutförts laddades telemetridatan ner till den lokala datorn där analysen genomfördes. Denna arkitektur möjliggjorde decentraliserad exekvering av simuleringar samtidigt som orkestrering och analys förblev centraliserad.

4. Experimentell utvärdering



Figur 4.4: GUI för visualisering av experimentets progression under distribuerad exekvering av simuleringar.

5

Resultat

Detta kapitel presenterar och analyserar resultaten från den experimentella utvärderingen. Analysen bygger på data som bearbetats med de utvecklade analysverktygen och visualiserats i form av grafer.

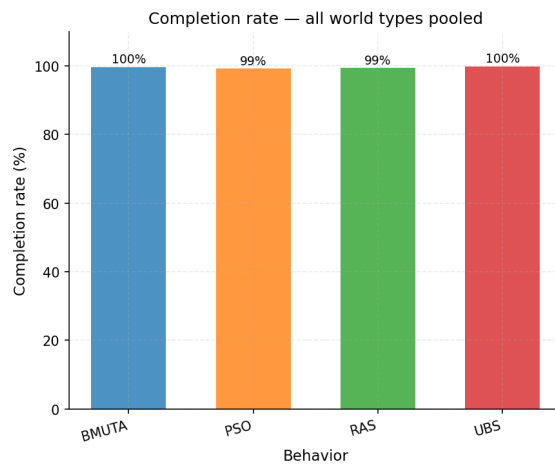
Samtliga visualiseringar som producerades av analysverktygen inkluderas inte i rapporten, eftersom flera av dem inte bedömdes bidra med ytterligare relevanta insikter för den genomförda utvärderingen. För en mer fullständig översikt av tillgängliga analysresultat kan mindre experiment exekveras i systemet och analyseras med de tillhörande analysverktygen.

5.1 Experimentets genomförande och datakvalitet

Den experimentella utvärderingen genererade totalt 1800 simuleringar fördelade över flera världstyper, startformationer och svärmstorlekar. Resultaten analyserades utifrån både övergripande systemmått och beteenderelaterade prestandamått.

Figur 5.1 visar andelen lyckade simuleringar i experimentet. Det framgår från figuren att majoriteten av simuleringarna genomfördes utan fel. Ett mindre antal simuleringar misslyckades dock under exekveringen. Detta var förväntat, då simuleringarna består av flera komplexa komponenter där fel kan uppstå utanför själva simuleringslogiken, exempelvis vid uppstart av ArduPilot-instanser.

Eftersom 1790 av 1800 simuleringar slutfördes uppgick bortfallet till cirka 0,6 %. Detta är långt under den ofta använda tumregeln att bortfall på 5 % eller mindre kan betraktas som lågt (Dong & Peng, 2013). Under antagandet att de misslyckade simuleringarna inte är systematiskt koncentrerade till en viss algoritm eller experimentparameter bedöms bortfallet därför vara begränsat i förhållande till experimentets totala omfattning.



Figur 5.1: Andel lyckade simuleringar i experimentet.

Utöver simuleringsfel saknas täckningsdata (**SearchFeedback**) för ett antal körningar där uppdraget avslutades innan octreens sökåterkoppling hann publiceras. Detta drabbar främst RAS, där 58 av 447 slutförda körningar (13%) saknar täckningsdata, jämfört med 2–6% för övriga algoritmer. En analys av bortfallets fördelning visar att det är jämnt fördelat över svärmstorlekar, världstyper och startformationer, vilket indikerar att bortfallet är slumpmässigt snarare än systematiskt. Analyser som baseras på täckningsdata bör därför tolkas med detta i åtanke, men bedöms inte påverka slutsatserna nämnvärt.

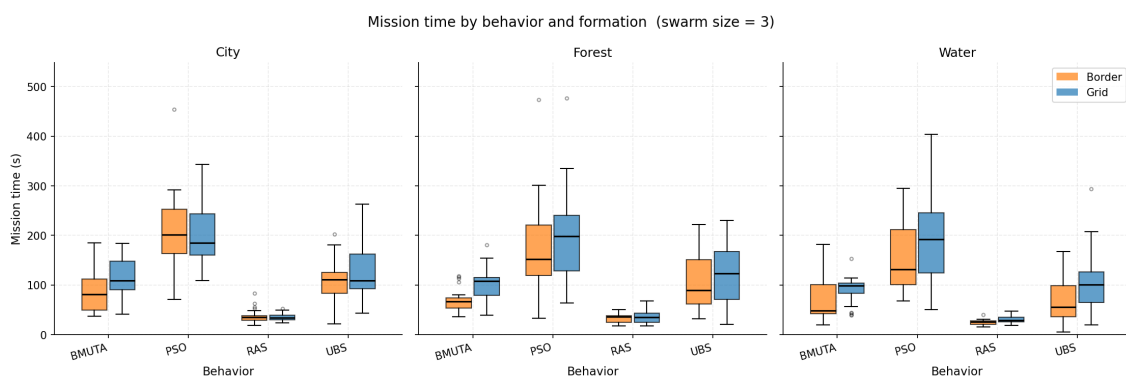
5.2 Hur påverkas uppdragstiden av experimentparametrarna?

Tid till måluppfyllelse är ett viktigt mått, exempelvis tiden tills ett visst målobjekt upptäcks (Martinez-Esteso, Castellanos, Rosello m. fl., 2025). I detta avsnitt utvärderas tid till måluppfyllelse, här kallat uppdragstid, vilket definieras som tiden tills samtliga mål har hittats. Avsnittet analyserar hur uppdragstiden påverkas av svärmstorlek, startformation och världstyp. Uppdragstid används här som ett övergripande mått på uppdragseffektivitet.

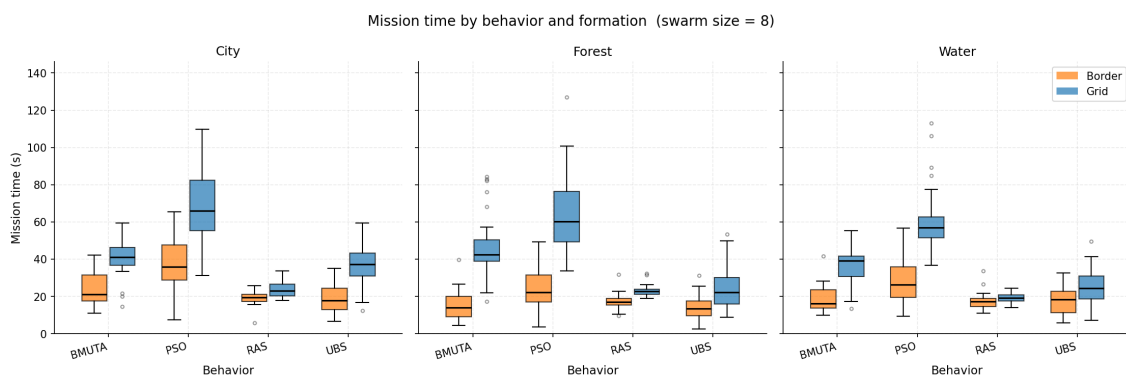
För att undersöka hur startformationen påverkar uppdragstiden jämförs resultaten för svärmar med tre respektive åtta drönare, se figur 5.2 och figur 5.3.

Figurerna indikerar att border-formationen generellt ger kortare uppdragstid än grid-formationen för både tre och åtta drönare. Ett undantag kan observeras för PSO i stadsmiljön, där skillnaden inte följer samma trend. Detta kan bero på variation mellan simuleringarna eller på att antalet simuleringar inte var tillräckligt stort för att tydligt urskilja trenden i just denna konfiguration.

5. Resultat



Figur 5.2: Jämförelse av total uppdragstid mellan olika världstyper och startformationer för svärmar med tre drönare.



Figur 5.3: Jämförelse av total uppdragstid mellan olika världstyper och startformationer för svärmar med åtta drönare.

För att kvantifiera skillnaden mellan startformationerna analyseras figur 5.4. Figuren visar den procentuella skillnaden i medianuppdragstid mellan grid och border, beräknad enligt:

$$\Delta = \frac{\tilde{t}_{\text{grid}} - \tilde{t}_{\text{border}}}{\tilde{t}_{\text{grid}}} \times 100\%$$

Ett positivt värde innebär att border ger kortare uppdragstid än grid, medan ett negativt värde innebär att border ger längre uppdragstid.

I figur 5.4 framgår det att border i genomsnitt är snabbare än grid över samtliga svärmstorlekar. Resultaten visar även en tendens till att den relativa skillnaden mellan formationerna ökar med svärmstorleken. BMUTA utgör dock ett undantag, där skillnaden är mindre för fem drönare än för tre drönare. Eftersom endast tre svärmstorlekar har undersökts bör detta tolkas försiktigt, och analysen begränsas därför till den observerade övergripande trenden snarare än en detaljerad modellering av sambandet mellan svärmstorlek och startformation.

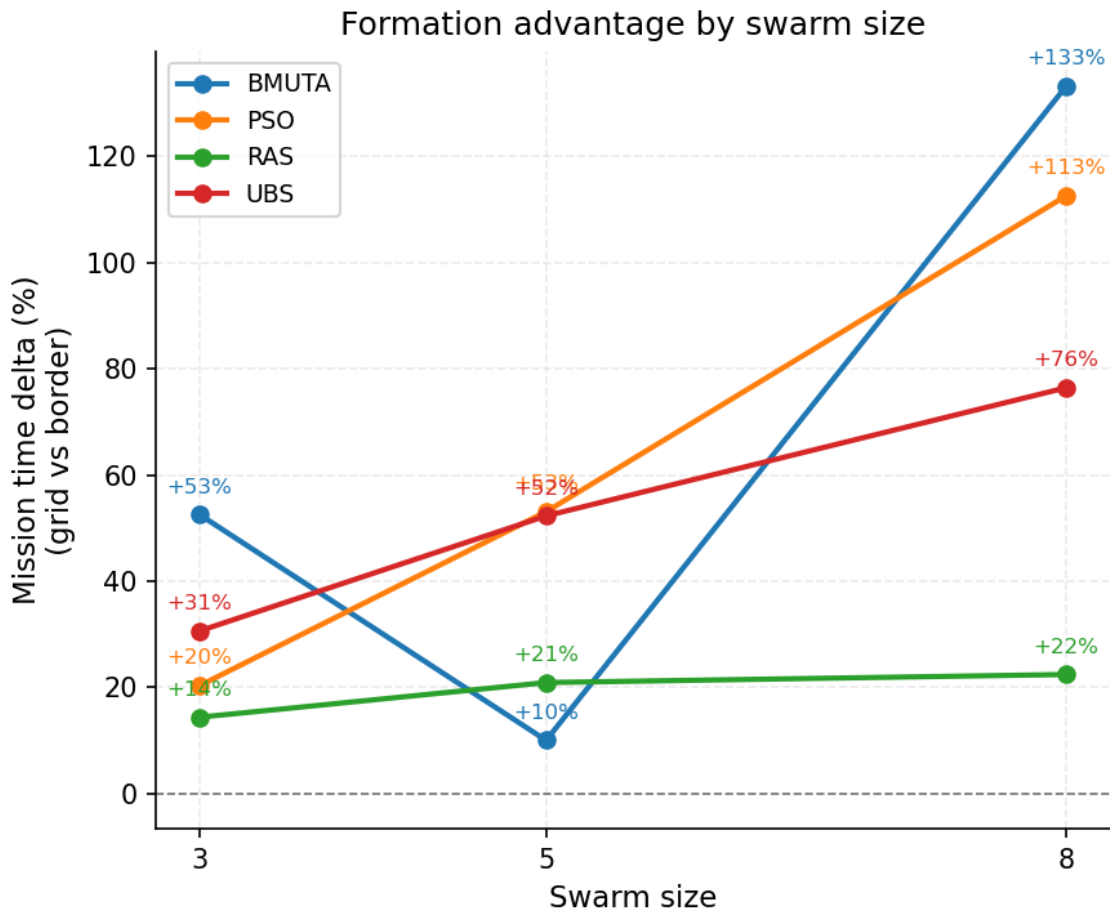
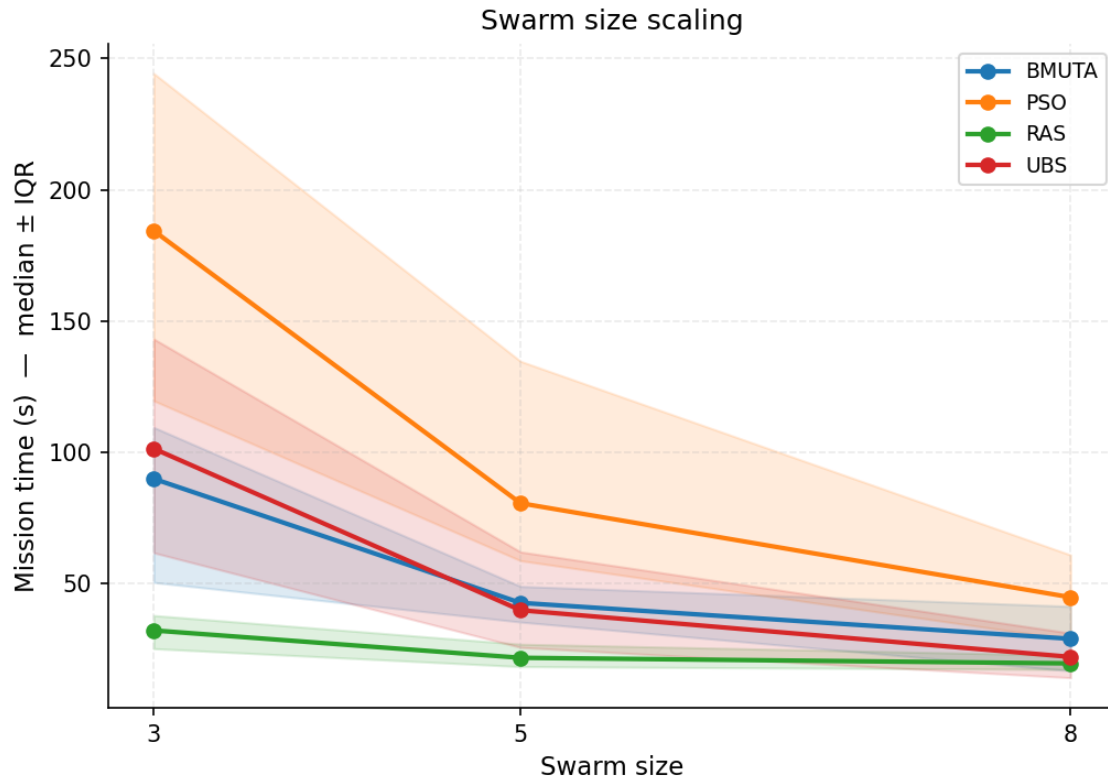


Figure 5.4: Procentuell skillnad i medianuppdragstid mellan startformationerna border och grid, grupperad per beteende och svärmstorlek.

För att analysera svärmstorlekens påverkan på uppdragstiden används figur 5.5, som visar hur algoritmernas uppdragstid förändras när antalet drönare ökar.



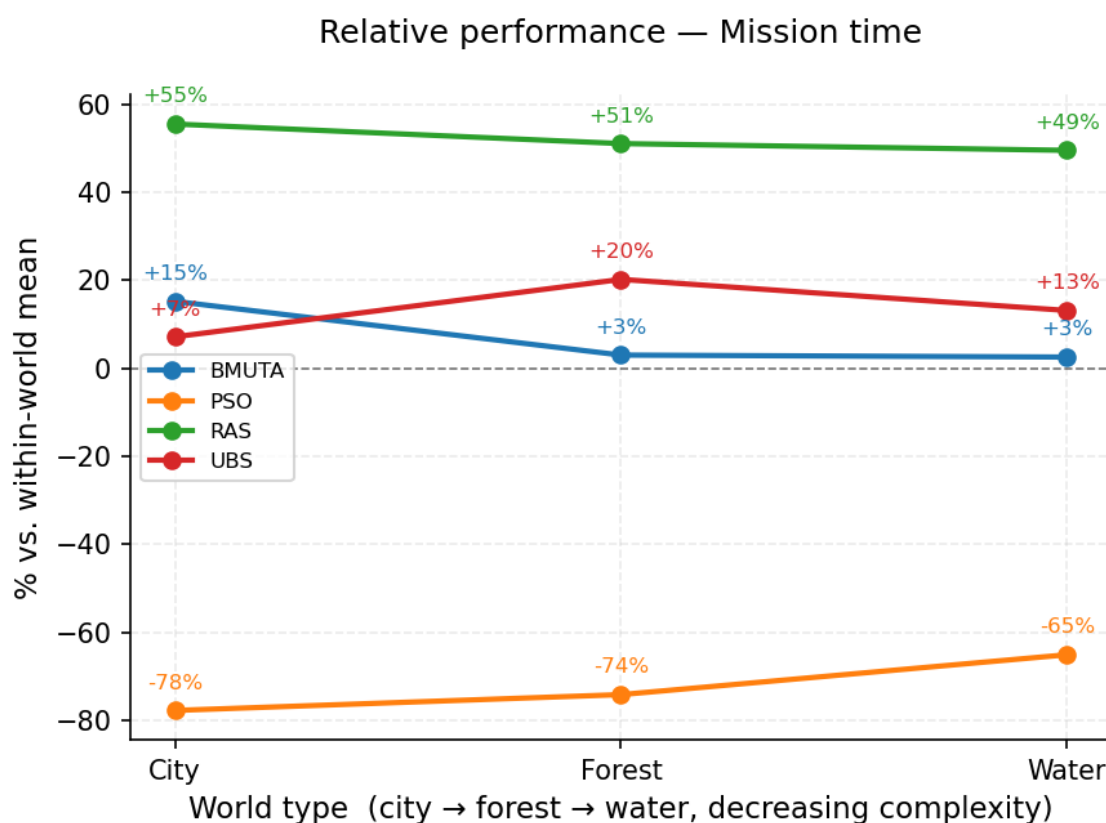
Figur 5.5: Skalning med svärmstorlek.

Samtliga algoritmer uppvisar förbättrad prestanda vid större svärmar, men förbättringens storlek varierar mellan beteendena. Resultaten indikerar även att skalningen inte är linjär, då förbättringen mellan tre och fem drönare i flera fall är större än förbättringen mellan fem och åtta drönare. För att tydligare visa dessa effekter sammanfattas resultaten även i tabell 5.1. Från tabellen framgår det att UBS och PSO har den största procentuella minskningen i uppdragstid mellan tre och åtta drönare, med minskningar på 78 % respektive 76 %. RAS påverkas minst av svärmstorlek, med en minskning på 39 %.

Algorithm	$\tilde{t}_3$ (s)	$\tilde{t}_5$ (s)	$\tilde{t}_8$ (s)	3→5	5→8	3→8
BMUTA	89,8	42,5	28,8	−53%	−32%	−68%
PSO	184,3	80,5	44,6	−56%	−45%	−76%
RAS	32,0	21,6	19,4	−33%	−10%	−39%
UBS	101,3	39,7	21,9	−61%	−45%	−78%

Tabell 5.1: Medianuppdragstid för olika svärmstorlekar samt procentuell minskning i uppdragstid mellan svärmstorlekar. $\tilde{t}_N$ anger medianuppdragstiden för svärmstorlek N , medan kolumnerna 3→5, 5→8 och 3→8 anger den procentuella minskningen i uppdragstid.

För att analysera hur uppdragstiden varierar mellan olika världstyper används figur 5.6. Figuren beskriver hur varje algoritm avviker från medelvärdet för en viss världstyp. Världstyperna ligger på x-axeln och minskar i komplexitet åt höger. Med komplexitet avses här en kvalitativ bedömning av mängden hinder och variation i höjdlägen i respektive världstyp.



Figur 5.6: Relativ uppdragstid per algoritm och världstyp, uttryckt som procentuell avvikelse från medelvärdet för respektive världstyp. Ett positivt värde innebär att algoritmen presterar bättre än genomsnittet i den världstypen. Världstyperna är ordnade från mest till minst komplex tredimensionell miljö.

Den viktigaste slutsatsen som kan dras från figur 5.6 är att världstyp inte verkar ha en tydlig påverkan på uppdragstiden i de genomförda experimenten. Ingen av algoritmernas linjer visar någon visuellt tydlig eller konsekvent lutning åt samma håll. Resultatet bör därför tolkas som att världstypen inte gav någon tydlig effekt i de genomförda experimenten, snarare än att världstyp generellt saknar betydelse.

Vissa tendenser kan dock observeras i figuren.

Den första tendensen är att RAS relativa uppdragstid minskar marginellt när världs-komplexiteten minskar. Detta är förenligt med att RAS använder mer information om världens struktur än övriga algoritmer. I de undersökta världstyperna tyder det-

ta på att RAS blir relativt mer fördelaktig i mer komplexa miljöer.

Den andra tendensen är att PSO har en motsatt relation till världskomplexitet. Ju högre världskomplexiteten är, desto mindre fördelaktig framstår PSO jämfört med medelvärdet för respektive världstyp. Detta resultat kan anses oväntat, eftersom PSO i detta arbete använder sökbarhets- och täckningsinformation som potentiellt skulle kunna vara fördelaktig i mer komplexa miljöer. Detta resultat analyseras inte i detalj här, men avsnitt 5.3, 5.4 och 5.5 visar möjliga anledningar till detta resultat.

5.3 Hur skiljer sig algoritmernas sökbeteende?

Detta avsnitt analyserar hur algoritmerna skiljer sig åt med avseende på täckningsgrad, redundant genomsökning och val av målpositioner. Dessa mått används för att beskriva hur algoritmerna söker igenom området, inte enbart hur snabbt uppdraget slutförs.

Ett centralt mått är *täckningsgrad*, det vill säga hur stor andel av uppdragsområdet som har observerats (Azmi m. fl., 2024). För tidskritiska uppdrag är det även viktigt att analysera *täckningsgrad över tid*, eftersom snabb täckning ofta är mer relevant än enbart slutresultatet (Arnold m. fl., 2018).

För att analysera hur snabbt algoritmerna täcker sökområdet används figur 5.7, som visar täckningsgraden över tid för de olika algoritmerna. Det är värt att notera att algoritmerna inte alltid når 100 % täckningsgrad. Detta beror på att en simulering avslutas när samtliga mål har hittats, snarare än när hela sökområdet har genomsökts.

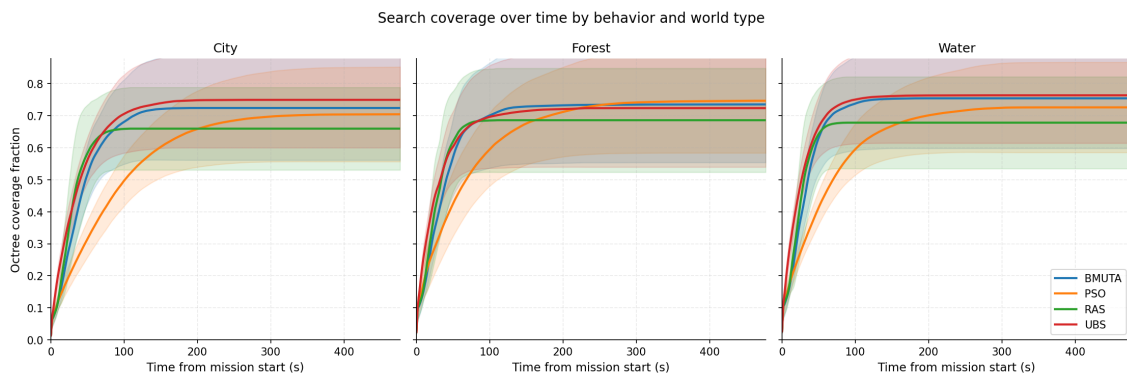


Figure 5.7: Täckningsgrad över tid för olika algoritmer.

PSO når 50 % täckningsgrad först efter ungefär 100 sekunder i stadsmiljö, medan övriga algoritmer når samma täckningsgrad vid ungefär 40 sekunder. Samma gäller för skog och havsmiljö, även om skillnaden i tid inte är lika markant så är det fortfarande en tydlig skillnad mellan PSO och övriga algoritmer.

RAS uppnår generellt en lägre slutlig täckningsgrad än övriga algoritmer. En möjlig förklaring är att RAS använder målens rörelsemodell i sin beslutslogik och därmed inte behöver avsöka lika stor andel av området innan samtliga mål hittas.

Figuren indikerar även att täckningsgraden ökar långsammare i stadsmiljö än i skogs- och havsmiljö. En möjlig förklaring är att stadsmiljön innehåller fler hinder, vilket även stöds av att stadsmiljön uppvisar högre redundans än skogs- och havsmiljö i figur 5.8. Ett sådant samband är förväntat, eftersom högre överlappning innebär att drönarna spenderar mer tid på att söka av områden som redan har observerats.

Redundansfaktorn definieras här som

$$r = \frac{N_{\text{redundant}}}{N_{\text{total}}},$$

där $N_{\text{redundant}}$ är antalet observationer av redan avsökta områden och N_{total} är det totala antalet observationer.

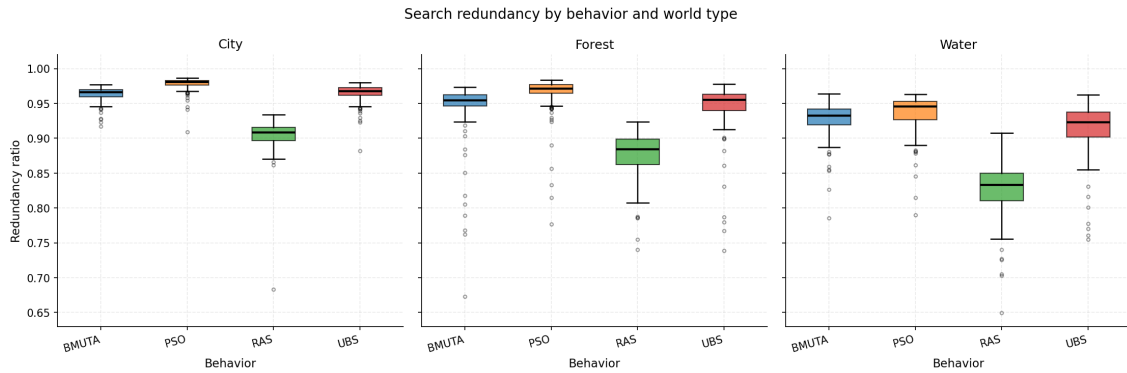
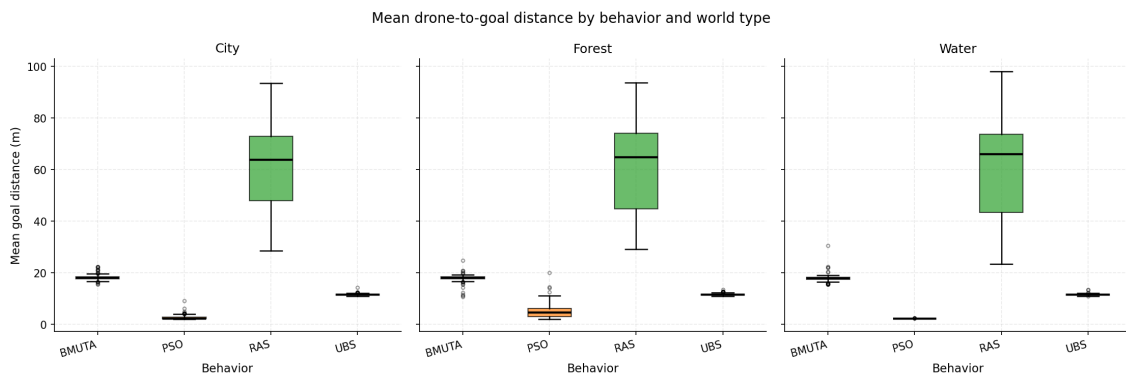


Figure 5.8: Andel redundant genomsökning för olika algoritmer.

Figure 5.8 visar andelen redundant genomsökning för de olika algoritmerna. Det framgår från figuren att samtliga algoritmer uppvisar en redundansfaktor mellan 0,8 och 1,0, vilket tyder på att samma områden genomsöks flera gånger. En möjlig förklaring är att algoritmer med lägre genomsnittlig hastighet befinner sig längre tid inom samma område, vilket kan öka sannolikheten för att samma yta avsöks flera gånger.

I figur 5.10 kan denna relation observeras, då PSO har lägst genomsnittlig hastighet och högst redundansfaktor, medan RAS har högst genomsnittlig hastighet och lägst redundansfaktor. Det finns även en marginell indikation på att stadsmiljö har högst redundansfaktor och havsmiljö lägst. Detta är förenligt med att stadsmiljön är mer geometriskt komplex.

För att undersöka hur långt algoritmerna tenderar att planera från drönarnas aktuella position analyseras figur 5.9. Figuren visar den genomsnittliga distansen mellan drönarnas aktuella position och deras valda målpositioner under simuleringarna.



Figur 5.9: Genomsnittlig distans mellan drönare och dess målposition.

Det framgår att drönare som använder PSO har det lägsta genomsnittliga avståndet till målposition i samtliga världstyper. Figuren visar resultaten uppdelade per världstyp, medan analysverktygens sammanslagna resultat över samtliga världstyper visar att PSO har den lägsta medianen, 2,2 m. RAS har den högsta medianen, 64,2 m.

Interquartile Range (IQR), det interkvartila avståndet, används här som ett mått på spridningen i resultaten. RAS uppvisar även den högsta IQR:n, 28,6 m. Detta är förenligt med RAS konstruktion, där distansen till nästa målposition inte är förbestämd, till skillnad från exempelvis BMUTA där måldistansen i stor utsträckning bestäms av regionernas storlek.

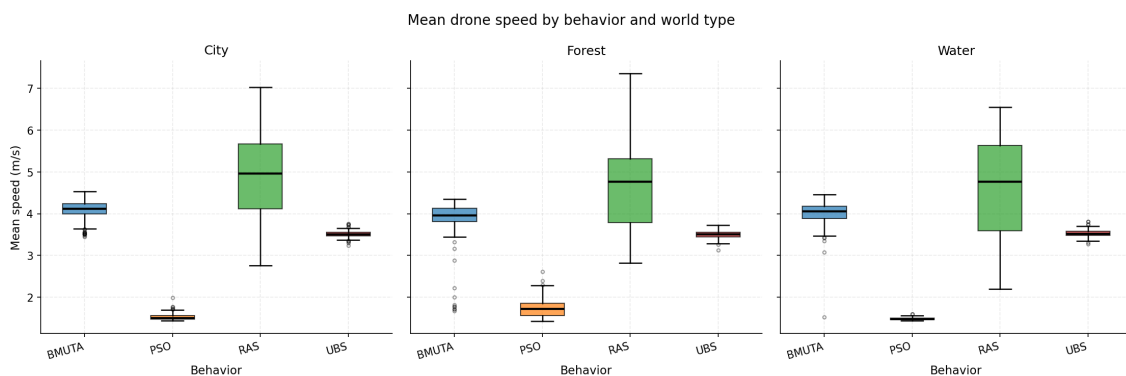
Det bör noteras att täckningsdata saknas för några körningar, mest för RAS-körningar, vilket diskuteras i avsnitt 5.1.

5.4 Hur skiljer sig algoritmernas rörelse- och koordinationsbeteende?

Detta avsnitt analyserar algoritmernas rörelsebeteende, framför allt genom hastighetsprofil och variation i rörelsemönster. Syftet är att koppla algoritmernas rörelsebeteende till tidigare observerade skillnader i uppdragstid, täckningsgrad och redundans.

För att analysera algoritmernas rörelsebeteende används figur 5.10, som visar den genomsnittliga hastigheten för de olika algoritmerna.

Från figur 5.10 framgår det att PSO har den lägsta genomsnittliga hastigheten över samtliga världstyper. Analysverktygens sammanslagna resultat över samtliga världstyper visar att PSO har en medianhastighet på 1,5 m/s, medan RAS ligger högre och uppvisar större spridning. Detta överensstämmer med figur 5.9, som visar att drönare som använder PSO tenderar att välja målpositioner nära sin nuvarande position. Detta kan bidra till lägre hastigheter, eftersom drönaren oftare behöver bromsa in inför närliggande målpositioner.

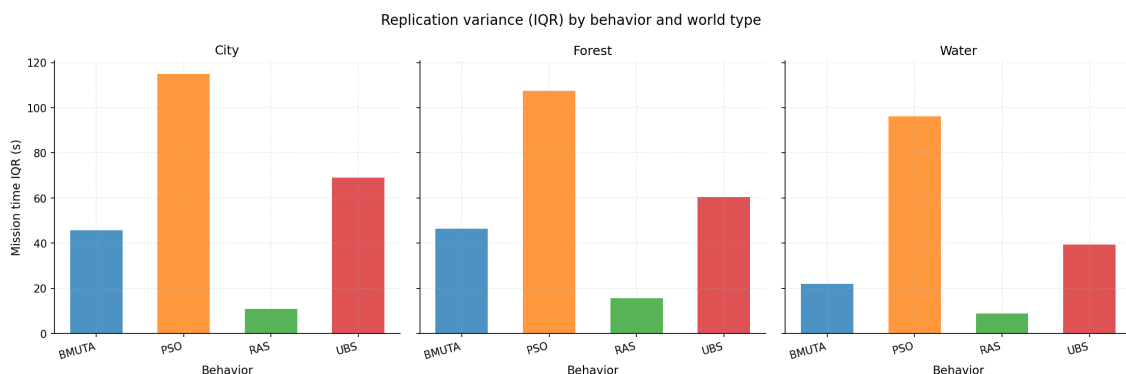


Figur 5.10: Genomsnittlig hastighet för olika algoritmer.

RAS har även störst spridning, med en sammanslagen IQR på 1,7 m/s över samtliga världstyper. Detta tyder på ett mer dynamiskt rörelsemönster där drönarnas hastighet varierar mer beroende på den aktuella planerade banan och nästa målposition.

5.5 Hur robusta är algoritmerna?

Detta avsnitt analyserar algoritmernas robusthet genom att undersöka variationen mellan olika replikeringar av samma experimentkonfiguration. Låg variation tyder på att algoritmen ger mer konsekventa resultat under varierande förutsättningar. Figur 5.11 visar variationen i uppdragstid mellan olika replikeringar av samma experimentkonfiguration.



Figur 5.11: Replikeringsvarians för olika världstyper.

PSO uppvisar störst variation mellan replikeringar, vilket indikerar att algoritmens prestanda är mer känslig för stokastiska faktorer såsom målens rörelse och initiala positioner. Den sammanslagna IQR:n för PSO över samtliga världstyper är 79,8 s.

RAS uppvisar däremot minst variation mellan replikeringar (IQR 22,4 s över alla världstyper), vilket tyder på ett mer stabilt och konsekvent beteende under varierande förutsättningar.

5.6 Sammanfattande jämförelse mellan algoritmer

Sammanfattningsvis visar resultaten att systemet kan användas för att identifiera skillnader mellan algoritmer med avseende på uppdragstid, täckningsgrad, redundans och rörelsemönster.

Analysen indikerar att svärmstorlek och startformation har tydlig påverkan på uppdragseffektiviteten, medan världstypen generellt har mindre påverkan på uppdragstiden i de genomförda experimenten.

Vidare uppvisar algoritmerna skillnader i både replikeringsvarians, koordinationsbeteende och rörelsemönster, vilket visar att analysverktygen kan användas för att identifiera beteendespecifika egenskaper hos decentraliserade sökstrategier.

6

Slutsats

Detta arbete har presenterat utvecklingen av ett modulärt system för implementation, simulering och utvärdering av decentraliserade beteenden för svärmar av drönare i tredimensionella miljöer. Genom den experimentella utvärderingen har systemets arkitektur, flexibilitet och analysmöjligheter undersökts med hjälp av flera separata beteenden och ett stort antal simuleringar.

I detta kapitel diskuteras arbetets resultat i relation till de mål som definierades i syftet, med fokus på systemets förmåga att stödja utveckling, exekvering och jämförbar analys av multi-UAV-beteenden.

6.1 Modulär systemarkitektur

En central målsättning med arbetet var att utveckla en modulär systemarkitektur där beslutslogik, simulering, datainsamling och analys separeras från varandra. Resultatet visar att denna uppdelning fungerade väl i praktiken då algoritmerna utvecklades oberoende av varandra och kunde exekveras inom samma experimentella ramverk. Resultatet visar även att simulering, datainsamling och analys kunde separeras från varandra. Simuleringarna kördes i separata instanser, telemetridatan sparades i en separat databas och analysen kunde därefter genomföras på en lokal dator.

Vidare möjliggjorde systemets scenario- och experimentorkestrering en tydlig separation mellan enskilda simuleringar och större experimentuppställningar. De algoritmer som utvecklades och testades i enskilda scenarier kunde användas i det slutliga experimentet utan att algoritmerna behövde ändras. Detta visar att systemets gränssnitt mellan beteenden, scenarier och experiment fungerade som avsett.

6.2 Decentraliserade algoritmer

Algoritmerna utvecklades separat av olika utvecklare, men använde samma centraliserade systemkomponenter. Algoritmerna skilde sig dessutom tydligt i både komplexitet och resurskrav. Detta indikerar att systemet hade en fungerande uppdelning mellan centraliserad infrastruktur och algoritmspecifik beslutslogik.

De mindre komplexa algoritmerna, exempelvis UBS, kunde implementeras med relativt lite kod jämfört med RAS. Samtidigt kunde mer avancerade algoritmer utnyttja

samma gemensamma funktionalitet för exempelvis datainsamling, simulering och grundläggande rörelsehantering. Detta visar att systemet kan stödja både enklare referensbeteenden och mer komplexa decentraliserade algoritmer.

6.3 Simulerade uppdrag och experiment

Resultatet visar att systemet möjliggjorde uppsättning av reproducerbara experiment i varierande tredimensionella miljöer. Genom att kombinera olika världstyper, startformationer och svärmstorlekar kunde algoritmerna utvärderas under flera typer av förutsättningar relevanta för sök- och räddningsinspirerade uppdrag.

Vidare visade resultatet att systemet, med hjälp av integrationen med AWS, kunde hantera ett större experiment med 1800 simuleringar. Den distribuerade exekveringen reducerade den praktiska körtiden till ungefär fem timmar. Detta indikerar att systemet kan skalas till större experiment, även om ytterligare skalning skulle kräva mer resurser och tid. Analysen genomfördes på en lokal dator, vilket visar att den insamlade datan kunde bearbetas utan distribuerade analysresurser. Det är dock oklart om analysdelen skalar lika väl som simuleringarna som kördes på AWS.

6.4 Utvärdering och analys

En viktig del av arbetets målsättning var att systemet inte enbart skulle möjliggöra utveckling och simulering av decentraliserade beteenden, utan även stödja systematisk utvärdering och analys av resultaten. Resultatet visar att den centraliserade telemetriinsamlingen fungerade väl i praktiken och möjliggjorde jämförbar analys mellan olika algoritmer och experimentdimensioner.

Genom att telemetriinsamling var centraliserad och all telemetridata sparades i ett gemensamt format oberoende av beteende kunde analysverktygen användas för samtliga algoritmer utan behov av beteendespecifika anpassningar. Detta möjliggjorde automatiserad generering av grafer och sammanställningar baserade på ett stort antal simuleringar, vilket hade varit svårt att genomföra manuellt.

Vidare visade resultaten att analysverktygen kunde användas för att identifiera skillnader mellan algoritmer under varierande förutsättningar, exempelvis mellan olika världstyper, startformationer och svärmstorlekar. Detta stödjer att systemet uppfyllde målet att möjliggöra reproducerbar och jämförbar utvärdering av decentraliserade multi-UAV-beteenden.

Resultaten visar att de utvecklade algoritmerna påverkas olika av varierande experimentförutsättningar. En viktig observation är dock att världstypen inte hade någon tydlig påverkan på uppdragstiden i de genomförda experimenten. Detta innebär inte att variation mellan världar är oväsentlig, utan snarare att de algoritmer som utvärderades i detta arbete inte uppvisade någon stark världstypsberoende skillnad.

En möjlig förklaring är att antalet replikationer och världsinstanser inte var tillräckligt stort för att tydligt urskilja sådana effekter. En annan möjlig förklaring är att de testade algoritmerna inte var tillräckligt anpassade för att utnyttja skillnader mellan olika tredimensionella miljöer. Detta bedöms dock inte som en begränsning i systemets arkitektur, eftersom antalet replikationer och världsinstanser kan ökas i framtida experiment. Begränsningen ligger snarare i den tids- och resursram som gällde för den genomförda experimentkampanjen.

7

Framtida arbete

Fortsatt arbete kan bygga vidare på systemet genom att lägga till fler plugins och konfigurationsmöjligheter. Exempelvis skulle användaren kunna välja drönartyp, sensorer och kameror för att bättre efterlikna den utrustning som används i verkliga tillämpningar. Detta skulle minska beroendet av standardmodellen Iris i Gazebo och ArduPilot (ArduPilot, u.å.) och göra simuleringarna mer anpassningsbara till olika användningsområden.

Systemet kan även utvecklas genom att inkludera mer realistiska miljöförhållanden till exempel vind, regn och reducerad sikt. Dessa faktorer kan påverka både drönarnas rörelse, energiförbrukning och sensorernas förmåga att identifiera mål. På liknande sätt skulle framtida arbete kunna införa mer realistiska kommunikationsmodeller, där avstånd, hinder, fördröjningar, paketförluster och kommunikationsbortfall påverkar informationsutbytet mellan drönarna. Detta skulle göra det möjligt att analysera hur robusta olika decentraliserade algoritmer är vid ofullständig eller fördröjd information.

För att simulera scenarion då drönare går sönder finns en funktion för att slumpvis stänga av drönare baserat på en Weibull-fördelning. Denna funktionen användes inte vid simuleringar på grund av tidsbrist och fortsatt arbete krävs för att möjliggöra för användaren att själv enkelt kunna styra hur det ska användas.

Slutligen bör framtida arbete undersöka skalbarhet och fysisk validering. Genom att testa systemet med ännu fler drönare kan man analysera hur beräkningstid och koordinering påverkas när systemet växer. Ett senare steg är att implementera utvalda algoritmer på faktiskt verkliga drönare eller i en hybridmiljö där fysisk hårdvara kombineras med simulering. Detta skulle ge möjlighet att jämföra simulerade resultat med verkliga flygförhållanden och därmed bättre bedöma systemets praktiska användbarhet.

8

Etiska aspekter

Även om projektet genomförs i en simulerad miljö finns flera etiska aspekter som är relevanta att beakta, särskilt eftersom arbetet fokuserar på autonoma och distribuerade multi-UAV system i sök- och räddningsinspirerade scenarier. Ett sådant fokus har ett tydligt samhällsnyttigt syfte, men tekniken aktualiserar samtidigt frågor om hur autonoma system bör utvecklas och användas på ett ansvarsfullt sätt.

En central aspekt är teknikens *dubbelanvändning* (*dual-use*). Metoder för samordning, informationsdelning och autonomt beslutsfattande som utvecklas för civila sök- och räddningsinsatser kan även vara användbara i andra sammanhang, inklusive militära. Ett system som lokaliserar nödställda personer i svåråtkomliga miljöer skulle exempelvis kunna anpassas för övervakning eller objektsökning i andra typer av operationer. Även om projektet avgränsas till civila sök- och räddningsrelaterade scenarier är det därför viktigt att reflektera över hur resultaten kan användas utanför den avsedda kontexten.

En annan etisk aspekt rör *ansvarsfördelning* i system där flera autonoma enheter fattar beslut distribuerat. När koordinering sker utan central styrning kan det bli otydligt var ansvaret ligger om systemet agerar felaktigt eller ineffektivt. I detta arbete testas algoritmerna enbart i simulering, vilket innebär att inga verkliga personer eller miljöer påverkas direkt. Frågan om juridiskt och operativt ansvar analyseras därför inte på djupet inom ramen för projektet, men bedöms vara viktig vid framtida praktisk tillämpning.

Vidare finns en etisk avvägning mellan *mänsklig kontroll* och *autonomi*. För att ett distribuerat multi-UAV-system ska fungera effektivt i sök- och räddnings-scenarion behöver varje UAV kunna fatta lokala beslut baserat på begränsad information, särskilt vid kommunikationsstörningar. Samtidigt kan ökad autonomi minska möjligheten till direkt mänsklig intervention. I detta projekt studeras frågan främst ur ett tekniskt perspektiv i simulerade miljöer, men den är etiskt relevant eftersom den påverkar hur sådana system bör utformas för framtida användning i säkerhetskritiska sammanhang.

Sammanfattningsvis innebär projektets fokus på simulering att de direkta etiska riskerna är begränsade under genomförandet, men flera principiella frågor, särskilt kring dubbelanvändning, ansvar och mänsklig kontroll, är fortsatt viktiga att synliggöra och beakta i tolkningen av arbetets resultat.

Referenser

- Amazon Web Services. (2026a). *What is Amazon EC2?* AWS Documentation. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html>
- Amazon Web Services. (2026b). *What is Amazon S3?* AWS Documentation. <https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html>
- Arnold, R. D., Yamaguchi, H., & Tanaka, T. (2018). Search and rescue with autonomous flying robots through behavior-based cooperative intelligence. *Journal of International Humanitarian Action*, 3(18). <https://doi.org/10.1186/s41018-018-0045-4>
- Azmi, I. N., Kassim, M., Yussoff, Y. M., & Tahir, N. M. (2024). Optimizing drone-assisted victim localization and identification in mass-disaster management: a study on feasible flying patterns and technical specifications. *International Journal of Electrical and Computer Engineering*, 14(4), 4097–4109. <https://doi.org/10.11591/ijece.v14i4.pp4097-4109>
- Babia, R. (2026). *Low Poly House*. BlenderKit. https://www.blenderkit.com/?query=author_id:1920311
- Banks, J., Carson, J. S., Nelson, B. L., & Nicol, D. M. (2010). *Discrete-Event System Simulation* (5. utg.). Pearson.
- Brambilla, M., Ferrante, E., Birattari, M., & Dorigo, M. (2013). Swarm Robotics: A Review from the Swarm Engineering Perspective. *Swarm Intelligence*, 7, 1–41. <https://doi.org/10.1007/s11721-012-0075-2>
- Bridson, R. (2007). Fast Poisson disk sampling in arbitrary dimensions [Conference Paper]. *ACM SIGGRAPH 2007 Sketches, SIGGRAPH'07*, 22. <https://doi.org/10.1145/1278780.1278807>
- Chen, Z., Yan, J., Ma, B., Shi, K., Yu, Q., & Yuan, W. (2023). A Survey on Open-Source Simulation Platforms for Multi-Copter UAV Swarms. *Robotics*, 12(2), 53. <https://doi.org/10.3390/robotics12020053>
- Cho, W. J., Kim, S., Kim, Y., & Moon, Y. H. (2022). Advanced Co-Simulation Platform for UAV Simulations Under Virtual Wireless Network Environments. *IEEE Access*, 10, 95498–95508. <https://doi.org/10.1109/ACCESS.2022.3201526>
- Coombes, M., McAree, O., Chen, W.-H., & Render, P. (2012). Development of an autopilot system for rapid prototyping of high level control algorithms [Conference Paper]. *Proceedings of the 2012 UKACC International Conference on Control, CONTROL 2012*, 292–297. <https://doi.org/10.1109/CONTROL.2012.6334645>
- Coulter, R. C. (1992). *Implementation of the pure pursuit path tracking algorithm* (CMU-RI-TR-92-01). Carnegie Mellon University. <https://publications.ri>

- cmu.edu/storage/publications/pub_files/pub3/coulter_r_craig_1992_1/coulter_r_craig_1992_1.pdf
- Cui, S., Li, H., Fan, X., Ni, L., & Hou, J. (2025). A Multi-UAV Distributed Collaborative Search Algorithm Based on Maximum Entropy Mechanism. *Drones*, 9(8). <https://doi.org/10.3390/drones9080592>
- David, A. (2025). *Stylised Low Poly Building*. BlenderKit. https://www.blenderkit.com/?query=author_id:183866
- DesMarais, A. M. (2017). Thermal imaging cameras in the detection of human cadaveric remains. I *Thermal Imaging: Types, Advancements and Applications* (s. 1–24). Nova Science Publishers, Inc. <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85035101626&partnerID=40&md5=926c49168362b3c453c6dfed6b986720>
- Dong, Y., & Peng, C.-Y. J. (2013). Principled missing data methods for researchers. *SpringerPlus*, 2(1), 222. <https://doi.org/10.1186/2193-1801-2-222>
- Garland, M., & Heckbert, P. S. (1997). Surface simplification using quadric error metrics. *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, 209–216. <https://doi.org/https://doi.org/10.1145/258734.258849>
- Kurdi, H., Al-Megren, S., Aloboud, E., Alnuaim, A. A., Alomair, H., Alothman, R., Ben Muhayya, A., Alharbi, N., Alenzi, M., & Youcef-Toumi, K. (2020). Bee-inspired task allocation algorithm for multi-UAV search and rescue missions. *International Journal of Bio-Inspired Computation*, 16(4), 252–263. <https://doi.org/10.1504/IJBIC.2020.112339>
- Longley, A. (1978). Radio propagation in urban areas [Conference Paper]. *28th IEEE Vehicular Technology Conference*, 28, 503–511. <https://doi.org/10.1109/VTC.1978.1622591>
- Martinez-Esteso, J. P., Castellanos, F. J., & Gallego, A. J. (2025). Enhancing Maritime Search and Rescue: Incremental Unsupervised Domain Adaptation with Synthetic Data and Pseudo-labeling. *Expert Systems with Applications*, 305, 130864. <https://doi.org/10.1016/j.eswa.2025.130864>
- Martinez-Esteso, J. P., Castellanos, F. J., Rosello, A., Calvo-Zaragoza, J., & Gallego, A. J. (2025). On the use of synthetic data for body detection in maritime search and rescue operations. *Engineering Applications of Artificial Intelligence*, 139. <https://doi.org/10.1016/j.engappai.2024.109586>
- Missing People Sweden. (2024). Effektrapport för 2024. <https://www.missingpeople.se/media/tjhnoy5k/effektrapport-2024.pdf>
- Open Robotics. (2022a). *ROS 2 Humble Hawksbill Documentation*. Open Robotics. <https://docs.ros.org/en/humble/>
- Open Robotics. (2022b). *RViz User Guide*. ROS 2 Documentation. <https://docs.ros.org/en/humble/Tutorials/Intermediate/RViz/RViz-User-Guide/RViz-User-Guide.html>
- Ordoukhanian, E., & Madni, A. M. (2017). Introducing resilience into multi-UAV system-of-systems network. I *Disciplinary Convergence in Systems Engineering Research* (s. 27–40). Springer International Publishing. https://doi.org/10.1007/978-3-319-62217-0_3

- Perlin, K. (1985). An Image Synthesizer. *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques [Conference Paper]*, 287–296. <https://doi.org/10.1145/325334.325247>
- Polismyndigheten. (2023). *Polismyndighetens efterforskning av försvunna personer* (2023:4). Polismyndigheten. https://polisen.se/siteassets/dokument/tillsynsrapporter/2023_4-rapport---polismyndighetens-efterforskning-av-forsvunna-personer.pdf
- Schedl, D. C., Kurmi, I., & Bimber, O. (2021). An autonomous drone for search and rescue in forests using airborne optical sectioning. *Science Robotics*, 6(55). <https://doi.org/10.1126/scirobotics.abg1188>
- Soni, L., Kaur, A., & Sharma, A. (2023). A Review on Different Versions and Interfaces of Blender Software [Conference Paper]. *7th International Conference on Trends in Electronics and Informatics, ICOEI 2023 - Proceedings*, 882–887. <https://doi.org/10.1109/ICOEI56765.2023.10125672>
- Strohhecker, S., Disney, S. M., & Naim, M. M. (2008). Understanding supply chain dynamics via simulation. I *Construction Supply Chain Management Handbook* (9–1 – 9–30). CRC Press. <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85055810857&partnerID=40&md5=e616de7bfec1f437a5f9ffde3c614098>
- Xee, L. (2022). *A Low Poly Kalamansi Tree*. BlenderKit. https://www.blenderkit.com/?query=author_id:179949
- Xiong, A. H., Banglu Ge, B., & Liu, C. J. (2023). An Improved Slime Mold Algorithm for Cooperative Path Planning of Multi-UAVs* [Conference Paper]. *2023 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 1–6. <https://doi.org/10.1109/ROBIO58561.2023.10354937>
- Yamauchi, B. (1997). A frontier-based approach for autonomous exploration [Conference Paper]. *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97. Towards New Computational Principles for Robotics and Automation*, 146–151. <https://doi.org/10.1109/CIRA.1997.613851>
- Zhao, X., Zhao, Z., Wan, Y., & Tang, J. (2025). Autonomous collaborative rescue of drone swarms in cluttered environments. *Cluster Computing*, 28(11). <https://doi.org/10.1007/s10586-025-05475-x>

A

Blender

A.1 Skogsgeneratorns parametrar

Skogsgeneratoren definierades med en uppsättning ingångsparametrar i en *input node*. Dessa parametrar används för att styra världens utbredning, terrängens detaljgrad, väggenereringen och den seedstyrda variationen mellan olika instanser av samma världstyp.

1. **World seed**: Ett heltal som används för att programmatiskt styra slumpmässigheten i världsgenereringen.
2. **Small world noise**: Ett flyttal som styr det småskaliga bruset i terrängen.
3. **Medium world noise**: Ett flyttal som styr det medelskaliga bruset i terrängen.
4. **Large world noise**: Ett flyttal som styr det storskaliga bruset i terrängen.
5. **Road noise amplitude**: Ett flyttal som reglerar medelvärdet av amplituden, det vill säga intensiteten, hos brusfunktionen för vägarna.
6. **Road depth**: Ett värde som anger medeldjupet för hur långt ned varje väg fördjupas i marken.
7. **World size**: Ett värde som anger sidlängden på världen, det vill säga dess utbredning i x - och y -led, där z -axeln pekar uppåt.
8. **World resolution**: Ett värde som anger upplösningen av världens markplan i x - och y -led.
9. **Road width**: Medelvärdet av vägbredden.
10. **Min number of roads**: Det minsta antalet stigar som genereras i en skogsvärld.
11. **Max number of roads**: Det största antalet stigar som genereras i en skogsvärld.

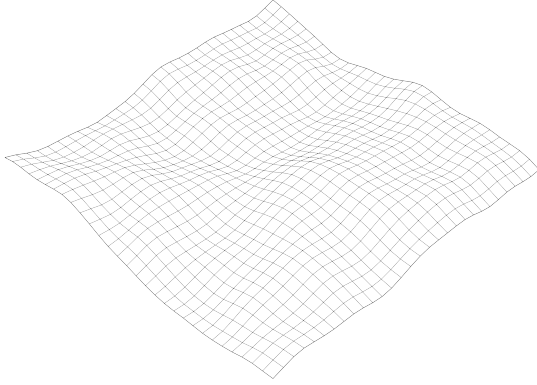
A.2 Skogsgeneratorns nodbaserade implementation

A.2.1 Terränggenerering

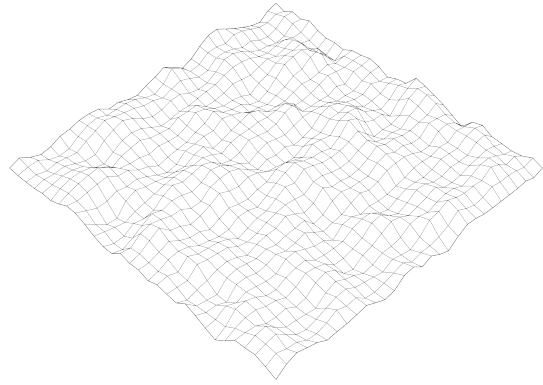
Höjdvariationen genereras genom att en plan rutnätsmesh deformeras med summan av tre Perlin-brus Perlin (1985) på olika skalor. Den minsta skalan motsvarar markråhet, den mellanstora kullar och den största storskaliga höjdformationer. Respektive amplitud styrs av parametrarna *Small/Medium/Large world noise*, medan

en seedstyrd slumpkomponent för varje skala säkerställer att varje värld blir unik men reproducerbar.

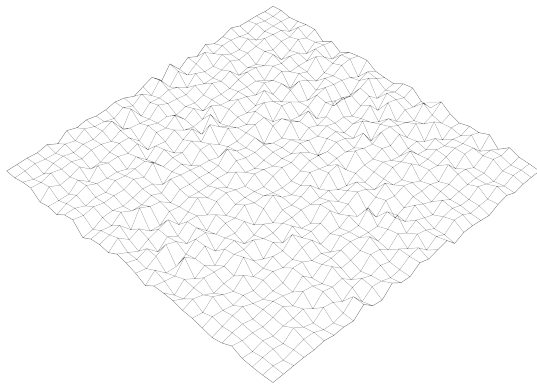
Innan förskjutningen appliceras subtraheras en konstant från brussumman för att motverka den uppåtgående medeldrift som uppstår när flera positiva brusbidrag adderas. Den justerade höjden förskjuter därefter rutnätets hörn vertikalt via en **Set Position**-operation, vilket ger den slutliga terrängytan i z -led.



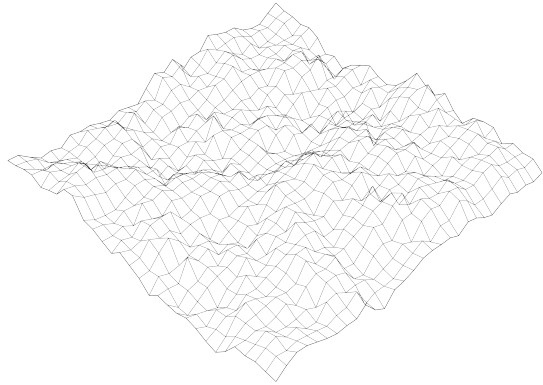
(a) Rutnätsplan med stort brus. Parametrar: stor skala = 0.25, detaljnivå = 2.0, ojämnhet = 0.5, lakunaritet = 2.0, förvrängning = 0.0 och skala = 2.5.



(b) Rutnätsplan med medium brus. Parametrar: stor skala = 1.0, detaljnivå = 2.0, ojämnhet = 0.5, lakunaritet = 2.0, förvrängning = 0.0 och skala = 1.5.



(c) Rutnätsplan med litet brus. Parametrar: stor skala = 5.0, detaljnivå = 2.0, ojämnhet = 0.5, lakunaritet = 2.0, förvrängning = 0.0 och skala = 1.0.



(d) Kombinerad representation av stort, medium och litet brus. Figuren visar hur de tre brusnivåerna kan samverka för att skapa en mer varierad terrängstruktur.

Figur A.1: Jämförelse mellan tre olika brusnivåer på ett rutnät samt en kombinerad version. Stort brus ger större övergripande variationer, medium brus skapar mellanstora förändringar och litet brus bidrar med finare detaljer.

A.2.2 Väggenerering

Vägnätet genereras inuti en **Repeat Zone** vars iterationsantal slumpas från intervallet $[Min, Max]$ *number of roads*. För varje iteration skapas en rät linjemesh som först roteras slumpmässigt kring den vertikala axeln och förflyttas till en slumpmässig position inom världens utbredning. Linjen deformeras därefter lateralt med en tvådimensionell Perlin-brusfunktion, vilket ger den en organisk, slingrande form snarare än en rak sträckning. Brusets frekvens och amplitud randomiseras individuellt per väg, så att olika vägar får olika grad av krökning. Samtliga genererade vägkurvor slås samman via en **Join Geometry**-operation till ett gemensamt vägnät.

För att vägarna ska följa terrängens topografi appliceras därefter en *shrinkwrap*-operation. Vägkurvorna lyfts till en hög nivå, varifrån strålar kastas rakt nedåt mot terrängytan med en **Raycast**-nod. Vägpunkter som inte träffar terrängen tas bort, medan övriga snäpps fast vid sina respektive träffpunkter via en **Set Position**-operation. Detta säkerställer att vägarna ligger i markplanet och följer landskapets konturer.

Vägbredden realiseras inte som geometri utan som en avståndsbaserad mask. En **Geometry Proximity**-nod beräknar varje terrängpunkts avstånd till närmaste vägpunkt, vilket via en **Map Range**-nod omvandlas till ett skalärt maskvärde styrt av parametern *Road width*. Maskvärdet sparas som ett namngivet attribut på geometrin och tilldelas tillsammans med ett dedikerat vägmateriäl, vilket möjliggör visuell blandning mellan väg- och omgivande terrängtextur i renderingen.

Eftersom samtliga slumpkomponenter är frögenererade från *World seed* och respektive iterationsindex är hela vägnätet reproducerbart för en given seed. Slutligen utnyttjas vägmasken som selektion i en avslutande **Set Position**-operation, vars vertikala offset styrs av parametern *Road depth*. Endast de hörn som tillhör vägen förskjuts nedåt, vilket geometriskt urholkar vägarna i terrängen och ger dem ett naturligt nedsänkt utseende snarare än att enbart utgöra ytliga texturer.

A.2.3 Skogsmask och trädplacering

Skogens utbredning bestäms av ett täthetsfält som genereras genom att en fyrdimensionell Perlin-brusfunktion samplas över terrängens markplan. Brusets fjärde dimension drivs av en frögenererad konstant från *World seed*, vilket innebär att varje värld effektivt erhåller ett unikt snitt genom samma brusrymd och därmed en distinkt skogsfördelning.

Brusvärdet passerar därefter en **Color Ramp** följt av två förstärkande **Multiply**-noder och en **Map Range**-operation, vilket tröskar fältet till tydligt avgränsade skogs-zoner respektive öppna gläntor istället för en jämn spridning över hela kartan.

Det resulterande täthetsfältet styr en **Scatter on Surface**-nod som fördelar trädinstanser över terrängens yta. Distributionen baseras på en Poisson-disk-algoritm, vilket garanterar ett minsta avstånd mellan intilliggande träd och därigenom en na-



Figur A.2: De olika stegen i genereringen av vägnätet. Först skapas ett slumpmässigt slingrande vägnät med Perlin-brus, därefter projiceras vägarna mot terrängytan med en shrinkwrap-liknande operation, och slutligen genereras en avståndsbaserad mask som används för att representera vägbredd och materialblandning.

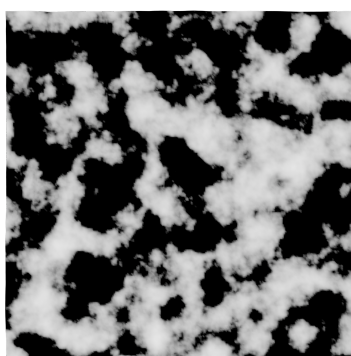
turlig spridning utan överlapp. Instanserna hämtas slumpmässigt från en kollektion av förberedda trädmodeller och orienteras längs terrängens ytnormaler, med slumpmässig variation i skala och rotation kring den vertikala axeln för att bryta synliga upprepningsmönster.

Slutligen konverteras instanserna till verklig geometri via en `Realize Instances`-operation. Detta steg krävs eftersom den färdiga världen senare exporteras och voxeliseras, vilket förutsätter faktisk mesh-geometri snarare än enbart procedurella instansinstruktioner.

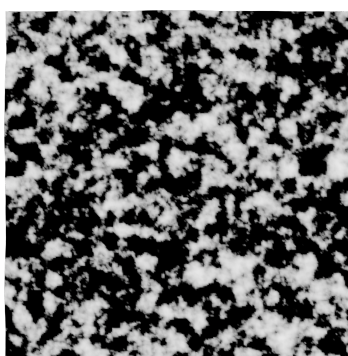
A.3 Stadsgeneratorns nodbaserade implementation

Stadsgeneratoren delar samma grundläggande pipeline-arkitektur som skogsgeneratoren men skiljer sig i hur de procedurellt placerade objekten fördelas över ytan. Markytan utgörs av ett platt rutnät vars dimensioner styrs av *World size* och som direkt tilldelas ett gräsmaterial via `Set Material`.

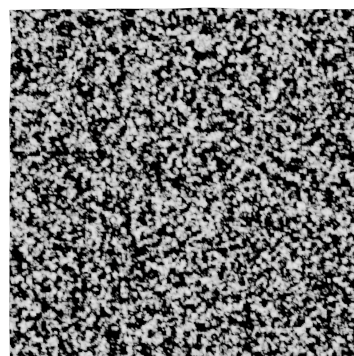
För att efterlikna en stads typiska klusterstruktur, där bebyggelsen koncentreras till vissa områden snarare än sprids jämnt, kombineras två tredimensionella Perlin-brus av olika skalor. Ett högfrekvent brus styr lokala täthetsvariationer inom kvarter, medan ett lågfrekvent brus definierar stadens storskaliga uppdelning i tätbebyggda kärnor och glesare ytterområden. Bidragen blandas i en `Mix`-nod till ett gemensamt



Skogsmask med seed 1.
Täthetsfält genererat genom sampling av en fyrdimensionell Perlin-brusfunktion, där *World seed* styr snittet genom brusrymden.



Skogsmask med seed 2.
Samma nodbaserade genereringsmetod används, men en annan seed ger en ny fördelning av skogszoner och öppna gläntor.



Skogsmask med seed 3.
Ytterligare en reproducerbar skogsfördelning, där vita områden motsvarar högre sannolikhet för trädplacering och svarta områden öppna ytor.

Figur A.3: Exempel på tre skogsmasker genererade med samma procedur men olika *World seed*. Den fjärde dimensionen i Perlin-bruset styrs av seed-värdet, vilket gör att varje seed ger ett distinkt men reproducerbart täthetsfält. Maskerna används för att styra var trädinstanser sprids över terrängen, där ljusare områden motsvarar högre trädthet och mörkare områden motsvarar öppnare ytor.

täthetsfält som även förskjuter markens hörn lätt vertikalt, vilket bryter den helt plana basytan och ger staden en mer organisk form.

Det resulterande täthetsfältet styr en **Scatter on Surface**-nod som distribuerar byggnadsinstanser från en kollektion benämnd **buildings** över markytan. Distributionen baseras på en Poisson-disk-algoritm Bridson (2007) med ett fastställt minsta avstånd mellan instanser, vilket implicit reserverar utrymme för gator och trottoarer mellan byggnaderna. Till skillnad från skogsgeneratoren är instansernas rotation inte uppriktad mot ytans normal, eftersom byggnader ska stå lodrätt oberoende av markens lutning.

För att uppnå realistisk variation i stadens fasader skalas varje byggnadsinstans individuellt i samtliga tre rymddimensioner via en **Scale Instances**-nod. Skalfaktorerna i *x*- och *y*-led styr varje byggnads horisontella utbredning, medan *z*-skalan styr dess höjd. Samtliga skalvärden slumpas oberoende per instans utifrån **Index**-baserad frögenerering, vilket säkerställer att samma byggnadsmodell kan återanvändas i flera olika proportioner och därigenom skapa intryck av en stadsbild utan att kollektionen behöver innehålla ett stort antal unika modeller. Slutligen konverteras instanserna till verklig mesh-geometri via **Realize Instances** inför export och voxelisering.

A.4 Havsgeneratorns nodbaserade implementation

Havsgeneratoren implementerar en betydligt enklare pipeline än skogs- och stadsgeneratorerna eftersom havsmiljön i denna studie endast utgörs av en vattenyta utan ytterligare scatter-objekt. En plan rutnätsmesh med dimensioner styrda av *World size* och vertex-täthet styrd av *World resolution* utgör basen.

Vattnets ytstruktur skapas genom att tre olika texturkällor kombineras: ett periodiskt vågmönster genererat av en **Wave Texture** med diagonal riktning för att efterlikna svallvågor, en högfrekvent Perlin-brusfunktion som introducerar lokal oregelbundenhet och bryter vågmönstrets periodicitet, samt en lågfrekvent brusbaserad mask som styr hur de två komponenterna blandas över ytan. Den senare maskeras genom en **Color Ramp** och fungerar som faktor i en **Mix**-nod, vilket innebär att olika delar av havet får olika balans mellan strukturerade vågor och kaotiska ytstörningar.

Resultatet förskjuter rutnätets hörn vertikalt via en **Set Position**-operation och tilldelas slutligen ett dedikerat vattenmaterial. Genom att aktivera **Set Shade Smooth** elimineras synliga polygonkanter, så att vattenytan upplevs som en kontinuerlig snarare än kantig yta i renderingen.

A.5 Förberedelse av träd- och byggnadsmodeller

Trädsamlingen består av sex lågpolymodeller från Xee (2022). Byggnadssamlingen består av fem distinkta lågpolymodeller från David (2025) och Babia (2026), där merparten kommer från Amal Davids modellpaket och en enskild modell från 3D Studios fria sortiment. Eftersom både skogs- och stadsvärldarna kan innehålla ett stort antal instanser, och eftersom hela geometrin i ett senare steg voxeliseras till ett octree för drönarnas sökoperationer, ställdes höga krav på att varje enskild modell skulle ha låg geometrisk komplexitet.

För att uppnå detta tillämpades Blenders **Decimate**-modifierare i *Collapse*-läge på samtliga träd- och byggnadsmodeller. Modifieraren reducerar antalet trianglar genom en kantkollapsmetod baserad på kvadratiska felmått, där närliggande hörn iterativt slås samman och kanter tas bort med så liten påverkan som möjligt på modellens övergripande form Garland och Heckbert (1997). Resultatet är förenklade modeller som bibehåller objektens visuella form samtidigt som den totala scenkomplexiteten hålls på en hanterbar nivå.

Trädmodellerna sparas i en **Collection** som senare refereras av **Scatter on Surface**-noden i skogsgeneratoren, medan byggnadsmodellerna sparas i en separat **buildings**-kollektion som används av stadsgeneratoren. På så sätt kan generatorerna återanvända samma förberedda modelluppsättningar i många olika världar utan att varje värld behöver modelleras manuellt.

A.6 Automatisk exportering av världar och texturer

För att automatisera genereringen av många världar utvecklades ett Python-skript som körs internt i Blender. Skriptet itererar över ett konfigurerbart antal världar och utför för varje iteration fem huvudsteg.

1. Duplicering av källobjekt. Det procedurella källobjektet, det vill säga skogs-, stads- eller havsgeneratoren med tillhörande **Geometry Nodes**-modifierare, dupliceras till en arbetskollektion. Originalen lämnas därmed orörta och kan återanvändas i efterföljande iterationer.

2. Randomisering av parametrar. Skriptet identifierar samtliga ingångssocketar i generatorns nodgrupp via `node_group.interface`-API:et och slumpar deras värden inom på förhand angivna intervall. Heltalsparametrar, såsom *World seed*, hanteras separat från flyttalsparametrar för att respektera Blenders interna typkrav. Eftersom *World seed* ingår i randomiseringen blir varje genererad värld deterministiskt unik samtidigt som hela genereringen är reproducerbar givet samma slumpfrö i Python-skriptet.

3. Applicering av Geometry Nodes. Efter parametersättningen appliceras modifieraren, vilket konverterar den procedurellt definierade geometrin till en statisk mesh av verkliga hörn och polygoner. Detta steg är nödvändigt eftersom **Geometry Nodes** inte överförs till externa filformat; den exporterade världen måste bestå av faktisk geometri snarare än instruktioner för att generera den.

4. Renderering och tilldelning av terrängtextur. Den procedurella terrängens material består i Blender av nodbaserade shader-grafer som inte överlever export till statiska format. För att bevara terrängens visuella utseende omvandlas färginformationen därför till en bildtextur. En initial ansats med Blenders **Smart UV Project** och traditionell bake-operation valdes bort eftersom smart-projektion delar upp meshen i många små UV-öar, vilket lämnar en stor andel av texturytan oanvänd.

Istället renderas en *topnedåtsbild* av enbart terrängytan via en ortografisk kamera, och resultatet appliceras som textur med en plan top-down-UV-projektion. På så sätt utgör hela texturens area en sammanhängande UV-ö som motsvarar markens projicerade yta sett uppifrån. Renderingen utförs enligt följande delprocess:

- En temporär kopia av meshen skapas där samtliga ytor som inte tillhör terrängens materialslot tas bort, så att endast markytan ingår i renderingen.
- Terrängens **Principled BSDF**-shader ersätts temporärt med en **Emission**-shader. Detta säkerställer att den renderade bilden representerar materialets rena färginformation utan påverkan från ljus, skuggor eller reflektioner.
- En ortografisk kamera placeras rakt ovanför scenen med en synfältsskala som matchar terrängens utbredning, varefter en bild med upplösningen 1024×1024 pixlar renderas via Blenders **EEVEE**-motor och sparas som PNG.

- Terrängmeshens UV-koordinater omprojicieras med en plan top-down-projektion så att den renderade bilden mappas korrekt till varje hörn.
- Den sparade bilden tilldelas till ett nytt **Principled** BSDF-baserat material som ersätter det ursprungliga proceduralt definierade materialet.

Asset-objekt såsom träd och byggnader hanteras med en kompletterande strategi. Istället för att tilldela dessa unika eller delade UV-mappar med tillhörande texturer sätts varje enskild face hos modellerna manuellt till en specifik solid basfärg innan modellerna placeras i sin kollektion. Eftersom dessa material redan utgörs av rena färgvärden utan texturer behövs varken UV-uppvikning eller bake-operationer på dem. Kombinationen av top-down-rendering för terrängen och solidfärgad face-baserad representation av assets gör att världens visuella komplexitet kan fångas i en enda relativt liten texturfil per värld.

5. Export. Den färdiga världen exporteras slutligen till formatet *glTF Binary* (.glb). Detta format valdes efter att flera alternativa exportformat, bland annat .fbx, .obj och .stl, utvärderades empiriskt med avseende på hur väl de bevarade material- och texturinformation vid import till Gazebo. Av de testade formaten gav glTF Binary den mest konsekventa och korrekta återgivningen av världens visuella utseende i simulatorn, samtidigt som det inkluderar geometri, material och inbäddade texturer i en enda binär fil.

Efter exporten städas den duplicerade meshen och de tillhörande temporära materialen och bilderna upp ur Blenders datablock-hantering, så att efterföljande iterationer påbörjas i ett rent tillstånd och minnesåtgången hålls konstant oavsett antal genererade världar.

B

Voxelisering

B.1 Voxeliseringskedja

Steg 1: Ytvoxelisering av mesh. I det första steget körs ett Python-skript baserat på `trimesh`-biblioteket som läser in världens GLB-fil och konverterar dess geometri till en uppsättning kubiska voxlar med konfigurerbar sidlängd. Voxeliseringen sker som en ren *ytvoxelisering* snarare än en fylld voxelisering, vilket innebär att endast de voxlar som skär meshens yta markeras som ockuperade medan voxlarna i objektens inre lämnas tomma.

Detta val gjordes eftersom en drönare inte fysiskt kan befinna sig inuti en byggnad eller en trädstam under simuleringen, och eftersom det endast är ytan som är relevant för kollisionsdetektion och raycast-baserad sensorsimulering. En fylld voxelisering skulle dessutom öka antalet ockuperade voxlar med flera storleksordningar och därmed öka både lagrings- och beräkningsbehoven utan att tillföra användbar information.

I detta steg utförs även de koordinattransformationer som krävs för att överbrygga skillnader mellan mjukvarukonventionerna i pipelinen. GLB-formatet följer glTF-standarderna där y -axeln pekar uppåt, medan ROS 2 och Gazebo använder en konvention där z -axeln pekar uppåt. Skriptet stöder därför en explicit rotation kring x -axeln med $+90^\circ$ för att hantera detta, samt en separat axelrotation för att överföra Blender-koordinater till ROS 2-konventionen. Resultatet av steg 1 är en CSV-fil där varje rad anger (x, y, z) -koordinaterna för centrum av en ockuperad voxel uttryckt i målsystemets koordinater.

Steg 2: Konvertering till OctoMap-binärformat. Det andra steget är implementerat i C++ och baseras på `OctoMap`-biblioteket. Skriptet läser CSV-filen och inserterar varje voxelkoordinat i en `octomap::OcTree`-instans med en specificerad upplösning. För varje punkt anropas `updateNode(...)` med ockupationsstatus markerad som sann, vilket placerar voxeln i trädets nivå som motsvarar upplösningen.

Efter att samtliga voxlar har lagts till anropas `updateInnerOccupancy()`, vilket propagerar ockupationsinformationen uppåt genom trädets hierarki så att även de inre föräldranoderna återspeglar tillståndet hos sina barn. Slutligen serialiseras trädets till en kompakt binärfil med ändelsen `.bt` via `writeBinary()`.

B.2 CSV- och BT-representationerna

Båda filtyperna beskriver i grunden samma uppsättning ockuperade voxlar men har olika egenskaper och användningsområden. CSV-filen är en *linjär lista* av voxelcentrum, vilket gör den enkel att inspektera, visualisera i externa verktyg och konvertera till andra format. Däremot är den beräkningsmässigt kostsam att söka i för att avgöra om en given punkt är ockuperad eller att utföra en raycast kräver en linjär genomsökning av samtliga rader och har därmed komplexitet $O(n)$.

Den binära `.bt`-filen lagrar däremot samma information som ett *octree*, en hierarkisk trädstruktur där varje nod rekursivt delar upp rymden i åtta lika stora oktanter. Denna struktur möjliggör logaritmisk komplexitet $O(\log n)$ för punktuppslagningar och god prestanda för raycasting och grannuppslagning, vilket är de operationer som dominerar drönarnas söklogik under simulering. Hierarkin innebär dessutom att stora homogena rymdvolymer, typiskt sett tomma områden ovanför terrängen, kan representeras av en enda nod på en hög nivå. Därmed blir formatet betydligt mer minneseffektivt än en motsvarande tät voxelmatris för glesa miljöer.

CSV-filen behålls trots detta även efter konverteringen, eftersom den fyller två kompletterande funktioner. Den används dels som grund för en intern datastruktur i `octree_search_node` där varje voxel bär per-voxel-tillstånd som uppdateras allteftersom området söks av, dels som en mer transparent och felsökningsvänlig grund för att vid behov regenerera `.bt`-filen med en annan upplösning utan att behöva voxelisera om GLB-meshen från grunden.

B.3 Val av voxelupplösning

Voxlarnas sidlängd är en central parameter som direkt styr avvägningen mellan rumslig precision och beräkningskostnad. En fin upplösning ger en mer noggrann representation av små hinder och fina detaljer, men antalet ockuperade voxlar växer kvadratisk med inversen av sidlängden för en given yta.

Inledningsvis utvärderades en upplösning på 0,2m, men för stadsvärldarna producerade detta så stora octree-filer att RAM-konsumtionen översteg flera tiotals gigabyte vid laddning. Detta gjorde simuleringen omöjlig att köra på den tilldelade hårdvaran. Som en konsekvens valdes en grövre upplösning på 1,0m för slutgiltig användning. Denna upplösning reducerar antalet voxlar för en typisk stadsvärld från cirka 2 000 000 till omkring 70 000 och håller minnesförbrukningen på en hanterbar nivå utan att meningsfullt försämra drönarnas förmåga att navigera och söka i miljön.

B.4 Användning under simulering

I det körande systemet används de genererade artefakterna parallellt och fyller kompletterande funktioner. Den exporterade GLB-filen laddas in i Gazebo, där den an-

vänds för den visuella renderingen av världen. CSV- och `.bt`-filerna laddas däremot in av en dedikerad ROS 2-nod, `octree_search_node`, som ansvarar för drönarnas raycast-baserade söklogik.

Inom noden fyller de två representationerna kompletterande roller: `.bt`-filen avserialliseras till en `octomap::OcTree`-struktur som används för effektiva raycast-frågor mot världen, medan CSV-filen läses in i en separat datastruktur där varje voxel bär ett tillståndsvärde som uppdateras allteftersom området söks av.

Visualiseringen i RViz sker indirekt. Noden konstruerar en `MarkerArray` baserad på voxelarnas aktuella tillstånd och publicerar denna på en ROS 2-topic som RViz prenumererar på. RViz läser därmed inte själva filartefakterna direkt, utan visualiserar enbart den lägesbild som noden i realtid sammanställer från octree-strukturen och voxeltilstånd. Genom denna uppdelning kan visuell rendering, algoritmisk söklogik och tillståndsvisualisering utvecklas och optimeras oberoende av varandra utan att påverka varandras prestanda.

C

Sök- och perceptionsinfrastruktur

C.1 Inläsning och publicering av världen

Vid uppstart läser `octree_search_node` in de två filerna som beskrevs i voxeliseringsavsnittet. `voxels.csv` läses in via en linjär genomsökning och varje rad konverteras till en `Voxel`-struktur som lagras både i en sekventiell vektor och i en hash-baserad indexstruktur (`std::unordered_map`) nycklad på voxelns kvantiserade lokala koordinater. Detta möjliggör konstant tid för uppslagning av en specifik voxel utifrån en godtycklig punkt i världen, vilket utnyttjas senare i sökregistreringen.

Den binära `.bt`-filen avserialiseras i sin tur till en `octomap::OcTree`-instans via `readBinary`, vilken används för effektiv raycasting.

Noden tillhandahåller även en användarkonfigurerbar transformation som vid inläsning kan applicera en skalning, en rotation kring de tre axlarna, parameteriserad som roll, pitch och yaw, samt en translation på samtliga voxelkoordinater. Rotationen sammanställs som $R = R_z(\text{yaw})R_y(\text{pitch})R_x(\text{roll})$. Mekanismen är avsedd för att vid behov kunna korrigera skillnader i axelorientering mellan Blender-exporten och simuleringens världssram.

I den aktuella konfigurationen är samtliga rotations- och translationsparametrar satta till noll och skalfaktorerna till ett, eftersom motsvarande korrigering redan utförs i voxeliseringssteget i `obj_to_voxels.py` via flaggan `--y_up_to_z_up`. Detta innebär att voxelarna är fullständigt orienterade i målkoordinatsystemet redan när de skrivs till CSV-filen, och att runtime-transformen reduceras till en identitetsoperation.

Efter inläsningen konstruerar noden en uppsättning ROS2-publicerare och prenumranter. På publiceringssidan tillhandahålls fyra centrala topics: `octree_markers` och `octree_drone_markers` (`visualization_msgs::msg::MarkerArray`) för visualisering i RViz, `/octree/search_feedback` (`swarm_msgs::msg::SearchFeedback`) för algoritmernas sökfeedback, `/octree/collision_alert` (`swarm_msgs::msg::CollisionAlert`) för kollisionslarm, samt `/octree/target_hits` (`swarm_msgs::msg::TargetHit`) för registrerade måldetekteringar. Visualiserings-topics publiceras med `transient_local` QoS, vilket innebär att noder som ansluter sent ändå får en aktuell ögonblicksbild av världstillståndet.

På prenumerantsidan lyssnar noden på `/drone/position` för positionsuppdateringar från varje drönare samt på `/targets/state` för aktuella målpositioner. En timer triggas vid frekvensen `publish_hz`, satt till 10 Hz i den aktuella systemkonfigurationen, och ansvarar för periodisk uppdatering av visualiseringsmarkörerna.

C.2 Konbaserad raycasting och voxelintersection

Konstruktionen av strålnippet sker en gång vid uppstart och bygger på två konfigurerbara parametrar: `cone_rays`, som anger antalet strålar och i den aktuella systemkonfigurationen är satt till 500, och `cone_half_angle_deg`, som anger konens halva öppningsvinkel och är satt till 30°.

Strålarna fördelas på ett kvadratisk rutnät i tangentplanet kring konens centralaxel, där alla rutpositioner som hamnar utanför konens cirkulära begränsning förkastas. Centralaxeln styrs av parametern `ray_axis_local` och pekar rakt nedåt $(0, 0, -1)$ i lokal koordinat.

När en drönares tillstånd kommer in i `OnDroneState`-callbacken konverteras positionen först från ArduPilots NED-konvention, där z -axeln pekar nedåt, till simuleringens ENU-konvention genom att z -komponenten inverteras. Den ENU-uttryckta positionen transformeras därefter till lokal koordinat via inversen av samma rotation och skalning som användes vid inläsningen, så att raycastingen kan ske direkt mot den lokala octree-strukturen.

För varje stråle i konen anropas `tree->castRay` med drönarens lokala position som ursprung och strålens normaliserade riktningsvektor som riktning, begränsad till maximalt avstånd `ray_max_range_m`, satt till 80 m i den aktuella systemkonfigurationen. Funktionen returnerar en boolesk indikator för om strålen träffar en ockuperad voxel innan den passerar maximalt avstånd, samt skriver i så fall träffpositionen till en utdataparameter. Resultatet av varje tick är en lista av strålsegment per drönare, märkt som antingen träff eller miss.

C.3 Sökregistrering och tillståndsspårning

Sökregistreringen implementeras i `MarkHitNeighborhood`. Varje träffpunkt expanderas till en sfärisk omgivning med radie `hit_radius_m`, satt till 7,0 m i den aktuella systemkonfigurationen för att täcka en meningsfull omgivning kring varje träffpunkt vid den valda voxelupplösningen på 1,0 m. Inom denna sfär itereras samtliga voxelceller fram via en boundingbox-genomsökning följt av ett euklidiskt avståndstest. För varje cell vars position finns indexerad i `voxel_index_` uppdateras den tillhörande `search_score`-attributen.

Sökpoängen varierar i intervallet $[0.05, 1.0]$. Implementationen tillhandahåller en distansberoende avbildning där värdet bestäms av strålens längd vid träfftillfället, så

att kortare strålar ger högre poäng än längre. Avbildningen är linjär mellan parametrarna `search_score_close_m` och `search_score_far_m`, vilket i princip möjliggör en gradvis viktning av sensors pålitlighet utifrån avståndet till den observerade ytan.

I den aktuella systemkonfigurationen är `search_score_far_m` satt till 800 m, ett värde betydligt större än någon stråle kan uppnå i praktiken. Därmed tilldelas samtliga träffar det högsta poängvärdet 1,0 oavsett avstånd. Detta val gjordes för att förenkla utvärderingen, eftersom en gradvis sökpoäng skulle ha krävt att drönarna flög mycket nära marken för att räkna en cell som fullständigt avsökt. För att kompensera för detta begränsas drönarnas tillåtna flyghöjd till maximalt 70 m. När en cell uppdateras behålls alltid det högsta hittills observerade värdet, så att senare observationer från sämre vinklar inte överskrider tidigare bättre observationer.

För att undvika onödig beräkning och kommunikationslast republiceras voxelvisualiseringen endast när söktillståndet faktiskt förändrats. Detta implementeras via en `voxels_dirty_-flagga` som sätts till sann när minst en cells `search_score` förbättras, och som rensas så snart en uppdaterad markörmängd har skickats.

Parallellt med visualiseringen publicerar noden ett **SearchFeedback**-meddelande på topic `/octree/search_feedback` efter varje raycast-tick. Meddelandet innehåller den observerande drönarens identifierare och nuvarande position i ENU-koordinat, en lista `hit_points` med samtliga direkta träffpunkter från den aktuella raycast-omgången, samt två aggregerade fält: `searched_count`, vilket är summan av samtliga voxlars `search_score` i världen, och `total_count`, vilket anger det totala antalet voxlar. Listan av träffpunkter inkluderar endast de positioner där raycast-strålarna faktiskt skär ockuperade voxlar, och inte den utvidgade voxel-omgivning som `hit_radius_m` markerar internt.

C.4 Kollisionsundvikande

Utöver den nedåtriktade sökkonen utför noden en separat omnidirektionell raycast för att detektera närliggande hinder. Riktningarna konstrueras vid uppstart genom att ett $5 \times 5 \times 5$ -rutnät centrerat på origo används, varifrån mittpunkten exkluderas och de återstående 124 vektorerna normaliseras.

I varje tick utförs raycasts från drönarens position längs samtliga 124 riktningar med maximal räckvidd `collision_range_m`, satt till 20,0 m i den aktuella systemkonfigurationen. För varje träff beräknas en motstående vektor riktad bort från träffpunkten, viktad med inversen av kvadraten på avståndet, så att närliggande hinder bidrar oproportionerligt mycket till den sammanvägda riktningen. De vikta bidragen summeras och normaliseras till en enhetsvektor som tillsammans med det minimala observerade hinderavståndet utgör innehållet i ett **CollisionAlert**-meddelande. Detta publiceras på topic `/octree/collision_alert` endast när minst ett hinder har upptäckts inom räckvidden.

Brain-noden prenumererar på `/octree/collision_alert` och underhåller alltid det senast mottagna larmet. När det är dags att skicka ett rörelsekommando jämförs avståndet till närmaste hinder med en konfigurerbar tröskel. Om avståndet ligger under tröskeln blandas drönarens målriktning, definierad av den ovanpåliggande sökalgoritmens publicerade waypoint, med den motstående vektorn från larmet via en linjär interpolation vars vikt ökar i takt med att hindret kommer närmare. Den blandade riktningen omvandlas slutligen till ett hastighetskommando i NED-konvention och skickas till flygkontrollen.

Utöver terränghinder hanterar brain-noden även separation mellan drönarna i svärmen som en additiv komponent i samma rörelseberäkning. Varje drönare jämför kontinuerligt sin position med övriga drönares senast kända positioner, och om avståndet till en granne underskrider en konfigurerbar separationsradie ackumuleras en evasionsvektor riktad bort från grannen. Beräkningen sker i tvådimensionellt horisontellt plan, eftersom drönarna i den aktuella konfigurationen alla flyger i samma höjdintervall och vertikal separation därför inte är relevant.

D

Målens representation och rörelse

D.1 Representation och placering

Varje mål utgörs av en logisk entitet med ett unikt heltalsidentifierare, en aktuell världsposition i ENU-koordinat, en boolesk `found`-flagga som anger om målet har upptäckts, samt en `last_dir`-vektor som lagrar den senaste rörelseförskjutningen för användning i rörelsemodellens tröghetsterm. Målen är inte fysiska Gazebo-entiteter med kollisionsegenskaper, utan rena ROS 2-entiteter vars positioner förflyttas via direkta tilldelningar i mållistan.

Detta designval gjordes eftersom drönarnas detektion sker via raycast-baserad geometrisk uppslagning i `octree_search_node` snarare än via Gazebos sensorsimulering krävs ingen fysisk representation av målen för att uppnå korrekt detektionslogik, och simuleringskostnaden hålls därigenom betydligt lägre.

Vid uppstart kan mål placeras enligt två alternativa strategier. Om det aktuella scenariot innehåller en `voxels_csv`-referens, det vill säga en voxeliserad representation av världen, samplas slumpmässiga ockuperade voxlar från denna och målen placeras vid motsvarande världskoordinater. Detta säkerställer att målen alltid hamnar på en navigerbar yta eller struktur snarare än i tomrum, vilket är nödvändigt för efterföljande rörelsebegränsningar.

Om voxelinformation saknas faller noden tillbaka på en alternativ strategi där målpositioner läses direkt från scenariots `mission.targets.positions`-fält. Målen placeras då vid de explicit angivna koordinaterna och hålls statiska under hela uppdraget.

D.2 Rörelsemodell

Målen förflyttas i diskreta tidssteg via en timer som triggas vid frekvensen `update_rate_hz`, satt till 1,0 Hz i den aktuella konfigurationen. Förflyttningen sker som en sannolikhetsbaserad slumpvandring på samma tredimensionella voxelnät som används av övriga delar av systemet.

För varje aktivt mål undersöks de 27 närliggande voxelpositioner som omger målets nuvarande cell, motsvarande hela $3 \times 3 \times 3$ -omgivningen inklusive nuvarande position. För varje grannposition genomförs ett ockupationstest mot voxelnätet. Endast

positioner som motsvarar en faktiskt existerande voxel utgör giltiga rörelsealternativ, vilket innebär att målen aldrig kan röra sig ut i tomrum eller passera genom föremål utan en sammanhängande yta att färdas på.

Den underliggande sannolikhetsfördelningen för rörelseval definieras i konfigurationsfilen `target_movement.json` via tre 3×3 -matriser, en per höjdrelativ nivå: *below*, *same* och *above*. Var och en av matriserna anger sannolikheten att målet förflyttas till respektive cell på den motsvarande nivån under ett tick.

Värdena i den aktuella konfigurationen är valda för att återspegla en realistisk rörelseprofil för en sök- och räddnings-relevant entitet såsom en vilsegången person. Centralcellen i *same*-matrisen, motsvarande att stå still, har det högsta värdet 0,30, riktningarna i samma plan har värdet 0,08, diagonalriktningarna 0,05, och samtliga vertikala rörelser har betydligt lägre sannolikheter mellan 0,02 och 0,05. Detta ger en modell där målen oftast antingen står still eller förflyttar sig kort horisontellt, och endast sällan byter höjdnivå.

Vid uppstart läses matriserna in och flätas samman till en enskild vektor med 27 element, vilket möjliggör snabb indexering under det tidskritiska samplingssteget.

D.3 Tröghet i rörelsen

För att undvika att målens rörelse uppfattas som en ryckig slumpvandring där riktningen kontinuerligt ändras varje tick implementerar noden en tröghetskomponent baserad på senaste rörelseriktningen. Vid varje tick beräknas, för samtliga giltiga grannar utöver centralcellen, skalärprodukten mellan målets `last_dir`-vektor och kandidatförskjutningen.

Denna skalärprodukt avbildas till en multiplikator $1 + g \cdot \cos \theta$, där g är parametern `heading_persist_gain`, satt till 0,8, och $\cos \theta$ är skalärprodukten normaliserad till intervallet $[-1, 1]$. Multiplikatorn appliceras på den ursprungliga sannolikheten för respektive grannposition. En kandidat i exakt samma riktning som föregående rörelse får därmed sin sannolikhet upp till 80% förstärkt, medan en kandidat i motsatt riktning får motsvarande nedjusterad.

De justerade sannolikheterna normaliseras därefter till en giltig fördelning över de tillåtna grannpositionerna, varefter en ny målposition väljs via stratifierad slumpning. Om målet faktiskt har förflyttats uppdateras `last_dir` till den valda förskjutningen så att tröghetstermen kan användas i nästa tick. Stillastående tick lämnar däremot `last_dir` oförändrad, så att rörelsemomentum bevaras även genom korta vilopausar.

D.4 Tillståndspublicering och detektion

Efter varje rörelsetick publicerar noden ett `TargetState`-meddelande på topic `/targets/state`. Meddelandet innehåller en lista av samtliga målidentifierare och deras nuvarande världspositioner. Informationen utnyttjas både av `octree_search_node` för måldetektionens avståndstest mot raycast-träffpunkter och av drönarnas brain-noder för loggning och utvärdering.

Publiceringen sker med `transient_local` QoS, vilket innebär att en nod som ansluter sent till topiken ändå omedelbart erhåller den senast publicerade målmängden utan att behöva vänta på nästa periodiska uppdatering.

När `octree_search_node` detekterar att en drönars raycast har passerat tillräckligt nära ett målobjekt publiceras ett `TargetHit`-meddelande på `/octree/target_hits`. `targets_manager_node` prenumererar på denna topic och uppdaterar det matchande målets `found`-flagga till sann när ett detekteringsmeddelande tas emot.

Effekten är att målet omedelbart upphör att förflyttas i efterföljande tick. Rörelseuppdateringen kontrollerar denna flagga vid början av varje cykel och hoppar över ett detekterat mål helt. Detta motsvarar antagandet att ett upptäckt mål därefter omhändertas av räddningspersonal och inte längre utgör ett aktivt sökobjekt, samtidigt som målet kvarstår i mållistan så att dess slutposition förblir tillgänglig för efterföljande utvärdering av algoritmernas prestanda.

E

Pseudokod

Listing E.1: Exempel på scenariokonfiguration.

```
{
  "behavior": {
    "name": "bmuta",
    "params": {
      "auction_timeout_s": 1.5
    },
    "version": ""
  },
  "drones": {
    "default": {
      "name_prefix": "iris_",
      "yaw_deg": 90.0,
      "z": 0.195
    },
    "spawns": [
      {
        "id": 0,
        "x": 0.0,
        "y": -10.0
      },
      {
        "id": 1,
        "x": 5.0,
        "y": -10.0
      },
      {
        "id": 2,
        "x": -5.0,
        "y": -10.0
      },
      {
        "id": 3,
        "x": -10,
        "y": -10
      }
    ]
  }
}
```

```
    ]
  },
  "mission": {
    "cone_rays": 8,
    "map_resolution_m": 1.0,
    "takeoff_alt_m": 50.0,
    "targets": {
      "area": {
        "xmax": 75,
        "xmin": -75,
        "ymax": 75,
        "ymin": -75
      },
      "count": 5,
      "placement": "random_uniform",
      "positions": [
        {
          "id": 0,
          "x": 55.0,
          "y": 40.0
        },
        {
          "id": 1,
          "x": -50.0,
          "y": 55.0
        },
        {
          "id": 2,
          "x": -60.0,
          "y": -45.0
        },
        {
          "id": 3,
          "x": 45.0,
          "y": -55.0
        },
        {
          "id": 4,
          "x": 30.0,
          "y": 65.0
        }
      ]
    }
  },
  "scenario_id": "cond_c310d8da2515",
  "schema_version": 1,
```

```
"seeds": {
  "agents": 73443978,
  "targets": 4259893201
},
"swarm_size": 4,
"vehicle": {
  "ardupilot": {
    "home_yaw_deg": 90.0,
    "param_file": "configs/ardupilot/iris_default.parm"
  },
  "model_sdf": "gz_ws/install/share/ardupilot_gazebo/models/iris_wi
},
"world": {
  "bounds": {
    "xmax": 75,
    "xmin": -75,
    "ymax": 75,
    "ymin": -75
  },
  "map_asset": {
    "collision_enabled": false,
    "mesh_path": "world/Maps/CityMaps/glb_worldsize150_worldres37
    "octree_bt": "world/MapsBT/CityMaps/glb_worldsize150_worldres
    "scale": {
      "x": 1.0,
      "y": 1.0,
      "z": 1.0
    },
    "transform": {
      "pitch": 0.0,
      "roll": 0.0,
      "x": 0.0,
      "y": 0.0,
      "yaw": 0.0,
      "z": 0.0
    },
    "voxels_csv": "world/MapsBT/CityMaps/glb_worldsize150_worldre
  },
  "name": "iris_runway_swarm",
  "sdf": "gz_ws/install/share/swarm/worlds/iris_runway.sdf"
}
}
```

Listing E.2: Exempel på experimentkonfiguration.

```
campaign:
  name: "thesis_draft"
```



```
    output_root: "runs"

execution:
  replications_per_world: 6
  stop_on_failure: false
  global_timeout_s: 600

seeding:
  base_seed: 271828

sweep:
  world_types: [city, forest, water]
  world_sizes: [80, 150]
  world_instances_limit: 4

behaviors:
  - name: "recontamination_search"
    params: { map_resolution_m: 1.0 }
  - name: "pso"
    params: { map_resolution_m: 1.0 }

swarm_sizes: [3, 5, 8]
target_counts: [10]

spawn_formation:
  - { name: "grid_origin", type: "grid", spacing_m: 5.0 }
  - { name: "border", type: "border", margin_m: 5.0 }
  - { name: "corner_sw", type: "corner", corner: "sw" }
```

E.1 Experiment-fil för analys

Listing E.3: Experimentkonfiguration för experimentuppställning.

```
campaign:
  name: "thesis_final"
  output_root: "runs"

execution:
  replications_per_world: 5
  stop_on_failure: false
  global_timeout_s: 650

seeding:
  base_seed: 271828

mission:
```

```
takeoff_alt_m: 50.0
map_resolution_m: 1.0
cone_rays: 8
targets:
  count: 10
  placement: "random_uniform"
  min_spawn_dist_m: 15.0

sweep:
  world_types: [city, forest, water]
  world_sizes: [150]
  world_instances_limit: 5

behaviors:
  - name: "bmuta"
  - name: "search_rescue"
  - name: "recontamination_search"
    params: { config_file: "sim/behavior_configs/recontamination_search" }
  - name: "pso"

swarm_sizes: [3, 5, 8]
target_counts: [10]

spawn_formation:
  - { name: "grid", type: "grid", spacing_m: 5.0 }
  - { name: "border", type: "border", margin_m: 5.0 }
```