Hamburg University of Technology
Institute for Software Systems

Diploma Thesis

AURIS
Autonomous Robot Interaction Simulation

Author: Gerry Siegemund
Supervisors: Prof. Dr. Ralf Möller
Prof. Dr. Karl-Heinz Zimmermann

Start: January 3 2011
End: July 4 2011

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own work except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged. All resources used are explicitly referenced in the bibliography.

Hamburg, July 4, 2011

Gerry Siegemund

Acknowledgments

Thanks to all the people that contributed to this thesis and that supported me along the way. Especially, Katrin Preiß who kept me sane while not writing and August Betzler who kept my writing sane.

Hamburg, July 4, 2011

Contents

Abstract

This thesis extends an environment that simulates quadrotors and their physical properties. Furthermore, a program is developed that coordinates the autonomous behavior of a number of unmanned aerial vehicles. Tests of the system are conducted with single agents as well as with groups of quadrotors. This setup can be adapted to fit a number of different requirements to be of use for a wide range of developers. Beyond that, algorithms are proposed which motivate the employment of quadrotors, and that can be used to measure the performance of the proposed system.

1 Introduction

In the fields of miniature robotics various researches are conducted. Especially wheeled robots have been studied over the last decades [16] [3]. Since these robots drive on the ground their working area is usually mapped to a two-dimensional space. The analysis of flying robots on the other hand is a newer field, which evolves rapidly. New problems arise in this filed because the work area is a three-dimensional environment. These so-called Unmanned Aerial Vehicles (UAVs) can be equipped with different scanners, ranging from altitude and attitude sensors to cameras and advanced laser measurement technologies. Furthermore, provided with high computation power, the UAVs can evaluate sensor data while in the air. Quadrotors, i.e., miniature helicopters with four instead of one rotor, are one example to represent this field of research. Figure 1.1 shows a picture of the Hamburg University of Technology's quadrotor (project website: [46]).

Figure 1.1: Apollo Quadrotor Hamburg Tech. courtesy of [11]

Humans usually steer these vehicles, but due to the sensors and the computation power, autonomous behavior can be programmed. Autonomy for robots means, no human guidance, sensing and evaluating the environment, and rationally acting upon these informations. Different concepts have been designed to enable autonomous behavior [1] [19]. Therefore, UAV could also be changed to AAV, Autonomous Aerial Vehicle, in analogy to Autonomous Underwater Vehicles AUVs.

Equipped with communication devices, robots can also interact and build groups, which can accomplish tasks. These tasks vary from scanning areas for certain objects, to contentiously monitoring different entities, or transporting substances to a marked position. For wheeled robots swarm ideas have been studied thoroughly, for UAVs on the other hand, researches and especially implementations in this area have just begun, as can be seen in Figure 1.2. Creating groups of quadrotors, or other UAVs, costs money and work power. If erroneous behavior is accidentally programmed, crashes can be the result, increasing the expenses further.

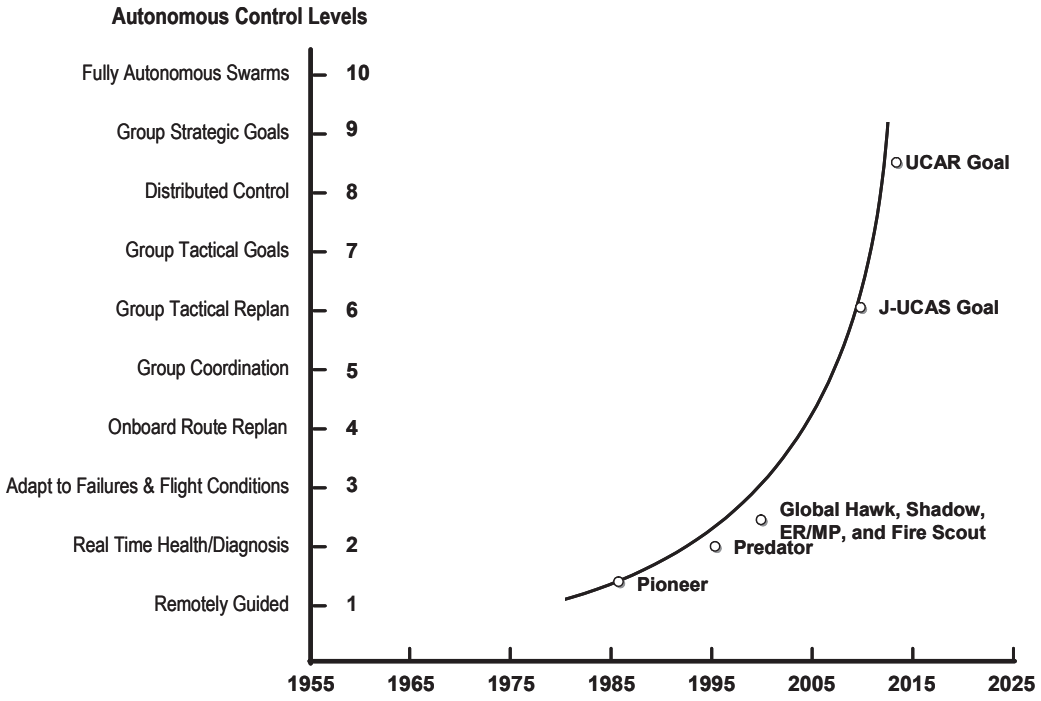


Figure 1.2: "Unmanned Aircraft Systems Raodmap" courtesy of [20]

Therefore, a simulation software should be used, to be able to minimize errors of proposed algorithms. The goal of this thesis is to extend a system that simulates robots, and their physical properties in a three-dimensional environment. Furthermore, a control-program named AURIS is developed, which coordinates the autonomous behavior of an agent. The program's design needs to be amendable to be used in guidance control of actual UAVs, too.

The simulation software adopted is USARSim, a middle-ware build upon the game Unreal Tournament 3. AURIS communicates with the simulation trough a TCP/IP connection. In a real world application, a link between AURIS and the UAV's controller system needs to be established, to interpret the simple instructions given. Figure 1.3 shows the communication setup.

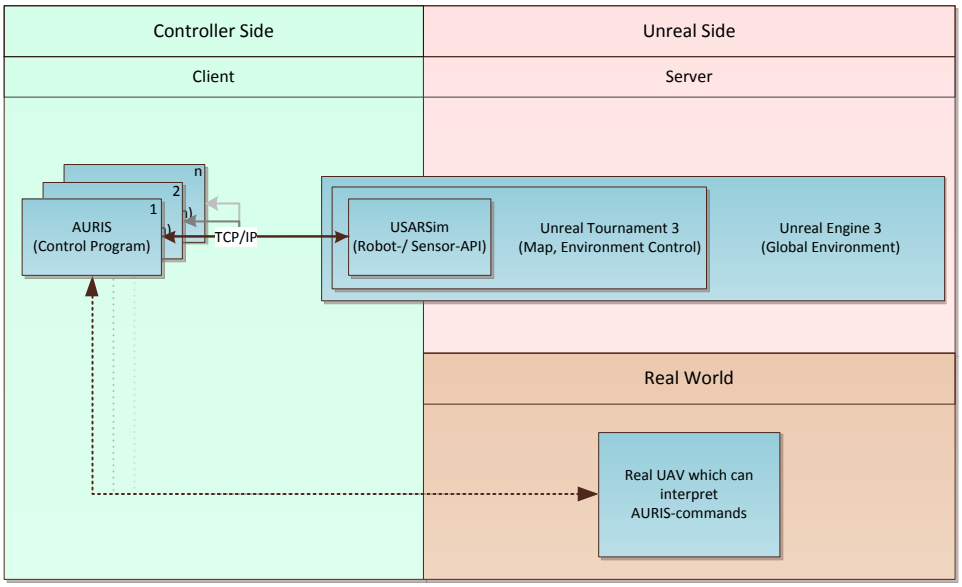


Figure 1.3: Communication setup, between AURIS and the simulation environment

First the simulated model and the appropriate physical behavior of a quadrotor are added to the USARSim system. A flight stabilizer is introduced to make the movement of the UAV easier controllable by AURIS. For proposed tasks, e.g., fire and smoke detection, new sensors are developed. The simulation is supposed to be as close to reality as possible. Therefore, errors to sensor output are added, this degraded data has then to be handled by the control-program.

Search missions, where the UAV has to find a fire, are run, thereby verifying the functionality of AURIS, and of the built quadrotor-model, as well as its attached sensors.

In Chapter 2, preliminaries, the underlying control-theory basics are explained, as well as, mathematical fundamentals. Chapter 3 discusses the design of the simulation environment, in particular the quadrotor assembly. Followed by the development of AURIS in Chapter 4. Chapter 5 introduces applications of multi agent systems. The last chapters evaluate the presented implementations, as well as drawing conclusions based on the accomplished experimental and theoretical work.

2 Preliminaries

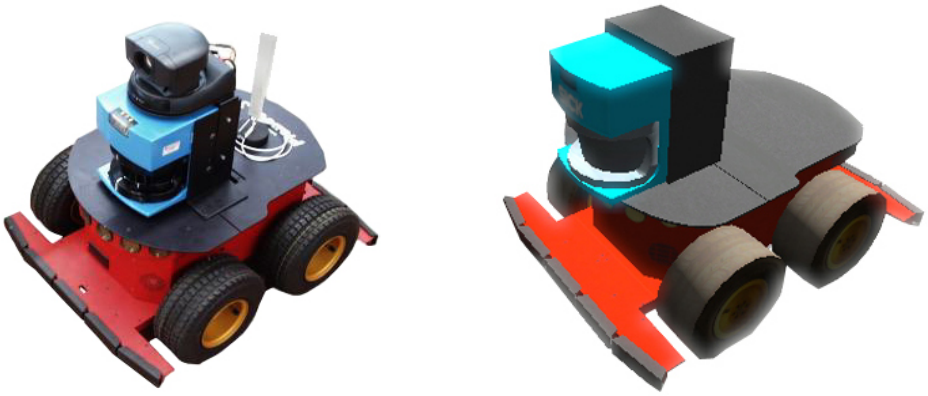
This chapter describes the basics behind the construction of the simulation, the robots, and the underlying control systems. For the reader it can be seen as a reference for later chapters to come back to. At the beginning the simulation software (USARSim) and the system it is built upon (Unreal Game Engine 3) are explained. Furthermore, control theory basics used in later chapters are touched.

2.1 Simulation Environment

USARSim (Unified System for Automation and Robot Simulation) [49] is based on the Unreal Tournament game engine. It emulates the behavior of multiple robots in an environment. As a middle-ware between a control program and the Unreal Engine (UE) its used to manage a number of agents in various situations. USARSim is mainly intended as a research tool, but among others it is also the basis for the RoboCup rescue virtual robot competition (RoboCup Rescue) [48], the IEEE Virtual Manufacturing Automation Competition (VMAC) [50], and the DARPA grand challenge [44].

USARSim is a very powerful, well designed tool for robot simulation of any kind. It enables the developer to load different three-dimensional (3D) environments and add a number of agents which are able to behave in arbitrary ways. Several robots are already built as a 3D model, i.e., consisting of 3D-meshes and Unreal Script classes, to be used in the USARSim environment. All of them are four-wheel robots that can be moved on the ground, for example, the P3AT (Figure 2.1) from Pioneer. The environments, or maps, that host the simulation can be designed to meet virtually any requirement given by underwater, space, town, or desert scenarios.

Note that USARSim only enables developers to use a variety of robots and sensors, it is not a generator for autonomous behavior.



(a) Original **(b)** Simulated Version

Figure 2.1: Pioneer robot

2.1.1 Projects and Related Work

USARSim enables researchers and developers to test their products before they are actually built. Wherever 3D simulation might be helpful, USARSim could be an option. So far, a variety of different robots and environments have been developed, among others, highway robots, the DARPA grand challenge, robotic soccer (see Figure 2.2), submarines, humanoids, and helicopters. For every developed environment, there are again multiple possible scenarios. Even though a number of websites concern themselves with USARSim ([44], [50], [48], [52]) and robot simulation, no other research team is working on swarm behavior of UAVs with USARSim, yet.

Besides USARSim there are many other robot simulation development kits available. Appendix A.4 shows a comparison between different softwares, some of them are mentioned here in more detail. Gazebo [45] for instance is a multi-robot 3D simulator, similar to USARSim. The physics engine used is Open Dynamics Engine (ODE [51]). The graphics of Gazebo can not keep up with the Unreal Engine's, especially because UE was built for commercial 3D game simulation.

Another promising simulation tool is SimRobot [37], it has been developed by the University of Bremen, Germany, over 16 years now. The software is open-source, just like Gazebo and the physics engine is ODE, too. SimRobot consists of several modules which are linked to become one single application. This differs from Gazebos or USARSims

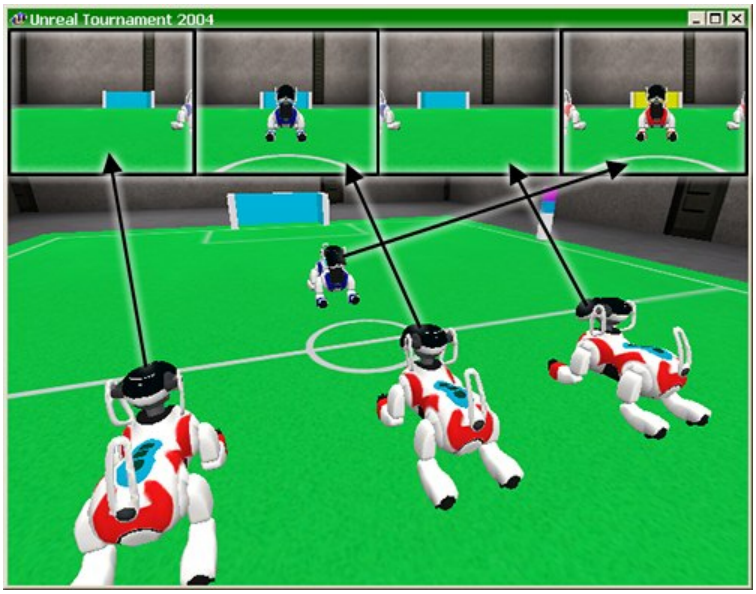


Figure 2.2: UT2004 (USARSim 2) robotic soccer simulation [29]

client-server architecture, it offers the possibility of pausing or stepwise executing of the whole simulation. The main shortcoming is that new robots can not be added to a running simulation, therefore making this simulator a poor choices.

Webots [53] is a commercial development environment used to model, program and simulate mobile robots. Being a commercial software, the biggest advantage of Webots is user support and constant documentation. Nevertheless, the modeling tools of Webots are not as powerful as the Unreal Editor, also the graphics can not compete. Furthermore, the price of about 250 EUR[1] is a five fold of the cost for Unreal Tournament 3. USARSim is an open-source distribution, therefore the base code can be modified to fit special simulation needs, which Webots might not be able to fulfill.

For the mentioned reasons, USARSim is the chosen platform for the quadrotor simulation. Especially the client-server architecture of USARSim enables the developer to write a control program in any programming language able handle networking, completely decoupling the simulation from the control unit.

[1] Approximatly 350 USD for the EDU version, stated on the website [53]

2.1.2 USARSim Progression

USARSim is built with the Unreal Game Engine (UE). Three versions of this engine have been developed so far. The first generation, with games like Unreal Tournament 2003, was used for the original USARSim. As the Unreal Engine progressed so did USARSim. After the second generation, with games like Unreal Tournament 2004, followed the current third generation. Unreal Tournament 3 which is based on UE3, has been around for about three years now (release: Nov. 2007), but not all the features of USARSim 2 (UE2) have been ported to the present engine, yet. Between the two versions UE2 and UE3, there have been more changes than between UE1 and UE2. For example, the KARMA physics engine was replaced by nVidia PhysX in UE3. Therefore, the transition from USARSim 2 to USARSim 3 is more time consuming. The robot design changed, too. The ready-to-use quadrotor from the prior USARSim was not implemented in USARSim 3. Accordingly, recreating the flying robot is the first challenge to be overcome. Section 3.1 will elaborate the work done further.

2.1.3 Coordinates and Units

USARSim resembles a deterministic simulation, which uses discrete time measurement with interpolation. In the contrary, stochastic simulation is typically used for systems where events occur probabilistically. In this simulation, the time t evolves with respect to the real world, and is measured by the computers CPU. At a discrete time value Δt the systems current state can be evaluated. During the elapsed time all system states are interpolated. The time value Δt is unspecified, it usually stays in the bounds of 0.01 seconds to 0.1 seconds. Since USARSim is a 3D-simulation the picture a spectator sees is also updated every Δt, that means, the frame-rate, or frames per second (FPS) is directly proportional to Δt ($1FPS = \frac{1}{\Delta t}$). For the user to watch a fluent 3D-scene the frame-rate needs to stay above 20FPS ($\Delta t = 0.05s$).

The spacial units in the simulation are proportional to meters. An Unreal Unit (UU) resembles 0.04 centimeters. Because time and distance measurements are a like, all calculated units using these values are, too, e.g., velocity $1\frac{m}{s} \Leftrightarrow 250\frac{UU}{s}$. For convenience, unit converters are in place, i.e., the user can always refer to meters while internally Unreal Units are used.

Three different coordinate systems are used to determine the position of the UAV, and its surrounding. The local reference frame can be seen in Figure 2.3, it is the quadrotor's coordinate system. The global frame is the Unreal Engine's view. Last there is the map

drawn by the agent on the user's computer, in two-dimensions.

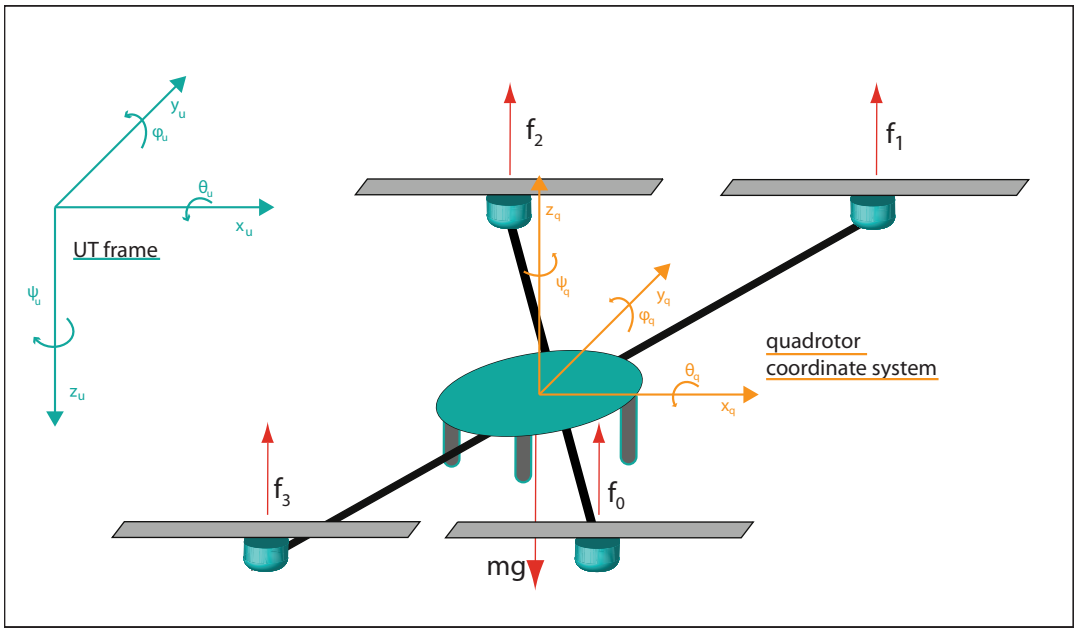


Figure 2.3: Quadrotor sketch with local coordinate system (q) and Unreal Coordinate System reference (u)

Conversions between the different coordinate systems can be done by rotation matrices. Either the rotation around each angle R_ϕ, R_θ, R_ψ is done consecutively or with one rotation matrix R, i.e.,

$$\begin{aligned} R &= R_\phi R_\theta R_\psi \\ &= \begin{bmatrix} \cos\theta\cos\psi & \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi \\ \cos\theta\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi \\ -\sin\theta & \sin\phi\cos\theta & \cos\phi\cos\theta \end{bmatrix}. \end{aligned} \tag{2.1}$$

The map drawn by the quadrotor uses the computer picture format, that is, the origin is in the left upper corner and x- and y-coordinates grow only positively. The map is a two dimensional entity so a projection from the three dimensional environment has to be done. Furthermore, the Unreal Engine uses positive and negative position values, with the origin usually in the middle of the map. For the 3D to 2D conversion the z-value is dropped and the x- and y-values are adapted using this formula:

$$\begin{pmatrix} x_m \\ y_m \end{pmatrix} = s \begin{pmatrix} -y_u \\ x_u \end{pmatrix} + \begin{pmatrix} X_o \\ Y_o \end{pmatrix}, \tag{2.2}$$

where $(x_m, y_m)^T$ are the new coordinates in the map, $(x_u, y_u)^T$ the Unreal Coordinates, $(X_o, Y_o)^T$ the offset between the drawn map origin and the Unreal environment origin, and s a scaling factor to reduce the drawn map size. The scaling factor might be different for any environment, depending on its size.

For the conversion, the z-values are not used but when the map is drawn, the color used is influenced by the z-coordinate. The lighter a color gets the higher the UAV's altitude. This is done for any information drawn, e.g., scanned area, obstacles, or other UAVs in range. Figure 2.4 shows how the color is adjusted for a certain altitude.

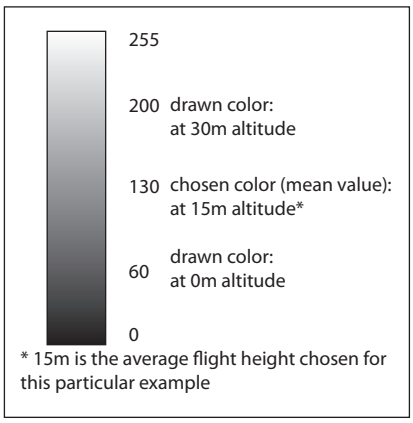


Figure 2.4: Color profile proportional change of color to altitude

2.2 Unreal Game Engine 3

The Unreal Engine 3 (UE3) is a computer game engine developed by Epic Games, succeeding the Unreal Engine 1 and 2. It is designed for DirectX 9/10/11 PCs, the Xbox 360, and the PlayStation 3. It also offers OpenGL-based operation system (Mac OS X, iOS) support. A Linux version was proposed, but has not been released.

The underlying physical computations are done by Nvidias PhysX physics engine. It exceeds the Karma engine from the prior Unreal Engines.

This thesis will only be concerned about the PC / Windows Version of the software, precisely, with the game Unreal Tournament 3 (UT3). Figure 2.5 shows the DVD cover of the game.

Figure 2.5: UT3 DVD cover

2.2.1 Unreal Script

The syntax used by the unreal scripting language resembles the C++ syntax. UnrealScript has object oriented features and supports inheritance, interfaces and operator overloading. Actual C++ functions can also be included to support a certain functionality in the Unreal environment, these classes are referred to as native code.

When a class is inherited from a basic UE class (like the "Pawn" Class, for instance[2]), then also a tick function is present. The Unreal Engine updates every mesh, every picture, and anything of movement once every tick. A tick is a time Δt, which in most cases is smaller than one second ($0.01s < \Delta t < 0.1s$). Δt is the shortest time in which all actors in a running environment are updated (as mentioned in Section 2.1.3). The update rate is only limited by CPU and graphics accelerator power.

A class can be added into the simulation if it is marked *placeable*. From inside a placeable class, other classes can be called, too. Any motion controlling calculations need to be in the tick-function, of a placeable class. In a way it can be understood as a "main"-function, which is executed at every time step Δt.

2.2.2 Building Robots, Sensors, and Maps

USARSim is a collection of features, ready-to-use for a simulation. Only wheeled robots are currently available. Furthermore, prepared sensors and maps might not fit all purposes desired. Therefore, new robots, sensors and maps need to be developed. Developing new sensors or robots makes use of Unreal Script. Preexisting robot and sensor classes can be used as templates to further augment the system.

First of all, a static mesh is developed. For this purpose Autodesk's 3ds Max [34] or Maya [35] are recommended by the USARSim community. Figure 2.6 shows the 3ds Max environment used to set up the quadrotor. A skeleton mesh is built to ensure certain parts, e.g., wheels or rotors, are movable. Next, the corresponding Unreal Script class needs to be written. It links the mesh to its behavior. Sensors are built in a similar manner. First a mesh is constructed then the appropriate Unreal Script Class needs to be designed. In Chapter 3 certain steps on how to actually build new equipment is described in more detail.

[2] The Pawn class is a basic UT3 class, it is the parent class of all actors that can be controlled.

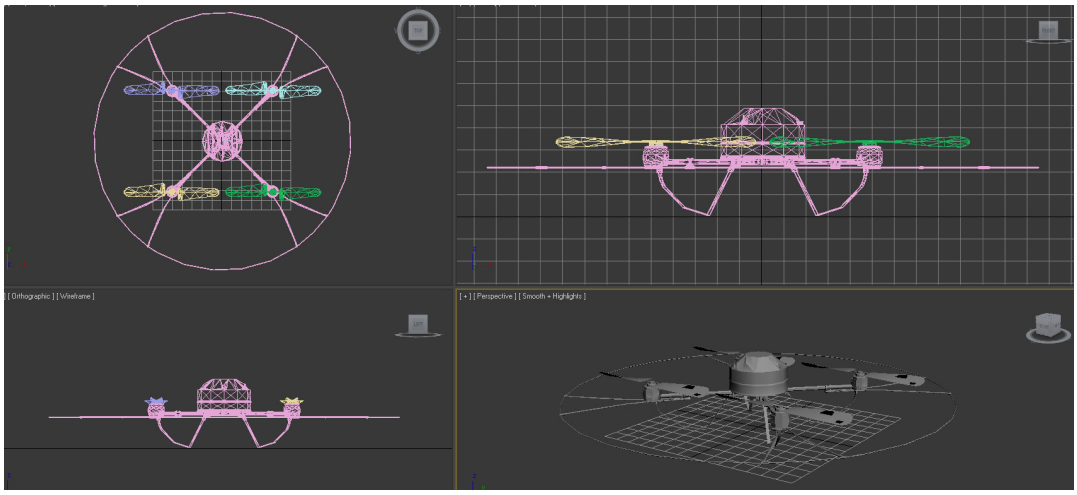


Figure 2.6: 3ds Max working environment

Maps are designed using the Unreal Tournament 3 Editor, which has various tools to design new maps or to manipulate existing ones. Meshes are added from an toolbox with a variety of objects, moving and stationary. They define the ambiance of the simulated scene. Completely new objects can be designed, and added to the scene, as well.

Meshes can have certain behaviors, e.g., a bird could be designed, flying around in the scenario. In addition, basic responses to other actors are implementable. Artists and developers have been designing maps for different reasons, mostly to play the game UT3 in another setting. These maps can be adapted to fit special simulation needs, without the time consuming process of building an own map [54].

Unreal Editor The game Unreal Tournament 3 comes with an editor, the Unreal Editor (UnrealEd), it is used to import new meshes, making it possible to place them into a simulation scenario. Furthermore, it helps manipulating maps, enabling a developer to change all objects in the current scene, as well as creating new ones from scratch. In addition, the UnrealEd includes tools to change the behavior of actors. Anim Trees and Physics Assets, as subprograms in the UnrealEd, are mentioned because they are needed to create robots.

Anim Trees or Animation Trees are graphical representations of Unreal Script Classes which can be used to specify their behavior. Animations, Blend Nodes, Bone Controllers and Morph Nodes can be added to Anim Tree Nodes. Especially for graphical performance specification Anim Trees are necessary. For instance, if an object in a 3D

simulation gets destroyed, then the Anim Tree specifies which mesh is loaded to represent the broken object. For more informations on Anim Trees the Epic Games Website can be helpful [39].

With Physics Assets physical behavior can be specified [40], e.g., the collision area, that is the area around an object which can have an effect on the object itself and on the one it collides with. Every placeable object needs a Physics Asset and an Anim Tree.

2.2.3 Vector Rotations

In a three-dimensional space, an orientation vector ($\vec{\theta} = (\theta_x, \theta_y, \theta_z)^T$) describes an attitude. When turning around this vector, it is not clear in which direction.

Therefore, quaternions [3] are used to describe attitudes in a three-dimensional environment, with the advantage over an Eulerian angle representation, that quaternions define the rotation uniquely. Internally, quaternions are used to represent vector rotations. Therefore, their mathematical structure is explained here. Quaternions are reminiscent to the complex numbers. As a vector space quaternions are defined as:

$$w + xi + yj + zk, \tag{2.3}$$

where i, j, and k are complex numbers, with the following rules

$$\begin{aligned} i^2 = j^2 = k^2 = ijk = -1, \\ ij = k, \\ ji = -k, \end{aligned} \tag{2.4}$$

here x, y, z, and w are real numbers. w is a scalar and $(x, y, z)^T$ is called the vector part of the quaternion. This representation for a quaternion will be used:

[3] First described by Hamilton, in 1843 [9]

$$\vec{q} = \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix}. \tag{2.5}$$

For rotations between different coordinate systems quaternions can be convenient. Let $\vec{q}_{u\to t} = (w,x,y,z)^T$ describe the transformation from a coordinate system u to system t. The inverse of a quaternion also describes the inverse transformation, it is constructed as shown:

$$(\vec{q}_{u\to t})^{-1} = \begin{pmatrix} w \\ -x \\ -y \\ -z \end{pmatrix} = \vec{q}_{t\to u}. \tag{2.6}$$

To rotate a vector two multiplications and a vector augmentation are needed. The vector is treated as a quaternion with its scalar value set to zero:

$$\begin{aligned} \vec{v} \in \Re_3 \Leftrightarrow (0, v_x, v_y, v_z)^T \in \Re_4 \\ \vec{\tilde{v}} = \vec{q}\,\vec{v}\,\vec{q}^{-1}. \end{aligned} \tag{2.7}$$

More informations on quaternions can be found at [6]. For completeness the quaternion multiplication is shown in Appendix A.2, because it is used internally in the Unreal Engine.

2.3 System Theory and Filter Basics

For every simulation trying to be as close to nature as possible is the most important proposition. Accordingly, errors in measurements are essential to accomplish a realistic simulation. Dealing with erroneous input and output requires the same tools as in the real world, for example, the Kalman filter approach is used to filter and correct sensor data.

2.3.1 Increasing the Realism of Sensor Output

A sensor is nothing else than a unreal script class. It can use virtually any function given by the unreal engine. That means, the data captured by a sensor is always perfect, because it is data retrieved from the unreal engine. For an realistic simulation, the data needs to be corrupted.

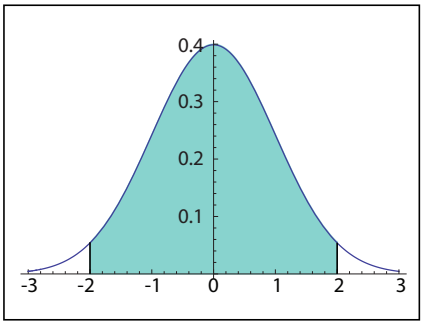


Figure 2.7: Gaussian distributed, $\sigma = 1$, $\alpha = 2$

A function has been implemented to add Gaussian noise to any sensor data. The variance σ and a cutoff value α can be set for any sensor that measures discrete values (Figure 2.7 shows a Gaussian distributed with a cutoff value of 2). For sensors which return boolean values, e.g., the victim detection sensor, false positives and false negatives can be added. The crucial part is to set the tuning parameters according to the used sensor. A laser scanner, for example, has a smaller variance compared to a sonar scanner. Most scanners in the real world state their expected error, this explicit value should be used in the simulation, too.

Figure 2.8 shows the process of data capturing and evaluation. First the data is distorted then it its transfered to AURIS, which tries to mitigate the error. The best possible outcome is $n(t) = \tilde{n}(t)$.

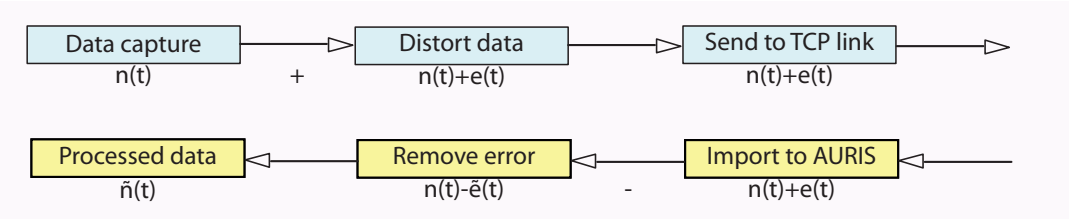


Figure 2.8: Data processing, from the UE to AURIS

2.3.2 Kalman Filter

To gain knowledge of a system, which is only described by erroneous data, a Kalman filter is used. Since all sensors in USARSim can be modified to produce imperfect data, this approach is used wherever sensor data is evaluated.

System Model Considering a linear system with measured data that is afflicted with white noise. The system model with the state vector $\vec{x}$ is:

$$\vec{x}_{k+1} = A_k \vec{x}_k + B_k \vec{u}_k + \vec{w}_k, \tag{2.8}$$

where A is the state transition matrix, B the input matrix, and $\vec{u}_k$ the control vector. $\vec{w}_k$ describes the Gaussian distributed model-error with the covariance matrix Q_k.

The observation $\vec{y}_k$ with the observation matrix H_k at time k looks as follows:

$$\vec{y}_k = H_k \vec{x}_k + \vec{v}_k, \tag{2.9}$$

where $\vec{v}_k$ is the measurement error, as well Gaussian distributed with the covariance matrix R_k.

Prediction Uncertainties are the covariance matrices of the errors, they need to be estimated. The predicted system state vector[4]:

$$\tilde{x}_{k+1} = A_k \tilde{x}_k + B_k u_k, \tag{2.10}$$

where the error covariance matrix is:

$$\tilde{P} = A_k P_k A_k^T + Q_k. \tag{2.11}$$

Estimation Now the prediction needs to be improved by using the current measurements. The new (estimated) system state vector

$$x_k = \tilde{x}_k + K_k \left(y_k - H_k \tilde{x}_k \right), \tag{2.12}$$

and its covariance matrix

[4]The variables x, $\tilde{x}$, and y are vectors, their arrows have been spared for readability reasons.

$$P_k = \tilde{P}_k - K_k H_k \tilde{P}_k^T \tag{2.13}$$

are calculated using the Kalman gain

$$K_k = \tilde{P}_k H_k^T (H_k \tilde{P}_k H_k^T + R_k)^{-1}. \tag{2.14}$$

More informations on Kalman filters can be found in [30, p. 129-147]

2.3.3 PID controller

The proportional integral derivative controller, short PID, is a control scheme, or feedback control system, with the gains K_p, K_i, and K_d as possible tuning parameters. The PID controller is used for various purposes, e.g., to calculate the stabilizing factors for the quadrotor flight.

The controller output $u(t)$ is given by:

$$u(t) = K_p \epsilon(t) + K_i \int_0^t \epsilon(\tau) d\tau + K_d \frac{d}{dt} \epsilon(t), \tag{2.15}$$

where $K_{p,i,d}$ are the tuning parameters, ϵ is the error, and t is the time. The error is calculated as the difference between the current value, and the desired one. Figure 2.9 shows a block diagram of a parallel PID controller.

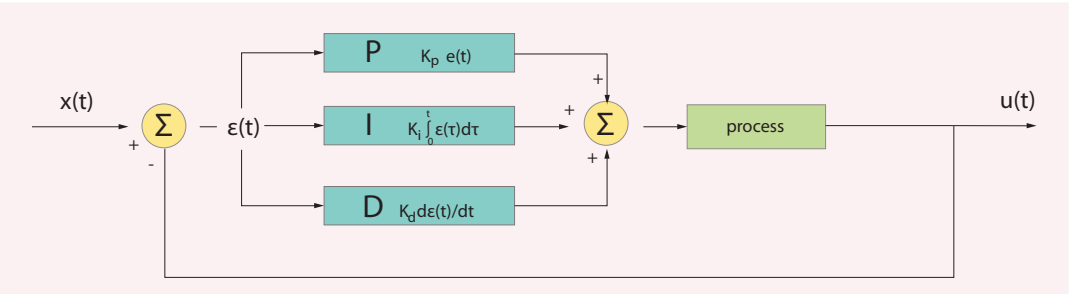


Figure 2.9: PID control loop as block diagram

3 Design

No quadrotor is defined in USARSim 3, and copying the physical behavior from USARSim 2 is impossible, because their underlying physics engine was replaced (see Section 2.1.2). Using the older version was not proposed, because it would diminish the advancements of the Unreal Engine 3. To realize the simulation of the quadrotor in the desired environment, first a computer-aided design (CAD), or mesh, and the essential physical behavior needed to be defined. To unravel the full potential of the UAV, new sensors are needed which have not, or only partly been developed, by the USARSim team.

This chapter elaborates the realization of the quadrotor, and shows the building process of newly designed sensors.

3.1 Building a Quadrotor for USARSim

To build the UAV two subtasks need to be addressed, first the quadrotor mesh has to be defined, second the Unreal Script Class, describing the physical behavior has to created. To enable movement in different directions, ideas how to stabilize the quadrotor are introduced.

3.1.1 Setting up the Static Mesh

To build the quadrotor's static mesh, knowledge of a 3D molding tool is necessary. Either Autodesks 3ds Max or Maya are recommended by the USARSim community. Instead of creating a new quadrotor model that fits the UE3 requirements, a prior version of the UAV used in USARSim 2 (UE2) is remodeled.

Various type conversions are necessary to accomplish this task, because UE2 and UE3 use different mesh types. Additionally, remodeling of the body and the rotors needs to be

done. After creating the static mesh bones are placed on the model to become a skeleton mesh. With a skeleton mesh movement is possible. On each bone torques can act. Therefore, bones are added where the rotors attach to the chassis (Figure 3.1 shows the quadrotor wireframe, with rotors and chassis).

The Unreal Tournament 3 Editor is used to import the modified mesh. With the UnrealEd a Physics Asset is created which is needed to define the collision area around the UAV and other characteristics, e.g., its mass. Furthermore, an Anim Tree is added. It specifies how certain robot parts are moved. Here it is used to state in what way the rotors are turned, i.e., around which axis. Anim Tree and Physics Asset are tools to setup characteristics of a simulation, as mentioned in Section 2.2.2.

After the static mesh is constructed it needs to be assigned to an Unreal Script Class. From this class the static mesh is called, and added into a simulation. Moreover, it defines the basic behavior of the robot, e.g., its translatory movement. The next subsection elaborates the class design further.

3.1.2 Defining Physical Behavior

To be able to explore all features of USARSim to its full extent, knowledge of Unreal Script a necessity. In Section 2.2.1, basics about Unreal Script are mentioned. The following paragraphs explain in detail the underlying concept of the quadrotor flight in a three-dimensional environment.

All vehicles in USARSim inherit the “USARVehicle” Class, it manages the functionality of USARSim, i.e., handling the sensors, the robots, or decorations. For none flying robots, like the P3AT (Figure 2.1), the subsequent hierarchical class is called “GroundVehicle”. It mainly handles the communication between the Unreal Engine and the robot. The next underlying class for this particular robot is the “SkidSteeredRobot” Class the actual movement of the wheels and the speed-handling are its main purpose. Different types of driving robots and of steering types have been developed. Last in the hierarchy is the actual “P3AT” Class, here all the default values for dimensions and movement are set. In addition, the static mesh is linked to its class, enabling the possibility to add the robot to a simulation.

For the quadrotor, the same class design was maintained. The “USARVehicle” Class is still the primary class. Followed by the “AirVehicle” Class, the “RotorVehicle” Class, and the “Quadrotor” Class, as can be seen in Figure 3.1. Preserving this design enables later developers to add other types of flying vehicles like airplanes or helicopters efficiently.

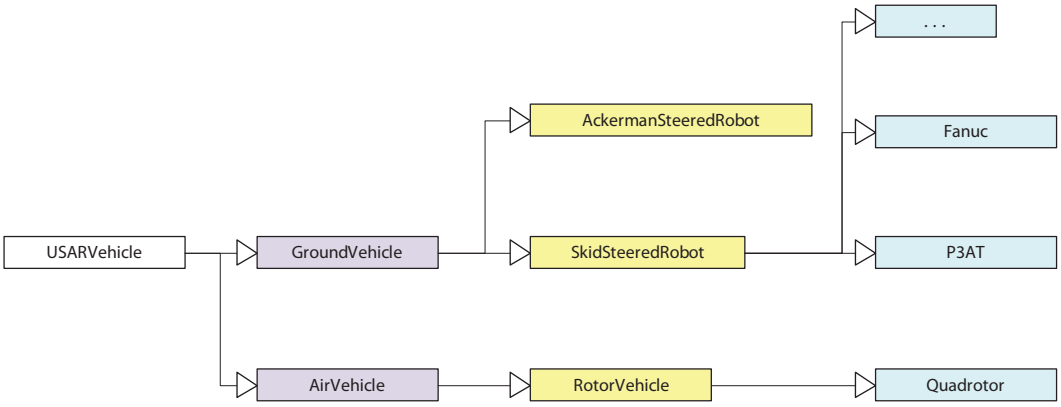


Figure 3.1: Basic USARSim Robot Class Design

Rotor Movement Assembling a realistic simulation requires inertia and friction to act on objects. When the UAV starts the engine the rotors will turn. They need to start slowly until their desired turning speed is reached, vice versa, when shutting off the engine. The uplifting force is proportional to the rotor movement, that means, after starting the engine it will take a moment until the quadrotor lifts off.

To accomplish this behavior four PID controllers[1], one per rotor, are used. In each Tick-Function of the "RotorVehicle" Class, the current error at time k is calculated, $\epsilon_{v_k} = v_g - v_i$, where the current, and the given speed are v_g, and v_i. The cumulative error $\sum \epsilon_{v_f}$ is permanently computed.

The needed acceleration[2] $\vec{a}(t)$, hence the result of the PID controller is given by:

$$a(t) = \ddot{\vec{r}}(t) = P\epsilon_{v_k} + I\epsilon_{v_f} + D\frac{(\epsilon_{v_k} - \epsilon_{v_{k-1}})}{\Delta t}, \tag{3.1}$$

where Δt is measured by the engine and specifies the time between each tick. This is done for each rotor separately. In the next step, the appropriate force will be applied and the calculation continues.

[1] The basics about the PID controller can be found in Chapter 2.3.3
[2] Alternatively denoted by $\ddot{\vec{r}}$, the second derivation of the position vector

The proportional gain K_p, the integral gain K_i, and the derivative gain K_d have been determined by experiments. Settings of $K_p = 1.5$, $K_i = 0.8$, and $K_d = 0.04$ let the rotor movement look realistic. Due to these values the response and settling time are high, which simulates friction, that has to be overcome by the rotors.

Altitude Control Even though the rotors are moving now, the quadrotor will still not leave the ground, because there is no actual force coming from the rotors, which interacts with the mesh of the quadrotor. No lift force or aerodynamic drag is implemented in UE3. Nevertheless, gravity is available as a simulated force vector, which usually points to the ground[3]. The counterforce to overcome it, is implemented to make the UAV fly.

On a static mesh an arbitrary number of forces can be applied. They are attached as vectors on a distinct position on the model. The magnitude and direction of the force-vector are adjustable. Wherever a rotor is placed on the static mesh, there a force-vector $\vec{F}_n$ is attached, as can be observed in Figure 3.2, on the quadrotors wire-frame.

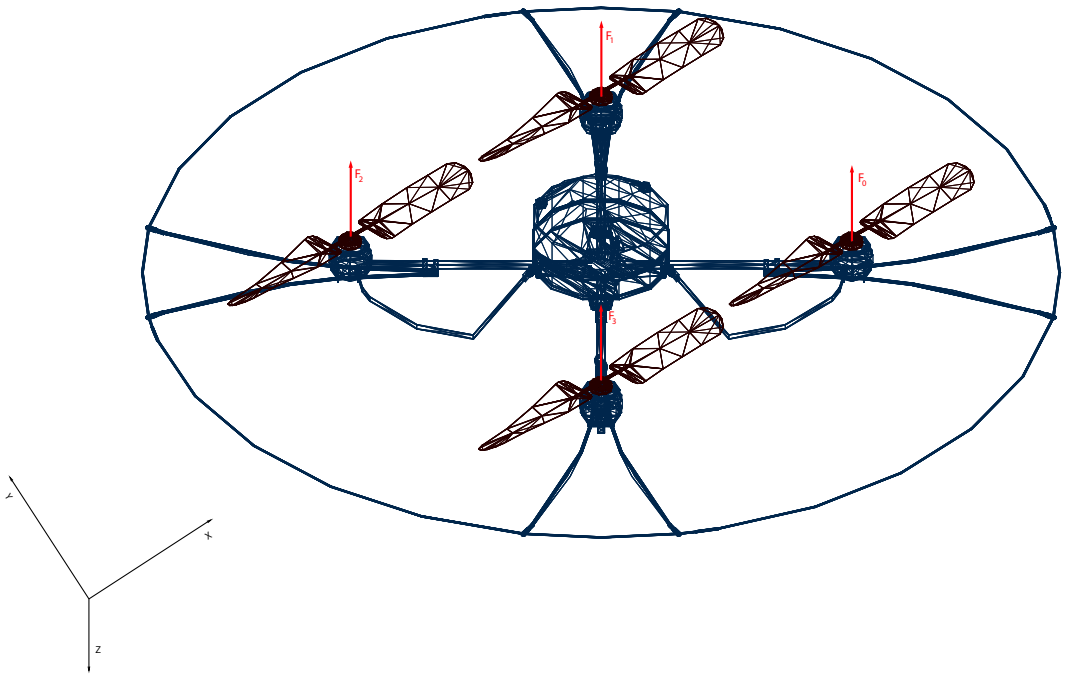


Figure 3.2: The quadrotor wire-frame with attached forces

[3] Gravity is implemented as $\vec{F}_G = gm$, its value and direction can be changed, to simulate different environments, e.g., space $\vec{F}_G = 0$

The rotor speed is implemented to be proportional to the force's strength. For the altitude velocity, the UAV's control program, specifies the rotor speed not the quadrotor's velocity. If the given speed, i.e., the uplift force is below a threshold (gravity) the UAV will not lift off, matching the real world behavior.

The altitude changes as long as the UAV does not roll or pitch. Should there be a discrepancy in those rotations, a different behavior can be witnessed. Because the force-vector points in z-direction of the quadrotor coordinate-frame, a change in roll or pitch angle leads to a rotation of this vector in Unreal Engine coordinates. Should the force-vectors be rotated in UE coordinates sense, then the UAV moves in x- or y-direction.

Flight Stabilizing Lifting off from an uneven ground lets the quadrotor unintentionally fly off to either side, since the force vector does not point upward in UE coordinates. Moreover, if the UAV hovers over one point, due to rounding errors, each rotor will have a different speed after a given amount of time. Even though, this might be an infinitesimal difference, the robot could turn slowly, and then drift off to either side.

This points out the need for a flight stabilizer. If the UAV starts to drift of in one direction, the stabilizer needs to change the speed of the according rotor to compensate. The PID controller approach is used to accomplish this task. If the set speed vector $v_s = (x, y, z)^T$ for the quadrotor is $v_s = (0, 0, \neq 0)$, then the roll (ϕ) and pitch (θ) angle have to be zero. Is that not the case the speed of the appropriate rotors needs to be adjusted.

If the pitch angle differs the rotor pair 0/1 and 2/3 need to be adjusted. The current error in the pitch angle ($\varepsilon_\theta = \theta_s - \theta_i$) the cumulative error ($\varepsilon_{\theta_\int} = \sum \varepsilon_\theta \Delta t$), and the previous error ($\varepsilon_{\theta_{k-1}}$), with the gain values K_p, K_i, and K_d give a correction value

$$c = P\varepsilon_{\theta_k} + I\varepsilon_{\theta_\int} + D\frac{(\varepsilon_{\theta_k} - \varepsilon_{\theta_{k-1}})}{\Delta t}. \tag{3.2}$$

The correctional value c is applied to every rotor, in the current example it is added to the rotor pair 0/3 and subtracted from the other rotor pair.

Virtually, the same is done when $\phi_s - \phi_i \neq 0$. Both corrections are done consecutively, that means, that two PID controller, which each calculate a correction value c_ϕ for the roll angle and c_θ for the pitch angle, are used. The overall correctional value for each "n"-th rotor is given by:

$$v_n = A_n c_\phi + B_n c_\theta \quad n = 0..3\,, \tag{3.3}$$

with $A_n = \{1,-1,-1,1\}$ and $B_n = \{1,1,-1,-1\}$. This approach guarantees, that even if both angels are faulty, the adjustment is applied and the quadrotor is stabilized.

The gain values chosen in the current stabilizer design are $K_p = 3$, $K_i = 0.03$, and $K_d = 0.2$ where found in experiments. The high K_p value helps to respond quick. The chosen tuning parameters prevent any overshoot because the UAV shouldn't oscillate in the air until it is stabilized.

In all run simulations the PID controller worked and the UAV was stabilized. Even though, slight differences between rotor speeds cause strong imbalances. This is a limitation of the simulation environment, which does not interfere with test of swarming or area searches, due to the reliable stabilizer.

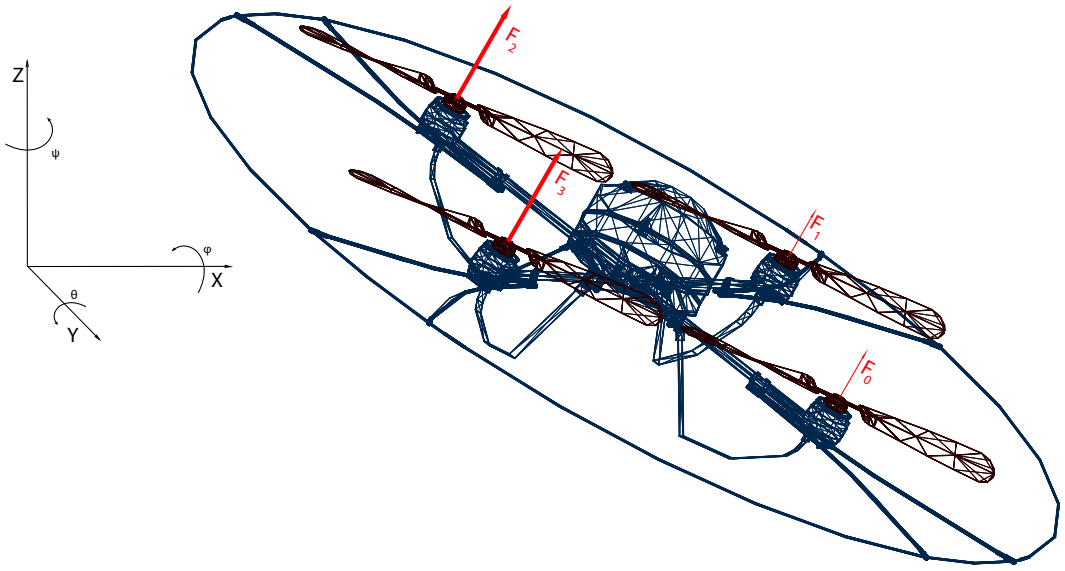


Figure 3.3: Increment of certain rotor forces change the translation of the UAV

Translational Control To move the quadrotor along the x-axis, only the rotor pairs 0/1 or 2/3 need to be adjusted. If the rotor pair 2/3 turns faster, a higher vertical force is applied, causing the UAV to turn around the pitch angle and to fly forward. This behavior can be seen in Figure 3.3. Is the speed of the mentioned rotors declined, then the agent turns in the opposite direction and flies backward. Adjusting the speed of any rotor has

an effect on the vertical speed of the quadrotor. Assuming setting a speed in x-direction, this turns the UAV around its pitch angle and affects the x- and the z-speed of the robot.

The influence of a particular vertical speed-setting u, corresponding to certain angles and speeds are defined by:

$$\begin{aligned} m\ddot{x} &= -u\sin\theta, \\ m\ddot{y} &= u\cos\theta\sin\phi, \\ m\ddot{z} &= u\cos\theta\cos\phi - mg, \end{aligned} \tag{3.4}$$

where m is the mass of the UAV and $\ddot{x}, \ddot{y}, \ddot{z}$ are the accelerations in the corresponding direction. These equations can be converted, then stating what force results in which correlated angle. If the translational accelerations, $\ddot{x}$ and $\ddot{y}$, are given, then that is:

$$\begin{aligned} \theta &= \arcsin\left(\frac{m\ddot{x}}{-u}\right), \ \ and \\ \phi &= \arcsin\left(\frac{m\ddot{y}}{-u\cos\theta}\right). \end{aligned} \tag{3.5}$$

In the simulation the x-speed can be specified by AURIS. It internally is transformed into the appropriate pitch angle, the underlying Unreal Script class (developed in this thesis) controls this. The robot turns around this angle, resulting in a forward movement. If the z-speed is low, that means, the UAV is barley holding its height, setting a x-speed results in forward but also downward motion. The control-program must accommodate to this situation (which AURIS does, as described in Section 4.3).

Rotation around a yaw angle is obtained by alternately increasing and decreasing the speed of the even and odd rotor pairs. This is done while keeping the overall thrust constant. The yaw turning-speed v_ψ can be adjusted. More elaborate deductions about the physics behind quadrotors are state here [5].

To maneuver in y-direction two different approaches are possible. First the speed of the rotor pair 0/3 or 1/2 can be adapted. Alternatively, the quadrotor can turn around its z-axis, i.e., its yaw (ψ) angle, and then fly straight. After reaching the desired coordinates it turns back around ψ, ultimately being in the exact location, and facing the identical

position as in the former approach. The decision to use either procedure lies at the programmer of the agent-control program. In Section 4.2 the currently used approach is explained.

The proposed quadrotor model enables tests of flight scenarios and autonomy. The physical attributes introduced by the Unreal Engine mimic the real world realistically. With the restricted flight control easy traversal through different environments is possible. In a newer revision these restrictions will probably be reduced. Nevertheless, for test scenarios proposed, i.e., way point finding, search missions, basic swarming the implemented system is adequate.

3.2 Building Sensors

USARSim has a variety of predefined sensors, e.g., Range Scanners, GPS Sensors, and Cameras. One or more of any sensor type can be added to any given Robot. The sensor's energy intake and its weight can be specified, which has an immediate impact on the battery charge of the agent. Intuitively, more attached sensors lead to increased energy consumption, due to amplified battery drain, and to the higher weight of the quadrotor, that needs to be overcome by increased torque.

For test reasons all of those delimiters can be turned off. The battery can also be set to unlimited so long-term tests that would be impossible in the real world can be done. New sensors can be developed as well, for example, a Smoke Detector, a Sound Scanner and a modified Range Scanner have been developed over the course of this thesis, their design and setup is explained in the next section. For the sensor setup, conveniently, an initialization file has been created, here base-values for the scanners can be assigned, i.e, resolution, range, noise or other thresholds. This file also handles the number and type of sensor that get mounted to each robot. No Unreal Script or any further programming language skill is necessary to attach sensors to robots, this makes USARSim interesting for people that want to do some quick tests.

A sensor, speaking in Unreal terms, is an item, i.e, it extends the "Item" Class. This class is used to register an object to engine. That means, it specifies a name not a certain behavior. For scanners the base-class is the "Sensor" Class, it is a child of the "Item" Class. Here the default values are set and imported from the main initialization file mentioned before. The role an individual sensor has, is defined in its own class.

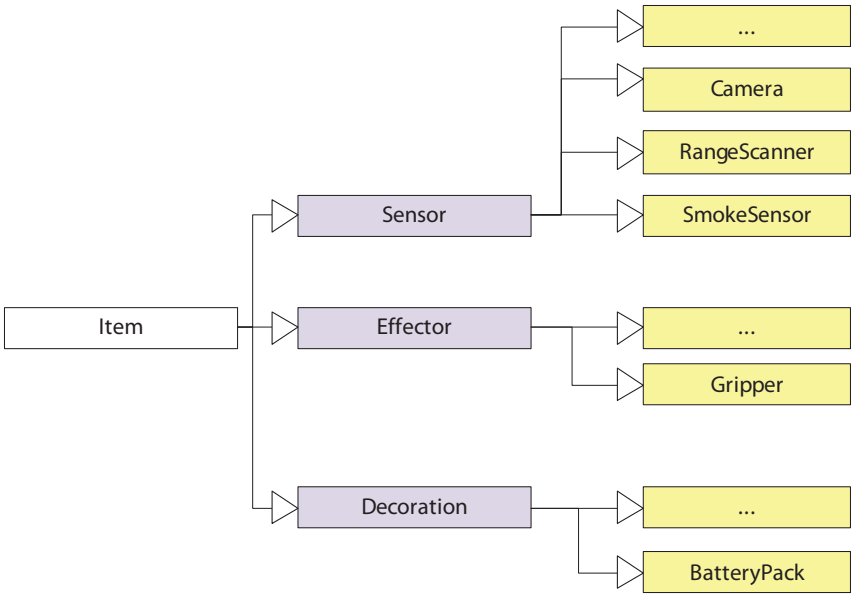


Figure 3.4: Basic USARSim "Sensor"-, "Effector"-, and "Decoration"-Class design

Sensors, effectors, and decorations are all based on the "Item" Class. Figure 3.4 shows their inheritance. These three classes are used to enhance the robots appearance and its utilization. Effectors are moving objects that cannot exist on their own, for example, a hand part of a robot-arm. The hand or each finger would be an effector. Decorations are used to complement the demonstration.

3.2.1 Developed Sensors

Acoustic Sensors An acoustic scanner is developed to detect simulated human voices, or any other source of sound. In a search and rescue mission it might be of great assistance to find injured or buried victims. For test reason, all quadrotors send out a distinct sound signal. It is used to locate the specific position among each UAV. To simulate swarm behavior this approach was utilized, as elaborated in Section 5.3.

The traveling of sound as waves is not implemented in the Unreal Engine. The developed acoustic sensor simulates the reduction of signal strength over the traveled distance. Any sound that shall be detectable by this scanner needs to be specified in the development of the sensor. Focusing on the sound each UAV sends out, from this moment onward it is called a beep. The engine notifies the sensor of the beep, its position and the

distance d. The sensor uses d to calculate the actual loudness L in dB using this formula found in [38]:

$$L = L_{max} - 20\log(d)\frac{1}{\log(10)}, \tag{3.6}$$

where L_{max} is the loudness of the sound when it is emitted. A "hearing" threshold L_c can be specified, should the loudness of the received signal be below, it is considered not distinguishable. In the real world it is also referred to a to low Signal-Noise-Ratio. With $L_{max} = 60dB$ and $L_c = 20dB$ a realistic model of actual sound damping is found (see Figure 3.5).

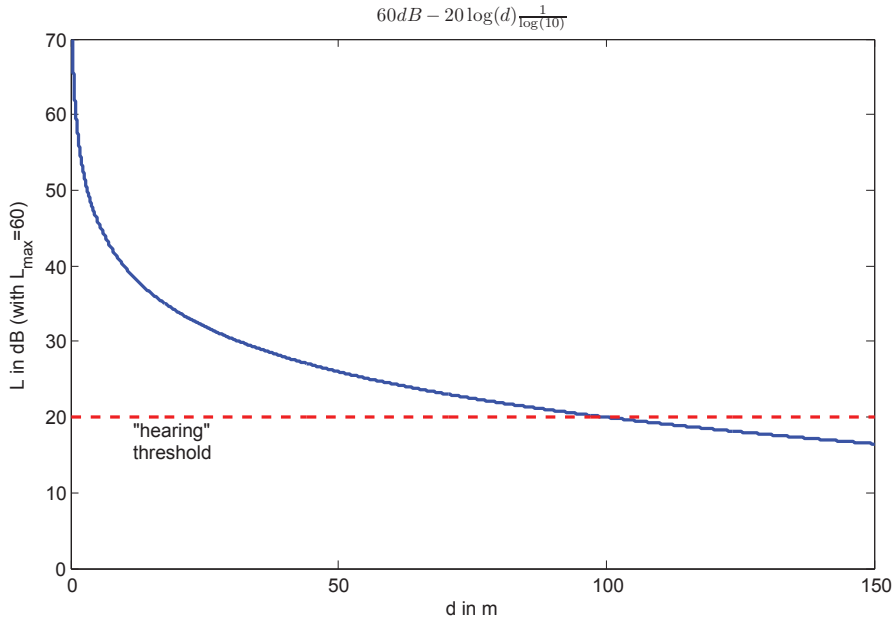


Figure 3.5: Acoustic sensor distance to volume

The proposed damping function works if no object is between sender and receiver. Is that not the case different techniques need be applied. With the *Trace*-function of the Unreal Engine objects in the line of sight can be detected. One approach is to double the distance d when there is a wall between the UAVs. This idea is implemented and tested.

If two quadrotors are in two adjacent rooms, that means, completely separated by a wall, this approach make sense and the results are promising. Although if there is only a corner

of a wall, or any other minor object, detected by the *Trace*-Function, doubling the distance is not reasonable. The damping in this case is to strong.

With this concept there are lots of errors if the quadrotors are used in town like environments. The detected loudness of the beep increases and then decreases abruptly whenever a part of a wall is shortly between two agents, even though in the real world, the sound would not have been distorted significantly. Therefore, this part of the simulated sensor is unrealistic.

Extensive work and calculating power are necessary to build a model which can actually measure the area of the walls blocking the sound, their density, and their influence on the sound travel. To this point, it is considered that the beeps are only needed in open environments, where sound is not distorted significantly. For the given purpose the sound scanner generates sufficient behavior.

After calculating the loudness of the beep, its direction is calculated. The UE has already a set of basic functions implemented. Rotations, normalization, and other vector operations are available. Obtaining the global position of the sound source is archived by a rotation of the location vector.

Quaternions are internally used to specify the location and rotation of a vector as described in Section 2.2.3. The local direction of the sound is

$$\overrightarrow{p_l} = \overrightarrow{p_s} - \overrightarrow{p_m}, \tag{3.7}$$

with the location of the sound $\overrightarrow{p_s}$ and the position of the sensor that hears the sound $\overrightarrow{p_m}$. $\overrightarrow{p_l}$ is now transformed internally into a global direction $\overrightarrow{p_g}$, with the given functions:

$$\begin{aligned} \overrightarrow{q} &= f_r(\overrightarrow{\sigma_m}) && \leftrightarrow QuatFromRotator(), \\ \overrightarrow{q}_{-1} &= f_{inv}(\overrightarrow{q}) && \leftrightarrow QuatInvert(), \\ \tilde{\overrightarrow{p_g}} &= f_{rv}(\overrightarrow{q}_{-1}, \overrightarrow{p_l}) && \leftrightarrow QuatRotateVector(), \\ \overrightarrow{p_g} &= f_n(\tilde{\overrightarrow{p_g}}) && \leftrightarrow Normal(), \end{aligned} \tag{3.8}$$

where $\overrightarrow{\sigma_m}$ is the current rotator, i.e., rotation vector $(\phi,\theta,\psi)^T$ of the sound receiver.

The built sensor is different to a sensor in the real world. It actually has all the information, but it distorts it and returns a degraded set of information. The data sent to the client is

the direction $\overrightarrow{p_g}$, the volume L, and the duration of the sound t_s, hence, the information provided by this sensor. Furthermore, these values can, alike other sensors, be distorted by virtually any probability distribution.

For the acoustic sensor a Gaussian distribution is used to alter the sensor data. For the loudness a higher variance is applied than for the direction, because in the real world, different sounds in an area interfere with the distinction of a certain signal. Subjectively, this seems to be the right choice, because for the human ear it is easier to determine the direction of a sound rather than its distance.

The Unreal Engine limits the function of this sensor, because it is difficult to determine witch objects need to be surpassed by a sound signal on its way between sender and receiver. For the given purpose, i.e., using the beep in an outside environment to distinguish the location of different UAVs the current approach is satisfactory.

Smoke Sensor The smoke scanner can neither illustrate the functionality nor the robustness of a real smoke sensor. The currently implemented version has no actual coupling to reality, and it is merely used to test different robot behaviors.

It uses the *Trace*-Function mentioned in the prior paragraph. This function returns the type of mesh it has encountered in a straight line, with a given range, and in a certain direction. A special “Smoke” Mesh is used [42] to add fire and smoke to a test environment. When the *Trace*-Function detects the “Smoke”-Mesh the sensor is activated and returns the location of the fire. The sensor checks an area in front of the quadrotor in a cone shape. In Figure 3.6 the smoke, the UAV, and the basic detection area of the smoke sensor in a running simulation environment can be seen. The smaller picture-in-picture is the view of the quadrotor camera. The sensor cone is invisible in a running simulation.

The cone size, and the range can be adjusted. In addition, false outputs can be produced to make the scanner more realistic. With a camera an optical smoke sensor can possibly be developed. First approaches have been undertaken by the developers of the smoke mesh as can be seen in Appendix A.3.

Figure 3.6: Quadrotor with smoke detector and smoke

4 Controlling a Quadrotor

Autonomous actions need to be backed up by sensor data of the environment. Without these informations a robot can not prevail in an unknown surrounding. This chapter investigates the processing of gathered sensor data, their evaluation, and the actions followed by the extracted informations.

The control program introducing autonomous behavior is AURIS. In which manner it communicates with the steered robot is explained in the upcoming section as well. In addition, the processing of information collected by the robot and its influence to the course of action are described. Protocols to safeguard the quadrotor at all times are introduced.

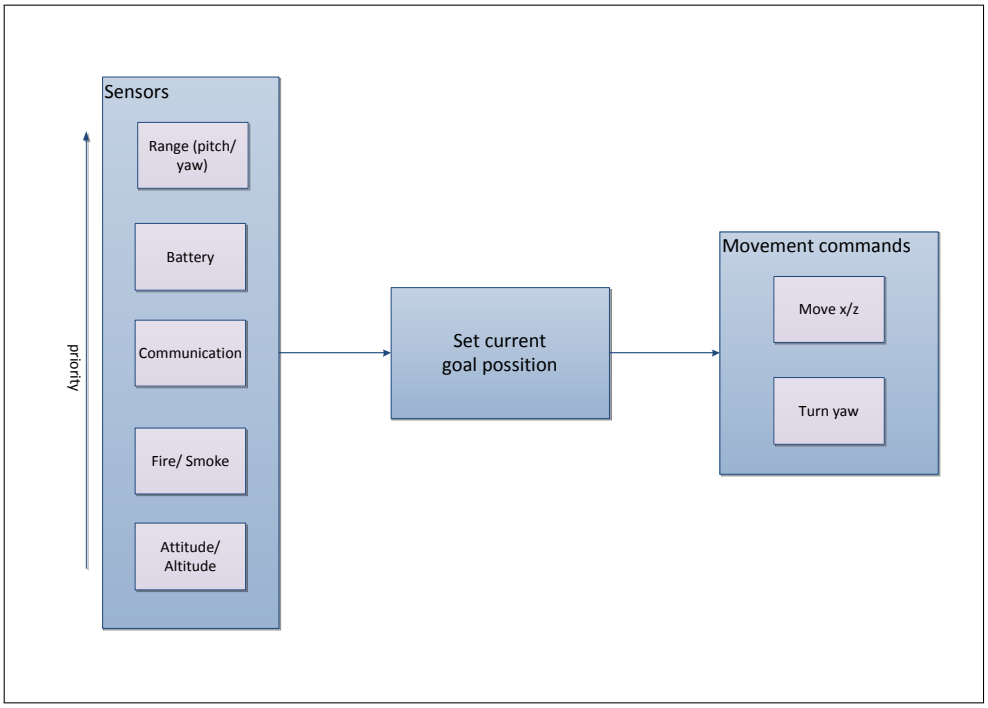


Figure 4.1: Evaluated sensor data leads to movement

This chapter shows in detail how the guarding systems, range detection in yaw and pitch direction, and the power management work. The difficulty of flight planing and of developing knowledge of the surrounding, are discused as well. Figure 4.1 shows the basic sensors and there prioritized impact on the decision marking process. Only if no obstacle needs to be surpassed and the battery is charged to a certain extent, a mission can be accomplished.

4.1 Communication

Sensors and robots communicate with AURIS through a client server setup. Hence, more than one robot can be spawned in the same simulation environment. All data available in the UE can also be sent over a TCP connection. AURIS is an external program written in C++. It has basic commands, which enable it to control a robot in the simulated environment. Nevertheless, it would also be possible to use the same program (possibly with altered I/O) to navigate a quadrotor in the real world.

The Unreal Engine loads the surrounding, i.e., a map, as soon as the simulation starts. Here different static meshes are placed to introduce a certain ambiance. The simulated domain can be adjusted as needed, it can a town, or an open field, or even another planet. After AURIS sends out an initialize command, the specified robot spawns, that means, it gets added to the map.

All sensors that belong to the robot have been specified prior to the start of USARSim. After the start, all sensor data is broadcast to the TCP link between AURIS and USARSim.

Four different message types are used to communicate:

- “Info” (NFO), for initialization purposes only
- “Sensor” (SEN), all sensor data that is sent from a sensor to the control program
- “Status” (STA), messages that inform about location and different situations concerning the Unreal Engine, and the spawned robots

  STA messages are not used to control the robot. They are background data for debug purposes only. Using this data in a simulation can not be allowed, because this data would not be present in the real world.

- "DRIVE", the only command the control program can send out, besides initialization

  With "DRIVE" the movement of the robot is specified. For every distinct robot-type there might be a different variation of the DRIVE command, e.g., a driving robot will not have "DRIVE{zspeed}", because it cannot move in z-direction.

Due to the message protocol, the introduced design can be converted to control an actual UAV. The sensor messages need to be modified to be read by AURIS, and the "DRIVE" command has to be handled by the UAV, to make this possible.

4.1.1 Inter-UAV Communication

Communication between robots in a simulation was not part of USARSim. This feature is built over the course of the thesis. A wireless networking adapter is developed, it attaches to robots in the same way sensors do.

The communication mimics a reliable transmission protocol, that means, if a message is transfered to a certain agent, the sender can be sure that the data is delivered (similar to TCP). The communication can only take place if UAVs are in the range of the wireless networking adapter. For test purposes the maximum range of the adapter can be specified before the simulation starts. Damping, or blocking of the communication signal is not provided. Nevertheless, the possibility to implement it is given.

Internally three different Unreal Script classes enable this behavior. AURIS sends its message to the Unreal Game Server, as it is done for "DRIVE" commands. The data handler is called, which checks for other wireless networking modules in range. Either a message is broadcast to all communication adapters, or to a specified ID.

With this setup complex swarms of distributed agents can be established. Chapter 5 introduces strategies for setup and control of such groups of UAVs. To implement the communication link between AURIS and a real world quadrotor, respectively, between different UAVs an interface has to be proposed to establish the communication between agents first.

4.2 Collision Avoidance

Assuming the quadrotor design is finished, a mesh is designed and the physical behavior is defined. That means, the UAV is ready to perform tasks. Since the ultimate goal is autonomous behavior, it is important that the agent can navigate on its own. Therefore, collision avoidance functions are essential. Even if a robot can go from one point to another, there is no use for it if it can not surpass the first obstacle it encounters.

In the current implementation the UAV can only fly in positive x-direction, in z-direction and turn around the z-axis. The movement looks therefore constricted, this makes it easier to detect obstacles and to control the quadrotor motion. With this approach only two sensors are used to detect objects blocking the way, these are, range scanners around the z-axis and the y-axis (see Figure 4.2).

The range scanners sampling rate can be adjusted to simulate its real world counterpart. A Hokuyo UTM-30LX laser range finder, for example, has a sampling rate of 40Hz. To safeguard a quadrotor completely a whole sphere of laser scanners would be necessary. With two attached scanners, and the restricted maneuverability, the UAV can only detect obstacles that can be harmful because of its own motion. Another moving entity, e.g., a bird, can still hit the agent from an angle that is not scanned. The likelihood of such an event is minuscule, therefore, the current scanner setup is considered sufficient. Furthermore, this design is a trade-off between power consumption, especially due to the weight[1] of additional scanners, and obstacle detection.

Each laser scanner scan at N (=16) positions around the perimeter, depicted as dotted lines in Figure 4.2 . As mentioned, the UAV can only fly forward or upward. Therefore, the reaction to an obstacle found directly in flight direction needs to be more severe than to an object in the outer rim. Accordingly, two weighting functions were developed, one for the distance d of an object, and the other one for its radial position ψ compared to the quadrotor. In addition, the bigger the object the more sensor nodes respond, and that also increases the measurement of severity.

The weight w_f is calculated as

$$w_f = \sum_{i}^{N} \left(f_\psi(\psi) f_d(d) \right) \; and \tag{4.1}$$

[1] The UTM-30LX laser range scanner [47] used by one of the MITs quadrotors [1] has a weight of 370g. The lightest laser scanners still weighting approximately 150g. With quadrotor payloads lower than 1kg adding more sensors needs to be well-considered.

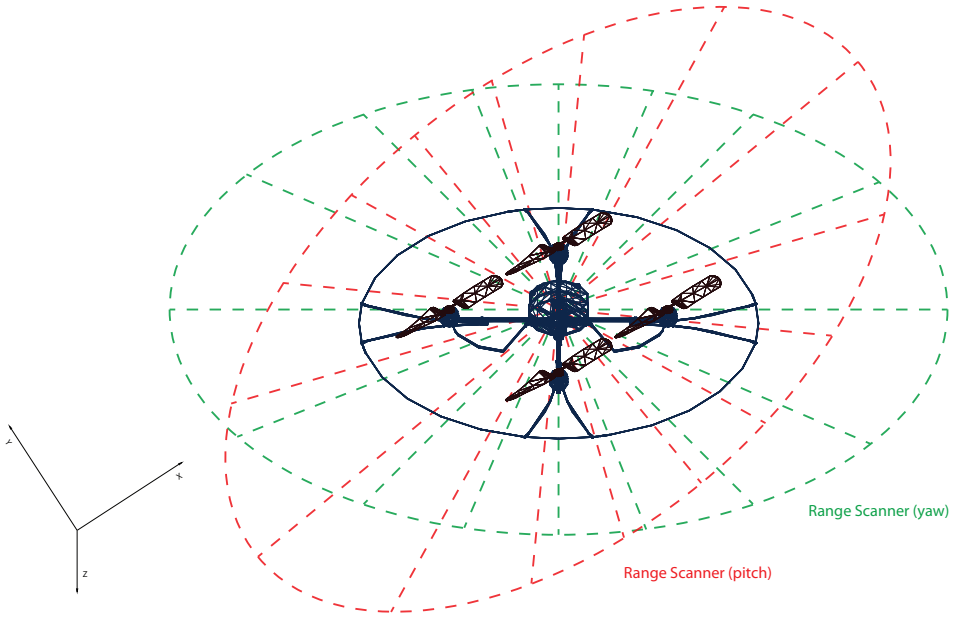


Figure 4.2: Quadrotor with yaw and pitch range scanner

where

$$f_\psi(\psi) = \exp(-\psi_i^2), \tag{4.2}$$

$$f_d(d) = \left(\frac{d_{Max} - d}{d_{Max} - d_{Min}}\right). \tag{4.3}$$

A plot of the multiplied weighting functions can be seen in Figure 4.3, it shows how close, frontal obstacles have a bigger impact on the generated weight, than further away or outside objects. Even though the maximum value of w_f might exceed *1*, it is treated the same. The maximum distance d_{max} resembles the maximal range of the scanner. The critical value of d_{min} specifies the distance, to an obstacle, which leads to a complete stop of the UAV, because $f_d(d = d_{min}) = 1$. For the range scanner around the pitch angle, the same weighting functions are used.

After an obstacle is detected the agent reacts to it according to the generated weight. If the quadrotor tries to get to a certain location, the course has to be altered after an object

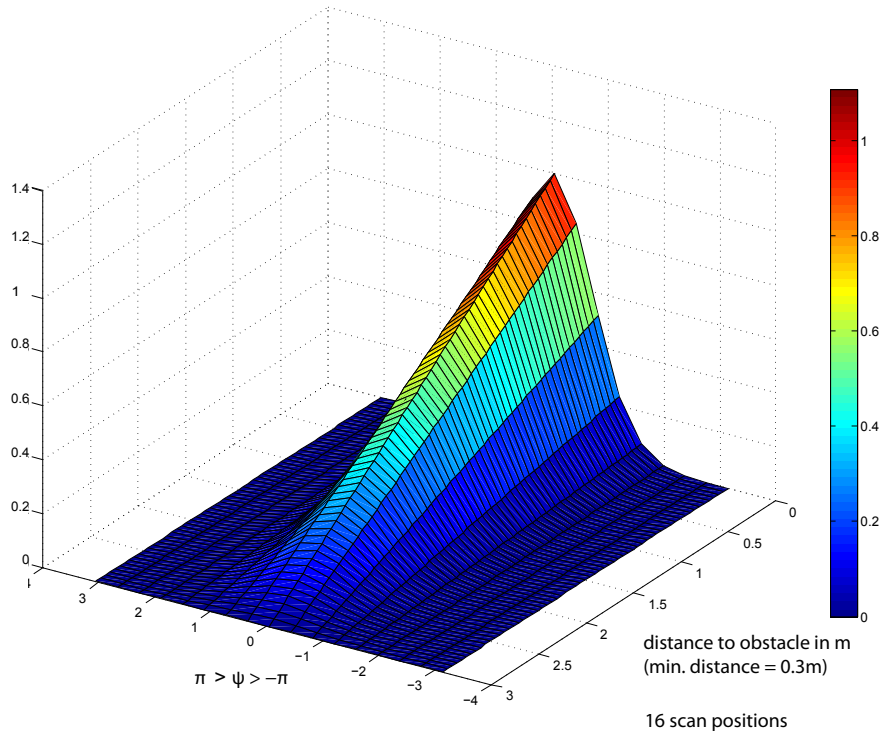


Figure 4.3: Multiplication of weighting functions to show area of severity

blocking the way has been found. When the pitch-range scanner finds an obstacle the UAV changes its altitude to avoid it. If the yaw-range scanner gets activated, the course is altered in x- and y-direction. Should both scanners get activated then a combined direction change takes place.

Figure 4.4 shows the behavior of the yaw-range scanner if an obstacle is detected on the right front. First w_f is computed, then a new direction is calculated $\psi_a = f_w(w_f)$ away from the obstacle and from the original route, to a unique temporary goal in a given distance. If another obstacle is detected, the temporary goal will be recalculated. After reaching it, the quadrotor continues to the main goal. Should the pitch-range-sensor signal, then the severity is calculated. The UAV stops its advance in x-direction and adjust its altitude according to the calculated weight. After reaching the desired height the forward movement continues. When the UAV is landing this rule is disabled, right after getting to the correct x- and y-coordinates.

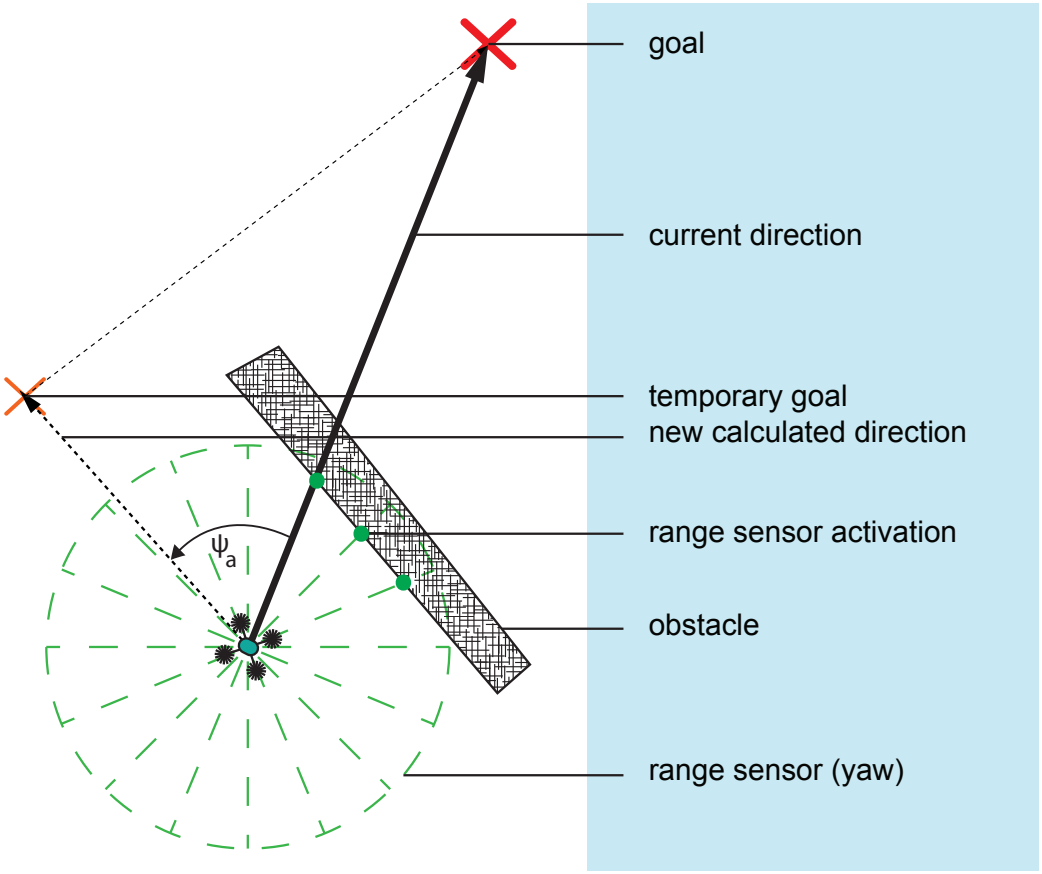


Figure 4.4: Collision avoidance operation

All simulations conduced have shown that this approach leads to satisfying results without extensive calculation or way planing. Basically, any obstacle can be surpassed. If the final destination cannot be reached due to a scenic characteristic, e.g., the goal-position is inside a building without a way in, then the algorithm will not come to an end. Setting the quadrotor on a circular course around the goal position, until the battery interrupt (see Section 4.5) commands it to fly back to the base station.

This collision avoidance control is designed for outside use. It safely guides the UAV away from trees, rocks, or buildings. Inside, there needs to be an extended control algorithm, which searches more thoroughly for a position to surpass an obstacle. Furthermore, it needs to be concluded that the goal position cannot be reached.

4.3 Altitude Control

When the UAV advances in any direction, it turns around the roll or pitch angle. Therefore, rotation its force-vector, away from a straight upward direction, i.e., converting uplift force in forward thrust. That means, the z-speed of the quadrotor changes. The altitude control keeps the height on a constant level. If the collision avoidance control changes the z-position of the UVA, then altitude control does not interfere.

To accomplish the controller, the current height is estimated first. In the real world, it is not an ordinary task to determine the altitude of an UAV correctly. Different sensor data can be combined to improve the final output, e.g., barometric data can be supported by the quadrotors vertical acceleration (see. [11]). This behavior can be simulated by the UE. For now a pseudo sensor is used, which returns adequate data. This data can be distorted by white noise with a given variance, but it does not behave like an actual barometric sensor.

To change the current altitude, the speed in z-direction is added (or subtracted), i.e., turning the rotors faster (slower). This is done to all rotors simultaneously, to keep the UAV steady. To land the quadrotor it is important to lower the speed continuously until the downward speed is fast enough. Decreasing the altitude too cautiously lengthens the descend unnecessarily, but declining to hastily might cause a crash possibly damaging the UAV.

Due to accurate sensor data, height control was not a focus area. Considering that the significant part of simulations done were in an outside environment, fast responses to vertical direction changes have not been developed. If a quadrotor is supposed to work in a complex environment, especially inside a building over a number of floors, more powerful algorithms need to be developed.

4.4 Mapping

In a simulation the UAV could get full knowledge of the map. In the contrary, it is possible to add the quadrotor into the environment without any a priori knowledge. The latter premise is focused on in this thesis and in all experiments, because it relates to the real world.

4.4.1 Simultaneous Localization and Mapping

Every robot in the simulation has sensors to trace its surrounding. One of the goals of autonomous robots is to navigate in an unknown environment and to localize themselves in that area. Due to noise of range sensors, GPS, and all other sensors used for position estimation, the internal map created by a robot can never be one hundred percent accurate. Therefore, methods have been developed to overcome shortcomings in detail and precision. Simultaneous Localization and Mapping (SLAM) is such an approach. Historically the construction of environment models from moving sonar platforms, in *photogrammetry* [13] and computer vision [26], are predecessor of SLAM, without covering the field of robotics. Preliminaries of the SLAM approach are explained in this section, a comprehensive illustration is given by S. Thrun [25].

For a driving robot, the world is basically two-dimensional. Surpassing obstacles by simply driving over them is virtually never possible. Obstacles are usually walls, trees, or whole buildings, hence localizing and mapping are done in two dimensional space. For the quadrotor, it is usually possible to fly over objects, increasing the complexity of the mapping. Especially, because the search area grows in a third dimension. On a search mission, for example, the maximum altitude needs to be restricted, to ensure the UAV gets done with its task in a reasonable time. On the upside, growing sensor ranges increase the productivity, but also raise the computational complexity.

Changing environments, e.g, people walking by, objects get moved by wind, or a spreading fire, add to the map building difficulty. Locations that are visited are marked, e.g., *empty* or *obstacle*, depending on the sensor reading. There has to be an update process of all measured points as soon as new sensor data arrives. The SLAM approach considers all those properties, with five steps: Landmark extraction, data association, state estimation, state updating and landmark updating.

Landmark Extraction Different scanners can be chosen to retrieve the data for the landmark extraction. Either laser range scanners, sonars or cameras can be used. In a simulation where costs do not matter, a laser scanner will be attached, due to its accuracy. Nevertheless, in the real world, a sonar is much cheaper. In addition, it is helpful in an underwater scenario, where the range of a laser scanner is more restricted.

Cameras could be used, too, but the computational complexity, to calculate the optical flow of every pixel of a captured picture, is relatively high. Therefore, the camera drains more power from the UAV. Furthermore, the processor used might not be able to handle all necessary calculations in real time. Although, the range of a camera in an outside envi-

ronment is greater than a laser scanner's. All three scanner types are already implemented in USARSim, and can be chosen for different scenarios.

Landmarks need to be re-observable, hence, they need to be distinguishable from each other, as well as stationary. A moving landmark increases the complexity to update the location of an agent precisely, because location and speed can never be observed accurately. The distinguishability is a necessity, because it is unclear if a certain landmark is unique in a given environment. Different methods have been created to find landmarks, e.g., spike landmarks, or RANSAC [24].

RANSAC is used in different fields when parameters need to be estimated from a dataset. It is an iterative method, with the number of iterations used the quality of the result increases. For landmark extraction, RANSAC is used to fit lines on the measured data from the laser scanner. If lines are found, they are used as landmarks. This is especially useful in indoor environments, hence, walls resemble straight lines.

Spike landmark extraction looks for extrema in laser scan readings. A tree in an open field can be represented as a spike in the graphical representation of a sensor reading, as can be seen in Figure 4.5.

The spike landmark approach should not be used by itself [17], since the failure rate is very high, e.g., a person walking trough a scene will be classified as a landmark. As the person moves, the landmark can not be associated correctly, increasing the possibility of erroneous map building.

RANSAC is much more error prone, if certain parameters (minimum line length) are set correctly. Spike landmarks can only be beneficial in combination with other landmark extraction protocols. To enhance the extraction even further scan matching [18] can be used. Scan matching is basically a test of the shape. If the

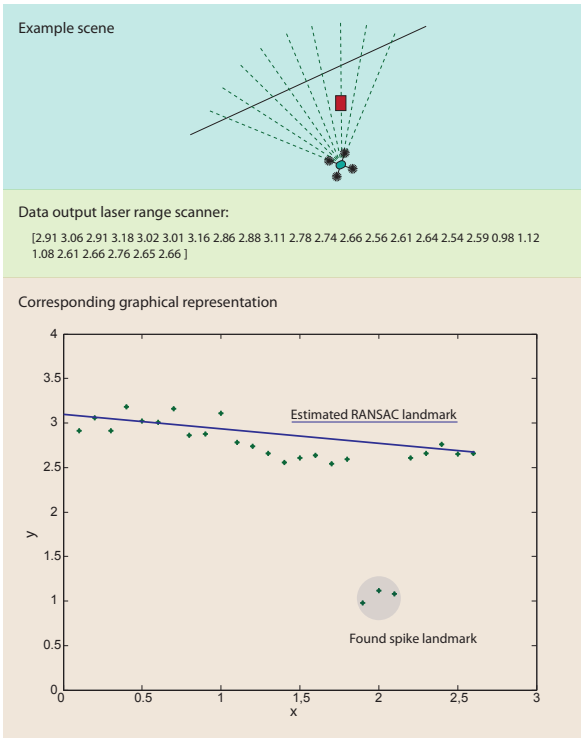


Figure 4.5: RANSAC and spike landmark extraction

landmark found has a different (to a certain extent) shape, than the landmark it is supposed to be associated with, it is rejected.

Data Association Landmarks in concurrent scans are evaluated, if they are identical or if a new landmark has been found. A number of errors can lead to wrong detection. If a line (RANSAC) can not be re-observed in every time step, there needs to be an algorithm to add them together. After a landmark has been found it might not be re-observable, then, after a number of iterations, it has to be deleted. The biggest mistake is associating unrelated landmarks, this completely destroys the understanding of the scene.

Every landmark needs to be observed a number of times (N) to be considered to be used for SLAM. Should it be accepted it is called key. For every data reading the extracted landmarks are compared to the keys. With a certain threshold, the landmark is either *new*, or interpreted as re-observed. This so called nearest-neighbor approach is also used in other visual related areas. For distance calculations several concepts can be used, typical are Euclidean or Mahalanobis (see Appendix A) distances. Data association with a chosen threshold λ is done by:

$$\zeta_k^T Q_k^{-1} \zeta_k \leq \lambda, \tag{4.4}$$

where ζ_i the innovation, and Q_i the corresponding covariance matrix from the Kalman Filter used to merge the existing data. The Kalman Filter handles the last three steps of the SLAM approach. The final outcome is supposed to be the agent's position, as close to the real location as possible. An Extended Kalman Filter (EKF) is usually used for this purpose. In addition to a regular Kalman Filter, the EKF can also handle nonlinearities in the process model, the observation model, or both. The following paragraphs will continue to explain the usage of the Kalman Filter, for actually simulated basic map-building designs.

4.4.2 Basic Mapping with Error Correction

SLAM still is an area in robotics that holds challenges, therefore its implementation is complex and extensive. Because basic flying test, swarms, and searches for fire where the main focus points of this thesis, a complete SLAM approach is not implemented. Although, a less sophisticated mapping concept is carried out.

A standard stimulation to evaluate the flight behavior of the quadrotor, is that a goals is set, and the agent has to reach it autonomously. This elemental test involves one or more quadrotors. The second type of test is checking areas for fires. After the flames or smoke is found the UAV has to find its way back.

No a priori map knowledge is given. The quadrotor "memorizes" the areas it flies through, where it finds fire, and any obstacle it encounters. The object is saved as a dot at the position on the map where the range scanner got initiated. Figure 4.6 shows a map drawn by the agent on a simulated mission, it had to fly to certain way points without colliding with walls. Note that, the UAV cannot distinguish between certain obstacles, because it has only range scanners attached. This approach is sufficient to get a basic understanding of the map and of obstacles which have to be surpassed. It is merely used to retrace the quadrotor's steps.

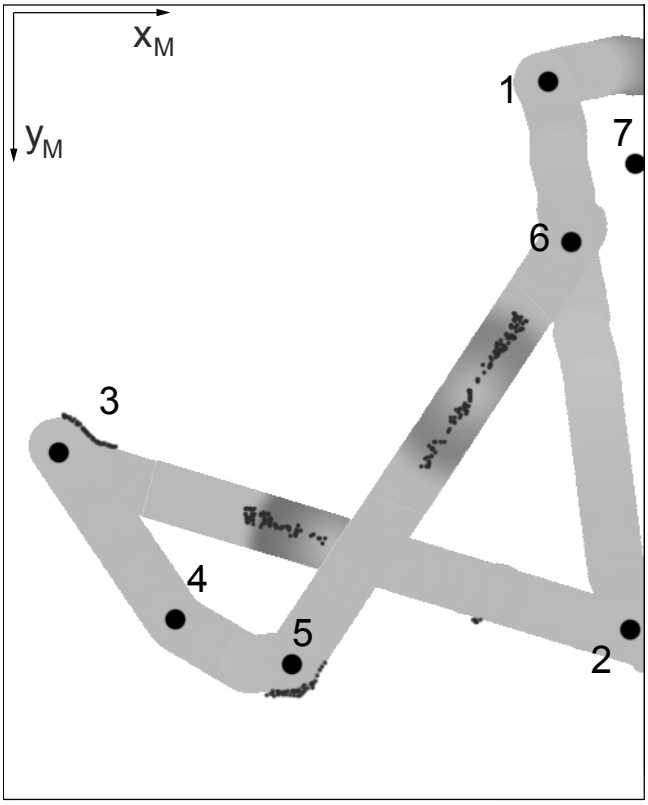


Figure 4.6: Basic map drawn by the UAV. The small dots represent positions where the range scanner (pitch or yaw) was activated. The gray area represents the already scanned environment, where the different shadings symbolize to the altitude of the robot in that point in time.

Evaluating Sensor Data In "Underwater SLAM for Structured Environments Using an Imaging Sonar" [23], different methods have been introduced to solve the SLAM problem for an autonomous underwater vehicle (AUV). A voting algorithm, an EKF approach, a hybrid of EKF and voting, and an adapted voting scheme was introduced to merge and correct sensor data efficiently. Some ideas can be adopted and integrated for AURIS. An underwater area relates partly to an aerial one, due to its three dimensional complexity.

In this thesis, a Kalman Filter is used as well to filter and correct measured data. The global, instead of the local attitude and location is traced filter. The Kalman Filters mathematical basics are explained in 2.3.2. With the current location data $(x, y, z)^T$ from the simulated position sensor and the speeds (v) respectively, the state vector is:

$$\overrightarrow{x_k} = \begin{bmatrix} x \\ y \\ z \\ v_x \\ v_y \\ v_z \end{bmatrix}, \quad A = \begin{bmatrix} 0 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{4.5}$$

The filter is initialized with date of the first observation, hence the first measurement. The observation is followed by the prediction. The prediction step projects the estimates of the state vector and its error covariances ahead in time. The measurement error covariance and the model error covariance matrix are defined before the filter is started. In a simulation the expected error is known, therefore the correct values for the covariance matrices can be set. Nevertheless, simulations are conducted with imperfect settings. The Kalman Filter has proven to be reliable in those test as long as the errors reflect possible real world settings.

With the measurement matrix

$$H = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.6}$$

the new sensor data $(\overrightarrow{x_k} = (x\ y\ z)^T)$ is taken into account and in the next step, the system model is updated.

Furthermore, extreme errors in measurements are considered. If a reading appears to deviate considerably from the Kalman Filter prediction, it is not used for updates. After such

an outlier was found, the possibility to find another one is increased. If the Euclidean distance of the measured position, $(x_m, y_m, z_m)^T$, and the estimated position, $(x_p, y_p, z_p)^T$, exceeds the threshold:

$$|(x_p, y_p, z_p)^T - (x_m, y_m, z_m)^T| > \varrho + 2^L * v_{max} * \Delta t, \tag{4.7}$$

where ϱ is the maximal tolerable error, v_{max} the maximum speed of the quadrotors, L is the number of concurrently found outliers, and Δt the time difference between two prediction steps.

Assuming that a quadrotor gets hit by a gust of wind, changing its current location rapidly. The measured position values are considered to be outliers. After dumping a couple of values the possibility of another error increases to the point that a measurement is considered correct again. This data is now added to the Kalman estimation, which continues. Chapter 6.1.2 states tests of the implemented Kalman Filter, for comparison with and without outlier reduction.

4.5 Power Management

One of the worst things that can happen in the real world, is that the quadrotor crashes, or gets lost. When the battery is empty the motors stop and the UAV falls from the sky. For an autonomous agent, that does not constantly report its position, this is especially problematic, because the agent might never be retrieved. Accordingly, it is of vital importance that the quadrotor returns to the base station before it is out of power.

An algorithm is derived to implicate this behavior. It interrupts any task the quadrotor is conducting and set course to the home base. With knowledge of the surrounding, recorded while exploring the perimeter, it is possible to calculate the way back. As long as the map is not changing drastically[2]. The agent, internally, computes how long it takes to get back to the base station. This is done for the first time when the battery drops below sixty percent, to conserve computation power. The quadrotor uses already verified areas for its retreat, hence, in the worst case, it flies back exactly the way it came from.

To find the way back the UAV saves the cells it has been to. With a cell, the cube around the sphere of all sensors is described (see Section 5.4 and Figure 5.8). Should a particular cell be reached twice, or a neighboring cell has been visited before, then the list will be amended, i.e., cut off at the current position. Accordingly, the shortest known path back

[2] Fire or earthquakes, for example, could change the surrounding. If this is the case, the current algorithm needs to be augmented.

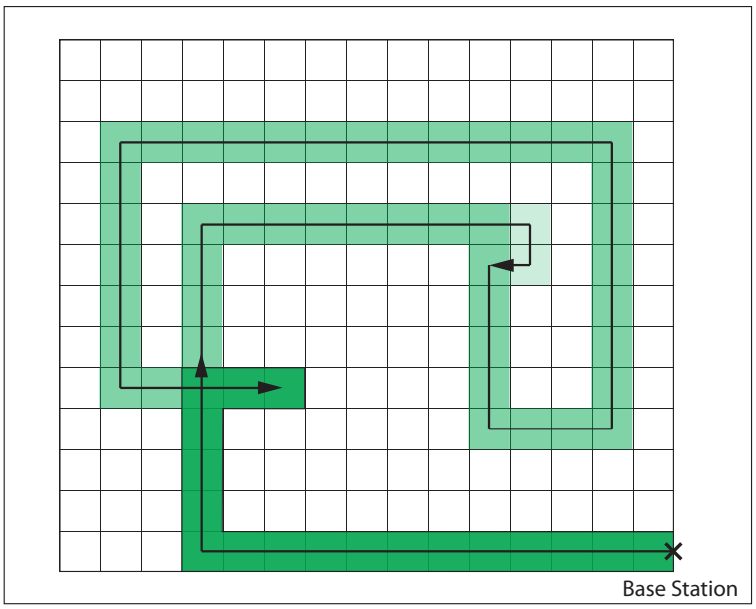


Figure 4.7: UAV saves shortest known path back to base station; light green route will be "forgotten"; black arrow shows traveled path

home is maintained. Figure 4.7 elaborates this approach, in a two dimensional fashion. Even though the UAV traverses a three-dimensional environment, most problems can be considered from a 2D perspective. Nevertheless, the actual path saved internally is a 3D route.

For this algorithm the computational complexity is eligible, because only the list of visited cells must be checked if a new position is reached, the distance to the home base needs to be computed. With a growing area, the complexity of this algorithm increases linearly. The cell size is determined by the sensor range, and the number of cells depend on the work area. Considering that the UAV operates in a limited height. Ignoring the fact that not all cells in an environment ever need to be saved for the way back, (because then they all would have neighbors reducing the way to the diagonal) the size for the list of points would be:

$$C_a = c_x c_y c_z, \tag{4.8}$$

where $c_{x,y,z}$ are the number of cells in each direction. In the worst case scenario, half the cells are blocked, and the others are free to move through. In the 3D worst case, only

three cells in a cell neighborhood can be open to fly through, the others are inaccessible. So three of seven cells need to be saved. That means the biggest list that needs to be stored is:

$$C_a = \frac{3c_x c_y c_z}{7}. \tag{4.9}$$

A map that might actually look like that could be the corridors of a skyscraper, over different floors. There could be special cases where the number of saved points gets even bigger. In a 1x1000x1 cells map, for example, with no obstacle, clearly, all cells are saved. This is due to the reduced cell neighborhood at the sides and corners of a map. Here the neighborhood is three cells big, including the considered cell itself. This cancels down Equation 4.9 to $C_a = 3c_x c_y c_z/3$. Generally, it is assumed that the cell neighborhood consists of seven cells.

When calculating the required battery power to fly the return route, the number of saved cells in the list is added, and then multiplied by a given factor. Moreover, a ten percent reserve for safety reasons is considered in the computation. As a second lifeline the quadrotor lands as soon as only five percent of battery power are left, keeping in mind that the landing process also coast power. The agent lands wherever it is at this moment, rather getting lost than crashing. The remaining power will then be used to send out distress signals. For certain terrains the described approach might still lead to broken quadrotors. Considering an island group that has to be searched. Should the agent powers down its engine, because it determined it cannot make it back base station with the remaining power, it will be landing in the water. Two different ideas are introduced to mitigate this problem.

First, a new cell type, additional to *blocked* and *free*, has to be introduced, e.g., *dangerous*. That means, the UAV does not power down its engines if flying through a cell like that. Algorithms can be introduced to try to avoid all dangerous cells altogether. Second, the power consumption for the return flight is reduced. Only the needed sensors for collision avoidance are not shut off. Because the return route has been evaluated before, it can be considered known, that means scanning it again is unnecessary. With this small power boost the possibility of returning to the base station increases.

5 Multi Robot Scenarios and Swarms

This chapter introduces applications and possible setups for groups of agents. All of these ideas can be implemented in the simulation. It is the main purpose of the upcoming paragraphs to show the diversity and power of UAV swarms and their benefit to different fields of research. First of all two concepts are explained, swarms and emergence.

A swarm is a number of entities that knowingly or unknowingly work together. While working does not necessarily mean to accomplish a task, it could also mean, the collective motion of a number of entities [21]. Usually, swarm behavior is mentioned in the combination with animals, especially fish (schooling), quadrupeds (herding), birds (flocking) or insects. In this thesis, a swarm is a collection of robots, i.e., quadrotors, which are supposed to accomplish something together. The number of robots forming the swarm is not restricted aside from being more than one.

Different approaches for the forming of swarms are discussed, because the level of communication changes the organization structures among the swarm, from a hierarchical set of quadrotors to a decentralized self-organizing system. The later introduces the concept of emergence. A coherent system which evolves from the actions of its members is exhibiting emergence, that is what De Wolf and Holvoet state in their paper on “Emergence Versus Self-organisation” [32]. The self-organization can also be realized by communication among the members of the group, whereas emergence is a process not comprehend by the components of the subsystem.

5.1 Full Disclosure

With the simulation software described in the Chapters 3 and 4 in place, a number of different scenarios can be tested. First it is considered that the quadrotors have a direct communication channel open at all times. So all information necessary for team-setup, way planing, objective control, and time management can constantly be communicated among all team members.

One possible swarm building approach is to inform every entity of the group about the current position, status, and other required factors of each other member. This is called full disclosure.

Communication Methods It is considered that all UAVs possess a wireless-LAN (WLAN) controller. The communication between each group member might not be direct. It is possible that only a couple of quadrotors can correspond with each other at a certain time. That means, the UAVs build a communication chain or circle to make every agent aware of the current setup. This controller can be implemented in the simulation.

The basic characteristics of a real WLAN controller need to be imitated. That means, the controller is faster and more reliable if quadrotors exchanging data are in closer proximity. Figure 5.1 elaborates this setting. To get messages from one outer rim to the other, messages need to be forwarded by all agents in between. The bigger the intersection of the WLAN-range circles, the higher the possibility that a message can be sent and received, because the error probability is lower.

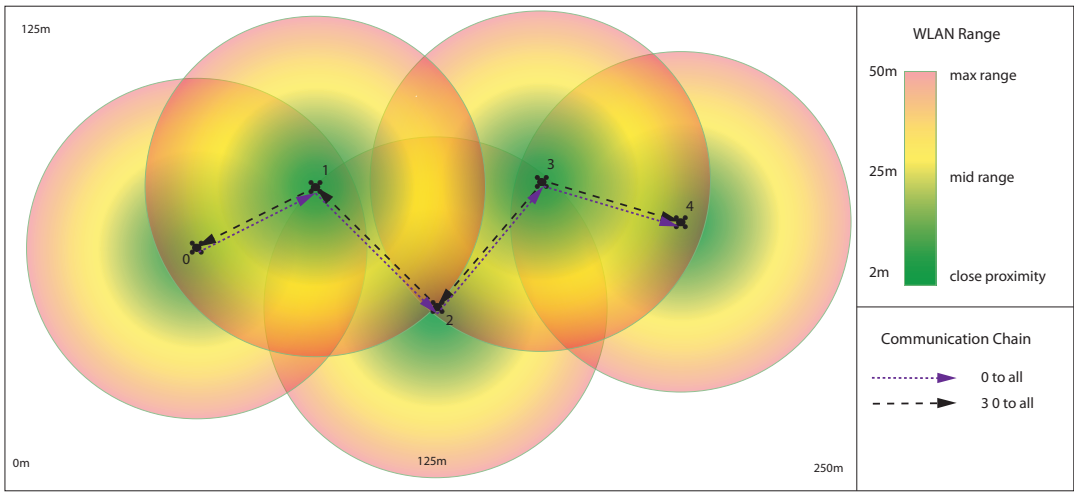


Figure 5.1: Possible communication setup

Other communication methods are also possible. Assuming that each quadrotor has a GPRS [1] module attached. Is the current application-area a town or city, then the range of GPRS is virtually unlimited. All UAVs can continuously, not only exchange data between each other, but also inform the base-station or any other entities (firefighters, police, ambulances, or other robots) of their current status. Furthermore, GPRS uses less energy

[1] When talking about GPRS newer standards (GMS, EDGE, UMTS, or HSDPA) are not excluded.

compared to the requirements of WLAN usage. Moreover, the communication does not require a set of agents, that have to forward messages, anymore. Every quadrotor connects through an antenna tower with each other, that makes communication more reliable. Because, if one UAV in the middle of a communication-chain fails, two agent groups can not exchange data anymore. Whereas, if the same happens with a quadrotor that uses a GPRS module, only one entity looses communication, without effecting the other group members.

Considering an open field, or a sparsely populated or uninhabited area, GPRS is not an option. WLAN should be used for this purpose, and is the proposed way of communication throughout this thesis. Mainly because, the simulation is linked to the actual quadrotor at Hamburg Tech., and on account of fire-monitoring and search scenarios in open environments, which has been focused on. In Figure 5.1 the WLAN range is set to a maximum of 100 meters, even though, current WLAN generations (IEEE 802.11n) are supposed to have a maximum range of over 200 meters (outdoors) [36]. For error prevention indoor environments are always considered, that means, the maximum range is approximately 100 meters. The actual quadrotor at Hamburg Tech also has a range of circa 100 meters, even in outdoor environments. This is mainly due to less advanced equipment.

Communication between all quadrotors at all times can never be guaranteed. There have to be safety protocols, when the WLAN peer-to-peer connection gets lost. The robots need to either continue their current task, with the information available, return to the base station, or wait to pick up a signal in the future. Since the data transmitted between the agents is not limited, the goal coordinates can be broadcast when starting the communication. That means, should a quadrotor get out of WLAN range, it can still fly to the desired location, and possibly rejoin with the others. A variety of protocols have been proposed to deal with these situations, e.g., “Communications Recovery for Multi-Robot Teams” by P. Ulam [28] provides more information.

Formation Either centralized or decentralized hierarchical structures are possible, each type can be subdivided in a number of special applications. The focus lies on the centralized methods with one leader per group. Let the communication be established by a WLAN module (Figure 5.1). The information sent to all other quadrotors is the current position, the status, and perhaps certain sensor data. The status is *following, leaving formation* to recharge battery, or *found something* of interest. Of course other states are possible.

A leader has to be appointed first. Let every quadrotor have a unique natural number η assigned, when they meet (get into communication range of each other) the one with the

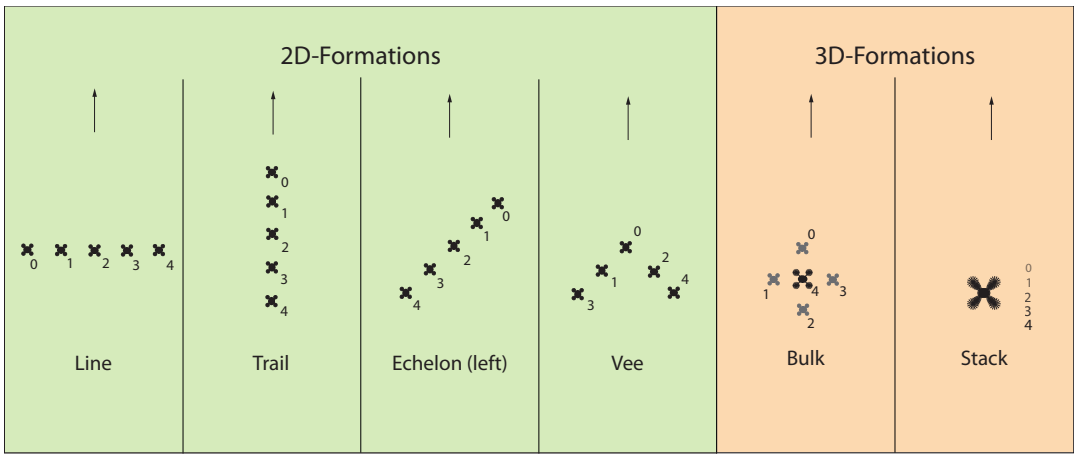


Figure 5.2: Different flight formations

smallest number is the designated leader. In the simulation these numbers are assigned and incremented as a robot is spawned. In a real world scenario the MAC address[2] or any other number, a priori known to all robots, can be used.

The commanding UAV needs to specify the formation used. In Figure 5.2 a number of formations are illustrated, each for different purposes. The quadrotor number[3] is shown in the figure for each setup.

With different formations a variety of task, the UAVs are capable of, can be conduct. The line, vee, and echelon (left or right) formation can be used for searches, because they all cover about the same surface area with their combined sensor range. The trail formation is applicable for special purposes, e.g., certain air-pressure experiments. Inside buildings it might be the only organization pattern possible, because there is no space to fly next to each other. Three-dimensional patterns are uncommon but possible. Bulk, for example, can be used to travel a distance where no sensor measurements are needed. Since the formation looks like a single large entity, it decreases the possibility of colliding with birds. Measurements of the flowfield of wind turbines have been conducted [55]. The stack formation can be a way to get concurrent data right behind the rotors.

Every quadrotor needs to keep its position in the formation, only two capabilities have a higher priority. Collision avoidance and the low battery interrupt can interfere with the movement to a certain position. From a follower point of view, a control system has to be in place to set the altitude, attitude, and velocity [19]. With an offset (S_η, R_η, T_η) described

[2] Media Access Control address (MAC address) is the unique, physical address of every network entity.
[3] Note: The formation can be reordered. It is not necessary that the leader flies in front of all other UAVs.

by the chosen formation, the reference trajectory for a follower $(x_\eta, y_\eta, z_\eta)^T$, hence, the desired position of the pursuer can be calculated by:

$$\begin{pmatrix} x_\eta \\ y_\eta \\ z_\eta \end{pmatrix} = \begin{pmatrix} x_0 + R_\eta \cos\psi_0 - S_\eta \sin\psi_0 \\ y_0 + R_\eta \sin\psi_0 - S_\eta \cos\psi_0 \\ z_0 + T_\eta \end{pmatrix}, \ (p = 1, 2, ..., n), \tag{5.1}$$

where $(x_0, y_0, z_0)^T$ is the position of the leader and ψ_0 its yaw angle. An offset value for T_η is unnecessary for 2-dimensional formations. Figure 5.3 shows a representation of this position setup, where $P_\eta = (x_\eta, y_\eta, z_\eta)^T$.

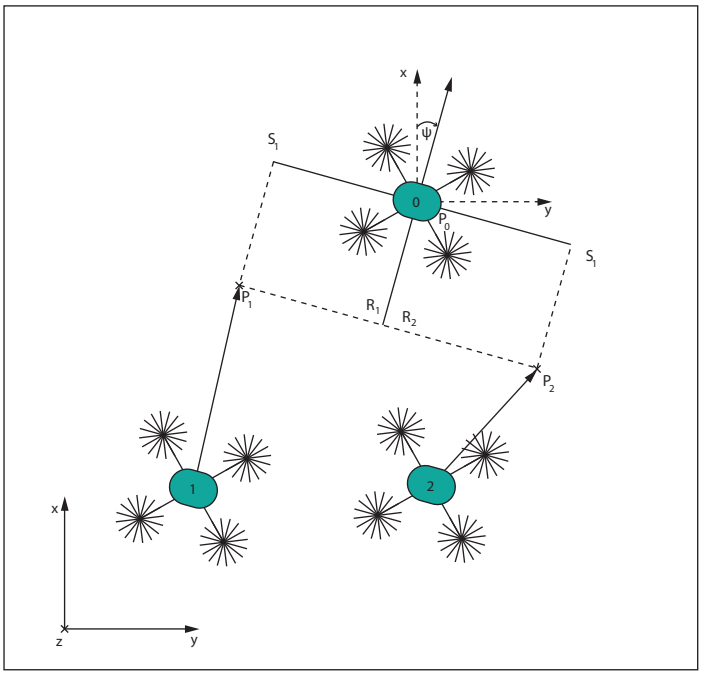


Figure 5.3: Desired position of two followers in a vee formation

After the position has been calculated, the velocity has to be controlled. The UAV has to match the speed of the leader at the right time, it first has to be faster than the agent in front, then it slows down. In [19] a sophisticated control system has been developed to oversee velocity and acceleration constantly.

A simplified version is currently implemented to test basic swarm setups. First the distance d_η, between the current location and the desired point, P_η is calculated, then the speed v_η for the "η"th follower is given by:

$$f_{v_\eta}(d_\eta) = -\exp\left\{\left(\frac{-2d_\eta}{d_{max_\eta}}\right)^3\right\}(v_{max_\eta} - v_0) + v_{max_\eta}, \tag{5.2}$$

and

$$v_\eta = \begin{cases} 0 & \text{if } f_{v_\eta}(d_\eta) \leq 0 \\ f_{v_\eta}(d_\eta) & \text{else} \end{cases}. \tag{5.3}$$

Here d_{max_η} determines the distance where the slowdown of the follower, trying to take its place, begins. v_{max_η} is the maximum velocity of the "η"th quadrotor. Figure 5.4 shows a representation of Equation 5.3 . As soon as P_η is reached the speed of the follower v_η and the leader v_0 is identical. Small errors of the position in the formation are tolerated.

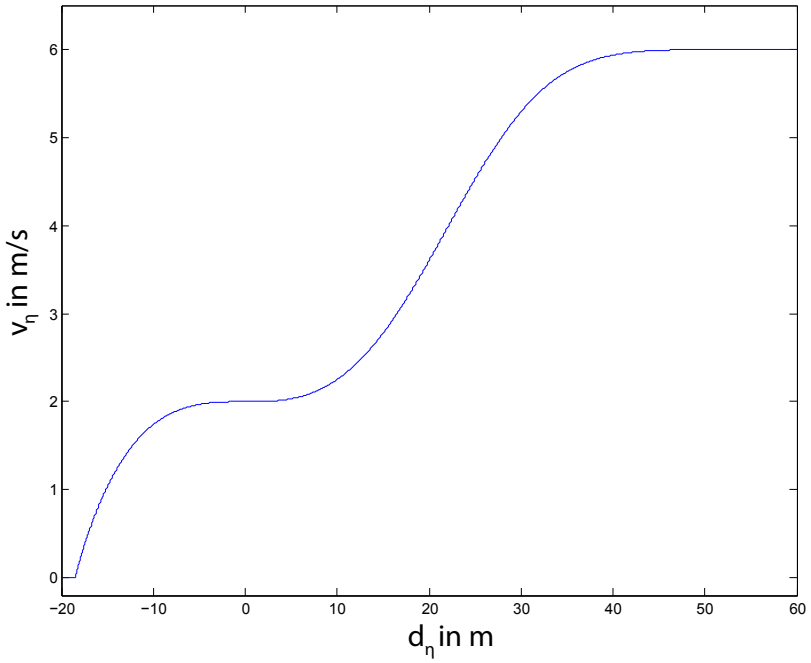


Figure 5.4: $f_{v_\eta}(d)$ with $d_{max_\eta} = 50m$, $v_{max_\eta} = 6\frac{m}{s}$, and $v_0 = 2\frac{m}{s}$

Should a pursuer get to close to the leader or another follower the collision avoidance protocol is activated. It will disrupt the advance, safely keeping the distance between the UAVs.

5.2 Discussion on Limited Communication

The limitation of the communication among agents arose from the idea of conserving energy. This approach resulted from the misconception that communication has a considerable impact on the power consumption of the quadrotor flight. This is not the case as can be anticipated from Figure 5.5. Nevertheless, there are other advantages to limit the communication, e.g., in military operations (radio silence). UAVs can perform tasks without the possibility of picking up their radio signals.

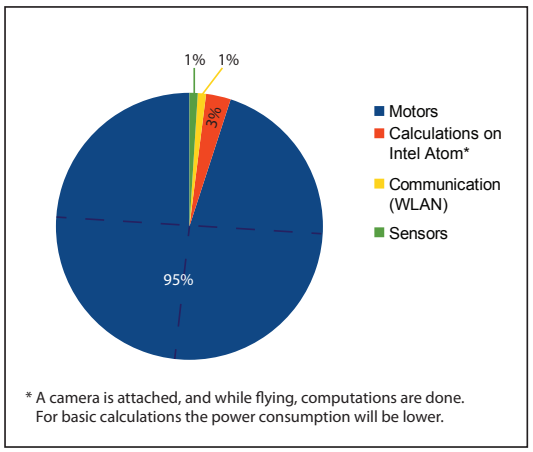


Figure 5.5: Average power consumption of the quadrotor at Hamburg Tech.

With limiting the communication, not the amount of data transfered is meant but rather the time spots when information exchange is permitted. All UAVs update their status at these times, as well as the understanding of the surrounding, i.e., the position of the team members, the actual position in the formation, and all necessary data about the mission. The locations on the map, and the time slots when communication is allowed are known before the start of the mission.

Only with advanced position estimation a formation can be obtained over a longer period of time without updating the location of at least a singe teammate. The probability of losing teammates, due to inaccurate sensor data, resulting in a difference between position estimation and the actual location of the UAV increases with time, because the positioning errors accumulate. Should an agent get lost, i.e., drift outside of the communication range, it determines that it is missing its teammates. If this happens measures need to be undertake, for example, it can continue straight to the next meeting point, and wait for its peers, or return right back to the base station. As soon as the team realizes the loss of members it has to accommodate to the new situation, e.g., redefine the search area, for the limited number of agents.

With an attached camera, team-maneuvers are also imaginable. Motion-flow, and object recognition enable UAVs to follow each other without active communication. The computational complexity is much higher for approaches like this, but with stronger processors these calculations can be realized. OpenCV is a collection of algorithms for vision

based approaches [43]. The package has been made available in the AURIS setup already. Currently it is used to draw the internal map, but OpenCV is much more powerful. It has ready-to-use functions for motion-flow and object recognition. The swarm setup and the formation stability can benefit from the 3D-vision algorithms during the time communication is prohibited.

Because the difficulties and the complexity outweigh the benefits of the limited communication approach, the idea is only shortly discussed and not further pursed. In "Distributed Collaboration with Limited Communication using Mission State Estimates" [8], more possible approaches for limited communication are introduced.

5.3 No Communication

One idea developed during this thesis is the emergent forming of a swarm without actual communication, and without the members being aware of the forming of the swarm. With a minimum set of commands all UAVs in range gather, stay together, and work on a task.

Each quadrotor sends out a signal, which channel will be used for that purpose in real application is not specified, yet. It could either be acoustic, radio, or even an optical signal. Most certainly a radio signal will be chosen. The communication protocol will be one of the current standards used in WLANs or WPANs. In the simulation an acoustic signal, observable by any quadrotor in range, is used. A sound sensor has been built to enable this behavior, as mentioned in Section 3.2.1.

Grouping The location of other quadrotors compared to the current position can be determined by the beep. Therefore, an agent can calculate how many other UAVs are in proximity. Figure 5.6 shows the behavior of four quadrotors which will be building groups. With the range of the acoustic scanner as the radius a hemisphere is created in front of each robot. Is another agent in that area and travels in the same direction, then it is the designated leader of the pair. The head quadrotor does not alter its behavior, if it has a follower.

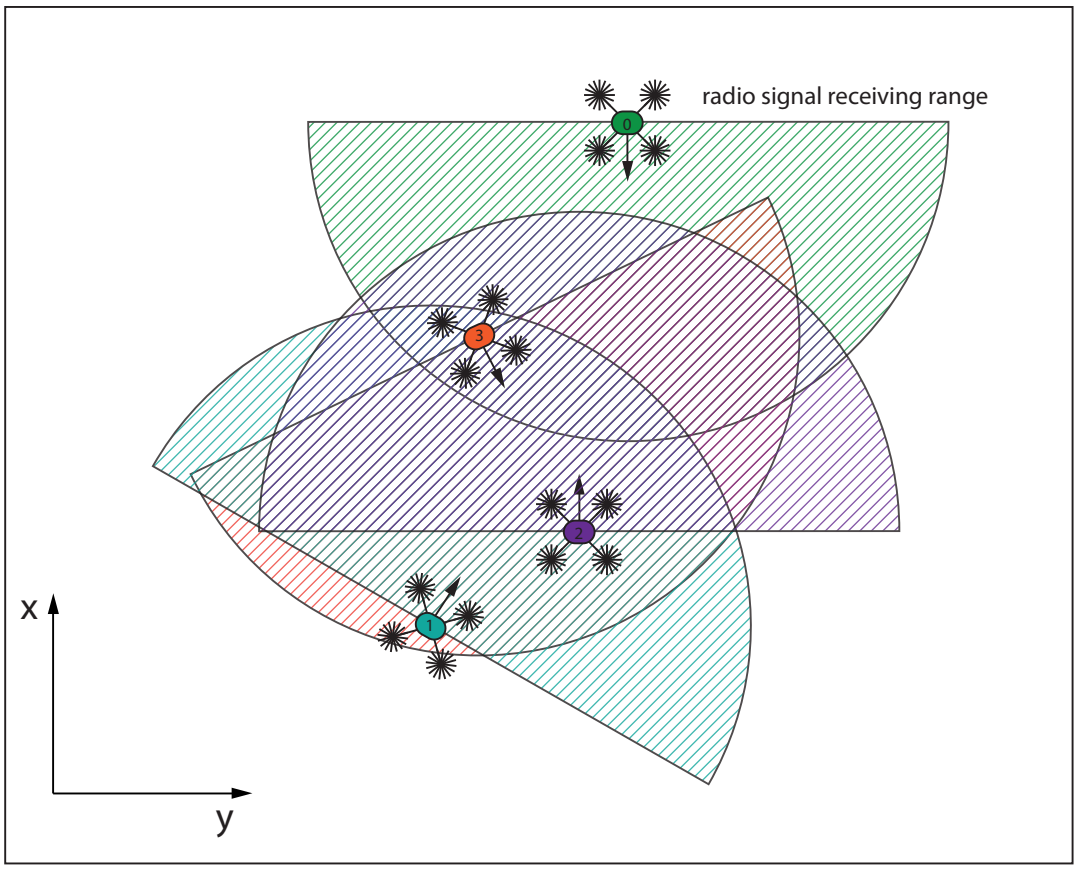


Figure 5.6: Quadrotor grouping

With the following basic set of rules, evaluated by each UAV, the swarm forms.

a. Is a beep in front of the UAV, i.e., x-coordinate of beep direction positive?

b. After the 2nd beep: Is the beep traveling in the same direction (dot product positive)?

c. Is there more than one beep with Question **a.** and **b.** true? Yes, calculate the distances between your current position and each beep, then follow the closer one.

In the example seen in Figure 5.6, two groups form, due to these simple rules, 1 & **2** and 0 & **3**, where the bold printed one is the leader of each group. Should the leader come to the boarder of a search area, then it determines a new flight direction. This can also happen due to other reasons, depending on the leader's configuration. The swarm

reconfigures itself at that point, because the follower does not recognize the quadrotor in front of it as leader anymore, due to the violation of **b** . The direction changing leader, is not trying to group with the UAVs in front of it, because they travel in another direction.

Figure 5.7 elaborates this behavior with four agents that all form one group. The dashed curves show the direction each quadrotor wants to take after loosing the leader. How the new direction is determined is unimportant right now. In the contrary, the dotted lines show the affiliation between leader and pursuer. The brown formation is formed shortly until it dissolves and one new swarm emerges. In this case, both groups are identical, that is not a necessity. This type of grouping will have the effect that all quadrotors seem to follow one leader. In fact, each UAV just follows the quadrotor in front of it.

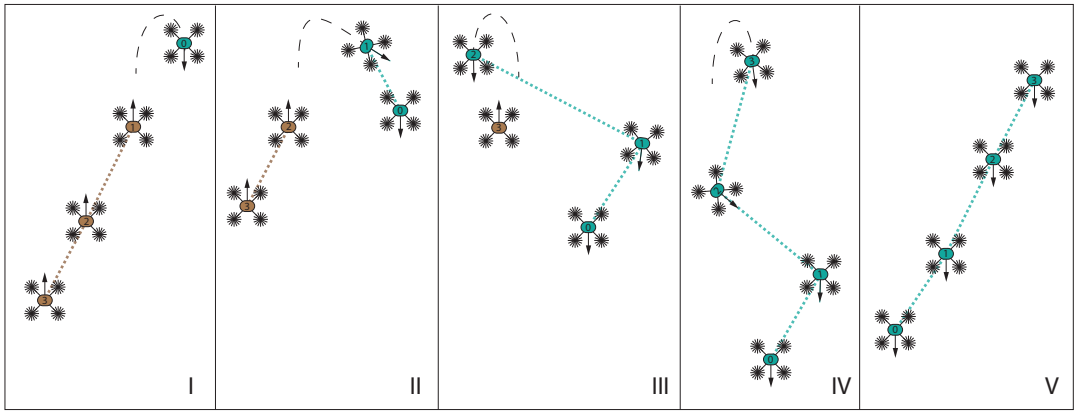


Figure 5.7: Quadrotor loosing formation and regrouping

The echelon or tail formations can be obtained with a basic offset[4] value given. Instead, the vee formation requires knowledge of other followers, which can be obtained, but this destroys the simplicity of this swarming algorithm. With an augmented offset solution, where the closest position, right or left of the leader, is chosen, a quasi vee formation can be established. Depending on the position the quadrotors add to the formation an overhang to either side will occur. The line formation will also need an adjusted algorithm, because no UAV is actually in front of the other, hence, **a.** would always be false.

With this self-organized formation, a wider sensor range than with a single UAV, is possible. That means, fire searches as mentioned in Section 5.4, and shown in Figure 5.12 can benefit. For a line by line search the formation is only beneficial if the leader considers the position of its followers, or the same area would be searched repeatedly.

[4]The offset usage is mentioned in Section 5.1 and shown in Figure 5.3 for the vee formation.

The “basically no communication approach” established, is a theoretical idea with almost no actual benefit over the methods involving communication. For a quadrotor being advanced equipment with different sensors and enough power for fast communication, this team building algorithm is not applicable. It might be possible to use the “beep”-design as a backup system. Nevertheless, with simple linear commands a group forms without the members of it being actually aware of it, this resembles emergents.

5.4 Search

A basic field robots and especially UAVs can be used for is the search of an area. Either for a certain stationary or moving object, or for safety reasons, to verify that the area has not changed, e.g., no fire broke out, or after an earthquake: nothing got destroyed. Searches can be conducted with a single agent or with teams. For a group of multiple UAVs, with the ability to communicate, new problems arise. First of all, each agent needs to search a different area to maximize the covered area at a time. With limited communication range the UAVs have to, either stay in close proximity, or reunite after splitting up, to share the progress they have made. Furthermore, a refuel strategy needs to be in place, alike the algorithm introduced in Section 4.5, but with an augmentation, because, if the UAV returns on way it came from it covers an area twice, increasing the overall search time.

For a valuable search, the robot needs to know its whereabouts in the sense of global coordinates. Since this is a simulation it can be assumed that precise movement is possible, and that the position knowledge of each UAV is accurate. Tests can be done with or without erroneous data to validate the robustness of introduced algorithms.

Layers and Cells Three-dimensional way planning and searching is still a rare research field, only few papers can be found. Kuwata and How [15] are mentioned, because they consider 3D trajectories for UAVs. Most 3D problems are considered in 2D space, this is true for team searches [2][27], or single robots. A three-dimensional search can be transformed in a number of smaller 3D areas, these can be understood as two-dimensional. Therefore, the 3D setting is divided into *layers*, which each are located in a certain altitude. When searching an area the number of layers needs to be defined a priori. For the tracing of a wild fire, for example, one layer is enough, because only the ground needs to be scanned. Whereas, air pressure measurements might need an increased number of layers to get the desired readings.

In addition to layers, search areas are split into cells. A cell is a quadratic area that can be covered by the sensors, this takes longer than a single time step (Δt). The whole time it takes to cover a cell is $T = c\Delta t$, where c is determined by the movement speed and the sample rate of the sensor, hence, T determines the time needed to move from one cell to another, Figure 5.8 explains what is meant by one cell. Small errors, due to the different shapes of a cell and of the radial sensor coverage, and due to the discrete sampling time, are discarded. On account of traversing a 3D environment, a cell is a cube ($A = r_s^3$). Where r_s is the sensor range, considered in all directions with the same value. With regard to the layer approach, hence, the agent does not change the altitude, a cell can be represented as a square (Figure 5.8 on the right).

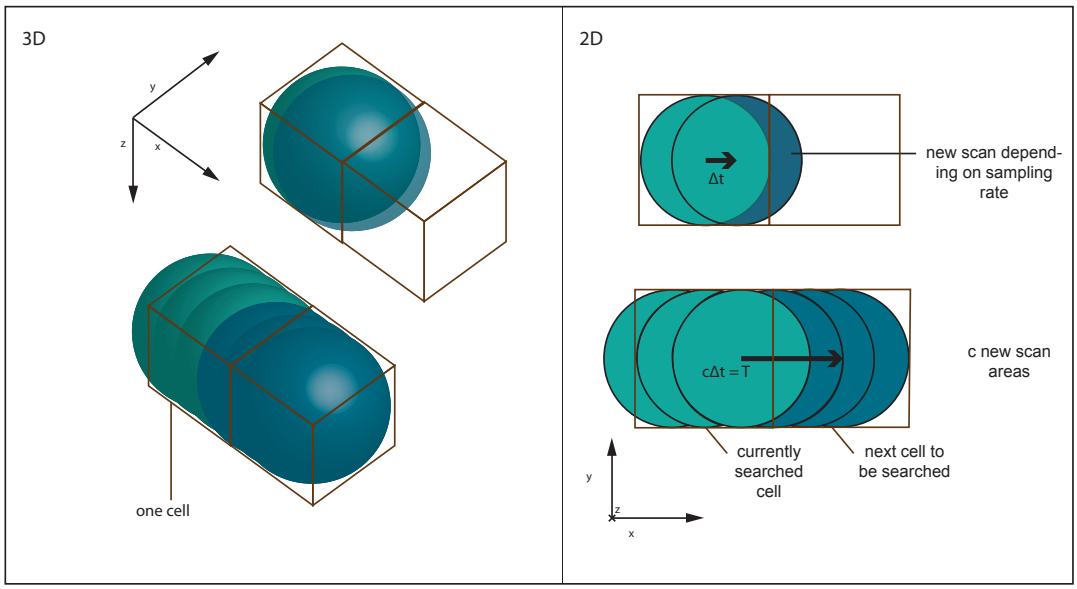


Figure 5.8: Clarification of sensor sampling rate and cell search

The total number of free cells is N. The sum of all blocked cells is defined as B. Cells can be either blocked or free to move trough. Covering all cells (N and B) needs $N+B$ time steps, because to identify that an area is blocked needs the UAV to scan this area, too. With a number of quadrotors R this time is reduced to:

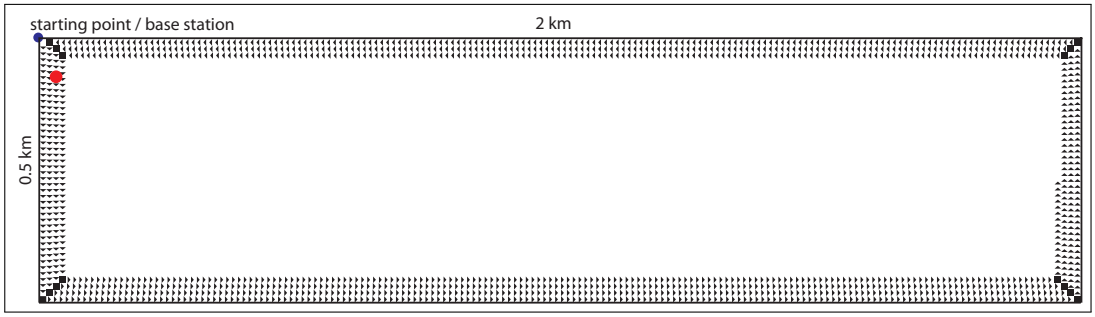
$$f_c\left(\frac{N+B}{R}\right). \tag{5.4}$$

When considering that the starting point of the robots (i.e, the base station) is not part of the search area, a distance D has to be traveled to reach the first cell of the search area. Hence, the lower bound for a complete search is:

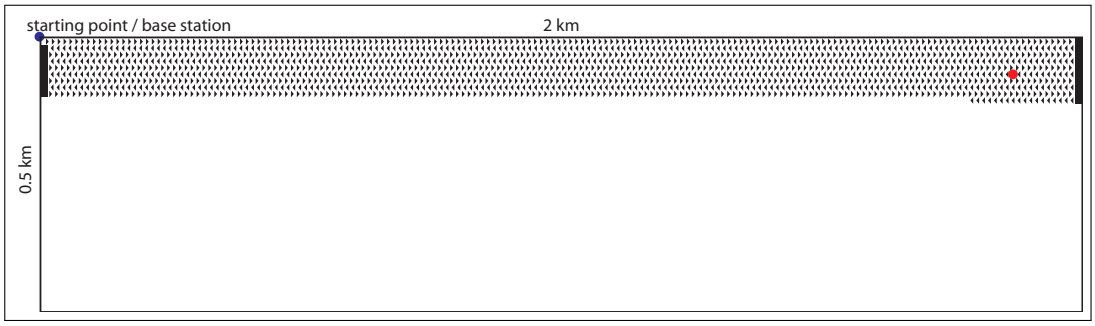
$$f_c\left(\frac{N+B}{R}+D\right). \tag{5.5}$$

5.4.1 Item Search

As the first actual search scenario an object retrieval is assumed. For example someone lost a bag in a corn filed and sends out an quadrotor to find it. Considering that the UAV is able to distinguish the bag from any other object in the search area. Figure 5.9 shows two simple search patterns for such a assignment. Flying on the boarder, in concentric circles until the middle is reached, or lane by lane. Arbitrary other strategies are possible, e.g., subdividing the area.



(a) Around and around



(b) Lane by lane

Figure 5.9: Quadrotor flight for full search coverage

To be more precise, the corn filed has an one square kilometer big area and the quadrotor in this scenario can be 30 minutes in the air, with a maximum speed of six meters per

second. Furthermore, the maximum sensor range is five meters, hence, the cell size is ten times ten meter. 10000 cells need to be searched, but the agent can only search 1080 cells ($6m/s * 1800s$) until the battery is empty. The red dot in the Figures 5.9 show the approximate location where this happens.

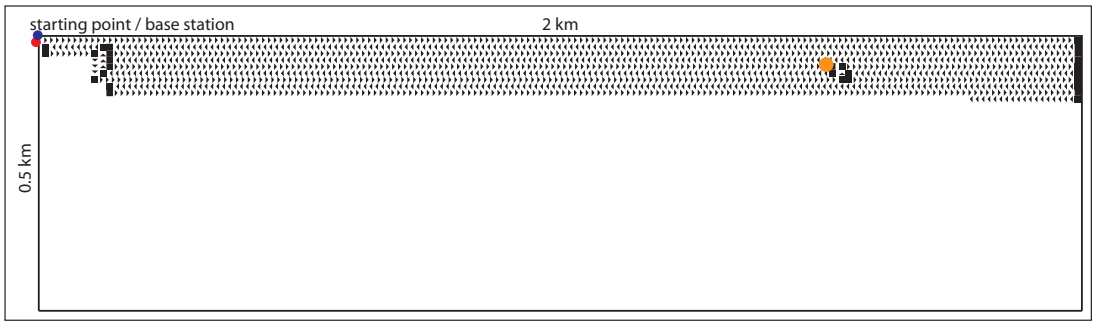


Figure 5.10: Quadrotor flight for full search with refueling strategy

Figure 5.10 shows an augmentation for the lane by lane search approach. At the orange dot the quadrotor determines that the battery charge is insufficient to continue the search, i.e., it proceeds on a return route close to the already covered area. A corridor in close proximity to the base station as been established to ensure only the minimum number of cells are searched twice. Calculations of the best possible route can be done before releasing the quadrotor.

This elemental search example already shows some of the difficulties that have to be faced when missions are conducted with robots. When more than one agent is available the search time can be decreased as shown in Equation 5.4.

Multi Agent Search An arbitrary number of different approaches to search an area with a team of robots can be proposed. Two main types can defined to divide those concepts, i.e., all UAVs search close enough together to stay in communication range (see Section 5.1, and Figure 5.2). Alternatively, each agent explores by itself and meets with the team to update the overall status. The later is subdivided in two different concepts again, i.e, they meet at a known location, or they meet randomly. The second approach should not be used by itself, because the possibility of meeting each other converges to zero as the perimeter grows. Should the agents miss each other, hence they do not meet, each agent searches the complete area, that means, the number R is reduced to being a value of redundancy instead of acting as a divider for the coverage time. Divided searches are more complicated than swarm approaches where all teammates can communicate over the whole course of a mission. Especially in an outside scenario it seems not applicable to

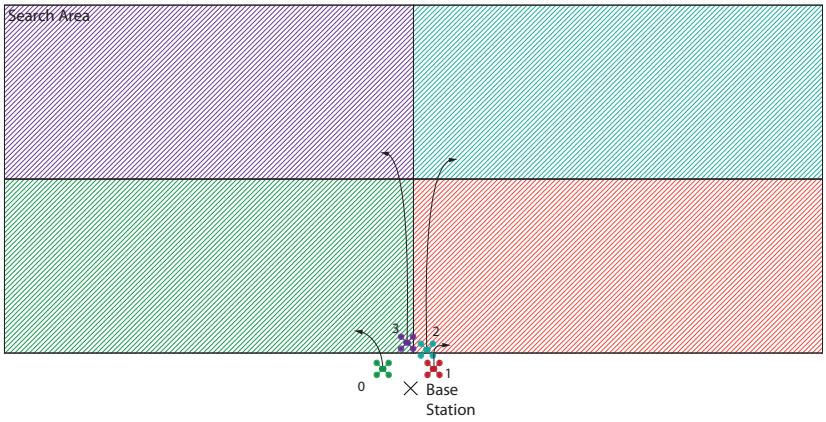
use any approach other than the formation flight. When using a split-up concept, there needs to be time to meet and to share the current status of the mission, this is overhead, compared the concept of agents staying in close proximity. For an open field this is true, in the contrary, assuming there is a big lake with a number of islands. All isles need to be searched but the water is not. Flying in formation to each small search area is increasing the overall search time, while divided, the completion time decreases.

A possible concept algorithm is introduced to deal with a group of quadrotors that split up to cover an area. All R agents split up dividing the search area equally among them. An estimate is proposed for the duration of the search of each subarea. After completing the search, or when a given timeout is reached, all quadrotors agree to meet again on a meeting line, which is close to all search areas. Not a single point is used to discribe the meeting area, because this point might be an obstacle, if the meeting is in an undiscovered area (i.e., when meeting for the first time this is always the case).

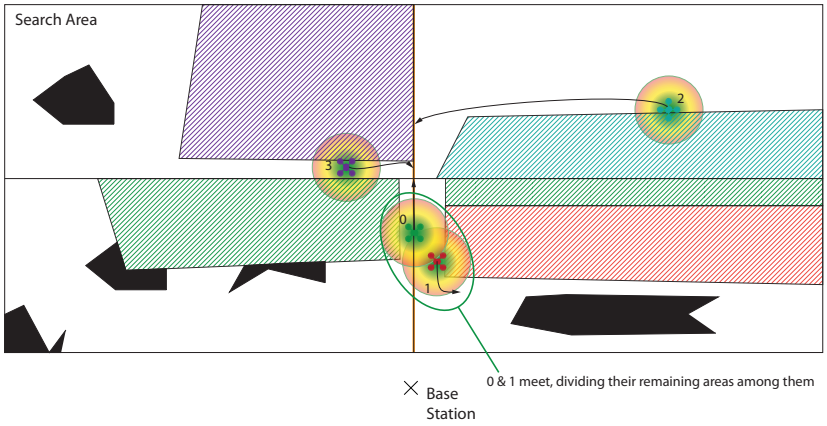
After getting to the meeting line, the agents travel toward the position where they expect the other members to come from. One point of the line has to be in a known area, hence the first meeting is, in the worst case, at the base station. After reaching the end of the line they turn around, until all team members are together. Should the meeting occur in an already searched area the meeting line disintegrates to a point. Should agents meet (adjacent search areas) before they are done, or the time is up, they update their current understanding of the map. Thereafter, adding the covered and uncovered areas, and newly dividing them among them. Furthermore, those two that randomly met, agree on a new meeting time, and line. Randomly, one of both UAVs attend the meeting with the other group.

When the whole group meets all data is updated, and the remaining search areas, minus the one which is still being searched by members that did not attend the meeting, is divided again. Figure 5.11 shows a rectangular area to be searched by four UAVs. Found obstacles are black polygons. In (b) one can see how agent 0 and 1 divide their remaining area among them and that number 1 does not attend the meeting of the other quadrotors, but keeps on searching.

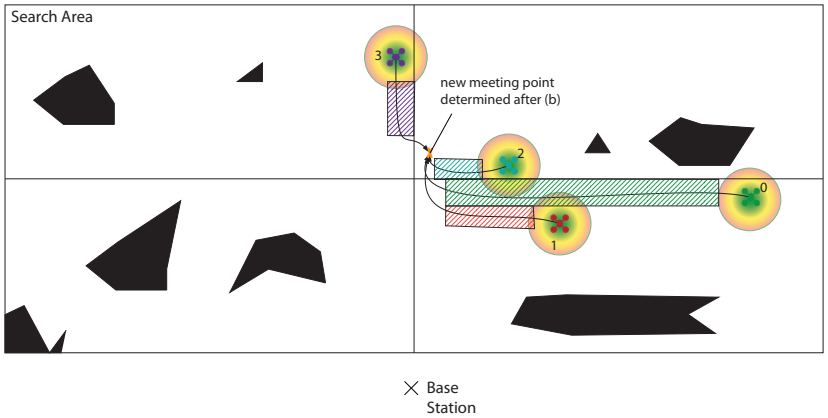
Tests might show that the subdividing among team members proves to be too complicated, or that possible errors occur. Then that part of the algorithm needs to be adapted. Without this augmented concept, the basic algorithm should hold, enabling a group of UAVs to split up an area between them while saving time in the progress. The longer an agent has to wait for its team mates attend a meeting, the higher the total search time. Error handling needs to be in place, if a quadrotor gets lost the others still need to continue the search.



(a) Start; subareas are determined



(b) Everyone is going to the meeting line (vertical bisect), besides 1 it keeps searching



(c) End; everyone is about to get done and meet

Figure 5.11: Four UAVs searching an area; colored areas still need to be searched by the quadrotor in the same color; black areas are obstacles

5.4.2 Reconnaissance Search

When searching an area for a fire a lane by lane search, as mentioned in Section 5.4.1 is not applicable. It needs to be searched over a wider spectrum. Especially if the size of the possible finding is unknown, it might make sense when partial areas are skipped. For example, it is considered that a quadrotor is used to search for a fire which has a size of 5x5 meters. Missing lanes in both directions of 4x4 meters is unproblematic because the flames reach into a searched area.

Therefore, a new search algorithm is proposed. To resume the example from Section 5.4.1, the base station is in the top left corner, and the UAV can only cover about a tenth of the complete search area without refueling. The agent flies one to the opposite side of the search area dividing it into two smaller areas. The areas centroid is determined (Equation 5.6) and the UAV continues on a line between the current position and the centroid, until reaching a border of the search area.

Area calculation of a polygon and centroid computation:

$$\begin{aligned} A &= \frac{1}{2}\sum_{i=0}^{N-1}(x_i\ y_{i+1} - x_{i+1}\ y_i), \\ C_x &= \frac{1}{6A}\sum_{i=0}^{N-1}(x_i + x_{i+1})(x_i\ y_{i+1} - x_{i+1}\ y_i), \\ C_y &= \frac{1}{6A}\sum_{i=0}^{N-1}(y_i + y_{i+1})(x_i\ y_{i+1} - x_{i+1}\ y_i), \end{aligned} \tag{5.6}$$

In the next step, the algorithm computes the largest subarea, and calculates a new course for the quadrotor, through the centroid of this subarea. Figure 5.12 elaborates the proposed algorithm. A Safety protocol is in place to recharge the battery as needed, the yellow dot in the figure indicates the moment the quadrotor has to fly back.

From the computational point of view, this approach is suboptimal since in the worst case, 2^n (where n is the number of flights from one side to another) areas need to be sorted. However, not every cross flight can split all areas in half, so the complexity is reduced. Furthermore, these calculations can be done before liftoff by the base station, i.e., a faster computer. Only the way points are saved on the quadrotor side, and processed one by one.

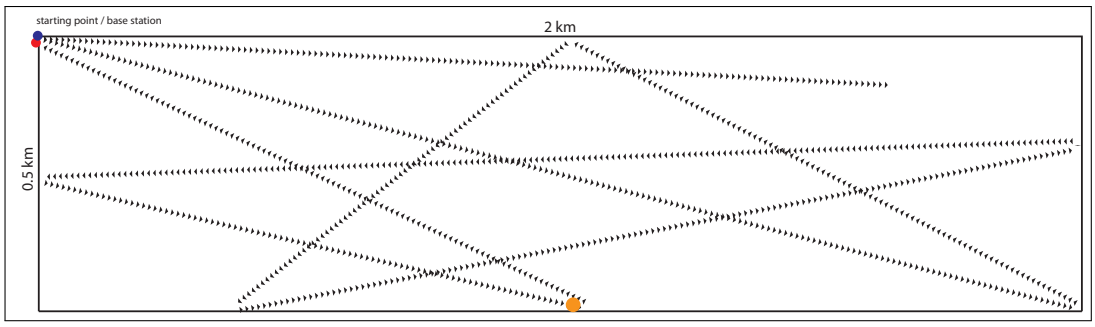


Figure 5.12: Quadrotor flight for full search (fire) zig-zag pattern

5.5 Tracking

The following section introduces another usage area for quadrotors. The algorithms proposed can be implemented with the introduced simulation environment and AURIS. Only the detection of a fire, and the return to the base station, have been tested so far, the advanced concepts introduced here, still need to be implemented, yet.

Ideas discussed so far, especially regarding different search patterns are always concerned with finding a single entity, and reporting the discovery. If the object found needs to be monitored, after it has been detected, an augmented behavior needs to be introduced.

Considering a child in a cornfield, it got lost. A quadrotor found the kid, but it has to stay in close proximity, because the child will probably keep wandering around eventually getting lost again. The UAV could work as a beacon, flying up in the air to be seen from far away. A possible second UAV would have to transmit the message of the retrieval to the base station, hence, the search team, while the other stays with the kid.

Fire Perimeter Tracking Another scenario where UAVs could be useful is the tracking of bush fires. Every year a large number of fires cause potential harm to people, and cost a lot of money. Especially the helpers, i.e., firefighters are endangered by the blazes. Changing winds and different outbreak areas make fires extremely hard to predict. It is proposed to use a number of quadrotors to monitor the fire constantly [22, pages 247-264].

The UAVs are equipped with infrared sensors, and cameras to locate and track the fire. Moreover, due to the camera the firefighters have visual contact to the fire without getting into harm's way. Expecting the burning area to be larger than the communication range of a single quadrotor points out, that data captured by the UAV can not be analyzed by the

ground crew in real time. Therefore, minimizing the offset, between capturing the data and delivering it, is crucial.

First, it is assumed that only a single quadrotor is tracking the fire. The UAV has autonomous guidance as described in Chapter 4 (coalition avoidance, limited fuel consideration are in place). The way-points are calculated by using the infrared sensor, always staying in close proximity of the fire without flying into it. After finishing one lap around the fire perimeter the quadrotor delivers all captured data to the base station. If possible it flies another lap or recharge its battery. The time t_l it took the quadrotor increases as the fire grows. The development during this time can only be predicted, thus t_l needs to be minimized.

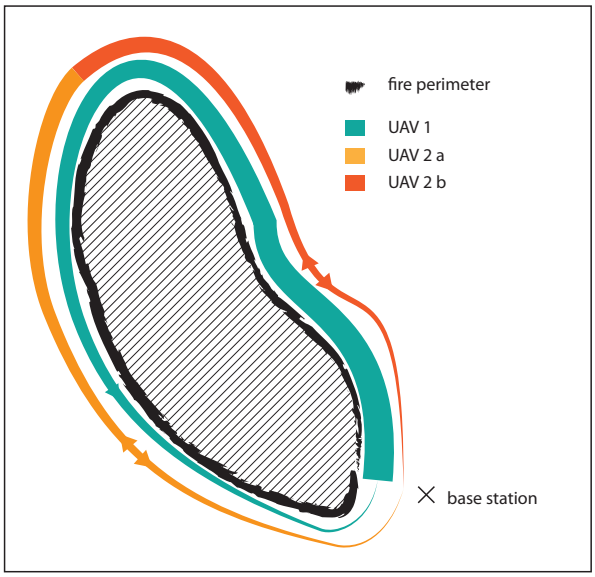


Figure 5.13: Fire monitoring, 1 vs 2 UAVs (after [22])

For one UAV the latency t_l cannot be changed, because it is equal to the time the quadrotor needs to traverse the perimeter ($\frac{s_{perimeter}}{v_{UAV}} = t = t_l$). Using two quadrotors cuts the latency in half. Figure 5.13 shows the coverage of a fire hazard. The thickness of the flight path displays the value of t_l, that is, how old the information about that point in time is when returning to the base. Adding any number of UAVs to this setup does not reduce t_l any further.

To be sure not to have missed any part of the perimeter, a single quadrotor has to conduct its search until it meets with a UAV coming its way, which has covert the other side of the fire. Then it returns to the base station (still searching, to minimize the latency). The fly-back time is as long as t_l for the furthest away point (half the perimeter;

Equation 5.8 & 5.10). Adding more UAVs returns more updates in the same time, but t_l cannot decrease any further.

For one UAV the latency $t_l(x)$ at a point x is given by:

$$t_l(x) = \frac{P - x}{v}, \tag{5.7}$$

where v is the current velocity of the quadrotor. The total latency for each point on the perimeter is therefore:

$$\int_0^P t_l(x)dx = \frac{0.5P^2}{v}. \tag{5.8}$$

For two quadrotors, one going the opposite direction of the other, the equation looks like this

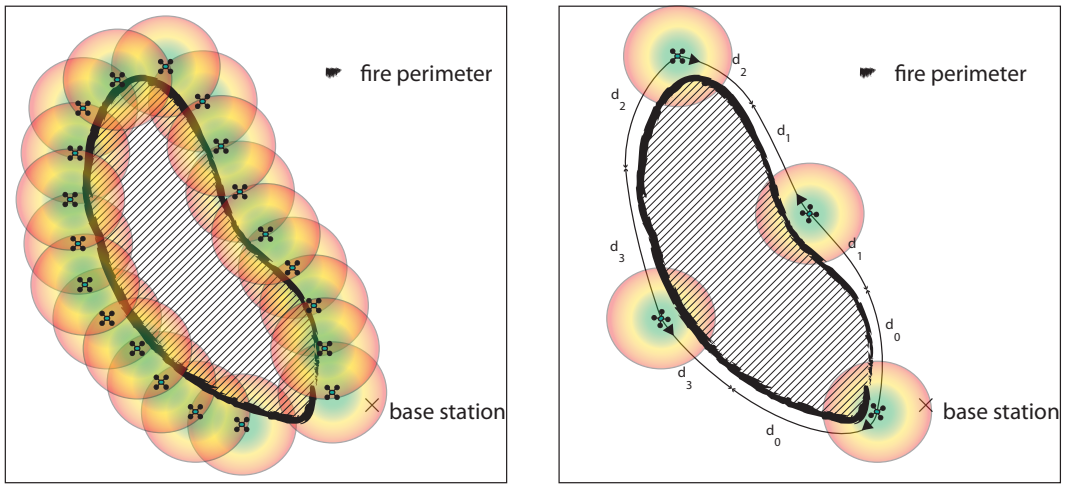
$$t_l(x) = \begin{cases} \frac{x}{v}, & \text{if } 0 \leq x \leq \frac{P}{2} \\ \frac{P-x}{v}, & \text{if } \frac{P}{2} < x \leq P \end{cases}, \tag{5.9}$$

with an overall latency of

$$\int_0^P t_l(x)dx = \frac{0.25P^2}{v}. \tag{5.10}$$

With a growing fire, the latency is going to get higher, too. As a fire spreads out t_l grows two. One approach to overcome this shortcoming is to evenly spread out UAVs all over the fire perimeter. All agents can communicate, and they send basic updates right to the base station. The latency is the time the signal needs to surpass all quatrotors, hence, it can be considered instant ($\frac{s_{perimeter}}{c_{\approx speed\ of\ light}} = t_l$). Figure 5.14a shows a possible setup.

If that many UAVs are available this approach reduces t_l to the minimum. The main problem thereby being the large and growing number of UAVs needed. More problems arise if the communication chain breaks or when the battery of certain agents need to be recharged. Concluding that this idea is not actually feasible.



(a) All UAVs stay in communication range

(b) Each UAV monitors a part of the fire

Figure 5.14: Cooperative fire monitoring

An alternative approach, where t_l cannot be minimized, is shown in Figure 5.14 b. Pairs of UAVs divide parts of the perimeter among them. For each pair, one agent is flying in the clockwise direction until it meets its partner flying toward it. When they meet data is being transferred. The distance traveled (d_i for the ith UAV) between rendezvous is tracked by each quadrotor. When meeting, each pair adds up their distances and divides them evenly ($d_w = \frac{d_i + d_j}{2}$).

The agent with the shorter traveled distance waits for the other at the calculated meeting point d_w (the midpoint between the pair). Figure 5.16 shows how four pairs of quadrotors evenly distribute over a circular steady perimeter. With this algorithm, a growing fire can be covered as well because, no more than one UAV waits for its partner, the other keeps going until meeting its neighbor. Adding UAVs at any time is also possible as can be seen in the figure.

Switching positions on the perimeter can be helpful to conveniently replace UAVs. When meeting, the agents compare their remaining battery charge, the one with the lesser reserves takes the place closer to the base station. Ensuring the quadrotor with the least power left to be nearer to the recharger.

The latency for this approach is the same as for two quadrotors traveling across the entire perimeter, because the information can only travel as fast as the quadrotrs can fly to their meeting points. Nevertheless, this algorithm ensures that UAVs can be replaced easily,

especially for refueling, and that a growing perimeter is dealt with automatically. Furthermore, should one agent leave the formation, because of an unexpected event, the others adjust and continue to cover the search.

A modification of the explained algorithm states that UAVs should not wait for the other to check back at a meeting point, but to continue to traverse the perimeter until meeting their partner [12]. After joining, the pair flies together to their designated meeting point, recalculating it if the perimeter has changed (due to growing fire, or movement of other agents). On average this approach converges slower compared to the first algorithm. Monte-Carlo simulations conducted in [12] state that the first algorithm converges on average 0.67T faster than the altered idea. Waiting at a certain point for a teammate to show up might be considered a waste of energy, especially if the perimeter grows quickly up to half the team will be loitering, therefore this algorithm might still be preferred.

Figure 5.15 shows a comparison of two teams of UAVs getting released at the same circular perimeter. With the y-axis being the distance from one end of the base station to the other, that means the middle is the furthest away point. Every line shows the position of a single quadrotor traveling across the perimeter.

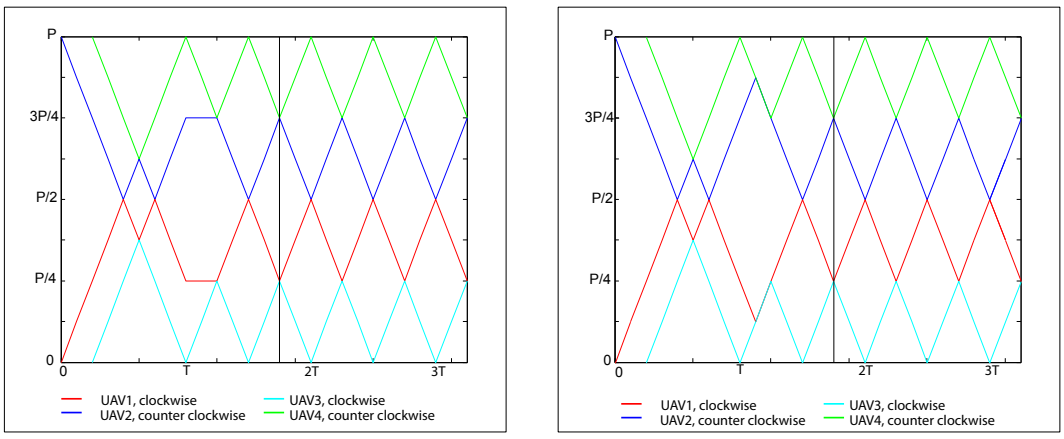
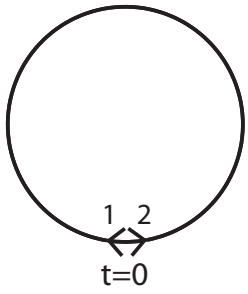
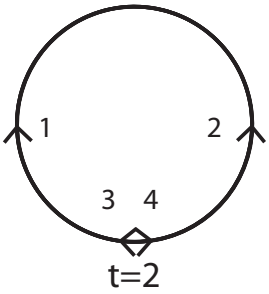


Figure 5.15: Comparison (a) algorithm where UAVs wait for one another, (b) 2nd algorithm, UAVs pair up until they reach the midpoint of each section

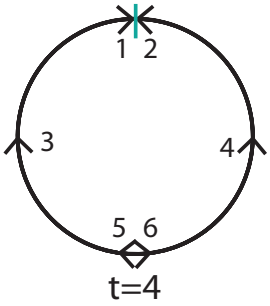
In the worst case scenario, if all agents start at the same location, and fly in the identical direction, e.g., start at the base station, and all flying clockwise, then the algorithm takes longer than the first one proposed. All agents will travel the entire perimeter together, until they start to evenly distribute. Launching the quadrotors in a structured matter, is therefore important.



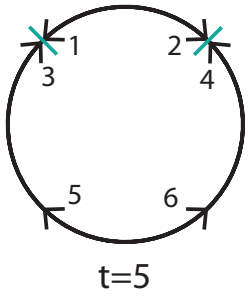
(a) launching 1&2



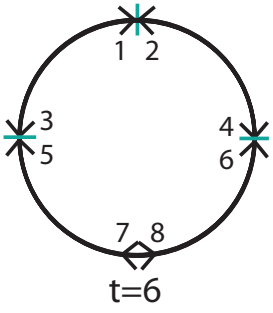
(b) launching 3&4



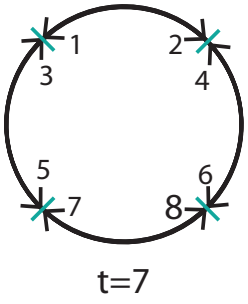
(c) launching 5&6; $d_1 = d_2 = P/2$; no waiting



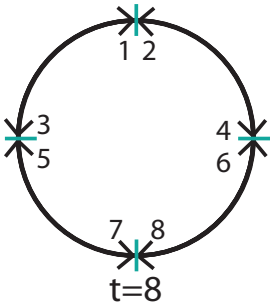
(d) $d_1 = P/8 < d_3 = 3P/8$, thus 1 will wait



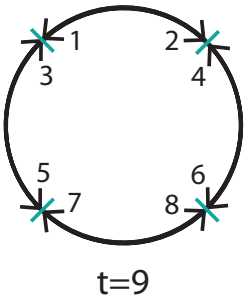
(e) launching 7&8; $d_3 = P/8 < d_5 = P/4$, thus 7 will wait



(f) all $d_i = P/8$, no waiting



(g) all $d_i = P/8$, no waiting



(h) same state as (**f**), algorithm converged, minimum latency found

Figure 5.16: The load balancing algorithm(1st introduced algorithm) with eight quadrotors in analogy to [22, pages. 247-264]

6 Results and Evaluation

For a simulation the main indicator of completeness and usefulness depends very much on the realism employed by it. Various experiments are conducted to verify the scalability of the simulation environment. In diverse maps, different quantities of quadrotors are spawned to explore the area.

All test conducted show that the simulation environment is stable, and that the physical behavior of the quadrotor is plausible. Visually the simulation looks entirely convincing, as is demonstrated in Figures 6.1 and 6.3.

6.1 Tests

The functionality of all AURIS routines is tested in detail. The development of the control program is the main purpose of this thesis, therefore, its effectiveness needs to be validated thoroughly. In detail, the correctness of collision avoidance, basic flight planning, and the effectiveness of the senors is constantly verified. The setup of an actual simulation with two quadrotors can be seen in Figure 6.1, it shows AURIS for both UAVs, the 3D simulation, the maps drawn by each agent, as well as currently processed sensor data.

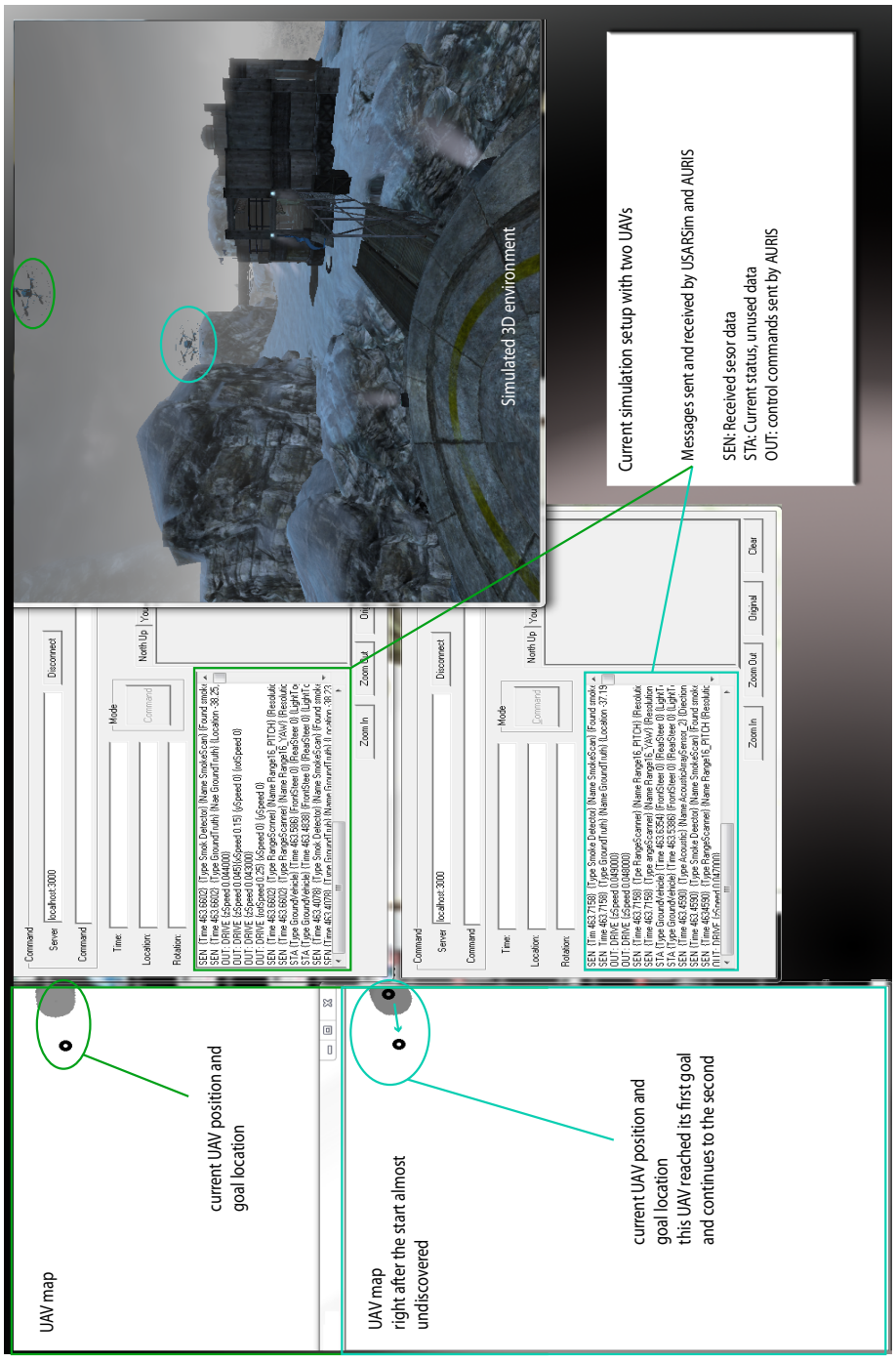


Figure 6.1: Simulation setup with two UAVs, AURIS, and the drawn map.

6.1.1 Scalability

A system that simulates a number of entities needs to be stable and scalable, to be valuable. Especially for swarm behavior a group of agents with enough space to spread out needs to be realizable. Furthermore, the cost of the simulation should be smaller than for the actual implementation. This is the case with AURIS. A quadrotor, with a set of sensors enabling it to communicate with peers, costs about 400 Euro[1], this is about the same amount needed to setup AURIS and Unreal Tournament 3 for at least twenty quadrotors, which will not be vulnerable to erroneous implementations. Therefore, the scalability is verified. Moreover, stress tests on the actual hardware system are carried out, to ensure the durability of USARSim and AURIS.

Swarm Size As soon as the simulation environment (USARSim) is started, a window opens where the current map can be seen. Should a UAV be attached with a camera, then the spectator can verify the data captured by the quadrotor in a smaller window on top of the main frame (as can be seen in Figure 3.6). Since the data in the small and in the big window needs to be rendered separately, the graphic accelerator has to do almost twice the work. Adding more UAVs with attached cameras degenerates the frame rate to the point that the simulation is corrupted, because Δt might become bigger than $0.1s$. All calculations that have to be done internally, e.g., force computing or PID controller settings, might be interrupted. Table 6.1 shows the frame rate per added quadrotor [2].

#UAVs	Avg. CPU usage in %	FPS (max-min)
0	54.5	62-37
1	56	32-21
2	59.5	27-16
3	61.5	23-15
4	62.5	20-14
5	64	19-14
6	65	17-13
...	...	...
10	70	12-9
...	...	...
20	80	—

Table 6.1: CPU usage and FPS with multiple simulated UAVs

[1] http://www.conrad.de/ June 2011

[2] All test have been conducted on the same machine. An Intel i5 quad core, with 6GB of RAM, and a low end graphics accelerator the NVIDIA GeForce 210

A frame rate below 15 FPS is not acceptable. Therefore, the camera view is turned off when more than three UAVs need to be simulated. The data captured by the different cameras can still be used for calculations. Just the user, who is watching the simulation, will not be able to see what every single quadrotor is seeing, anymore. Without the multi-view the frame rate is constantly between 62 to 25 FPS. These values can be increased even further when upgrading the hardware. The CPU usage is almost not affected by this, because the graphic accelerator does most of the work. In addition, increasing the number of UAVs is not as stressful for the CPU, because AURIS does not need that much processing power. Until now, 20 agents have been simulated simultaneously, for all scenarios mentioned so far this number is big enough. With better hardware this number can be increased. An upper bound has not been defined yet.

Map Size When the first simulations with USARSim started, a small map was used to concurrently test new features of AURIS and the environment. While progressing with the work, especially focusing on swarms, bigger areas were necessary. The currently largest map simulated has an area of 250000 square meters. The biggest possible map, with the current conversion between UU and meters ($1m = 250UU$), is 2097m x 2097m, i.e., approximately 4.4 square kilometers. Changing the conversion factor to $1m = 50UU$, an area of over 100 square kilometers can be simulated [41], this is comparable to the expanse of a medium sized town. Should an even bigger environment be needed it has to be streamed from a number of smaller maps.

The maximum altitude is currently set to four kilometers. This is sufficient for all test conducted. Considering changing the conversion factor to $1m = 50UU$ increases the maximum height by a five-fold, i.e., 20.9 kilometers, this is even higher than the standard cruise level of a commercial airplane.

6.1.2 Kalman Filter and Positioning

With the Kalman Filter approach, errors in positioning of either other UAVs in a swarm setup, or of the current location of a quadrotor itself, are reduced. In addition, outliers in collected data are distinguished and their impact is decreased successfully as mentioned in Chapter 4.4.2. The Kalman Filter reduces errors in data sets. In Figure 6.2 the dotted, brown line shows the correct value, the blue line is the received data with outliers due to erroneous measurement. Furthermore, the green line shows the smoothed data output of the filter.

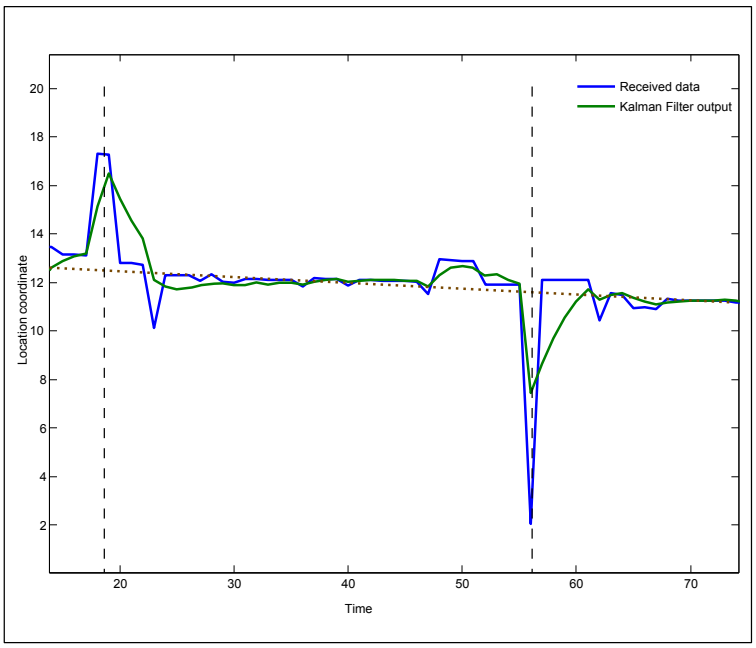
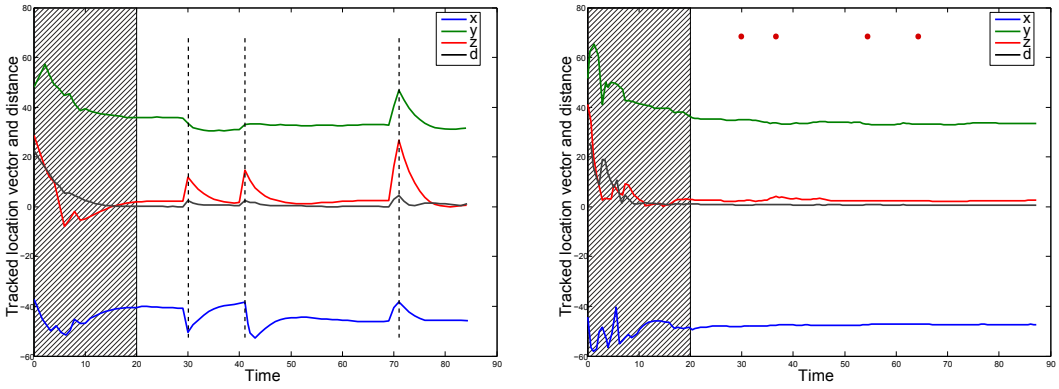


Figure 6.2: Kalman Filter error reduction. Brown dotted line: correct value; Blue: measured and received data; Green: Kalman Filter output; Black dotted line: position where outliers occurred

Figure 6.3a shows a graph of Kalman Filter data. The current position ($(x,y,z)^T$) and the distance (d) of another UAV is tracked. For this test, both agents try to hold a steady position in the air. The hovering is a well controlled task in the simulation, and the corrections needed to stay in place are minimal.

When looking at the figure, $(-48,3,33)^T$ is the location of the tracked quadrotor. The dashed lines show situations when outliers occurred. The Kalman Filter adapts these erroneous values, but spikes can clearly be seen. Furthermore, concurrent data is not represented correctly because the Kalman Filter needs a couple of steps to level out. Figure 6.2b, shows how the outlier reduction works. The red dots symbolize the position where a possibly wrong value was discarded.

All three plots show the importance of outlier reduction, and the robustness of the Kalman Filter. To accommodate to different types of data sets and errors the underlying covariants matrix of the filter can be adjusted. Therefore, the Kalman Filter is a valuable asset to the fundamental principles of the control program design. In a real world application, the same approach can be applied. Possibly the model error and the measurement error need to be adjusted to correspond to the applied sensors.



(a) without outlier correction

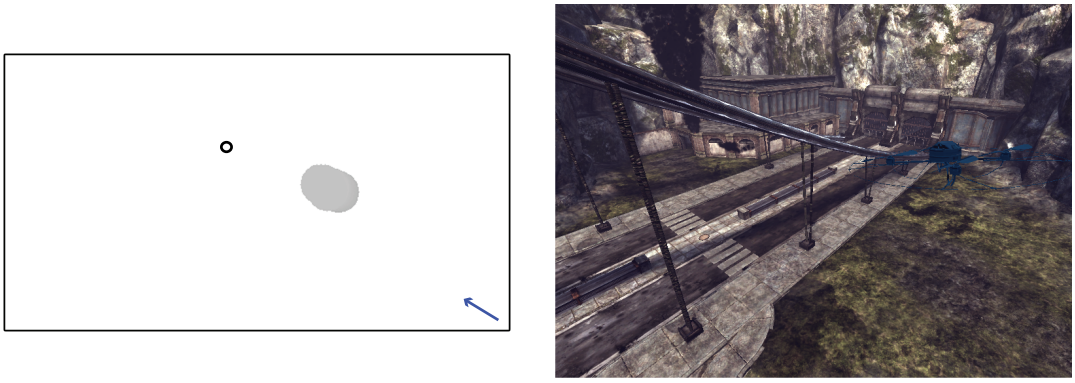
(b) with outlier reduction

Figure 6.3: Tracking of a position and distance. The gray area illustrates the time steps the Kalman Filter needs to even out, outlier reduction is not done during this time.

6.1.3 Scenarios

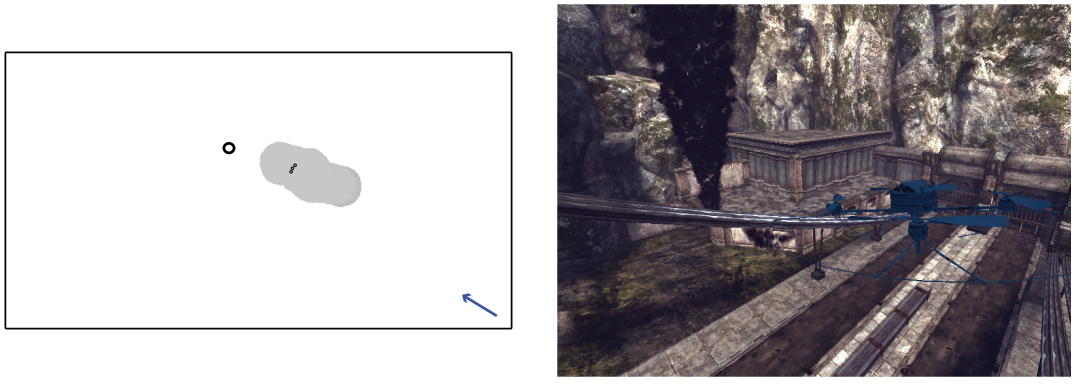
To validate the functionality of AURIS and of all sensors, different scenarios are executed. A basic test looks like this: a number of way points are introduced and the quadrotor has to reach the final position without hitting any obstacle along the way. The collision avoidance algorithms stated in Section 4.2 is verified this way. Figure 4.6 shows a picture of the map drawn internally by the UAV, while such a scenario was conduced.

The second test scenario is sending out one or more UAVs to search for a fire, and after finding it, to return to the starting point. In Figure 6.3 one quadrotor is released, it is already pointed into the right direction to shorten the search time. All guarding systems are in place, to shield the UAV from hitting obstacles. The smoke detector described in Section 3.2.1 is also attached. After passing by objects in the way, the quadrotor detects the fire and returns to the home base. The complete developed system can be verified this way. The balancing, and the physical behavior of the robot, as well as the functionality of AURIS.



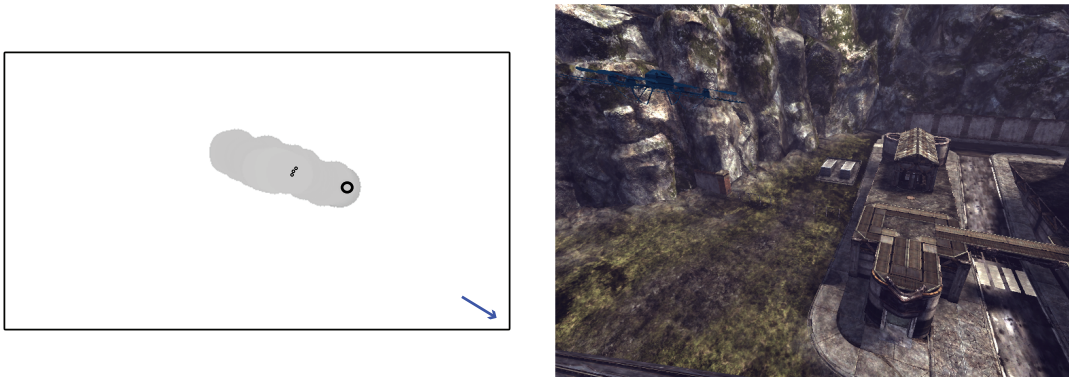
(a) A quadrotor is started to search for a fire, the first goal (black dot), is in proximity of the fire to shorten the search time

(b) The UAV detected an obstacle blocking the way, small black circles; the agent turns to pass by the object



(c) After passing the obstacle the UAV continues straight to the goal location

(d) The fire sensor detected the flames, the UAV turns around to fly back to the starting point



(e) The agent continues back to the start

(f) The quadrotor lands at the starting point

Figure 6.3: Search mission: UAV finds a fire hazard and returns to the start location; blue arrow denotes UAV flight direction

6.2 Evaluation

The simulation, from a physical point of view, is only as realistic as the underlying game and physics engine. With the Unreal Engine 3 game engine and the Nvidia PhysX physics engine, state-of-the-art handling of three dimensional modeling and computing of physical events is acquired. Furthermore, the modeled quadrotor is bound to the physical rules of the engine. The UAV flight is modeled with attached forces, where the rotors attach, as well as the balancing is only achieved through a variation of those forces. Therefore, the simulation is as realistic as possible.

In [33] it is stated that a valuable simulation needs flexibility, physical realism, visual realism, efficiency, modularity and effective control. In fact, USARSim fulfills all those requirements. It allows the simulation and creation of different robots, sensors and environments. Interaction between robots and its environment is handled, as described in the prior paragraph, by the Nvidia PhysX engine. The visual realism is not actually of great importance, as long as the calculations are done correctly. Nevertheless, since the Unreal Engine is also used in the game industry the graphic output is currently one of the best possible. The efficiency of newly introduced scripts has to be evaluated by each developer separately, for USARSim 3, it can be said that the efficiency is very high. As can be seen in Table 6.1, 20 UAVs can be simulated on the same machine, therefore claiming that the current setup and USARSim are efficient. After the creating process of robots and sensors is completed, adding and replacing them can be done conveniently in a setup file. Additionally, elementary parameters are controlled in the same manner, hence, enabling modularity. Since USARSim uses a TCP/IP interface to send control information to the robots, any programing language capable of networking can be used to implement controllers. This client-server architecture can be used to conduct complex computations on dedicated machines, i.e., separating the simulation from the control software. Another point to mention, when evaluating *effective control*, is that the simulation can be replaced by a real world application (a real UAV), on account of the TCP/IP interface.

Nevertheless, a simulation can never be one hundred percent like the real world. With USARSim only time discrete events can be investigated, what ever happens between sampling intervals can only be predicted. Furthermore, certain aspects can not be simulated with USARSim. Some might be integrable with intensive scripting, others are shortcomings of the Unreal Engine itself.

Emission of radio waves and heat has not been implemented so far, they would be a great asset to the realism of the simulation. Especially the effects of interference of waves could be helpful to integrate better wireless-LAN adapters. Infrared cameras could be valuable if thermal radiation would be implemented. The missing Linux support, and the

semi-finished port from USARSim 2 to the third generation of the Unreal Engine diminish the performance of USARSim 3.

Finally it is to be said that the simulation will be a great asset to the further development of quadrotors, and especially UAV swarms. The advantages of USARSim outweigh the shortcomings by far. Especially the expandability will encourage developers to upgrade this simulation environment further.

7 Conclusion and Future Work

The autonomous control program AURIS and the simulation of physically convincing quadrotor in USARSim 3 are ground-breaking first attempts which introduce diverse opportunities for further development. Different simulations have been conducted to test the scope for expandability in the USARSim environment. The flexibility to adapt to variable concepts is significant. With the power to build new sensors virtually any scenario predictable can be implemented, considering certain adjustments.

Furthermore, possible areas where UAVs can be of aid for research and human live are described. The search or the tracking of fires is explained, as well as algorithms to introduce and handle the swarm behavior of different agents. The goals, adding the quadrotor into the three-dimensional simulation environment and conditioning it to autonomously find flames and reporting the discovery back to a base station have been fulfilled. Newly developed sensors work as predicted and enable the quadrotor to pair up with teammates.

With the setup now in place more advanced simulations can be undertaken and problem solutions can be verified before they are implemented in a real world system. A major task in the future will be enhancing the simulation for indoor assignments. Sensor upgrades and faster flight maneuvers will be necessary to achieve this.

Self-organization, and advanced autonomy is a possible new research field, considering that UAVs determine independently how to solve tasks or even search for new assignments themselves. Groups of UAVs might emerge from a single agent broadcasting for help. This approach might be more applicable in space, where flight times of small drones could be extended over several months due to constant sunlight exposure, and the usage of solar panels.

References

Books and Articles

[1] M. Achtelika, A. Bachrach, R. He, S. Prentice, and N. Roy. *Autonomous Navigation and Exploration of a Quadrotor Helicopter in GPS-denied Indoor Environments*. Technical report, Technische Universität München, Germany and Massachusetts Institute of Technology, Cambridge, MA, USA.

[2] R. W. Beard and T. W. McLain. Multiple UAV Cooperative Search under Collision Avoidance and Limited Range Communication Constraints. In *42nd IEEE Conference on Decision and Control*, December 2003.

[3] R. A. Brooks. *A Robust Layered Control System For A Mobile Robot*. IEEE Journal of Robotics and Automation, 1986.

[4] M. Carter. *Minds and Computers, an Introduction to the Philosophy of Artificial Intelligence*. Edinburgh University Press, 2007.

[5] P. Castillo, R. Lozano, and A.E. Dzul. *Modelling and Control of Mini-Flying Machines*. Springer-Verlag London, 2005.

[6] J. H. Conway and D. A. Smith. *On Quaternions and Octonions: Their Geometry, Arithmetic, and Symmetry*. A.K.Peters Ltd., Natick, Massachusetts, 2003.

[7] A. P. Engelbrecht. *Fundamentals of Computational Swarm Intelligence*. Wiley, 2005.

[8] M. F. Godwin, S. Spry, and J.K. Hedrick. Distributed collaboration with limited communication using mission state estimates. In *American Control Conference, 2006*, June 2006.

[9] W. R. Hamilton. On quaternions, or on a new system of imaginaries in algebra. *Philosophical Magazine*, 1844-1850.

[10] J. Kennedy and R.C. Eberhart. *Swarm Intelligence*. Morgan Kaufmann Publishers, 2001.

[11] R. Kesten. *Data Fusion of Accelerometric, Gyroscopic, Magnetometric and Barometric Information for Attitude and Altitude Estimation of an Unmanned Quadrocopter*. Technical report, Hamburg University of Technology, 2010.

[12] D. Kingston, R. Beardy, and R. Holt. Decentralized perimeter surveillance using a team of UAVs. *AIAA Journal of Guidance, Control, and Dynamics*, 2007.

[13] G. Konecny. *Geoinformation: Remote Sensing, Photogrammetry and Geographical Information Systems*. Taylor and Francis, London, 2002.

[14] K. Kumar and S. Reel. Analysis of Contemporary Robotics Simulators. In *PROCEEDINGS OF ICETECT*, 2011.

[15] Y. Kuwata and J. How. Three Dimensional Receding Horizon Control for UAVs. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, August 2004.

[16] D. Lee and M. Recce. *Quantitative evaluation of the exploration strategies of a mobile robot*. International Journal of Robotics Research, 1997.

[17] Massachusetts Institute of Technology. *Cognitive Robotics (6.834J); Personal Insights in Course Project*, 2004.

[18] J. Nieto, T. Bailey, and E. Nebot. *Recursive Scan-Matching SLAM*. Technical report, ARC Centre of Excellence for Autonomous Systems (CAS) The University of Sydney, NSW, Australia, 2007.

[19] K. Nonami, F. Kendoul, S. Suzuki, W. Wang, and D. Nakazawa. *Autonomous Flying Robots*. Springer Tokyo Dordrecht Heidelberg London New York, 2010.

[20] Office of the Secretary of Defense. *Unmanned Aircraft Systems Roadmap 2005-2030*. Technical report, Office of the Secretary of Defense, Washington D.C., 2005.

[21] O.J. O'Loan and M.R. Evans. Alternating steady state in one-dimensional flocking. *Journal of Physics*, 1998.

[22] W. Ren and R. W. Beard. *Distributed Consensus in Multi-vehicle Cooperative Control*. Springer-Verlag London Limited, 2008.

[23] D. Ribas, P. Ridao, and J. Neira. *Underwater SLAM for Structured Environments Using an Imaging Sonar*. Springer Berlin Heidelberg, 2010.

[24] S. Riisgaard and M. R. Blas. *SLAM for Dummies a Tutorial Approach to Simultaneous Localization and Mapping*. Technical report, Massachusetts Institute of Technology, 2005.

[25] S. Thrun. Simultaneous Localization and Mapping. *Robotics and Cognitive Approaches to Spatial Mapping*, 2008.

[26] C. Tomasi and T. Kanade. Shape and motion from image streams under orthography: A factorization method. *International Journal of Computer Vision*, 1992.

[27] V. K. Tzanov. Distributed area search with a team of robots. Master's thesis, Massachusetts Institute of Technology, 2006.

[28] P. Ulam and R. Arkin. When good comms go bad: Communications recovery for multi-robot teams. In *International Conference on Robotics and Automation*, April 2004.

[29] J. Wang and S. Balakirsky. *USARSim, A Game-based Simulation of Mobile Robots*, v3.1.3 edition.

[30] J. Wendel. *Integrierte Navigationssysteme*. Oldenbourg Wissenschaftsverlag GmbH, 2007.

[31] M Wölfel and H. K. Ekenel. *Feature Weighted Mahalanobis Distance: Improved Robustness for Gaussian Classifiers*. Technical report, Institut fuer Theoretische Informatik, Universität Karlsruhe (TH).

[32] T. De Wolf and T. Holvoet. *Emergence Versus Self-Organisation: Different Concepts but Promising When Combined*. Springer Berlin / Heidelberg, 2005.

[33] M. Zaratti, M. Fratarcangeli, and L. Iocchi. *A 3D Simulator of Multiple Legged Robots based on USARSim*. Technical report, Dipartimento di Informatica e Sistemistica, Universià "La Sapienza", Rome, Italy, 2006.

Webpages

[34] Autodesk. *Autodesk 3ds Webpage*. http://usa.autodesk.com/3ds-max/, 2011.

[35] Autodesk. *Autodesk Maya Webpage*. http://usa.autodesk.com/maya/, 2011.

[36] P. Belanger and K. Biba. *802.11n Delivers Better Range*. http://www.wi-fiplanet.com/tutorials/article.php/3680781, 2007.

[37] University Bremen. *SimRobot - Robotics Simulator*. http://www.informatik.uni-bremen.de/simrobot/index_e.htm, 2010.

[38] eberhard@sengpielaudio.com. *Damping of sound level with distance*. http://www.sengpielaudio.com/calculator-distance.htm, 2011.

[39] Inc Epic Games. *Animation System Overview*. http://udn.epicgames.com/Three/AnimationOverview.html, 2001-2010.

[40] Inc Epic Games. *Physics Asset Tool User Guide*. http://udn.epicgames.com/Three/PhATUserGuide.html, 2001-2010.

[41] Inc Epic Games. *Terrain Design: Guidelines and Information*. http://udn.epicgames.com/Three/TerrainDesign.html, 2001-2010.

[42] O. Formsma, N. Dijkshoorn, S. van Noort, and A. Visser. *usarsim-smoke-fire*. http://code.google.com/p/usarsim-smoke-fire/, 2010.

[43] Willow garage. *OpenCV 2.1 C++ Reference*. http://opencv.willowgarage.com/documentation/cpp/index.html, 2010.

[44] Laptop Geek. *DARPA grand challenge*. http://www.darpagrandchallenge.com/, 2007-2011.

[45] Inc. Geeknet. *Gazebo; 3D multiple robot simulator with dynamics*. http://playerstage.sourceforge.net/gazebo/gazebo.html, 2005.

[46] jonas.witt@tuhh.de Jonas Witt. *TUHH Quadrokopter Projekt*. http://quadro.fst.tu-harburg.de/, 2011.

[47] Active Robots Ltd. *Hokuyo Laser and Infrared Range Finders*. http://www.active-robots.com/products/sensors/hokuyo.shtml, 2003-2011.

[48] RoboCupRescue.org and RoboCupRescue.com. *RoboCup Rescue*. http://www.robocuprescue.org/, 2006-2011.

[49] roboSim@nist.gov. *USARSim Webpage*. http://usarsim.sourceforge.net/wiki/index.php/Main_Page, 2011.

[50] IEEE Robotics and Automation Society. *IEEE/NIST VMAC Competition*. http://www.ieee-ras.org/event/ieee-nist-vmac-competition.html, 2007-2011.

[51] Russell Smith. *Open Dynamics Engine (ODE) Community Wiki*. http://opende.sourceforge.net/wiki/index.php/Main_Page, 2010.

[52] Chemnitz UT. *Simulation of Robots and Sensors*. http://www.tu-chemnitz.de/etit/proaut/forschung/simulation.html, 2011.

[53] webmaster@cyberbotics.com. *Webots: the mobile robotics simulation software*. http://www.cyberbotics.com/, 2011.

[54] XPac27. *UT3 Map Factory*. http://www.map-factory.org/ut3, 2007-2011.

[55] ETH Zurich. *Wind energy fast-response-aerodynamic-probe theologies*. http://www.lec.ethz.ch/research/wind_energy, 2011.

List of Figures

List of Tables

List of Symbols

Abbreviations

AAV	**A**utonomous **A**erial **V**ehicle
AURIS	**Au**tonomous **R**obot **I**nteraction **S**imulation
AUV	**A**utonomous **U**nderwater **V**ehicle
DARPA	**D**efense **A**dvanced **R**esearch **P**rojects **A**gency
EKF	**E**xtended **K**alman **F**ilter
FPS	**F**rames **P**er **S**econd
GPRS	**G**eneral **P**acket **R**adio **S**ervice
NFO	**Inf**o message
ODE	**O**pen **D**ynamics **E**nginey
RANSAC	**Ran**dom **Sa**mpling **C**onsensus
SEN	**Sen**sor message
SLAM	**S**imultaneous **L**ocalization **a**nd **M**apping
STA	**Sta**tus message
UAV	**U**nmanned **A**erial **V**ehicle
UE	**U**nreal Game **E**ngine
USARSim	**U**nified **S**ystem for **A**utomation and **R**obot **Sim**ulation
UT	**U**nreal **T**ournament
UU	**U**nreal **U**nit

WLAN	**W**ireless **L**ocal **A**rea **N**etwork
WPAN	**W**ireless **P**ersonal **A**rea **N**etwork

Variables

$(x_m, y_m)^T$	map coordinates
$(x_q, y_q, z_q)^T$	coordinates in the quadrotor frame
$(x_u, y_u, z_u)^T$	coordinates in the simulated environment
Δt	time between two ticks in the UE
ϕ, θ, ψ	roll, pitch, yaw angle; attitude of quadrotor
ψ_i	angle of found obstacle to quadrotor, retrieved from range scanner
$\vec{q}$	Quaternion
A_k	State transition matrix
B_k	Input matrix
d	distance in m
H_k	Observation matrix
$K_{p,i,d}$	Tuning parameters of PID controller
Q_k	Covariants matrix for the model error in a Kalman Filter
R_k	Covariants matrix for the measurement error in a Kalman Filter
w_f	weight of turning necessity (collision avoidance)
g	standard gravity, $g = 9.80665 \frac{m}{s^2}$
m	mass

A Appendix

A.1 Distance Measures

Mahalanobis Distance In Section 4.4.1 Mahalanobis and Euclidean distance measures are mentioned in context with data association in the SLAM approach.

$$\begin{aligned} d_i &= \sqrt{d_i} \\ &= \sqrt{(x-\mu_i)^T \sum_i^{-1} (x-\mu_i)} \end{aligned} \tag{A.1}$$

sum_i^{-1} being the inverse of the covariance matrix of a class I. The range of variable of the sample point is the expressed by the covariance matrix [31]. Therefore it is a weight, should it be set to one, then the Mahalanobis distance becomes the Euclidean distance.

Euclidean Distance

$$d_i = \sqrt{(x-\mu_i)^T (x-\mu_i)} \tag{A.2}$$

Here μ_i represents the mean vector of class I and x is the sample vector to be classified.

A.2 Quaternion Product

Multiplication of quaternions might not be well known that is why it is mentioned here. It can be thought of as a matrix vector product, and can be represented with a black dot, e.g., $\vec{q} \bullet \vec{r}$.

Given two quaternions $\vec{q}_1 = (w, x, y, z)^T$ and $\vec{q}_2 = (s, u, v, w)^T$ their product is given by:

$$\vec{q}_1 \bullet \vec{q}_2 = \begin{pmatrix} w & -x & -y & -z \\ x & w & -z & y \\ y & z & w & -x \\ z & -y & x & w \end{pmatrix} \begin{pmatrix} s \\ u \\ v \\ w \end{pmatrix} \tag{A.3}$$

A.3 Optical Smoke Detector

The team which devolved the smoke mesh, mentioned in Section 3.2.1, also introduced Matlab-code to measure the smoke density compared to a white wall, as can be seen in Figure A.1. This approach can possibly be adopted to acquire a more realistic smoke sensor [42].

(a) No Smoke **(b)** Smoke

Figure A.1: White screen to measure smoke intensity, courtesy of [42]

A.4 Overview of Robotic Simulators

Table A.1 shows a comparison between different robotic simulators, on account of, cost, transference of controllers to a real robot, compatibility, 3D visualization, multi agent support, physical behavior, and the reconfiguration of robots, hence, of the simulation code.

Simulator	A	B	C	D	E	F	G	H
X-Plane	1994	M	WLM	Y	N	N	N	N
Webots	1996	M	WLM	Y	Y	Y	Y	N
AUV-Workb.	2002	M	WLM	Y	Y	N	N	N
MATLAB	1984	H	WLM	Y	Y	Y	N	Y
MS FlightSim	1982	M	W	Y	N	N	N	N
Actin	2001	L	WL	Y	N	Y	Y	Y
MS RDS	2006	L	W	Y	N	Y	Y	N
USARSim	2003	OS(L)	W(L)	Y	(Y)	Y	Y	Y
OpenSim	2001	OS	WLM	Y	N	N	N	N
ÜberSim	2003	OS	WL	Y	Y	N	Y	Y
Simbad	1981	OS	WLM	Y	N	Y	Y	N
FlightGear	1997	OS	WLM	Y	N	N	N	N
Breve	2008	OS	WLM	Y	N	Y	Y	N
Gazebo	2000	OS	WML	Y	(Y)	Y	Y	Y
SimRobot	1994	OS	WML	Y	N	Y	Y	(Y)

Table A.1: Comparison of different commercial and open-source robotics simulators, courtesy of [14]
A = Year of origin
B = Cost: L=Low, M=Medium, H=High, OS=Open source
C = Platform compatibility: W=Windows, L=Linux, M=Mac OS X
D = 3D visualization: Y=Yes, N=No
E = Controller can be used for a real robot: Y=Yes, N=No
F = Multi agent simulation: Y=Yes, N=No
G = Correct physics: Y=Yes, N=No
H = Reconfigurable: Y=Yes, N=No