TUHH
Technische Universität Hamburg-Harburg

Institute for Software Systems

Realization of a UAV simulation environment for USARSim-UDK

Diploma Thesis

Author:

Paul-David Piotrowski

Supervisors:

Prof. Dr. Ralf Möller
Prof. Dr. Herbert Werner

July 16, 2012

Declaration

I, Paul-David Piotrowski, solemnly declare that I have written this diploma thesis independently, and that I have not made use of any aid other than those acknowledged in this diploma thesis. Neither this diploma thesis, nor any other similar work, has been previously submitted to any examination board.

Paul-David Piotrowski

Contents

List of Figures

List of Tables

Abstract

In this diploma thesis a control program for quadrocopters is implemented based on the new Unreal Development Kit version of USARSim in order to create a simulation environment for UAVs. The purpose of the implemented software is to be helpful in research, development and testing of control algorithms for quadrocopters operating in a three dimensional environment as well as in research of learning algorithms for autonomous quadrocopter behavior. The control program is supposed to act as a framework and a foundation which can easily be extended with new algorithms that can then be tested for functionality in the simulated environment without any risk for actual hardware. Furthermore the implemented software should allow users to focus on the implementation of their algorithms, releasing them of the task to create whole new control programs and deal with low level programming of underlying software layers every time a new algorithm is to be evaluated.

Chapter 1

Introduction

Robotics have been a field of enormous academic interest in which a lot of research has been conducted in the last few decades. Especially smaller robots, so called micro robots, are an interesting tool for research in the academic world for they allow to do research in several disciplines at a relatively low cost when compared to the robots that are available for big commercial companies or institutes. While a lot of this research was focused primarily on wheeled robots, flying robots, so called unmanned air vehicles (UAVs), have gained more and more attention in recent years.

UAVs in contrast to ground based, wheeled robots which operate in an environment that can be mapped into two dimensional space, operate in three dimensional space from which a new layer of difficulties and problems arises. Just like ground based, wheeled robots UAVs can be equipped with a battery of sensors like cameras, gyroscopes, laser scanners or GPS navigation receivers in order to allow for autonomous behavior. Autonomous behavior in wheeled robots has been researched for some time already and has made a lot of progress in the recent decade as could be witnessed in events like the *DARPA Grand Challenge in 2005* where autonomously driving cars have managed to complete the task of driving a 132,2 mile long course trough open terrain in the desert within the given time limit of ten hours without human guidance [1] or the *DARPA Urban Challenge in 2007* in which six autonomous vehicles managed to absolve a city course of 60 miles in under six hours in which they had to interact with other autonomous vehicles and human drivers among other difficult tasks [2]. While autonomous behavior for wheeled ground vehicles has seen a lot of research and has progressed in huge steps in the recent years autonomous behavior for unmanned aerial vehicles is still a relatively young field of research and high academical interest offering a lot of opportunity for scientific work. Many of the algorithms used for autonomous robot navigation on the ground need to be adapted to either operate in three dimensional space or to work sufficiently fast in order to allow for real time control of flying devices. While ground based robots can usually just stop to evaluate their environment and wait until they are done with their calculations and ready to proceed this is usually not possible for air vehicles for which such a behavior surely would result in a crash.

UAVs provide an opportunity for cost reduction in several fields. The U.S. Department of Defense for example calculated that 70% of their non-combat aircraft losses are attributed to human errors, and a large percentage of the remaining losses

have this as a contributing factor. Even though aircraft are modified, training emphasized, and procedures changed as a result of these accidents, the percentage attributed to the operator remains fairly unchanged. UAVs are expected to reduce this percentage significantly for three main reasons. First, UAVs, like for example the Global Hawk, have demonstrated the ability to operate completely autonomously from take-off through rollout after landing. Software based performance, unlike its human counterpart, is guaranteed to be repeatable when circumstances are repeated. Second, the need to conduct training and proficiency sorties with unmanned aircraft actually flying could be reduced in the near term with high fidelity simulators. Third, with such simulators, the level of actual flying done by UAVs can be reduced, resulting in fewer aircraft losses and lower attrition expenditures. The U.S. Department of Defense expects that although some level of actual UAV flying will be required to train manned aircraft crews in executing cooperative missions with UAVs, a substantial reduction in peacetime UAV attrition losses can probably be achieved. [3]

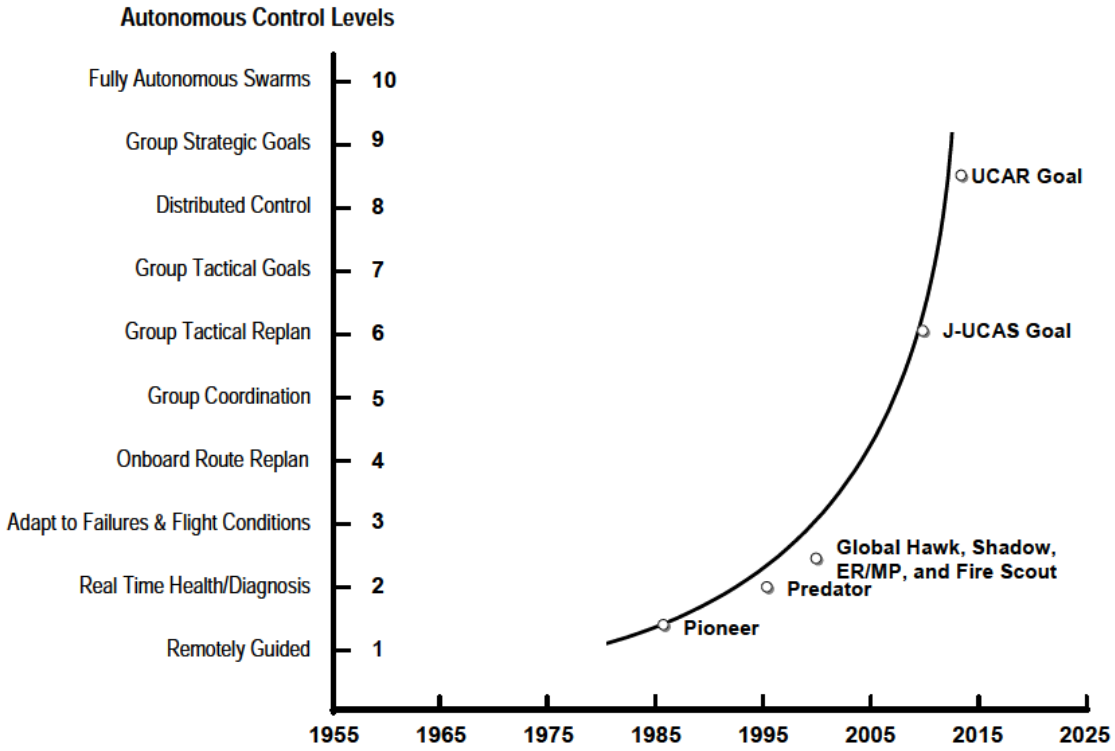

Figure 1.1: Predicted trends in UAV autonomy, courtesy of [4]

Another economic case to be made for UAVs is their typically reduced size compared to manned aircraft. For example in a local surveillance scenario where the task would otherwise be carried out by a light aircraft with one or two aircrew. The removal of aircrew has a great simplifying effect in the design and the reduction of cost of the aircraft. To accommodate two aircrew and their equipment, like seats, controls and instruments typically about 1.2 m^3 are required resulting in a frontal area of about $1.5m^2$. A UAV to carry out the same task would only require $0.015m^3$ for housing an automatic flight control system with sensors and computer, a stabi-

lized high-resolution color TV camera and radio communication links. The frontal area would be merely $0.04m^2$. The masses required to be carried by the manned aircraft, together with the structure, windscreen, doors, frames, and glazing, would total at least $230kg$. The equivalent for the UAV would be about $10kg$. Compared to the UAV the light aircraft has to carry a 220 kg heavier payload and has about 35 times the frontal area. On the assumption that the disposable load fraction of a light aircraft is about 40% and of this 10% is fuel, then its gross mass will be typically of order 750 kg. For the UAV, on the same basis, its gross mass will be of order 35 kg. The reduced size of the aircraft results in lower fuel costs as well as lower hangarage cost while the simpler design of the unmanned aircraft allows for cheaper first costs and lower maintenance costs. [5]

Autonomous UAVs are usually controlled by computer programs. These computer programs can, given enough processing power, be run on board the UAV itself or otherwise on a remote computer system connected to the UAV via a wireless communication link which provides enough bandwidth for all sensory information to be transmitted sufficiently fast to the controlling ground station and control commands to be transmitted back to the UAV. Although on-board processing and decision making is the more autonomous approach and therefore more desirable, todays UAVs are usually not capable of processing all their sensory data by means of on-board computers and programs and making difficult decisions in critical situations based only on such calculations [4]. Today usually a hybrid approach is used, letting the UAV do the easy processing and decision making tasks with its on-board capabilities while heavy computational tasks like processing and evaluation of high resolution video footage is usually done in a ground station, which receives the data over a powerful enough downlink [4].

UAVs can be equipped with additional communication capabilities allowing them to communicate among each other and thus making it possible for them to group and create swarms of UAVs. As can be seen in figure 1.1 fully autonomous self coordinating swarms of UAVs are treated as one of the highest and most advanced levels in UAV autonomy which has not been reached yet and is one of the future goals of the U.S. Department of Defense.

As stated before simulation can be very helpful in the task of training personnel for UAV missions but simulation is not restricted to human training alone. It has a long-standing tradition in robotics as a useful tool for testing ideas on virtual robots in virtual settings before trying them out on real robots [6]. Due to the increase in the computational power of computers, which made it possible to run computationally intensive algorithms on personal computers instead of special purpose hardware, and the increased effort of the game industry to create realistic virtual environments in computer games, it has been getting even more attention in the last decade [6]. Since the goals targeted by the game industry and the requirements for robot simulation lie so close together, game engines and game technology is used for robot simulation more often, recently.

Taking into consideration the costs for the actual hardware needed to carry out

experiments with UAVs and repairs due to crashes, the time spent constructing experiments and the restrictions imposed by the number of functional UAVs available, possibly forcing research staff to work on available hardware in a schedule like manner, the idea to have a simulation environment for UAVs becomes very enticing. In a simulation experiments can be carried out repeatedly in the exact same manner, failed experiments and crashes do not result in expensive repairs and loss of time, while, given powerful enough computers, several experiments can be done in parallel and virtual hardware and environments can be replicated as often as necessary to provide everyone willing with material to work with. While humans can be trained in a simulation environment to control UAVs, so can computers utilizing machine learning algorithms. The simulation's task is to provide an environment which is close enough to reality that learning and testing of different control algorithms in the simulation leads to comparable results as would have been achieved in a real world environment. While high fidelity simulations have been vastly expensive and not widely available in the past this has changed in recent years with the advent of the modularized game engines, mentioned earlier. These are highly popular for game development and are capable of generating high fidelity graphics and doing highly accurate physical calculations on consumer hardware computers. Due to their popularity these game engines come at a much more affordable price and in some cases even for free, providing a cost efficient alternative for robot simulation, further increasing the financial advantage of simulating robots over having to work exclusively with real hardware.

In the course of this diploma thesis a UAV simulation environment will be presented based on a simulation of urban search and rescue robots and environments which again itself is based on a popular 3D game engine. The UAV simulation environment is supposed to act as a framework to enable an easy implementation of UAV control algorithms and learning algorithms which can then be tested and evaluated in the simulated environment.

Chapter 2

Fundamentals

2.1 Quadrocopters

The kind of UAVs this diploma thesis focuses on are small quadrocopters which due to their little size are often categorized as micro unmanned air vehicles (MUAVs). Quadrocopters are aircrafts with four propellers mounted on a frame construction which have flying capabilities that are comparable to that of helicopters.

Figure 2.1: Apollo quadrocopter, Hamburg University of Technology, courtesy of [7]

The basic structure of a quadrocopter is depicted in figure 2.1. They are capable of vertical take-off, hovering in midair and fast maneuvers like flips or loopings. In addition to their flying capabilities come their, compared to a helicopter, much simpler and therefore massively less sensitive mechanics [8]. These capabilities together with their usually small size and relatively low costs make these aircrafts attractive for researchers in the academic field. Scientists have been doing research in machine learning [9], autonomous navigation of UAVs [10], swarm behavior [11] and other fields with this kind of devices. Aside from the scientific field quadrocopters are also used in the military for surveillance, in the civil area for cartography, traffic monitoring, observation of big crowds on popular events and by hobbyists around the world for aerobatics, to name just a few applications.

A quadrocopter can be configured to fly in one of two orientations. The first one is the so called *x mode* where two rotors are placed at the front of the quadrocopter and two at the rear, the second one is the so called *+ mode* in which one rotor is positioned in forward, one in backward direction and one in each right and left direction. The two modes are depicted in figure 2.2. In the following description we will assume the quadrocopter to fly in *x mode*.

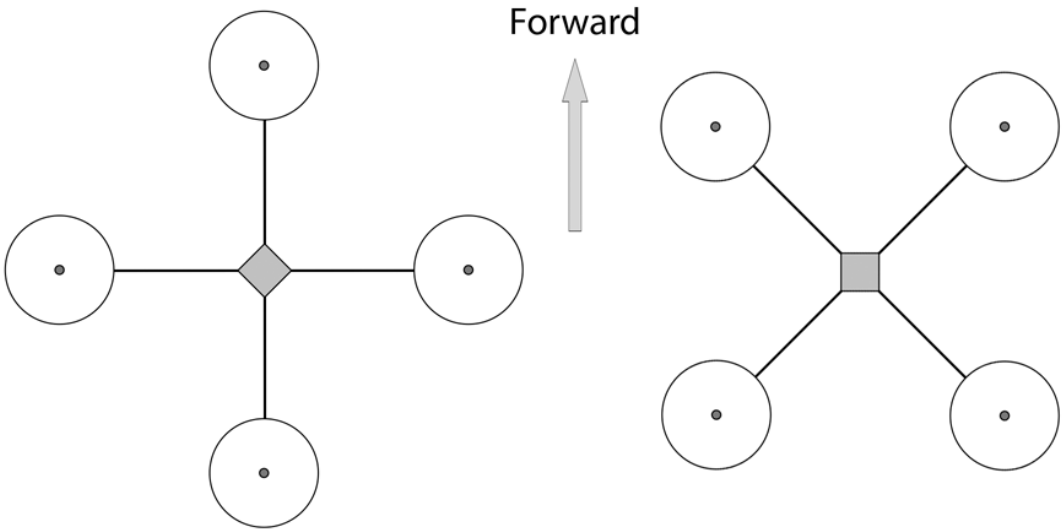

Figure 2.2: A quadrocopter oriented in + Mode *(left)* and x Mode *(right)*

2.1.1 Coordinate Systems

As depicted in figure 2.3 a quadrocopter can either be looked at from an inertial coordinate system or from the vehicle's local coordinate system. Any vector can be converted between the two coordinate systems by the help of rotation matrices R_ϕ, R_θ, R_ψ which can be applied separately for the rotation around each angle or combined into one rotation matrix R_{OV} rotating the vector around all angles at

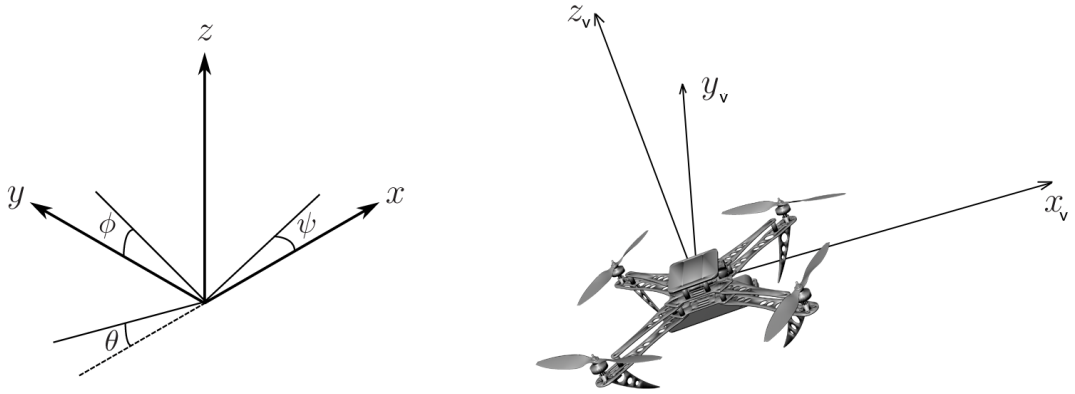

Figure 2.3: Inertial and local coordinate system

once. The conversation from the inertial coordinate system to the local coordinate system can be done by the combined matrix R_{OV} given below.

$$R_{OV} = R_\phi R_\theta R_\psi =$$

$$\begin{bmatrix} cos\theta cos\psi & sin\phi sin\theta cos\psi - cos\phi sin\psi & cos\phi sin\theta cos\psi + sin\phi sin\psi \\ cos\theta sin\psi & sin\phi sin\theta sin\psi + cos\phi cos\psi & cos\phi sin\theta sin\psi - sin\phi cos\psi \\ -sin\theta & sin\phi cos\theta & cos\phi cos\theta \end{bmatrix}$$

Since these matrices are rotation matrices the back conversion of a vector from the vehicle's coordinate system to the inertial system can be done by the help of the transposed of the matrix used before so that

$$R_{VO} = R_{OV}^{-1} = R_{OV}^{T}$$

2.1.2 Quadrocopter Movement

A quadrocopter being an air vehicle has six degrees of freedom. It can carry out translational movements which is moving up/down, left/right and forward/backward and it can carry out rotational movements as well, which would be roll, pitch and yaw as is depicted in figure 2.4. The four rotors each create a force and a torque on the quadrocopter's frame and are the only means by which the quadrocopter is able to move.

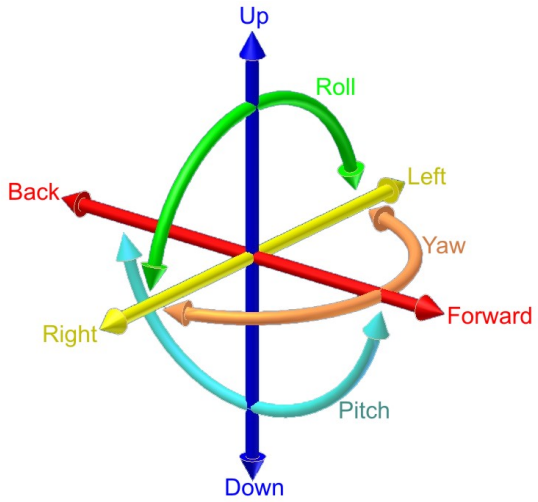

Figure 2.4: Six degrees of freedom, curtesy of [12]

In the following overview of quadrocopter movement we will assume that the quadrocopter's center of mass lies in the middle of the quadrocopter when viewed from top and in the same plane as the four rotors and we will not take into consideration any aerodynamic or other effects that might in reality have an additional impact on the quadrocopter.

Vertical Movement

The four rotors create a collective thrust in negative direction of z_v. As long as the quadrocopter's coordinate system's z-axis is oriented along the inertial coordinate system's z-axis and the the speeds of all four rotors are the same, upward and downward movement is easily accomplished by increasing or decreasing the rotor speed of all four rotors simultaneously by the same amount. If the sum of the forces generated by the rotors exceeds the gravitational force mg the quadrocopter moves upward, if it falls below the force mg the vehicle moves downward. If the four forces sum up to match mg exactly the quadrocopter will hover in the air. As long as the forces F_i are all of the same size the moments generated by them will neutralize each other since they are working against each other which results in a rotational acceleration of 0.

Horizontal Movement

In order to accomplish horizontal movements the quadrocopter needs to be tilted into the appropriate direction first, so that as a result the four forces created by the rotors are also tilted and can be decomposed into a horizontal and a vertical component. While the vertical component needs to compensate the force created by the quadrocopter's weight and thus keeps the quadrocopter from falling down the horizontal component results in a horizontal movement in the direction of the quadrocopter's tilt as can be seen in figure 2.5.

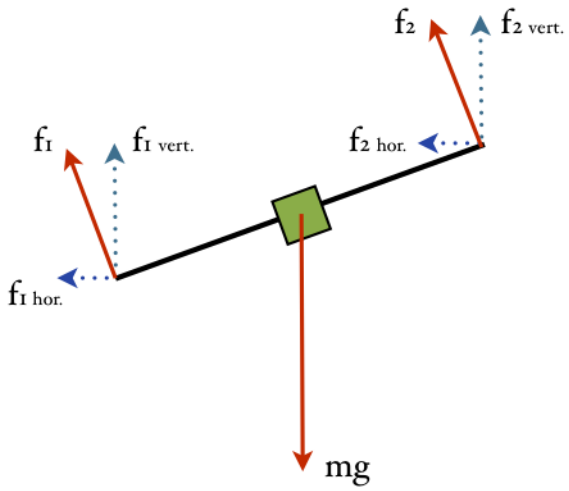

Figure 2.5: Horizontal movement of a quadrocopter

Pitch and Roll

To get the quadrocopter to pitch both front rotors need to create an equal amount of thrust while both rotors at the back also need to create an equal amount of thrust

which needs to be different from the thrust of the front rotors though.

$$F_{front} = F_1 = F_2,$$
$$F_{back} = F_3 = F_4,$$
$$F_{front} \neq F_{back}$$

As a result the moments in roll direction will sum up to zero while the moments in pitch direction will be nonzero and the quadrocopter will begin to pitch. Analogous to this the quadrocopter can be made to roll by letting the rotors on the right each generate the same thrust and the ones on the left side generate an equal thrust that is different from the thrust generated by the right rotors.

$$F_{right} = F_2 = F_3,$$
$$F_{left} = F_1 = F_4,$$
$$F_{left} \neq F_{right}$$

This time the moments in pitch direction will sum up to zero while the sum of moments in roll direction will be nonzero resulting in the quadrocopter beginning to roll.

Yaw Movement

Yaw movement is created in a different way. In order to compensate the torque resulting from the rotors' rotation two of the rotors are rotating clockwise while the two remaining ones rotate in the opposite direction. Usually rotors rotating in the same direction lay opposite to one another as is depicted in figure 2.6. Assuming an

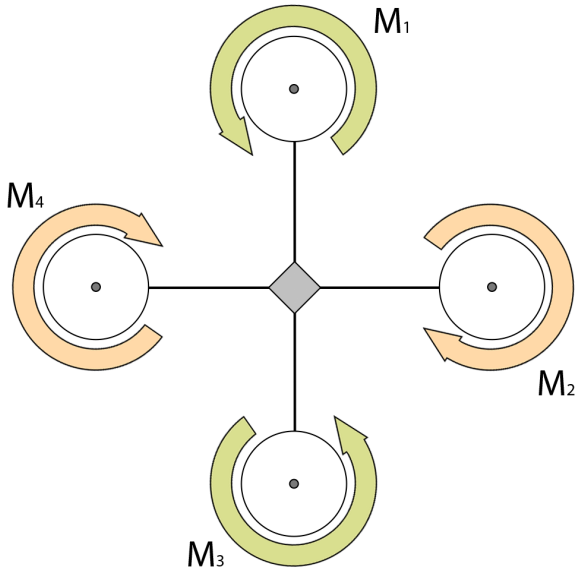

Figure 2.6: Rotor arrangement

arrangement of rotors as depicted in 2.6 the overall torque sums up to

$$M_{total} = \sum_{i=1}^{4} M_i = (M_1 + M_3) - (M_2 + M_4)$$

As long as M_{total} equals zero there will be no yaw movement. But as soon as M_{total} is nonzero yaw movement is generated.

Stabilization

Since a human being would not be able to easily stabilize and fly a quadrocopter without additional sensor support quadrocopters usually have an on-board controller taking care of stabilizing the vehicle [8]. To achieve this a quadrocopter is usually equipped with a controller board that contains sensors like gyroscopes and acceleration sensors. The controller board usually implements a PD or PID controller which generates appropriate control signals for the four rotor engines to stabilize the quadrocopters flight. A human with a remote control usually only tells the quadrocopter to roll, pitch or yaw and to increase or decrease overall thrust generated by the four rotors. He is not supposed to control the quadrocopter's four rotors directly. The on-board controller will then translate the operator's commands into appropriate rotor speeds.

2.1.3 Sensors

Quadrocopters can be equipped with a lot of different sensors which serve different purposes. In order to make flight stabilization possible gyroscopes and acceleration sensors are commonly used which usually are relatively cheap and lightweight. Given these sensors a quadrocopter can determine its attitude as well as acceleration and take measures to achieve a stable position.

Sensors like different kinds of cameras or measurement instruments like Geiger counters, thermometers or humidity sensors which are not needed for the quadrocopter's functionality itself can be added to a quadrocopter to make it possible to fulfill certain mission goals like providing live video streams or measured data from a certain destination area the quadrocopter is supposed to operate in. Cameras can also be used to remotely operate a quadrocopter independent of the fact if the UAV is in sight of the operator or not, increasing the quadrocopter's range of operability and flexibility [13].

Aside from sensors for flight stabilization and information gathering quadrocopters can also be equipped with sensors that enable them to perceive their environment in order to allow for autonomous behavior. Laser scanners have been used to allow quadrocopters to perceive obstacles and draw maps of their environments [10], ultrasonic sensors can be used to implement autonomous obstacle evasion behavior while wireless communication devices can be used for communication between quadrocopters and the internal organization of a swarm of quadrocopters [11]. In order to allow a quadrocopter to to sense its position a GPS receiver or some other

kind of positioning system hardware can be carried by the quadrocopter.

With all the different kinds of sensors available it is necessary to remember that a quadrocopter can only carry a limited amount of payload and in order to not overload it, it might be necessary to consider the advantages and disadvantages of different sensor combinations in respect to each other and the necessity of each sensor taken into consideration. Further on more payload results in a greater mass which has to be lifted into the air by the quadrocopter. This will require a higher thrust generated by the quadrocopter's rotors which in turn means that a quadrocopter needs to consume more energy and therefore will drain its batteries more quickly, eventually decreasing its employment radius.

2.1.4 Example: AirRobot AR 100-B

The AirRobot AR 100-B is an unmanned aerial vehicle with autonomous flight and navigation capabilities produced by the company AirRobot. Its primary applications are reconnaissance, surveillance, search and rescue missions, intelligence, documentation and inspection [13]. It is capable of autonomously holding its position and hovering over a place either via GPS or optical position lock while the optical positioning system can keep the UAV in place even in areas where a GPS signal is not accessible [13]. Stabilization is achieved with the help of gyroscopic, barometric and magnetic sensors [13]. The AirRobot AR 100-B can be controlled remotely via a life video feed which makes it unnecessary for the vehicle to be in direct sight of the operator, the operator can remotely operate and fly the UAV from a computer seeing through the camera as if sitting in the cockpit [13].

Figure 2.7: AirRobot AR 100-B [13]

The payloads available for the AR 100-B are build in a modular fashion to allow for a fast change of equipment. At the moment AirRobot provides a daylight video camera, a low light black and white camera, a 10 MP still camera and an infrared camera for their vehicle. All cameras can be tilted up to 100 degrees and can also look straight down.

The German army tested the AirRobot AR 100-B in Australia for its capabilities in desert environments and is now using it in Afghanistan as a reconnaissance device [14]. The AR 100-B has also been used in forest fire detection, mine eviction and gas leak detection among other applications [14]. Also autonomous UAV swarms have been created based on AR 100-B quadrocopters which were modified to carry a smart camera, that controlled the UAV via its serial interface and provided basic autonomous capabilities like take-off, landing and GPS waypoint following [11].

The numbers in the following table are taken from the manufacturer's specifications and describe some of the vehicle's key features.

Max. Ceiling	1000m
Endurance	<30 min
Max. Payload	200g
Deployable Radius	500m limited by analogue video signal 1500m limited by digital video signal
Max. Wind Load	8 m/s
Max Airspeed	50 km/h
Gross Weight	1kg
Diameter	1m

Table 2.1: AR 100-B properties [13]

2.1.5 Example: Microdrones md4-1000

Another example for a mature quadrocopter system being available on the free market is the md4-1000 quadrocopter manufactured by the Microdrones GmbH. It has been designed for tasks in the field of documentation, coordination, exploration, surveying, communication, inspection and observation [15]. Just as the AirRobot AR100-B it has a GPS receiver integrated, provides modularized payloads, which are basically the same as for the AR100-B, and can be remotely operated either via line of sight or a live video feed coming from the quadrocopter's camera. On top of these capabilities it provides a system to autonomously navigate between waypoints which can be set by the operator of the vehicle so that its operational range is not limited by the range of its communication interfaces [15].

Figure 2.8: Microdrones md4-1000 [15]

Max. Ceiling	1000m
Endurance	<70 min
Max. Payload	1200g
Deployable Radius	500m controlled remotely up to 40km when following waypoints
Max. Wind Load	12 m/s
Max Airspeed	54 km/h
Gross Weight	2.65kg
Diameter	1.03m

Table 2.2: Microdrone md-1000 properties [15]

As table 2.2 shows, although the md-1000 has almost the same dimensions as the AR 100-B it has a higher weight, can carry heavier loads, has a higher deployable radius and can stay in the air for a longer time. Since the md-1000 can carry relatively heavy camera equipment it is also used for panorama and landscape photography as well as aerial filming for cinematic purposes and special effects [15].

2.2 Simulation

Simulation makes it possible to test control algorithms and system behaviors of machines without the need for a lot of expenses for the actual hardware that is simulated. While a real system does only exist once, a system in a simulation can be reproduced several times if needed, allowing for economies in otherwise particularly expensive areas such as multi-robot control. Furthermore a lot of simulations can be run in parallel if the needed processing power is available, possibly speeding up research and development processes. On top of this simulated hardware does not need to be repaired, maintained or stored. A crash of a prototype system in real life could mean hours of repairs during which the system is unavailable for further tests and development, resulting in high costs. A crash in a simulation though needs only a hand full of mouse clicks and the simulation can be started anew. This can be especially advantageous in the case of machine learning where new algorithms might not always act as expected or where they are expected to produce crashes in the beginning before converting more or less slowly to a desired behavior. In a simulation a machine can learn the basics of how to behave and to get along well within its environment. After that the algorithms developed in the simulation can be applied to a real machine letting it learn the remaining details in the real world. In some simulations it is also possible to adjust the speed of time and thus let a system learn much faster than it ever could in a real situation.

2.2.1 Unreal Engine

Typically real-time 3D simulations have always been difficult, time consuming and expensive to build and require specialized hardware and personnel [16]. The cost of developing ever more realistic simulations has grown so huge that even game developers can no longer rely on recouping their entire investment from a single game [16]. This has led to the emergence of game engines - modular simulation code - written for a specific game but general enough to be used for a family of similar games [16]. The Unreal Engine is such a game engine developed by Epic Games. Its first version was released in 1998 with the popular first person shooter *Unreal Tournament*. Since then many games have been developed based on this game engine like *Bioshock*, *Deus Ex* and a host of other major titles, and the engine itself has been ported to several different platforms. It is now available for Windows, Linux, Mac OS X, iOS and other operating systems. The engine is capable of creating high quality 3D environments and makes use of nVidia's PhysX engine in order to calculate physical effects. At the moment of writing the Unreal Engine is available in version 3 while version 4 is still under development and has not yet been officially released. The Unreal Engine's core is written in the C++ programming language but a lot of content is also written in the engines own object oriented scripting language, namely Unreal Script, which also allows for users to create additional content and game-play items on their own.

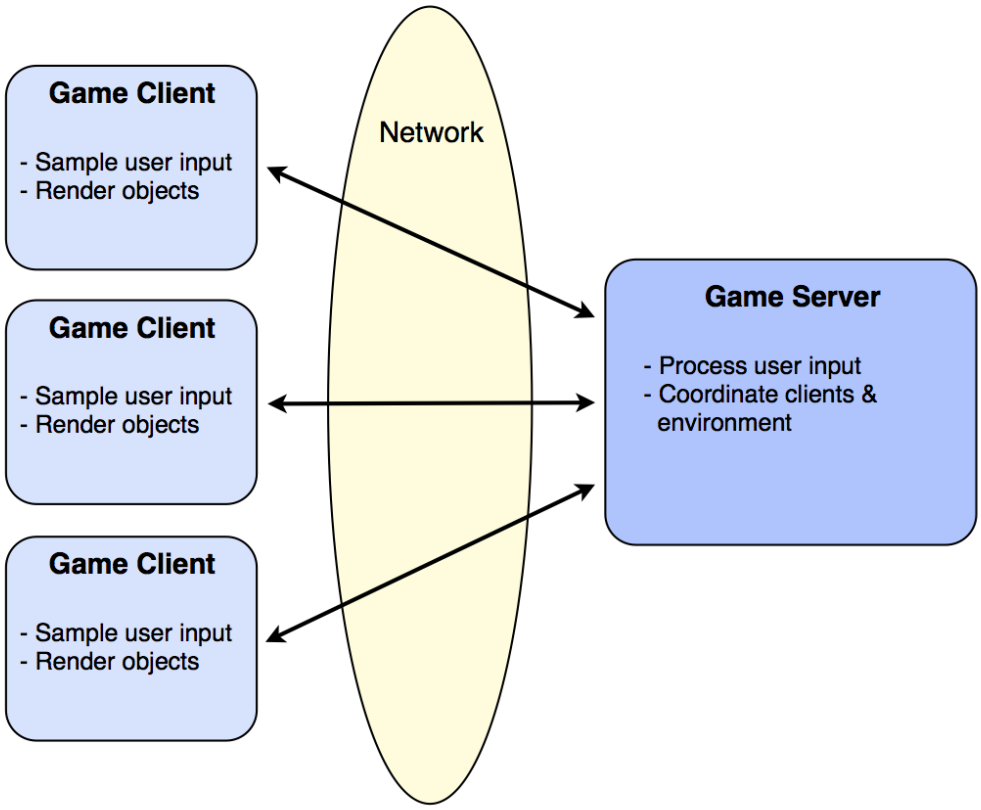

Figure 2.9: FPS Client-Server Architecture

Multi-player first person shooter games like Unreal Tournament use a client server architecture in which each player takes the role of a client. The fast rendering of graphics is usually done on the client-side while the server is responsible for coordinating the different players and environmental interactions as is depicted in figure 2.9.

Unreal Development Kit

In 2009 Epic Games released the first beta version of the Unreal Development Kit which is a free version of their engine that is available to the general public. It contains the Unreal Editor and other utilities which can be used to create new content like maps, models or entire new games for the engine and comes with a huge collection of art and sound assets including textures, 3D models and animations. If the contents or games built with the UDK are not commercial ones then the UDK can be used free of charge. Recently new beta versions of the Unreal Development Kit have been released on a monthly schedule and , according to Epic Games, have been downloaded more than 1.5 million times as of June 2012.

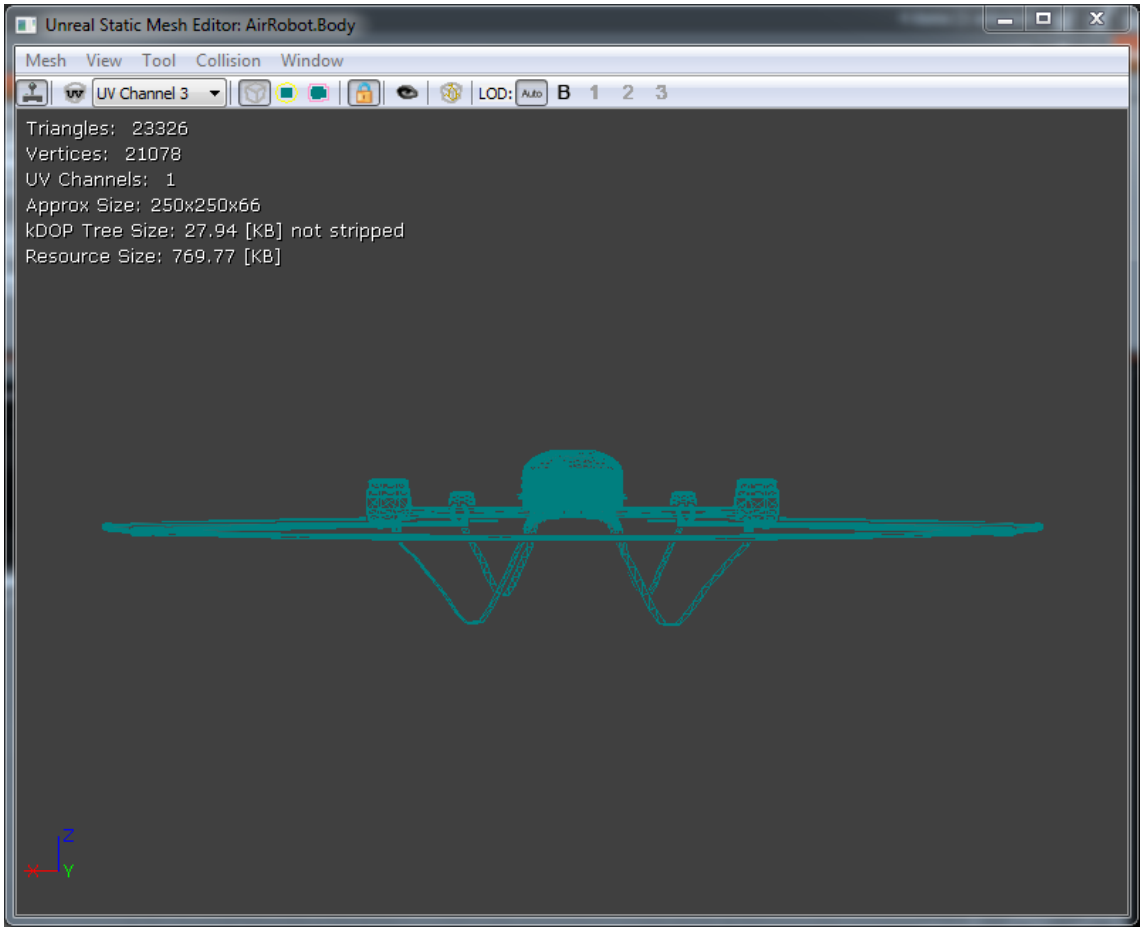

Figure 2.10: A quadrocopter mesh in the UDK editor

Unreal Script

Unreal Script is the scripting language of the unreal engine, which allows for content creation and modification of game logic. It was created to provide the Unreal development team and third party Unreal developers with a powerful, built-in programming language that maps naturally onto the needs and nuances of game programming. Its original design was based entirely on the Java programming language, therefore its syntax strongly resembles Java and C++. Unreal Script is an object oriented programming language which supports inheritance, interfaces and operator overloading and provides a pointerless environment with automatic garbage collection. [17]

Time critical and low level functions in the Unreal Engine are usually written in the C++ programming language, for compiled C++ code runs much faster than Unreal Script code. Other, higher level and more abstract functionality is often implemented in Unreal Script only. The use of C++ functions from within Unreal Script is allowed though. These functions are referred to as native code then.[17]

In order to manage time in the simulation, Unreal divides each second of game-

play into so-called "ticks". A tick is the smallest unit in which all actors in a level are updated and typically takes between 1/10th and 1/100th of a second. The tick time is limited only by CPU and GPU power. The faster the machine the lower the tick duration is. Unreal Script classes which can be added to the simulation contain a tick-function which is executed every tick. All motion controlling code of such classes needs to be placed inside the tick-function. While all Unreal scripts are executed independently of each other and each actor in a level seems to have its own thread, internally Unreal does not use Windows threads for efficiency reasons. Instead threads are simulated by Unreal Script which is transparent for Unreal Script code but becomes apparent when writing C++ code that interacts with Unreal Script. [17]

2.2.2 USARSim

USARSim is an acronym for *Unified System for Automation and Robot Simulation* [18] or also *Urban Search And Rescue Simulation* [18]. It was designed as an open source high-fidelity simulation of robots and environments based on the Unreal Tournament game engine. It's intended as a general purpose research tool with applications ranging from human computer interfaces to behavior generation for groups of heterogeneous robots. In addition to research applications, USARSim is the basis for the RoboCup rescue virtual robot competition as well as the IEEE Virtual Manufacturing Automation Competition (VMAC). [18]

USARSim loads off the most difficult parts of simulation to the Unreal game engine, so that the developers behind the project and users can concentrate more on the robot-specific tasks of modeling platforms, control systems, sensors, interface tools and environments [18]. The 3D rendering and physics calculations are all handed by the underlying Unreal Engine. USARSim itself provides several legged and wheeled robots, aerial robots and even submarine robots, as well as a battery of sensors and actuators and virtual environments, i.e. maps, that the robots can be placed in.

The sensors provided by USARSim can be generally divided into three categories. First there are proprioceptive sensors, including battery state and headlight state, second position estimation sensors, including location, rotation, and velocity sensors and third there are perception sensors, including sonar, laser, sound, and pan-tilt-zoom cameras [18]. USARSim defines a hierarchical architecture for sensor models as well as for robot models. A sensor class defines a type of sensor and every sensor is defined by a set of attributes stored in a configuration file. Perception sensors, for example, are commonly specified by range, resolution, and field of view. Beyond that USARSim provides users with the capability to build their own sensors and robots.

USARSim is available in an older version for Unreal Tournament 2004, in an Unreal Tournament 3 version and is right now in the process of being ported from the Unreal Tournament 3 system to the newer free Unreal Development Kit. Since this

process is not finished yet and the UT3 version of USARSim is not compatible with the UDK version, the UDK version of USARSim does not yet support all Robots, Maps and Sensors the previous versions of USARSim supported and models, maps and sensors from the previous version of USARSim are usually not usable with the new UDK version.

USARSim provides a socket based interface which is based on *Gamebots* [19], a modification for Unreal Tournament, through which it is possible to communicate with the simulated robots directly, bypassing the Unreal Client. This also enables controller applications to reside on different computers than the Unreal Engine thus allowing to control a virtual robot over a network connection. A sketch of this architecture can be seen in figure 2.11. A brief overview of *Gamebots* is provided in [20]. The communication protocol implemented by USARSim is based on simple text messages being sent between the controller application and USARSim. All this enables users to develop and test their own control programs and user interfaces without limitations in programming language and operating system.

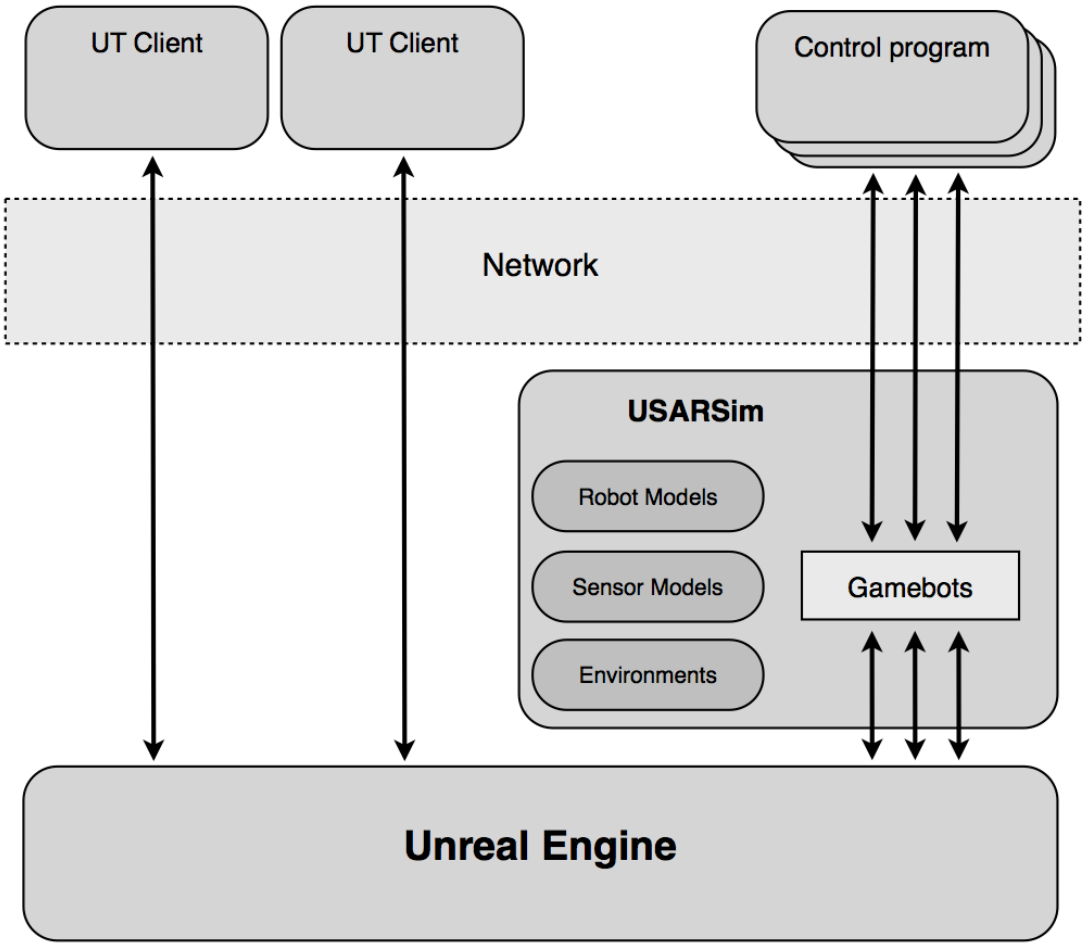

Figure 2.11: USARSim architecture

USARSim is an open source project licensed under the *Gnu Public Licence* and is freely available on the internet.

Chapter 3

UAV Scenarios

In this chapter some possible applications for UAVs in general and for quadrocopters in particular will be presented. General applications for single UAVs will be discussed, showing the versatile possibilities for quadrocopter applications, and where possible and useful extended to multi UAV scenarios. Any of the presented scenarios in this chapter can be implemented in the simulation environment developed in this diploma thesis if proper agents are provided for quadrocopter control and if any additionally needed sensors are attached to the quadrocopter model.

3.1 Search

A common scenario for robots and UAVs is the search scenario in which the robot or the UAV is supposed to search for a certain object in a given area. If the object has been found the UAV is either supposed to report to the base station where the object has been found and then hover over the object's position or return right away to the base station. As with other robot scenarios a UAV needs tor return to its base station regularly in order to refuel or exchange its batteries. This is an important limiting factor for the UAV, for it can possibly not reach every location in the search area in one visit and still return safely. Furthermore it will be able to only search smaller parts of the search area in one visit if these are further away from the base station since reaching them will already consume some part of the UAVs available energy resources.

A UAV can in general take two approaches towards searching a given area. First, it can move in the search area in a random pattern. Given enough time the whole search area will be searched by the UAV eventually. This approach has some disadvantages, though. Assuming that the random search will be interrupted by a battery event when the UAV needs to return to the base to recharge, it becomes less likely for the UAV to visit the areas which are farther away from its starting point, because every time when recharged the UAV will begin its random search at the part of the search area which is closest to its base station, possibly taking a turn before reaching a more distant part, thus depleting its batteries before having reached the more distant areas after several turns. Another disadvantage of this approach is that while more distant parts of the search area are harder to reach other parts of the search area are overflown multiple times where a UAV crisscrosses

its own path.

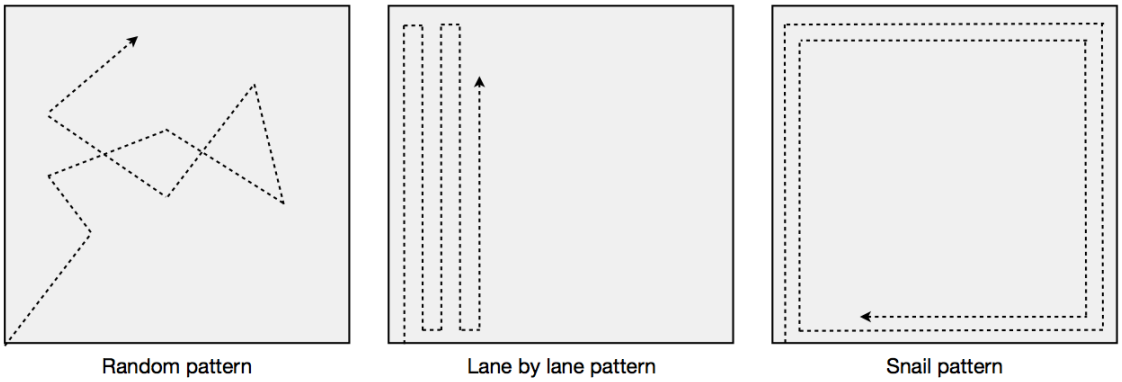

Figure 3.1: Different search patterns

A better approach towards searching in an area is to search the area systematically. This could for example be done in a lane by lane fashion where the UAV flies the search area up and down in lanes which are displaced a little bit every time so that eventually the whole area has been covered. A snail like pattern is thinkable too, so that the UAV flies around the perimeter of the search area and after every completed round decreases its distance to the area's center a little. These systematical approaches have the advantage that less area is covered multiple times and more distant areas will be reached in a predefined time, not depending on random decisions. When searching an area in a lane by lane fashion as stated by [21] the time needed to cover the entire search area consists of the time needed to travel along the rows plus the time needed to turn around at the end of the rows. Although different sweep directions result in rows of approximately the same total length there can be a large difference in turns required [21] as illustrated in figure 3.2. Helicopter and quadrocopter turns take a significant amount of time [21], so in order to decrease search time a lane pattern should be chosen that yields a relatively small number of turns.

In order to increase efficiency even more the ways covered by the UAV in order to return to its base-station and refuel can be taken into consideration, too, when planning the route the UAV is to take in the search area. One could prevent the UAV from overflying the same areas multiple times when returning to the base-station or flying to the place where the last search was interrupted.

While ground based robots can only operate in a more or less two dimensional area, UAVs can operate in three dimensions providing for other opportunities than for ground vehicles. If for example a UAV was to search a given area with the help of a video or infrared camera, it could increase its cruising altitude in order to cover a bigger part of the search area in one sweep thus trading image resolution for amount of area searched, or in other words quality for quantity. This could speed up search drastically. A UAV could fly so high it may not immediately recognize the object it is looking for as what it is but low enough to not accidentally miss it.

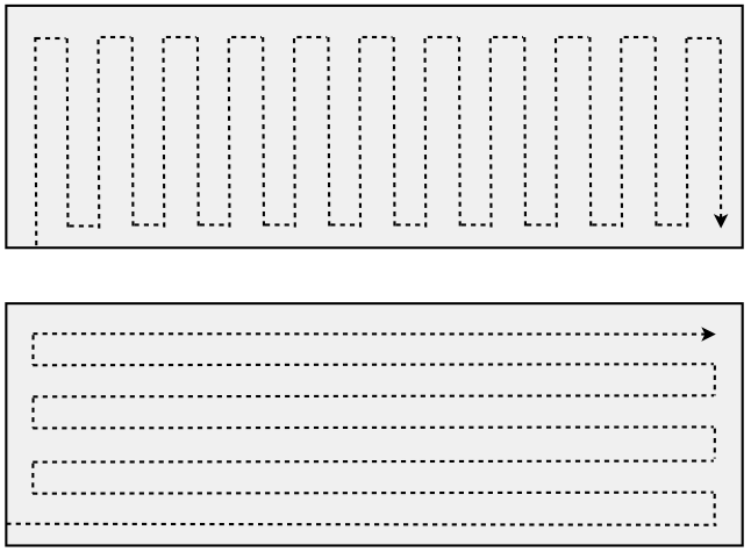

Figure 3.2: Different lane orientation

Has it discovered some suspected object it can decrease its flying altitude and take a closer look at the object to decide whether its the object that it has been looking for.

A search scenario can also be executed by several UAVs simultaneously. Although each of the UAVs could behave independently the way described above, either searching in some random or systematical pattern, in order to reduce search time and increase search efficiency a coordinated approach is to be preferred. The coordination of the UAVs could take different forms.

The search area could be split up into several parts, one for each of the UAVs participating in the search. This would allow for independent search to be carried out by all participating UAVs without the drawback of the same area being unknowingly searched multiple times by different UAVs. The size of the parts could be weighted depending on the distance of the respective part from the base-station. More distant parts could be made smaller than parts closer to the base-station in order to compensate the distance covered by the UAVs to get to their parts, allowing for all UAVs to fly the same total distance in the end, and finish their search simultaneously.

UAVs could also search the area in a schedule like fashion. They could form pairs or bigger units if needed so that if one member of the unit needs to return to recharge another one can take up its colleaques search, thus allowing for a defined constant number of UAVs permanently present in the search area.

One more way of performing a search in a group of UAVs would be to fly in a formation. In order to cover as much of the area at a time as possible the UAVs would be supposed to fly side by side, possibly in a displaced fashion, this way overseeing n times the area of a single UAV in one sweep, where n is the number of UAVs participating in the formation.

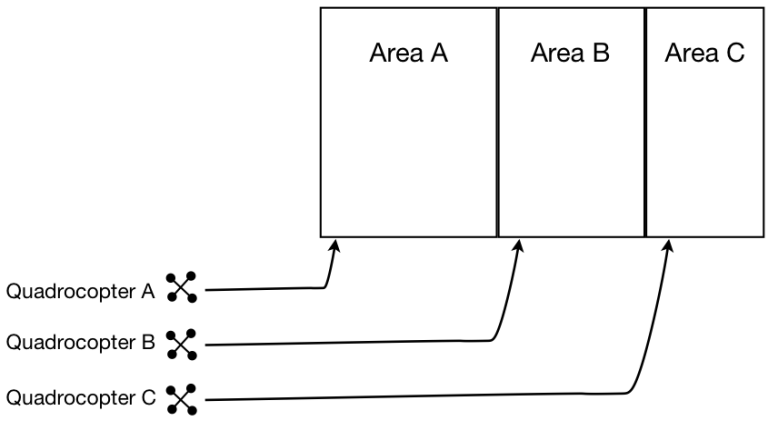

Figure 3.3: Partitioning of a search area considering distance to base station

The principles of the search scenario can be applied to a couple of other scenarios, too. *Area coverage* scenarios for example make use of the same considerations as just presented for the search scenario. While in a search scenario the search ends if the object searched is found, or is at least interrupted, in many area coverage scenarios like for example crop spraying or unmanned aerial mapping there is no such interruption. The UAV is supposed to cover the entire area once to either spray all plants in the field with a given pesticide or in order to take photos of every part of the area so that they can later be stitched together providing a high resolution map of the ground.

3.2 Providing Communication Services

Another application scenario for UAVs and quadrocopters is the provision of communication services. For example as proposed in [22] UAVs could be used to provide satellite coverage in urban areas with shadowing reducing visibility time or in disaster areas where terrestrial communication infrastructure is unavailable or interrupted. The UAV would be equipped with a link to a satellite on the one side and on the other provide for example a wireless network access point and act as a proxy for connected devices [22]. In a similar fashion a UAV could also provide a local communication service acting as a router, bridging the distance between more distant communication partners as depicted in figure 3.4.

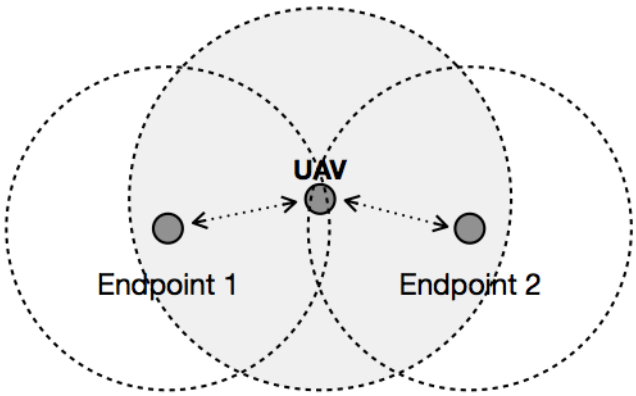

Figure 3.4: UAV acting as a router for two communication endpoints

UAVs could also act as airborne GPS pseudo satellites to provide a stronger and possibly more accurate GPS signal which is harder to jam than satellite based GPS as proposed in [3].

While this appears as a simple task for a single UAV, for it only needs to place itself in an advantageous position between the two participating communication parties things get more interesting when considering multiple UAVs which are to provide an entire area with a communication infrastructure. The UAVs could be equipped with small wireless network devices like for example the IRIS node manufactured by the company Crossbow Technology [23]. This network node is so small and lightweight it could be easily carried by a quadrocopter like the AR 100-B. The sensor nodes could then be utilizing a routing protocol like Dynamic Source Routing (DSR) [24] to create a mobile ad hoc network and provide a wireless communication network to a certain area without any preexisting infrastructure required. DSR allows the created network to be completely self-organizing and self-configuring without the need for administration [24]. The number of UAVs needed to cover a certain area would depend on the area size and the range of the communication devices placed on-board the UAVs. Assuming a communication range of 300 meters a group of 9 quadrocopters would easily be able to completely cover an area of one square kilometer with a wireless network as depicted in figure 3.5.

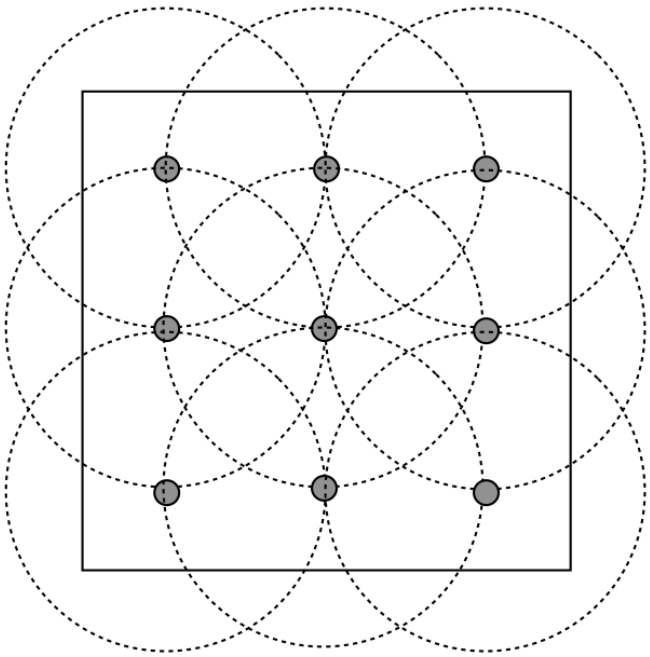

Figure 3.5: Covering an area with a wireless network

As in the search scenario the UAVs would need to return to their base station from time to time to recharge their batteries. In order for the MANET to maintain full coverage of the desired area additional UAVs are required to take the place of their colleagues returning to the base station. Due to the self-configuring capabilities of DSR this change of network nodes would have no negative effect on the communication service provided. While single nodes can leave and join the network without difficulties the UAVs participating in the network setup can also move and change their relative positions. This could be used for example to move the nodes in the formation in some kind of pattern in order to allow for a constant rotation of nodes. This way quadrocopters which participated in the formation for a longer period of time and whose batteries are almost depleted will slowly come into a position from where they can quickly return to the base station. At the same time a new quadrocopter coming from the base-station can immediately take the place left by its predecessor minimizing the time each of the quadrocopters is in the air without participating in the network thus maximizing the quadrocopter's efficiency.

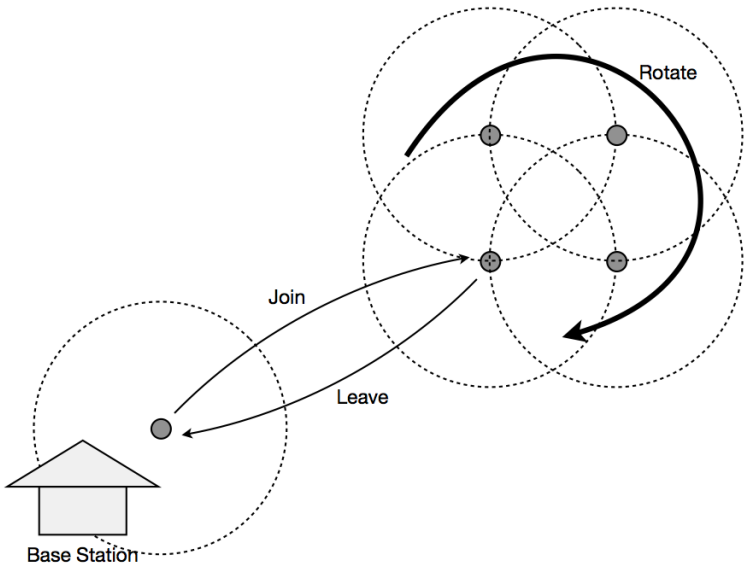

Figure 3.6: Rotating group of quadrocopters

Groups of quadrocopters could also be used to connect two communication endpoints which are too far apart for a single quadrocopter to bridge the distance between the two. Therefore a long formation of quadrocopters would be necessary. A difficulty with this approach though is that replacement quadrocopters do always have to join the formation before any quadrocopter with depleted batteries leave for the base station in order to prevent an interruption of the communication between the two endpoints.

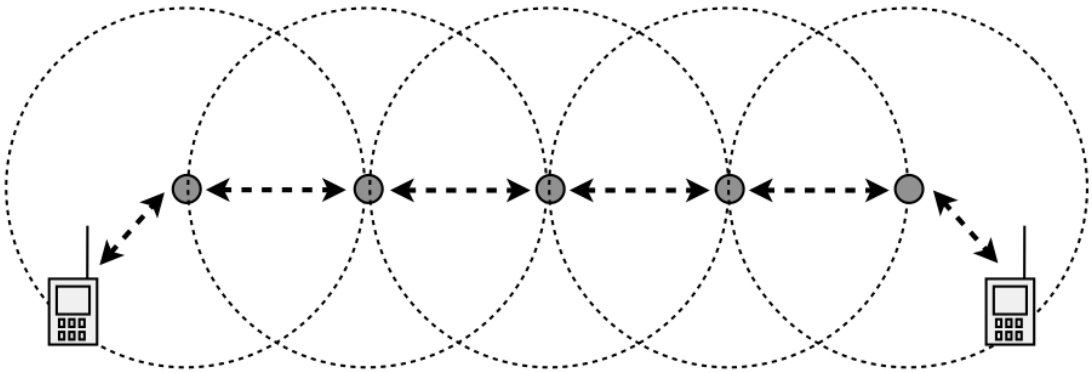

Figure 3.7: A group of quadrocopters connecting two endpoints

Furthermore the quadrocopters participating in the creation of the MANET can not only move in respect to each other but also the entire formation could move and thus provide communication services for moving entities, like for example rescue forces in disaster areas. In such a case the UAVs could also conduct other tasks at the same time like providing video footage of the surrounding area or

measure radioactivity in the case of a nuclear disaster in order to warn rescue forces should radioactivity increase to dangerous levels. The approach of UAVs providing a MANET and following a given entity could also be combined with the idea of a line of UAVs connecting two endpoints. If two entities followed by UAVs move closer together a small number of UAVs from the MANETs of both entities could fan out and create a communication link between the two MANETs thus creating a single network allowing for communication of all participants of the two networks.

3.3 Tracking

Tracking is another possible application for quadrocopters. In a police scenario for example a car or a pedestrian could be tracked by a quadrocopter equipped with a camera to allow for unobtrusive observation. In an aircraft crash scenario over the sea or when a ship has sunk in the ocean quadrocopters equipped with infrared cameras could be sent out to find survivors and when having found someone hover over the victims and following them should strong currents drift them off, acting as a beacon to draw helpers to the survivor's position. Furthermore as proposed in [25] quadrocopters could be used to track forest fires or bush fires in order to provide as accurate data of the fire's spread, its spreading velocity and direction as possible.

Since a single quadrocopter would probably be not able to observe the whole area of the forest fire from one position, in order to track a forest fire's spread it would be required to fly around the fire's perimeter [25]. Furthermore in order to make the information about the fire available to ground crew as quickly as possible under the assumption that the UAV does only have a limited communication range and cannot communicate with its base-station all the time it would be required to regularly return close enough to the base-station to deliver the newly collected data as depicted in figure 3.8.

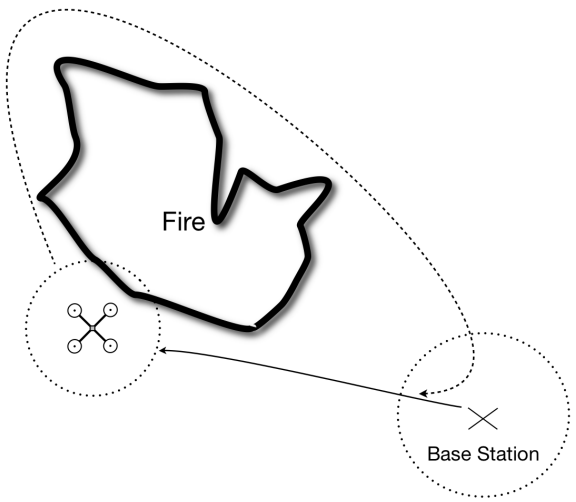

Figure 3.8: A single quadrocopter tracking a fire

When assigning multiple UAVs the task of tracking the fire, data about the fire's spread can be delivered more frequently as with a single quadrocopter. Assuming

the quadrocopters fly around the fire's perimeter in uniform distances to each other the time between updates of the fire's state would decrease with the number of UAVs. Let t_r be the time one quadrocopter needs to fly from within communication range of its base-station around the fire once and return into communication range of the base station. The time between updates about the fire's state t_u when using n quadrocopters then equals

$$t_u = t_r/n. \tag{3.1}$$

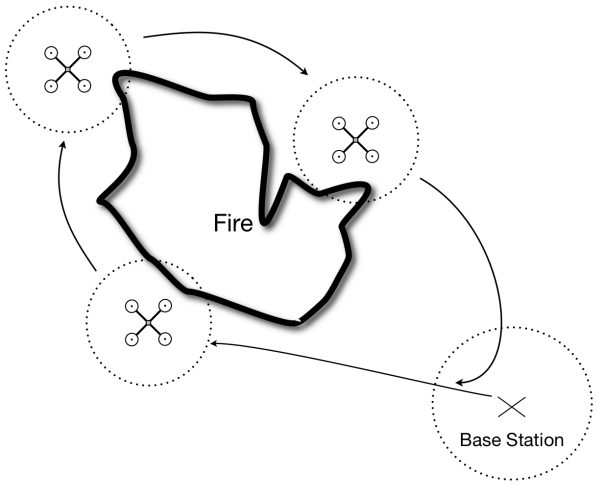

Figure 3.9: Multiple quadrocopters tracking a fire

As can be seen in equation 3.1 the update time decreases with increasing numbers of quadrocopters. If a sufficient number ob UAVs is available the update time can be decreased to any arbitrary value. The data arriving at the base station following this approach does still come with a certain delay. While the data about the fire's perimeter at the end of a quadrocopter's round is relatively fresh the data about the fire gathered at beginning of the quadrocopter's round is still about t_r old even though it arrives at a frequency of $1/t_u$.

This disadvantage can be mitigated taking into consideration the possibilities considered in the previous section about quadrocopters used to build communication networks. Instead of relying on the single quadrocopters to fly around the fire's perimeter as quickly as possible in order to deliver the most up to date information about the fire a number of quadrocopters could fly around the fire and create a communication network after the fashion presented in the previous section. The number of UAVs would only depend on communication range and the fire's perimeter size. Data could be propagated through the network quickly to the base-station with some UAVs possibly providing a communication connection between the UAVs circling the fire and the base station as depicted in figure 3.10.

The efficiency could be further increased by applying the rotation approach presented in the previous section. The UAVs could circle the fire in a little less time than would result in their batteries depleted. This way every UAV would make

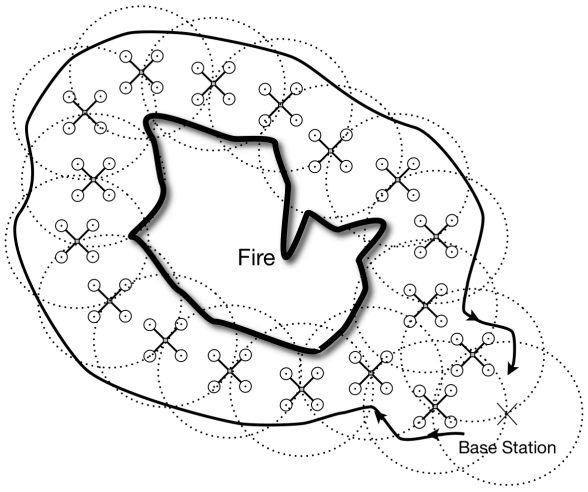

Figure 3.10: Multiple quadrocopters with MANET tracking a fire

exactly one round around the fire's perimeter and would be required to travel only a minimum distance from the fire's perimeter back to the base station in order to recharge. This would also make complicated scheduling of UAVs that need to recharge, planning of their paths from far-off locations to the base-station and any concerns about communication gaps arising from UAVs leaving the formation unnecessary while at the same time saving energy due to the UAVs' lower cruising speed and possibly also increase information quality.

Should the size of the perimeter of the fire increase this can be countered by a higher number of UAVs used and increased overall cruising speed while in the opposite case less UAVs would be required to track the fires perimeter and cruising speed could be decreased.

3.4 Payload delivery

One more application for quadrocopters or UAVs in general is the delivery of a payload to a certain location. An autonomous UAV supposed to accomplish this task would need to possess some kind of waypoint following capability in order to reach the destination where the payload is to be placed, as well as autonomous take-off and landing capabilities if the payload is not to be dropped from mid-air. Furthermore the quadrocopter would need some kind of obstacle evasion algorithms implemented when flying at lower altitudes before landing and after take-off and possibly indoor navigation capabilities, too, if required to deliver something into buildings.

The scenarios for payload delivery are numerous. For example quadrocopters could be used for delivery of documents between offices in a building, or in crowded metropolitan regions like Tokyo between offices situated in very high floors of opposing skyscrapers. This way documents could be flown directly from one tower to the other through a window making it unnecessary for a person to climb down to the ground floor of the first building and then climb up the second one only to

repeat this procedure after delivering the documents in order to return to its office. Considering the preconditions required for this scenario, like reliable autonomous indoor navigation capabilities in order not to harm any persons, this application is rather unlikely to be seen anytime soon and is more an outlook on what is a thinkable use for quadrocopters in a more distant future.

Another scenario for payload delivery could be to let a quadrocopter build up a MANET by placing network nodes at different locations. In contrast to the MANETs with network nodes being carried by a group of quadrocopters presented earlier this approach to building a MANET would require only one single quadrocopter that would carry a number of network nodes, delivering them one at a time to their destined positions. While the mobility of the MANET is decreased to zero once it is in place this approach allows for minimized costs because only one single UAV is needed to create the MANET. Further on network nodes could be stored in places that are hard to reach by a human like roofs of buildings, lantern posts and others. The costs saved due to the lower number of UAVs in use could be invested into network nodes, thus allowing for coverage of a greater area by the MANET being deployed. If the UAV can also take up a network node autonomously this would allow for easy maintenance of MANETs, the UAV replacing nodes of which the batteries are depleted with new ones making it unnecessary for humans to climb on roofs or other dangerous locations.

Chapter 4

Implementation

In this chapter first the quadrocopter model used in USARSim and the sensor configuration it has been equipped with will be presented after which the implemented control program for the quadrocopter is discussed. The control program itself is entirely written in C++ for Microsoft's .Net platform while the USARSim dependent parts, like sensors, are written in the UnrealScript language. The goal of the program is to provide a basis on which control algorithms and learning algorithms for autonomous quadrocopters in USARSim can be implemented without the need to care for all the underlying aspects of the simulation environment and to rid developers of the laborious task of implementing a complete control program themselves. Ideally a developer should be able to use the control program and the provided quadrocopter and sensor configuration as is, and to concentrate solely on his control algorithms. These should be implemented it in a C++ class that is derived from an agent base class, which is provided in the control program and discussed later in this chapter, to get the algorithms to work in the control program and test it in the simulation environment.

4.1 UAV Model

This section will provide an overview of the UAV model and sensors used in the simulation. The specific properties of both can be adapted in the *UDKUSAR.ini* file which holds the configurations of most USARSim robots, sensors and actors.

4.1.1 Quadrocopter Model

Although USARSim comes with several robots ready to use the transition to the UDK version of USARSim is not finished yet. At the moment of writing only one quadrocopter model has been ported from the old USARSim to the UDK version and is available for use. The quadrocopter model provided by USARSim is a model of the AirRobot AR 100-B presented in chapter 2.1.4. The control possibilities for the AirRobot model in USARSim are the same as for a human operator who is controlling the AirRobot through a remote control which means the AirRobot can move in direction of the x, y and z-axis and turn around the yaw. Direct control of the four rotors' rotation speed is not available at the moment but can be implemented if desired by changing the quadrocopter model itself and its properties in the UDK

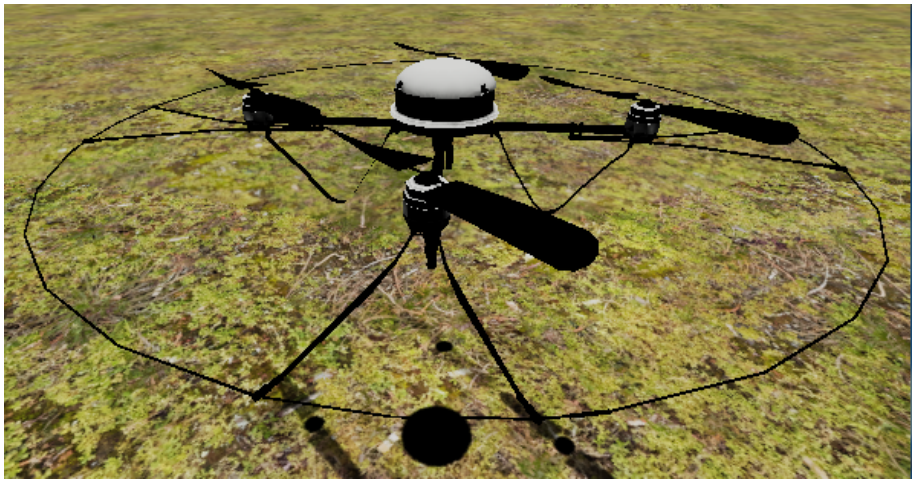

Figure 4.1: Screenshot of the USARSim AirRobot model

Editor and editing the quadrocopter's UnrealScript implementation file. Since flight stabilization usually takes place within a quadrocopter itself though and the above way of quadrocopter control resembles a stabilized quadrocopter it is completely sufficient for the purposes of this diploma thesis.

4.1.2 Sensor Models

Since sensors are important to robot control USARSim simulates proprioceptive sensors like battery or headlight state sensors, position estimation sensors providing information about location, rotation and velocity and perception sensors which include, lasers, sonar and touch sensors among others. In USARSim every sensor is an instance of a sensor class, which specifies a sensor type. The class hierarchy of USARSim sensors is depicted in figure 4.4. Almost all sensors in USARSim can add noise and apply distortion to their data in order to provide a more realistic sensor behavior and the quality of sensor data can be adapted via parameters. Several of the sensors provided by USARSim can be used together with the AirRobot, some of which will be described here briefly.

Sonar Sensor

The sonar sensor implemented in USARSim, in order to mimic a real sonar sensor, sends out a number of traces which in combination form a cone.

The size of the cone and the number of traces can be specified in the sensor's configuration section in the *UDKUSAR.ini* file. For the traces inside the cone the distance from the sensor and any surface within the sensor's range is measured and the smallest distance measured is returned by the sensor if this distance lies within the minimum and maximum range of the sensor. If an object is detected to be closer than the minimum detection distance then the minimum detection distance is returned by the sensor, if an object is further away than the maximum distance the sensor can detect the sensor will return the maximum distance. Anything outside

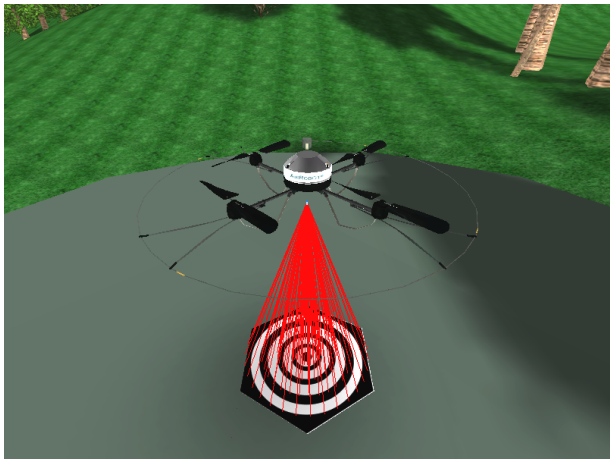

Figure 4.2: AirRobot with a sonar sensor (traces visualized), courtesy of [18]

the sensor's cone is assumed to not throw an echo back to the sensor and therefore is treated as undetectable by the sensor.

Laserscanner HOKUYO URG-04LX

One of the laser-scanners coming with USARSim is a model of the HOKUYO URG-04LX. It's only 5cm wide and long and 7cm high with a weight of only 160g. Although it is a very small and light laser-scanner it provides high quality scans of its environment with a resolution of about 1cm in range and 0.36 degrees in angle. It has a maximum range of 4m and has a field of view of 240 degrees. The HOKUYO URG-04LX is mentioned here because with its 160 g of weight it does not exceed the AirRobot maximum payload limit and thus could be mounted to the quadrocopter even in reality. Laser-scanners comparable to this one have been used for autonomous quadrocopter navigation in indoor environments by A. Bachrach and al. in [26] and [10] among others. The SICK-LMS200 laser-scanner is also available as a model in USARSim but with its 4.5 kg of mass, though, it is much to heavy to be loaded on a quadrocopter like the AirRobot. The USARSim model of the HOKUYO URG-04LX can be adapted in the *USARUDK.ini* configuration file.

Inertial measurement unit (IMU)

USARSim also implements an IMU sensor model. Real IMUs are in principle based on three accelerometers of which the axes are aligned orthogonally to one another and which are mounted on three gyro stabilized gimbals in order to maintain the instruments orientation during maneuvers [27]. The accelerometers each measure the acceleration in the direction of their orientation thus providing a representation of the overall acceleration applied to the device in three dimensional space. One time integration of an accelerometer's output yields velocity while a second integration yields a change of position along the accelerometer's orientation [27]. The gyros used for stabilization measure the angular acceleration, which by one time integration yields the angular velocity and by a second integration yields an angle by which the device was rotated around the gyroscopes measuring axis.

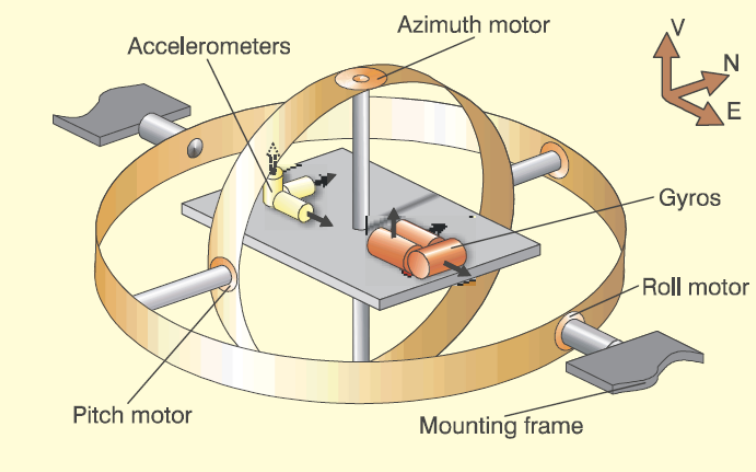

Figure 4.3: Gimbaled Inertial Platform [27]

The IMU sensor model available in USARSim provides information on acceleration in direction of the x, y and z-axis, as well as pitch, roll and yaw accelerations and angles.

4.1.3 Newly implemented sensors

As stated before USARSim allows for the creation of new sensors and sensor types. Sensors in USARSim are derived from the *Sensor* class, which itself is a subclass of the *Item* class. The *Item* class also serves as a base class for the *Actuator* and *Decoration* classes, as is depicted in figure 4.4. The *Decoration* class as the name implies is used merely as decoration on a robot, like for example battery packs or motor boxes whereas actuators are parts for robots which cannot exist on their own and can be made to perform actions, like for example a tiltable camera or a robot arm. In order to create a new sensor, a new class has to be derived either from the sensor class or one of its subclasses.

Position sensor

Although a GPS sensor is provided with USARSim it seems much easier and more intuitive to use coordinates in a three dimensional inertial system with an x, y and z-component instead of altitude, latitude and longitude values measured in degrees, minutes and seconds as is done with a GPS system. When implementing a control algorithm it may be a little bit of an overhead to implement a function understanding GPS coordinates and using these to derive quadrocopter motion, especially if the quadrocopter is to be used in an application where there is no real need for utilization of a GPS signal. This is for example the case in indoor scenarios or when the quadrocopter is to operate in a very limited area or over short distances where GPS coordinates may not even be of high enough resolution. In order to simplify handling of the quadrocopter's position data a new position sensor has been implemented. If, for some reason, coordinates in a GPS format are required, the GPS sensor of USARSim can of course still be attached to the quadrocopter model and used. The new position sensor class is named *stsPositionSensor*. It is

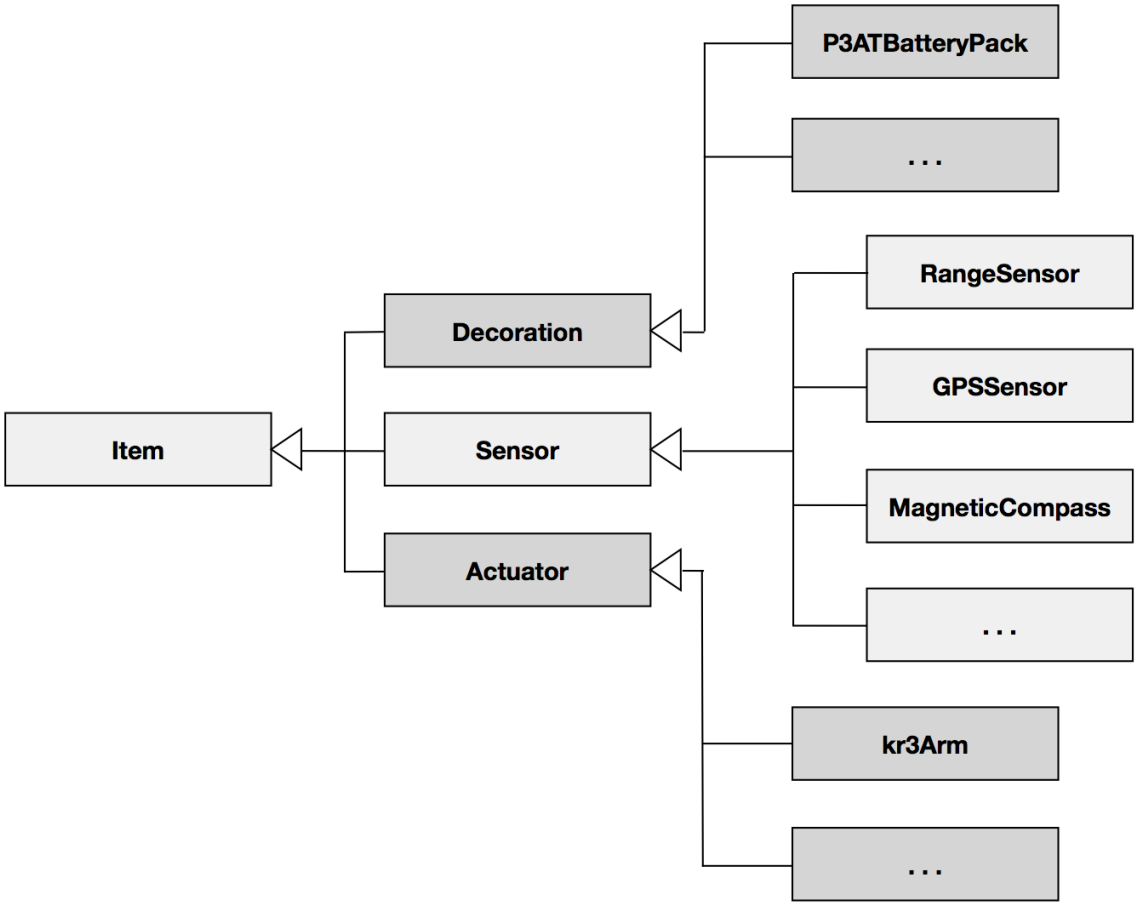

Figure 4.4: USARSim class design for sensors, actuators and decoration

directly derived from the *Sensor* class and provides Cartesian coordinates based on the coordinate system of the simulated environment in meters.

Velocity sensor

While USARSim comes with sensors like a tachometer which allow to determine a wheel based ground robot's velocity it does not provide a sensor which accomplishes the same for an air vehicle like the used quadrocopter model. In order to compensate for this a new velocity sensor has been implemented. The new sensor provides velocity information as well from the view of an inertial coordinate system, as would be measured by a spectator from the ground watching the quadrocopter fly, as velocity information from the quadrocopter's local coordinate system. The former is of interest for example for movement in a global frame like navigating on a map making it possible to see how fast the UAV is going towards which position the latter is interesting if the quadrocopter is to fly maneuvers and needs to plan its movement in the local frame. The new velocity sensor class is named *stsVelocitySensor* and like the previous sensor is directly derived from the *Sensor* class. All values coming from this sensor are given in meters per second.

4.1.4 Base Equippment of the quadrocopter

In order not to create any conflicts with other USARSim based applications a new instance of the AirRobot model available in USARSim has been created and named *stsAirRobot*. To allow for autonomous behavior the sensor configuration of the original AirRobot coming with USARSim has been extended in the stsAirRobot model. The stsAirRobot is equipped with the newly implemented velocity sensor mentioned before providing the UAV with information about its absolute and relative velocities as well as the newly implemented position sensor. Also an IMU sensor allowing the UAV to sense its current linear and angular accelerations as well as orientation and angular velocity has been added to the quadrocopter model. Furthermore the stsAirRobot is equipped with a battery sensor allowing it to check its current battery state. In addition to sensing its location, orientation, velocity and battery state a quadrocopter that is supposed to act autonomously also needs to perceive its environment. In order to recognize and evade obstacles a number of sonar sensors have been attached to the quadrocopter model. Eight sonar sensors monitor the horizontal plane of the quadrocopter around the perimeter while two sonar sensors are monitoring the vertical directions. One of them is oriented straight down the other straight up from the quadrocopter, so that it is capable of perceiving objects above or below it. In addition to the sonar sensors a model of the Hokuyo URG-04LX laser scanner has been attached to the stsAirRobot in order to allow for a more precise measurement of its environment than it is possible with sonar sensors only. The laser scanner covers 240 degrees of the frontal area of the quadrocopter with 228 laser beams. With such a fine angular resolution the laser scanner can also be used to generate maps of the environment to allow the quadrocopter to memorize obstacles and find efficient ways of moving within its environment. The arrangement of sonar sensors and the laser scanner on the vehicle can be seen in figure 4.5.

The sensor equipment provided with the stsAirRobot model should be enough to allow for the implementation of algorithms which enable the quadrocopter to autonomously fulfill a number of basic tasks. Should any different or additional sensors be needed they can easily be attached to the quadrocopter model by editing a handful of configuration files.

4.2 Control Program

4.2.1 Overview

The control program is generally responsible for steering a quadrocopter in the three dimensional simulation environment created by the Unreal Engine and made accessible via USARSim. The program first requires the user to enter some basic data considering the server properties, used quadrocopter model and missions for the UAV to accomplish. The program will create a connection to the communication socket of USARSim via which the control program will communicate with the UAV in the simulation. The UAV will send its sensor data and status data over the link created to the control program which then based on this data will decide what action the UAV is supposed to take and sends the appropriate commands over the

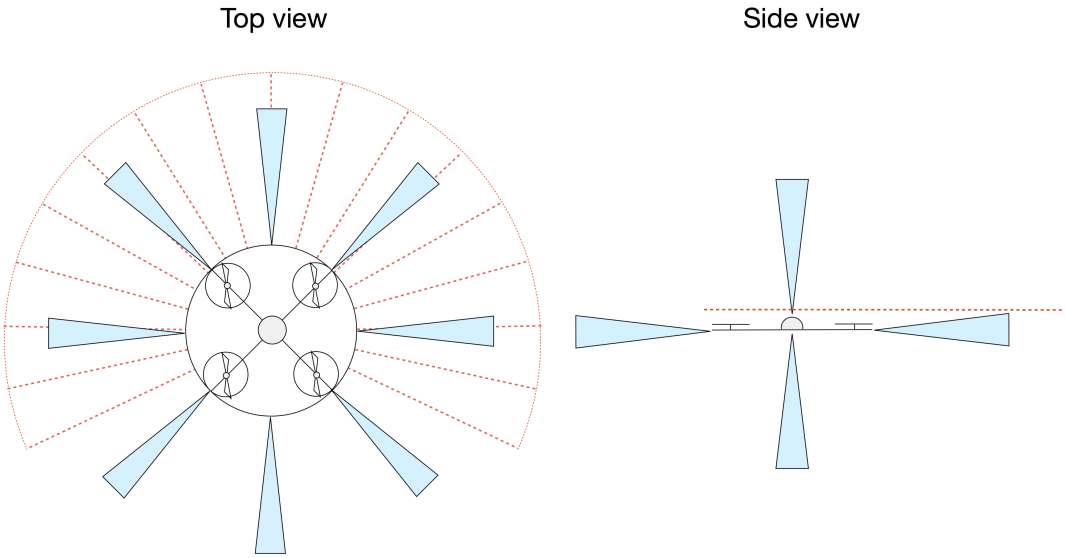

Figure 4.5: Quadrocopter with sonar (blue) and laser (red) sensors

communication link back to the UAV.

Because the control program is based on a client-server architecture it is not restricted to controlling simulated quadrocopters only. Given a real quadrocopter implemented the USARSim communication protocol the presented control program could receive sensor readings from and send commands to the quadrocopter and thus control the quadrocopter in a real environment. Further on, since the underlying Unreal Engine allows for multiple connections at the same time, it is possible to simultaneously have several quadrocopters in the same simulated environment, each of which is controlled by another instance of the control program. This allows for multi-UAV scenarios to be tested and evaluated.

4.2.2 Architecture

As stated before the whole simulation environment is based on the Unreal Engine which is responsible for generating the graphics and calculating the physics of the three dimensional virtual environment. On top of this builds USARSim which provides models of robots as well as sensors and their functionalities, a number of standardized environments and a communication interface to use for robot control. The control program then is built on top of this basis, as can be seen in figure 2.11 and takes responsibility for the interaction between user and simulation environment and robot control.

To provide an easy to understand and easy to modify structure the control program is divided into several modules. Each of this modules has its own functionality and purpose. The modules can be categorized into three main parts in a model-view-controller like fashion. First there is the interface to the user which contains

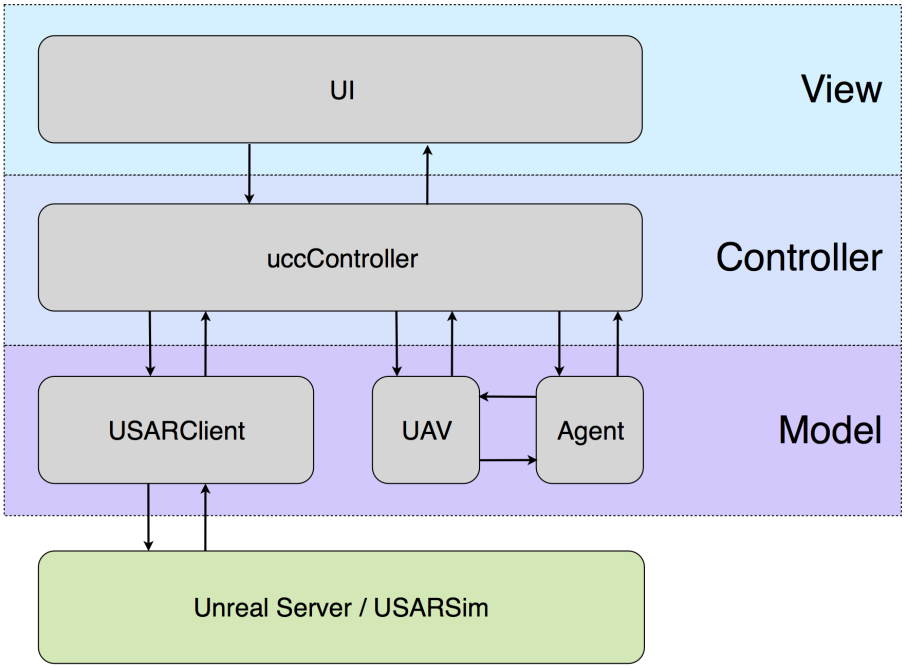

Figure 4.6: Control program architecture overview

the main window and any dialog windows appearing in the program. These are responsible for gathering needed information, reacting to user requests and providing the user with information about the UAV that is being controlled. Second there is the part that is responsible for the program's functionality and models the different aspects of it. This part contains components like for example a client which handles the communication link between control program and USARSim or a UAV component encapsulating all the different aspects of a UAV's functionality and others. The third part is a control layer responsible for coordinating the different components among each other and with the user interface. The basic structure and communication between components can be seen in figure 4.6.

4.2.3 Implemented Classes

The components depicted in figure 4.6 and some additional classes will be described in more detail in this section.

User Interface

The control program's user interface is a collection of different windows containing several Windows forms objects which are to collect data needed for the proper functionality of the control program, like the server name and portnumber, type of agent to use for controlling the quadrocopter or the different missions the controlling agent is supposed to execute. Furthermore the user interface provides information to the user via a log window showing messages from the diferent components and by displaying certain data like for example the quadrocopter's position in the main

window or a map showing the traveled path of the UAV and its mission goal coordinates. Every window is contained in an own class. Figure 4.7 gives an overview of the control program's main windows. Beside these there are a couple of smaller dialog windows which are used in data aquisition from the user.

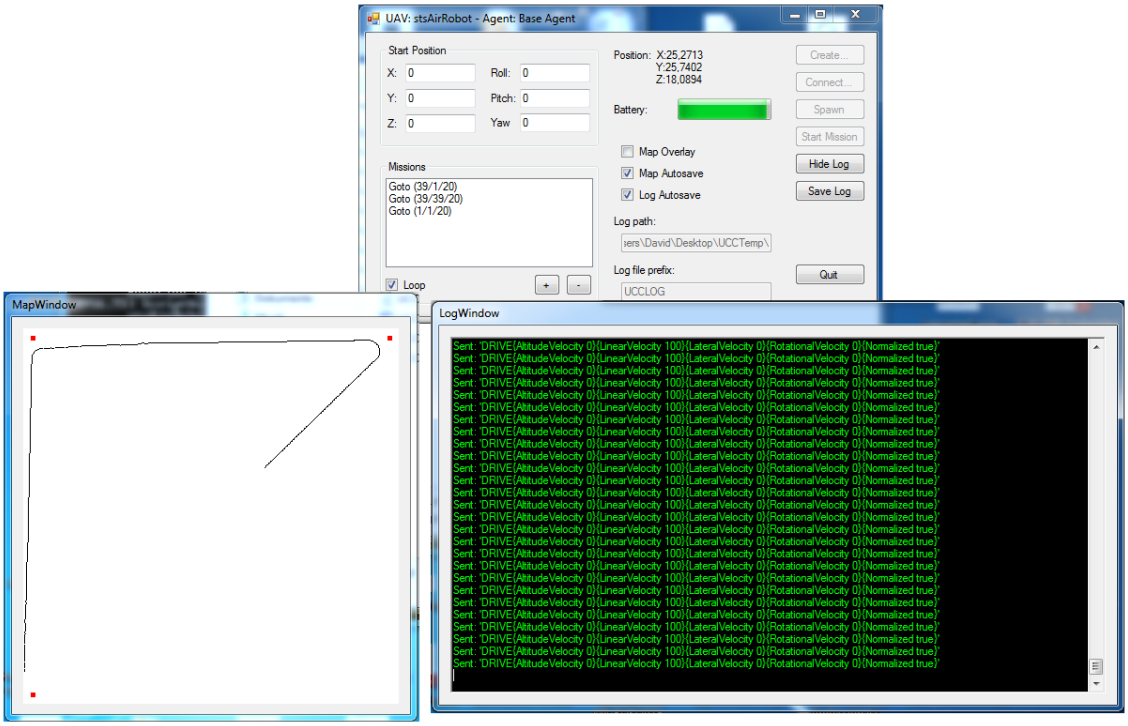

Figure 4.7: The control program's user interface

The three windows depicted in figure 4.7 are the main window which provides the user with several possibilities for input of data, the log window which prints out all log events raised in the program to the screen and the map window, which shows a trail of the path taken by the quadrocopter and its mission coordinates. The main window, its controls and functionality will be thoroughly explaned in section 6.2.

Coordinate Class

A program that is to control a quadrocopter in a three dimensional environment can be expected to make heavy use of coordinates. Coordinates in a Cartesian coordinate system for three dimensional space consist of three components. This would usually require to either create three separate variables for each coordinate or creation of an array with three elements, each of which has to be accesses via an index in between 0 and 2. Although the processing of coordinates can be done this way it tends to complicate and bloat the source code of a program. In order to allow for easy and uncomplicated handling of coordinates within the source code of the control program a wrapper class has been implemented which encapsulates an array and makes the elements accessible as .net class properties via their familiar names, meaning if there is a coordinate object named *coord* then the x, y and z component

of that coordinate are accessible via *coord->x*, *coord->y* and *coord->z*.

As can be seen in the UML class diagram depicted in figure 4.8 an object of this class can be instantiated via its constructors, which as parameters take either three floating point numbers, one for each coordinate component, or an array of three floating point numbers in order x, y and z.

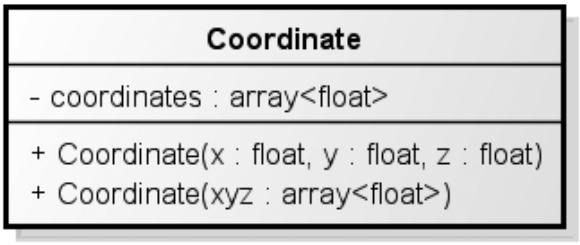

Figure 4.8: Coordinate Class

The .net properties of the Coordinate class which are not shown in the class diagram in figure 4.8 are listed in table 4.1.

Property	**Description**
x	Gets or sets a float value for the x component
y	Gets or sets a float value for the y component
z	Gets or sets a float value for the z component

Table 4.1: Coordinate class properties

Mission Class

The task of the quadrocopter which is to be controlled in its environment by the control program is to fulfill some mission goals. Although these missions can be hard-coded inside the quadrocopter's control algorithm this would make any control program terribly inflexible and require the recompilation of the program for every new mission goal or type of mission the quadrocopter is to accomplish. In order to allow for a more flexible handling of missions a *Mission* class has been implemented.

Most kinds of missions are somehow related to coordinates. Be it following a number of waypoints, patrolling between two positions or operating in a certain area. All these examples have one thing in common. They need coordinates to describe the *place* where a mission has to be accomplished. While coordinates are needed to describe the location for a mission another distinction between missions has to be made if there are to be more than just one kind of mission. An agent controlling a quadrocopter not only needs to know where to accomplish its mission but also what kind of mission it is to accomplish. In order to distinguish between different kinds of missions and to determine *what* has to be done at or with the mission's coordinates, in addition to the coordinates the *Mission* class has a *type* attribute. This way an agent that is given a number of missions can easily determine what to

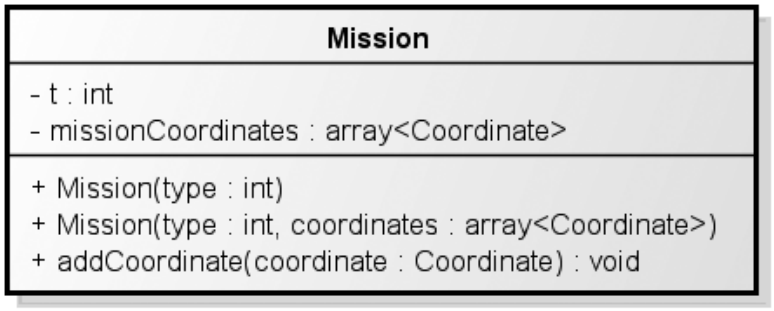

Figure 4.9: Mission Class

do in each mission by the mission's type attribute and new mission types can easily be implemented by just defining a new type number for the new kind of mission and implementing a special behavior for that kind of mission in the controlling agent.

The constructors for the Mission class take either just an integer representing the type of mission or an integer for the type and an array of coordinates for the mission. The *addCoordinate()* method allows to add coordinates to an already instantiated Mission object.

Property	**Description**
type	Gets an integer value for type of mission
coordinates	Returns an array with the mission's coordinates

Table 4.2: Mission class properties

As with the *Coordinate* class .net properties have been defined to allow for easier access to the class' variables. The properties are listed in table 4.2 and allow read only access to the mission's attributes.

USARClient Class

The USARClient class is responsible for setting up a communication channel to the USARSim server, receiving, parsing and passing on incoming messages from the server and the robot in the virtual environment as well as sending messages to the server and to the simulated robot.

The constructor of the USARClient class does not take any parameters. The *setHost()* command takes a host name and a port number as a parameter and sets up the host to use for communication. The *connect()* and *disconnect()* commands are self-explanatory. They connect the client to or disconnect it from the server. When invoking the *receive()* method message reception is started. Message reception is handled by a new thread which is created by the USARClient object. This reception thread is constantly running and listening for incoming messages. If a message is received it is parsed and an event is raised by the USARClient object.

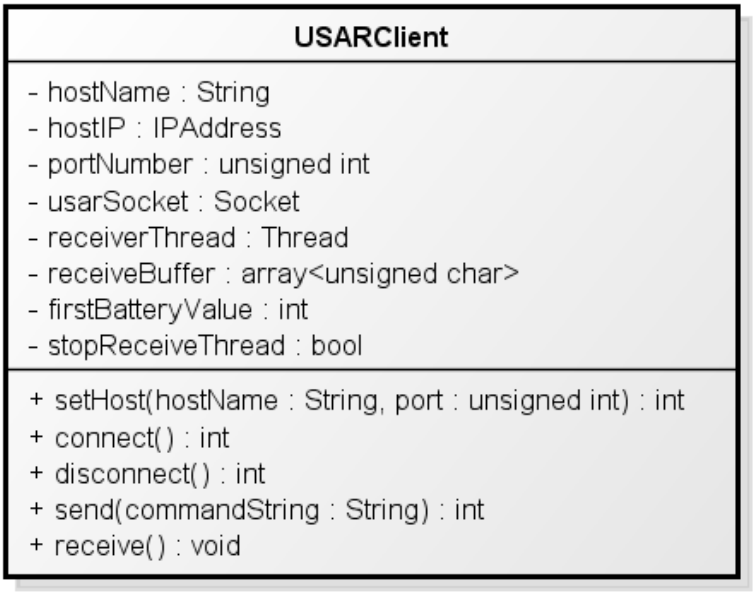

Figure 4.10: USARClient Class

These events are then received by the controller object which will be discussed in section 4.2.3 but could also be received by other objects if they registered for it. Different kinds of messages result in different kinds of events which trigger a different behavior in the controller object. In order to send command messages to the server the USARClient class provides a *send()* command which takes a message string containing the command as a parameter.

The class' attributes are almost all self-explanatory. *hostName*, *hostIP* and *portNumber* are needed to describe the USARServer end of the communication link, *usarSocket* is an object representing the socket used for communication. *receiverThread* is the thread object for reception of incoming messages, while *receiveBuffer* is needed for message buffering upon reception. The *firstBatteryValue* variable is a helper variable which is needed to determine what value of battery charge is to be interpreted as a fully charged battery and last but not least the *stopReceiveThread* variable indicates whether the receiver thread is to be stopped.

The events that are generated by the USARClient class are listed in table 4.3

Event	Description
UAVPositionEvent	new position data received
UAVBatteryEvent	new battery data received
UAVVelocityEvent	new velocity datareceived
UAVLinearAccelerationEvent	new linear acceleration data received
UAVAngularVelocityEvent	new angular velocity data received
UAVAngularAccelerationEvent	new angular acceleration data received
UAVOrientationEvent	new orientation data received
UAVSonarTopEvent	new data from the top sonar received
UAVSonarBottomEvent	new data from the bottom sonar received
UAVLaserScannerEvent	new data from laser scanner received
LogEvent	there is something to log
UAVSonarHorizontalEvent	new data from horizontal sonar sensors received

Table 4.3: Events generated in USARClient

UAV Class

This class is supposed to encapsulate the properties of the quadrocopter or any given UAV, which is the current state and position of the vehicle and all its recent sensor readings. Further on it is supposed to provide the interface to the UAV for any control algorithm that is to interact with the air vehicle. A UML class diagram showing the UAV class' attributes and methods is depicted in figure 4.11.

The most recent values of all the quadrocopter's sensor readings are stored via setter methods, which all begin with the *update* prefix, in internal variables inside this class and are made accessible via getter methods. These two make up the bulk of the UAV class' methods. Instead of getter methods the internal variable values could have been made accessible via class properties, giving them more convenient names and allowing for easier use. It would be less obvious then, though, that this results in a function call every time the property is used which in a method where the UAV object's internal variables are heavily used could lead to a noticeable degradation of performance. On top of that the implementation of new sensors would require the additional overhead of providing any new internal variables of the UAV class as property values to remain consistent with the current implementation and thus increase the amount of code to be written. Aside from getter and setter methods there are only two more methods available in the UAV class. First, the UAV class' constructor which takes no further arguments. Second, the *sendSpeedCommand()* which sends a control command to the quadrocopter model in the virtual environment when invoked, telling it the desired velocities in direction of the three axes and around the yaw. These velocities are all integers in the range from -100 to +100 indicating the percentage of the maximum possible speed in the desired direction. The *sendSpeedCommand()* command is supposed to be used by agent objects to control the UAV in the simulation. All agent interaction with the quadrocopter model happens through a UAV object.

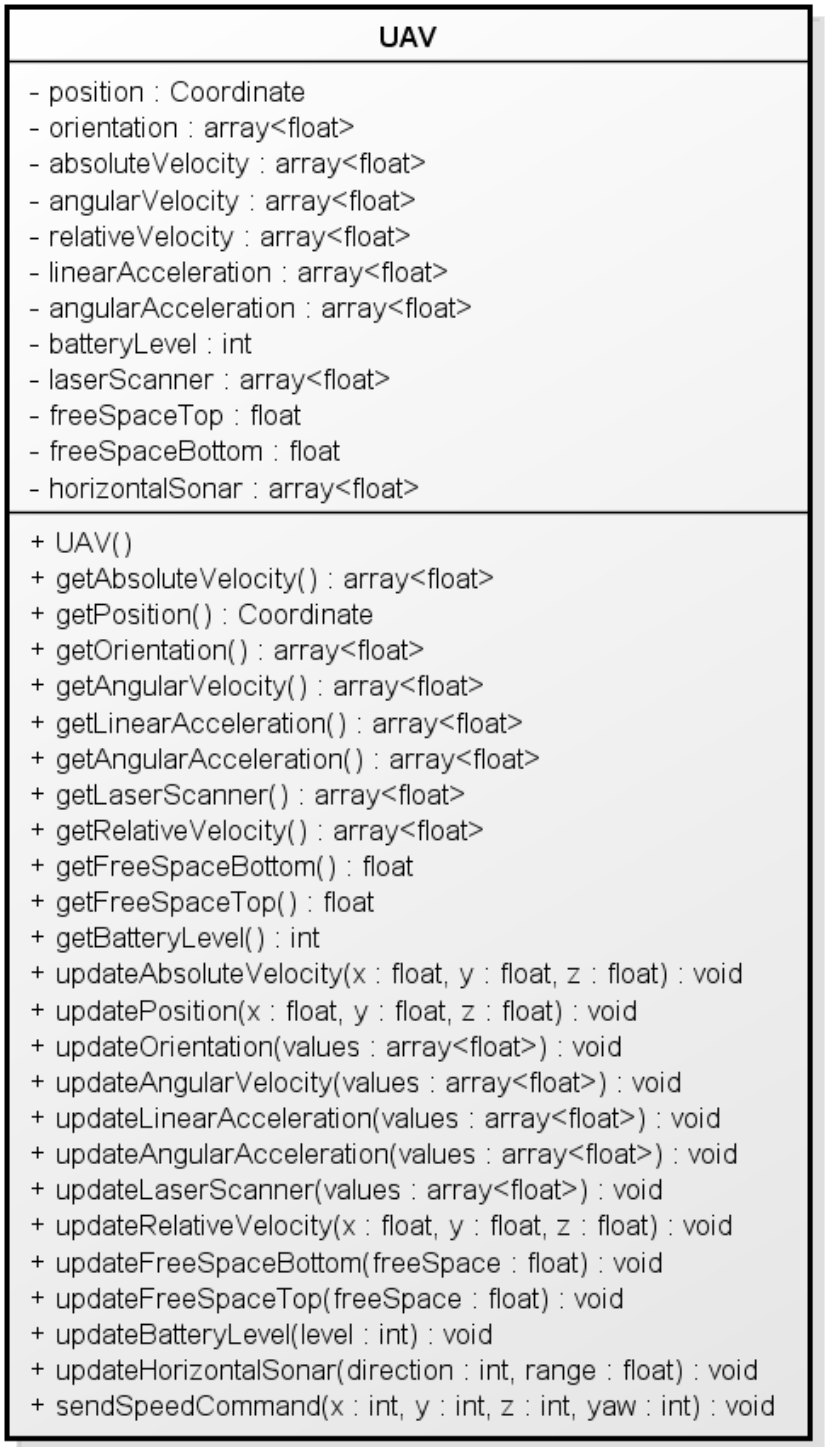

Figure 4.11: UAV Class

UAV objects, just like objects of *USARClient* can raise a *LogEvent* which is useful for checking certain sensor values when debugging an algorithm.

Agent Class

The Agent class is intended as a base class for agents which control the quadrocopter model in the virtual environment.

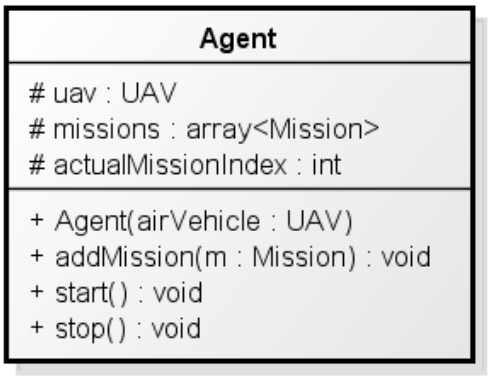

Figure 4.12: Agent Class

As can be seen in figure 4.12 it has three attributes. First, it has a pointer to the UAV object which represents the quadrocopter in the simulation it is to control. Second, it contains an array which holds all the missions the agent is supposed to execute and third, it has an integer which is used as an index to the mission in the missions array which is currently being executed. Furthermore it implements three methods beside its constructor which takes a pointer to the controlled UAV object. The *start()* method is supposed to start the execution of the agent and let it send control commands to the quadrocopter, while the *stop()* command is supposed to stop the agent and quadrocopter control. The third method is the *addMission()* method which is supposed to insert additional entries to the list of missions the agent is supposed to execute. These three methods are declared virtual in order to generate polymorphism in subclasses, so that every class inherited from the Agent class can implement its own *start()*, *stop()* and *addMission()* methods, containing its own routines and logic while at the same time allowing these methods to be called from the controller class via a simple pointer to the base class of all agents which lies at the top of the inheritance tree.

Objects of the *Agent* class can raise two different event types. The first event, *LogEvent*, results in a log message in the log window containing the provided string argument. The second event is the *AllMissionsDoneEvent* which signals that the agent has finished executing its missions, allowing for appropriate actions.

The Agent class itself does not provide any useful functionality concerning the control of a quadrocopter, yet. This has to be implemented by subclasses of the Agent class.

uccController Class

The uccController class is responsible for setting up all components from the model layer presented earlier, according to the data provided by the program's user in the user interface section, as well as coordinating all these components in respect to each other.

Figure 4.13: uccController Class

All objects of the underlying model layer are created and controlled by the *uccController* class. The procedure basicly goes like this. The *uccController* creates a *USARClient* object which upon command by the user connects to USARSim and begins to receive messages sent by the server. Then it creates a *UAV* object as well as an *Agent* and the *Agent* object is associated with the *UAV* object it is supposed to operate on. When the user presses the respective buttons, the quadrocopter model is spawned in its virtual environment, the *Agent* object is handed over the missions it is to accomplish, and then begins to read sensor values from the quadrocopter and generate control commands for it to achieve its missions.

The uccController object is also responsible for receiving all events and messages sent by the underlying objects and forward them to the right destination or take the right actions. These events can be of different kinds. Each component in the model

layer can for example issue a *log event*. The controller object upon reception of such an event takes care of printing the log message received in the user interface's log window. If new sensor readings are received by the USARClient object, these result in an event, too, upon which the uccController object updates the UAV object's internal variables according to the new values. All in all there are fourteen methods handling different kinds of events defined in the *uccController* class. They all follow the naming convention *onXYZEvent* where XYZ is replaced by the event name.

The constructor of the *uccController* class takes three delegate methods as arguments so it can set up the functionality for writing log messages to the log window and printing position and battery information in the main window. Which methods to call is stored in the *log*, *setUIPosition* and *setUIBattery* attributes of the class. The *agent*, *uav* and *client* attributes are needed for the model layer objects. The *missionLoop* attribute defines if missions should be executed in a loop or only once. *startPosition*, *startRotationRoll*, *startRotationPitch* and *startRotationYaw* are needed to save the quadrocopter model's start position and attitude to always respawn the model at the same position when missions are executed in a loop. The remaining methods do what their names imply. *createUAV()* initializes the *uav* and *agent* attributes, *setHost()* provides the *USARClient* object with information about the server, *connect()* makes the *USARClient* object establish a connection to the server, *spawn()* places the quadrocopter at the given position in the simulation and *startMissions()* starts the execution of the agent. Missions are added to an agent by calling the *addMissions* method and *stop()* stops the agent and the client and their threads.

4.2.4 A waypoint following agent

As stated in the previous section about the *Agent* class the class is designed as a a base class for agents to be derived from and does not implement any useful behavior itself. Control algorithms are supposed to be implemented in subclasses of the *Agent* class and because of the polymorphy achieved by declaring the base class' *start()*, *stop()* and *addMission()* methods as virtual methods should work without any bigger code modification in the rest of the program. Any code that called one of the base class' three methods should be able to use the methods provided by the subclass right away. In order to verify the functionality of the concept a sub-classing the *Agent* class a new agent will be implemented in this section.

One very basic capability of autonomous UAVs is the ability to autonomously follow a given number of waypoints and reach a target destination. In order for the new agent to do something more useful the task for the new *Agent* subclass will be exactly that. It will take control of the quadrocopter in the simulation, use its position and orientation sensors and, using a simple but effective algorithm, will generate control commands for the quadrocopter to provide a basic waypoint following functionality. The new class has been named BaseAgent and its class diagram can be seen in figure 4.14.

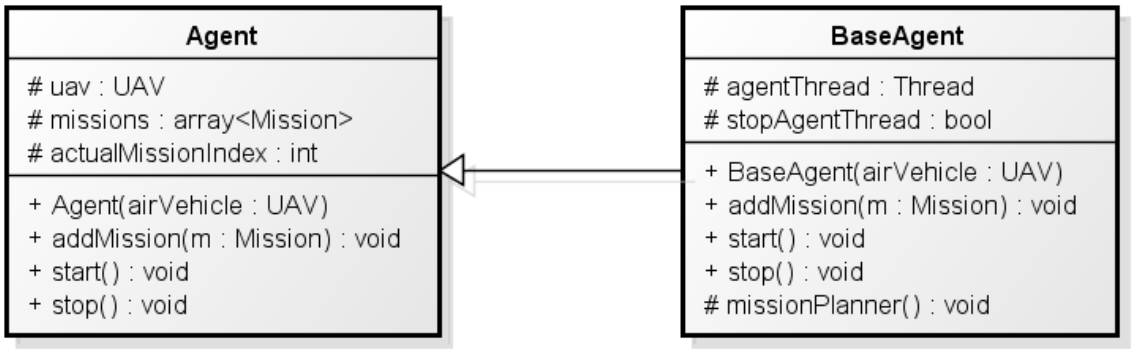

Figure 4.14: BaseAgent Class

Control Algorithm

In order to follow a number of waypoints an agent has to be able to reach a single destination position first. When it is capable of that it can be commanded to fly to a sequence of given positions thus producing the desired waypoint following capability. The algorithm used for the *BaseAgent* class to generate control commands for the quadrocopter will be presented here. As already explained earlier the quadrocopter model in USARSim can be commanded to move in three directions and to turn around the yaw. The algorithm presented here will only generate control outputs for the quadrocopter's linear velocity, its altitude velocity and its yaw rotation. The lateral velocity output generated by the algorithm will always be zero. An approach with this restriction may not make use of a quadrocopters full maneuvering capabilities but this does not prohibit the quadrocopter's ability of following a sequence of waypoints.

The control algorithm divides the control problem into two parts. One is control of the quadrocopter in the x/y-plane while the other is the altitude control of the quadrocopter. Control of the quadrocopter in the plane will be discussed first.

Flight control in the plane: Lets assume we have a quadrocopter in its local frame as depicted in figure 4.15 with the x axis extending in forward direction and the y axis extending to the right. Let the point P be the destination coordinate the quadrocopter is to reach and α the angle between the y-axis and the direction vector of the destination. The problem of generating control outputs for the quadrocopter in the plane can be boiled down to the generation of a velocity value for movement in direction of x and an angular velocity for yaw movement. Taking a look at the sine and cosine functions depicted in figure 4.16 the following coherence of the distance to the destination P, the angle α, and the linear velocity can be observed. If the destination point is situated in front of the quadrocopter the angle α lies between 0 and 180 degrees, if it is behind the quadrocopter the angle lies somewhere in the range from 180 to 360 degrees. This means that the sine function of the angle α will be equal 1 if the destination is situated directly forward from the quadrocopter, -1 if it is situated directly behind and 0 if it is exactly to the left or right of the quadrocopter. Since the commands for the quadrocopters velocities take values between -100 and +100, which are to be understood as a percentage of the maximum

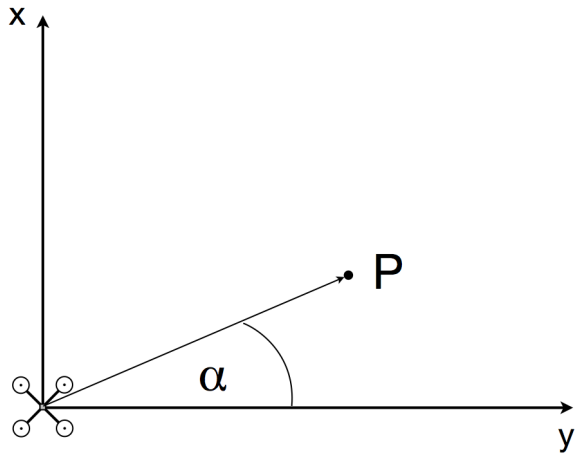

Figure 4.15: Control in the x/y-plane

possible speed at which the quadrocopter is capable of flying, the sine function provides a pretty comfortable way of manipulating the quadrocopters linear velocity. The control algorithm's output for the linear velocity is generated with the formula shown in equation 4.1

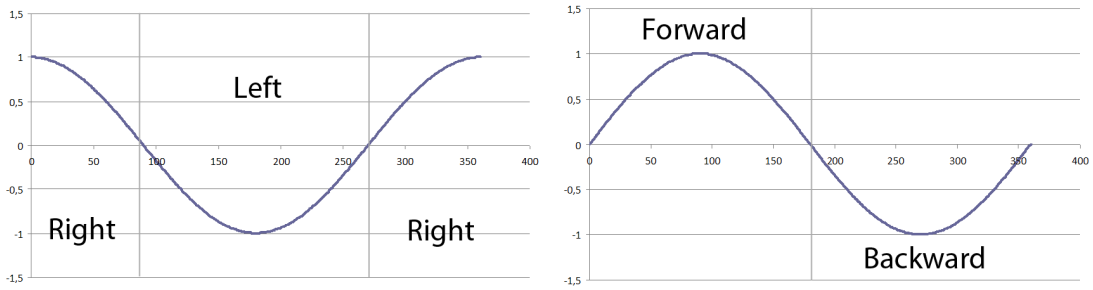

Figure 4.16: Cosine and sine with marked areas

$$v_x = \begin{cases} 100sin(\alpha), & \text{if } 0 \leq \alpha \leq 180 \\ 0, & \text{else} \end{cases} \tag{4.1}$$

As a result the quadrocopter will move forward faster if the destination point is right in front of it and increasingly slowly the further the destination point lies to the side of the quadrocopter. If the destination point lies behind the quadrocopter there will be no forward movement at all.

The second component for the quadrocopters movement in the x/y-plane is the yaw velocity. For its value again a trigonometric function is used, this time the cosine function. Considering the plot of the cosine function depicted in figure 4.16 and our quadrocopter's coordinate system one can see that if the destination point lies at the left side of the quadrocopter the angle α lies in the range between 90 and 270

degrees which results in a value of the cosine function of α in between 0 and -1 and if the destination point lies on the right side of the quadrocopter α lies in between 270 and 90 degrees resulting in a cosine of α in the range of 0 and +1. Accordingly the cosine function of α is a good indicator for the angular velocity around the yaw. Again the allowed control values for the rotational velocity are from -100 to +100 so that the control algorithm's yaw velocity is calculated by the formula in equation 4.2.

$$v_{yaw} = 100cos(\alpha) \tag{4.2}$$

As a result of the control values generated by equation 4.1 and 4.2 the quadrocopter's flight behaviour in the x/y-plane is as follows. The more directly it is heading into the direction of its destination the faster it will fly and the lower its yaw velocity will be. On the other hand the more the direction the quadrocopter is heading in is deviating from the direction of its destination the slower it will fly and the faster it will turn into the desired direction. Should the destination even lie behind the quadrocopter it will come to a full stop and turn until the destination enters the front side of the quadrocopter.

Altitude control Since the quadrocopter is not to fly in a two but in a three dimensional environment the agent in control of the quadrocopter also needs to generate control values for the UAV's velocity in direction of the z-axis. Let's assume that the quadrocopter lies in the z/x-plane as depicted in figure 4.17 where the x-axis again increases to the front of the quadrocopter while the z-axis points from the quadrocopter up.

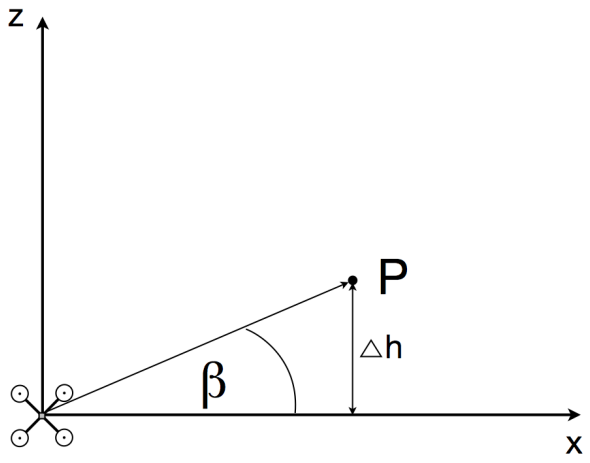

Figure 4.17: Altitude control

Let P again be the destination coordinate and β the angle between the x-axis and the direction vector of P. This is basicly the same problem as with the quadrocopter's linear velocity. This time if the destination point P lies above the quadrocopter we want it to rise with speed increasing the more β approaches a value of 90 degrees

and otherwise we want it to decrease its height also with increasing speed the more β approaches a value of 270 degrees. Thus the agent's velocity output in direction of z is calculated with the formula in equation 4.3.

$$v_z = 100sin(\beta) \tag{4.3}$$

Implementation

Before a waypoint following agent could be implemented a new kind of mission for this purpose had to be defined. This has been done in the *Mission.h* file by adding a line defining a constant named MISSION_GOTO_XYZ. Missions of this type are supposed to contain one coordinate, which to reach is the mission goal. In order for the new *Agent* subclass to be able to add the new kind of mission to its mission array the addMission() method of the BaseAgent is redefined. Any missions of type MISSION_GOTO_XYZ passed to that method are added to the array of missions inside the agent. All other missions are dropped with a LogEvent announcing that an attempt was made to pass an unknown mission type to the agent object.

In addition to the methods and attributes provided by the base class two new protected methods and two new protected attributes have been added to the subclass. The first new attribute is a thread object named *agentThread*, the second one a helper variable of boolean type indicating whether the thread should be stopped or not. The *threadLoop()* method implements a small loop which first checks if the *stopAgentThread* variable is true or false. If it is true, the loop will stop immediately. If it is false the *missionPlanner()* method will be executed, after which the thread is paused for a predefined period of time before the loop begins anew.

The *missionPlanner()* method is responsible for generating the control commands for the simulated quadrocopter and implements the previously described algorithm. The *missionPlanner()* method will guide the quadrocopter to the target coordinate of the next mission in the missions array. It does this by constantly getting the most recent position and orientation data from the *UAV* and evaluating equations 4.1, 4.2 and 4.3. After having calculated the appropriate values for the quadrocopter's velocities it sends a control command to the quadrocopter via the *sendSpeedCommand()* method of the *UAV* object. If a waypoint has been reached the next waypoint will be targeted by the *missionPlanner()* method until all missions in the missions array have been executed upon which the loop will be quit and an *AllMissionsDoneEvent* will be raised.

The *start()* and *stop()* methods have also been changed in the new subclass. The *start()* method now resets the mission index to zero in order to point at the first waypoint mission and then initializes the thread object to execute the *threadLoop()* method which will then be started in a new thread. The *stop()* method will simply

set the *stopAgentThread* variable to true and return after the thread was stopped.

By the relative minor additions of two new variables and methods the new *BaseAgent* class is in fact capable of following a series of waypoints implementing an important base capability for an autonomous air vehicle as will be shown in a series of exemplary tests in chapter 5.1. Furthermore this shows the ability of the provided framework to allow for an iterative extension of an agent's capabilities by sub-classing an existing *Agent* class and adding new features to it.

Chapter 5

Tests

5.1 Waypoint following

The *BaseAgent* class presented in chapter 4.2.4 was implemented to provide the capability of following a given number of waypoints. In order to test whether the new agent type can follow a series of waypoints sufficiently well a coulple of exemplary testruns have been conducted. In each test run the agent was given a couple of waypoits it had to visit before returning to its starting point. The screen-shots depicted in figure 5.1 show two exemplary paths taken by the quadrocopter to reach the waypoints depicted in red squares on the map in the specified order.

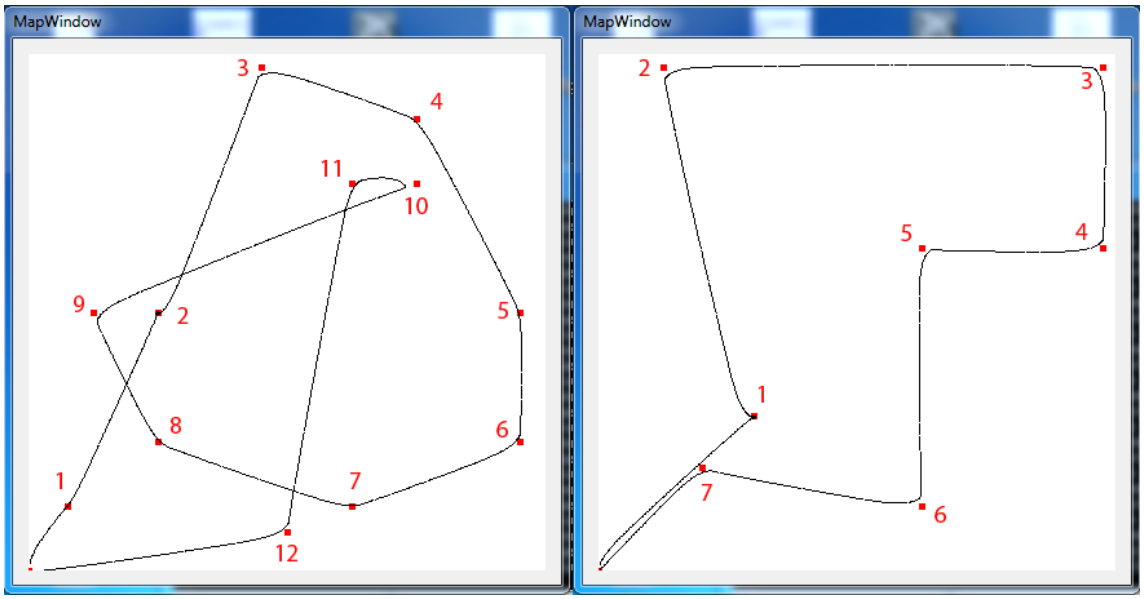

Figure 5.1: UAV following waypoints

As can be seen the agent moves from one waypoint to the next as it is supposed to do. Figure 5.2 shows two screen-shots of maps depicting five consecutive runs of the same experiment in a row.

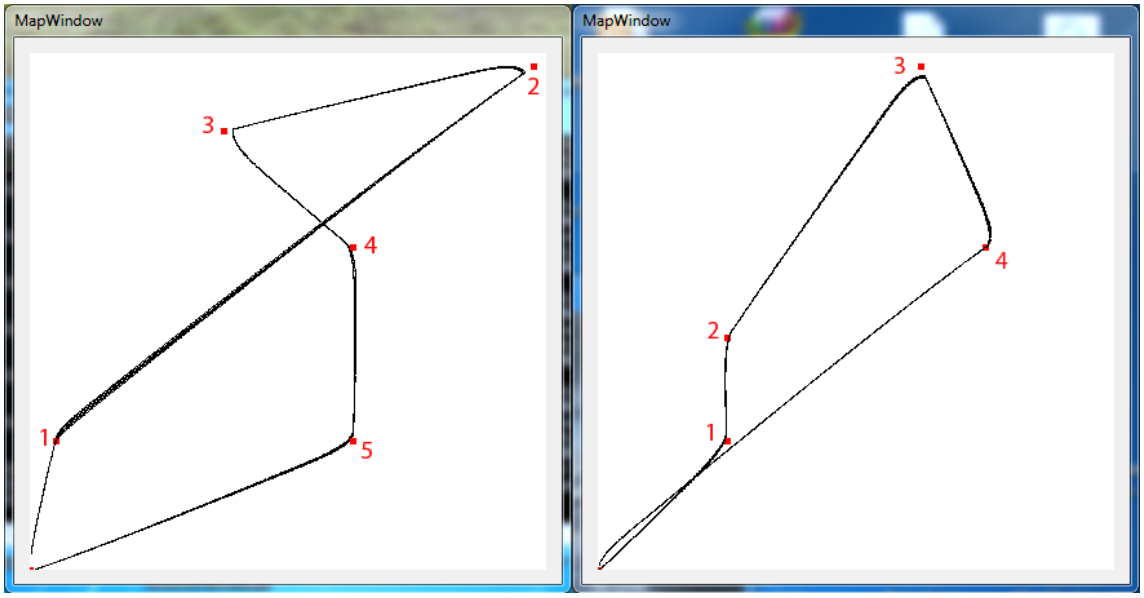

Figure 5.2: UAV following waypoints, overlaid paths

As can be seen in the overlaid experiment maps in figure 5.2 the paths created by the algorithm are almost identical in each run. This shows first, that the algorithm creates reproducible results and second, that the simulation environment allows for almost identical execution of the same experiment for any given number of times.

5.2 Repeating Experiments

The simulation environment and the control program provided in this diploma thesis were also supposed to be usable for the case of machine learning algorithms. As previously mentioned many learning algorithms need to perform numerous iterations of a given experiment optimally every time under the exact same conditions in order for noticeable learning to take place. The control program provides the possibility to execute a once set up experiment in exactly the same way in a loop, over and over again, as could be seen in the overlaid maps of the waypoint following tests. In order to test the stability and reliability of the control program an exemplary experiment has been executed in a loop for a time period of three hours. In the experiment the simulated quadrocopter had to follow a course of three waypoints after which it was to return to its starting position. In the course of this test the experiment was repeated 237 times without problems or crashes showing that the program can be used for a high number of iterations of experiments over a long period of time and can be of use in the testing of learning algorithms.

5.3 Scalability

For the simulation environment to be useful for the creation of groups or even swarms of UAVs the system has to be scalable. In order to test the scalability of the simulation environment, tests have been conducted with an increasing number of UAVs being placed in the simulation environment and the CPU usage of the

used PCs and the displayed frames per second in the client window being measured. The tests have been conducted with two different PCs, one being an older MacBook the other an also older desktop PC. The systems' specifications are listed in table 5.1.

	MacBook	**Desktop PC**
CPU type	Intel Core2 Duo P8600	AMD Athlon64x2 5200+
CPU speed	2.4 GHz	2.6 GHz
Memory	8GB	8GB
Graphics adapter	nVidia GeForce 9400M	nVidia GeForce 8600 GT
OS	Windows 7 Professional	Windows 7 Professional
Network speed	1000Mbps	1000Mbps

Table 5.1: Testsystems' specifications

Because of the relatively poor hardware the Unreal Client has been executed in window mode instead of full screen mode with a reduced resolution of 640x360 pixels on both systems. First tests were done on the MacBook running the Unreal Server as well as the Unreal Client and all the instances of the control program controlling the different quadrocopter models in the simulation. The simulation environment was started with no quadrocopters being placed inside the virtual environment, after which successively one quadrocopter after the other has been added to the virtual environment. The results of this test are listed in table 5.2.

No. of UAVs	**FPS**	**CPU usage (%)**
0	61.96	35.94
1	61.83	52.09
2	59.64	58.52
3	56.6	70.03
4	52.33	72.55
5	45.7	75.11
6	18.13	70.93
7	3.87	71.25

Table 5.2: Multi UAV test results on MacBook

As can be seen from the test results in table 5.2 the frame rate in the client window drops below 20 frames per second if more than five quadrocopters have been placed in the simulation. Frame rates below 20 are impractical for simulation since the rendering of movement becomes choppy to the naked eye and the long lags between two successive frames keep the controlling agent from reacting to any events in the simulated environment in a reasonable time. A frame rate below 4 frames per second as is observed with 7 quadrocopters is totally unacceptable because sensor data will be updated only about four times a second and the agent will react to any sensor readings with a delay of at least 0.25 seconds.

The second test was conducted with all software running on the desktop PC, following the same procedure as described for the MacBook. The test's results are listed in table 5.3

No. of UAVs	FPS	CPU usage (%)
0	62	24.18
1	62	34.8
2	61.99	38.75
3	61.6	42.39
4	61	46.83
5	59.5	49.97
6	58.4	58.92
7	57	61.93
8	53.5	64.55
9	51.2	67.24
10	44.2	69

Table 5.3: Multi UAV test results on single desktop PC

The test data for the desktop PC yields better results than on the MacBook before. While on the MacBook values got into a critical range for more than five simulated quadrocopters, on the desktop PC even with ten quadrocopters being simulated at the same time the measured values remain in an acceptable range. This is probably due to the much more powerful graphics adapter found in the desktop PC.

Since the simulation environment is based on a client-server architecture and two PCs were available for testing purposes one more test has been conducted, this time with the Unreal Engine and Unreal Client running on the desktop PC while the control programs controlling the simulated quadrocopters were run on the MacBook. Both computers were connected via a 1000 Mbit ethernet network. Results from that test are shown in table 5.4.

No. of UAVs	CPU usage MB(%)	CPU usage DT(%)	FPS
0	0.9	26.63	61.84
1	2.44	30.42	61.99
2	2.8	34	61.86
3	2.48	40.17	61.99
4	5.41	44	60.82
5	5.7	48.67	60.64
6	5.92	49.47	60.54
7	7.64	52.07	59.04
8	8.52	53.11	55.7
9	9.03	55.41	57.71
10	9.57	58.1	53.84

Table 5.4: Multi UAV test results with network setup

As could be expected the number of frames displayed per second in the Unreal Client window were noticeably higher and also the CPU usage on the desktop PC was noticeably lower than in the second experiment conducted. The measured values also show that the much more resource consuming component of the simulation environment is the Unreal Engine and the Unreal Client, doing graphics rendering and physics calculations, while the control program itself makes only moderate use of CPU resources.

It can be concluded from the test results that the provided simulation environment is capable of simulating at least moderately sized groups of quadrocopters on the hardware used, proving the system to be scalable to a certain degree. Furthermore it can be assumed that group size and scalability can be greatly increased if more up to date hardware like a decent graphics adapter and more powerful multi core CPU were used to run the simulation environment.

Chapter 6

Usage

6.1 Installation of the Simulation Environment

6.1.1 Unreal Development Kit

The latest released version of the Unreal Developer Kit can be found and downloaded from *http://www.unrealengine.com/udk/*. The size of the installation file at the moment of writing has grown to more than 1.7 GB, so that it can take a little while to download. The installer when started will suggest to install the UDK in a directory like *C:/UDK/UDK-YYYY-MM* where *YYYY* stands for the year and *MM* for the month of the UDK's release. The suggested installation directory is a good choice and can be kept as is since USARSim sometimes fails to install if the UDK is installed into a directory with special characters. In the following it will be assumed that the UDK has been installed in the directory suggested by the installer. If that is not the case paths in the following instructions concerning the UDK installation directory have to be adapted to the path actually chosen during installation.

6.1.2 USARSim

In order to install USARSim a working version of the Git source code management system needs to be installed on the computer used. A new directory should be created and a console window opened. The working directory of the console window needs to be changed to the just created directory and there the following command has to be issued to download all USARSim files from the USARSim Git repository.

```
git clone git://usarsim.git.sourceforge.net/gitroot/usarsim/usarsim
```

Once the download is finished all the files in the subdirectory *usarsim* have to be copied into the UDK installation directory. If a dialog pops up and asks whether existing files are supposed to be overwritten by the new files this has to be answered with *yes*. After copying of files has completed, the batch program *make.bat* has to be executed to compile USARSim on the machine. If the compilation has been successfully finished the installation of USARSim is completed.

6.1.3 UAV Control Center

In order to make the modifications applied to USARSim in the course of this diploma thesis available the usarsim folder on the CD-Rom coming with this work has to be copied into the installation directory of USARSim. If Windows asks whether already existing files should be overwritten by new files with the same name this has to be answered with *yes* again. Once the files have been copied the USARSim installation needs to be recompiled, which is done by invoking the *make_clean.bat* batch program lying in the UDK installation directory. After the compilation has finished all modifications needed for the control program should have been successfully applied.

The control program can then either be copied from the *debug* directory in the Visual Studio project folder to any desired directory on the PC and executed, or directly started from the CD-Rom. In order to start the Unreal Server and USARSim and load a virtual environment one of the map files in the directory *C:/UDK-YYYY-MM/USARRunMaps* needs to be double-clicked. Once the virtual environment has been started the control program can be used to spawn a quadrocopter assign missions and control the UAV.

In order for the program to run the *Microsoft Visual C++ 2010 Redistributable* libraries coming with Visual Studio 2010 or separately downloadable from the Microsoft website need to be installed on the system.

6.2 Control program usage

In this section a description of the main window and its elements will be given followed by a basic introduction into the program's usage.

6.2.1 The main window

The main window of the control program with its controls is depicted in figure 6.1. Its content can be divided into the six different parts shown in figure 6.1 of which each one has a different purpose. In the following these will be briefly presented.

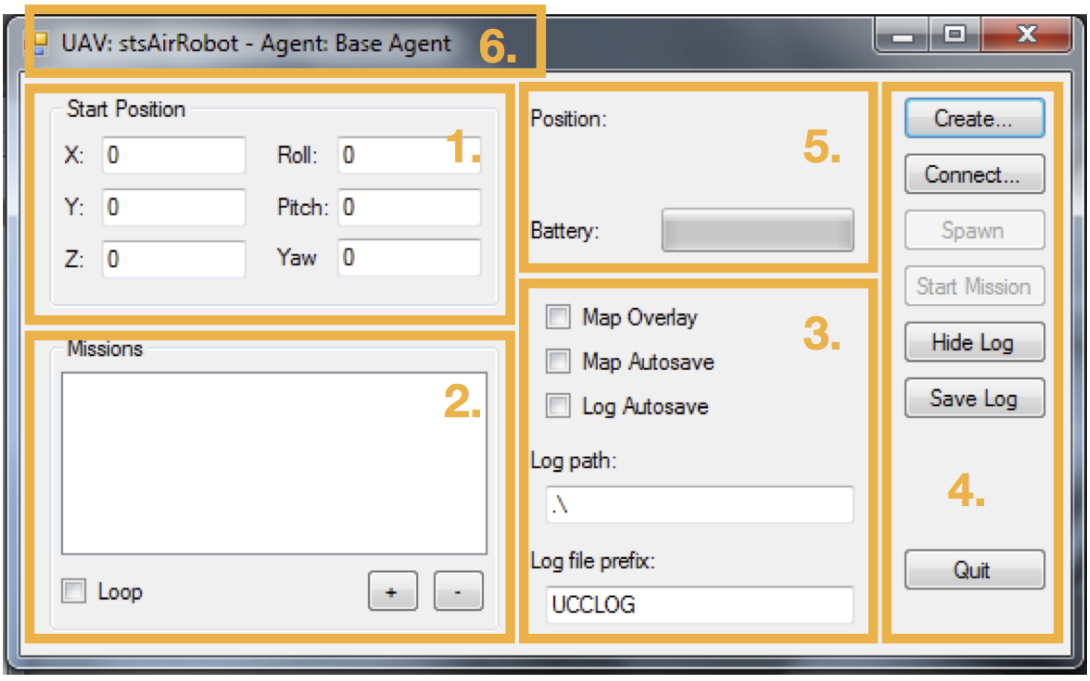

Figure 6.1: The main window and its elements

Area 1: This area takes the coordinates at which a quadrocopter model is to be spawned in the virtual environment as well as its orientation. The coordinates are measured in meters from the origin of the coordinate system of the map being in use when spawning the robot. The orientation angles are measured in degrees.

Area 2: This area is used to add missions to the agent being in control of the quadrocopter. The '+' button will open the dialog window depicted in figure 6.2 in which the mission type can be selected from a drop-down list and the coordinates can be added to or deleted from the mission with the '+' and '-' button respectively. Any missions added to the agent will be displayed with coordinates in the white text box. In order to remove a mission from the text box it has to be highlighted by clicking on it and then the '-' button has to be pressed. The *Loop* check-box indicates whether the missions are to be restarted again once they have been finished. If the check-box is checked the missions will be executed over and over again until

either the check-box is unchecked or the program is ended.

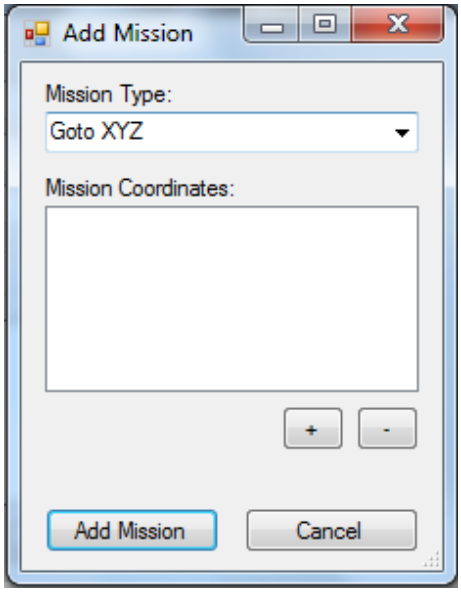

Figure 6.2: AddMission dialog window

Area 3: The controls in this area are used to define the logging behaviour of the control program. The *Map overlay* checkbox defines whether the map drawn in each run of an experiment is to be reset with the start of the next run or if all tests should be drawn onto the same map. If the *Map autosave* and *Log autosave* check-boxes are checked then after every finished experiment the contents of the log window and the map window will be saved to disk and afterwards reset, unless the *Map overlay* checkbox is checked in which case the map will not be reset. The *Log path* text-box allows to define a path under which the files are to be saved and the *Log file prefix field* allows to define a prefix for all files created during the experiment. All files will then be created in the specified directory with the provided prefix and a trailing number indicating the number of iteration of the experiment. The maps created will be saved as bitmap files, while the log files are saved in the *rich text format*. If the check-boxes for autosave are unchecked no files will be created and the log will be written continuously to.

Area 4: This area provides the main functionality of the program. The *Create...* button opens the dialog window depicted in figure 6.3 which allows to select one of the UAV models that are known to the application and one of the agent types that have been implemented in order to control the UAV. Furthermore server configuration data has to be provided in this dialog to setup the Unreal Server that the controller program is to connect to. The *Connect...* button will then connect the control program to the server after which message reception is turned on. After the connection to the server has been established the quadrocopter model can be spawned in the virtual environment with the *Spawn* button at the location specified before in the *start position area*. If missions have been specified then the *Start*

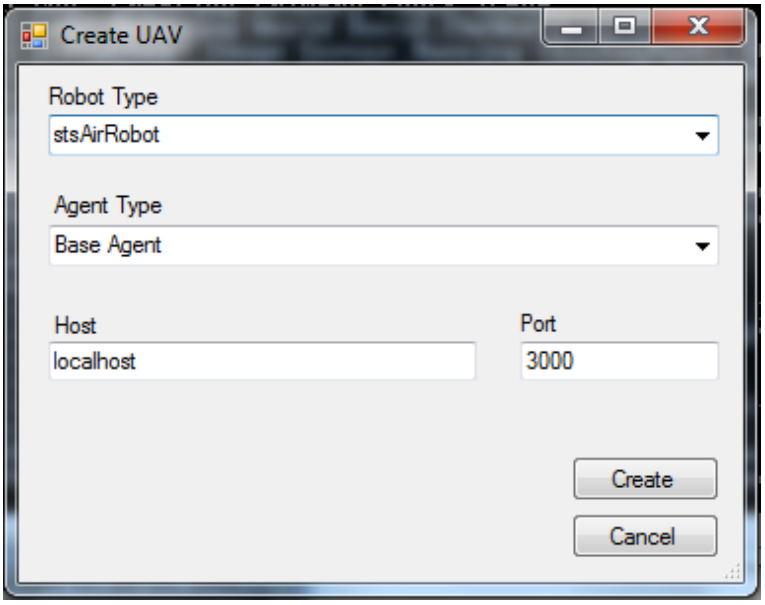

Figure 6.3: CreateUAV dialog window

Mission button will start the agent which will then begin to execute its missions. The *Hide log* button can be used to hide the log window in case its output is of no interest and the *Save Log* button can be used to save the current content of the log window to an arbitrary file. The *Quit* button will disconnect the control program from the server, stop all running threads and end program execution.

Area 5: This area provides information about the quadrocopter's status by displaying its current position and battery charge level. This information is constantly updated as soon as new sensor data from the UAV model is available.

Area 6: The last area in the main window is found in the title bar which displays the type of UAV model that is being controlled in the simulation environment as well as the type of agent that is controlling it.

6.3 How to create new Agent classes

In order to implement any control algorithms in the framework provided it is necessary to create a new agent class which is derived from the *Agent* base class. This is basicly a three step process. First of all the new class needs to be implemented, then the class has to be made available in the user interface so that the user can select it as the controlling instance for the quadrocopter and then the *uccController* class needs to be adapted so it knows when to create an object of the new class. The detailed procedure of adding a new agent is presented below in this section. After performing the steps described here the new agent type is selectable in the user interface and can take control over the quadrocopter in the simulated environment.

6.3.1 Deriving a new Agent class

A new Agent is basicly implemented by creating a subclass of the Agent class. The uccController class calls the start() method of all Agent subclasses to start the agent's execution and the stop() method to stop it. So in order for the new deviated agent class to do something useful it is necessary to overwrite the these two functions which are defined as virtual in the base class and have no real functionality implemented. It is probably a good idea to have the start() method start a thread reading sensor values from the UAV object and generating control commands for it. The stop() method is then supposed to end any running threads when called. Furthermore the addMission() method needs to be overwritten. Any new subclass of an agent class can possibly execute different types of missions and therefore needs a different implementation of this method. On top of overwriting these three methods any number of new methods and attributes can be added to a new agent subclass.

6.3.2 Make the class appear in the UI

In order to make the new agent type selectable in the user interface the *Create UAV Dialog* has to be opened in the Visual Studio Designer and an entry has to be added to the *ComboBox* found under the *Agent Type* label. Any appropriate name not already taken can be chosen here.

6.3.3 Adapt uccController

To adapt the controller the *uccController.cpp* file needs to be edited. First of all the header for the newly implemented *Agent* class has to be included at the top of the file. Then at the bottom of the *createUAV()* method there is a series of if-statements that check for the agent name provided by the ComboBox just edited in the user interface. A new if-branch has to be added there resembling the old ones but this time checking if the aiType variable equals the name just entered for the agent in the user interface. In the if statement itself the agent variable gas to be assigned an object of the new created *Agent* subclass as can be seen the listing below. The code depicted here adds a new *Agent* subclass of type BetterAgent to the series of if statements and the name of the exemplary agent selected in the user interface is *"My Better Agent"*.

```
// Check for known Agent types
if(aiType=="Base Agent") {
        agent = gcnew BaseAgent(uav);
}
if(aiType=="Base Agent 2") {
        agent = gcnew BaseAgent2(uav);
}
. . . . .
if(aiType=="My Better Agent") {
        agent = gcnew BetterAgent(uav);
}
```

Chapter 7

Conclusion and future work

The control program implemented in this work, the provided quadrocopter configuration and sensors together with USARSim and the underlying Unreal Engine constitute a flexible, easy to use and easily expandable simulation environment for UAVs. The modular design of the agent in the program's architecture and the applied principle of polymorphism allow for development of new agents completely independent from the other parts of the control program. This way the remaining parts of the control program can serve as a basis and a foundation for the development of new agents which implement new behaviors, control algorithms and learning algorithms. The programmer creating the new agents does not need to handle any code except the code needed for the algorithms implemented in the agent. In particular does he not have to deal with any programming of network communication, message parsing or user interfaces as this has already been taken care of in the control program.

The sensor configuration provided with the quadrocopter model coming with the control program is broad enough to allow for many different behavior implementations without the need for any modifications to the quadrocopter model or sensors thus introducing numerous opportunities for further development. With the possibility to build new sensors in USARSim the provided quadrocopter configuration can be extended and virtually any scenario can be implemented, considering certain adjustments. Given the control program's capability to run a set up scenario virtually any given number of times in the exact same way the program can also be used for the training and evaluation of learning algorithms before they are applied to real hardware.

With the Unreal Development Kit being still in its beta phase and USARSim for UDK still standing at the beginning of its transition from the older UT3 to the new UDK a lot of further progress in the development of USARSim can be expected and the simulation environment should not be outdated any time soon. Although the simulation already produces visually highly appealing graphics and highly realistic physical behavior these can be expected to further increase with new upcoming versions of the two software packages. Furthermore it is needless to say that with *Epic Games* allowing the use of the Unreal Development Kit for absolutely free if used for noncommercial purposes and USARSim being also free and open source the simulation environment realized in this thesis comes at no costs whatsoever.

Bibliography

[1] M. Buehler, K. Iagnemma, and S. Singh. *The 2005 Darpa Grand Challenge: The Great Robot Race*. Springer Tracts in Advanced Robotics. Springer, 2007.

[2] M. Buehler, K. Iagnemma, and S. Singh. *The Darpa Urban Challenge: Autonomous Vehicles in City Traffic*. Springer Tracts in Advanced Robotics. Springer, 2009.

[3] United States. Dept. of Defense. Office of the Secretary of Defense. *Unmanned Aerial Vehicles Roadmap, 2000-2025*. Department of Defense, Office of the Secretary of Defense, 2001.

[4] United States. Dept. of Defense. Office of the Secretary of Defense. *Unmanned Aircraft Systems Roadmap, 2005-2030*. Department of Defense, Office of the Secretary of Defense, 2005.

[5] R. Austin. *Unmanned Aircraft Systems: UAVs Design, Development and Deployment*. Aerospace Series. John Wiley & Sons, 2010.

[6] Michael Reckhaus, Nico Hochgeschwender, Jan Paulus, Azamat Shakhimardov, and Gerhard K. Kraetzschmar. An overview about simulation and emulation in robotics. *Proceedings of SIMPAR 2010 Workshops, Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS*, pages 365–374, nov 2010.

[7] Robert Kesten. Data fusion of accelerometric, gyroscopic, magnetometric and barometric information for attitude and altitude estimation of an unmanned quadrocopter.

[8] R. Büchi. *Fascination Quadrocopter*. Books on Demand GmbH, 2011.

[9] Swiss Federal Institute of Technology Zurich. Flying machine arena. http://www.idsc.ethz.ch/Research_DAndrea/FMA/. [Online; 2012].

[10] Abraham Bachrach, Rujijie He, and Nicholas Roy. Autonomous flight in unknown indoor environments. *International Journal of Micro Air Vehicles*, 1(4):217–228, dec 2009.

[11] Axel Bürkle, Florian Segor, and Matthias Kollmann. Towards autonomous micro uav swarms. *J. Intell. Robotics Syst.*, 61(1-4):339–353, January 2011.

[12] Horia Ionescu. Wikimedia commons. http://commons.wikimedia.org/wiki/File:6DOF_en.jpg. [Online; 2012].

[13] AirRobot Australasia Pty Ltd. *AirRobot - Micro Unmanned Aerial Vehicle With Autonomous Flight and Navigation Capabilities and Modular Payloads*, 2007.

[14] AirRobot GmbH & Co KG. Airrobot homepage. http://www.airrobot.de/. [Online; 2012].

[15] Microdrones GmbH. Microdrones homepage. http://www.microdrones.com/. [Online; 2012].

[16] Michael Lewis and Jeffrey Jacobson. Game engines in scientific research. *Commun. ACM*, pages 27–31, 2002.

[17] Epic Games. Unreal script reference. http://udn.epicgames.com/Three/UnrealScriptReference.h [Online; 2012].

[18] Usarsim project homepage. http://sourceforge.net/apps/mediawiki/usarsim/. [Online; 2012].

[19] Andrew N. Marshall. Gamebots project homepage. http://gamebots.sourceforge.net/. [Online; 2012].

[20] Gal A. Kaminka, Manuela M. Veloso, Steve Schaffer, Chris Sollitto, Rogelio Adobbati, Andrew N. Marshall, Andrew Scholer, and Sheila Tejada. Gamebots: a flexible test bed for multiagent team research. *Commun. ACM*, 45(1):43–45, January 2002.

[21] I. Maza and A. Ollero. Multiple UAV cooperative searching operation using polygon area decomposition and efficient coverage algorithms. In *Proceedings of the 7th International Symposium on Distributed Autonomous Robotic Systems*, pages 211–220, Toulouse, France, 2004.

[22] C. E. Palazzi, C. Roseti, M. Luglio, M. Gerla, M. Y. Sanadidi, and J. Stepanek. Satellite coverage in urban areas using unmanned airborne vehicles (uavs). *Vehicular Technology Conference*, 5:2886–2890, may 2004.

[23] Crossbow Technology. Crossbow homepage. http://www.xbow.com/index.html. [Online; 2012].

[24] D. Johnson, Y. Hu, and D. Maltz. The dynamic source routing protocol (dsr) for mobile ad hoc networks for ipv4. *RFC 4728*, 2007.

[25] Gerry Siegemund. Auris - autonomous robot interaction simulation.

[26] Markus Achtelik, Abraham Bachrach, Rujijie He, Samuel Prentice, and Nicholas Roy. Autonomous navigation and exploration of a quadrotor helicopter in gps-denied indoor environments. In *2009 First Symposium on Indoor Flight Issues*, jul 2009.

[27] A. D. King. Inertial navigation: Forty years of evolution. *GEC review*, pages 140–149, January 1998.