Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Intelligente Systeme und Robotik

Stabilisierung und Verbesserung des Multicopter-Flugverhaltens anhand einer Simulation

Martin Kühn
Matrikelnummer: 4367487
martin.kuehn@fu-berlin.de

Betreuer: Prof. Dr. Marco Block-Berlitz

Abgabedatum: 29. August 2013

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis oder ggf. in Fußnoten angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, den 28. August 2013

(Martin Kühn)

Inhaltsverzeichnis

1 Motivation und Einführung

1.1 Projekt Archaeocopter

Diese Bachelorarbeit ist im Rahmen des Forschungsprojekt *Archaeocopter*[1] an der Freien Universität Berlin entstanden. Die Intention dieses Projektes ist es, Ansätze zu finden, um mithilfe von Multicoptern Archäologen bei der Datenerfassung und -Auswertung während archäologischer Ausgrabungen zu unterstützen. Grundidee hierbei ist, dass der Archäologe den Multicopter autonom über die Ausgrabungsstätte fliegen lassen kann, eine am Fluggerät angebaute Kamera tätigt währenddessen in einem vorgegebenem Schema selbstständig Foto- bzw. Videoaufnahmen. Die aufgenommen Bilder lassen sich für die 3D-Rekonstruktion nutzen: Eine Software konstruiert anhand der Bilder ein maßstabsgetreues und texturiertes dreidimensionales Abbild der Ausgrabungsstätte. Für Archäologen ergibt sich hieraus eine Vielzahl an Vorteilen: Zum einen ist es möglich die Ausgrabungsstätte objektiv und präzise zu visualisieren, da sich das 3D-Modell aus allen Richtungen betrachten lässt und sich virtuell frei darin bewegt werden kann - dies stellt eine große Verbesserung gegenüber zweidimensionalen, eher subjektiven Bildern wie Fotos und Zeichnungen dar. Zum anderen lässt sich durch Gegenüberstellung von zu verschiedenen Zeitpunkten aufgenommenen 3D-Modellen augenfällig der Fortschritt der Ausgrabung darstellen. Nicht zuletzt bietet der Einsatz von Multicoptern die Möglichkeit Bilder und Modelle von unzugänglichen Bereichen der Ausgrabungsstätte zu erstellen.

1.2 Motivation

Beim Einsatz als bewegliche Kamera werden an den Multicopter viele verschiedene Anforderungen gestellt. Unter anderem ist eine hohe Akkukapazität, welche eine lange Flugdauer ermöglicht, sowie eine hohe Zuladungskapazität, um die Kamera mitzutransportieren vonnöten. Ein wichtiger Aspekt ist hierbei jedoch auch, dass der Multicopter während des Fluges möglichst unbewegt und stabil in der Luft verharren kann, damit aus einem passenden Winkel scharfe Fotos und unverwackelte Videos aufgenommen werden können. Zur Stabilität des Multicopters gehört insbesondere auch die Unempfindlichkeit gegen Wind.
Im Folgenden sollen Ansätze untersucht werden, die Stabilität des Multicopters zu verbessern, sie soll Untersuchungsgegenstand dieser Abschlussarbeit sein. Hierzu wird untersucht, inwieweit die Stabilität von verschiedenen Eigenschaften des Multicopters abhängt, wie unter anderem von der Anzahl der Propeller oder vom Neigungswinkel der Propeller. Es ist anzumerken, dass Änderungen am Aufbau eines fertigen Multicopters oftmals nur aufwändig zu realisieren sind und hier ein hohes Risiko besteht den Multicopter zu beschädigen. Bei bereits konstruierten Modellen lässt sich beispielsweise der Neigungswinkel nicht umstandslos verändern. Ebenso müsste zur Untersuchung der Auswirkung verschiedener Anzahlen und Anordnungen der Rotoren jeweils ein neuer Multicopter bzw. Multicoptertyp erworben werden. Daher wird im Rahmen dieser Arbeit eine Computersimulation entwickelt, welche es ermöglichen soll unkompliziert, schnell und unfallfrei Anpassungen an dem virtuellem Multicopter vorzunehmen.

1.3 Aufbau der Arbeit

In Abschnitt 2 findet zunächst eine Einführung in die Grundlagen und physikalischen Hintergründe von Multicoptern statt, was für das Verständnis der weiteren Abschnitte unabdingbar ist, außerdem werden Ansätze verwandter Arbeiten genannt. Der Begriff der Stabilität wird genauer erklärt, desweiteren wird auf die Rauschfunktion Simplex-Noise eingegangen, welche später für die Windsimulation verwendet wird. Ein wichtiger Aspekt, welcher bei der Simulation des Multicopters umgesetzt werden muss ist gleichermaßen, dass

[1] `http://www.archaeocopter.de/`

das Fluggerät fähig ist an einer festen Stelle im Luftraum zu schweben, hierfür werden PID-Regler eingesetzt. Die Computersimulation wurde unter Zuhilfenahme der Softwaresammlung *Microsoft Robotics Developer Studio* erstellt, auch die hierfür nötigen Grundlagen werden im Abschnitt genannt. In Abschnitt 3 wird eine Übersicht über das Projekt geboten, es werden die in dieser Arbeit untersuchten Ansätze zur Verbesserung der Stabilität vorgestellt, ebenfalls wird eine Übersicht über die Softwarearchitektur der Computersimulation gezeigt. Abschnitte 4, 5 und 6 behandeln Einzelaspekte der Simulation verschiedener Multicoptertypen, die Aspekte der Windsimulation sowie die softwaretechnische Umsetzung der PID-Regler. In Abschnitt 7 werden, nachdem ein Maß für die Stabilität festgelegt ist anhand verschiedener im Simulator durchgeführter Experimente die Verbesserungsansätze überprüft und in einem Fazit bewertet. Zusammengefasst werden die Ergebnisse der Arbeit in Abschnitt 8, ebenfalls wird ein Ausblick über weiter zu untersuchende Aspekte geboten und eventuell weiterführende Ansätze genannt.

2 Theorie und verwandte Arbeiten

2.1 Multicopter

Bei Multicoptern handelt es sich um unbemannte Luftfahrzeuge, im englischen UAVs (Unmanned Aerial Vehicles) genannt, welche zumeist über drei oder mehr in einer Ebene angeordnete Propeller verfügen. Sie gehören zur Klasse der VTOL-Luftfahrzeuge (Vertical Take Off and Landing) [TM06, S. 1], die Schubkraft aller Propeller wirkt vertikal nach oben und insgesamt senkrecht zum Grundaufbau des Flugkörpers. Als häufigste Konstruktionsform anzutreffen ist eine Rahmenplattform bestehend aus einem zentral gelegenem Element, von welchem zumeist gleichwinklig verzweigt mehrere Stäbe, *Ausleger* genannt, ausgehen. In dem zentralen Element sind Steuerelektronik und Energiezellen enthalten, an den davon ausgehenden Auslegern sind jeweils Motoren und Propeller angebracht. Dieser Grundaufbau wird im Folgenden *Flugplattform* genannt.

Im Gegensatz zu anderen Luftfahrzeugen haben Multicopter den Vorteil, dass sie relativ preiswert zu erwerben sind - hier gibt es eine große Auswahl kommerzieller Produkte (beispielsweise die *Parrot AR.Drone* welche in Abbildung 1 dargestellt ist, sowie der *DJI Phantom DIY*). Durch ihren einfachen mechanischen Aufbau [ANT11, S. 1195] sind sie jedoch auch vergleichsweise einfach selbst konstruierbar - im Internet ist eine große Anzahl Bauanleitungen, fertiger Umsetzungen und Implementierungen von Steuersoftware zu finden. Erwähnenswert sind hier unter anderem die Projekte *OpenPilot*[2], in dessen Rahmen eine Open-Source Steuersoftware für Multicopter entwickelt wird, sowie *MikroKopter*[3], bei welchem Bausätze und Tutorials zur Erstellung von Flugplattformen angeboten werden. Durch ihre physikalischen Gegegebenheiten sind Multicopter, auch dank ihrer oft nur geringen Größe, einfach und präzise steuerbar. Sie verfügen über eine hohe Manövrierbarkeit [ROR10, S. 29] und sind im Gegensatz zu Flugzeugen in der Lage vergleichsweise

Abbildung 1: Standardaufbau eines Multicopters, hier vom Typ Quadrocopter. Abgebildet ist ein kommerzielles Produkt, die *AR.Drone 2* von der Firma *Parrot*

[2] `http://www.openpilot.org/`
[3] `http://www.mikrokopter.de/de/startseite/`

unbewegt in der Luft zu schweben, was sie für viele Anwendungsfälle einsetzbar macht [ANT11, S. 1195].
Die geringe Größe bietet ebenfalls den Vorteil, dass sie gut in Gebäuden bzw. engen Räumen eingesetzt werden können [CS12, S. 2]. Multicopter verfügen im Vergleich zu ihrem Gewicht über eine hohe Zuladungskapazität und machen sie damit sehr geeignet als mobile Flugplattformen. Sie sind gut nutzbar, um auf einfachem Wege Foto- bzw. Videoaufnahmen von beliebiger Position im Luftraum aus zu tätigen. Dies kann praktischen Nutzen haben: Das Forschungsprojekt *Agrocopter*[4] untersucht beispielsweise Ansätze Multicopter zur zielgerichteten Düngung in der Landwirtschaft einzusetzen, um Überdüngung zu vermeiden. Denkbar ist auch der Einsatz von UAVs bei der Waldbrandbekämpfung, Alexis et al. haben hierzu in [ANTD09] Ansätze präsentiert, mittels durch Multicopter erzeugter Luftbilder und erweiterter Sensorik die Ausbreitungsrichtung des Feuers vorauszuschätzen.

2.1.1 Flugdynamik

Bei propellergetriebenen Flugkörpern stellt oft ein Problem dar, dass durch die Rotation der Propeller ein Drehmoment in die entgegengesetzte Richtung auf die tragende Flugplattform übertragen wird. Dies hat zur Folge, dass die Flugplattform während des Fluges beginnt selbst um die eigene Achse zu rotieren. Zu diesem Zweck ist bei Helikoptern oft ein zweiter Propeller, zumeist senkrecht als Heckrotor angebracht, welcher diesem Drehmoment entgegensteuert, sodass die gewünschte Flugrichtung beibehalten werden kann. Eine herausragende Eigenschaft des Multicopters ist, dass diesem Effekt der Drehmomentübertragung einfach entgegengewirkt werden kann: Bei Multicoptern kann jeder Propeller einzeln angesteuert werden, die Schubkraft der Propeller kann proportional über die Drehzahl der Motoren erhöht oder verringert werden [MKC12, S. 20]. Wird eine gerade Propelleranzahl verwendet, kann diese Fähigkeit angewendet werden, indem die eine Hälfte der Propeller im Uhrzeigersinn rotiert und die andere Hälfte der Propeller mit gleicher Geschwindigkeit gegen den Uhrzeigersinn (s. Abbildung 2). Die dadurch von den Propellern auf die Flugplattform übertragenen Drehmomente heben sich auf diese Weise gegenseitig auf und der Multicopter behält die Orientierung bei, ohne dass ein Heckrotor nötig ist [ebenda].

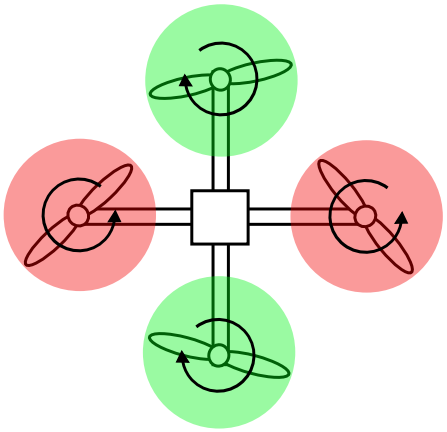

Abbildung 2: Drehmomentausgleich bei einem Quadrocopter durch gegenläufige Rotationsrichtungen der Propeller

Ein weiterer Vorteil, der sich aus diesem Prinzip ergibt ist, dass im Gegensatz zum Helikopter alle Propeller für den Antrieb genutzt werden. Weiterhin lässt sich durch Veränderung der Propellergeschwindigkeiten der Drehmomentausgleich vorübergehend deaktivieren, um auf diese Weise die Fähigkeit zum Drehen um die eigene Achse (Gieren) vergleichsweise einfach zu ermöglichen. Es ist ebenfalls erwähnenswert, dass bei Multicoptern im Gegensatz zu Hubschraubern keine bestimmte Vorwärtsrichtung vorgeben ist, da Multicopter auch ohne vorherige Drehung jederzeit in alle Richtungen des Raums bewegt werden können. Meistens wird willkürlich ein beliebiger Ausleger bzw. eine beliebige Richtung als „vorne" definiert, durch eine spezielle Markierung wird dann angezeigt in welche Richtung der Multicopter fliegt, wenn auf der Fernbedienung der Befehl zum Vorwärtsflug gegeben wurde. An den Multicopter angebrachte Kameras sind meistens ebenfalls mit Blick in

[4] *http://agricopter.de/*

Vorwärtsrichtung angebracht. In den folgenden Abbildungen dieses Abschnitts wird davon ausgegangen, dass ein rot markierter Ausleger die Vorwärtsrichtung vorgibt. Für die kommenden Beschreibungen wird ein rechtshändiges, dreidimensionales Koordinatensystem entsprechend Abbildung 3 eingeführt, dieses entspricht auch der Betrachtungsweise in der später vorgestellten Computersimulation. Die Höhe wird hierbei durch die Y-Achse gegeben, steigende Höhe bedeutet also einen ebenfalls steigenden Y-Wert, die XZ-Ebene bildet die horizontale Grundfläche des Bodens. Der Multicopter verfügt über ein körpereigenes Koordinatensystem bestehend aus Nick-, Gier- und Rollachse (s. Abbildung), welches an die Ausrichtung der Flugplattform gebunden ist. Es wird davon ausgegangen, dass der Multicopter in der Startkonfiguration aufrecht steht, die Gierachse also parallel zur Y-Achse ausgerichtet ist. Der Multicopter ist so orientiert, dass die o.g. Vorwärtsrichtung entlang der positiven Richtung der Rollachse verläuft.
Die Flugplattform kann entlang aller Freiheitsgerade rotiert werden, durch eine Neigung der Flugplattform entlang Nick- und Rollachse wird ein Flug in alle Richtungen ermöglicht. Die Rotationen um die jeweiligen Achsen wird durch unterschiedliche Ansteuerung der einzelnen Propeller ermöglicht, was anhand der folgenden Punkte erörtert werden soll (zur Beschreibung wird von der o.g. Startkonfiguration ausgegangen):

Steigflug und Sinkflug Um den Multicopter steigen (Bewegung in positive Richtung der Gierachse) oder sinken (Bewegung in negative Richtung der Gierachse) zu lassen muss gleichsam die Drehgeschwindigkeiten aller Propeller erhöht bzw. verringert werden. Je höher die Drehgeschwindigkeit der Propeller ist, desto höher ist die Kraft, welche an den einzelnen Propellern senkrecht zur Grundfläche des Multicopters anliegt. In Abbildung 4 sind diese Kräfte exemplarisch an einem Quadrocopter veranschaulicht. Damit der Quadrocopter in der Luft mit konstanter Höhe schwebt, muss die Summe der an den einzelnen Propellern wirkenden Kräfte (Gesamtpropellerkraft) $F_{P1} + ... + F_{P4}$ genau gleich der Gewichtskraft F_G sein. Für einen Steigflug muss die Gesamtpropellerkraft größer als die Gewichtskraft sein, für einen Sinkflug entsprechend kleiner als die Gewichtskraft.

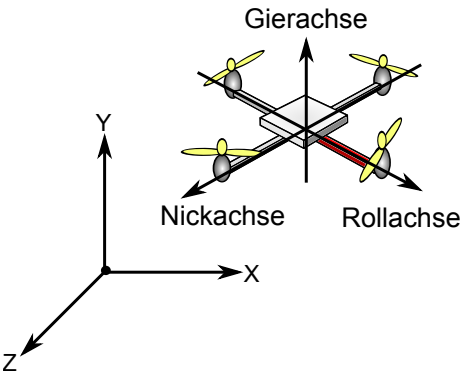

Abbildung 3: Das in dieser Arbeit verwendete dreidimensionale Koordinatensystem

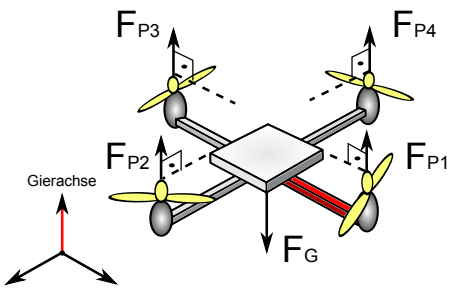

Abbildung 4: Steigflug und Sinkflug

Nicken Mit diesem Begriff sei im Folgenden eine Rotation um die Nickachse bezeichnet. Durch Verringerung der Drehgeschwindigkeit der vorderen Propeller und Erhöhung der Drehgeschwindigkeit der hinteren Propeller lässt sich der Flugkörper vowärts nicken. Dies bewirkt einen Flug in die Vorwärtsrichtung entlang der positiven Rollachse, da durch das Nicken die Kraftvektoren F_{P1} bis F_{P4} rotiert werden und so Komponenten der Kraft in die Vorwärtsrichtung wirken. Wird dieses Prinzip umgekehrt, der vordere Propeller beschleunigt und der hintere Propeller verlangsamt, lässt sich der Multicopter nach hinten nicken, was einen Flug in die

rückwärtige Richtung (negative Richtung der Rollachse) ermöglicht. In Abbildung 5 ist dieser Vorgang anhand des Beispiels eines Quadrocopters skizziert: Durch Erhöhung von F_{P1} und Verringerung von F_{P3} findet ein rückwärtsgerichtetes Nicken statt.

Gieren Mit diesem Begriff sei im Folgenden eine Rotation um die Gierachse gemeint, hierdurch ändert sich die Richtung der Vorwärtsmarkierung. Um den Multicopter zu drehen wird die eine Hälfte der gegenüberliegenden Propeller beschleunigt und die andere Hälfte der gegenüberliegenden Propeller verlangsamt. Durch diese Änderung wird der zu Anfang dieses Abschnitts genannte Drehmomentausgleich außer Kraft gesetzt und der Multicopter dreht sich um die Y-Achse. Die Rotationsrichtung des Multicopters um die Gierachse ist abhängig davon, welche Propeller einer bestimmten Drehrichtung verlangsamt bzw. beschleunigt werden. Dieser Vorgang ist in Abbildung 6 veranschaulicht (die dickeren Kreislinien sollen die größere Kraft andeuten): Durch Verringerung der Drehgeschwindigkeit von $P1$ und $P3$ sowie Erhöhung von $P2$ und $P4$ dreht sich der Quadrocopter gegen den Uhrzeigersinn um die Gierachse.

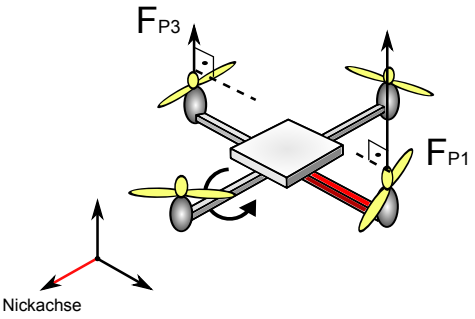

Abbildung 5: Rotation um die Nickachse bei einem Quadrocopter

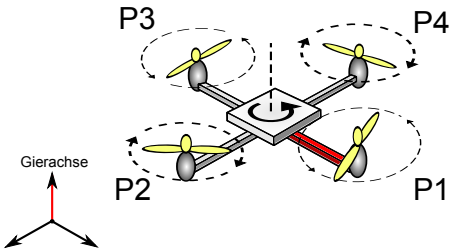

Abbildung 6: Rotation um die Gierachse bei einem Quadrocopter

Rollen Mit diesem Begriff sei im Folgenden eine Rotation um die Rollachse gemeint. Es wird hierbei das selbe Prinzip wie beim Nicken angewendet, allerdings werden hier die anderen Propeller beschleunigt bzw. verlangsamt, d.h. die von der Vorwärtsrichtungs aus gesehen seitzlich platzierten Propeller. Hierdurch wird der Multicopter zur Seite gekippt, was einen Seitwärtsflug in die entsprechende Richtung (positive oder negative Richtung der Nickachse) ermöglicht, da das Rollen ebenfalls eine Rotation der Kraftvektoren verursacht. Dieser Vorgang ist in Abbildung 7 veranschaulicht: Durch Verringerung von F_{P2} und Erhöhung von F_{P4} wird der Multicopter in die angegebene Richtung gekippt.

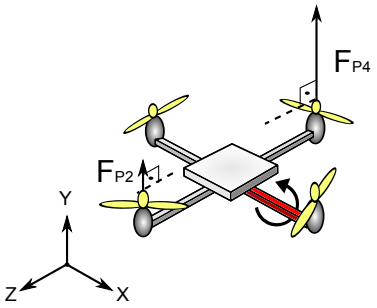

Abbildung 7: Rotation um die Rollachse bei einem Quadrocopter

Durch Kombination der verschiedenen Achsenrotationen kann der Multicopter in sämtliche Richtungen bewegt werden. Oft wird durch die Steuerelektronik ein vereinfachter Zu-

griff auf diese Rotationen geboten, indem an diese einfach die Steuerbefehle zum Nicken, Gieren und Rollen gesandt werden können und die Elektronik automatisch die jeweiligen Propeller beschleunigt bzw. verlangsamt.

2.1.2 Anzahl der Rotoren

Die im oberen Abschnitt gezeigten Methoden zur Steuerung des Multicopters lassen sich auf alle Multicopter mit einer geraden Propelleranzahl übertragen, bei einer ungeraden Anzahl Propeller treten allerdings zusätzliche Schwierigkeiten auf. Beispielsweise funktioniert die im vorigen Abschnitt genannte Funktionsweise zum Drehmomentausgleich bei einer ungeraden Anzahl Propeller nicht: Die Propeller können in diesem Fall nicht in zwei gleichwertige Hälften aufgeteilt werden, es müssen andere Überlegungen getroffen werden. Dennoch werden in dieser Abschlussarbeit auch Multicopter mit ungerader Propelleranzahl berücksichtigt, da sie zumindest theoretisch modellierbar sind.
Bei dem Wort Multicopter handelt es sich lediglich um einen Oberbegriff, je nach Anzahl der Propeller haben die verschiedenen Multicopter-Typen unterschiedliche Bezeichnungen, welche in der folgenden Auflistung näher aufgeführt sind:

Tricopter Verfügt über 3 Propeller. Es existieren funktionierende Implementierungen dieser Bauart, beispielsweise wurde von der Universität des Saarlandes ein funktionierender Prototyp entwickelt[5]. Die Stabilisierung gegenüber auf den Flugkörper gewirkten Drehmomenten findet bei diesem Modell mittels periodischem Schwenken der Rotoren sowie über gleichmäßige Veränderung der Motordrehzahlen statt. Diese Implementierung bringt den Vorteil, dass für einen Flug entlang der Freiheitsgerade keine Achsenrotation stattfinden muss.

Quadrocopter Verfügt über 4 Propeller. Dieser Multicopter-Typ wird am häufigsten eingesetzt.

Pentacopter Verfügt über 5 Propeller. Es wurde im Rahmen der Recherche dieser Arbeit keine Implementierung dieser Bauart gefunden.

Hexacopter Verfügt über 6 Propeller.

Heptacopter Verfügt über 7 Propeller. Es wurde im Rahmen der Recherche dieser Arbeit keine Implementierung dieser Bauart gefunden.

Octocopter Verfügt über 8 Propeller.

In der Praxis werden ebenfalls Multicopter mit 12 Rotoren eingesetzt, diese werden in dieser Arbeit allerdings vernachlässigt, da deren Einsatz nur in seltenen Fällen Anwendung findet. Durch Steigerung der Propelleranzahl erhöht sich das Gewicht des UAVs, da zusätzliche Rotorblätter, Motoren und ggf. zusätzliche Ausleger erforderlich sind. Eine Mehrzahl an Propellern hat ebenfalls einen höheren Stromverbrauch und daher eine kürzere Maximalflugdauer zur Folge. Es bietet jedoch auch Vorteile eine größere Anzahl Propeller zu verwenden, da sich durch jeden zusätzlichen Motor und Propeller die Schubkraft steigern lässt. Ebenfalls ergibt sich der Vorteil der Redundanz: Bei Ausfall eines Motors kann, sofern genug andere Propeller vorhanden sind, der Flug trotz des Motordefekts sicher fortgesetzt werden.

2.1.3 Neigungswinkel der Propeller

Der Neigungswinkel ψ der Propeller ist im Folgenden definiert als der Winkel zwischen der Kraftwirkung des Propellers und der Senkrechte zur Flugplattform (s. Abbildung 8).

[5] `http://www.golem.de/news/tricopter-saarbruecker-drohne-fliegt-mit-drei-rotoren-1304-98694.html`

- Ein positiver Neigungswinkel bedeutet, dass alle Propeller nach innen, in Richtung Zentrum des Multicopters gebogen sind. Je größer der Neigungswinkel, desto stärker ist auch die Neigungs nach innen. Das hat zur Folge, dass nicht mehr die volle Kraft des Propellers nach oben wirkt, sondern je nach Größe des Neigungswinkels eine Komponente der Kraft Richtung Mitte des Multicopters. Durch die Symmetrie gleichen sich bei einer positiven Neigung die horizontal wirkenden Kräfte des Multicopters aus, so dass ein stabiler Steig- und Sinkflug stattfinden kann.

- Ein Neigungswinkel von 0° heißt, dass die Kraft genau vertikal nach oben wirkt, senkrecht zur Flugplattform. Dies ist die Standardkonfiguration der meisten Multicotper, sämtliche Kraft wird für den Steigflug verwendet.

- Ein negativer Neigungswinkel heißt, dass alle Propeller von der Mitte weg nach außen gebogen sind. Je kleiner der Neigungswinkel ist, desto weiter sind die Propeller weggebogen. Auch hier wirkt nicht mehr die volle Kraft des Propellers nach oben, sondern die horizontale Komponente der Kraft weg vom Multicopter. Bei einer negativen Neigung gleichen sich auch hier durch die Symmetrie der Flugplattform die Kräfte des Multicopters aus, so dass ein stabiler Steig- und Sinkflug stattfinden kann.

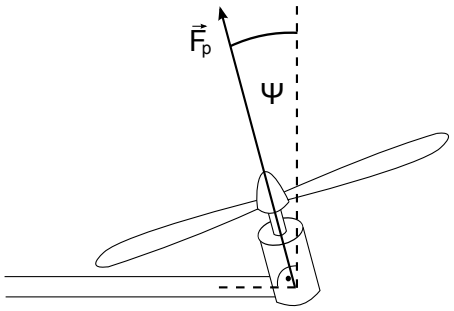

Abbildung 8: Neigungswinkel der Propeller

2.1.4 Lageregelung und automatische Steuerung

Für Kameraaufnahmen mit einem Multicopter wird häufig die Fähigkeit zum Schwebflug ausgenutzt. Hierbei muss die Schubkraft der Propeller gleich der Gewichtskraft sein, sodass eine konstante Höhe gehalten wird. Desweiteren muss die Flugplattform möglichst parallel zum Boden (bzw. der XZ-Ebene) ausgerichtet sein, so dass die an den Propellern anliegenden Kraftvektoren parallel zur Y-Achse nach oben zeigen und keine Komponente zur Seite wirkt. Auf diese Weise wirkt keine horizontale Kraft und der Multicopter behält die aktuelle Position bei. Oft ist es jedoch schwer solch eine Orientierung manuell zu erhalten, da ein Pilot aus der Bodenperspektive die Ausrichtung des Multicopters nur schlecht abschätzen kann, ebenfalls kann durch Wind die Lage des Flugkörpers unerwartet verändert werden, was die Neuorientierung erschwert. Desweiteren reagiert die Steuerung häufig sehr empfindlich und muss präzise und feinfühlig vorgenommen werden. Aus den genannten Gründen verfügen Multicopter daher oftmals über eine Lageregelung, mit welcher sich eine automatische Ausrichtung des Multicopters umsetzen lässt. Lim et al. schlagen als Sensorik der inertiale Lageregelung eine Kombination aus Gyroskopen und Beschleunigungssensoren vor [LPLK12, S. 38]. Mithilfe von Gyroskopen lassen sich Winkelgeschwindigkeiten um die Rotationsachsen messen, durch Integration dieser Geschwindigkeiten kann die aktuelle Orientierung des Multicopters im Raum approximiert werden. Die Beschleunigungssensoren können hierbei ergänzend wirken, in dem sie die Richtung der aus der Erdanziehungskraft verursachten Beschleunigung ermitteln können. Anhand der Sensordaten können also unter anderem die Rotationswinkel um Roll- und Nickachse festgestellt werden. Diese Winkel

lassen sich als Eingabedaten an eine in der Steuerelektronik enthaltene Regelungstechnik (Controller) übergeben [LPLK12, S. 39], diese kann durch direkte Ansteurung der Propeller den Multicopter automatisch entsprechend nicken und rollen lassen. Dies lässt sich beispielsweise nutzen, um den Multicopter automatisch parallel zum Boden auszurichten (Fluglageregelung), sobald keine Steuerung mehr durch die Fernbedienung erfolgt. Häufig ist neben der Fluglageregelung eine zusätzliche Anforderung, dass der Multicopter selbstständig eine feste Höhe bzw. eine feste Position im Raum halten kann. Hierfür muss weitere Regelungstechnik verwendet werden an welche als Eingabedaten beispielsweise eine mittels GPS berechnete Position im Raum übergeben wird. Grzonka et al. zeigen hierzu alternative Herangehensweisen auf, die es ermöglichen sollen Positionsbestimmung und Navigation eines Multicopters in Gegenden ohne GPS-Empfang durchführen zu können. Es wird beispielsweise ein SLAM-Algorithmus (Simultaneous Localization and Mapping) zur Positionsbestimmung als mögliches Konzept aufzeigt [GGB12]. Das Konzept der Lageregelung kann weiter ausgedehnt werden, so dass der Multicopter komplett autonom fliegt, D'Andrea und Hehn zeigen hierzu in [DH11] weiterführende Techniken, um den Multicopter eine vorgegebene Flugbahn abfliegen zu lassen.
Ein klassischer Ansatz zur Umsetzung einer Lageregelung ist die Verwendung von PID-Reglern, Lim et al. zeigen hierzu in [LPLK12, S. 40] eine Übersicht von in verschiedenen Projekten verwendete Kontrollschemata, welche PID-Reglern verwenden. Ghadiok et al. zeigen ebenfalls Ansätze PID-Regler zur Ausrichtung von Höhe und Position zu verwenden [GGR12]. Desweiteren wird in [CS12] untersucht, inwieweit die in der Steuerelektronik enthaltenen PID-Regler weiter optimiert werden können. Bouabdallah et al. haben in [BNS04] den klassischen Ansatz einen PID-Regler zu verwenden einem neueren Regelverfahren namens LQ-Controller (Linear quadratic controller) gegenübergestellt, es werden jedoch nur mittelmäßige Ergebnisse erzielt, da das von ihnen gewählte dynamische Multicopter-Modell Schwachstellen hat. In der Arbeit [BS07] werden die Untersuchungen mit einem verbesserten dynamischen Modell fortgeführt, sie verwenden zur Lageregelung von Fluglage, Höhe und Position eine Kombination von PID-Reglern und einem Regelungsverfahren namens *Integral Backstepping*, mit welcher eine Verbesserung der Lageregelung erzielt wurde. Als häufiger Problemfall in der Regelungstechnik ist zu nennen, dass die Sensordaten häufig Signalrauschen enthalten, was zusätzliche Filterprozesse z.B. mit Kalman-Filtern nötig macht. Ghadiok et al. evaluieren in[GGR12] verschiedene Filterprozesse und zeigen Ansätze zur Verbesserung auf.

2.1.5 Stabilität des Multicopters

Einen Nachteil bei Multicoptern stellt die hohe Empfindlichkeit gegenüber Windböen dar [ROR10, S. 30]. Erwähnenswert ist hier ebenfalls der sogenannte *Bodeneffekt*, dieser tritt bei der Landung von Hubschraubern [Bit09, S. 67], allerdings auch bei Multicoptern auf: Die Rotation der Propeller sorgt je nach Stärke der Schubkraft der Propeller für einen zum Boden gerichteten Luftstrom. Dieser Luftstrom wird in Bodennähe vom Boden reflektiert und wirkt dem Luftstrom des Fluggeräts entgegen, was ein Aufsetzen auf den Boden und allgemein die Kontrolle in Bodennähe erschwert.
Die Stabilität des Multicopers sei im Folgenden definiert als die Unempfindlichkeit der Flugplattform gegenüber atmosphärische Störungen wie Windböen bzw. allgemein die Fähigkeit diese Störungen auszugleichen.
Generell ist die Dämpfung der auf den Flugkörper wirkenden Winde beliebter Forschungsgegenstand bei Multicoptern, ein häufiger Ansatz ist es Anpassungen an der Steuerelektronik bzw. der darin enthaltenen Steuersoftware und der Regelungstechnik vorzunehmen. Alexis et al. beschreiben diesen Ansatz, welche ebenfalls repräsentativ für ähnliche Arbeiten ist wie folgt:

"Moreover, during flights in low-altitudes, quadrotors are prone to sudden wind gusts that

can significantly affect their flight performance and even cause instability. Therefore the development of specialized controllers, that could take under consideration the quadrotor's modeling nonlinearities and the uncertainties while reacting to such sudden wind gusts is desired." [ANT11, S. 1195]

YANG et al. untersuchen in der Arbeit [YGP09] Ansätze die Steuerelektronik eines unbemannten Helikopters mithilfe eines PD-Regler so zu optimieren, dass die Wirkung von atmosphärische Störungen, darunter Windböen, speziell beim Landeanflug möglichst gedämpft wird. Sie verwenden hierfür ein Windmodell, wonach Wind als zufälliger Prozess mithilfe von weißem Rauschen modelliert ist und ein Modell zur Vorhersage der zukünftigen Störungsintensität. In [YGP11] wird dieser Ansatz fortgeführt, die Effekte der atmosphärischen Störungen nicht nur beim Landeanflug, sondern auch beim allgemeinen Schwebeflug abzuschwächen. Einen ähnlichen Ansatz verfolgen ALEXIS et al., es wurde mit Erfolg eine Stabilisierung der Fluglage bei horizontal wirkenden Windböen mithilfe modellprädiktiver Regelung erreicht [ANT11]. Auch bei dieser Methode werden Vorhersagen über den zukünftigen Verlauf der Windböe getroffen und die Regelungstechnik entsprechend angesteuert.

2.2 Perlin-Noise und Simplex-Noise

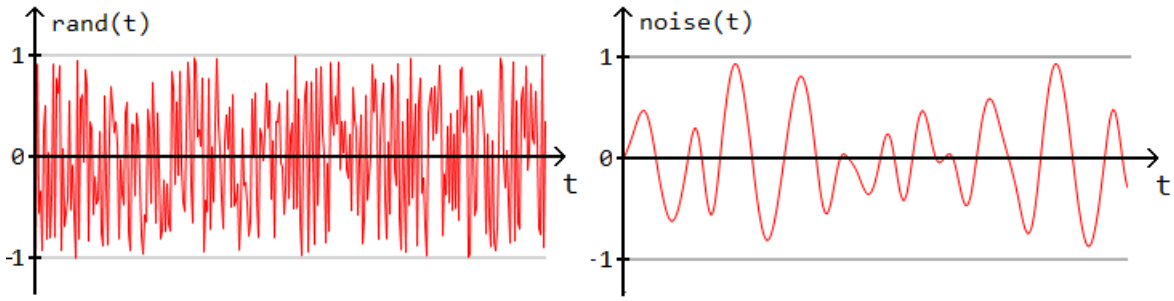

Abbildung 9: Zeit-Wert-Graph einer Abfolge von pseudozufällig generierten Werten (links) und einer eindimensionalen Simplex-Noise-Funktion (rechts)

Bei *Perlin-Noise* handelt es sich um eine von KEN PERLIN entwickelte mathematische Funktion die scheinbar zufälliges Rauschen erzeugt. Der Unterschied gegenüber einer normalen Pseudozufallsfunktion ist, dass die zufälligen Werte bei der Rauschfunktion kontinuierlich und stetig aufeinander folgen, was ein weicheres, natürlicheres Funktionsbild zur Folge hat (s. Abbildung 9).
Die eindimensionale Perlin-Noise-Funktion lässt sich nach [Bur08, S. 1] definieren als:

$$noise(x) : \mathbb{R} \to \mathbb{R}$$

Es handelt sich um eine gradientenbasierte Rauschfunktion [Bur08, S. 2]. Bei diesen Funktionen sind an diskreten Abzissenpunkten $p \in \mathbb{Z}$ pseudozufällige Steigungen (die Gradienten) durch eine Gradientfunktion $grad(p)$ gegeben:

$$grad(p) : \mathbb{Z} \to [-1, 1]$$

Die Werte der Rauschfunktion werden mithilfe einer Interpolationsfunktion F stückweise in $[0, 1]$-Intervallen zwischen den diskreten Gradientpunkten interpoliert. Das Ergebnis der Interpolation von $noise(x)$ hängt dabei ausschließlich von den Gradienten der zu x benachbarten ganzen Zahlen ab:

$$noise(x) = F(x, grad(\lfloor x \rfloor), grad(\lfloor x \rfloor + 1))$$

Ohne Beschränkung der Allgemeinheit kann daher die Interpolation auf ein $[0,1]$-Interval begrenzt werden [Bur08, S. 3]. Sei $\dot{x} = x - \lfloor x \rfloor$, $g_0 = grad(\lfloor x \rfloor)$ und $g_1 = grad(\lfloor x \rfloor + 1)$, dann ist $\dot{x} \in [0,1]$ und die Interpolationsfunktion kann angepasst werden zu:

$$F(\dot{x}, g_0, g_1) = F(x, grad(\lfloor x \rfloor), grad(\lfloor x \rfloor + 1))$$

Geeignete Interpolationsfunktion sind Polynomfunktion mit mindestens Grad 3 [ebenda]. Da die Gradientenwerte die Steigungen für die diskreten Werte $p \in \mathbb{Z}$ angeben, können an diesen Punkten Tangenten konstruiert werden, lineare Geraden mit der Steigung $grad(p)$ welche die X-Achse an Stelle p schneiden [Bur08, S. 5]. Die Gerade der Tangente wird durch die Funktion

$$h_p(x) = grad(p) * (x - p)$$

gegeben. Die Interpolationsfunktion lässt sich als Spline-Interpolation der dazugehörigen Tangenten auffassen [Bur08, S. 6], wobei die Gradienten g_0 und g_1 die Steigungen der Tangenten an der jeweiligen Intervallgrenzen angeben. Nach der zuletzt genannten Formel ergibt sich für die Tangenten an den Intervallgrenzen:

$$\begin{aligned} h_0(x) &= g_0 \cdot x \\ h_1(x) &= g_1 \cdot (x - 1) \end{aligned}$$

Burger nennt als Interpolationsfunktion [ebenda]:

$$F(\dot{x}, g_0, g_1) = h_0(\dot{x}) + s(\dot{x}) \cdot (h_1(\dot{x}) - h_0(\dot{x}))$$

Die Funktion $s(x)$ gibt hierbei eine polynomielle Überblendungsfunktion an. In der aktuellsten Referenzimplementierung[6] von Perlin-Noise verwendet Perlin als Überblendungsfunktion [Per02, S. 682] (Formelzeichen wurden angepasst):

$$s(x) = 6x^5 + 15x^4 + 10x^3$$

Insgesamt ergibt sich als also als Perlin-Noise-Funktion:

$$noise(x) = F(\dot{x}, g_0, g_1) = h_0(\dot{x}) + s(\dot{x}) \cdot (h_1(\dot{x}) - h_0(\dot{x}))$$

Durch ständiges Aufrufen der Funktion von $noise(x)$ und Erhöhung von x wird ein Funktionsbild analog Abbildung 9 rechts erzeugt. Die Rauschfunktion verfügt über eine anpassbare Frequenz [Bur08, S. 11f]: Wird der Parameter der Rauschfunktion mit einem Wert f skaliert, so dass $noise(f \cdot x)$ aufgerufen wird, so führt ein Wert von $f > 1$ zu einer Stauchung des Funktionsbild. Analog führt ein Wert von $f < 1$ zu einem gedehnten Funktionsverlauf. Anhand der Frequenz lässt sich also die Geschwindigkeit mit welcher die Funktionswerte fluktuieren anpasssen.
Am Rande sei hier erwähnt, dass die Perlin-Noise-Funktion mehrdimensional definiert ist. Mit der zweidimensionalen Variante $noise(x,y) : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ können prozedurale Texturen für dreidimensionale Objekte (beispielsweise zur Darstellung von Wolken) gestaltet, die dreidimensionale Funktion $noise(x,y,z) : \mathbb{R}^3 \to \mathbb{R}$ lässt sich beispielsweise zur Generierung von 3D-Landschaften einsetzen. Die mehrdimensionalen Varianten werden im Folgenden allerdings außer Acht gelassen, da sie in dieser Arbeit nicht verwendet werden.
Die Funktion Simplex-Noise ist eine Optimierung der Perlin-Noise-Funktion, welche vor allem in höheren Dimensionen durch Verwendung mehrer Simplizes weniger Rechenzeit benötigt [Bur08, S. 40]. Sei n die Anzahl Dimensionen der Rauschfunktionen. Perlin nennt als Komplexitätsklasse von Perlin-Noise $\mathcal{O}(2^n)$, wohingegen als Komplexitätsklasse von Simplex-Noise $\mathcal{O}(n^2)$angegeben ist[7]. Die Unterschiede des Funktionsverlauf gegenüber der Perlin-Noise-Funktion sind marginal. Das eindimensionale Simplex-Noise beinhaltet aus rein algorithmischer Sicht keine Veränderung gegenüber dem eindimensionalen Perlin-Noise.

[6] Diese Referenzimplementierung ist zu finden unter: `http://mrl.nyu.edu/~perlin/noise/`
[7] Website von Ken Perlin: `http://www.noisemachine.com/talk1/32.html`

2.3 PID-Regler

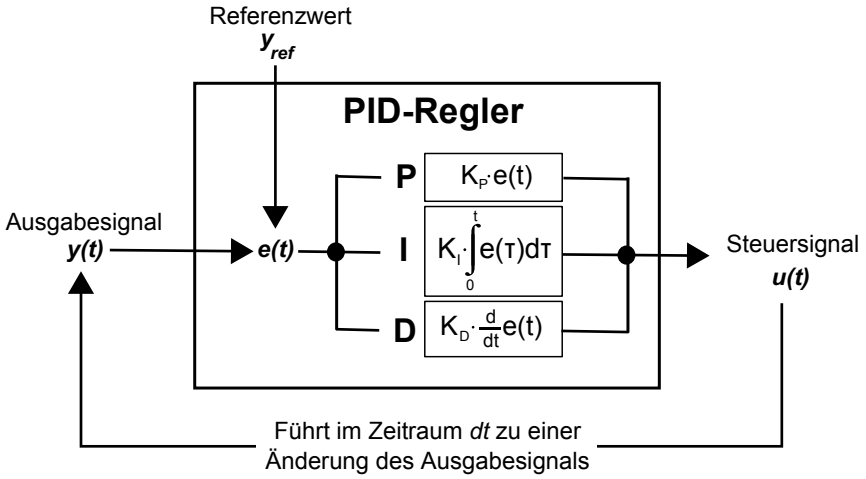

Abbildung 10: Grafische Darstellung des Kontrollkreislaufes bei einem PID-Regler

Der PID-Regler (im englischen *PID controller* genannt) ist ein Mechanismus aus der Regelungstechnik mit welchem physikalische Werte auf Basis gegebener Werte und Signale angepasst werden können. Der PID-Regler basiert auf der kontinuierlichen Verarbeitung bzw. Festlegung der folgenden drei Werte [Vag12, S. 3]:

Steuersignal Der Wert des Steuersignals $u(t)$ ist der Wert, welcher in einem bestimmten Zeitintervall vom PID-Regler eingestellt wird und eine Änderung des Messwertes bewirkt. Das Signal entspricht hierbei bildlich gesprochen der „Stellschraube" mit welcher eine Änderung des Ausgabesignals erreicht werden kann.

Ausgabesignal Der Wert des Ausgabesignals $y(t)$ ist der zu einem bestimmten Zeitpunkt gemessene Wert eines zu regulierenden Signals.

Referenzwert Der Referenzwert y_{ref} ist ein Idealwert, auf welchen der Wert des Ausgabesignals durch Anpassung des Steuersignals gebracht werden soll.

Die Anpassung des Steuersignals wird hierbei kontinuierlich immer wieder aufs neue vorgenommen, wahlweise auch in einem festen Zeitintervall. Die Anpassung funkioniert nach einem rückgekoppelten Prinzip: Der Ausgabewert beeinflusst direkt den Eingabewert (s. Abbildung 10).

Die Abweichung des Ausgabesignalwerts vom Referenzwert wird als Fehler bezeichnet. Aus der Differenz zwischen Referenzwert und Ausgabesignalwert ergibt sich die *Fehlerfunktion* $e(t)$ mit:

$$e(t) = y_{ref} - y(t)$$

Die Abkürzung PID steht für die drei Anpassungsschritte, die zur Steuerung des Steuersignalwertes bei diesem Regelverfahren konsekutiv unternommen werden:

1. Bei der *Proportionalen Komponente* (P) wird die Regulierung des Steuersignals ausschließlich mittels der Fehlerfunktion *e(t)* und einer Konstanten K_P, welche die *proportionale Verstärkung* angibt [Sel01, S. 2] vorgenommen:

$$u_P(t) = K_P \cdot e(t)$$

2. Bei der *Integralen Komponente* (I) wird das Steuersignal mithilfe eines Integrals über den gesamten Zeitverlauf angepasst, was durch deren Akkumulierung eine Berücksichtigung der vergangenen Fehler ermöglicht. Für diese Komponente wird eine Konstante K_I verwendet, welche die *integrale Verstärkung* angibt [Sel01, S. 5]. Die Integrationsvariable ist durch das Formelzeichen τ angegeben.

$$u_I(t) = K_I \cdot \int_0^t e(\tau) d\tau$$

3. Bei der *Derivativen Komponente* (D) wird das Steuersignal mithilfe einer Ableitung über die seit der letzten Anpassung vergangenen Zeit angepasst, worüber eine Vorhersage des zukünftigen Fehlers einberechnet und dem entgegengesteuert werden soll. Für diese Komponente wird eine Konstante K_D verwendet, welche die *derivative Verstärkung* angibt [Sel01, S. 6].

$$u_D(t) = K_D \cdot \frac{d}{dt} e(t)$$

Insgesamt ergibt sich als Formel für die Regelung des Steuersignals die Summe aller drei Komponenten:

$$u(t) = K_P \cdot e(t) + K_I \cdot \int_0^t e(\tau) d\tau + K_D \cdot \frac{d}{dt} e(t)$$

Wird ausschließlich die proportionale Komponente verwendet, führt dies zwar zu einer groben Anpassung des Ausgabesignals an den Referenzwert, jedoch tritt hier häufig der unerwünschte Nebeneffekt auf, dass das Signal ständig um den Referenzwert oszilliert (überschwingt). Dies wird durch die integrale Komponente kompensiert, da diese durch ihre Akkumulierung eine Dämpfung bewirkt [Sel01, S. 4]. Durch die integrale Komponente kann es jedoch zu einem „Wind-up" kommen: Die Akkumulierung der Integralfunktion wird zumeist über eine Aufsummierung der Fehler erreicht. Wenn der Wert der Fehlerfunktion jedoch nie sein Vorzeichen ändert, kann dies zu einer sehr großen Summe bei der integralen Komponente führen. Dies hat zur Folge, dass das Steuersignal falsch angepasst wird, dieses erreicht eventuell irgendwann einen Maximal- bzw. Minimalwert und das Ausgabesignal nähert sich nie dem Referenzwert erreicht - die Regelung versagt [Sel01, S. 5]. Dieser Wind-up-Effekt kann durch eine gute Wahl der integralen Verstärkung verhindert werden.
Mit der derivativen Komponente wird die Änderung des Fehlers gegenüber der letzten Anpassung berücksichtigt. Dies führt bei richtiger Anwendung dazu, dass der Regler schneller auf Änderungen des Ausgabesignals bzw. des Referenzwertes reagiert. Weiterhin kann durch diese Komponente ein anfängliches Überschwingen ausgeglichen werden [Sel01, S. 6]. Die derivative Komponente bringt allerdings den Nachteil, dass sie den Regler empfindlich macht gegenüber Rauschen beim Ausgabesignal [CS12, S. 2]: Durch Rauschen im Signal kann eine große Änderung der Intensität in einem kurzen Zeitraum stattfinden. Die Ableitung über die Zeit kann so eine unerwünschte, abrupte Änderung des Steuersignals verursachen.
Mit den Konstanten der proportionalen, integralen und derivativen Verstärkung kann reguliert werden, wie viel Einfluss die jeweiligen Komponenten auf die Regelung haben. Für die Konstanten müssen heuristisch Werte für eine gute Regelung gefunden werden, es wird hier auch vom *Tunen* des PID-Reglers gesprochen. Ein schlechtes Tuning kann dazu führen, dass die Änderung des Steuersignals zu schnell stattfindet und es starke Oszillationen beim Ausgabesignal gibt. Ungünstig gewählte Konstanten, könn auch zu einem gegenteiligen Effekt führen, indem sehr viel Zeit benötigt wird, bis das Ausgabesignal sich dem Referenzwert nähert.

Wie in Unterabschnitt 2.1.4 aufgeführt lassen sich bei Multicoptern unter anderem PID-Regler zur Regulierung von Fluglage, Höhe und Position einsetzen.

2.4 Microsoft Robotics Developer Studio

Bei dem *Microsoft Robotics Developer Studio* (im Folgenden abgekürzt mit MRDS) handelt es sich um ein Softwareframework zum Entwickeln und Testen von Steuersoftware in der Robotik.
Ein wichtiger Aspekt bei Robotertechnologie ist, dass oft viele Prozesse gleichzeitig ablaufen müssen. Ein denkbarer Anwendungsfall wäre beispielsweise, dass ein mit einem Fahrwerk ausgestatteter Roboter einer Taschenlampe folgen soll. Es müssen hierzu gleichzeitig Sensordaten eines Lichtsensors ausgewertet und dementsprechend gleichzeitig das Fahrwerk angesteuert werden. Das Beispiel soll zeigen, dass Robotersoftware oft nichtsequentiell ausgeführt werden muss. Die threadbasierte Entwicklung von nichtsequentieller Software birgt allerdings viele Probleme, u.a. stellt die Koordinierung von gleichzeitigem Zugriff auf gemeinsame Ressourcen (Mutex-Problem) und die Gefahr von Deadlocks häufig eine große Herausforderung dar. Mithilfe der im Folgenden vorgestellten Technologien CCR (s. Abschnitt 2.4.1) und DSS (s. Abschnitt 2.4.2) können die verteilten Prozesse in einem Roboter abgebildet und koordiniert werden. Das Ziel dieser Bibliotheken ist, die Arbeit mit nichtsequentieller Software erheblich zu vereinfachen.
Desweiteren wird die Simulationsumgebung VSE zur Verfügung gestellt, in welcher sich Roboter virtuell nachbauen und testen lassen. Aufgrund dessen Umfang wird diese gesondert in Abschnitt 2.5 behandelt.
MRDS ist stark plattformabhängig und nur unter dem Betriebssystem Microsoft Windows lauffähig. Für diese Abschlussarbeit wurde die zu diesem Zeitpunkt aktuelleste Version Microsoft Robotics Developer Studio 4 aus dem Jahr 2012 verwendet.

2.4.1 Concurrency and Coordination Runtime

Bei dem *Concurrency and Coordination Runtime* (kurz CCR) handelt es sich um eine Softwarebibliothek zur Entwicklung von nichtsequentiellen Programmen. CCR soll hierbei dem Programmierer durch Bereitstellung von zusätzlichen Klassen und Operationen eine Hilfestellung bieten. Ziel ist es, die Koordination stark zu vereinfachen und durch ein klares Programmmodell die Lesbarkeit des Codes zu erhöhen [JT08, S. 30].
Die Hauptidee ist, dass das Programm in isolierte Komponenten unterteilt wird. Der Datenaustausch zwischen den Komponenten soll kontrolliert über Nachrichten stattfinden (*Message passing*). Es wird eine Warteschlange in Form der Klasse *Port* zur Verfügung gestellt, über welche die Nachrichten asynchron an andere Komponenten gesendet werden können [JT08, S. 50]. Die Nachrichten können primitive Datentypen, aber auch Objekte beliebiger serialisierbarer Klassen sein. Die gesendeten Nachrichten können so lange vorgehalten werden, bis sie von einem Empfänger abgeholt werden. Ebenfalls bereitgestellt wird eine Klasse *Dispatcher*, durch welche das Scheduling in Form einer Thread-Pool-Implementierung zur Verfügung gestellt wird [JT08, S. 65]. Über den Dispatcher können mithilfe einer Klasse *Arbiter* Threads zum Empfangen der Nachrichten (*Handler*) gestartet werden, um diese unter kontrollierten Bedingungen abzuarbeiten. Die Arbiter-Klasse koordiniert hierbei den Thread-Pool, wartet auf freie Threads sowie Ressourcen und weist automatisch Aufgaben an die Threads zu. Auf programmiertechnische Konzepte wie beispielsweise Semaphore oder Sperren bei kritischen Abschnitten soll so verzichtet werden können, diese werden im Rahmen der CCR-Bibliothek automatisch angewandt. Dadurch soll das Risiko von Deadlocks und weiteren Problemen minimiert werden [JT08, S. 30].
Weiterhin zur Verfügung gestellt wird ein Konzept zur vereinfachten Fehlerbehandlung bei Benutzung mehrerer Threads. Es wird ermöglicht, Fehler isoliert von den anderen Kompo-

nenten abzufangen und gesondert zu behandeln, ohne dass das gesamte Programm instabil wird. Das Auftreten von Fehlern in einer Komponente kann an andere Komponenten gemeldet werden, so dass bei Abhängigkeiten entsprechend reagiert werden kann [JT08, S. 77].

2.4.2 Decentralized Software Services

Mit den *Decentralized Software Services* (kurz DSS) wird ein Framework zum Entwickeln verteilter Anwendungen, auch außerhalb der Roboterwelt bereitgestellt. In CCR besteht ein Programm aus einem Prozess mit mehreren Threads, Prozesse dieser Art werden in MRDS *Services* genannt [JT08, S. 6]. DSS bildet eine programmiertechnische Ebene oberhalb von CCR, die Idee wird fortgeführt: Ein Programm wird bei DSS in mehrere Services unterteilt. Die Services können auf verschiedenen über ein Netzwerk verbundenen Rechnern gestartet werden und untereinander kommunizieren. Services werden in Form einer Klasse implementiert, die nach einem von MRDS vorgegebenen Schema aufgebaut ist u.a. muss die Service-Klasse von der Klasse *DsspServiceBase* erben. Jeder Service verfügt über einen sogenannten *State*, mit welchem der Zustand eines Services abgebildet werden soll [JT08, S. 87]. Als Zustand werden die konkreten Belegungen von Variablen zu einem bestimmten Zeitpunkt aufgefasst, die Variablen sind gebündelt in einer State-Klasse innerhalb des Services gespeichert. Services können nur auf *(DSS-)Nodes* ausgeführt werden: Ein *Node* ist die Instanz einer speziellen DSS-Serveranwendung, welche auf ein oder mehreren Computern gestartet werden kann. Die Nodes koordinieren die einzelnen Services untereinander und ermöglichen zwischen ihnen die Kommunikation via HTTP und SOAP [JT08, S. 92]. Werden Nachrichten über das Netzwerk gesendet, findet vorher eine Serialisierung in XML statt, auf der Empfängerseite werden die Nachrichten nach der Übertragung wieder deserialisiert. Wenn zwei Services miteinander kommunizieren, werden sie *Partner* genannt [JT08, S. 93]. Durch diese Verfahrensweise soll eine hohe Flexibilität erreicht werden, da Programme wahlweise auf einem, aber auch auf beliebig vielen anderen Computern verteilt ausgeführt werden können. In einer *Manifest-Datei* können im XML-Format Voreinstellungen an den Node übergeben werden, unter anderem auch die zu startenden Partner-Services und andere Nodes, mit welchen eine Verbindung aufgebaut werden soll [ebenda].
Das folgende Beispiel soll die Idee in einem Anwendungsfall erläutern: Es sei ein Roboter mit einem Rechner als Steuereinheit sowie eine Bodenstation, die mit dem Roboter verbunden ist gegeben. Der Roboter soll Kameradaten sammeln, welche dann ausgewertet werden sollen. Die Auswertung soll auf der Bodenstation erfolgen, weil diese über eine höhere Rechenleistung verfügt. Die Implementierung nach dem DSS-Prinzip wäre auf dem Roboter einen Node laufen zu lassen, welcher einen Service zum Sammeln der Kameradaten ausführt. Dieser Service schickt die Daten an die Bodenstation, auf welcher auch ein Node mit einem Service ausgeführt wird. Der Service der Bodenstation empfängt die Daten vom Roboter und kann diese weiter auswerten.
Die Hauptidee ist insgesamt, dass Services isoliert voneinander ausgeführt werden können. Dies bietet den Vorteil, dass bei einem Fehler nur der betreffende Service beendet wird und die anderen unbeeinflusst weiterlaufen können. Durch Anwendung von DSS lässt sich eine lose Kopplung des Programms ermöglichen [JT08, S. 29]: Jeder Service erfüllt eine ganz bestimmte ausschließlich ihm zugewiesene Aufgabe und sendet seine Ergebnisse zur Weiterverarbeitung an andere Services.
DSS bietet eine Anbindung zu Windows Forms, einer Windows-Programmierschnittstelle zum Entwickeln grafischer Benutzeroberflächen. Auf diese Weise lassen sich in DSS-Programme Formulare bzw. Fenster einfügen, welche zur Anzeige oder Eingabe von Daten genutzt werden können [JT08, S. 351].
Für die Entwicklung von DSS-Services können ausschließlich .NET-Programmiersprachen wie Visual C#, Visual C++ und Visual Basic verwendet werden.

2.5 Microsoft Visual Simulation Environment

Bei der Entwicklung von Steuersoftware in der Robotik haben Programmierfehler oft ernste Auswirkung auf den Roboter, bis hin zu dessen Zerstörung. Es stellt daher, insbesondere für Fluggeräte einen großen Vorteil dar, die entwickelte Software zunächst in einer virtuellen Umgebung zu testen, in welcher kein Schaden angerichtet werden kann [Dis09, S. 2]. Weiterhin bieten Simulationen die Möglichkeit auf einfache Weise Prototypen zu entwickeln, zu testen und Anschauungen über das verwendete Modell anzustellen. Bouabdallah et al. nutzen beispielsweise eine Simulation für das Tuning der Regelungstechnik [BNS04, S. 5], in [BS07] werden ebenfalls der Abhebevorgang, das Halten der Höhe, sowie der Landevorgang zunächst in einer Simulation getestet.
Mit der *Microsoft Visual Simulation Environment* (kurz VSE) wird als Teil des Microsoft Robotics Developer Studio-Pakets zu den genannten Zwecken eine frei programmierbare Simulationsumgebung geboten. Die VSE ist ein DSS-Service, in welchem eine Rendering-Engine basierend auf dem *XNA Framework* (eine von Microsoft entwickelte Bibliothek zur Spieleentwicklung, es wird ein vereinfachter Zugriff auf Grafik, Sound und Eingabe geboten*)* und Zugriffmöglichkeit auf die Physik-Engine *NVIDIA PhysX* zur Verfügung gestellt wird [JT08, S. 229], durch welche sich physikalische Effekte annähernd korrekt simulieren lassen. Als Integrationsverfahren der Physik-Engine wird ein *Symplectic Integrator* verwendet, welcher eine vergleichsweise hohe Genauigkeit (d.h. kleiner relativer Approximationsfehler) bei hoher Performanz ermöglicht [BB07, S. 285]. Eine grundlegende Anforderung an Simulationsumgebungen ist ein dreidimensionales Achsensystem mit 6 Freiheitsgerade, dieses ist in der VSE analog dem Koordinatensystem in Abbildung 3 umgesetzt. Es sind weiterhin Datentypen für Vektoren und Quaternionen (Quadrupel, welche such gut für Orientierungsangaben eines Objektes im Raum erleichtern) vorgegeben. Einen großen Vorteil stellt dar, dass bei vielen Werteangaben SI-Einheiten verwendet werden können, beispielsweise kann bei Positions- und Dimensionsangaben die Einheit Meter verwendet werden, bei Gewichtsangaben Kilogramm. Ähnlich wie in einem Computerspiel können die zu simulierenden Roboter in einer 3D-Welt nachgebaut werden. In dem MRDS-Paket ist eine Auswahl fertiger 3D-Welten gegeben, welche durch eine Manifest-Datei in die Simulationsumgebung geladen werden können. Die VSE umfasst zusätzlich einen Editor zum Entwerfen eigener Welten.
Um eine den eigenen Anforderungen entsprechende Simulationsumgebung zu schaffen, muss ein neuer Service programmiert werden, welcher einen von der VSE bereitgestellten Service namens *SimulationEngine* als Partner-Service aufruft. Es ist es möglich verschiedene Kameras in Form von *CameraEntities* einzufügen, mit welchen sich die Szenen aus verschiedenen Blickwinkeln anschauen lassen. Standardmäßig vorgegeben ist eine mit Tastatur und Maus frei bewegbare Kamera (*MainCamera* genannt).

2.5.1 Visuelle Entitäten

Dem VSE-Service lassen sich über ein Port visuelle Entitäten wie beispielsweise der zu simulierende Roboter, aber auch andere Elemente, mit welchem der Roboter interagieren soll hinzufügen. Visuelle Entitäten sind im allgemeinen Entitäten, die in der 3D-Umgebung durch ein Modell und Texturen dargestellt werden, weiterhin können auf diese Form von Entitäten physikalische Effekte angewandt werden. Hierzu muss eine eigene Entitäten-Klasse entwickelt werden, welche von der Klasse *VisualEntity* erbt [JT08, S. 247].
Jede visuelle Entität verfügt über eine Pose, diese wird beschrieben durch eine Kombination aus Position und Orientierung im Raum. In der VSE ist dies durch eine gleichnnamige Klasse *Pose* umgesetzt: Die Position wird in Form eines dreidimensionalen Vektors gespeichert, die Orientierung wahlweise durch ein Eulerwinkel (die Komponenten geben die Rotation um die jeweilige Achsen an) oder durch ein Quaternion.
Desweiteren kann für visuelle Entitäten ein physikalisches Modell beschrieben werden.

Durch Zusammenfügung verschiedener Grundkörper wie Quader, Kugeln, Zylinder oder Kegel kann die physikalische Form der Entität, sowie die Gewichte der einzelnen Elemente vorgegeben werden [JT08, S. 287]. Das physikalische Modell kann bei der Physik-Engine angemeldet werden, die visuelle Entität unterliegt dann automatisch der Schwerkraft und kollidiert mit anderen Objekten der Simulationsumgebung. Auf visuelle Entitäten kann die Methode *ApplyForce* aufgerufen werden, welche eine lineare Kraftwirkung auf die Entität simuliert. Betrag und Wirkrichtung der Kraft können in Form eines Vektors angegeben werden. Analog funktioniert die Methode *ApplyTorque*, mit welcher sich die Wirkung eines Drehmoments auf die Entität simulieren lässt. Zusätzlich kann ein Wirkpunkt angegeben werden und eine Angabe, ob die Koordinaten aus Sicht des Objekts bzw. aus Sicht der 3D-Welt angegeben sind.
Desweiteren ist zu nennen, das visuelle Entitäten über Kindentitäten verfügen können, diese sind selber wieder visuelle Entitäten, welche der Hauptentität untergeordnet sind. Kindentitäten sind physikalisch mit der Hauptentität verbunden [JT08, S. 248]: Sie werden mit ihr mitbewegt, als wären sie an die Hauptentität angebaut. Beispielsweise lassen sich bei einem fahrenden Roboter die Räder als Kindentitäten einfügen. Mit Kindentitäten kann der Code semantisch aufgeteilt und so ggf. übersichtlicher gestaltet werden.

Jede visuelle Entität besteht aus folgenden Methoden, welche durch Überschreibung nach individuellen Bedürfnissen angepasst werden können [JT08, S. 272f]:

Initialize Diese Methode wird aufgerufen, nachdem die Entität in die Simulationsumgebung eingefügt wurde. Hier kann das visuelle und physikalische Modell initialisiert werden, bei Bedarf können auch im späteren Programmablauf benötigte Werte der Entität initialisiert werden.

Update In dieser Methode können Aktualisierungen der Programmlogik vorgenommen werden wie beispielsweise die Überprüfung und Neuberechnung von Werten. Ebenfalls können hier physikalische Effekte wie das Wirken eines Krafts bzw. eines Drehmoments gewirkt werden.

Render Diese Methode wird bei der Darstellung der Entität als 3D-Modell aufgerufen, hier kann bei Bedarf die Optik angepasst werden.

Dispose Diese Methode wird beim Löschen der visuellen Entität aus der Simulationsumgebung aufgerufen. Es können nicht mehr benötigte Ressourcen freigegeben werden.

2.5.2 Programmablauf einer implementierten Simulation

Die Programmierung einer Simulation mithilfe von VSE ähnelt der Entwicklung eines Computerspiels: Zu Beginn des Programms wird für alle hinzugefügten Entitäten die *Initialize*-Methode aufgerufen. Die Simulationsumgebung unterteilt die Zeit in *Frames,* wobei ein Frame einem Einzelbild der 3D-Simulation besteht. In einer Schleife werden die Frames ständig neu berechnet und anschließend angezeigt [JT08, S. 242]. Die folgende Auflistung führt die sich wiederholenden Schritte der Frameberechnung auf:

1. Zu Beginn jedes Frames werden die im letzten Frame berechneten Berechnungsergebnisse der Physikengine für den aktuellen Frame zur Verfügung gestellt.

2. Bei jeder Entität, die in die Simulationsumgebung hinzugefügt wurde wird deren *Update*-Methode aufgerufen. Hierdurch wird die Programmlogik der einzelnen Entitäten aktualisiert, beispielsweise werden Kräfte gewirkt usw.

3. Anschließend wird bei jeder Entität deren *Render*-Methode aufgerufen.

4. Das Gesamtbild aus allen sichtbaren Entitäten wird aus Perspektive der aktuell ausgewählten Kamera-Entität gerendert.

5. Das als Ergebnis entstande Bild wird als ein Frame im Fenster der Simulationsumgebung angezeigt.

6. Vorbereitung des nächsten Frames, beginne wieder bei Punkt 1

Die dynamischen Komponenten, welche eine ständige Aktualisierung und Neuberechnung benötigen, müssen also in der Update-Methode ausgeführt werden. Wird das Programm geschlossen, werden bei allen Entitäten die jeweiligen *Dispose*-Methoden aufgerufen und das Programm beendet.

3 Projektübersicht

3.1 Verbesserung der Stabilität

Die Stabilität des Multicopters sei im weiteren Verlauf der Arbeit unterteilt in folgende Komponenten, da für diese unterschiedliche Regelungstechniken notwendig sind:

1. Stabilität der Fluglage: Beschreibt den Grad des Nickens, Gierens und Rollens ohne dass dies vom Pilot vorgesehen ist. Je geringer dieser Effekt ist, d.h. je weniger der Multicopter in der Luft schwankt, desto größer ist die Stabilität der Fluglage. Diese Form der Stabilität ist besonders wichtig für Foto- und Filmaufnahmen mit einer an den Multicopter angebrachten Kamera, ist die Fluglage instabil können die Aufnahmen stark verwackelt sein.

2. Stabilität der Höhe: Beschreibt die Fähigkeit des Multicopters eine bestimmte Höhe im Raum zu halten und unempfindlich gegenüber ungewollten Änderungen der Höhe zu sein.

3. Stabilität der Position: Bezieht sich auf die Position in der XZ-Ebene. Beschreibt zusammen mit der Stabilität der Höhe die Fähigkeit eine bestimmte Position im Raum zu halten und unempfindlich gegenüber unwillkürlichen Positionsänderungen zu sein.

Die Grundidee soll es sein, eine höhere Stabilität bei den Komponenten zu erreichen. Es wird hierbei anhand einer Computersimulation untersucht, ob die Anzahl der Propeller und der Neigungswinkel der Propeller Auswirkungen auf die eben genannten Stabilitäten haben. Die Vermutung ist, dass die nach innen geneigten Propeller in irgendeiner Form die vom Wind verursachten Schwingungen abfangen bzw. dass eine höhere Propelleranzahl dem Wind genügend Kraft entgegensetzt und so die Unempfindlichkeit gegenüber Windböen verstärkt. Weiterhin hat es eventuell unterschiedliche Auswirkungen insbesondere auf die Fluglagestabilität, ob eine gerade oder ungerade Anzahl an Propellern verwendet wird.

3.2 Anforderungen an den Simulator

An die Computersimulation werden folgende Anforderungen gestellt:

1. Physik-Simulation: Um die in der Natur auftretenden Phänomene wie Kraftwirkung durch Gewicht, Propeller und Wind zu simulieren muss die Computersimulation ebenfalls die Wirkung der Physik berücksichtigen. Da MRDS mit einer Physik-Engine ausgestattet ist, kann diese Anforderung umgesetzt werden. Das in Unterabschnitt 2.5 genannte Integrationsverfahren der Physik-Engine sollte eine ausreichende Genauigkeit aufweisen.

2. Simulation des Multicopter-Flugverhaltens: Das Flugverhalten des Multicopters sollte möglichst realistisch nachgebildet werden. Der Multicopter sollte der Schwerkraft unterliegen, ebenso sollten die Propellerkräfte umgesetzt werden, so dass Steigflug

und Sinkflug möglich sind. Weiterhin sollte sich der Multicopter durch manuelle Steuerung nicken, gieren und rollen lassen, die Flugmanöver sollen eine Bewegung in alle Richtungen ermöglichen.

3. Möglichkeiten zur Parameteranpassung: Es sollte möglich sein Parameter des simulierten Multicopters, wie die Anzahl der Propeller und den Neigungswinkel ändern zu können.

4. Umsetzung einer Windsimulation: Die Stabilität des Multicopters wird vom Wind gemindert. Um einen Vergleich aufstellen zu können müssen die Auswirkungen vom Wind auf den Multicopter, wie die Wirkung von Kraft und Drehmomenten auf die Flugplattform nachgebildet werden.

5. Stabilisierung des Multicopters durch Regelungstechnik: Für einen Vergleich ist ebenfalls nötig, dass der Multicopter automatisch eine bestimmte Stelle im Raum anfliegen und halten kann.

6. Möglichkeiten zur Datenauswertung: Um die Stabilität messen zu können muss eine Möglichkeit gegeben werden Daten anzuzeigen und ggf. aufzeichnen zu lassen.

3.3 Aufbau des Multicopter Simulators

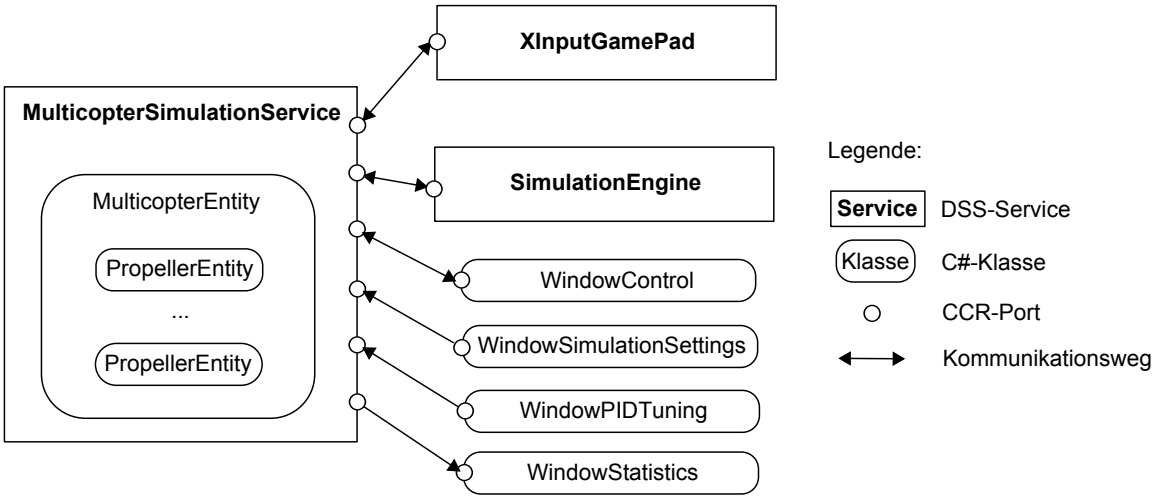

Abbildung 11: Software-Architektur des Multicopter-Simulators

Der Multicopter-Simulator besteht mehreren DSS-Services und Klassen, zur Umsetzung wurde die Programmiersprache *Visual C#* gewählt, da diese für die Programmierung von DSS am geeignetsten erschien. In Abbildung 11 ist ein Diagramm der Softwarearchitektur dargestellt, welche im folgenden näher erläutert werden soll.

Den Kern der Computersimulation bildet der neu geschriebene DSS-Service *MulticopterSimulationService*. Es handelt sich hierbei um einen Orchestrierungs-Service, welcher die anderen Services aufruft und als Schnittstelle zwischen allen Komponenten fungiert. Dieser ruft als Partner-Service den in Abschnitt 2.5 genannten *SimulationEngine*-Service aus. Zusätzlich wird ein Service *OutdoorSimulation* geladen, welcher über die *SimulationEngine* in die Simulationsumgebung ein bereits mit MRDS mitgeliefertes Outdoor-Level (siehe Abbildung 12) lädt. Dieses bietet genügend Platz und eine realistische Umgebung für Multicopter-Flüge. Als weiterer Partner-Service wird ebenfalls *XInputGamePad* gestartet, welcher gegebenenfalls den Zugriff auf ein angeschlossenes *Xbox 360* Gamepad ermöglicht, mit welchem der Simulator manuell gesteuert werden kann. Der *SimulationEngine* wird ei-

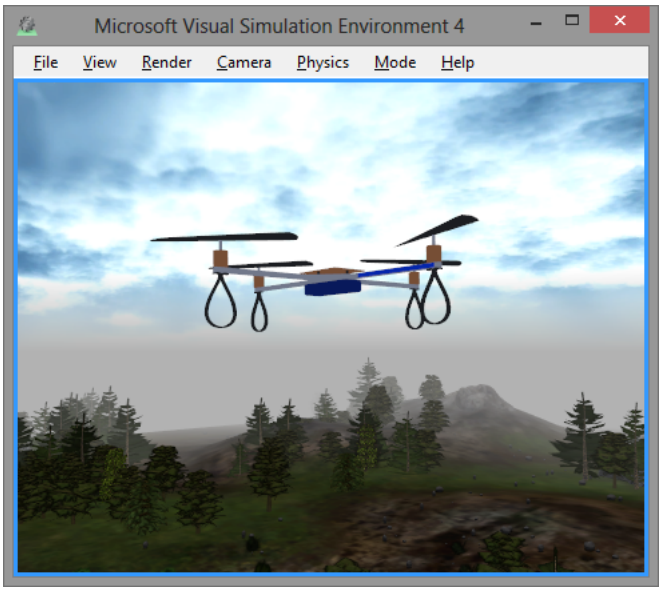

Abbildung 12: Darstellung der Simulationsumgebung in der VSE inklusive visueller Entität des Multicopters

ne neue visuelle Entität hinzugefügt: Die *MulticopterEntity*. In ihr sind die zu simulierenden Eigenschaften des Multicopter gekapselt umgesetzt, ebenso die automatische Stabilisierung mithilfe von PID-Reglern und die Simulation des Windes auf den Multicopter. Zur Simulation werden weiterhin verschiedene Kamera-Entitäten hinzugefügt: Eine frei bewegbare *MainCamera*, eine *PursuitCameraEntity* welche eine Verfolgerkamera implementiert, sowie eine *AttachedCameraEntity* welche eine an den Multicopter angebrachte Kamera simulieren soll. Zwischen allen Kameras kann jederzeit umgeschaltet werden. Desweiteren werden vom Service verschiedene Windows-Fenster geöffnet, welche die Steuerung des Simulators ermöglichen und eine Möglichkeit zum Visualisieren der Werte bieten:

Fenster „Simulatorsteuerung" Im Fenster zur Simulatorsteuerung (siehe Abbildung 13 links), welches in der Klasse *WindowControl* implementiert wurde kann der Multicopter wahlweise per Tastatur und Maus gesteuert werden . Es ist ebenfalls möglich die automatische Stabilisierung der Fluglage („Keep attitude"), der Höhe („Keep altitude") und der Position („Keep position") zu aktivieren bzw. zu deaktivieren. Ebenfalls anpassbar ist die zu haltende Höhe in Metern, sowie die zu haltenden X- und Z- Koordinaten der Position. Weiterhin werden Information über Höhe und Geschwindigkeit des simulierten Multicopters angezeigt.

Fenster „Simulatoreinstellungen" In diesem Fenster (siehe Abbildung 14 rechts), welches in der Klasse *WindowSimulationSettings* implementiert wurde lassen sich die Einstellungen der Multicopter-Entität bzw. der Simulationsumgebung anpassen. Es ist möglich sowohl die Anzahl als auch den Neigungswinkel der Propeller einzustellen, durch Klick auf den Button „Apply" lassen sich diese Einstellungen aktualisieren[8]. Ebenfalls lassen sich Einstellungen an der Windsimulation vornehmen, beispielsweise können Intensität, Richtung und Grad der Fluktuierung dieser Werte angegeben werden.

Fenster „PID-Tuning" Dieses Fenster (siehe Abbildung 14 links) wurde in der Klasse *WindowPIDTuning* umgesetzt. Über Schieberegler können die Werte der Konstanten K_P,

[8] Aus nicht rekonstruierbaren Gründen führt ein Klick auf diesen Button gelegentlich zum Absturz des Programmes. Es war bis zum Zeitpunkt der Abgabe dieser Arbeit nicht möglich die Ursache dieses Fehlers zu finden und zu beheben.

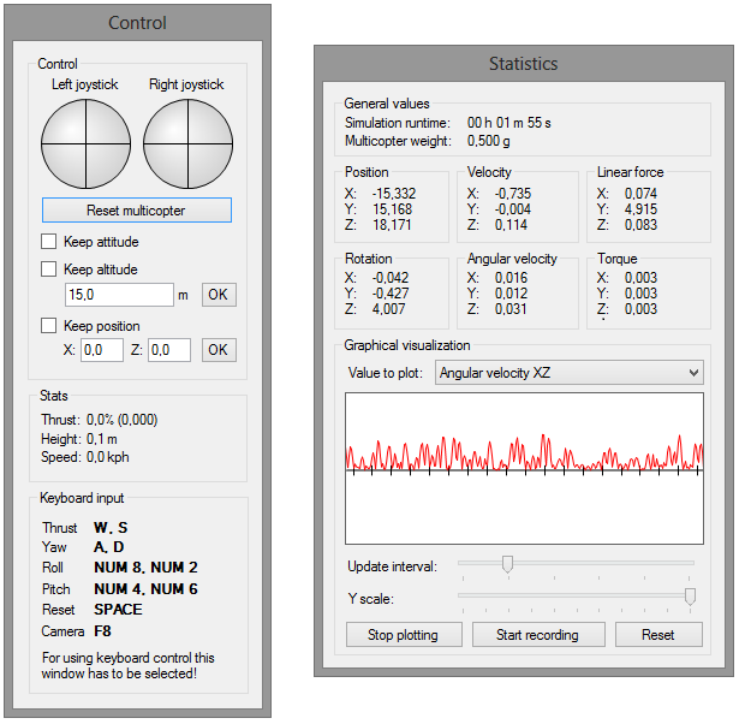

Abbildung 13: Das Fenster „Simulatorsteuerung" (links), ermöglicht die Steuerung des Multicopters. Über das Fenster „Statistik" (rechts) können zur Laufzeit verschiedene Daten ausgelesen werden. Ebenfalls steht eine grafische Darstellung vom zeitlichen Verlauf einiger Werte sowie eine Möglichkeit zum Aufzeichnen zur Verfügung.

Abbildung 14: Das Fenster „PID-Tuning" (links) ermöglicht das Tuning der drei verwendeten PID-Regler. Im Fenster „Simulatoreinstellungen" (rechts) können Parameter des Multicopters sowie die Windsimulation angepasst werden.

K_I und K_D für die jeweiligen verwendeten PID-Regler angepasst werden.

Fenster „Statistik" Mit diesem Fenster, welches in der Klasse *WindowStatistics* implementiert ist, lässt sich eine Auswahl Daten, wie beispielsweise die aktuellen Positionsdaten, Geschwindigkeit und Rotation des Multicopters anzeigen, was vor allem für Debug-Zewcke benötigt wurde (siehe Abbildung 13 rechts). Ebenfalls angezeigt wird die Länge der Laufzeit der aktuellen Simulation („Simulation Runtime"), sowie das Gewicht des Multicopters. Außerdem wurde mit diesem Fenster die Anforderung 6 aus dem vorigen Abschnitt umgesetzt: Von einer Auswahl Werte kann der Verlauf aufgezeichnet und grafisch dargestellt werden.

Der Datenaustausch zwischen Programm und den Fenstern findet über mehrere Ports statt. Für jedes Fenster wird ein eigenes Port definiert, über welche die Kommunikation in beide Richtung stattfinden kann. Beispielsweise werden über ein Port die aktualisierten Werte vom *MulticopterSimulationService* an das Fenster *Statistik* gesandt, anders herum empfängt der Simulator-Service über ein Port die Steuerungsdaten, welche vom Fenster *Simulator-Steuerung* gesandt wurden. Die Datenflussrichtungen sind in Abbildung 11 dargestellt. Mittels *States* (s. Abschnitt 2.4.2) können beim Verlassen des Programms die in den Fenstern vorgenommenen Einstellungen in einer XML-Datei gespeichert werden und beim Start der Simulation wieder geladen werden, was die Arbeit mit dem Simulator ein wenig vereinfachen soll.

Die folgenden Abschnitte sollen näher ins Detail der einzeln umgesetzten Simulator-Module gehen. In Abschnitt 4 wird das im Simulator verwendete Modell des Multicopters genauer erläutert, ebenso die Implementierung als visuelle Entität. Um Experimente durchzuführen ist es nötig, auf den Multicopter wirkende Windeffekte zu simulieren, wie dies in die Praxis umgesetzt wurde ist in Abschnitt 5 wiedergegeben. Die Umsetzung der automatischen Stabilisierung mithilfe von PID-Reglern wird in Abschnitt 6 genauer dargelegt.

4 Simulation des Multicopter-Flugverhaltens

4.1 Objekt und Weltkoordinaten

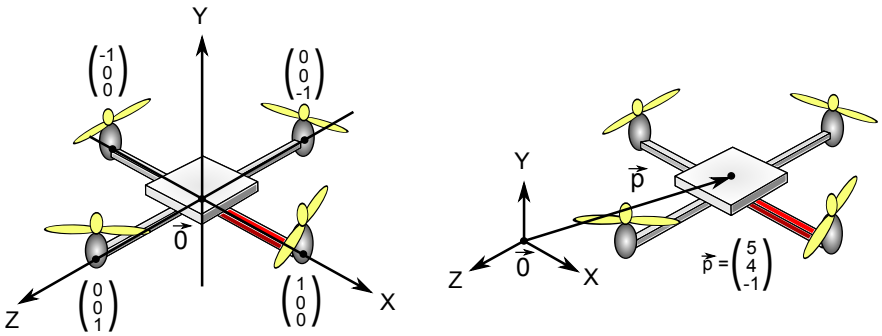

Abbildung 15: Betrachtung des Multicopters in Objektkoordinaten (links) und Weltkoordinaten (rechts)

Im Folgenden wird unterschieden zwischen den Objektkoordinaten des Multicopters und Weltkoordinaten (s. Abbildung 15). Die Weltkoordinaten (auch globale Koordinaten genannt) bilden ein universales, festes Koordinatensystem, welches alle Objekte der Simulation enthält, unter anderem auch den Multicopter. Die Positionen und Rotationen aller Objekte der gesamten Szene sind in absoluter Form aus Sicht der gesamtheitlichen 3D-Welt angegeben. Die Objektkoordinaten (auch lokale Koordinaten genannt) dagegen sind

an den Multicopter gebunden, sie beziehen sich auf die aktuelle Position und Orientierung des Multicopters. Der Ursprung des Koordinatensystems ist immer am Mittelpunkt des Multicopters positioniert, bei Bewegung des Multicopters wird das Koordinatensystem entsprechend mitbewegt. Wird der Multicopter global um die Achsen rotiert, rotiert das Koordinatensystem mit, so dass die Positionsangaben ständig relativ zum Multicopter bleiben. Bei dem in Objektkoordinaten angegebenen Multicopter bildet die Y-Achse die Gierachse, die X-Achse die Rollachse und die Z-Achse die Nickachse. Findet eine Rotation um die Y-Achse statt, werden so beispielsweise X- und Z-Achse gleichermaßen mit dem Multicopter um die Y-Achse mitrotiert.

4.2 Modell des Multicopters

Multicopter und dessen Propeller sind im Folgenden als starre, symmetrische Körper modelliert. Ein Multicopter verfügt über einen bestimmten Multicopter-Typ, welcher durch die Anzahl Propeller beschrieben wird (Tricopter, Quadrocopter, ...). Es werden die in Unterabschnitt 2.1.2 aufgeführten Bezeichnungen verwendet. Als weitere Eigenschaft ist der Neigungswinkel der Propeller analog der Beschreibung in Unterabschnitt 2.1.2 gegeben.

Für den Multicopter können folgende Werte gesetzt werden:

- n repräsentiert im Folgenden die *Anzahl der Propeller*. Für ein Quadrocopter gilt beispielsweise $n = 4$.
- ψ repräsentiert den *Neigungswinkel der Propeller*. Sind die Propeller nicht geneigt, gilt $\psi = 0°$.

Da von einem symmetrischen Modell ausgegangen wird, muss der Winkel zwischen allen benachbarten Auslegern bzw. Propellern gleich sein. Jeder Ausleger hat einen bestimmten Rotationswinkel um die Gierachse des Multicopters.

- Jeder Propeller sei eindeutig durch P_x mit $x \in \{1, ..., n\}$ identifiziert. Propeller P_1 bezeichne den Propeller, welcher auf dem Ausleger, der die Vorwärtsrichtung angibt angebaut ist.
- Der *Winkel des Auslegers*, welcher Propeller P_x enthält sei durch $\phi(P_x)$ bezeichnet. Für alle Propeller gilt:
$$\phi(P_x) = (x - 1) \cdot \frac{360°}{n}$$

Die Modellierung berücksichtigt ebenfalls das Gewicht der Konstruktion:

- Das *Gewicht des zentralen Elements*, welches Steuerelektronik und Akkus enthält wird durch den Wert m_Z repräsentiert.
- Das *Gewicht eines Auslegers* samt Stab, Motor und Rotor ist durch den Wert m_A gegeben. Es wird davon ausgegangen, dass jeder Ausleger gleich schwer ist.
- Das *Gesamtgewicht* des Multicopters m ergibt sich aus der Summe der Gewichtskomponenten und der Anzahl der Propeller: $m = m_Z + n \cdot m_A$
- Der *Schwerpunkt* des Multicopters befindet sich stets an der lokalen Koordinate $(0, 0, 0)$ und damit also genau am Mittelpunkt des Multicopters.

In den folgenden Unterabschnitten wird die Modellierung des Multicopter-Zustands und ebenfalls die Flugmechanik, sowie das Wirken von Kräften und Drehmomenten näher dargelegt.

4.2.1 Zustand des Multicopters

Der Zustand des Multicopters ist durch eine Sammlung von Werten und Vektoren beschrieben, welche abhängig von der Simulationszeit sind.

- Die *Position* des Multicopters im dreidimensionalen Raum in Weltkoordinaten wird durch folgenden Ortsvektor repräsentiert:

$$\vec{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}$$

- Die *lineare Geschwindigkeit* des Multicopters in Weltkoordinaten wird durch folgenden Vektor repräsentiert:

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

- Der *Rotationszustand* des Multicopters wird in Form eines Eulerschen Winkels $\vec{r}$ beschrieben, wobei als Referenzsystem das Weltkoordinatensystem verwendet wird. r_x, r_y und r_z bezeichnen die Winkel um die jeweiligen Achsen:

$$\vec{r} = \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix}$$

- Die *Winkelgeschwindigkeit* wird ebenfalls in Form eines Eulerschen Winkels, mit demselben Referenzsystem definiert, wobei ω_x, ω_y und ω_z die Winkelgeschwindigkeiten um die jeweiligen Achsen bezeichnen:

$$\vec{\omega} = \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix}$$

4.2.2 Modellierung der Flugdynamik

Auf den Multicopter wirkt durch die Gravitation ständig eine globale Kraft, die Gewichtskraft. Die Wirkung ist unabhängig von Position und Ausrichtung des Multicopters, daher ist die Kraft durch einen in Weltkoordinaten angegebenen Vektor

$$\vec{F_g} = \begin{pmatrix} 0 \\ -m \cdot g \\ 0 \end{pmatrix}$$

gegeben, diese Kraft wirkt in negative Y-Richtung auf den im vorigen Abschnitt angegebenen Schwerpunkt des Multicopters. g sei hier die Konstante der Gravitationsbeschleunigung auf der Erde ($g = 9,81 \frac{m}{s^2}$).
Ebenfalls berücksichtigt wird der Luftwiderstand, welcher sich in Form einer Dämpfung von Geschwindigkeit und Winkelgeschwindigkeit über die Zeit auswirkt.

Der Multicopter verfügt über verschiedene Eingabewerte, welche die Steuerung durch eine Fernbedienung ermöglichen:

- Der *Schub* f_t ist proportional zur Drehgeschwindigkeit der Propeller. Ein höherer Schub bedeutet eine höher wirkende Kraft an den Propellern. Der Schub ist begrenzt durch die maximale Drehgeschwindigkeit des Motors. Dies wird durch eine Konstante f_{tmax}modelliert. Es muss gelten: $0 \leq f_t \leq f_{tmax}$

- Der *Gierfaktor* f_y bestimmt den Betrag und die Richtung des Drehmoments um die Gierachse. Der Betrag des Gierfaktor ist begrenzt durch eine Konstante f_{ymax}, so dass gilt: $-f_{ymax} \leq f_y \leq f_{ymax}$
- Der *Nickfaktor* f_p bestimmt den Betrag und die Richtung des Drehmoments um die Nickachse. Der Betrag des Gierfaktor ist begrenzt durch eine Konstante f_{pmax}, so dass gilt: $-f_{pmax} \leq f_p \leq f_{pmax}$
- Der *Rollfaktor* f_r bestimmt den Betrag und die Richtung des Drehmoments um die Rollachse. Der Betrag des Gierfaktor ist begrenzt durch eine Konstante f_{rmax}, so dass gilt: $-f_{rmax} \leq f_r \leq f_{rmax}$

Auf jeden Propeller des Multicopters wirkt eine Propellerkraft. Diese wirkt direkt am Propeller und ist orthogonal zur Flugplattform ausgerichtet (vgl. Abbildung 4). Da die Kraft somit abhängig ist von der Ausrichtung des Flugkörpers, wird sie durch einen in Objektkoordinaten angegebenen Vektor $\vec{F}_{P_x}$ modelliert. Sie ist ebenfalls direkt proportional zum Schubwert. Weiterhin ist zu beachten, dass die Kraftrichtung abhängig ist vom Neigungswinkel der Propeller, daher muss zunächst der Kraftwirkungsvektor durch Rotation um die Z-Achse dem Neigungswinkel ψ angepasst werden. Durch anschließende Rotation um die Y-Achse wird die Richtung des Kraftvektors an die Position des für die Kraft zuständigen Propellers rotiert. Sei $\vec{F}_{P_x}$ sei die auf den Propeller P_x wirkende Kraft, $R_y(\alpha)$ und $R_z(\alpha)$ beschreiben eine Rotation des Vektors um den Winkel α und die Y- bzw. Z-Achse, dann gilt:

$$\vec{F_{P_x}} = R_y(\phi_{P_x}) \cdot (R_z(\psi) \cdot \begin{pmatrix} 0 \\ f_t \\ 0 \end{pmatrix})$$

In diesem Modell wird vernachlässigt, dass die Achsenrotationen (Gieren, Nicken, Rollen) durch verschiedene Propellerdrehgeschwindigkeiten bzw. divergente, an den Propellern wirkenden Schubkräfte verursacht werden. Stattdessen wird vereinfachend angenommen, dass die Achsenrotationen durch ein Drehmoment M_{rot} zustande kommen. Da, wie in Abschnitt 2.1 beschrieben Gier-, Nick- und Rollachse an die Orientierung des Multicopters gebunden sind wird dieses Drehmoment in Objektkoordinaten angegeben:

$$M_{rot} = \begin{pmatrix} f_r \\ f_y \\ f_p \end{pmatrix}$$

Da die Rotationen um die Achsen global betrachtet eine gleichsame Rotation der Propellerkraft-Vektoren verursachen, kann auf diese Weise ein Flug in alle Richtung stattfinden.

4.3 Der Multicopter als visuelle Entität

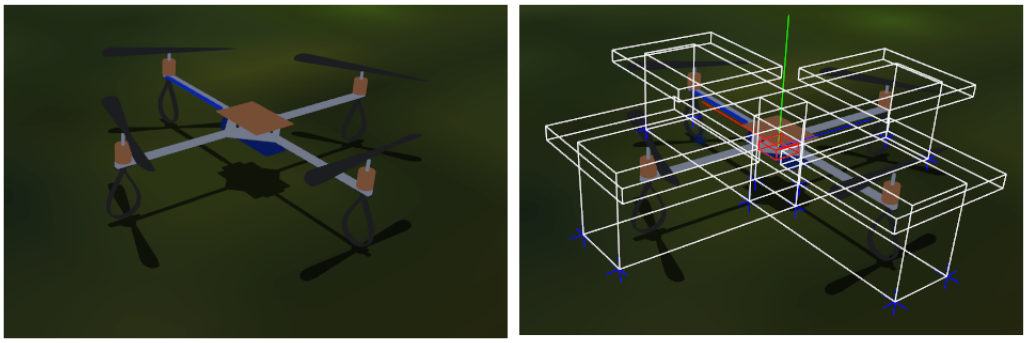

Abbildung 16: Die visuelle Entität des Multicopters (links) und dessen physikalisches Modell (rechts)

Einen Nachteil von MRDS stellt dar, dass das Framework keine Hilfsmittel zur Erstellung von Flugrobotern vorgibt. Dementsprechend müssen die Flugaspekte alle manuell „from scratch" umgesetzt werden, was allerdings gleichzeitig eine höhere Flexibilität erlaubt. Das zuvor dargelegte Modell des Multicopters wird in Form einer visuellen Entität, repräsentiert durch die Klasse *MulticopterEntity* gekapselt implementiert. *MulticopterEntity* muss daher von der in Abschnitt 2.5.1 beschriebenen Klasse *VisualEntity* erben und wird über ein Port der *SimulationEngine* (also der VSE) hinzugefügt. Beim Einfügen einer *MulticopterEntity* müssen drei Startwerte festgelegt werden: Der Multicopter-Typ (welcher die Anzahl der Propeller enthält) der Neigungswinkel der Propeller in Grad, sowie der Startpunkt des Multicopters in Weltkoordinaten. Für die Simulation der Propeller wird die Fähigkeit von Entitäten genutzt, Kindentitäten zu enthalten: Es wurde eine neue visuellen Entität namens *PropellerEntity* implementiert, welche einen Propeller repräsentiert. Instanzen dieser Entität werden je nach Anzahl der Propeller dynamisch als Kindentität zur *MulticopterEntity* hinzugefügt und sind an diese gebunden.

Die hauptsächliche Umsetzung des Modells findet in den Methoden *Initialize* und *Update* der Multicopter-Entität statt, welche nachfolgend beschrieben sind.

Initialize-Methode

In der *Initialize*-Methode (und ebenfalls im Konstruktor) der *MulticopterEntity* werden zunächst die nötigen Initialisierungen und die anfängliche Belegung der Werte vorgenommen. Zunächst wird das durch die Propelleranzahl festgelegte, texturierte 3D-Modell (*Mesh*) mit der entsprechenden Anzahl Ausleger geladen und über die Eigenschaft *MeshScale* so skaliert, dass die Maßstäbe des Modells mit denjeningen der Computersimulation übereinstimmen. In Abbildung 16 links ist als Beispiel das 3D-Modell des Quadrocopter-Typs dargestellt.

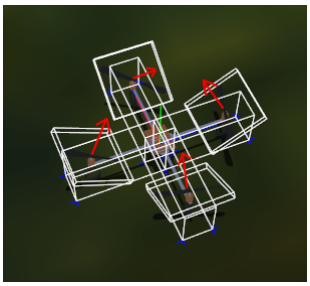

Abbildung 17: Die roten Pfeile markieren die Wirkrichtung der Propellerkräfte in Abhängigkeit vom Neigungswinkel der Propeller (hier 20°)

Anschließend wird die Initialisierung der Physik-Engine vorgenommen. Mithilfe einer Variable *SolverIterationCount* wird zunächst die Anzahl der Iterationsschritte, welche in jedem Frame bei der Integration durch den Symplectic Integrator der Physik-Engine vorgenommen werden festgelegt. Ein höherer Wert bedeutet hier eine größere Genauigkeit aber auch einen höheren Berechnungsaufwand. Ein Kompromiss aus beiden Aspekten wurde mit dem Wert 60 gefunden. Anschließend wird das physikalische Modell der Multicopter-Entität festgelegt: Die Flugplattform ist modelliert als ein Objekt mit 30 cm Breite, 30 cm Länge und einer Höhe von 14 cm. Der Grundaufbau wird dabei in grundlegende geometrische Formen unterteilt: Hierzu werden um das Zentrum des Multicopters, um die Ausleger und um die Propeller Kollisionsboxen gelegt, welche die jeweiligen Formen approximieren. Das Ergebnis ist in Abbildung 16 in Form eines Drahtgittermodells visualisiert. In derselben Abbildung ist ebenfalls der Schwerpunkt durch einen roten Quader dargestellt, dieser wird genau im Mittelpunkt der Entität positioniert. Bei den einzelnen Kollisionsboxen kann nun eine Masse angegeben werden: Das Gewicht der Box, welche das zentrale Element umfasst wurde ein Wert von $m_Z = 200g$ gesetzt, den Boxen für die Ausleger wurde jeweils ein Wert von $m_A = 125g$ als Gewicht zugewiesen. Die Gesamtmasse wird später zusammengerechnet im Statistik-Fenster angezeigt. Es können weiterhin für jede Box Materialeigen-

schaften wie die Elastizität des Materials, sowie statische Reibung (bezeichnet die Reibung zwischen unbewegten Flächen) und dynamische Reibung (bezeichnet die Reibung zwischen bewegten Flächen) festgelegt werden. Hier wurde die Einstellung einer geringen Elastizität getroffen, so das der Multicopter bei einer unsanften Landung vom Boden reflektiert wird, weiterhin wurden Einstellung für hohe Reibungskräfte vorgenommen. Zuletzt wird das definierte physikalische Modell mithilfe der Methode *CreateAndInsertPhysicsEntity* bei der Physik-Engine angemeldet. Die *MulticopterEntity* verfügt nach diesem Schritt über eine Kollisionserkennung, kann also mit allen Objekten der 3D-Umgebung kollidieren. Für die Umsetzung der Gravitation müssen keine speziellen Anpassungen unternommen werden, durch die Anbindung an die Physik-Engine unterliegen alle dem Simulator hinzugefügten *VisualEntities* automatisch einer simulierten Erd-Schwerkraft, so auch die *MulticopterEntity*. Weiterhin bietet die Physik-Engine die Möglichkeit einen Luftwiderstand zu simulieren: Mit der Zuweisung eines Wertes zu *LinearDamping* lässt sich für die Entität eine Reibung bei linearen Impulsen simulieren, analog legt ein positiver Wert für *AngularDamping* die Reibung gegenüber Drehimpulsen fest. Hier wurden entsprechende Werte für Luft eingetragen.
Anschließend werden gemäß der angegebenen Propelleranzahl die Propeller-Entitäten hinzugefügt. Ist ein Neigungswinkel angegeben, werden die Propeller-Entitäten zunächst entsprechend des Neigungswinkels nach innen und dann zur Position des Auslegers rotiert. Jede *PropellerEntity* verfügt über zwei Vektoren *position* und *forceDirection,* welche in Objektkoordinaten bezüglich zum Multicopter angegeben sind. Dem Vektor *position* wird dabei die Position des Propellers auf dem Multicopters zugewiesen, der Vektor *forceDirection* ist normalisiert und soll die Wirkrichtung der Kraft des Propellers gemäß der Neigung des Propellers angeben. Durch jeweilige Rotation des Vektors *position* um die lokale Y-Achse werden alle Propeller-Entitäten auf der Flugplattform an die entsprechenden Stellen rotiert. Analog werden wie beim Modell jeweils die *forceDirection*-Vektoren rotiert, so dass die Kraft aller Propeller bei einem positiven Neigungswinkel teilweise nach innen wirkt (s. Abb. 17).

Update-Methode In der *Update*-Methode der *MulticopterEntity* finden die Berechnungen für den nächsten Frame statt. Die Variablen *raw_thrust, raw_yaw, raw_pitch* und *yaw_roll* beinhalten jeweils die Werte für Schub, Gierfaktor, Nickfaktor und Rollfaktor, welche im Frame angewandt werden sollen. Durch mehrere Kontrollstrukturen werden die Werte innerhalb der im Modell beschriebenen Intervallgrenzen f_{tmax}, f_{ymax}, f_{pmax} und f_{rmax} gehalten.
Mittels der Prozedur *ApplyForceAtPosition* kann die Physik-Engine angewiesen werden unter Angabe einer Wirkposition und eines Kraftvektors eine Kraft auf den Multicopter wirken zu lassen. In zwei zusätzlichen Boolean-Parametern *positionIsLocal* und *forceIsLocal* kann angegeben werden, ob die Vektorangaben sich auf Objektkoordinaten (*true*) oder auf Weltkoordinaten (*false*) bezogen . Diese Prozedur wird genutzt, um eine Kraft auf jeden Propeller wirken zu lassen: Als Wirkposition wird jeweils der *position*-Vektor des Propellers angegeben. Der Kraftvektor wird gebildet, indem die Wirkrichtung des Propellers, *forceDirection* (normalisierter Vektor), mit *raw_thrust* skaliert wird. Da beide Angaben sich auf Ortsvektoren beziehen, wird dies ebenfalls an die Prozedur übergeben. Auf diese Weise wurde die Propellerkraftwirkung dem Modell entsprechend umgesetzt.
Die Prozedur *ApplyTorque* weist die Physik-Engine an, unter Angabe eines Vektors ein Drehmoment auf den Ursprung wirken zu lassen. Auch hier kann über einen Parameter *torqueIsLocal* angegeben werden, ob sich das Drehmoment auf Objektkoordinaten oder Weltkoordinaten bezieht. Dies kann genutzt werden um die Achsenrotationen des Multicopters umzusetzen. Als zu wirkendes Drehmoment wird hierbei der folgende Vektor in

Objektkoordinaten übergeben:

$$torqueVector = \begin{pmatrix} raw\_roll \\ raw\_yaw \\ raw\_pitch \end{pmatrix}$$

Durch die Angabe in Objektkoordinaten bezieht sich die Rotation auf den Mittelpunkt des Multicopters. Die Flugmanöver Nicken, Gieren und Rollen wurden so ebenfalls gemäß dem Modell umgesetzt.
Weitere Implementierungen müssen im Zuge der Flugdynamik nicht vorgenommen werden. Durch die Angaben in Objektkoordinaten beziehen sich die Kraftvektoren immer auf die Ausrichtung des Multicopters, die Physik-Engine rechnet automatisch um. Auf diese Weise wird im Rahmen der Simulation eine Bewegung in alle Freiheitsgerade erlaubt.
In der Update-Methode wird ebenfalls die Kraftwirkung des Windes und die Regulierung der Werte durch die automatische Stabilisierung mithilfe einer Regelungstechnik realisiert, was in den folgenden Abschnitten näher erläutert wird. Zuletzt werden in der Methode über einen Port die aktuellen Wertebelegungen der Variablen und Vektoren an das Statistik-Fenster gesandt.

Nicht an der Umsetzung des Modells beteiligt, aber ebenfalls erwähnenswert sind die Methoden *Render*, *Dispose* und *Reset*: In der *Render*-Methode wird die Rotation der Propeller durch Drehung des 3D-Objektes visuell umgesetzt. Die *Dispose*-Methode wird beim Löschen der Entität aufgerufen und löscht alle Kindentitäten, in diesem Fall also die mit dem Multicopter verknüpften Propeller-Entitäten. Mithilfe der Methode *Reset* können die anfängliche Position und Orientierung des Multicopters wiederhergestellt werden, so dass dieser bei einem Absturz zurückgesetzt werden kann.

5 Windsimulation

5.1 Modell des Windes

Die Untersuchung der Stabilität machen es erforderlich, dass in der Simulation die Wikrung von Wind auf den Multicopter berücksichtigt wird. Der Wind ist in der Simulation als eine ständig global wirkende Kraft modelliert, dessen Betrag und Intensität sich unvorhersehbar ändern. Eine Komponente des Windes, die im Modell vorgesehen ist, ist ein konstant auftretender Wind, dessen Intensität sich nur langsam ändert, der aber von schwacher Intensität ist. Die andere Komponente des Windes besteht aus zufällig auftretenden Windböen, sind von starker Intensität, allerdings nur von vergleichsweise kurzer Dauer. Der Wind wirkt je nach Intensität ebenfalls Drehmomente auf den Multicopter, was zu einem Schwanken der Flugplattform führt, so dass der Multicopter ungewollt ein wenig nickt, giert und rollt. Der in Abschnitt 2.1 genannte Bodeneffekt wird in der Simulation vernachlässigt, da er für die Betrachtung der Stabilität bei den späteren Experimenten nicht relevant ist.

Im Modell sind weiterhin vier Parameter vorgesehen, welche die Windrichtung beeinflussen:

1. Windrichtung: Der Wind wirkt zum Großteil horizontal, also parallel zum Boden. Die Richtung aus welcher der Wind kommt kann in Form eines Winkels angegeben werden: 0° bedeuten in Weltkoordinaten betrachtet einen Wind in positive X-Richtung, 90° in positive Z-Richtung, 180° in negative X-Richtung und 270° in negative Z-Richtung.

2. Grundintensität des Windes: Die Windintensität fluktuiert ständig, über diesen Parameter kann angegeben werden, in welchen Intervallen die Windintensität schwankt. Hierdurch kann also die allgemeine Stärke der Windkraft angegeben werden.

3. Fluktuationsgeschwindigkeit der Windrichtung: Gibt an, wie schnell sich die Windrichtung ändert. Ein höherer Wert bedeutet eine schnellere Änderung der Windrichtung, ein niedriger Wert sorgt für eine beständigere Windrichtung.

4. Fluktuationsgeschwindigkeit der Windintensität: Gibt an, wie schnell die Windintensität fluktuiert. Ein höherer Wert bedeutet hier eine schnelle Änderung der Windintensität des konstanten wehenden Windes, ebenfalls erreichen die Böen schneller ihre Maximalstärke und sind von kürzerer Dauer.

5.2 Umsetzung des Simplex-Noise

Zur Umsetzung des Windes wird der eindimensionale Simplex-Noise-Algorithmus verwendet, da dieser die Zufallskomponente des Windes gut nachbildet. Hierfür wurde eine neue Klasse *SimplexNoiseGenerator* geschaffen. Die Implementierung der dortigen Funktion *Noise* basiert auf einer Open-Source-Implementierung von Stefan Gustavson[9] welche vom Autor in Gemeinfreiheit veröffentlicht wurde. Die Umsetzung entspricht hierbei auf der in Abschnitt 2.2 genannten Funktionsweise der Funktion $noise(x)$, es wird jedoch eine andere Übergangsfunktion verwendet. Der Vorteil dieser Umsetzung ist, dass die Werte der Funktion in einem Intervall $[-1, 1]$ abgebildet werden, was eine einfache Skalierung erlaubt.

Der *SimplexNoiseGenerator* soll die Verwendung der *Noise*-Funktion komfortabler gestalten sollen. Hierzu ist in der Klasse eine Zählvariable *currentNoiseX* gespeichert, die den aktuellen X-Wert der Noise-Funktion speichert. Diese wird von drei zusätzlichen Funktionen verwendet:

- *GenerateNext(step)* gibt den Wert von *Noise(currentNoiseX)* zurück, also einen Wert in einem $[-1, 1]$-Intervall. Anschließend wird *currentNoiseX* um den Wert von *step* erhöht. Diese Funktion ermöglicht das sequentielle Aufrufen mit einer festen Schrittweite.

- *GenerateNext_ OnlyPositive(step)* funktioniert wie *GenerateNext*, falls der Wert von *Noise(currentNoiseX)* negativ ist wird jedoch stattdessen 0 zurückgegeben. Es werden also nur Werte in einem $[0, 1]$-Intervall zurückgegeben. Diese Methode ist geeignet, um die Windböen nachzuahmen, da es nur gelegentliche, positive Ausschläge der Funktion gibt.

- *GenerateNext_ Between0And1(step)* funktioniert ähnlich wie *GenerateNext*, allerdings werden die Werte in ein $[0, 1]$-Intervall verschoben und skaliert.

5.3 Implementierung der Windsimulation

Die Windsimulation ist in der Prozedur *SimulateWind* umgesetzt, welche in Algorithmus 1 aufgeführt ist. Die Prozedur wird in der *Update*-Methode der *MulticopterEntity,* also zu Beginn jedes Frames der Simulation aufgerufen. Über das Fenster „Simulatoreinstellungen" lassen sich die oben genannten Windparameter einstellen: Die Variable *windDirection* speichert die Windrichtung in Grad um die X-Achse, die Grundintensität des Windes ist durch die Variable *windIntensity* gegeben. Die Geschwindigkeit mit welcher die Fluktuation von Windrichtung und Windintensität stattfinden soll wird über die Variablen *windDirectionFluctuation* und *windIntensityFluctuation* angegeben. Weiterhin werden vier Instanzen der *SimplexNoiseGenerator*-Klasse verwendet: *sngConstantWind* generiert die Intensität des ständig wirkenden Windes, *sngWindGust* dagegen ist für die Generierung der Windböen zuständig. Desweiteren wird *sngWindDirectionFluctuation* zur Fluktuierung der Windrichtung verwendet, *sngAttitudeDisturbances* erzeugt anhand der aktuellen Windintensität des Frames die durch den Wind wirkenden Drehmomente auf den Multicopter.

[9] *http://staffwww.itn.liu.se/~stegu/aqsis/aqsis-newnoise/*

Algorithmus 1 Simulation des Windes anhand mehrer Simplex-Noise-Generatorn

```
procedure SimulateWind()
begin
  constantWind: float
  windGust: float
  totalWindIntensity: float
  windDirectionChange: float
  windForceVector: Vector3
  torqueVector: Vector3

  constantWind := sngConstantWind.GenerateNext_Between0And1(
     windIntensityFluctuation);
   constantWind := (constantWind / 2) * (windIntensity / 2);
  windGust := sngWindGust.Generate_OnlyPositive(
     windIntensityFluctuation * 3) * windIntensity;
  totalWindIntensity := constantWind + windGust;
  windDirectionChange = 5 * sngWindDirectionFluctuation.
     GenerateNext(windDirectionFluctuation);
  windForceVector := (1, 0, 0);
  windForceVector.RotateAroundY(windDirection + windDirectionChange
     );
  windForceVector.RotateAroundX(windDirectionChange);
  torqueVector.X = sngAttitudeDisturbances.GenerateNext(rand(0.01,
     0.03)) / 250;
  torqueVector.Y = sngAttitudeDisturbances.GenerateNext(rand(0.01,
     0.03)) / 300;
  torqueVector.Z = sngAttitudeDisturbances.GenerateNext(rand(0.01,
     0.03)) / 250;
  windForceVector := windForceVector * totalWindIntensity;
  torqueVector := torqueVector * totalWindIntensity;

  ApplyForce(windForce); // in Weltkoordinaten
  ApplyTorque(torqueVector); // in Weltkoordinaten
end
```

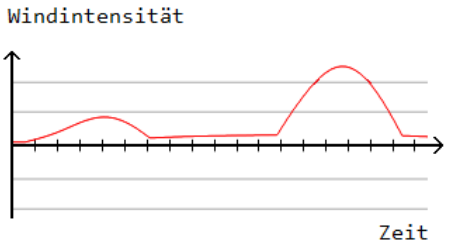

Abbildung 18: Verlauf der Windintensität im Simulator mit zwei Windböen

Bei Aufruf von *SimulateWind* werden zunächst die Intensitäten des konstanten Winds und der Windböe für den aktuellen Frame berechnet. Der konstante Wind wird über ein Simplex-Noise mit Intervall $[0, 1]$ und geringer Frequenz berechnet, die Windböe wird mithilfe der *GenerateNext_OnlyPositive*-Funktion und einer höheren Frequenz berechnet. Die Grundfrequenz beider Noise-Verfahren wird über den Wert von *windIntensityFluctuation* vorgegeben. Anschließend findet die Berechnung der Windrichtung statt: Hierzu wird dem Vektor *windForceVector* ein Einheitsvektor zugewiesen und entsprechend des durch *windDirection* vorgegebenen Winkel um die Y-Achse rotiert, ebenso fließt die über *GenerateNext* berechnete Richtungsfluktuation ein. Da Winde häufig nicht nur horizontal wirken, wird der Vektor ebenfalls um den Wert von *windDirectionChange* um die X-Achse rotiert, was eine vertikale Komponente einfließen lässt.
Weiterhin wird ein Vektor *torqueVector* berechnet, der das durch den Wind verursachte Drehmoment vorgeben soll. Die Komponenten dieses Vektors werden ebenfalls mit der Noise-Funktion berechnet, die Frequenz wird durch die Verwendung von *rand* hierbei zu-

fällig verändert, um ein chaotischeres Verhalten zu erzeugen. Anschließend wird gemäß der für den aktuellen Frame berechneten Windintensität eine Skalierung von *windForceVector* und *torqueVector* durchgeführt. Zuletzt werden die berechnete Kraft und das berechnete Drehmoment mittels *ApplyForce* bzw. *ApplyTorque* auf den Multicopter angewandt.

6 Simulation der Lageregelung

Algorithmus 2 PID-Regler umgesetzt in algorithmische Form

```
kp, ki, kd: float
integral, prevError: float

function control
  (outputValue: float, referenceValue: float): float
begin
  error, integral, derivate: float

  error := referenceValue - outputValue;
  integral := integral + error;
  derivate := error - prevError;

  prevError := error;

  return kp * error + ki * integral + kd * derivate;
end
```

Im Folgenden soll die Umsetzung der Lageregelung für den simulierten Multicopter erörtert werden. Diese ist, analog zur Unterteilung der Stabilität in Abschnitt 3.1 ebenfalls in drei Komponenten unterteilt:

1. Regulierung der Fluglage: Der Multicopter soll eine stabile Fluglage halten, d.h. eine aufrechte Lage aufweisen und parallel zum Boden positioniert sein.

2. Regulierung der Höhe: Der Multicopter soll ungeachtet seiner Fluglage möglichst eine konstante Höhe halten.

3. Regulierung der Position: Der Multicopter soll eine bestimmten Position in der XZ-Ebene halten. Wenn Wind den Multicopter von der zu haltenden Position wegtreibt, soll diese Stabilisierungskomponente dem entgegensteuern.

Die Umsetzung der genannten Lageregelungskomponeten wurde mithilfe von PID-Reglern vorgenommen, da diese vergleichsweise einfach zu verstehen und umzusetzen sind. Die Unterteilung der Stabilität liegt darin begründet, dass für die drei Stabilisierungsformen unterschiedliche Steuersignale geregelt werden müssen - es müssen daher ebenfalls mehrere Regler verwendet werden. Dies macht eine Möglichkeit zur Instanzierung vonnöten, daher wurde eine softwaretechnische Umsetzung in Form einer eigenen Klasse namens *PIDController* vorgenommen. Die grundlegende Funktionsweise der in der Klasse implementierten Regelung ist in Algorithmus 2 in Form von Pseudocode dargestellt. Im vornerein müssen die Variablen *kp, ki* und *kd* festgelegt werden, diese speichern die für den PID-Regler eingestellten Konstantenwerte K_P, K_I und K_D. Die Funktion *control* übernimmt den Regelvorgang, an diese wird über den Parameter *outputValue* der Ausgabesignalwert, sowie über den Parameter *referenceValue* der Referenzsignalwert übergeben. Als Ergebnis wird ein neuer Steuersignalwert zurückgegeben. Die Regelung findet gemäß der in Abschnitt 2.3 genannten Formeln statt, jedoch wurde die zeitliche Abhängigkeit der Werte vernachlässigt.

Algorithmus 3 Beispiel zur Anwendung der Regelfunktion

```
output: float
input: float
target: float

// Setze in target den Referenzwert fest

// output soll an den Wert target angeglichen werden
while True do
begin
  // input-Variable führt zu Änderung der output-Variable

  // Passe mittels Regelungsfunktion den Wert von input neu an
  input = control(output, target);
end
```

Die Funktionsweise der Funktion *control* wird nachfolgend erläutert: Zunächst wird die Fehlerfunktion berechnet und das Ergebnis in der Variable *error* gespeichert. Anschließend findet die Berechnung des Integrals der integralen Komponente statt. Die Akkumulierung ist hierbei über eine Aufsummierung der vorangegangenen Fehler in der Variable *integral* umgesetzt. Die Ableitung der derivativen Komponente wird über die Differenz des aktuellen Fehlers mit dem in der Variable *prevError* gespeicherten vorherigen Fehlers gebildet und in der Variable *derivate* gespeichert. Anschließend wird *prevError* auf den aktuellen Fehler gesetzt. Zuletzt werden die PID-Komponenten jeweils mit den zugehörigen Konstanten multipliziert und aufsummiert, dieser Wert wird anschließend zurückgegeben und entspricht dem neu einzustellenden Steuersignalwert. Wie die *control*-Funktion verwendet wird ist in Algorithmus 3 verdeutlicht: *output* sei hierbei der Ausgabesignalwert, *input* der Steuersignalwert und *target* der Referenzwert. Die eigentliche Regelung findet statt, in dem die Funktion *control* in einer Schleife immer wieder aufgerufen wird, die entsprechenden Parameter übergeben bekommt und den Wert des Steuersignals neu einstellt, was wiederum zu einer Änderung des Ausgabewertes führt. Die PID-Regler wurden in der Simulation nach dem eben genannten Prinzip innerhalb der Klasse *PIDController* verwirklicht.

In der Multicopter-Simulation wird die *Control*-Funktionen der PID-Regler ständig in der *Update*-Methode der *MulticopterEntity* aufgerufen. Dies ist auch der Grund, warum die zeitliche Abhängigkeit der Werte vernachlässt wurde: Die Framerate der Simulation ist häufig nicht konstant, was dazu führt, dass die *Update*-Methode in ungleichen Zeitintervallen ausgeführt wird. Dies hat zur Folge, dass kein konsistentes Tuning der PID-Controlle vorgenommen werden kann. Wird die zeitliche Abhängigkeit nicht beachtet tritt dieses Problem nicht auf. Ein weiterer Punkt der vernachlässigt wurde ist, dass die simulierten Sensoren in der Realität oft einem Rauschen unterliegen. Der Simulator hingegen gibt die benötigten Werte des Multicopters präzise wieder.

Im Folgenden wird die Umsetzung der oben genannten Lageregelungskomponenten beschrieben:

1. Implementierung der Fluglageregelung:
   Ziel ist es, dass der Multicopter aufrecht und parallel zum Boden ausgerichtet ist. Dies ist der Fall, wenn die Rotation um X- und Z-Achse 0° beträgt. Es werden zwei Instanzen der Klasse *PIDController* verwendet: *pidControllerAttitudeX* übernimmt die Regulierung der X-Rotation, *pidControllerAttitudeZ* ist für die Regulierung der Z-Rotation zuständig.
   In jedem Frame wird die *Control*-Methode beider Regler aufgerufen, beide bekommen als Referenzwert den Wert 0 für eine gewünschte Rotation von 0° übergeben. An

pidControllerAttitudeX wird hierbei als Ausgabewert die aktuelle Rotation um die X-Achse übergeben. Der Steuerwert, welchen die Funktion als Ergebnis zurückgibt wird in einer Variable *attitude_ dx* gespeichert. Analog wird an den *pidControllerAttitudeX* als Ausgabewert die aktuelle Rotation um die Z-Achse übergeben, der Steuerwert, welcher als Ergebnis zurückgegeben wird, wird in der Variable *attitude_ dz* gespeichert.
Anschließend wird mittels Aufruf der Prozedur *ApplyTorque* ein Drehmoment gewirkt, hierbei wird ein Vektor $\vec{v}$ in Form von Weltkoordinaten übergeben:

$$\vec{v} = \begin{pmatrix} attitude\_dx \\ 0 \\ attitude\_dz \end{pmatrix}$$

Das Drehmoment bewirkt eine allmähliche Stabilisierung der Fluglage - der Multicopter wird rotiert, bis er sich parallel zum Boden befindet.

2. Implementierung der Höhenregelung:
Die Höhe des Multicopters ist durch die Y-Komponente des in Weltkoordinaten betrachteten Positionsvektors repräsentiert. Im Steuerungs-Fenster des Simulators kann eine zu haltende Höhe in Metern festgelegt werden. Es wird ein PID-Regler-Instanz *pidControllerAltitude* verwendet, bei welchem als Referenzwert die zu haltende Höhe gesetzt wird. In jedem Frame wird die *Control*-Methode des Reglers aufgerufen, als Ausgabewert wird die aktuelle Höhe gesendet. Der zurückgebene Steuerwert wird in die Variable *raw_ thrust* geschrieben, welche den Schub des Multicopters steuert. Dieser Vorgang führt dazu, dass der Multicopter durch die Regulierung des Schubes die gewünschte Höhe anfliegt und hält.

3. Implementierung der Positionsregelung:
Die Position ist durch X- und Z-Komponente des in Weltkoordinaten betrachteten Positionsvektor repräsentiert. Ähnlich wie bei der Höhe können im Steuerungs-Fenster des Simulators die zu haltenden Koordinaten angegeben werden. Damit der Multicopter in eine bestimmte Richtung fliegt muss er entsprechend genickt bzw. gerollt werden. Die folgende Auflistung ordnet den Flugrichtungen die Achsenrotationen zu, welche dazu ausgeführt werden müssen (die Betrachtung bezieht sich auf Weltkoordinaten):

   - Damit der Multicopter in positive X-Richtung fliegt, muss er von der Ausgangslage mit einem negativen Wert um die Z-Achse rotiert werden
   - Damit der Multicopter in negative X-Richtung fliegt, muss er von der Ausgangslage mit einem positiven Wert um die Z-Achse rotiert werden
   - Damit der Multicopter in positive Z-Richtung fliegt, muss er von der Ausgangslage mit einem positiven Wert um die X-Achse rotiert werden
   - Damit der Multicopter in negative Z-Richtung fliegt, muss er von der Ausgangslage mit einem negativen Wert um die X-Achse rotiert werden

   Für die Steuerung der Position werden zwei PID-Regler verwendet. *pidControllerPositionX* reguliert die X-Komponente, *pidControllerPositionZ* die Z-Komponente der Position. Als Referenzwerte werden an die entsprechenden Regler die zu haltenden X- bzw. Z-Koordinate übergeben, als Ausgabewerte werden dementsprechend die X- bzw. Z-Koordinaten des aktuellen Frames zugewiesen. Als Steuerwerte werden zwei Variablen *pos_ x_ controlValue* und *pos_ z_ controlValue* verwendet.
   Die Auflistung zeigt, dass eine Anpassung der Position nur über eine Änderung der

Fluglage vorgenommen werden kann. *pos_ controlValue_ x* gibt den Winkel an, um welchen der Multicopter um die Z-Achse rotiert sein muss, *pos_ controlValue_ z* analog den Winkel für die X-Achse. Beide Steuerwerte werden auf einen Winkel von 25° bzw. -25° limitiert, damit der Multicopter nicht kopfüber fliegt. Es tritt nun ein Sonderfall ein: Die Steuerwerte werden als Referenzwerte an die PID-Regler der Fluglagereglung übergeben, *pos_ controlValue_ x* wird der Referenzwert von *pidControllerAttitudeZ* und *pos_ controlValue_ z* der Referenzwert von *pidControllerAttitudeX*. Durch die Mitverwendung der Fluglageregelung werden die Winkel genau eingehalten. Auf diese Weise kann der Multicopter an eine bestimmte Position annavigieren und halten.

7 Experimente und Auswertung

In den folgenden Experimenten soll untersucht werden, ob die Stabilität des Multicopters von der Anzahl der Propeller und dem Neigungswinkel der Propeller abhängt. Alle Experimente wurden unter Zuhilfenahme der im Rahmen dieser Arbeit entwickelten Computersimulation getätigt.

7.1 Versuchsaufbau

Für die Experimente wird der folgende virtuelle Versuchsaufbau verwendet: Der simulierte Multicopter soll durch Ansteuerung der Lageregelung auf eine Höhe von $15m$ steigen und anschließend den Ursprung der XZ-Ebene anfliegen. Mithilfe der drei PID-Regler soll also der Punkt $P = (0, 15, 0)$ im Luftraum gehalten werden, währenddessen wirkt ein Wind auf den Multicopter, welcher den Flugkörper ständig aus seiner Solllage bringt.
Im Simulator sind entsprechende Einstellungen vorgenommen worden. Für die PID-Regler ist nach heuristischen Methoden ein Tuning durchgeführt worden, weiterhin sind die Windeinstellungen an einen langsam fluktuierenden, mittelstarken Wind angepasst. Für alle Experimente werden dieselben Einstellungen verwendet, diese sind in Tabelle 1 aufgeführt.

Einstellung	Wert
PID-Regler Attitude (Fluglage)	$(P, I, D) = (2, 0.001, 100)$
PID-Regler Altitude (Höhe)	$(P, I, D) = (2, 0.001, 50)$
PID-Regler Position	$(P, I, D) = (5, 0, 50)$
Wind direction (Windrichtung)	45
Wind intensity (Intensität des Windes)	1.2
Wind direction fluctuation (Änderungsrate der Windrichtung)	1
Wind intensity fluctuation (Änderungsrate der Windintensität)	1

Tabelle 1: Einstellungen des Simulators, welche für alle Experimente verwendet werden

7.2 Maß der Stabilität

Um die Stabilität bewerten zu können, müssen gemeinsame Kriterien gefunden werden. Als Maß der Stabilität für die in Abschnitt 3.1 genannten Stabilitätskomponenten werden folgende Formeln verwendet:

1. Stabilität der Fluglage: Es ist hier besonders zu beachten, wie stark die vom Wind verursachte Schwingung und die Ausgleichsbewegung der Lageregelung wirkt. Dies wird über die Winkelgeschwindigkeiten entlang der Nick- und der Rollachse repräsentiert.

Sei ω_X die Winkelgeschwindigkeit der Rollachse und ω_Z die Winkelgeschwindigkeit der Nickachse. Als Maß für die Stabiltät der Fluglage wurde gewählt:

$$\delta_{att}(t) = |\omega_X(t)| + |\omega_Z(t)|$$

Auf die Winkelgeschwindigkeiten wird die Betragsfunktion angewendet, da die Winkelgeschwindigkeit abhängig von der Richtung ist und daher negativ sein kann. Die Richtung kann hier allerdings vernachlässigt werden, aussagekräftig ist nur der Betrag der Abweichung.

2. Stabilität der Höhe: Es wird gemessen wie stark die Abweichung der aktuellen Höhe $p_y(t)$ von der zu haltenden Höhe $15\,m$ ist. Auch hier ist lediglich der Betrag der Abweichung relevant.
$$\delta_{alt}(t) = |p_y(t) - 15m|$$

3. Stabilität der Position: Die Position bezieht sich auch hier auf die XZ-Ebene. Es wird gemessen wie stark die Abweichung des Multicopters von der zu haltenden Position ist. Da die zu haltende Position dem Ursprung der XZ-Ebene entspricht, kann die Abweichung der Koordinaten über die Betragsfunktionen der Koordinaten berechnet werden. p_x enthält hierbei die X-Koordinate, p_z die Z-Koordinate. Um die Gesamtabweichung zu erhalten, können die Koordinatenabweichungen aufaddiert werden:
$$\delta_{pos}(t) = |p_x(t)| + |p_z(t)|$$

7.3 Abhängigkeit der Stabilität von der Propelleranzahl

Um festzustellen, ob eine Abhängigkeit zwischen den im vorigen Abschnitt genannten Stabilitäten und der Propelleranzahl gibt wird eine Testreihe durchgeführt. In der Testreihe werden für Tricopter, Quadrocopter, Pentactoper, Hexacopter, Heptacopter und Octocopter alle im vorigen Abschnitt genannten Stabilitätswerte jeweils 30 Sekunden lang aufgezeichnet. Anschließend werden die Werte mit Box-Whiskers-Diagrammen grafisch gegenübergestellt.
In dem Diagramm in Abbildung 19 ist die Stabilität der Fluglage gegenüber der Anzahl

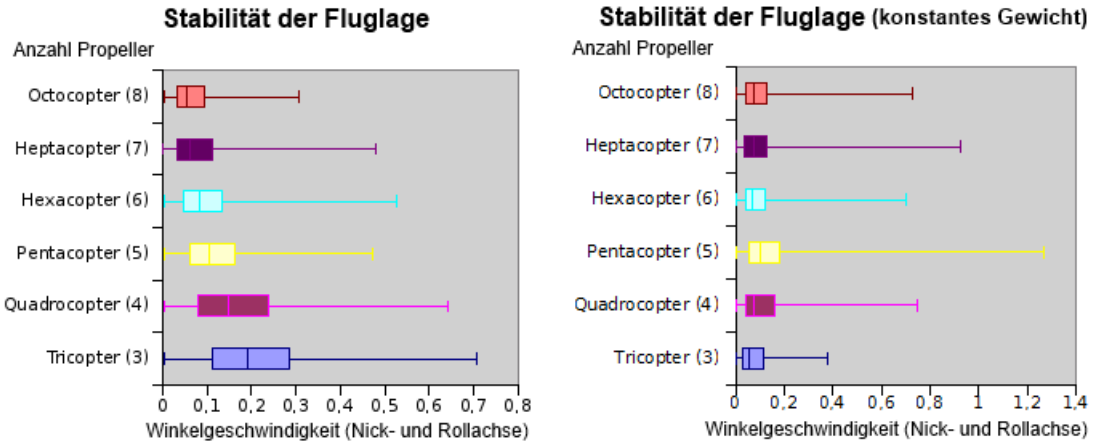

Abbildung 19: Box-Whiskers-Diagramm, welches die Stabilität der Fluglage in Abhängigkeit von der Anzahl der Propeller darstellt. Im rechten Diagramm wurde zusätzlich das Gewicht des Multicopters konstant gehalten.

der Propeller dargestellt. Aus dem linken Diagramm ist ersichtlich, dass mit steigender Anzahl der Propeller scheinbar auch die Stabilität der Fluglage steigt. Jedoch lag die Vermutung nahe, dass dies mit der Zunahme des Gewichts des Multicopters bei steigender

Propelleranzahl zusammen hängt. Deswegen wurde eine zweite Messreihe gestartet wurde, bei der das Gewicht des Multicopters konstant auf 500g gehalten wurde. Wie das rechte Diagramm derselben Abbildung zeigt, hat sich die Vermutung bestätigt: Es kann keine Abhängigkeit der Stabilität von der Propelleranzahl festgestellt werden, lediglich eine Erhöhung des Gewichts führt zu einer Stabilitätszunahme. Das Diagramm in Abbildung 20

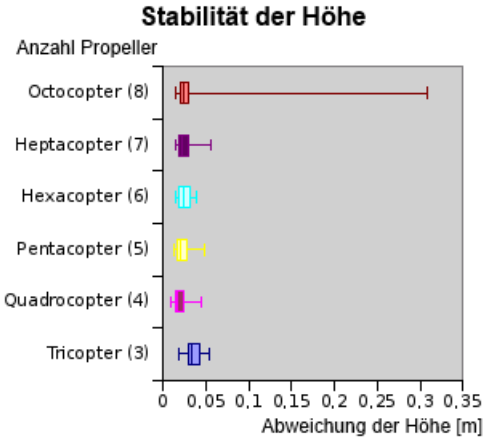

Abbildung 20: Box-Whiskers-Diagramm, welches die Stabilität der Höhe in Abhängigkeit von der Anzahl der Propeller darstellt

stellt die Stabilität der Höhe gegenüber der Anzahl der Propeller dar. Es konnte keine signifikante Abhängigkeit festgestellt werden. In Abbildung 21 sind zwei Diagramme, welche

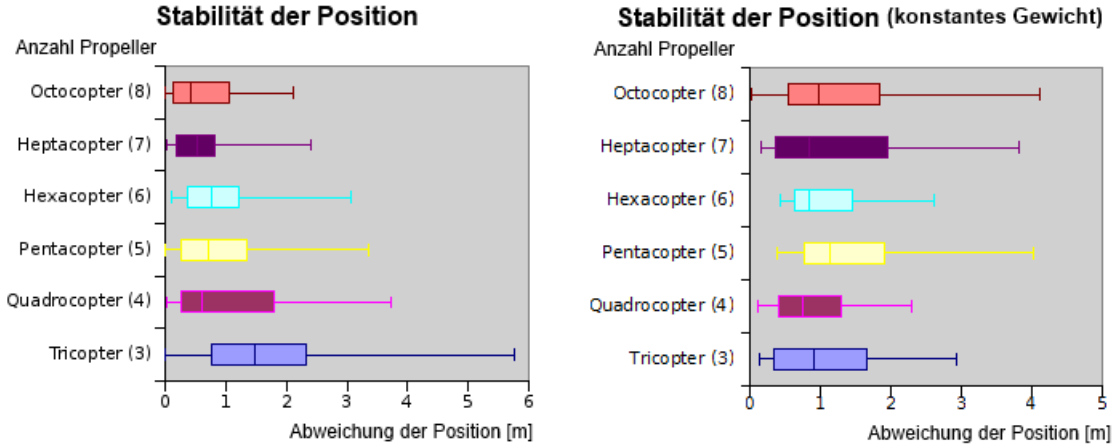

Abbildung 21: Box-Whiskers-Diagramm, welches die Stabilität der Position in der XZ-Ebene in Abhängigkeit von der Anzahl der Propeller darstellt. Im rechten Diagramm wurde zusätzlich das Gewicht des Multicopters konstant gehalten.

die Stabilität der Position in der XZ-Ebene gegenüber der Anzahl der Propeller dargestellen. Wie im linken Diagramm erkennbar, erhöht sich die Positionsstabilität scheinbar mit zunehmender Propelleranzahl. Auch hier lag der Verdacht nahe, dass dies mit einem höheren Gesamtgewicht zusammenhängt. Das rechte Diagramm bestätigt diese Annahme: Bleibt das Gewicht konstant für jede Anzahl der Propeller bei konstanten 500g kann keine Verbesserung der Positionsstabilität festgestellt werden.

7.4 Abhängigkeit der Stabilität vom Neigungswinkel

Um festzustellen, ob eine Abhängigkeit zwischen den im vorigen Abschnitt genannten Stabilitätskomponenten und dem Neigungswinkel der Propeller besteht, wird analog zum vo-

rigen Abschnitt auch hier eine Testreihe durchgeführt. Die Testreihe umfasst Anpassungen der Neigungswinkel 0°, 10°, 20°, 30° und 40° bei einem Quadrocopter (also einer konstanten Anzahl Propeller). Auch hier werden die drei Stabilitäten für die Menge der Neigungswinkel jeweils 30 Sekunden lang aufgezeichnet und grafisch dargestellt. In dem Diagramm in

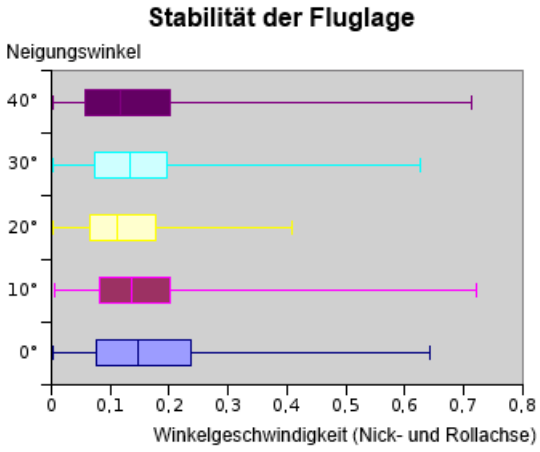

Abbildung 22: Box-Whiskers-Diagramm, welches die Stabilität der Fluglage in Abhängigkeit vom Neigungswinkel der Propeller darstellt.

Abbildung 22 ist die Stabilität der Fluglage gegenüber dem Neigungswinkel dargestellt. Es lässt sich keine signifikante Veränderung der Fluglagestabilität feststellen. Das Diagramm

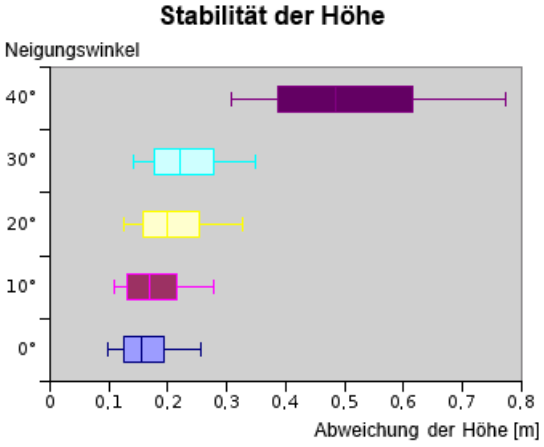

Abbildung 23: Box-Whiskers-Diagramm, welches die Stabilität der Höhe in Abhängigkeit vom Neigungswinkel der Propeller darstellt.

in Abbildung 23 zeigt die Stabilität der Höhe gegenüber dem Neigungswinkel. Scheinbar sinkt die Stabilität mit steigendem Neigungswinkel., jedoch ist während der Versuchsdurchführung ist aufgefallen, dass die PID-Regler mit steigendem Neigungswinkel eine längere Zeit zum Anpassen der Höhe benötigen, was die Ursache der Abweichungen erklärt. Ansonsten kann kein Zusammenhang zwischen Neigungswinkel und Höhenstabilität festgestellt werden. In Abbildung 24 stellt das Diagramm die Stabilität der Position in der XZ-Ebene in Abhändigkeit zum Neigungswinkel dar. Es ist keine signifikante Veränderung der Stabilität feststellbar, daher ist auch die Positionsstabilität offensichtlich unabhängig vom Neigungswinkel der Propeller.

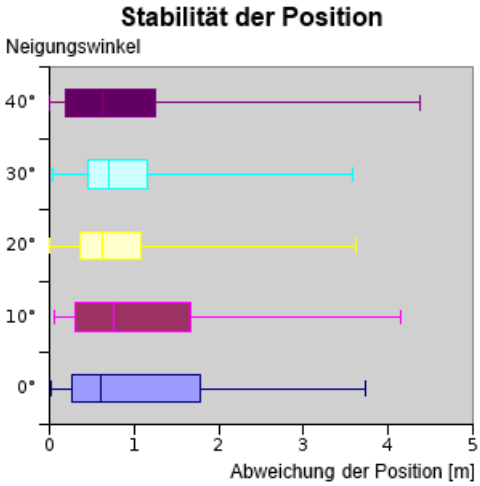

Abbildung 24: Box-Whiskers-Diagramm, welches die Stabilität der Position in der XZ-Ebene in Abhängigkeit vom Neigungswinkel der Propeller darstellt

7.5 Fazit

Die Untersuchungen haben ergeben, dass der Neigungswinkel und die Anzahl Propeller keine Auswirkung auf die Positionsstabiliät, Höhenstabilität und Fluglagestabilität haben. Lediglich eine Erhöhung des Gewichts hat eine höhere Positions- und Fluglagestabilität zur Folge, da offensichtlich mehr Kraft bzw. ein stärkes Drehmoment aufgewandt werden muss, um den Multicopter zu bewegen. Jedoch ist zu beachten, dass sich eine Erhöhung des Multicoptergewichts negativ auf die Maximalflugdauer auswirkt, da zum Fliegen eine höhere Propellerkraft und damit mehr Strom benötigt wird. Da die Nachteile in diesem Fall überwiegen, eignet sich eine Gewichtserhöhung nicht als Mittel zur Erhöhung der Stabilität,
Wie andere wissenschaftliche Arbeiten gezeigt haben, scheint der Ansatz durch Verbesserung der Lageregelung und der Steuersoftware eine höhere Stabilität zu erreichen vielversprechender zu sein. Es ist zu erwähnen, dass durch Ungenauigkeiten im Modell und dem Umstand, dass Computersimulationen die Realität nicht vollständig nachbilden können, die hier dargestellten Experimente Fehler aufweisen können.

8 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit ist es gewesen, Ansätze zu untersuchen, die Stabilität von Multicoptern und deren Unempfindlichkeit gegenüber Wind zu untersuchen, sowie Möglichkeiten zur Verbesserung zu finden. Die Grundidee war hierbei, dass die Anzahl der Propeller, sowie der Neigungswinkel der Propeller Auswirkung auf die Stabilität haben und die Veränderung dieser Parameter eventuell zu einer Verbesserung der Stabilität führen. Hierzu ist mithilfe von Microsoft Robotics Developer Studio 4 eine Simulationsumgebung umgesetzt worden, in welcher mithilfe einer Physikengine das Flugverhalten des Multicopters nachempfunden wurde. Es wurde hier die Möglichkeit gegeben die Propelleranzahl, sowie die Neigungswinkel dynamisch anzupassen. Ebenfalls umgesetzt wurde ein Modell zur Simulation von Wind mithilfe des Simplex-Noise-Algorithmus, durch welchen Richtung und Intensität des Windes fluktuieren. Mithilfe der zusätzlichen Anwendung von PID-Reglern zur Lageregelung konnte ein Versuchsaufbau umgesetzt werden, in welchem der Multicopter schwebend eine feste Position im Raum hält. Desweiteren sind Kriterien zur Bestimmung der Stabilität bestimmt worden. In den Experimenten wurde anhand dieser Kriterien untersucht, ob es Abhängigkeiten zwischen Stabilität und Anzahl bzw. Neigungswinkel der Propeller gibt, es konnte jedoch keine Abhängigkeit festgestellt werden. Lediglich durch

Erhöhung des Gewichts kann eine leichte Stabilitätsverbesserung erzielt werden, was in der Praxis jedoch nicht anwendbar ist. Insgesamt wurde keine Möglichkeit gefunden eine Verbesserung zu erzielen.
Es sollte eher der in anderen Arbeiten verwendete Ansatz, durch Verbesserung der Lageregelung und der Steuersoftware eine höhere Stabilität zu erreichen, weiterverfolgt werden. Weitere Konzepte könnten darin liegen, zu untersuchen, ob durch Neigung einzelner Propeller (wie beim Tricopter) eventuell eine Stabilisierung vorgenommen werden kann. Ferner könnte die Herangehensweise untersucht werden den Schwerpunkt des Multicopters tiefer zu legen, dies hat eventuell einen positiven Effekt auf die Stabilität der Fluglage, da ein höherer Widerstand gegenüber Drehmomenten besteht. Die in dieser Arbeit umgesetzte Computersimulation bietet hierfür die Basis weitere, neu gefundene Ansätze näher zu untersuchen. Es ist zu beachten, dass das Modell des Multicopters einige Schwächen aufweist: Beispielsweise erfolgt die Betrachtung des Luftwiderstands nur rudimentär. Eine Ungenauigkeit stellt ebenfalls die Achsenrotation des Multicopters dar, da nur ein Drehmoment gewirkt wird und nicht die einzelnen Propellerkräfte berücksichtigt werden. Das in dieser Arbeit verwendete Windmodell hingegen dürfte die in der realen Welt auftretenden Windphänomene wie Windböen ausreichend nachahen und als Basis für weitere Experimente nützlich sein. Werden Experimente mit verschiedener Regelungstechnik durchgeführt, erfordert dies ebenfalls nur geringe Anpassungen: Es müssen lediglich die dafür zuständigen Prozeduren mit den entsprechenden neuen Steuerungsmethoden angepasst werden. Die Computersimulation kann also für andere Projekte weiterverwendet werden, nicht zuletzt bietet er die Möglichkeit, das Fliegen des Multicopters zu üben, ohne Schaden anzurichten.

Literatur

[ANT11] Kostas Alexis, George Nikolakopoulos, and Anthony Tzes. Switching model predictive attitude control for a quadrotor helicopter subject to atmospheric disturbances. *Control Engineering Practice*, 19:1195–1207, 2011.

[ANTD09] K. Alexis, G. Nikolakopoulos, A. Tzes, and L. Dritsas. Coordination of helicopter uavs for aerial forest-fire surveillance. In Kimon P. Valavanis, editor, *Applications of Intelligent Control to Engineering Systems*, volume 39 of *Intelligent Systems, Control, and Automation: Science and Engineering*, pages 169–193. Springer Netherlands, 2009.

[BB07] Adrian Boeing and Thomas Bräunl. Evaluation of real-time physics simulation systems. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, GRAPHITE '07, pages 281–288, New York, NY, USA, 2007. ACM.

[Bit09] Walter Bittner. *Flugmechanik der Hubschrauber: Technologie, das flugdynamische System Hubschrauber, Flugstabilitäten, Steuerbarkeit.* Springer, 2009.

[BNS04] Samir Bouabdallah, André Noth, and Roland Siegwart. PID vs LQ Control Techniques Applied to an Indoor Micro Quadrotor. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2451–2456, 2004.

[BS07] Samir Bouabdallah and Roland Siegwart. Full control of a quadrotor. In *IEEE/RSJ International Conference on Intelligent Robots and Systems 2007*, pages 153–158, 2007.

[Bur08] Wilhelm Burger. Gradientenbasierte Rauschfunktionen und Perlin Noise. Technical Report HGBTR08-02, School of Informatics, Communications and Media, Upper Austria University of Applied Sciences, Hagenberg, Austria, November 2008.

[CS12] Roman Czyba and Grzegorz Szafranski. Control structure impact on the flying performance of the multi-rotor VTOL platform - design, analysis and experimental validation. *International Journal of Advanced Robotic Systems*, 2012.

[DH11] R. D'Andrea and M. Hehn. Quadrocopter trajectory generation and control. In *Proceedings of the IFAC world congress*, pages 1485–1491, 2011.

[Dis09] Dominic J. Diston. *Computational Modelling of Aircraft and the Environment: Platform Kinematics and Synthetic Environment*, volume 1 of *Aerospace Series*. John Wiley & Sons, Ltd., Chichester, West Sussex, UK, 2009.

[GGB12] S. Grzonka, G. Grisetti, and W. Burgard. A fully autonomous indoor quadrotor. *IEEE Transactions on Robotics*, 28:90–100, 2012.

[GGR12] Vaibhav Ghadiok, Jeremy Goldin, and Wei Ren. On the design and development of attitude stabilization, vision-based navigation, and aerial gripping for a low-cost quadrotor. *Autonomous Robots*, 33, 2012.

[JT08] Kyle Johns and Trevor Taylor. *Professional Microsoft Robotics Developer Studio*. Wrox Programmer to Programmer. Wrox Press Ltd., Birmingham, UK, 2008.

[LPLK12] Hyon Lim, Jaemann Park, Daewon Lee, and H.J. Kim. Build your own quadrotor: Open-source projects on unmanned aerial vehicles. *IEEE Robotics & Automation Magazine*, 19:33–45, 09 2012.

[MKC12] Robert Mahony, Vijay Kumar, and Peter Corke. Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor. *IEEE Robotics & Automation Magazine*, 19:20–32, 09 2012.

[Per02] Ken Perlin. Improving noise. In *SIGGRAPH'02: Proceedings of the 29th Annual COnference on Computer Graphics and Interactive Techniques*, pages 681–682, New York, NY, USA, 2002.

[ROR10] Guilherme V. Raffo, Manuel G. Ortega, and Francisco R. Rubio. An integral predictive/nonlinear control structure for a quadrotor helicopter. *Automatica*, 46:29–39, 2010.

[Sel01] David Sellers. An Overview of Proportional plus Integral plus Derivative Control and Suggestions for its Successful Application and Implementation. In *Proceedings of the First International Conference for Enhanced Building Operations*, Austin, Texas, USA, July 2001. Portland Energy Conservation Inc.

[TM06] Abdelhamid Tayebi and Stephen McGilvray. Attitude stabilization of a VTOL quadrotor aircraft. *IEEE Transactions on Control Systems Technology*, 14:562–571, 2006.

[Vag12] Marialena Vagia. *PID Controller Design Approaches - Theory, Tuning and AApplication to Frontier Areas.* Intech, 2012.

[YGP09] Xilin Yang, Matt Garratt, and Hemanshu Pota. Design of a gust-attenuation controller for landing operations of unmanned autonomous helicopters. *18th IEEE International conference on control applications*, pages 1300–1305, July 2009.

[YGP11] Xilin Yang, Matt Garratt, and Hemanshu Pota. Flight validation of a feedforward gust-attenuation controller for an autonomous helicopter. *Robotics and Autonomous Systems*, 59:1070–1079, 2011.