UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

ist
Institut für Softwaretechnik

Schnell wachsende Suchbäume zur Pfadplanung für allgemeine Gliederfahrzeuge

Masterarbeit

zur Erlangung des Grades eines Master of Science
im Studiengang Informatik

vorgelegt von

Christian Tobias Eiserloh

Erstgutachter: Prof. Dr. Dieter Zöbel
Institut für Softwaretechnik

Zweitgutachter: Dipl.-Inform. Benjamin Knopp
Institut für Softwaretechnik

Koblenz, im Mai 2013

Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbstständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel — insbesondere keine im Quellenverzeichnis nicht benannten Internet–Quellen — benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-Rom).

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Abstract

This master thesis deals basically with the design and implementation of a path planning system based on rapidly exploring search trees for general-n-trailers. This is a probabilistic method that is characterized by a fast and uniform exploration. The method is well established, however, has been applied only to vehicles with simple kinematics to date. General-n-trailers represent a particular challenge as their controllability is limited. For this reason the focus of this thesis rests on the application of the mentioned procedure to general-n-trailers. In this context systematic correlations between the characteristics of general-n-trailers and the possibilities for the realization and application of the method are analyzed.

Zusammenfassung

In dieser Masterarbeit geht es im Wesentlichen um die Umsetzung eines Pfadplanungsverfahrens basierend auf schnell wachsenden Suchbäumen für allgemeine Gliederfahrzeuge. Es handelt sich dabei um ein probabilistisches Verfahren, das sich durch eine schnelle und gleichmäßige Exploration auszeichnet. Das Verfahren ist etabliert, wurde bisher allerdings nur auf Fahrzeuge mit einfacher Kinematik angewendet. Die im Rahmen dieser Masterarbeit betrachteten Gliederfahrzeuge stellen mit ihrer eingeschränkten Steuerbarkeit eine besondere Herausforderung dar. Im Fokus dieser Masterarbeit steht daher die Anwendung des genannten Verfahrens auf allgemeine Gliederfahrzeuge. Dabei werden systematische Zusammenhänge zwischen den Fahrzeugeigenschaften von Gliederfahrzeugen und den Möglichkeiten zur Realisierung und Anwendung des Verfahrens untersucht.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Diese Masterarbeit ist in der Arbeitsgruppe Echtzeitsysteme im Fachbereich Informatik an der Universität Koblenz-Landau entstanden. Die Arbeitsgruppe Echtzeitsysteme beschäftigt sich unter anderem mit autonomem und assistiertem Fahren. Der Fokus liegt dabei auf allgemeinen Gliederfahrzeugen, sogenannten *General-n-Trailern*. In diesem Zusammenhang stellt die Pfadplanung einen Forschungsschwerpunkt dar.

1.1 Motivation

Die im Rahmen dieser Arbeit betrachtete Pfadplanung beschreibt eine Suche nach einem Pfad für ein Fahrzeug von einem Ausgangszustand zu einem Ziel. Der Pfad wird dabei zum Beispiel als Folge von Bewegungen verstanden. Bei der Pfadplanung möchte man sowohl die Fahrbarkeit des geplanten Pfades, als auch die Kollisionsfreiheit während der geplanten Bewegung sicherstellen. Unter Fahrbarkeit versteht man die Einhaltung bestimmter Randbedingungen, die den Fahrzeugzustand und die Fahrzeugbewegung betreffen. Unter Kollisionsfreiheit versteht man die Vermeidung von Kollisionen mit Hindernissen.

Für Fahrzeuge mit einfacher Kinematik lassen sich hierbei vollständige Verfahren wie z.B. A^* anwenden. Im Gegensatz dazu zeichnen sich allgemeine Gliederfahrzeuge (*General-n-Trailer*) durch einen hochdimensionalen Zustandsraum aus. Gleichzeitig sind sie in ihrer Steuerbarkeit eingeschränkt. Damit wird die Suche so komplex, dass

vollständige Verfahren nicht mehr anwendbar sind.

Als Alternative bieten sich probabilistische Verfahren an. Mit diesen Verfahren ist es möglich, auch hochdimensionale Suchräume effizient zu explorieren. Es existiert ein probabilistisches Verfahren, das auf schnell wachsenden Suchbäumen basiert, das Verfahren der sogenannten *Rapidly-exploring Random Trees* ([LaValle, 1998], [LaValle u. Kuffner, 2000]). Im Fokus dieser Arbeit steht die Anwendung dieses Verfahrens zur Pfadplanung für allgemeine Gliederfahrzeuge (*General-n-Trailer*).

1.2 Ziele der Arbeit

Bei dem Ansatz der schnell wachsenden Suchbäume (*Rapidly-exploring Random Trees*, kurz: *RRT*s) ([LaValle, 1998], [LaValle u. Kuffner, 2000]) handelt es sich um ein etabliertes, probabilistisches Verfahren zur Exploration von Such- bzw. Kartenräumen. Im Rahmen der Arbeit soll dieses Verfahren auf allgemeine Gliederfahrzeuge (*General-n-Trailer*) angewendet werden.

Es existieren bereits zahlreiche Anwendungen des Verfahrens auf Fahrzeuge oder generell auf Aktoren mit einfacher Kinematik. Die im Rahmen dieser Arbeit betrachteten Gliederfahrzeuge stellen mit ihrem hochdimensionalen Konfigurationsraum und ihrer eingeschränkten Steuerbarkeit eine besondere Herausforderung dar. Bisher liegen noch keine Erkenntnisse zu systemischen Zusammenhängen zwischen den Fahrzeugeigenschaften von Gliederfahrzeugen und den Möglichkeiten zur Realisierung des Verfahrens vor. Dazu zählt die Parametrisierung des Suchalgorithmus sowie die Möglichkeiten zur Anwendung von Heuristiken und zur Realisierung eines Kartenraumes.

Das Ziel dieser Arbeit ist die Umsetzung des Verfahrens für Gliederfahrzeuge sowie die Untersuchung der genannten Zusammenhänge.

Die konkrete Aufgabenstellung lässt sich anhand folgender Teilaufgaben verdeutlichen:

1. Zunächst soll eine Funktionsbibliothek zur Modellierung von (Glieder-)Fahrzeugen in einem Kartenraum entworfen und implementiert werden. Diese soll eine einfache grafische Darstellung der Fahrzeuge und ihrer Bewegung ermöglichen. Bei der Entwicklung dieser Softwarebibliothek kann sich ggf. an der bereits vorhandenen Bibliothek *EZauto* orientiert werden, die für ähnliche Zwecke in der Arbeitsgruppe Echtzeitsysteme entwickelt wurde. Im Gegensatz zu *EZauto* soll die zu entwickelnde Bibliothek allerdings nur wesentliche, für die Arbeit notwendige Funktionalitäten enthalten. Die Funktionsbibliothek soll als Grundlage für die Implementierungen und Experimente im weiteren Verlauf der Arbeit dienen.

2. Aufbauend auf der entwickelten Funktionsbibliothek soll im nächsten Schritt das Verfahren zur Pfadplanung mittels schnell wachsender Suchbäume implementiert werden. Des Weiteren soll auch die Modellierung von Hindernissen realisiert werden, um Umgebungen mit nur teilweise befahrbaren Bereichen zu simulieren und darin planen zu können. Dabei ist insbesondere auch eine Methode zur Kollisionserkennung zwischen modellierten Hindernissen und dem geplanten Pfad eines modellierten Fahrzeuges zu realisieren, um letztlich kollisionsfreie Pfade berechnen zu können.

3. Im Folgenden liegt der Fokus auf dem Forschungsaspekt der Arbeit. Nun soll das Verfahren zur Pfadplanung auf Gliederfahrzeuge übertragen werden. Hierfür sind folgende grundlegende Fragestellungen relevant:

   - **Diskretisierung:** Sowohl hinsichtlich des Suchraums als auch das Fahrzeug betreffend ist eine Diskretisierung der Zustände erforderlich. Die Wahl einer geeigneten Diskretisierung muss mit Blick auf Präzision und Aufwand des Verfahrens erfolgen.

     Aus der Wahl der Zustandsdiskretisierung leitet sich die Frage der Zustandsähnlichkeit ab: Welche Fahrzeugzustände können als ähnlich angenommen werden, ohne dass hieraus Einbußen hinsichtlich der Fahrbarkeit

folgen?

- **Parametrisierung:** Wie ist die Anzahl der Kind-Knoten (Baumbreite) zu wählen, wie sind die generierten Knoten zu verteilen? Wie kann dies gesteuert werden?

- **Heuristiken:** Welche Heuristiken lassen sich anwenden, um möglichst wertvolle Ergebnisse bei der Pfadplanung zu erzielen? Hierzu könnte beispielsweise untersucht werden, welche Streufunktion in Abhängigkeit von Umgebung und Zustand zu wählen ist. Außerdem stellt sich z.B. die Frage, wie mit kritischen Konfigurationen umgegangen wird oder wie man schwer fahrbare Manöver vermeidet. In diesem Zusammenhang muss immer auch die Fahrbarkeit berücksichtigt werden.

4. Im Hinblick auf die im letzten Punkt aufgezeigten Fragestellungen sollen wichtige Zusammenhänge im Rahmen der Anwendung des Verfahrens auf Gliederfahrzeuge mittels geeigneter Experimente untersucht werden. Insbesondere soll die Wechselwirkung zwischen Samplingstrategie und Diskretisierung beleuchtet werden.

   Anhand eines Beispiels lassen sich zwei unterschiedliche Ansätze verdeutlichen:

   - In einer fein aufgelösten Karte können kurze Planungselemente generiert und in wenige Nachfolgekonfigurationen mit geringer Winkelveränderung überführt werden.

   - In einer grob aufgelösten Karte können lange Planungselemente generiert und in viele Nachfolgekonfigurationen mit großer Winkelveränderung überführt werden.

   Beide Ansätze erzeugen bei entsprechender Parametrisierung Bäume mit gleicher Kantenzahl, also vergleichbarem Aufwand, welche sich jedoch hinsichtlich der Wegfindung unterscheiden.

   Durch geeignete Versuche sollen erste Erkenntnisse hinsichtlich einer geeigneten

Wahl von lokalen Pfadlängen und Verzweigungsmethodik gewonnen werden.

5. Abschließend soll eine Evaluation der in den vorherigen Schritten erzielten Ergebnisse stattfinden. Hierbei soll insbesondere eine Gegenüberstellung verschiedener Parametrisierungen und Heuristiken vorgenommen werden. Die Bewertung der im Einzelnen erzielten Ergebnisse, also der resultierenden Pfade, kann beispielsweise anhand vom Aufwand der Berechnung und bestimmten Gütekriterien erfolgen. In eine solche Güte-Funktion könnten Eigenschaften wie Länge, Glattheit und Risiko des Pfades einfließen. Hierbei sind entsprechend aussagekräftige Kriterien zu wählen und sinnvoll zu gewichten.

1.3 Einordnung in den wissenschaftlichen Kontext

Der Forschungsbereich der Pfadplanung hat insbesondere in den letzten 50 Jahren an Beachtung gewonnen und unterschiedlichste Lösungsansätze hervorgebracht. In *Classic and Heuristic Approaches in Robot Motion Planning — A Chronological Review* ([Masehian u. Sedighizadeh, 2007]) werden klassische, vollständige Ansätze mit moderneren, heuristischen Ansätzen verglichen. Hierbei wird vor allem aufgezeigt, dass sich heuristische Ansätze und insbesondere probabilistische Verfahren wie die *Rapidly-exploring Random Trees* gegenüber vollständigen Ansätzen durchgesetzt haben. In *Current Issues in Sampling-Based Motion Planning* ([Lindemann u. LaValle, 2005]) liegt der Fokus auf modernen probabilistischen Verfahren, die Sampling-basiert arbeiten. Die im Rahmen dieser Arbeit betrachteten *Rapidly-exploring Random Trees* lassen sich in diese Kategorie einordnen. Das Verfahren wird in *Rapidly-exploring random trees: A new tool for path planning* ([LaValle, 1998]) sowie *Rapidly-exploring Random Trees: Progress and Prospects* ([LaValle u. Kuffner, 2000]) vorgestellt.

Im Rahmen der Arbeitsgruppe Echtzeitsysteme schließt sich diese Masterarbeit hinsichtlich der Pfadplanung mit allgemeinen Gliederfahrzeugen der bisherigen Forschung an ([Zöbel, 2013] sowie [Schwarz, 2009]). Die kinematische Beschreibung der

General-n-Trailer und ihrer Bewegung basiert dabei auf den Ausführungen in *Some properties of the general n-trailer* ([Altafini, 2001]).

Diese Masterarbeit lässt sich außerdem dem Dissertationsvorhaben von Benjamin Knopp, dem Betreuer dieser Masterarbeit, zuordnen. In seiner Dissertation betrachtet er probabilistische Pfadplanung für *General-n-Trailer* in hochdimensionalen Suchräumen. Die im Rahmen dieser Masterarbeit betrachteten *Rapidly-exploring Random Trees* stellen in diesem Zusammenhang somit einen möglichen Lösungsansatz dar. Die Ergebnisse dieser Masterarbeit sollen daher letztlich auch zur Forschung im Rahmen der genannten Dissertation beitragen.

1.4 Aufbau der Arbeit

In Kapitel 2 werden zunächst einige Grundlagen behandelt. Dazu gehören zum einen die Grundlagen der Kinematik, insbesondere im Zusammenhang mit allgemeinen Gliederfahrzeugen. Zum anderen gehört dazu der Themenbereich der Pfadplanung. Dabei wird auch das im Rahmen der Arbeit umgesetzte Verfahren vorgestellt. In Kapitel 3 wird die eigene Arbeit vorgestellt. Dabei werden zunächst grundlegende Überlegungen zur Realisierung des Pfadplanungsverfahrens *RRT* für Gliederfahrzeuge präsentiert. Darauf aufbauend wird die softwaretechnische Umsetzung des Ganzen vorgestellt. In Kapitel 4 werden Experimente vorgestellt, die zur Untersuchung der in Abschnitt 1.2 genannten Zusammenhänge durchgeführt wurden. In Kapitel 5 findet eine Auswertung der durchgeführten Experimente statt. In Kapitel 6 folgt eine abschließende Bewertung sowie ein Ausblick.

2 Stand von Wissenschaft und Technik

In diesem Kapitel findet eine Einführung in die für die Arbeit relevanten Themengebiete statt. Zunächst werden die Grundlagen der Kinematik vorgestellt. Der Fokus dabei liegt auf der Beschreibung von allgemeinen Gliederfahrzeugen. Im Anschluss wird in den Themenbereich der Pfadplanung eingeführt. Dabei werden zunächst grundlegende Begrifflichkeiten und Zusammenhänge erläutert. Anschließend wird näher auf die in der Arbeit umgesetzten Verfahren eingegangen.

2.1 Kinematik

Die Arbeitsgruppe Echtzeitsysteme verwendet zur Modellierung von allgemeinen Gliederfahrzeugen das sogenannte Einspurmodell ([Zöbel, 2013]). Des Weiteren wird auf die Differentialgleichungen von Altafini ([Altafini, 2001]) zurückgegriffen, um die Bewegung der Fahrzeuggespanne zu beschreiben. Dieser Modellierungsansatz wird auch im Rahmen dieser Masterarbeit zur Modellierung von Gliederfahrzeugen verwendet.

Die Diplomarbeit *Entwicklung eines Regelungsverfahrens zur Pfadverfolgung für ein Modellfahrzeug mit einachsigem Anhänger* ([Schwarz, 2009]) ist ebenfalls in der Arbeitsgruppe Echtzeitsysteme entstanden. Die folgenden Erläuterungen zur genannten Modellierung eines Fahrzeugzuges orientieren sich im Wesentlichen an den entsprechenden Abschnitten dieser Diplomarbeit.

2.1.1 Beschreibung von allgemeinen Gliederfahrzeugen

Das Einspurmodell eignet sich zur Beschreibung von allgemeinen Gliederfahrzeugen (*General-n-Trailern*), die sich mit niedrigen Geschwindigkeiten bewegen (siehe [Schwarz, 2009] bzw. [Zöbel, 2013]). Man beschränkt sich dabei auf ein kinematisches Modell, d.h. die bei der Bewegung wirkenden Kräfte werden vernachlässigt. Damit setzt man auch voraus, dass die Bewegung des Fahrzeuggespanns schlupffrei ist und direkt von den Steuergrößen abhängt.

Im Rahmen dieser Abstraktion beschreibt man ein Fahrzeuggespann als Kette von Fahrzeuggliedern. Hintereinander liegende Fahrzeuggliedern sind dabei jeweils durch eine Kupplung miteinander verbunden.

Als weiteren Abstraktionsschritt wird jedem Fahrzeuggelied nur genau eine Achse zugewiesen. Falls bei dem realen Fahrzeug also mehrere Achsen starr miteinander verbunden sind, werden diese durch eine virtuelle Achse ersetzt, die genau in der Mitte zwischen den ursprünglichen Achsen liegt. Des Weiteren wird die Lenkachse als eigenes Fahrzeuggelied betrachtet. Ein PKW entspricht demnach zwei Fahrzeuggliedern.

Im abschließenden Abstraktionsschritt wird davon ausgegangen, dass die Räder den Boden in genau einem Punkt berühren. Die Räder einer Achse werden dabei durch ein Rad genau in der Mitte der ursprünglichen Räder ersetzt. Die Ausrichtung des Rades entspricht dem Mittel der Ausrichtungen der ursprünglichen Räder.

Das resultierende Fahrzeug hat somit nur noch eine Spur, daher die Bezeichnung als „Einspurmodell“ oder auch „Fahrradmodell“.

In Abbildung 2.1 sind die Bezeichner für die Größen im Fahrzeug dargestellt. Die Fahrzeuggelieder werden einzeln betrachtet und jeweils in gleicher Weise beschrieben. Jedes Fahrzeuggelied erhält dabei einen Index, beginnend bei 0.

Die Beschreibung des i-ten Fahrzeuggliedes erfolgt durch folgende Werte:

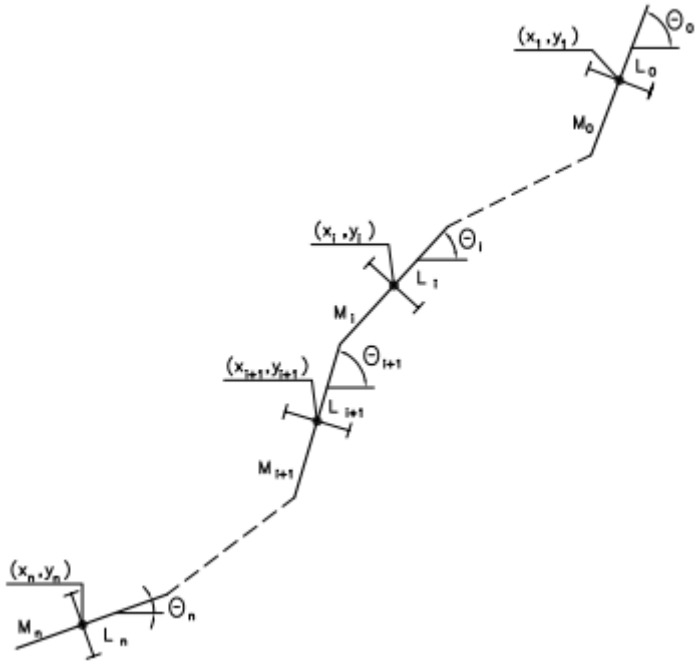

Abbildung 2.1: Bezeichner der Größen im Einspurmodell, aus [Schwarz, 2009].

- (x_i, y_i) beschreibt die **Position** des Fahrzeuggliedes bzw. seines Achsmittelpunktes.
- θ_i beschreibt die **Ausrichtung** des Fahrzeuggliedes.
- L_i beschreibt den Abstand des Achsmittelpunktes zum vorderen Kupplungsmittelpunkt. Dieser wird als **vorderer Kupplungsoffset** bezeichnet. Er ist positiv, wenn die Kupplung vor der Achse liegt, sonst negativ.
- M_i beschreibt den Abstand des Achsmittelpunktes zum hinteren Kupplungsmittelpunkt. Dieser wird als **hinterer Kupplungsoffset** bezeichnet. Er ist positiv, wenn die Kupplung hinter der Achse liegt, sonst negativ.

Ein solches Modell mit n Fahrzeuggliedern bezeichnet man als *General-*$(n-1)$*-Trailer*.

Wenn der hintere Kupplungspunkt eines Fahrzeuggliedes i mit der Achse zusammen

fällt, ist $M_i = 0$. Falls dies für alle Fahrzeuggelieder $i < n$ erfüllt ist, spricht man von einem *Standard*-$(n-1)$-*Trailer*.

Der Fahrzeugzustand bzw. die Lage des Fahrzeuges wird durch die aktuelle Konfiguration beschrieben. Dazu gehören beispielsweise die Position des ersten Fahrzeuggliedes, der Lenkwinkel sowie die Ausrichtungen aller Fahrzeuggelieder. Es ist üblich, die Konfiguration aus der Position und Ausrichtung des 1. Fahrzeuggliedes (Hinterachse Zugfahrzeug), dem Lenkwinkel und den Einknickwinkeln der dahinter liegenden Fahrzeuggelieder zu bilden. Daraus lassen sich die Positionen und Ausrichtungen aller Fahrzeuggelieder berechnen.

Lenk- und Einknickwinkel sind folgermaßen definiert:

- $\alpha_{L_1} := \theta_0 - \theta_1$ beschreibt den **Lenkwinkel** des Fahrzeugs.
- $\Delta\theta_{i(i+1)} := \theta_{i+1} - \theta_i$ beschreibt den i-ten **Einknickwinkel** des Fahrzeugs, also den Einknickwinkel zwischen i-tem und $(i+1)$-tem Fahrzeuggelied.

2.1.2 Beschreibung der Bewegung von allgemeinen Gliederfahrzeugen

Aufbauend auf der im vorherigen Abschnitt vorgestellten Fahrzeugbeschreibung lässt sich die Bewegung allgemeiner Gliederfahrzeuge beschreiben. Dazu wird zusätzlich zu den bereits eingeführten Bezeichnern ein weiterer Bezeichner eingeführt (siehe [Schwarz, 2009]):

- v_i stellt die Geschwindigkeit des i-ten Fahrzeuggliedes in Richtung θ_i dar. v_i ist positiv, wenn sich das Fahrzeuggelied in Richtung seiner Ausrichtung vorwärts bewegt, und entsprechend negativ, wenn es sich rückwärts bewegt.

Die Bewegung der einzelnen Fahrzeuggelieder lässt sich durch die bereits angesprochenen Differentialgleichungen darstellen (siehe [Schwarz, 2009] bzw. [Altafini, 2001]):

$$\dot{\theta}_{i+1} = \frac{1}{L_{i+1}}\Big(v_i \cdot \sin(\theta_i - \theta_{i+1}) - M_i \cdot \cos(\theta_i - \theta_{i+1}) \cdot \dot{\theta}_i\Big) \tag{2.1}$$

$$v_{i+1} = v_i \cdot \cos(\theta_i - \theta_{i+1}) + M_i \cdot \sin(\theta_i - \theta_{i+1}) \cdot \dot{\theta}_i \quad (2.2)$$

$$\dot{x}_i = v_i \cdot \cos(\theta_i) \quad (2.3)$$

$$\dot{y}_i = v_i \cdot \sin(\theta_i) \quad (2.4)$$

Als Anfangswerte lassen sich v_0 und $\dot{\theta}_0$ bei gegebenem $\dot{\alpha}_{L_1}$ und v_1 folgendermaßen berechnen:

$$v_0 = \frac{M_0\Big(v_1 \cdot \cos(\alpha_{L_1}) - L_1 \cdot \dot{\alpha}_{L_1} \cdot \sin(\alpha_{L_1})\Big) + L_1 \cdot v_1}{L_1 \cdot \cos(\alpha_{L_1}) + M_0} \quad (2.5)$$

$$\dot{\theta}_0 = \frac{v_1 \cdot \sin(\alpha_{L_1}) - L_1 \cdot \dot{\alpha}_{L_1} \cdot \cos(\alpha_{L_1})}{L_1 \cdot \cos(\alpha_{L_1}) + M_0} \quad (2.6)$$

Aus diesen Anfangswerten lassen sich schließlich mit den Gleichungen 2.1 - 2.4 die Änderungen von Position und Ausrichtung rekursiv für jedes folgende Fahrzeugglied berechnen.

Somit lässt sich die Änderung der Konfiguration des Fahrzeuggespanns berechnen, wenn die ursprüngliche Konfiguration sowie der Lenkwinkel α_{L_1} und die Geschwindigkeit v_1 des Zugfahrzeuges gegeben sind. Lenkwinkel und Geschwindigkeit stellen dabei die Steuergrößen dar.

2.2 Pfadplanung

Der Begriff der Pfadplanung lässt sich je nach wissenschaftlichem Kontext auf verschiedene Weise definieren. Im Rahmen dieser Arbeit versteht man darunter die Suche nach einem Pfad für ein Fahrzeug von einem Ausgangszustand zu einem Zielgebiet. Das Zielgebiet wird dabei durch eine Zustandsmenge ausgedrückt. Der Pfad umfasst zum Beispiel eine Folge von Bewegungen. Bei der Pfadplanung möchte man sowohl die Fahrbarkeit des geplanten Pfades, als auch die Kollisionsfreiheit während der geplanten Bewegung sicherstellen. Unter der Fahrbarkeit versteht man die Einhaltung bestimmter Randbedingungen, die den Fahrzeugzustand und die Fahrzeugbewegung

betreffen. Dazu gehört beispielsweise, dass festgelegte Grenzwerte für die Einknickwinkel nicht überschritten werden oder dass bestimmte Lenkwinkeländerungen unzulässig sind. Unter Kollisionsfreiheit versteht man die Vermeidung von Kollisionen mit Hindernissen.

Der Forschungsbereich der Pfadplanung hat insbesondere in den letzten 50 Jahren an Beachtung gewonnen und unterschiedlichste Lösungsansätze hervorgebracht. Je nach Betrachtungsweise werden die Ansätze in der Literatur auf verschiedene Weise kategorisiert und bewertet.

Die Verfahren lassen sich unter anderem dadurch unterscheiden, ob sie vollständig sind oder nicht. Vollständige Verfahren wie A^* betrachten jeden möglichen (diskreten) Zustand. Dies garantiert, dass, falls eine Lösung existiert, diese auch gefunden wird. Das Problem der Pfadplanung ist NP-vollständig. Ein entscheidender Nachteil vollständiger Verfahren ist daher die Komplexität. Die im Rahmen dieser Arbeit betrachteten *General-n-Trailer* zeichnen sich durch viele Freiheitsgrade aus. Ihr Zustand setzt sich aus Position und Ausrichtung des Zugfahrzeugs, dem Lenkwinkel und den Einknickwinkeln zwischen allen Fahrzeuggliedern zusammen (siehe Abschnitt 2.1.1). Zudem haben sie als einzige Steuergröße genau einen Lenkwinkel. Aus der Kombination dieser beiden Eigenschaften resultiert eine eingeschränkte Steuerbarkeit. Insbesondere in realistischen Szenarien mit einer hohen Dichte an Hindernissen und Engpässen im freien Raum ist damit die Menge an Folgezuständen, die das Fahrzeug erreichen kann, überschaubar. Dadurch wird das Finden einer Lösung deutlich erschwert. Aus diesen Gründen brauchen vollständige Verfahren viel zu lange, um eine Lösung zu finden.

Deutlich schnellere Ergebnisse erzielt man durch die Verwendung von Heuristiken. Auch aus diesem Grund haben heuristische Ansätze und insbesondere probabilistische Verfahren in den letzten 20 bis 30 Jahren zunehmend an Bedeutung gewonnen. Mit diesen Verfahren ist es möglich, auch hochdimensionale Suchräume effizient zu explorieren.

Probabilistische Verfahren sind nicht vollständig, d.h. es werden nicht alle möglichen Zustände betrachtet. Stattdessen werden Zufallswerte (z.B. Fahrzeugzustände oder Bewegungen), sogenannte Sampling-Werte erzeugt. Diese können durch verschiedene Sampling-Strategien oder Heuristiken beeinflusst werden. Das Ziel dabei ist es, Sampling-Werte zu erzeugen, die zu guten Lösungen führen. Mit diesem unvollständigen Ansatz ist nicht garantiert, dass, falls eine Lösung existiert, diese auch gefunden wird. Einige probabilistische Verfahren sind allerdings probabilistisch vollständig, das bedeutet, dass für alle gültigen Zustände gilt: Die Wahrscheinlichkeit, dass dieser Zustand betrachtet wird, ist größer als 0. Damit wird eine Lösung — falls diese existiert — bei hinreichend langer Suche auch gefunden.

Zu den probabilistischen Verfahren, die sich in den letzten Jahren durchgesetzt haben, zählt sowohl das *Probabilistic Roadmap*-Verfahren (*PRM*) als auch der im Rahmen dieser Arbeit verwendete *Rapidly-exploring Random Tree* (*RRT*) (siehe [Masehian u. Sedighizadeh, 2007] sowie [Lindemann u. LaValle, 2005]).

Die beiden Verfahren unterscheiden sich durch zwei grundsätzliche Vorgehensweisen hinsichtlich der Berechnung eines Planungsschrittes. Ein Planungsschritt bezeichnet den Vorgang innerhalb der Suche, durch den auf Basis der bisher gefundenen Zustände ein Folgezustand erzeugt wird. *PRM* basiert darauf, Verbindungen zwischen jeweils zwei ähnlichen Zuständen herzustellen. Es gehört damit zu den Verfahren, die aus dem aktuellen Zustand und dem Folgezustand die dazugehörige Bewegung berechnen. Mathematisch betrachtet wird also Folgendes berechnet:

> Sei X der Zustandsraum, $x \in X$ der aktuelle Zustand und $x' \in X$ der Folgezustand. Außerdem sei U die Menge der möglichen Bewegungen des Fahrzeuges und $u \in U$ die geplante Bewegung, die vom aktuellen zum Folgezustand führt. Dann berechnet sich die Bewegung u folgendermaßen:

$$u = \tilde{f}(x, x') \tag{2.7}$$

Diese Berechnung bezeichnet man auch als *Connection Problem* (vgl. [LaValle, 1998]). Verfahren wie *PRM* sind darauf angewiesen dieses Problem effizient zu lösen. Für Fahrzeuge mit einfacher Kinematik ist dies möglich. Für *General-n-Trailer* allerdings würde sich dabei ein Gleichungssystem mit Differentialgleichungen ergeben, das nicht (effizient) lösbar ist. Daher sind Verfahren wie *PRM*, die nach diesem Ansatz vorgehen, für *General-n-Trailer* nicht geeignet.

Der *RRT* dagegen gehört zu den Verfahren, die aus dem aktuellen Zustand des Fahrzeuges und einer festgelegten Bewegung einen Folgezustand berechnen. Mathematisch ausgedrückt wird also Folgendes berechnet:

$$\dot{x} = f(x, u) \tag{2.8}$$

Es handelt sich dabei um die sogenannte *state transition equation* ([LaValle, 1998]). Der Folgezustand x' lässt sich anschließend direkt aus x und $\dot{x}$ berechnen. Diese Berechnung ist selbst für Fahrzeuge mit komplexer Kinematik möglich. Somit ist das *RRT*-Verfahren aus kinematischer Sicht für *General-n-Trailer* geeignet.

Ein weiteres Kriterium, an dem die Unterscheidung der Verfahren ausgemacht werden kann, ist die Realisierung der Kollisionsfreiheit. Dabei spielt die Repräsentation von Hindernissen eine wesentliche Rolle. Es gibt auf der einen Seite Verfahren, die Hindernisse explizit modellieren und diese schon bei der Planung berücksichtigen. Dies kann beispielsweise so realisiert sein, dass ausschließlich im (Hindernis-)freien Raum X_{free}[1] gesucht wird. Dabei werden also nur Fahrzeugzustände betrachtet, die kollisionsfrei sind. Auf der anderen Seite gibt es Verfahren, die keine explizite Repräsentation der Hindernisse und damit auch keine vollständigen Informationen über den Zustands– bzw. Suchraum benötigen. Diese Verfahren führen zunächst einen Planungsschritt im Suchraum aus, und überprüfen erst anschließend, ob dieser kollisionsfrei ist. Dazu werden Algorithmen zur Kollisionserkennung verwendet. Diese sind inzwischen äu-

[1] Wenn der Konfigurationsraum den Zustandsraum darstellt, bezeichnet man üblicherweise X als C und X_{free} als C_{free} (siehe Abschnitt 2.2.1)

ßerst effizient. Außerdem stellt das Verfahren zur Kollisionserkennung im Rahmen der Pfadplanung eine eigenständige Komponente dar und ist somit austauschbar. Zu den Verfahren, die diesen Ansatz verwenden, gehört auch der *RRT*.

Die bisher betrachteten Kriterien legen nahe, dass der *RRT* sich bestens zur Pfadplanung eines *General-n-Trailers* eignet. Es handelt sich um ein probabilistisches Verfahren. Damit sind eine effiziente Suche sowie schnelle Ergebnisse möglich. Da das Verfahren probabilistisch vollständig ist, wird eine Lösung gefunden, falls eine existiert. Im Gegensatz zu anderen Verfahren zeichnet sich der *RRT* zudem dadurch aus, dass der Suchraum sehr schnell und gleichmäßig exploriert wird.

Der *RRT* wurde in unterschiedlichsten Varianten realisiert. Hier ist insbesondere eine Erweiterung zu nennen, die als *RRT** bezeichnet wird. Er erweitert den *RRT* folgendermaßen: Beim *RRT* ergibt sich in jedem Planungsschritt aus einem Ausgangszustand und einer Bewegung ein Folgezustand. Die Erweiterung des *RRT** überprüft daraufhin für den Folgezustand, ob eine Verbindung zu einem anderen in der Nähe befindlichen potentiellen Ausgangszustand zu einem besseren Ergebnis (z.B. kürzeren Pfad) führen würde. Dafür müsste man allerdings die dazugehörige Bewegung berechnen. Es müsste also das angesprochene *Connection Problem* (effizient) gelöst werden. Wie bereits erläutert ist dies für *General-n-Trailer* nicht möglich. Daher kann der *RRT** nicht verwendet werden.

Eine weitere Variante des *RRTs* ist der *Anytime RRT*. Dieser realisiert eine Online-Planung. Dabei bewegt sich das Fahrzeug, während gleichzeitig geplant wird. Es ändert sich also permanent der Startzustand. Zustände, die bereits erreicht und wieder verlassen wurden, können verworfen werden. Dieses Vorgehen hat letztlich aber nur den Vorteil eines geringeren Speicherbedarfs. Im Rahmen dieser Arbeit kann daher darauf verzichtet werden.

2.2.1 Rapidly-exploring Random Tree (RRT)

In *Rapidly-exploring Random Trees: A New Tool for Path Planning* ([LaValle, 1998]) sowie *Rapidly-exploring Random Trees: Progress and Prospects* ([LaValle u. Kuffner, 2000]) stellen die Autoren die *RRTs* vor. Die folgenden Erläuterungen orientieren sich größtenteils an den entsprechenden Ausführungen in diesen Forschungsberichten.

Beim *Rapidly-exploring Random Tree* (*RRT*) handelt es sich um ein probabilistisches Planungsverfahren, das inkrementell einen Suchbaum konstruiert. Das Verfahren exploriert den Suchraum sehr schnell und gleichmäßig. Die Verteilung der Knoten wird dabei durch die Verteilung der Sampling-Werte bestimmt. Durch eine entsprechende Sampling-Strategie können hiermit also schnell gute Lösungen gefunden werden. Das Verfahren ist zudem probabilistisch vollständig, d.h. für jeden Zustand ist die Wahrscheinlichkeit, als Sampling-Wert ausgewählt zu werden, größer als 0. Der erzeugte Suchbaum, den man ebenfalls als *Rapidly-exploring Random Tree* bezeichnet, bleibt immer zusammenhängend. Gleichzeitig bleibt die Anzahl der Kanten minimal.

Der *RRT* ist auf nicht-holonome Systeme ausgelegt. Es wird also nicht vorausgesetzt, dass das Fahrzeug aus seinem aktuellen Zustand zu einem beliebig gewählten Zustand gesteuert werden kann. Oder anders ausgedrückt: Es wird nicht vorausgesetzt, dass zu einem Start- und Zielzustand eine passende Bewegung berechnet werden kann. Im Gegensatz zu anderen Verfahren wie *PRM* muss der *RRT* also nicht das *Connection Problem* lösen (siehe Abschnitt 2.2). Aus kinematischer Sicht lässt sich das Verfahren daher problemlos auf *General-n-Trailer* anwenden.

Die Kollisionserkennung stellt einen entscheidenden Flaschenhals in der Pfadplanung dar. Der *RRT* ist komplett auf inkrementelle Kollisionserkennung ausgelegt. Nach jeder geplanten Bewegung wird nur genau für diese Bewegung überprüft, ob sie kollisionsfrei ist. Durch diesen Ansatz können sehr schnelle Algorithmen zur Kollisionserkennung verwendet werden.

Das grundlegende Vorgehen des *RRT* ist sehr allgemein und einfach formuliert. Daher kann das Verfahren als Pfadplanungsmodul in verschiedenen Planungssystemen eingesetzt werden.

Grundlegende Idee

Beim *RRT* wird ausgehend von einem initialen Zustand inkrementell ein Suchbaum aufgebaut. Der Baum wird erweitert, indem Steuervorgaben — also Vorgaben für jede Steuergröße des Fahrzeugs — über kurze Zeitintervalle auf das System angewendet werden. Im Zusammenhang mit *General-n-Trailern* handelt es sich bei den Steuergrößen um einen Lenkwinkel und eine Geschwindigkeit. Diese Steuervorgaben beschreiben Bewegungen, mit denen neue Zustände erreicht werden. Jeder Knoten im Baum repräsentiert dabei einen Zustand, jede Kante eine Steuervorgabe, mit der ein Zustand von einem vorherigen Zustand erreicht wurde. Wenn das Zielgebiet erreicht wurde, stellt der entsprechende Pfad im Baum die Trajektorie vom Start- zum Zielzustand dar.

Die Pfadplanung mit dem *RRT* besteht aus folgenden Komponenten:

1. **Zustandsraum** X.
2. **Startzustand** $x_{init} \in X$ und **Zielzustand** $x_{goal} \in X$ oder **Zielgebiet** $X_{goal} \subset X$.
3. **Kollisionserkennung**: Funktion $D : X \rightarrow \{true, false\}$, die angibt ob ein Zustand $x \in X$ kollisionsfrei ist.
4. **Steuervorgaben**: Menge U, enthält Steuervorgaben oder Aktionen, die sich auf den aktuellen Zustand auswirken.
5. **Inkrementeller Kinematik-Simulator**: Berechnet den resultierenden Zustand $x(t+\Delta t)$ zu einem gegebenen Zustand $x(t)$ und Steuervorgaben $\{u(t')|t \leq t' < t + \Delta t\}$.

6. **Distanz-Metrik**: Funktion $\rho : X \times X \rightarrow [0, \inf[$, die die Distanz zwischen zwei Punkten in X spezifiziert.

Basierend auf diesen Komponenten beschreibt die Pfadplanung mit dem *RRT* eine Suche im Zustandsraum X nach einem zusammenhängenden Pfad vom Startzustand x_{init} in das Zielgebiet X_{goal} oder zum Zielzustand x_{goal}. Die Suche ist sowohl durch lokale, differentiale Randbedingungen als auch durch globale Randbedingungen eingeschränkt.

Lokale, differentiale Randbedingungen beschreiben, wie sich das Fahrzeug bewegen kann, also welche Zustandsübergänge möglich sind. Der inkrementelle Kinematik-Simulator erzeugt einen neuen Zustand x_{new} für eine gegebene Steuervorgabe $u \in U$. Die Berechnung erfolgt durch eine vom Fahrzeug abhängige Funktion f, die den Zustandsübergang über die bereits erwähnte *state transition equation* (Gleichung 2.8) berechnet. Dadurch bestimmt der Kinematik-Simulator zusammen mit der Menge U an Steuervorgaben die möglichen Zustandsübergänge.

Globale Randbedingungen werden durch Hindernisse definiert. Die Kollisionserkennung stellt fest, ob ein gegebener Zustand $x \in X$ diese Randbedingungen erfüllt, d.h. ob x kollisionsfrei ist. Als X_{free} bezeichnet man den (Hindernis-)freien Raum, also alle Zustände, die die globalen Randbedingungen erfüllen.

Bezogen auf *General-n-Trailer* stellt der Konfigurationsraum C des Gliederfahrzeugs den Zustandsraum X dar. Die Konfiguration C eines *General-n-Trailers* besteht aus Position und Ausrichtung des Zugfahrzeugs, sowie dem Lenkwinkel und den Einknickwinkeln zwischen allen Fahrzeuggliedern (siehe Abschnitt 2.1.1). Damit stellt C_{free} den (Hindernis-)freien Raum X_{free} dar, q_{init} entspricht x_{init}, und q_{goal} entspricht x_{goal}.

Die Distanzmetrik gibt Aufschluss darüber, wie ähnlich oder wie nah sich zwei Punkte im Zustandsraum sind. Dies spielt im Algorithmus des *RRT* eine wesentliche Rolle (siehe folgender Abschnitt).

Algorithmus

Der grundlegende Algorithmus des *RRTs* ist in Abbildung 2.2 dargestellt.

```
BUILD_RRT(x_init)
 1  T.init(x_init);
 2  for k = 1 to K do
 3      x_rand ← RANDOM_STATE();
 4      EXTEND(T, x_rand);
 5  Return T

EXTEND(T, x)
 1  x_near ← NEAREST_NEIGHBOR(x, T);
 2  if NEW_STATE(x, x_near, x_new, u_new) then
 3      T.add_vertex(x_new);
 4      T.add_edge(x_near, x_new, u_new);
 5      if x_new = x then
 6          Return Reached;
 7      else
 8          Return Advanced;
 9  Return Trapped;
```

Abbildung 2.2: Der grundlegende Algorithmus des *RRTs*, aus [LaValle u. Kuffner, 2000].

Der Algorithmus konstruiert einen *RRT* τ mit K Knoten und dem Startzustand $x_{init} \in X_{free}$ folgendermaßen: Der Baum τ wird zunächst initialisiert, indem ein erster Knoten x_{init} erzeugt wird (Abb. 2.2, `BUILD_RRT` Zeile 1). Anschließend wird der Baum iterativ erweitert, bis die Anzahl der Knoten K erreicht ist (Abb. 2.2, `BUILD_RRT` Zeile 2) oder ein Pfad zum Ziel gefunden wurde. Bei der Erweiterung wird in jedem Schritt zunächst ein Sampling-Wert x_{rand} erzeugt (Abb. 2.2, `BUILD_RRT` Zeile 3). In Abhängigkeit von diesem Zufallswert wird der Baum durch eine `EXTEND`-Operation erweitert (Abb. 2.2, `BUILD_RRT` Zeile 4, sowie `EXTEND`). Der Sampling-Wert stellt im Rahmen dieser Erweiterungsmethode einen Zielzustand dar. Die `EXTEND`-Operation ist in Abbildung 2.3 veranschaulicht.

Die `EXTEND`-Operation erhält den *RRT* τ sowie einen (temporären) Zielzustand x. Die Methode arbeitet folgendermaßen: Zunächst wird mittels der Distanz-Metrik ρ derjenige Knoten im Baum ermittelt, der sich am nächsten zum Zielzustand x befindet (Abb. 2.2, `EXTEND` Zeile 1). Dieser Knoten wird als x_{near} bezeichnet (siehe

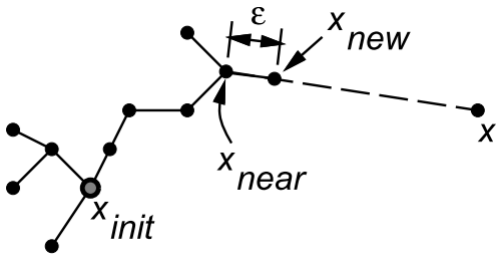

Abbildung 2.3: Die `EXTEND`-Operation, aus [LaValle u. Kuffner, 2000].

dazu auch Abbildung 2.3). Anschließend wird überprüft ob eine Bewegung von x_{near} in Richtung des Zielzustands x möglich ist. Dies geschieht mittels Aufruf der Methode `NEW_STATE` (Abb. 2.2, `EXTEND` Zeile 2).

Die Methode `NEW_STATE` erhält als Parameter die Werte x und x_{near}. Sie erzeugt eine Bewegung vom Ausgangszustand x_{near} in Richtung x, indem eine Steuervorgabe $u_{new} \in U$ für ein bestimmtes Zeitintervall Δt angewendet wird. Die Steuervorgabe kann dabei auf beliebige Weise gewählt werden. Neben einer zufälligen Wahl gibt es zum Beispiel auch die Möglichkeit, alle Steuervorgaben in U zu betrachten und als Ergebnis das $u \in U$ zu wählen, welches den am nächsten zu x liegenden Folgezustand erzeugt. Die letztlich gewählte Bewegung u_{new} vom Ausgangszustand x_{near} führt zum Folgezustand x_{new}. Nach der Berechnung dieses Folgezustands wird überprüft, ob alle Zustände zwischen x_{near} und x_{new} kollisionsfrei sind. Falls dies nicht der Fall ist, kann der Vorgang mit anderen Steuervorgaben wiederholt werden.

Falls die Methode `NEW_STATE` erfolgreich war, wird der Baum um die neuen Werte u_{new} und x_{new} erweitert. Der resultierende Zustand x_{new} wird als neuer Knoten angehängt (Abb. 2.2, `EXTEND` Zeile 3). Außerdem wird eine Kante von x_{near} nach x_{new} ergänzt, die die dazugehörige Bewegung u_{new} repräsentiert (Abb. 2.2, `EXTEND` Zeile 4). Falls bei der Bewegung x (näherungsweise) erreicht wurde, d.h. $x_{new} \approx x$, gibt die Methode `Reached` zurück, ansonsten `Advanced` (Abb. 2.2, `EXTEND` Zeilen 5–8).

Falls die Methode `NEW_STATE` nicht erfolgreich war, gibt sie `Trapped` zurück (Abb.

2.2, `EXTEND` Zeile 9). In diesem Fall konnte keine Steuervorgabe gefunden werden, die in einer kollisionsfreien Bewegung resultiert.

3 Eigene Arbeit

In den folgenden Abschnitten wird im Wesentlichen die im Rahmen dieser Masterarbeit entwickelte Software zur Anwendung des *RRT* auf *General-n-Trailer* vorgestellt. In diesem Zusammenhang wird auf grundlegende Überlegungen zur Realisierung verschiedener Komponenten des *RRT* eingegangen. Darauf aufbauend werden die Entwürfe und Implementierungen zur softwaretechnischen Umsetzung des Ganzen vorgestellt.

Die Software soll im Rahmen dieser Arbeit dazu dienen, die in den Formulierungen der Ziele (Abschnitt 1.2) beschriebenen Experimente durchzuführen. Zu diesem Zweck ist als grundlegende Komponente eine Funktionsbibliothek zur Modellierung von (Glieder-)Fahrzeugen zu realisieren (Abschnitt 1.2, Punkt 1). Darauf aufbauend ist das Verfahren zur Pfadplanung mittels schnell wachsender Suchbäume zu implementieren. Dazu gehört auch die Modellierung von Hindernissen und die Realisierung der Kollisionserkennung (Abschnitt 1.2, Punkt 2). Die einzelnen Komponenten sollen letztlich in einem Tool mit einfacher grafischer Darstellung zusammengeführt werden. Mit diesem Tool sollen dann die genannten Experimente durchgeführt werden können (Abschnitt 1.2, Punkte 4 und 5).

Im Hinblick auf die genannten Ziele wurden zunächst grundlegende Ideen zur Realisierung der einzelnen Komponenten entworfen und bewertet. In diesem Zusammenhang ist insbesondere die Komponente zur Modellierung von Hindernisse und zur Realisierung der Kollisionserkennung zu nennen. Nach der Festlegung auf bestimmte Ansätze zur Realisierung wurden nacheinander die einzelnen Komponenten implementiert.

Abschließend wurden die Komponenten in einem Tool zusammengeführt.

Die Software wurde unter der Linux-Distribution *Ubuntu*[1] in der Entwicklungsumgebung *Eclipse*[2] mit der Programmiersprache *C++* entwickelt. Zur effizienten Implementierung und Verwendung von Vektoren und Matrizen wurde auf die *C++*-Template-Bibliothek *eigen*[3] zurückgegriffen. Zur grafischen Darstellung wurde das *OpenGL Utility Toolkit*[4] (*GLUT*) verwendet. Zur Realisierung paralleler Berechnung wurde die *C++*-Bibliothek *boost*[5] verwendet. Zudem wurde die Software als *CMake*[6]-Projekt angelegt. Die Entscheidung für diese Komponenten zur Entwicklung wurde zum einen im Hinblick auf die in der Arbeitsgruppe Echtzeitsysteme üblichen Vorgehensweisen bei der Entwicklung neuer Software sowie die bereits bestehende Softwareprojektstruktur getroffen. Zum anderen kann sie mit dem Ziel einer plattformunabhängigen und einfach zu erweiternden Funktionsbibliothek begründet werden.

3.1 Gesamtkonzept

Wie bereits angedeutet, besteht die entwickelte Software aus mehreren Komponenten, die aufeinander aufbauen. Die grundlegende Komponente, auf der alle anderen aufbauen, ist eine Datenstruktur zur Modellierung von (Glieder-)Fahrzeugen. Diese beinhaltet sowohl statische als auch dynamische Eigenschaften des Fahrzeugs. Zu den statischen Eigenschaften gehören neben den Abmessungen zum Beispiel auch Beschränkungen von Lenk– und Einknickwinkeln. Die dynamischen Eigenschaften werden im Wesentlichen durch die Kinematik bestimmt. Diese legt fest, wie sich das Fahrzeug bewegt. Dazu gehört auch die aktuelle Konfiguration, also die Beschreibung des aktuellen Zustands des Fahrzeugs. Dadurch wird gleichzeitig der Karten– bzw. Zustandsraum realisiert. Eine ausführliche Beschreibung der Fahrzeugdatenstruktur

[1] `http://www.ubuntu.com`, Lizenz: Open Source (GNU General Public License u.a.)
[2] `http://www.eclipse.org`, Lizenz: Eclipse Public License
[3] `http://eigen.tuxfamily.org`, Lizenz: Mozilla Public License Version 2.0
[4] `http://www.opengl.org/resources/libraries/glut/`, Lizenz: Proprietär
[5] `http://www.boost.org`, Lizenz: Boost Software License
[6] `http://www.cmake.org`, Lizenz: BSD-Lizenz

findet in Abschnitt 3.2 statt.

Die zweite wichtige Komponente stellt eine Datenstruktur zur Darstellung der belegten Fläche durch das Fahrzeug oder durch Hindernisse dar. Es handelt sich dabei um einen Raster-basierten Ansatz. Sowohl das Fahrzeug als auch die Karte bzw. die Hindernisse werden dabei durch eine (boolsche) Matrix repräsentiert, die durch ihre Belegung angibt, ob die entsprechende Fläche belegt oder frei ist. Diese Komponente ist wesentlicher Bestandteil der Kollisionserkennung. In diesem Zusammenhang ist auch zu erwähnen, dass die Karte aus einer *PPM*-Datei eingelesen werden kann. Eine ausführliche Beschreibung der Komponente findet in Abschnitt 3.3 statt.

Die dritte Komponente umfasst die Realisierung des *RRTs*. Dazu gehören unter anderem Komponenten zur Realisierung des Suchbaumes, des Samplings, der Distanz-Metrik sowie der Auswahl einer Steuervorgabe in einem Planungsschritt. Eine ausführliche Beschreibung der Komponente findet in Abschnitt 3.4 statt.

Als weitere Komponente ist die grafische Darstellung zu nennen. Diese umfasst sowohl die Darstellung der Karte mit den Hindernissen und dem Fahrzeug als auch die Darstellung des Suchbaumes. Eine ausführliche Beschreibung der Komponente findet in Abschnitt 3.5 statt.

Aufbauend auf den genannten Komponenten wurde ein Tool entwickelt, mit dem Experimente durchgeführt werden können. Das Tool ermöglicht das Durchführen der Pfadplanung mit verschiedenen Gliederfahrzeugen auf verschiedenen Karten mit verschiedenen Parametrisierungen des Verfahrens. Es handelt sich dabei lediglich um eine erste Lösung. Es wird empfohlen, diese im Rahmen zukünftiger Implementierungen umzustrukturieren und mit weiteren Funktionalitäten auszustatten. Eine ausführliche Beschreibung des Tools findet in Abschnitt 3.6 statt.

Insgesamt wurde bei der Entwicklung auf einen modularen Aufbau der einzelnen Komponenten und ihrer Bestandteile geachtet. Zum Beispiel wurden häufig abstrakte Basisklassen verwendet. Die abgeleiteten Klassen sind somit leicht austauschbar. Gleichzeitig ist die gesamte Funktionsbibliothek damit relativ einfach erweiterbar.

Damit soll es letztlich möglich sein, ohne viel Aufwand z.B. zusätzliche Strategien und Heuristiken bezüglich der Pfadplanung zu implementieren. Außerdem soll beispielsweise die Fahrzeugdatenstruktur auch im Rahmen anderer Anwendungen Einsatz finden können.

Des Weiteren wurden Maßnahmen ergriffen, um den Berechnungsaufwand an verschiedenen Stellen zu reduzieren und somit eine möglichst schnelle Planung zu realisieren. Dazu wurde zum Beispiel die Pfadplanung auf Parallelisierung ausgelegt, sodass mehrere Threads gleichzeitig den Suchbaum erweitern. Eine entscheidende Herausforderung dabei war die Synchronisierung der Threads. Außerdem finden Vorberechnungen für die Kollisionserkennung statt. Damit konnte die Anzahl der Kollisionsüberprüfungen, die in einem bestimmten Zeitraum möglich sind, deutlich gesteigert werden. Näheres dazu ist an den entsprechenden Stellen in den folgenden Abschnitten zu finden.

3.2 Fahrzeugdatenstruktur

Die Fahrzeugdatenstruktur wird im Wesentlichen durch die Klassen `Vehicle`, `VehicleUnit` und `Coupling` realisiert (siehe Abbildung 3.1).

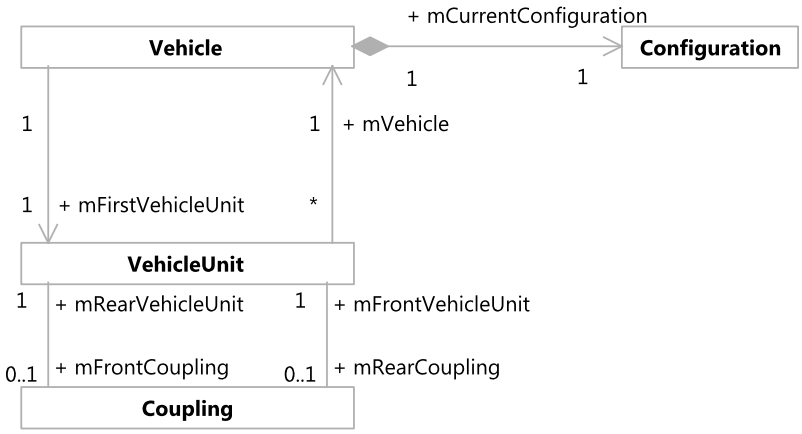

Abbildung 3.1: Zusammensetzung der Fahrzeugdatenstruktur.

Die Klasse `VehicleUnit` repräsentiert dabei ein einzelnes Fahrzeuggelied. Die Klasse `Coupling` stellt die hintere Kupplung eines Fahrzeuggliedes und damit die Verknüpfung zum nächsten Fahrzeuggelied dar. Durch entsprechende Zeiger sind die Fahrzeugglieder und Kupplungen miteinander verknüpft. Dies entspricht dem Kompositum-Entwurfsmuster. Die beiden Klassen haben primär den Zweck, die Zusammensetzung des Gliederfahrzeugs darzustellen und die Abmessungen strukturiert zu verwalten. An dieser Stelle sei darauf hingewiesen, dass das Zugfahrzeug durch **eine** `VehicleUnit` repräsentiert wird[7].

Die Klasse `Vehicle` repräsentiert dagegen das gesamte Fahrzeug und dient als Schnittstelle nach außen. Durch einen Zeiger auf das erste Fahrzeuggelied (`mFirstVehicleUnit`) hat sie indirekt Zugriff auf alle Fahrzeugglieder und Kupplungen. Sie stellt Methoden bereit, um einzelne Fahrzeugglieder und Kupplungen zu erzeugen und abzufragen. Des Weiteren beinhaltet die Klasse `Vehicle` alle kinematischen Aspekte der Fahrzeugdatenstruktur. Dazu gehört unter anderem die aktuelle Konfiguration (`mCurrentConfiguration`) des gesamten Fahrzeugzuges.

Die Fahrzeugdatenstruktur enthält außerdem Attribute und Methoden, die im Rahmen der Kollisionserkennung benötigt werden. In den folgenden Abschnitten werden diese kurz genannt. Weitergehende Erläuterungen zur Kollisionserkennung finden in Abschnitt 3.3 statt.

3.2.1 Geometrische Zusammensetzung

Die geometrische Zusammensetzung des Fahrzeuges wird durch die Klassen `VehicleUnit` und `Coupling` beschrieben. Sie ist sowohl für die Kinematik als auch für die Kollisionserkennung von Bedeutung. Zudem greift die grafische Darstellung auf die Abmessungen der Fahrzeugglieder zurück. Die beiden Klassen sind in Abbildung 3.2 dargestellt.

[7] Bei der Berechnung der Bewegung des Gliederfahrzeuges werden die beiden Achsen des Zugfahrzeuges als zwei hintereinander liegende Fahrzeugglieder betrachtet (s. Abschnitt 2.1.1)

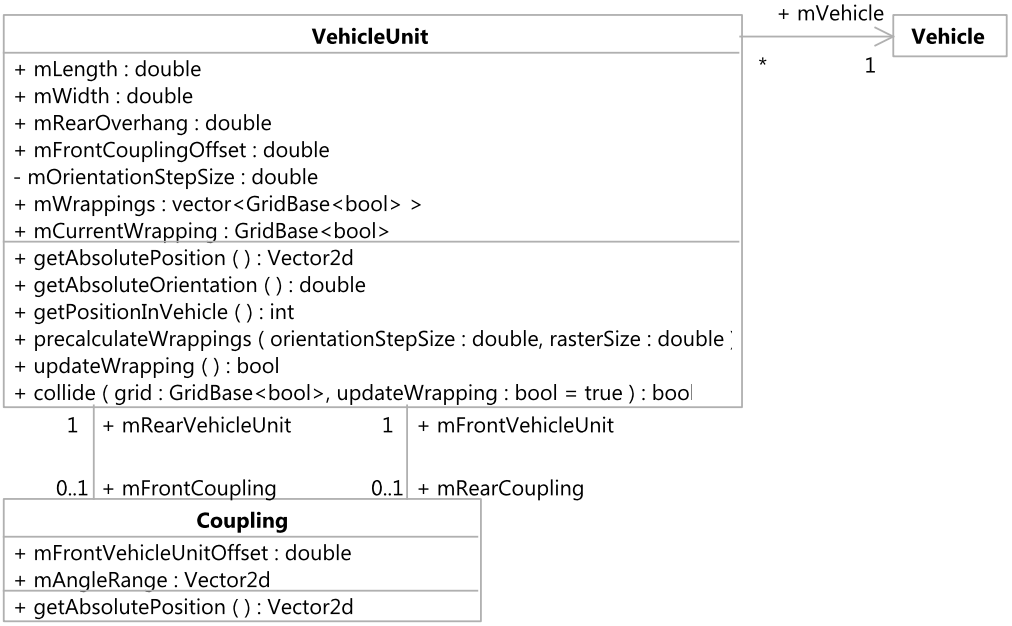

Abbildung 3.2: Die Klassen `VehicleUnit` und `Coupling`.

Die Abmessungen eines Fahrzeuggliedes werden durch folgende Attribute der Klasse `VehicleUnit` repräsentiert:

- `mLength`: Die Länge des Fahrzeuggliedes inklusive Deichsel (bei Anhängern).
- `mWidth`: Die Breite des Fahrzeuggliedes.
- `mRearOverhang`: Der Abstand von der Hinterachse zum Heck des Fahrzeuggliedes.
- `mFrontCouplingOffset`: Der vordere Kupplungsoffset (L_i) des Fahrzeuggliedes, also der Abstand der Hinterachse zur vorderen Kupplung (bei Anhängern) bzw. zur Lenkachse (beim Zugfahrzeug).

Die Klasse `Coupling` beinhaltet die relative Position einer Kupplung:

- `mFrontVehicleUnitOffset`: Der Abstand der Kupplung zur Hinterachse des (vorderen) Fahrzeuggliedes, zu dem die Kupplung gehört. Entspricht dem ne-

gativen Wert des hinteren Kupplungsoffsets (M_i).

3.2.2 Konfiguration

Die Konfiguration des gesamten Fahrzeugs wird durch die Klasse `Configuration` beschrieben. Die Klasse ist in Abbildung 3.3 dargestellt.

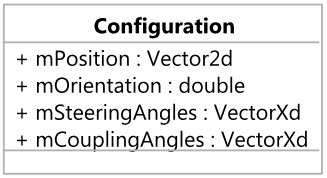

Abbildung 3.3: Die Klasse `Configuration`.

Die Konfiguration setzt sich zusammen aus der Position (`mPosition`), der Ausrichtung (`mOrientation`), dem Lenkwinkel bzw. den Lenkwinkeln (`mSteeringAngles`) und den Einknickwinkeln (`mCouplingAngles`). Als Datentypen werden hierbei die in der Template-Bibliothek *eigen* definierten Vektoren verwendet. Die Position ist zweidimensional (`Vector2d`). Lenk- und Einknickwinkel sind in ihrer Dimension dynamisch definiert (`VectorXd`), sodass erst zur Laufzeit feststehen muss, wie viele jeweiligen Werte es gibt. Beim Einknickwinkel richtet sich die Dimension nach der Anzahl der Fahrzeuggelieder bzw. Kupplungen. Beim Lenkwinkel wird im Rahmen dieser Arbeit davon ausgegangen, dass es genau einen gibt. Die Entscheidung für einen Vektor mit dynamischer Länge wurde im Hinblick auf die Verwendung der Fahrzeugdatenstruktur für beliebige andere Fahrzeugtypen getroffen. Die Implementierungen hinsichtlich der Kinematik folgen allerdings der Beschränkung auf einen Lenkwinkel.

Die Position und Ausrichtung der einzelnen Fahrzeuggelieder und Kupplungen können in den entsprechenden Klassen über die Methoden `getAbsolutePosition` und `getAbsoluteOrientation` (siehe Abbildung 3.2) abgefragt werden. Die Methoden berechnen die Werte rekursiv über die weiter vorne befindlichen Fahrzeuggelieder und

Kupplungen. Das erste Fahrzeugglied erhält die Werte aus der aktuellen Konfiguration des Fahrzeugs (`mVehicle`).

3.2.3 Kinematik

Eine Bewegung des Fahrzeugs wird durch entsprechende Steuervorgaben beschrieben. Diese werden durch die Klasse `Control` realisiert. Die Klasse ist in Abbildung 3.4 dargestellt.

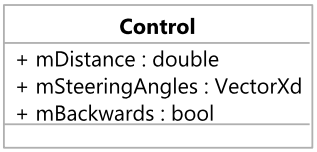

Abbildung 3.4: Die Klasse `Control`.

Eine `Control` setzt sich aus folgenden Bestandteilen zusammen:

- `mDistance`: Die Distanz, die das Fahrzeug zurücklegen soll.
- `mSteeringAngles`: Der Lenkwinkel (bzw. die Lenkwinkel für entsprechende Fahrzeuge, s. Abschnitt 3.2.2), der für diese Bewegung verwendet werden soll.
- `mBackwards`: Gibt an, ob das Fahrzeug vorwärts oder rückwärts fahren soll.

Damit sind alle Informationen gegeben, die eine Bewegung beschreiben.

Die Berechnung der Bewegung findet in der Klasse `Vehicle` statt. Die Klasse ist in Abbildung 3.5 dargestellt.

Die Methode `isValid` überprüft, ob eine gegebene Konfiguration die festgelegten Beschränkungen von Lenk– und Einknickwinkeln einhält. Die Beschränkung des Lenkwinkel ist durch das Attribut `mSteeringAngleRange` definiert. Die Beschränkungen der Einknickwinkel werden jeweils in der Klasse `Coupling` durch das Attribut `mAngleRange` festgelegt (siehe Abbildung 3.2).

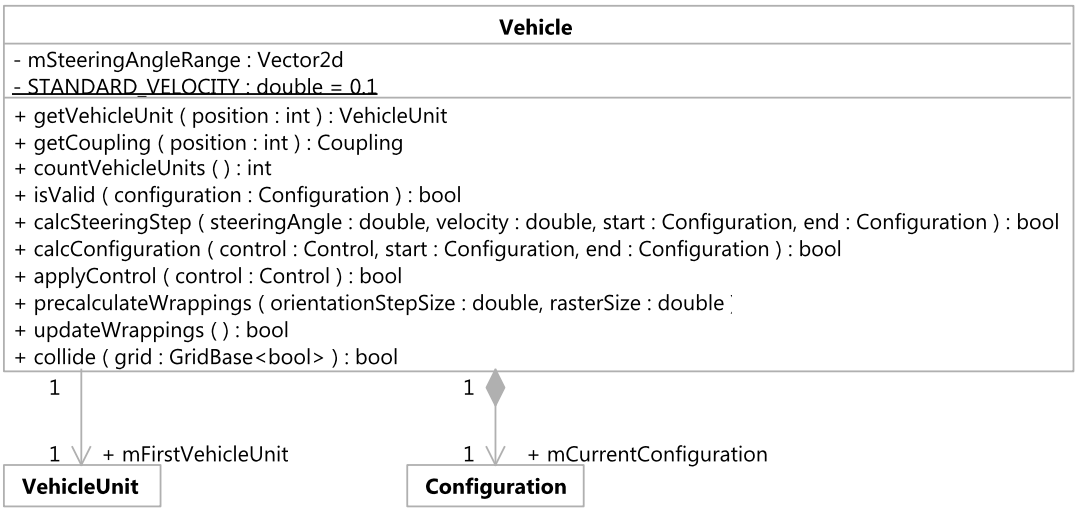

Abbildung 3.5: Die Klasse `Vehicle`.

Die Berechnung der Bewegung findet in der Methode `calcSteeringStep` statt. Die Methode führt die in Abschnitt 2.1.2 beschriebene Berechnung durch. Als Eingabeparameter werden der Lenkwinkel, eine Geschwindigkeit und die Ausgangskonfiguration erwartet. Die übergebene Geschwindigkeit entspricht der Distanz, die das Zugfahrzeug mit dieser Bewegung zurücklegt. Außerdem wird eine Adresse für die resultierende Konfiguration übergeben. Die Methode schreibt das Ergebnis (bei erfolgreicher Berechnung) auf diese Adresse. Abschließend wird überprüft, ob die resultierende Konfiguration für das gegebene Fahrzeug gültig ist (Methode `isValid`). Diese Überprüfung bestimmt den booleschen Rückgabewert. Die Methode nimmt keine Änderung am Zustand des Fahrzeugs vor. Die aktuelle Fahrzeugkonfiguration (`mCurrentConfiguration`) bleibt unverändert. Die Methode ist dafür vorgesehen, innerhalb der Klasse `Vehicle` von anderen Methoden aufgerufen zu werden. Von außen sollten die Methoden `calcConfiguration` und `applyControl` verwendet werden, da diese eine `Control` entgegen nehmen.

Die Methode `calcConfiguration` errechnet zu einer gegebenen Ausgangskonfiguration und einer Steuervorgabe die resultierende Konfiguration. Zur Berechnung wird iterativ die Methode `calcSteeringStep` aufgerufen. Als Geschwindigkeit (bzw. Di-

stanz) wird jeweils die in der Klasse `Vehicle` festgelegte `STANDARD_VELOCITY` (siehe Abbildung 3.5) verwendet. Die Aufrufe werden mit der jeweils erreichten Konfiguration wiederholt, bis die in der Steuervorgabe vorgegebene Distanz erreicht ist. Falls eine zwischenzeitlich erreichte Konfiguration nicht gültig ist (Methode `isValid`), bricht die Methode ab und gibt `false` zurück. Auch hier bleibt die aktuelle Fahrzeugkonfiguration (`mCurrentConfiguration`) unverändert.

Zur Anwendung einer Steuervorgabe auf das Fahrzeug kann die Methode `applyControl` aufgerufen werden. Diese ruft die Methode `calcConfiguration` mit der aktuellen Konfiguration (`mCurrentConfiguration`) als Ausgangskonfiguration auf. Falls die Anwendung der übergebenen Steuervorgabe erfolgreich war, wird die resultierende Konfiguration übernommen.

Bezug nehmend auf die in Abschnitt 2.2.1 vorgestellten Komponenten des *RRT* handelt es sich hierbei um den iterativen Kinematik-Simulator.

In diesem Zusammenhang sei noch darauf hingewiesen, dass es sich beim Ergebnis der iterativen Berechnung nur um eine Näherung handelt. Die Genauigkeit wird direkt durch die Wahl der Geschwindigkeit bzw. Distanz bei jeder Iteration (`STANDARD_VELOCITY`) beeinflusst. Je kleiner der Wert gewählt ist, desto genauer wird die Berechnung. Gleichzeitig steigt mit kleineren Werten aber auch der Berechnungsaufwand. Die Festlegung auf den Wert 0.1 wird im Rahmen dieser Arbeit als hinreichend genau angesehen.

3.2.4 Weitere Funktionalitäten

Die Fahrzeugdatenstruktur beinhaltet zusätzlich Funktionalitäten, die im Rahmen der Kollisionserkennung eine Rolle spielen. In der Klasse `VehicleUnit` gehören dazu die Attribute `mWrappings` sowie `mCurrentWrapping` (siehe Abbildung 3.2). Es handelt sich dabei um Datenstrukturen, die die durch das Fahrzeug belegte Fläche modellieren. Außerdem gehören dazu die Methoden `precalculateWrappings`,

`updateWrapping` bzw. `updateWrappings` sowie `collide` in den Klassen `VehicleUnit` und `Vehicle` (siehe Abbildungen 3.2 und 3.5). Die Funktionalitäten dieser Attribute und Methoden werden im Rahmen der Kollisionserkennung in Abschnitt 3.3 ausführlich behandelt.

3.3 Kollisionserkennung

Die Realisierung der Kollisionserkennung im Rahmen dieser Masterarbeit basiert darauf, dass die vom Fahrzeug und von den Hindernissen belegten Flächen modelliert werden. Eine Überschneidung der belegten Flächen zweier Objekte (Fahrzeugglied oder Hindernis) entspricht dabei einer Kollision der beiden Objekte.

Zur Umsetzung wurde ein Raster-basierter Ansatz gewählt. Dabei wird zunächst das Koordinatensystem in x- und y-Richtung gerastert (siehe Abbildung 3.6 links). Die Rastergröße legt dabei fest, wie fein oder grob die belegten Flächen modelliert werden. Im Beispiel, das in Abbildung 3.6 rechts dargestellt ist, bilden die gelb eingefärbten Felder im Raster die belegte Fläche des dargestellten Objektes.

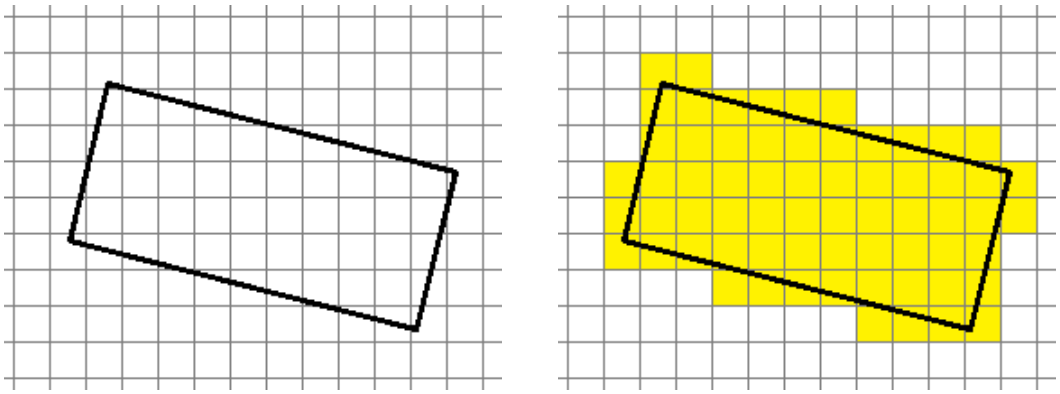

Abbildung 3.6: Rasterung des Koordinatensystems (links) und belegte Fläche eines Objektes (rechts).

Basierend auf dem Raster wird die belegte Fläche eines Objektes durch eine boolesche Matrix repräsentiert (siehe Abbildung 3.7). Dazu ist die Matrix am Raster und somit auch an den Achsen des Koordinatensystems ausgerichtet. Sie repräsentiert

eine rechteckige Fläche, die genau den Bereich im Raster abdeckt, der vom jeweiligen Objekt belegt wird. Das rote Rechteck in Abbildung 3.7 links zeigt den Bereich, der von der Matrix abgedeckt wird. Die entsprechend Matrix ist in Abbildung 3.7 rechts dargestellt. Jedes Feld der Matrix repräsentiert dabei genau ein Feld im Raster. Die Belegung (`true` oder `false` bzw. 1 oder 0) gibt an, ob das Feld durch das Objekt belegt ist oder nicht. Durch bitweise Durchschnittsbildung zweier Belegungsmatrizen kann sehr einfach und effizient überprüft werden, ob die belegten Flächen sich überschneiden.

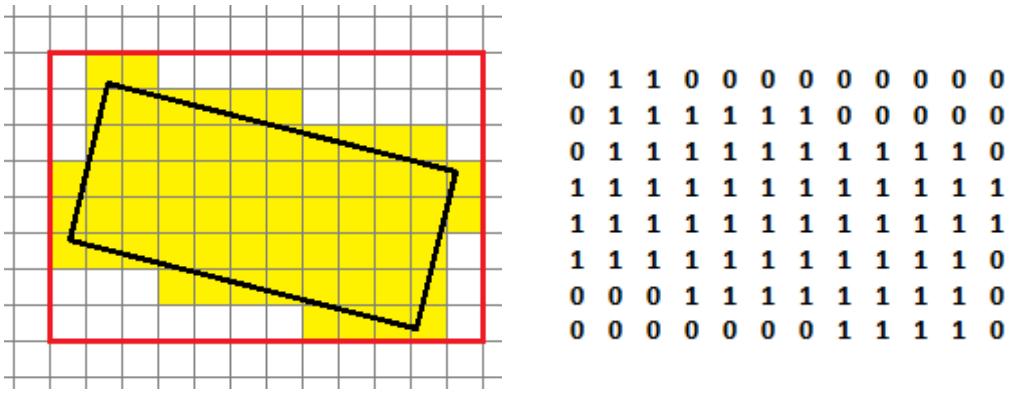

Abbildung 3.7: Von der Belegungsmatrix abgedeckter Bereich im Raster (rotes Rechteck, links) und entsprechende Matrix (rechts).

3.3.1 Implementierung der Belegungsmatrix

Die im vorherigen Abschnitt beschriebenen Belegungsmatrizen für Fahrzeuggliederund Hindernisse werden durch die Klasse `GridBase` realisiert. Die Klasse ist mit ihren wichtigsten Attributen und Methoden in Abbildung 3.8 dargestellt.

Die Klasse `GridBase` wurde als Template-Klasse implementiert. Der Template-Typ wurde `FieldType` genannt. Dieser legt fest, welchen Typ die Belegungsmatrix (Attribut `mArea`) zur Laufzeit hat. Explizit spezialisiert wurde die Klasse für `bool` und `double`. Semantisch betrachtet ist die boolesche Belegungsmatrix ausreichend, um festzulegen, ob ein Feld belegt oder frei ist. Möchte man mehr Informationen für jedes

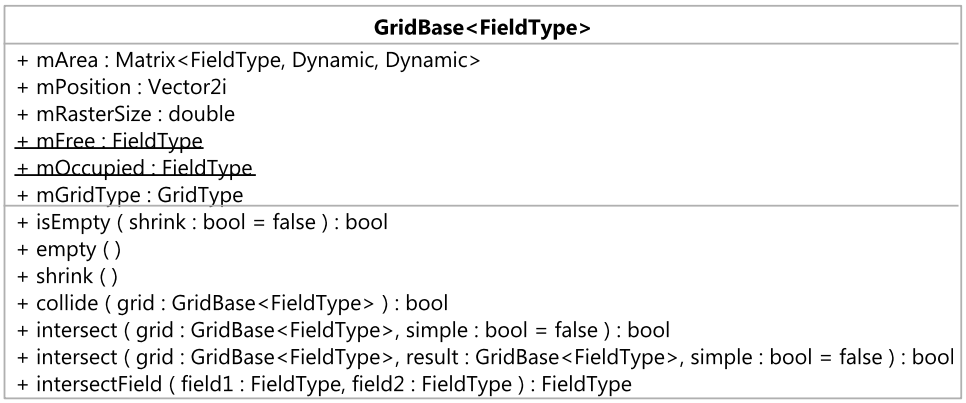

Abbildung 3.8: Die Klasse `GridBase`.

Feld speichern, kann man andere Datentypen verwendet. Das ist unter anderem dann sinnvoll, wenn man zwischen verschiedenartigen Hindernissen unterscheiden möchte. Zum Beispiel möchte man einen Bürgersteig nur in Ausnahmefällen befahren, auch wenn dies grundsätzlich unproblematisch ist. Es macht also Sinn, zwischen der belegten Fläche eines Bürgersteigs und der eines Gebäudes zu unterscheiden. Hierzu ist es möglich, als Template-Typ der Belegungsmatrix Datentypen wie `double` oder `char` zu verwendet, um unterschiedliche „Belegungstypen“ auszudrücken. Im Rahmen der Masterarbeit wurde ausschließlich mit `bool` bzw. mit `GridBase<bool>` gearbeitet. Damit hat man den geringsten Speicherverbrauch für die Belegungsmatrizen und die schnellste Durchschnittsbildung zweier Matrizen.

Das bereits angesprochene Attribut `mArea` der Klasse `GridBase` stellt die Belegungsmatrix dar. Es ist als Matrix der Template-Bibliothek *eigen* realisiert. Der Datentyp entspricht — wie bereits erklärt — dem Template-Typ der Klasse (`FieldType`). Die Matrix ist in ihrer Größe dynamisch definiert, sodass erst zur Laufzeit feststehen muss, wie groß sie ist.

Zur Realisierung der Rasterung gehören außerdem die Attribute `mPosition` sowie `mRasterSize` (siehe Abbildung 3.8). `mRasterSize` stellt die Rastergröße dar. Der Wert legt somit auch fest, welche Fläche sowohl ein einzelnes Feld der Belegungsma-

trix (`mArea`) als auch die komplette Matrix abdecken. `mPosition` gibt die Position der Belegungsmatrix im Raster an. Das Attribut ist als `int`-Vector (`Vector2i`) der Template-Bibliothek *eigen* implementiert.

Als weiteres Attribut hat die Klasse einen `mGridType` vom Datentyp `GridType`. Es handelt sich dabei um eine Aufzählung, die in der Klasse `GridBase` selbst definiert ist. Der Datentyp soll aussagen, welche Art von Objekt repräsentiert wird. Unterschieden wird zwischen folgenden Typen:

- `Unspecified`: Standard-Wert für beliebige Objekte.
- `Vehicle`: Sagt aus, dass es sich um die belegte Fläche eines Fahrzeugs bzw. Fahrzeugglieds handelt.
- `Obstacle`: Sagt aus, dass es sich um die belegte Fläche eines Hindernisses handelt.
- `Collision`: Sagt aus, dass es sich um die Kollisions– bzw. Überschneidungsfläche zweier anderer Flächen handelt.

Im Rahmen der Masterarbeit ist dieser `GridType` lediglich im Zusammenhang mit der grafischen Darstellung (siehe Abschnitt 3.5) relevant. Er legt fest, in welcher Farbe die jeweilige Belegungsmatrix dargestellt wird.

Die statischen Attribute `mFree` und `mOccupied` vom Template-Typ `FieldType` stellen die Belegungswerte für „frei“ und „belegt“ dar. Sie ermöglichen die Implementierung aller Methoden (mit Ausnahme von `intersectField`) mit dem Template-Typ `FieldType`. Dadurch ist eine explizite Spezialisierung lediglich für diese beiden Attribute (und die genannte Methode) notwendig.

Die angesprochene Methode `intersectField` berechnet den Durchschnitt von zwei Werten des Template-Typs `FieldType`. Sie realisiert damit die anfangs angesprochene bitweise Durchschnittsbildung (siehe Abschnitt 3.3). Grundsätzlich kann die Methode mit zwei unterschiedlichen Template-Typen umgehen. Die Methode muss aber zunächst im Rahmen der expliziten Spezialisierung für den jeweiligen Typ oder

die beiden jeweiligen Typen implementiert werden.

Die Methode `intersect` realisiert die Durchschnittsbildung zweier Belegungsmatrizen. Dazu ermittelt sie anhand der Positionen und Größen der Matrizen den überlappenden Bereich und führt auf diesem Bereich die bitweise Durchschnittsbildung mit Hilfe der Methode `intersectField` durch. Der Parameter `simple` (`bool`) legt fest, ob eine aus der Durchschnittsbildung resultierende Überschneidungsmatrix erzeugt werden soll (`false`) oder lediglich festgestellt werden soll, ob eine Überschneidung vorliegt (`true`). Letztere Vorgehensweise gibt über den booleschen Rückgabewert an, ob es eine Überschneidung gibt. Zudem ist sie effizienter. Sie wird daher im Zusammenhang mit der Kollisionserkennung verwendet. Die Methode erlaubt unterschiedliche Template-Typen für die beiden Objekte der Klasse `GridBase`. Allerdings setzt sie voraus, dass die Rastergrößen der beiden `GridBase`-Objekte gleich sind.

An dieser Stelle sei noch erwähnt, dass analog zu den Methoden zur Durchschnittsbildung auch Methoden zur Vereinigung (`unify` etc.) implementiert wurden. Diese eignen sich beispielsweise dazu, die Belegungsmatrizen mehrerer Hindernisse oder mehrerer Fahrzeuggelieder zu vereinen. Im Rahmen der Masterarbeit wurden die Methoden allerdings nicht verwendet.

Die Methode `collide` realisiert letztlich die Kollisionserkennung. Sie überprüft, ob das Objekt der Klasse `GridBase` mit einem anderen Objekt der Klasse „kollidiert“. Die Methode ruft die `intersect`-Methode (mit `simple` = `true`) für die jeweiligen Matrizen auf und gibt das Ergebnis zurück. Die Methode erlaubt ebenfalls unterschiedliche Template-Typen für die beiden Objekte der Klasse `GridBase`. Allerdings setzt sie genauso voraus, dass die Rastergrößen der beiden `GridBase`-Objekte gleich sind. Falls dies für zwei gegebene Objekte nicht der Fall ist, kann die Belegungsmatrix eines der Objekte vorher angepasst werden. Entsprechende Funktionalitäten werden in Abschnitt 3.3.5 erläutert.

3.3.2 Verwendung der Belegungsmatrix in der Fahrzeugdatenstruktur

Die belegte Fläche des Gliederfahrzeugs wird in der Fahrzeugdatenstruktur verwaltet (siehe Abschnitt 3.2.4). Die Belegungsmatrizen werden in den einzelnen Fahrzeuggliedern (Klasse `VehicleUnit`) gespeichert und verwaltet. Der Zugriff auf die entsprechenden Funktionalitäten ist aber auch über die Klasse `Vehicle` möglich, da diese als Schnittstelle nach außen dienen soll (siehe Abschnitt 3.2).

Das Attribut `mCurrentWrapping` vom Typ `GridBase<bool>` in der Klasse `VehicleUnit` stellt die aktuelle Belegungsmatrix für das Fahrzeuggglied dar (siehe Abbildung 3.2). Der Vektor `mWrappings` enthält vorberechnete Belegungsmatrizen für das Fahrzeuggglied (Näheres zur Vorberechnung ist in Abschnitt 3.3.3 zu finden).

Die Methode `precalculateWrappings` in der Klasse `VehicleUnit` realisiert die angesprochene Vorberechnung. Die Methode `updateWrapping` aktualisiert die aktuelle Belegungsmatrix (`mCurrentWrapping`) in Abhängigkeit von der aktuellen Position und Ausrichtung des Fahrzeuggglieds. Dabei wird auf die vorberechneten Belegungsmatrizen im Attribut `mWrappings` zurückgegriffen.

Die Methode `collide` wird im Rahmen der Kollisionserkennung verwendet. Sie ruft für die eigene Belegungsmatrix (`mCurrentWrapping`) und die übergebene Belegungsmatrix die gleichnamige Methode der Klasse `GridBase` auf (siehe Abschnitt 3.3.1).

Die Methoden `precalculateWrappings`, `updateWrappings` sowie `collide` in der Klasse `Vehicle` führen die jeweils gleichnamige Methode der Klasse `VehicleUnit` für alle Fahrzeuggglieder aus.

3.3.3 Berechnung der Belegungsmatrizen für das Fahrzeug

Zur Berechnung der Belegungsmatrix für ein Fahrzeuggglied wurde die Operator-Klasse `VehicleUnitRasterizer` implementiert.

Die `operator()`-Methode der Klasse erzeugt zum übergebenen Fahrzeuggglied (Klasse

`VehicleUnit`) eine entsprechende Belegungsmatrix, die auf das übergebene Objekt der Klasse `GridBase` geschrieben wird. Die Rastergröße für die Belegungsmatrix wird durch das übergebene `GridBase`-Objekt festgelegt.

Die Methode bietet die Möglichkeit, eine „sichere Hülle“ um das Fahrzeugglied zu legen. Dabei wird die erzeugte Belegungsfläche größer gemacht als die vom Fahrzeugglied tatsächlich belegte Fläche. Dazu werden die Abmessungen des Fahrzeugglieds zu beiden Seiten um ein festgelegtes Vielfaches der Rastergröße verbreitert und nach vorne und hinten um dasselbe Vielfache der Rastergröße verlängert. Der Faktor, mit dem die Rastergröße dabei vervielfacht wird, wird durch den Parameter `wrappingFactor` festgelegt.

Die Methode ruft intern die private Methode `rasterizeMethodDiscreteCorners` auf. Diese Methode realisiert die Berechnung der Belegungsmatrix nach einer bestimmten Vorgehensweise. Auf eine Erklärung dieser Vorgehensweise wird an dieser Stelle verzichtet. Zu erwähnen ist jedoch, dass die von der Methode erzeugte Belegungsmatrix die tatsächlich belegte Fläche des Fahrzeugglieds in jedem Fall vollständig abdeckt.

Die `operator()`-Methode existiert in zwei Versionen. Die erste Version erzeugt die Belegungsmatrix in Abhängigkeit von der aktuellen Position und Ausrichtung des Fahrzeugglieds. Die zweite Version übernimmt die als Parameter übergebenen Werte für Position und Ausrichtung.

Vorberechnung

Die Kollisionserkennung findet im Rahmen des *RRT* in jedem Planungsschritt einmal statt. Das bedeutet, dass in jedem Planungsschritt die Belegungsmatrix für die aktuelle Position und Ausrichtung eines jeden Fahrzeuggliedes vorliegen muss. Die Berechnung einer Belegungsmatrix für ein Fahrzeugglied in einer bestimmten Position mit einer bestimmten Ausrichtung ist verhältnismäßig aufwändig. Daher wurde

eine Vorberechnung der Belegungsmatrizen für die einzelnen Fahrzeuggliedern implementiert.

Zur Vorberechnung wird zunächst der Bereich der möglichen Ausrichtungen von 0° bis 360° diskretisiert. Jede diskrete Ausrichtung repräsentiert dabei einen bestimmten Bereich realer Ausrichtungen. Diskretisiert man beispielsweise mit einer Schrittweite von 3°, so steht die diskrete Ausrichtung 0° für die realen Ausrichtungen im Intervall $[0°, 3°[$, die diskrete Ausrichtung 3° für die realen Ausrichtungen im Intervall $[3°, 6°[$ und so weiter.

Für jede diskrete Ausrichtung des Fahrzeuggliedes wird nun eine Belegungsmatrix vorberechnet. Im Rahmen der Kollisionserkennung wird die resultierende Belegungsmatrix für alle zugeordneten realen Ausrichtungen verwendet. Es wird also für jedes Fahrzeugglied diejenige vorberechnete Belegungsmatrix gewählt, die die aktuelle Ausrichtung repräsentiert.

Ausgehend von einer bestimmten Ausrichtung sind die Belegungsmatrizen für verschiedene diskrete Positionen eines Fahrzeuggliedes gleich. Es genügt also, die Belegungsmatrix für die jeweilige diskrete Ausrichtung nur für eine Position zu berechnen. Bei der Vorberechnung wird als Position jeweils der Ursprung verwendet.

Im Rahmen der Kollisionserkennung wird also zunächst eine Belegungsmatrix in Abhängigkeit von der aktuellen Ausrichtung des Fahrzeuggliedes gewählt. Anschließend wird die gewählte Belegungsmatrix in Abhängigkeit von der aktuellen Position im Raster verschoben. Dazu wird einfach nur das Attribut `mPosition` der Klasse `GridBase` angepasst. Abschließend können die Methoden zur Kollisionserkennung aufgerufen werden.

Die Vorberechnungen für verschiedene Ausrichtungen sind in Abbildung 3.9 veranschaulicht.

Die Diskretisierung der Ausrichtungen kann beliebig genau gewählt werden. Sie sollte allerdings genau genug sein, sodass die Belegungsmatrix einer diskreten Ausrichtung

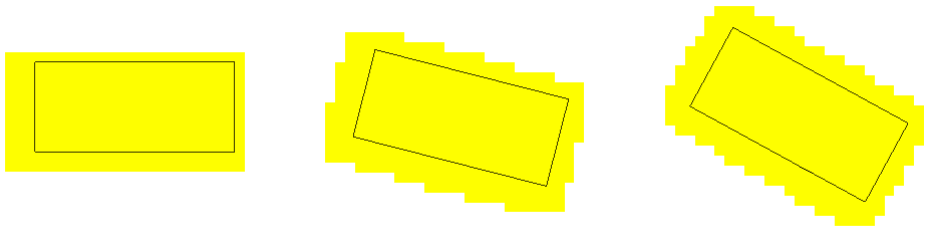

Abbildung 3.9: Belegungsmatrizen für die Ausrichtungen 0°, 15° und 30°.

die Belegungsfläche der jeweils zugeordneten tatsächlichen Ausrichtungen abdeckt. Im Rahmen dieser Masterarbeit wurde mit einer Schrittweite von 1° gearbeitet. Dieser Wert wird im Hinblick auf die verwendeten Rastergrößen als genau genug angesehen.

Implementierung der Vorberechnung

Die Vorberechnung wird von der Operator-Klasse `VehicleUnitWrappingPrecalculator` realisiert.

Die `operator()`-Methode der Klasse `VehicleUnitWrappingPrecalculator` berechnet zum übergebenen Fahrzeugglied (Klasse `VehicleUnit`) Belegungsmatrizen für mehrere Ausrichtungen und schreibt die Ergebnisse in den übergebenen Vektor des Typs `GridBase`. Der Parameter `orientationStepSize` legt die Schrittweite der Diskretisierung der Ausrichtungen fest. Der Parameter `rasterSize` bestimmt die Rastergröße der zu erzeugenden Belegungsmatrizen. Die Methode verwendet den anfangs beschriebenen `VehicleUnitRasterizer` zum Erzeugen der Belegungsmatrizen (siehe Anfang des Abschnitts 3.3.3).

Die bereits angesprochene Methode `precalculateWrappings` der Klasse `VehicleUnit` erzeugt ein Objekt der vorgestellten Klasse `VehicleUnitWrappingPrecalculator` und ruft die vorgestellte `operator()`-Methode auf. Die Parameter werden beim Aufruf weitergegeben. Als Vektor des Typs `GridBase` für das Ergebnis wird das Attribut

`mWrappings` mitgegeben. Das Attribut beinhaltet anschließend alle vorberechneten Belegungsmatrizen für das Fahrzeugglied.

3.3.4 Erzeugen einer Belegungsmatrix für die Karte

Die Belegungsmatrix der Karte soll die Hindernisse im Rahmen der Pfadplanung darstellen. Es besteht natürlich die Möglichkeit die Belegungsmatrix manuell anzulegen, indem ein entsprechendes `GridBase`-Objekt erzeugt wird und die dazugehörige Matrix manuell gefüllt wird. Dies ist allerdings mit erheblichem Aufwand verbunden. Mit der Operator-Klasse `PPMToGridTransformer` ist es deutlich einfacher möglich, die Belegungsmatrix für eine Karte zu erzeugen.

Es handelt sich beim `PPMToGridTransformer` um eine Operator-Klasse, die eine Karte aus einer *ppm*-Datei einlesen kann. Die `operator`-Methode erzeugt aus den eingelesenen Daten direkt eine Belegungsmatrix und schreibt sie in das übergebene Objekt der Klasse `GridBase`. Die Rastergröße wird durch dieses Objekt festgelegt. Jeder Pixel im *ppm*-Bild entspricht genau einem Feld in der Belegungsmatrix. Das bedeutet gleichzeitig, dass mit der Rastergröße auch die Größe eines Pixels festgelegt wird. Falls die Hindernisse und somit auch die Pixel eine bestimmte Größe haben sollen, muss die Rastergröße entsprechend gewählt werden. Eine nachträgliche Veränderung der Rastergröße — ohne Änderung der realen Größe — ist aber möglich (siehe dazu Abschnitt 3.3.5).

Weiße Pixel werden als freie Flächen interpretiert, schwarze Pixel als belegte Flächen. Die Karte kann gemäß dieser Rahmenbedingungen zum Beispiel in einem beliebigen Grafikprogramm gemalt werden. Bei einer Umwandlung einer entsprechenden Bilddatei in das *ppm*-Format muss darauf geachtet werden, dass die Daten im *ASCII*-Format gespeichert werden.

3.3.5 Nachträgliches Anpassen einer Belegungsmatrix

Ein Objekt der Klasse `GridBase` kann auf unterschiedliche Weise nachträglich angepasst werden. Zum einen ist es möglich, sowohl die Position (Attribut `mPosition`), die Rastergröße (`mRasterSize`) als auch die interne Matrix (`mArea`) nachträglich zu ändern, da die Attribute nicht privat sind. Eine solche Änderung eines Attributes wirkt sich allerdings nicht auf die jeweils anderen Attribute aus. Beispielsweise bleibt beim Verkleinern der Rastergröße die Matrix unverändert. Dadurch wird also nicht die Rasterung genauer, sondern die Belegungsfläche kleiner. Das macht semantisch wenig Sinn, da man ja in der Regel die belegte Fläche eines in der Größe unveränderten Objektes abbilden möchte.

Nachträgliches Anpassen der Rastergröße einer Belegungsmatrix

Möchte man die Rastergröße in dem Sinne verkleinern, dass die Rasterung genauer wird, kann man auf die dazu implementierte Operator-Klasse `GridRastersize-Transformer` zurückgreifen.

Die `operator()`-Methode der Klasse `GridRastersizeTransformer` überträgt ein Objekt der Klasse `GridBase` auf ein neues Objekt der Klasse, das eine andere Rastergröße hat. Die interne Matrix (`mArea`) des neuen Objekts hat entsprechend eine andere Größe. Die letztlich abgedeckte Fläche bleibt dabei gleich. Die Methode erlaubt unterschiedliche Template-Typen für die beiden Objekte der Klasse `GridBase`.

Die von der Klasse `GridRastersizeTransformer` implementierte Funktionalität kann man zum Beispiel nutzen, wenn man gemäß der in Abschnitt 3.3.4 beschriebenen Vorgehensweise eine Karte mit einer bestimmten Größe einlesen möchte, und nachträglich die Rasterung verändern möchte, ohne dass die Größe der Karte dabei verändert wird.

Nachträgliches Anpassen des Template-Typs einer Belegungsmatrix

Eine Änderung eines Objektes der Klasse `GridBase` ist auch bezüglich des Template-Typs denkbar. Eine solche Funktionalität ist in der Operator-Klasse `GridFieldType-Transformer` realisiert.

Die `operator()`-Methode der Klasse `GridFieldTypeTransformer` überträgt ein Objekt der Klasse `GridBase` auf ein neues Objekt der Klasse, das einen anderen Template-Typ besitzt. Dazu werden Rastergröße und Position einfach übernommen. Die interne Matrix (`mArea`) wird in der Weise übertragen, dass die Belegung der Felder mit `mFree` oder `mOccupied` (siehe Abschnitt 3.3.1) übernommen wird. Die Werte von `mFree` und `mOccupied` entsprechen dann den in der expliziten Spezialisierung des Template-Typs festgelegten Werten.

3.4 Realisierung des Pfadplanungsverfahrens RRT

In den beiden vorherigen Abschnitten (3.2 und 3.3) wurde die Realisierung der Fahrzeugdatenstruktur und der Kollisionserkennung vorgestellt. Darauf aufbauend wird in diesem Abschnitt die Realisierung des Pfadplanungsverfahrens basierend auf dem *Rapidly-exploring Random Tree* behandelt.

Die Realisierung des Verfahrens orientiert sich im Wesentlichen an dem in Abschnitt 2.2.1 vorgestellen Algorithmus des *RRTs*. Dazu wurden unter anderem die in Abschnitt 2.2.1 beschriebenen Komponenten realisiert, die im Algorithmus zum Einsatz kommen.

Die zentrale Komponente zur Realisierung des Verfahrens ist die Klasse `RRT`. Die Klasse realisiert primär das Verfahren selbst. Dazu bietet sie Funktionalitäten, um die Suche zu konfigurieren, zu starten, zu beenden und auszuwerten. Die Klasse erzeugt und verwaltet zudem die wichtigsten Komponenten des Verfahrens wie z.B. Sampling und Distanz-Metrik. Sie stellt damit auch die Verknüpfung zwischen den

verschiedenen Komponenten her. Weitergehend kann die Klasse auch als Container-Klasse des Suchbaums verstanden werden. Im Rahmen zukünftiger Implementierungen macht es unter Umständen Sinn, die genannten Funktionalitäten auf mehrere Klassen aufzuteilen.

Da die Klasse `RRT` verschiedenste Funktionalitäten realisiert, soll an dieser Stelle lediglich ein Überblick über die grundlegenden Attribute und Methoden gegeben werden. Im weiteren Verlauf des Kapitels wird dann auf die Funktionalitäten eingegangen, die für die jeweils vorgestellte Komponente relevant sind. Die grundlegenden Attribute und Methoden der Klasse `RRT` sind in Abbildung 3.10 dargestellt.

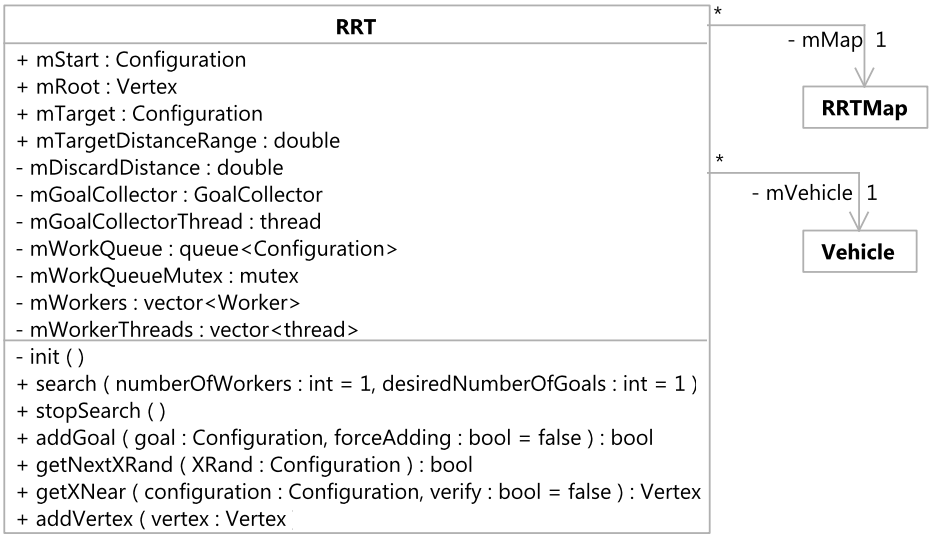

Abbildung 3.10: Die grundlegenden Attribute und Methoden der Klasse `RRT`.

Das Attribut `mVehicle` (Klasse `Vehicle`, siehe Abschnitt 3.2) zeigt auf das Fahrzeug, mit dem geplant werden soll. Das Attribut `mMap` zeigt auf die Karte (Klasse `RRTMap`, wird im Laufe dieses Abschnittes noch vorgestellt), auf der geplant werden soll.

Das Attribut `mStart` vom Typ `Configuration` gibt die Start-Konfiguration der Suche an. Das Attribut `mRoot` vom Typ `Vertex` stellt den Wurzel-Knoten des Suchbaums dar. Dieser repräsentiert die Start-Konfiguration. Er wird angelegt sobald die Su-

che gestartet wird. Über die Wurzel des Baumes hat man Zugriff auf den gesamten Suchbaum (Näheres dazu wird in Abschnitt 3.4.2 vorgestellt).

Das Attribut `mTarget` vom Typ `Configuration` gibt das Zentrum des Zielgebiets an. Bei der aktuellen Implementierung der Distanz-Metrik (wird in Abschnitt 3.4.3 vorgestellt) ist dabei aber nur die Position relevant. Das Attribut `mTargetDistanceRange` gibt den Radius des Zielgebiets an.

Das Attribut `mDiscardDistance` legt fest, ab welcher Distanz (bzw. Nähe) eines neuen Knotens zu einem bereits bestehenden Knoten im Suchbaum der neue Knoten verworfen wird. Die Idee dahinter ist es, zu vermeiden, dass sich an einer bestimmten Stelle etliche Knoten anhäufen. Daraus resultiert unter anderem auch eine schnellere Exploration. Je nach gewählter Parametrisierung des Verfahrens kann dieser Wert aber unter Umständen sogar den Aufbau des Suchbaums komplett blockieren. Der Wert sollte daher möglichst klein (ggf. sogar 0) gewählt werden. Näheres dazu ist in Abschnitt 3.4.6 zu finden.

Das Verfahrens wurde mittels paralleler Berechnung realisiert. Dabei laufen sowohl das Sampling als auch die Planung in eigenen Threads und somit parallel zur Anwendung mit der grafischen Darstellung. In diesem Zusammenhang ist es notwendig, die Threads zu synchronisieren. Dazu werden im Wesentlichen Mutex-Objekte verwendet. Die Klasse `GoalCollector` realisiert die Komponenten, die für das Erzeugen der Sampling-Werte zuständig ist. Die erzeuten Sampling-Werte werden in einer Warteschlange verwaltet (Attribut `mWorkQueue`). Das Mutex-Objekt `mWorkQueueMutex` vom Typ `boost::mutex` schützt die Warteschlange vor gleichzeitigem Zugriff durch mehrere Threads. Die Klasse `Worker` nimmt die Sampling-Werte aus der Warteschlange und realisiert die eigentliche Planung. Das System ist so ausgelegt, dass es einen `GoalCollector` (Attribut `mGoalCollector`) und einen oder mehrere `Worker` (Attribut `mWorkers`) gibt. Zu jedem Objekt der beiden Klassen gibt es einen entsprechenden Thread (`boost::thread`) (Attribute `mGoalCollectorThread` sowie `mWorkerThreads`). In dem Thread läuft jeweils die `run`-Methode der jeweiligen Klasse. Die beiden Klas-

sen `GoalCollector` und `Worker` und ihre jeweiligen Funktionalitäten werden ausführlich im Abschnitt 3.4.1 vorgestellt.

Die Methode `search` startet die Suche. Die Parameter legen die Anzahl der `Worker`-Threads sowie die maximale Anzahl der in `mWorkQueue` gleichzeitig aufbewahrten Sampling-Werte fest. Die Methode erzeugt den `GoalCollector` und die `Worker` und startet die dazugehörigen Threads.

Die Methode `stopSearch` bricht die Suche ab. Dabei werden alle Threads beendet.

Die Methode `init` wird zu Beginn der Methode `search` aufgerufen. Sie initialisiert verschiedene Komponenten. Unter anderem initialisiert sie die Distanz-Metrik. Außerdem legt sie den Wurzel-Knoten `mRoot` an.

Die Methode `addGoal` wird vom `GoalCollector` verwendet, um neue Sampling-Werte in die Warteschlange (`mWorkQueue`) zu schreiben. Die Methode realisiert die Synchronisierung durch Verwendung des Mutex-Objekts `mWorkQueueMutex`.

Die Methode `getNextXRand` liefert den nächsten Sampling-Wert aus der Warteschlange. Die Methode wird von den `Worker`-Threads verwendet. Das Ergebnis wird in das übergebene Objekte der Klasse `Configuration` geschrieben. Der Rückgabewert gibt an, ob noch Werte in der Warteschlange waren und somit ein Ergebnis gefunden wurde.

Die Methode `getXNear` ermittelt den Knoten im Suchbaum, der die geringste Distanz zur übergebenen Konfiguration hat. Die Distanzen werden dabei durch die Distanz-Metrik bestimmt. Der Parameter `verify` legt fest, ob nur solche Knoten betrachtet werden sollen, die noch über mögliche Steuervorgaben verfügen. Dies ist im Hinblick auf die Verwendung der Methode im Rahmen des Algorithmus des Verfahrens relevant (die Realisierung des Algorithmus wird in Abschnitt 3.4.1 vorgestellt, Näheres zu den Steuervorgaben ist in Abschnitt 3.4.5 zu finden). Die Methode wird in diesem Zusammenhang von der Klasse `Worker` verwendet.

Die Methode `addVertex` informiert die Klasse `RRT` darüber, dass ein neuer Knoten zum Suchbaum hinzugefügt wurde. Dabei ist hervorzuheben, dass die Methode selbst den Knoten nicht hinzufügt. Es wird davon ausgegangen, dass der Knoten bereits an den Suchbaum angefügt wurde (Knoten werden am jeweiligen Eltern-Knoten angehängt, mehr dazu in Abschnitt 3.4.2). Die Methode wird von der Klasse `Worker` aufgerufen, nachdem diese den Knoten erzeugt und an den Baum angehängt hat. Die Klasse `RRT` kann im Rahmen dieser Methode verschiedenste Funktionalitäten realisieren. In der aktuellen Implementierung wird durch diese Methode beispielsweise die Anzahl der Knoten im Suchbaum gezählt und der kürzeste Pfad vom Start zum Ziel ermittelt. Außerdem wird überprüft, ob mit dem neuen Knoten das Ziel (`mTarget` und `mTargetDistanceRange`) erreicht wurde.

Alle weiteren Funktionalitäten und dazugehörigen Attribute und Methoden der Klasse `RRT` werden in den folgenden Abschnitten gemäß ihrer Relevanz für die jeweils vorgestellte Komponente eingeführt und erklärt.

Die Möglichkeiten zur Anwendung des Verfahrens wird in Abschnitt 3.4.6 beschrieben. Dabei wird sowohl auf die Parametrisierung des Verfahrens als auch auf die Möglichkeiten zur Auswertung einer Suche eingegangen. Beides baut auf den im folgenden vorgestellten Komponenten auf. Aus diesem Grund wird das Ganze erst am Ende des Abschnittes vorgestellt.

Die Karte

Bevor auf die eigentlichen Komponenten des *RRTs* eingegangen wird, soll an dieser Stelle noch die Verwaltung der Karte vorgestellt werden.

Die Karte wird durch die Klasse `RRTMap` realisiert. Diese ist in Abbildung 3.11 dargestellt.

Die Klasse stellt eine Wrapper-Klasse für eine Belegungsmatrix (siehe Abschnitt 3.3 bzw. 3.3.1 und 3.3.4) dar. Die Belegungsmatrix ist im Attribut `mObstacles` vom Typ

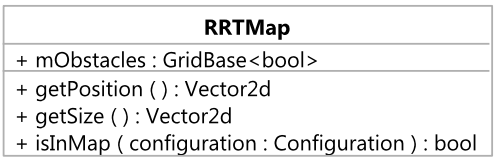

Abbildung 3.11: Die Klasse `RRTMap`.

`GridBase<bool>` gespeichert. Es repräsentiert die Hindernisse auf der Karte.

Die Methode `getPosition` liefert die Position der Karte im Koordinatensystem. Es handelt sich dabei um die Position der intern gespeicherten Belegungsmatrix. Analog dazu liefert die Methode `getSize` die Größe der Karte. Diese wird ebenfalls von der intern gespeicherten Belegungsmatrix abgefragt.

Die Methode `isInMap` gibt an, ob die übergebene Konfiguration (Klasse `Configuration`) innerhalb der Karte liegt. Dabei ist nur die Position der Konfiguration relevant. Die Methode wird zum Beispiel dazu verwendet, zu überprüfen, ob eine im Laufe der Suche erreichte Konfiguration innerhalb der Karte liegt. Ist dies nicht der Fall, wird auch kein entsprechender Knoten im Suchbaum erzeugt.

3.4.1 Realisierung des Algorithmus

Der Algorithmus des *RRTs* wurde in Abschnitt 2.2.1 vorgestellt. Die genaue Vorgehensweise wurde dabei insbesondere in Abbildung 2.2 dargestellt.

Wie am Anfang des Abschnitts 3.4 bereits angedeutet, wird der Algorithmus im Wesentlichen durch die Klassen `GoalCollector` und `Worker` realisiert. Die Klassen sind beide von `boost::thread` abgeleitet. Ihre jeweilige `run`-Methode kann somit in einem eigenen Thread ausgeführt werden.

Die Klasse `GoalCollector`

Die Klasse `GoalCollector` ist für das Sammeln der Sampling-Werte zuständig. Die Klasse ist in Abbildung 3.12 dargestellt.

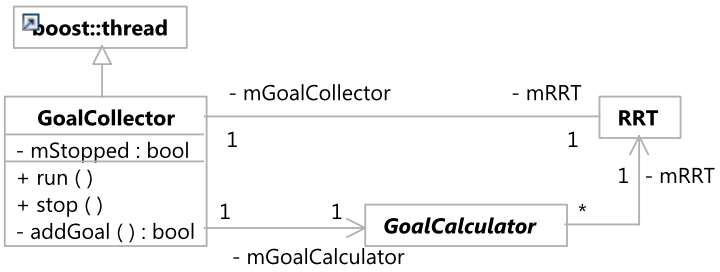

Abbildung 3.12: Die Klasse `GoalCollector`.

Der `GoalCollector` ist genau dem `RRT`-Objekt zugeordnet, von dem er erzeugt wurde. Der Zeiger `mRRT` auf das Objekt ermöglicht das Hinzufügen von Sampling-Werten über die Methode `addGoal` der Klasse `RRT`.

Die Methode `run` läuft in dem Thread, der zum jeweiligen Objekt der Klasse `GoalCollector` gehört. Die Methode erzeugt zunächst ein Objekt der Klasse `GoalCalculatorRandom`. Das Objekt wird dem Attribut `mGoalCalculator` des abstrakten Basistyps `GoalCalculator` zugewiesen. Die abstrakte Klasse `GoalCalculator` und ihre abgeleitete Klasse `GoalCalculatorRandom` realisieren das Sampling. Ihre Methode `calculateGoal` liefert jeweils einen Sampling-Wert. Das Sampling und die beiden Klassen werden ausführlich in Abschnitt 3.4.4 beschrieben.

Nachdem die Methode `run` den `GoalCalculator` erzeugt hat, ruft sie in einer Endlosschleife die private Methode `addGoal` auf. Die Methode `addGoal` ruft die Methode `calculateGoal` des `GoalCalculator`-Objektes auf. Der zurückgelieferte Sampling-Wert wird über die Methode `addGoal` der Klasse `RRT` an das dazugehörige `RRT`-Objekt weitergegeben.

Das Attribut `mStopped` gibt an, ob der Thread gestoppt werden soll. Durch die Me-

thode `stop` wird der Wert auf `true` gesetzt. Die Methode `run` weiß dadurch Bescheid und verlässt die Endlosschleife. Der Thread wird dann beendet.

Die Klasse `Worker`

Die Klasse `Worker` realisiert die eigentliche Planung, die in Abschnitt 2.2.1 als `EXTEND`-Operation beschrieben wurde (siehe insbesondere Abbildung 2.2). Die Klasse ist in Abbildung 3.13 dargestellt.

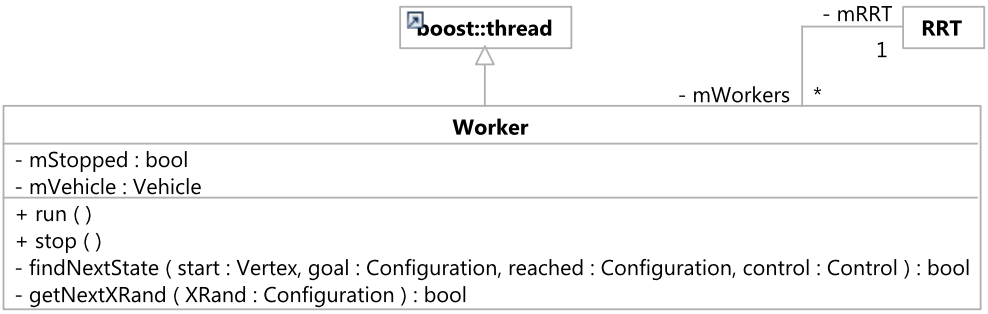

Abbildung 3.13: Die Klasse `Worker`.

Analog zum `GoalCollector` ist der `Worker` genau dem `RRT`-Objekt zugeordnet, von dem er erzeugt wurde. Durch den Zeiger `mRRT` auf das Objekt hat der `Worker` Zugriff auf die Sampling-Werte und kann auf benötigte Komponenten wie z.B. die Distanz-Metrik zugreifen. Vom `RRT` bekommt die Klasse außerdem das Fahrzeug (Attribut `mVehicle`). In der aktuellen Implementierung übernimmt die Klasse einfach den Zeiger auf das Objekt der Klasse `Vehicle`. Dies hat die Auswirkung, dass ein Arbeiten mit mehreren `Worker`-Objekten nicht möglich ist. Für zukünftige Implementierungen sollte die bereits vorhandene Methode `copyVehicle` der Klasse `Vehicle` korrigiert und im `Worker` verwendet werden, um das Fahrzeug zu kopieren und somit auf einem eigenen Fahrzeug arbeiten zu können.

Wie beim `GoalCollector` läuft die Methode `run` in dem Thread, der zum jeweiligen Objekt der Klasse `Worker` gehört. Die Methode führt in einer Endlosschleife die

Planung aus.

Analog zum `GoalCollector` gibt das Attribut `mStopped` an, ob der Thread gestoppt werden soll. Durch die Methode `stop` wird der Wert auf `true` gesetzt. Die Methode `run` weiß dadurch Bescheid und verlässt die Endlosschleife. Der Thread wird dann beendet.

Die Planung in der Methode `run` läuft folgendermaßen ab (zum Vergleich kann der *RRT*-Algorithmus in Abbildung 2.2 betrachtet werden).

1. Über die Methode `getNextXRand` wird die gleichnamige Methode des `RRT`-Objekts (`mRRT`) aufgerufen. Diese liefert den nächsten Sampling-Wert (Klasse `Configuration`), der als zwischenzeitliches Ziel dient (x_{rand} im *RRT*-Algorithmus).

2. Über die Methode `getXNear` des `RRT`-Objekts wird der Knoten (Klasse `Vertex`) im aktuellen Suchbaum ermittelt, der die geringste Distanz zum Sampling-Wert hat (x_{near} im *RRT*-Algorithmus). Dabei werden nur die Knoten berücksichtigt, die noch Steuervorgaben anbieten (mehr dazu in Abschnitt 3.4.5). Falls kein Knoten gefunden wird, wird Schritt 1 wiederholt. Bei der aktuellen Implementierung ist in diesem Fall aber davon auszugehen, dass keiner der vorhandenen Knoten noch Steuervorgaben anbietet. Die Exploration kommt somit zum Stillstand.

3. Über die eigene Methode `findNextState` wird versucht, am gewählten Knoten eine Steuervorgabe anzuwenden (`NEW_STATE`-Operation im *RRT*-Algorithmus). Falls dies nicht gelingt, wird Schritt 2 wiederholt.

4. Konnte eine Steuervorgabe angewendet werden und eine entsprechende Nachfolge-Konfiguration erzeugt werden, wird ein entsprechender neuer Knoten erzeugt und an den Baum angehängt (`add_vertex`– und `add_edge`-Operation im *RRT*-Algorithmus). Dazu wird am Eltern-Knoten die Methode `createChild` aufgerufen (Näheres zum Suchbaum in Abschnitt 3.4.2). Abschließend wird das

RRT-Objekt durch Aufruf der Methode addVertex über den neuen Knoten informiert.

Die Methode findNextState versucht, vom übergebenen Ausgangsknoten start eine Steuervorgabe anzuwenden, um das Fahrzeug in Richtung des übergebenen Ziels goal zu bewegen. Dazu greift die Methode auf den ControlOptionsOrganizer (wird in Abschnitt 3.4.5 ausführlich vorgestellt) des Knotens start zurück. Dieser verfügt über mehrere Steuervorgaben (Klasse Control) für den Knoten und liefert diese nacheinander durch Aufruf der Methode getNextControlOption.

Für die jeweils nächste Steuervorgabe führt die Methode findNextState die folgenden Schritte aus:

1. Zunächst wird die Steuervorgabe auf das Fahrzeug (mVehicle) angewendet. Dazu wird die Methode applyControl der Klasse Vehicle aufgerufen (siehe Abschnitt 3.2.3).

2. Wenn die Bewegung gültig war (Rückgabe true) wird die resultierende Konfiguration des Fahrzeugs überprüft. Die resultierende Konfiguration ist ungültig, wenn sie nicht innerhalb der Karte liegt. Zudem ist sie ungültig, wenn sie näher an einem der bisherigen Knoten des Suchbaums liegt, als es die in der Klasse RRT definierte mDiscardDistance zulässt.

3. Falls die resultierende Konfiguration gültig ist, wird anschließend die Kollisionserkennung durchgeführt. Dazu wird im Vehicle-Objekt mVehicle die Methode collide aufgerufen. Als Parameter wird die Belegungsmatrix der Karte des RRT-Objekts übergeben.

4. Falls keine Kollision auftritt, werden die Ergebnisse zurückgegeben. Dazu werden die Steuervorgabe sowie die resultierende Konfiguration in die beiden übergebenen Objekte geschrieben. Als Rückgabewert wird true zurückgegeben.

Sobald die jeweilige Überprüfung in einem der Schritte fehlschlägt, wird mit der nächsten Steuervorgabe von vorne begonnen. Falls keine Steuervorgabe angewendet

werden kann und der `ControlOptionsOrganizer` keine mehr anbietet, gibt die Methode `findNextState` `false` zurück.

3.4.2 Realisierung des Suchbaums

Der Suchbaum wird im Wesentlichen durch die Klasse `Vertex` realisiert. Die Klasse repräsentiert einen Knoten im Baum. Durch Verknüpfungen mit dem Eltern-Knoten und den Kind-Knoten wird die Baumstruktur realisiert. Die Wurzel des Baumes ist als Attribut `mRoot` in der Klasse `RRT` gespeichert. Die Klasse `RRT` stellt somit eine Container-Klasse für den Suchbaum dar. Über die Klasse `RRT` bzw. die Wurzel kann man alle Knoten im Baum erreichen. Für den Zugriff sollte man allerdings den extra dafür implementierten `RRTVertexIterator` verwenden. Im Folgenden wird zunächst die Klasse `Vertex` detailliert behandelt. Anschließend wird der `RRTVertexIterator` vorgestellt.

Die Klasse `Vertex`

Die Klasse `Vertex` repräsentiert einen Knoten im Suchbaum. Damit bildet sie einerseits die Baumstruktur, stellt andererseits aber auch ein Zwischenergebnis der Suche im Rahmen der Pfadplanung dar. Ein Objekt der Klasse repräsentiert somit unter anderem eine Konfiguration, die im Laufe der Suche erreicht wurde. Gleichzeitig beschreibt es — zusammen mit den Vorgänger-Knoten — den Pfad von der Wurzel zum jeweiligen Knoten. Die Klasse stellt dazu entsprechende Informationen und Funktionalitäten bereit.

Die wichtigsten Attribute und Methoden der Klasse `Vertex` sind in Abbildung 3.14 dargestellt.

Das Attribut `mRRT` verweist auf den Suchbaum (Klasse `RRT`), zu dem der Knoten gehört.

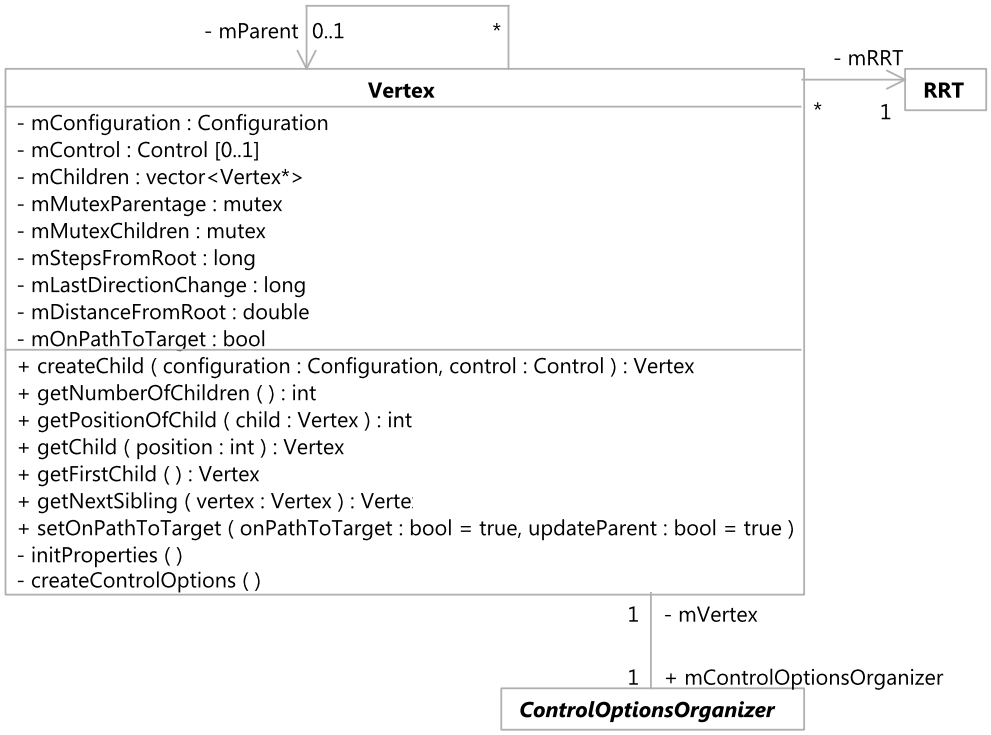

Abbildung 3.14: Die Klasse `Vertex`.

Das Attribut `mConfiguration` vom Typ `Configuration` (siehe Abschnitt 3.2.2) stellt die vom Knoten repräsentierte Konfiguration dar.

Die Herkunft des Knotens wird durch die Attribute `mParent` sowie `mControl` beschrieben. Das Attribut `mParent` zeigt auf den Eltern-Knoten (ebenfalls Klasse `Vertex`). Das Attribut `mControl` vom Typ `Control` (siehe Abschnitt 3.2.3) stellt die Steuervorgabe dar, mit der dieser Knoten vom Eltern-Knoten aus erreicht wurde. Im Hinblick auf die Synchronisierung (siehe Anfang des Abschnittes 3.4) wird der Zugriff auf die beiden Attribute durch das Mutex-Objekt `mParentage` (`boost::mutex`) geschützt.

Der Wurzel-Knoten des Suchbaumes hat weder einen Eltern-Knoten (`mParent` ist `NULL`) noch eine Steuervorgabe (`mControl` ist in diesem Fall nicht initialisiert). Die Klasse hat entsprechende Konstruktoren für normale Knoten und für Wurzel-Knoten.

Das Attribut `mChildren` vom Typ `std::vector<Vertex*>` beinhaltet alle Kind-Knoten des Knotens. Im Hinblick auf die Synchronisierung wird der Zugriff auf das Attribut durch das Mutex-Objekt `mMutexChildren` geschützt.

Die Methode `createChild` erzeugt einen neuen Kind-Knoten. Die übergebenen Parameter werden an den Konstruktoraufruf des Kind-Knotens weiter gegeben. Das Attribut `mRRT` wird ebenfalls weitergegeben. Als Eltern-Knoten wird der aktuelle Knoten übergeben. Die Methode wird von der Klasse `Worker` verwendet, um den Suchbaum zu erweitern (siehe Abschnitt 3.4.1).

Es wurden mehrere Methoden zum Zugriff auf die Kind-Knoten implementiert. Die Methoden werden hauptsächlich vom `RRTVertexIterator` verwendet, um unter Gewährleistung der Synchronisierung die Baumstruktur zu traversieren. Die Methode `getNumberOfChildren` liefert die Anzahl der Kind-Knoten zurück. Die Methode `getPositionOfChild` liefert die Position des übergebenen `Vertex`-Objektes im Vektor `mChildren` zurück. Die Methode `getChild` liefert den Kind-Knoten an der übergebenen Position im genannten Vektor. Die Methode `getFirstChild` liefert den ersten Kind-Knoten. Die Methode `getNextSibling` liefert den nächsten Geschwister-Knoten zum übergebenen Knoten.

Die Klasse `Vertex` enthält des Weiteren Eigenschaften des Knotens oder des Pfades zum Knoten, die im Hinblick auf die Pfadplanung von Interesse sind. Dazu gehören folgende Attribute:

- `mStepsFromRoot`: Der Ganzzahlwert gibt die Tiefe des Knotens und somit die Anzahl der Vorgänger-Knoten an. Der Wert ist im Hinblick auf die Höhe des Baumes relevant.
- `mLastDirectionChange`: Der Ganzzahlwert gibt die Anzahl der Vorgänger-Knoten seit der letzten Änderung der Fahrtrichtung (vorwärts oder rückwärts) an.
- `mDistanceFromRoot`: Die Fließkommazahl gibt die Länge des Pfades von der

Wurzel zum Knoten an. Der Wert ist im Hinblick auf den kürzesten Pfad relevant.

Die Attribute werden durch Aufruf der Methode `initProperties` im Konstruktor initialisiert. Die Werte werden von den jeweiligen Eigenschaften des Eltern-Knotens abgeleitet. Im Hinblick auf zukünftige Implementierungen ist es denkbar, dass weitere Eigenschaften ergänzt werden. Diese sollten dann ebenfalls in der genannten Methode initialisiert werden, damit sie unmittelbar nach dem Konstruktoraufruf zur Verfügung stehen. Dies ist im Hinblick auf die Synchronisierung wichtig.

Eine weitere Eigenschaft wird durch das Attribut `mOnPathToTarget` beschrieben. Der boolesche Wert gibt an, ob der Knoten zum kürzesten Pfad zum Ziel gehört. Der Wert wird ebenfalls in der Methode `initProperties` gesetzt, und zwar zunächst auf `false`. Über die Methode `setOnPathToTarget` kann der Wert von außen verändert werden. Dabei wird der neue Wert rekursiv an alle Vorgänger-Knoten weitergeleitet. Damit wird der gesamte Pfad von der Wurzel zum Knoten als kürzester Pfad markiert bzw. umgekehrt die Markierung wieder aufgehoben.

Die bisher vorgestellten Attribute und Methoden beschreiben die Herkunft des Knotens oder den Knoten selbst. Die Klasse `Vertex` enthält darüber hinaus Informationen, die für die weitere Suche relevant sind. Es handelt sich dabei um mögliche Steuervorgaben (Klasse `Control`, siehe Abschnitt 3.2.3), die vom Knoten aus angewendet werden können bzw. sollen. Das Attribut `mControlOptionsOrganizer` zeigt auf ein Objekt der Klasse `ControlOptionsOrganizer`, in der die Steuervorgaben verwaltet werden. Das Attribut wird durch Aufruf der Methode `createControlOptions` im Konstruktor als Objekt der abgeleiteten Klasse `ControlOptionsOrganizerPV` initialisiert. Die Klasse `Worker` greift zur Auswahl einer Steuervorgabe auf dieses Objekt zurück. Eine ausführliche Beschreibung der genannten Klasse und der Vorgehensweise findet in Abschnitt 3.4.5 statt.

Die Klasse `RRTVertexIterator`

Die Klasse `RRTVertexIterator` bietet eine einfache Möglichkeit, den Suchbaum zu traversieren. Dabei wird die Synchronisierung gewährleistet und gleichzeitig eine mögliche Erweiterung der Baumstruktur während der Traversierung berücksichtigt. Es wird nach der Tiefensuche vorgegangen.

Die Klasse `RRTVertexIterator` ist in Abbildung 3.15 dargestellt.

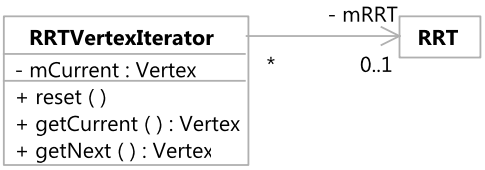

Abbildung 3.15: Die Klasse `RRTVertexIterator`.

Das Attribut `mRRT` zeigt auf den Suchbaum (Klasse `RRT`), der traversiert werden soll. Das Attribut `mCurrent` vom Typ `Vertex` zeigt auf den Knoten, der aktuell betrachtet wird. Zunächst wird der Wurzel-Knoten (Attribut `mRoot` der Klasse `RRT`) betrachtet.

Die Methode `getCurrent` liefert den aktuell betrachteten Knoten. Die Methode `getNext` liefert den nächsten Knoten im Sinne der Tiefensuche und aktualisiert `mCurrent`. Die Methode greift unter anderem auf die im vorherigen Abschnitt vorgestellten Methoden der Klasse `Vertex` zum Zugriff auf die Kind-Knoten zurück.

Die Methode `reset` setzt die Traversierung zurück. Dazu wird `mCurrent` wieder auf den Wurzel-Knoten des Baumes gesetzt.

3.4.3 Realisierung der Distanz-Metrik

Die Distanz-Metrik wurde als Komponenten des *RRTs* in Abschnitt 2.2.1 vorgestellt. Sie bestimmt die Distanz zwischen zwei Punkten im Konfigurationsraum. Im Algo-

rithmus des *RRTs* wird sie zur Berechnung des Wertes `XNear` verwendet.

Idee zur Realisierung der Distanz-Metrik

Die Distanz-Metrik ist als Funktion realisiert, die die Distanz zwischen zwei Punkten im Konfigurationsraum in folgender Weise berechnet: Ausgehend von zwei Konfigurationen c und $\tilde{c}$ der Dimension n sei $c_i, (0 \leq i < n)$ der Wert der i-ten Dimension von c und $\tilde{c}_i, (0 \leq i < n)$ der Wert der i-ten Dimension von $\tilde{c}$. Des Weiteren sei $d_i(c, \tilde{c}) = |c_i - \tilde{c}_i|$ der Betrag der Differenz zwischen c und $\tilde{c}$ in der Dimension i. Jeder Dimension der Konfiguration wird sowohl ein Faktor f_i als auch ein Exponent e_i für die Differenz zugewiesen. Die Distanz wird dann berechnet als

$$d(c, \tilde{c}) = f_0 \cdot d_0(c, \tilde{c})^{e_0} + \ldots + f_i \cdot d_i(c, \tilde{c})^{e_i} + \ldots + f_n \cdot d_n(c, \tilde{c})^{e_n} \tag{3.1}$$

Neben dem Faktor und dem Exponent kann für jede Dimension ein Minimal- und ein Maximalwert für die Differenz festgelegt werden. Falls diese Beschränkungen von der errechneten Differenz mindestens einer Dimension nicht eingehalten werden, wird die resultierende Distanz auf unendlich gesetzt ($d(c, \tilde{c}) = \infty$).

Durch geeignete Parametrisierung der Distanz-Metrik (Faktoren, Exponenten und Beschränkungen) ist es möglich, verschiedenste Metriken zu realisieren. Im Rahmen der Masterarbeit wurde eine euklidische Distanz-Metrik verwendet, die lediglich von der Position abhängt. Die Faktoren für die x- und y-Dimension sind dabei auf 1.0 gesetzt, die Exponenten auf 2.0. Als Beschränkungen werden die Randwerte der Karte (Attribut `RRTMap` der Klasse `RRT`) verwendet. Alle anderen Dimensionen werden ignoriert.

Implementierung

Die Parametrisierung für eine Dimension wird durch die Klasse `DistanceMetricFE-Params` realisiert. Sie legt fest, ob und in welchem Maße die jeweilige Dimension in die Distanz-Metrik einfließt. Die Klasse ist in Abbildung 3.16 dargestellt.

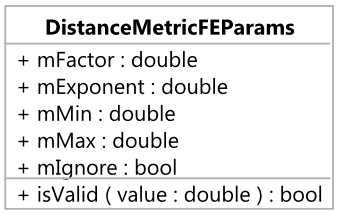

Abbildung 3.16: Die Klasse `DistanceMetricFEParams`.

Folgende Attribute der Klasse vom Typ `double` stellen die Parameter der Distanz-Metrik dar:

- `mFactor`: Der Faktor für die Differenz in dieser Dimension.
- `mExponent`: Der Exponent für die Differenz in dieser Dimension.
- `mMin`: Der Minimalwert für die Differenz in dieser Dimension.
- `mMax`: Der Maximalwert für die Differenz in dieser Dimension.

Das Attribut `mIgnore` bietet die Möglichkeit, die Beschränkungen durch Minimal- und Maximalwert zu ignorieren.

Die Methode `isValid` gibt an, ob die übergebene Differenz in der Dimension gültig ist oder nicht. Dies wird anhand der durch `mMin` und `mMax` festgelegten Beschränkungen entschieden. Falls `mIgnore` = `true` ist, werden die Beschränkungen ignoriert und als Ergebnis in jedem Fall `true` zurückgegeben.

Möchte man eine Dimension bei der Berechnung der Distanz komplett ignorieren, so sollte man zum einen den Faktor (`mFactor`) auf 0.0 setzen und zum anderen `mIgnore` auf `true` setzen. Die restlichen Parameter spielen dann keine Rolle mehr und können beliebig gewählt werden.

Die Distanz-Metrik wird schließlich durch die Klasse `DistanceMetricFE` realisiert. Die Klasse sowie ihre abstrakte Basisklasse `DistanceMetric` sind in Abbildung 3.17 dargestellt.

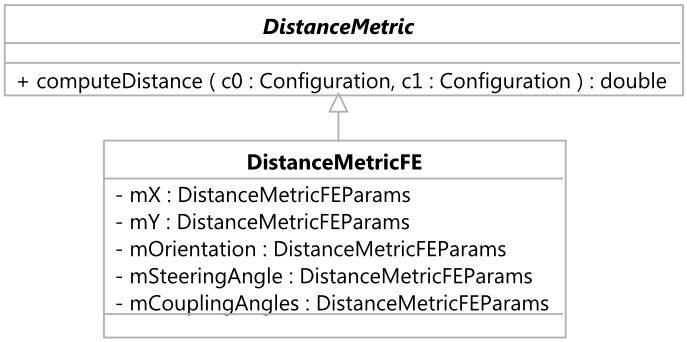

Abbildung 3.17: Die abstrakte Klasse `DistanceMetric` und die davon abgeleitete Klasse `DistanceMetricFE`.

Für jede Dimension des Konfigurationsraums besitzt die Klasse ein Attribut des Typs `DistanceMetricFEParams`, das die beschriebene Parametrisierung einer Dimension darstellt. Folgende Attribute sind vorhanden:

- `mX`: Legt den Einfluss der x-Koordinate der Position fest.
- `mY`: Legt den Einfluss der y-Koordinate der Position fest.
- `mOrientation`: Legt den Einfluss der Ausrichtung fest.
- `mSteeringAngle`: Legt den Einfluss des Lenkwinkels fest.
- `mCouplingAngles`: Legt den Einfluss eines Einknickwinkels fest. Das Attribut wird für alle Einknickwinkel und somit unter Umständen für mehrere Dimensionen verwendet.

Die Bezeichner der Attribute sind an die Bezeichner der jeweiligen Attribute der Klasse `Configuration` angelehnt (siehe Abschnitt 3.2.2).

Die Methode `computeDistance` berechnet die Distanz zwischen zwei Objekten der Klasse `Configuration` nach der im vorherigen Abschnitt beschriebenen Vorgehens-

weise.

Die abstrakte Klasse `DistanceMetric` wurde implementiert, um ohne viel Aufwand weitere Realisierungen der Distanz-Metrik ergänzen zu können.

Die Distanz-Metrik wird in der Methode `init` der Klasse `RRT` für die Durchführung des Verfahrens initialisiert.

3.4.4 Realisierung des Samplings

Die Sampling-Strategie des *RRTs* wählt zufällig eine Konfiguration aus dem Kartenraum (siehe Abschnitt 2.2.1). Der Zufallswert ist dabei gleichverteilt. Die Klasse `GoalCalculatorRandom` realisiert diese Sampling-Strategie. Die Klasse sowie ihre abstrakte Basisklasse `GoalCalculator` sind in Abbildung 3.18 dargestellt.

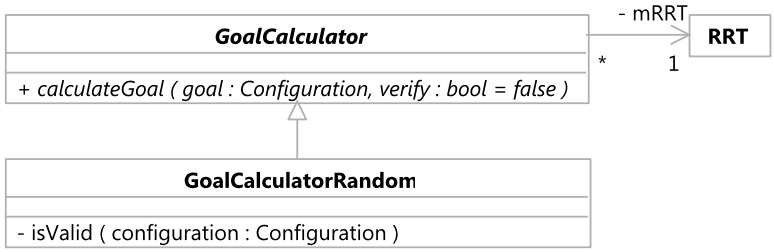

Abbildung 3.18: Die abstrakte Basisklasse `GoalCalculator` und die davon abgeleitete Klasse `GoalCalculatorRandom`.

Das Attribut `mRRT` zeigt auf das Objekt der Klasse `RRT`, für das die Sampling-Werte erzeugt werden. Über dieses Attribut hat die Klasse Zugriff sowohl auf die Karte (Attribut `mMap` vom Typ `RRTMap`), als auch auf das Fahrzeug (Attribut `mVehicle` vom Typ `Vehicle`). Die Konfiguration (Klasse `Configuration`, siehe Abschnitt 3.2.2) des Fahrzeugs gibt vor, wie die Sampling-Werte — bei denen es sich ja ebenfalls um Konfigurationen handelt — zusammengesetzt sein sollen. Im Wesentlichen geht es dabei um die Anzahl der Einknickwinkel. Die Karte gibt die Beschränkung für die Position eines Sampling-Wertes vor.

Die Methode `calculateGoal` erzeugt zufallsbasiert einen entsprechenden Sampling-Wert. Das Ergebnis wird in das übergebene Objekt der Klasse `Configuration` geschrieben. Der zweite Parameter `verify` legt fest, ob nur gültige Sampling-Werte geliefert werden sollen. Die Gültigkeit eines erzeugten Sampling-Wertes wird anhand der Beschränkungen für Lenk– und Einknickwinkel des Fahrzeugs überprüft. Die Klasse `GaolCalculatorRandom` ruft dazu ihre private Methode `isValid` auf, die wiederum die gleichnamige Methode im Fahrzeug aufruft und das Ergebnis zurückgibt. Bei der aktuellen Implementierung fließt lediglich die Position in die Distanz-Metrik ein (siehe Abschnitt 3.4.3). Daher ist auch bei den Sampling-Werten lediglich die Position relevant. Aus diesem Grund werden die erzeugten Sampling-Werte in der aktuellen Implementierung auch nicht anhand des Fahrzeugs verifiziert (Parameter `verify` ist `false`).

Weitere Strategien

In die Sampling-Strategie kann man verschiedenste Heuristiken einfließen lassen. Da dies im Allgemeinen die Verteilung der Sampling-Werte verändert, handelt es sich bei dem resultierenden Verfahren streng genommen nicht mehr um einen *Rapidly-exploring Random Tree*. In erster Linie wird dabei die schnelle und gleichmäßige Exploration abgeschwächt, die beim *RRT* durch die Gleichverteilung erreicht wird. Insofern ist beim Ändern der Sampling-Strategie darauf zu achten, dass sich die Verteilung nicht allzu stark ändert.

Im Rahmen der Masterarbeit wurde die vom *RRT* vorgesehen Gleichverteilung realisiert. Zusätzlich wurde eine weitere Strategie implementiert, die mit einer festgelegten Wahrscheinlichkeit den Zielzustand der Suche als Sampling-Wert auswählt. Die Implementierung befindet sich in der Klasse `RRT`. Das Attribut `mTargetAsXRandProb` legt die genannte Wahrscheinlichkeit fest. In der Methode `getNextXRand` wird anhand dieses Wahrscheinlichkeitswertes entschieden, ob der Zielzustand oder der nächste aus den in der Schlange `mWorkQueue` gesammelten Sampling-Werten gewählt wird.

3.4.5 Generierung und Auswahl der Steuervorgaben

In jedem Planungsschritt des *RRTs* wird ein Knoten (x_{near}) ausgewählt, von dem eine weitere Bewegung ausgeführt werden soll (siehe Abschnitt 2.2.1). Dazu muss eine Steuervorgabe ausgewählt werden. Der Algorithmus des *RRTs* lässt die Frage nach der Auswahl einer Steuervorgabe offen. Wie im genannten Abschnitt beschrieben, gibt es unter anderem die Möglichkeiten, zufällig eine Steuervorgabe zu wählen oder mehrerer Steuervorgaben auszuprobieren und nach geeigneten Kriterien diejenige mit dem besten Ergebnis zu wählen. Bevor die Wahl einer Steuervorgabe getroffen werden kann, muss — unabhängig von der Auswahl-Strategie — festgelegt werden, welche Steuervorgaben überhaupt in Frage kommen. Hierbei ist es zum Beispiel denkbar, alle vom Fahrzeug tatsächlich fahrbaren Steuervorgaben in Betracht zu ziehen. Andererseits macht es sicherlich Sinn, die Menge der möglichen Steuervorgaben in Abhängigkeit von der aktuellen Konfiguration oder weiteren aktuell relevanten Rahmenbedingungen einzugrenzen. Letztlich kann man bei der Festlegung auf eine Menge von möglichen Steuervorgaben und auf eine Auswahl-Strategie verschiedene Heuristiken einfließen lassen. Dies kann unter Umständen erhebliche Auswirkungen auf die Ergebnisse der Pfadplanung haben.

Idee zur Generierung und Auswahl der Steuervorgaben

Im Hinblick auf die Fahrbarkeit ist es erstrebenswert, bei der Pfadplanung möglichst glatte Pfade zu erzeugen. Starke, ruckartige Lenkbewegungen möchte man vermeiden, da sie in der Realität nicht wie geplant umsetzbar sind. Zudem können sie ein Risiko im Hinblick auf die Sicherheit des Fahrzeugs und des Fahrers darstellen. Bei der Pfadplanung ist es daher sinnvoll, Bewegungen mit möglichst ähnlichem Lenkwinkel bzw. möglichst geringer Lenkwinkeländerung aneinander zu reihen.

Des Weiteren möchte man im Allgemeinen eine häufige Änderung der Fahrtrichtung vermeiden. Eine Richtungsänderung ist in der Regel umständlich. Üblicherwei-

se kommt sie nur dann in Betracht, wenn die äußeren Rahmenbedingungen — also im Wesentlichen die Hindernisse — die Steuerbarkeit derart einschränken, dass das gewünschte Fahrziel durch Vorwärtsfahrt alleine nicht erreicht werden kann.

Der im Rahmen der Masterarbeit verwendete Ansatz zur Auswahl der Steuervorgabe folgt diesen Zielsetzungen. Bei dem gewählten Ansatz werden Steuervorgaben für einen Knoten des Suchbaums erzeugt, indem die vorherige Steuervorgabe vom Eltern-Knoten zum jeweiligen Knoten leicht variiert wird. Die Variation betrifft dabei ausschließlich den Lenkwinkel. Bei der Auswahl einer Steuervorgabe werden die geringsten Lenkwinkeländerungen bevorzugt. Die Distanz, die in einem Schritt gefahren wird, ist für die gesamte Planung konstant. Die Einbeziehung der Rückwärtsfahrt kann nach Wunsch eingeschränkt oder ganz vermieden werden.

Implementierung

Die beschriebene Strategie wird durch die Klasse `ControlOptionsOrganizerPV` realisiert. Die Klasse sowie ihre abstrakte Basisklasse `ControlOptionsOrganizer` sind in Abbildung 3.19 dargestellt. Der Zusatz „PV“ im Bezeichner steht für „Parent Variation“, womit ausgedrückt werden soll, dass die Klasse die vorgestellte Strategie realisiert.

Die Klassen sind dafür vorgesehen, die Steuervorgaben (Klasse `Control`, siehe Abschnitt 3.2.3) für genau einen Knoten (Klasse `Vertex`, siehe Abschnitt 3.4.2) zu definieren und auszuwählen. Das Attribut `mVertex` der Klasse `ControlOptionsOrganizer` und ihrer abgeleiteten Klassen zeigt entsprechend auf den jeweiligen Knoten. Umgekehrt kennt der Knoten das entsprechende Objekt der Klasse `ControlOptionsOrganizer`.

Die Parametrisierung und Vorgehensweise der Klasse `ControlOptionsOrganizerPV` zur Generierung und Auswahl der Steuervorgaben wird im nächsten Abschnitt erläutert. Dabei werden auch die in Abbildung 3.19 dargestellten Attribute erklärt.

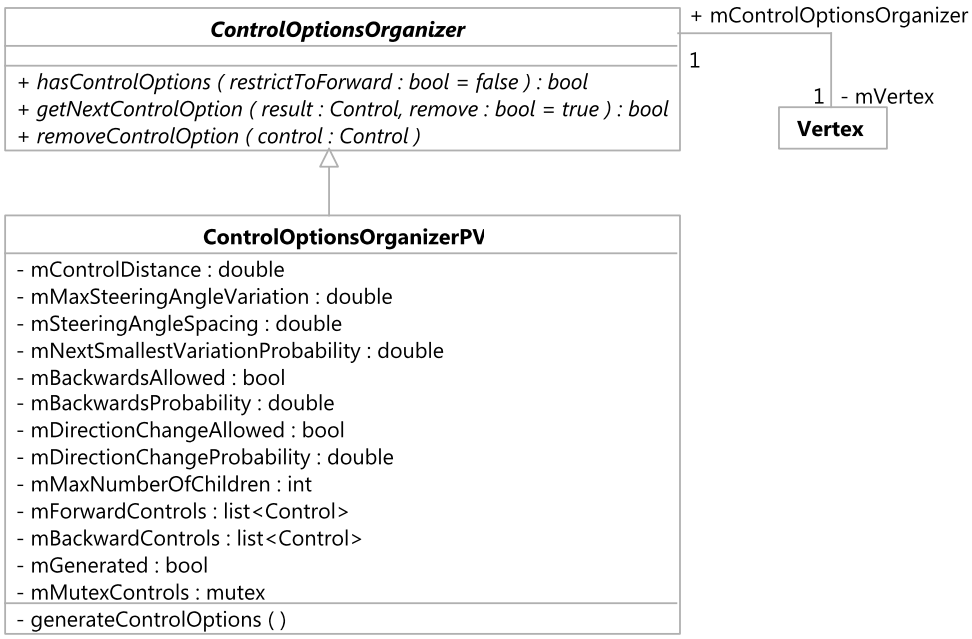

Abbildung 3.19: Die abstrakte Basisklasse `ControlOptionsOrganizer` und die davon abgeleitete Klasse `ControlOptionsOrganizerPV`.

Die abstrakte Basisklasse wurde implementiert, um die Möglichkeit zu haben, das Verfahren ohne viel Aufwand um weitere Strategien zur Generierung und Auswahl der Steuervorgaben zu erweitern. Denkbar wäre zum Beispiel eine Strategie, die die Nähe zu Hindernissen mit einbezieht. Zur Realisierung einer weiteren Strategie kann man einfach eine entsprechende abgeleitete Klasse implementieren.

Je nach Vorgehensweise der jeweiligen abgeleiteten Klasse, kann diese die Menge der relevanten Steuervorgaben vorberechnen und speichern. Dies kann im Konstruktor oder beim ersten Zugriff über eine der Methoden stattfinden. Dadurch hat die Klasse zum Beispiel die Möglichkeit, sich zu merken, welche Steuervorgaben bereits ausprobiert oder erfolgreich angewendet wurden. Die aktuelle Implementierung des Verfahrens geht davon aus, dass die Klasse eine Steuervorgabe nicht mehr als ein Mal auswählt. In diesem Hinblick ist eine Verwaltung der Steuervorgaben in der Klasse

sinnvoll. Alternativ wäre es denkbar, eine Steuervorgabe immer erst bei der nächsten Abfrage zu erzeugen. Die abgeleitete Klasse `ControlOptionsOrganizer` berechnet die Steuervorgaben bereits im Konstruktor.

Die Methode `getNextControlOption` (siehe Abbildung 3.19) liefert — falls möglich — die nächste anzuwendende Steuervorgabe für den Knoten zurück. Das Ergebnis wird in das übergebene Objekt der Klasse `Control` geschrieben. Der Rückgabewert informiert darüber, ob ein Ergebnis gefunden wurde. Der zweite Parameter `remove` legt fest, ob die ausgewählte Steuervorgabe gleichzeitig gelöscht werden soll. Dies ist für diejenigen abgeleiteten Klassen relevant, die die Steuervorgaben vorberechnen und verwalten. Das Löschen soll sicherstellen, dass die gewählte Steuervorgabe anschließend nicht mehr gewählt werden kann.

Die Methode `removeControlOption` löscht alle Steuervorgaben, die der übergebenen Steuervorgabe entsprechen. Diese Methode ist ebenfalls nur für diejenigen abgeleiteten Klassen relevant, die die Steuervorgaben vorberechnen und verwalten.

Die Methode `hasControlOptions` informiert darüber, ob der Knoten noch über mögliche Steuervorgaben verfügt. Mit dem Attribut `restrictToForward` kann die Entscheidung auf die Steuervorgaben mit Fahrtrichtung vorwärts (Attribut `mBackwards` ist `false`) beschränkt werden.

Die Methoden `hasControlOptions` und `getNextControlOption` werden von der Klasse `Worker` verwendet, um in einem Planungsschritt die Steuervorgaben für einen Knoten zu wählen (siehe Abschnitt 3.4.1). Des Weiteren kommt die Methode `hasControlOptions` in der Methode `getXNear` der Klasse `RRT` zum Einsatz. Die Methode betrachtet bei der Auswahl des Knotens `XNear` nur diejenigen Knoten, die noch über Steuervorgaben verfügen (`hasControlOptions` liefert `true`). Daher sollte der Zugriff auf die Steuervorgaben möglich sein, sobald der Knoten erzeugt wurde. Aus diesem Grund wird das Attribut `mControlOptionsOrganizer` der Klasse `Vertex` schon im Konstruktor initialisiert. Außerdem werden die Steuervorgaben des erzeugten `ControlOptionsOrganizerPV` schon in dessen Konstruktor erzeugt. Für

zukünftige Implementierungen ist dies unter Umständen ungünstig. Vielleicht macht es Sinn, die Initialisierung des Attributes `mControlOptionsOrganizer` der Klasse `Vertex` erst nachträglich von außen vorzunehmen. Denkbar wäre beispielsweise, die Aufgaben vom `Worker` nach dem Anlegen des Knotens durchführen zu lassen. Eine Änderung diesbezüglich sollte auf jeden Fall mit der beschriebenen Auswahl des Knotens `XNear` in Einklang gebracht werden.

Parametrisierung und Vorgehensweise der Klasse `ControlOptionsOrganizerPV`

Die Klasse `ControlOptionsOrganizerPV` generiert die Steuervorgaben in der privaten Methode `generateControlOptions` (siehe Abbildung 3.19), die im Konstruktor aufgerufen wird. Die erzeugten Steuervorgaben werden abhängig von ihrer Fahrtrichtung (Attribut `mBackwards`) auf die zwei Listen `mForwardControls` und `mBackwardControls` (beide vom Typ `std::list<Control>`) verteilt und dort verwaltet. Beim Aufruf der von der Basisklasse `ControlOptionsOrganizer` geerbten Methoden wird entsprechend auf diese Listen zurückgegriffen. Im Hinblick auf die notwendige Synchronisierung (siehe Anfang des Abschnittes 3.4) werden die Listen durch das Mutex-Objekt `mMutexControls` geschützt. Durch das Attribut `mGenerated` merkt sich die Klasse, ob die Steuervorgaben bereits generiert wurden.

Die restlichen Attribute bilden die Parametrisierung der Generierung und Auswahl der Steuervorgaben. Die Parameter zur Generierung der Steuervorgaben haben im Einzelnen folgende Bedeutung:

- `mControlDistance`: Die Fließkommazahl bestimmt die Distanz, die jeweils gefahren werden soll. Er legt somit das Attribut `mDistance` der `Control`-Objekte fest.
- `mMaxSteeringAngleVariation`: Die Fließkommazahl bestimmt die maximale Lenkwinkeländerung im Vergleich zur vorhergehenden Steuervorgabe (Attribut `mControl` des Knotens, für den die Steuervorgaben erzeugt werden).

- `mSteeringAngleSpacing`: Die Fließkommazahl legt die Differenz der Lenkwinkel bzw. der Lenkwinkeländerungen fest, die durch die erzeugten Steuervorgaben beschrieben werden.

Auf Basis dieser drei Parameter werden mehrere Steuervorgaben erzeugt und in die Listen `mForwardControls` und `mBackwardControls` geschrieben. Die Steuervorgaben werden folgendermaßen generiert: Sei s der Wert des Attributes `mSteeringAngleSpacing`. Dann werden nacheinander folgende Lenkwinkeländerungen generiert:

$$0, s, -s, 2s, -2s, 3s, -3s, \ldots \tag{3.2}$$

Nach dieser Vorgehensweise werden alle Lenkwinkeländerungen generiert, dessen Betrag kleiner ist als die maximal zugelassene Lenkwinkeländerung `mMaxSteeringAngleVariation`. In der angegebenen Reihenfolge werden Steuervorgaben mit der jeweiligen Lenkwinkeländerung erzeugt. Wie bereits erläutert, bezieht sich die Änderung auf die vorherige Steuervorgabe im Suchbaum (Attribut `mControl` des Knotens, für den die Steuervorgaben erzeugt werden). Als Distanz wird jeweils der durch `mControlDistance` vorgegebene Wert verwendet. Jede Steuervorgabe wird für beide Fahrtrichtungen (vorwärts und rückwärts) erzeugt und in den beiden genannten Listen gespeichert.

Die vorgestellte Reihenfolge der Lenkwinkeländerungen bewirkt einen „Linksdrall“ des Suchbaums, da immer zuerst die positive Lenkwinkeländerung angewendet wird. Unter Umständen macht es im Rahmen zukünftiger Implementierungen Sinn, jeweils eine Liste für positive und negative Lenkwinkeländerungen anzulegen. Für die Auswahl im Rahmen der Methode `getNextControlOption` könnte dann eine Wahrscheinlichkeit festgelegt werden, mit der positive bzw. negative Änderungen ausgewählt werden. Damit könnten beispielsweise diejenigen Lenkwinkeländerungen bevorzugt werden, die letztlich einen kleineren Lenkwinkel bewirken.

Die Parameter zur Auswahl aus den generierten Steuervorgaben im Rahmen der Methode `getNextControlOption` haben im Einzelnen folgende Bedeutung:

- `mNextSmallestVariationProbability`: Die Fließkommazahl legt die Wahrscheinlichkeit fest, mit der die jeweils nächstkleinere vorberechnete Lenkwinkeländerung gewählt wird.
- `mBackwardsAllowed`: Dieser boolesche Wert gibt an, ob Steuervorgaben mit Fahrtrichtung rückwärts erlaubt werden. Falls der Wert `false` ist, werden entsprechende Steuervorgaben erst gar nicht erzeugt.
- `mBackwardsProbability`: Die Fließkommazahl legt die Wahrscheinlichkeit fest, mit der eine Steuervorgabe mit Fahrtrichtung rückwärts ausgewählt wird.
- `mDirectionChangeAllowed`: Dieser boolesche Wert gibt an, ob Steuervorgaben mit der umgekehrten Fahrtrichtung der vorhergehenden Steuervorgabe im Suchbaum erlaubt werden.
- `mDirectionChangeProbability`: Die Fließkommazahl legt die Wahrscheinlichkeit fest, mit der eine Steuervorgabe mit der umgekehrten Fahrtrichtung der vorhergehenden Steuervorgabe im Suchbaum ausgewählt wird.
- `mMaxNumberOfChildren`: Der Ganzzahlwert beschränkt die Anzahl der möglichen Kind-Knoten. Sobald der Knoten, für den die Steuervorgaben generiert und ausgewählt werden (Attribut `mVertex`), diese Anzahl an Kind-Knoten erreicht hat, bietet die Klasse `ControlOptionsOrganizerPV` keine Steuervorgaben mehr an (auch wenn in den Listen `mForwardControls` oder `mBackwardControls` noch welche vorhanden sind).

Auf Basis der beschriebenen Parameter wird in der Methode `getNextControlOption` die jeweils nächste Steuervorgabe ausgewählt und zurückgeliefert. Die Wahrscheinlichkeiten für die Fahrtrichtung rückwärts (`mBackwardsProbability`) und die Änderung der Fahrtrichtung (`mDirectionChangeProbability`) werden dabei in Abhängigkeit von der vorherigen Steuervorgabe im Suchbaum kombiniert.

3.4.6 Parametrisierung und Auswertung

Nachdem nun alle Komponenten zur Realisierung des Pfadplanungsverfahrens vorgestellt wurden, soll an dieser Stelle auf die Anwendung des Verfahrens eingegangen werden. Dazu werden zum einen Möglichkeiten zur Parametrisierung des Verfahrens auf Basis der aktuellen Implementierung vorgestellt. Zum Anderen werden die Möglichkeiten zur Auswertung der Durchführung einer Suche aufgezeigt.

Parametrisierung

Die Parametrisierung des Verfahrens findet in der aktuellen Implementierung in der Klasse `RRT` statt. Sie wird sowohl über den Konstruktor als auch über die Methode `configurate` vorgenommen.

Dem Konstruktor wird neben dem Fahrzeug und der Karte auch die Start- und Ziel-Konfiguration der Suche übergeben. Alle weiteren Parameter für die Suche werden in der erwähnten Methode `configurate` festgelegt. Dabei werden folgende Attribute der Klasse `RRT` belegt, die als Parameter für die Suche eine Rolle spielen:

- `mDiscardDistance`: Relevant im Rahmen der Planung im `Worker` (siehe Anfang des Abschnittes 3.4 sowie Abschnitt 3.4.1).
- `mTargetAsXRandProb`: Relevant für das Sampling (siehe Abschnitt 3.4.4).
- `mTargetDistanceRange`: Radius des Zielgebietes (siehe Anfang des Abschnittes 3.4).
- `mStopIfTargetReached`: Legt fest, ob die Suche beendet werden soll, sobald das Ziel erreicht wurde.

Außerdem werden in der Methode `configurate` die Parameter für die Klasse `ControlOptionsOrganizerPV` zur Generierung und Auswahl der Steuervorgaben festgelegt (siehe dazu Abschnitt 3.4.5).

Die aktuelle Implementierung der Parametrisierung, insbesondere die Aufteilung auf den Konstruktor und die Methode `configurate`, stellt nur eine Übergangslösung dar. Für zukünftige Implementierungen macht es unter Umständen Sinn, die Parametrisierung des Verfahrens in eine eigene Klasse auszulagern.

Auswertung

Zur Auswertung der Durchführung einer Suche stellen sowohl die Klasse `RRT` als auch die Klasse `Vertex` Funktionalitäten bereit.

Die Klasse `Vertex` (siehe Abschnitt 3.4.2), die einen Knoten im Suchbaum darstellt, verfügt dazu über folgende Attribute:

- `mOnPathToTarget`: Gibt an, ob der Knoten auf dem Pfad zum Ziel liegt.
- `mStepsFromRoot`: Gibt die Tiefe des Knotens im Suchbaum an.
- `mLastDirectionChange`: Gibt die Anzahl der Vorgänger-Knoten seit der letzten Änderung der Fahrtrichtung an.
- `mDistanceFromRoot`: Gibt die Länge des Pfades von der Wurzel zum Knoten an.

Die vorgestellten Werte könnten im Hinblick auf eine Auswertung des Suchbaumes während oder nach der Suche von Interesse sein. Zudem fließen sie in die Funktionalitäten zur Auswertung der Klasse `RRT` mit ein.

Die Klasse `RRT` beobachtet den Aufbau des Suchbaums durch die Methode `addVertex` (siehe Anfang des Abschnittes 3.4). Die Methode wird vom `Worker` (siehe Abschnitt 3.4.1) beim Hinzufügen eines neuen Knotens aufgerufen. Die Klasse `RRT` wird damit über den neuen Knoten informiert und kann die Eigenschaften des Knotens auswerten.

In der Methode `addVertex` wird unter anderem überprüft, ob mit dem hinzugefügten Knoten das Ziel erreicht wurde. Falls bereits ein anderer Knoten das Ziel erreicht hat,

wird überprüft, ob der Knoten auf einem kürzeren Pfad zum Ziel liegt. Die Klasse `RRT` speichert immer den Knoten, der das Ziel auf dem kürzesten Pfad erreicht hat. Damit hat die Klasse auch Zugriff auf die Länge dieses Pfades (Attribut `mDistanceFromRoot` der Klasse `Vertex`).

Des Weiteren zählt die Klasse `RRT` in der Methode `addVertex` die Anzahl der Knoten (Attribut `mNumberOfVertices`). Dazu wird die Anzahl in der Methode `init` beim Erzeugen des Wurzel-Knotens auf 1 gesetzt und bei jedem Aufruf der Methode `addVertex` inkrementiert. Außerdem wird die Baumhöhe aktualisiert. Dazu wird das Attribut `mStepsFromRoot` des Knotens (Klasse `Vertex`) abgefragt.

Um verschiedene Durchführungen der Suche miteinander vergleichen zu können, wurde außerdem eine Zeitmessung implementiert. Die Zeitmessung wird beim Starten der Suche (Methode `search`) gestartet und beendet, sobald ein Knoten das Ziel erreicht hat (Methode `addVertex`). Die benötigte Zeit wird zusammen mit der Anzahl der Knoten sowie der Pfadlänge von der Wurzel zu dem Knoten, der das Ziel erreicht hat, auf der Konsole ausgegeben.

Die Implementierungen zur Auswertung stellen nur einen ersten Ansatz dar und lassen sich noch beliebig erweitern. Es wird empfohlen, zukünftige Implementierungen in diesem Zusammenhang analog zu den bereits vorhanden Funktionalitäten in der Klasse `Vertex` unterzubringen und in der Methode `addVertex` der Klasse `RRT` zu aggregieren. Unter Umständen macht es Sinn, die Auswertung und insbesondere die aktuell in der Klasse `RRT` dazu angelegten Attribute in eine eigene Klasse auszulagern.

3.5 Grafische Darstellung

In diesem Abschnitt wird die Umsetzung der grafischen Darstellung der vorgestellten Komponenten behandelt. Dazu gehört sowohl die Darstellung der Karte mit den Hindernissen und dem Fahrzeug als auch die Darstellung des Suchbaumes. Zur Um-

setzung wurde *GLUT* verwendet.

Bei der Umsetzung wurde auf einen modularen Aufbau Wert gelegt. Zu jeder Komponente, die visualisiert werden soll, wurde daher eine eigene Klasse implementiert. Alle Klassen zur Visualisierung werden letztlich von einer abstrakten Basisklasse abgeleitet. Es handelt sich dabei um die Klasse `GlutCanvasPainter`.

Neben den Klassen, die die Visualisierung einzelner Komponenten realisieren, wird zusätzlich eine Klasse benötigt, die ein Fenster erzeugt und damit eine Leinwand realisiert, auf der die Visualisierungen gezeichnet werden können. Die dazu implementierte Klasse heißt `GlutCanvas`. Sie hat ebenfalls eine abstrakte Basisklasse, die Klasse `Canvas`.

Die drei bereits vorgestellten Klassen sind in Abbildung 3.20 dargestellt.

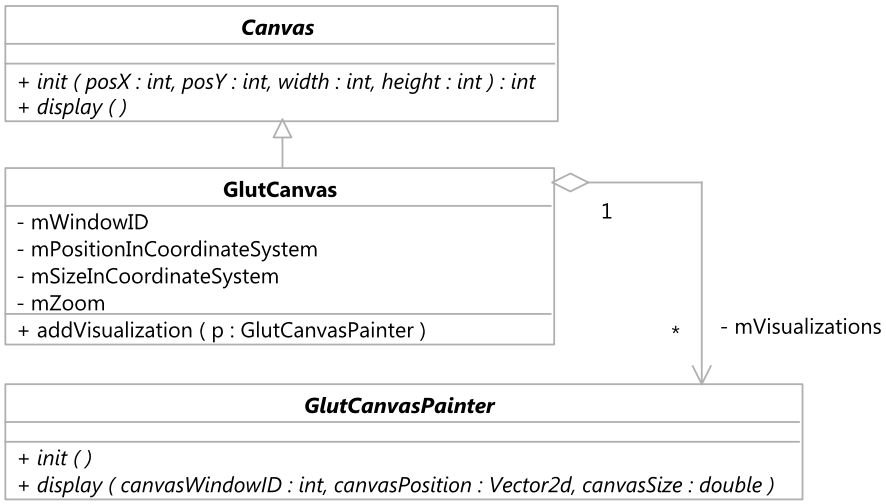

Abbildung 3.20: Basisklassen der Visualisierung.

Die Klasse `GlutCanvas` realisiert die Leinwand, auf der gezeichnet wird. Gleichzeitig stellt sie eine Container-Klasse für alle Objekte der Klasse `GlutCanvasPainter` dar, die auf die Leinwand zeichnen wollen.

Die Methoden `init` und `display` erbt sie von der abstrakten Basisklasse. In der Methode `init` wird über die *GLUT*-Methode `glutCreateWindow` ein neues Fenster

erzeugt. Die *GLUT*-Methode liefert einen `int`-Wert zurück, über den in das erzeugte Fenster gezeichnet werden kann. Dieser Wert wird im Attribut `mWindowID` gespeichert. Die übergebenen Parameter der Methode `init` bestimmen die Position und Größe des Fensters auf dem Bildschirm.

Das Attribut `mVisualizations` vom Typ `std::list<GlutCanvasPainter*>` beinhaltet alle Objekte der Klasse `GlutCanvasPainter`, die auf die Leinwand zeichnen. Über die Methode `addVisualization` können neue Objekte hinzugefügt werden.

Die drei Attribute `mPositionInCoordinateSystem`, `mSizeInCoordinateSystem` sowie `mZoom` geben an, welcher Bereich des Koordinatensystems aktuell im Fenster dargestellt wird. Die Visualisierungen (`mVisualizations`) müssen diese Werte beim Zeichnen berücksichtigen. Die Attribute können über entsprechende Setter-Methoden gesetzt und über Getter-Methoden abgefragt werden. Damit ist es letztlich möglich, im Fenster zu scrollen und zu zoomen.

In der Methode `display` werden alle Visualisierungen gezeichnet. Dazu wird zunächst über die *GLUT*-Methode `glScalef` der Zoom (Attribut `mZoom`) eingestellt. Dazu wird über die Objekte in `mVisualizations` iteriert und für jedes Objekt die `display`-Methode aufgerufen. Übergeben werden die Fester-ID (`mWindowID`) sowie die Position und Größe im Koordinatensystem, die der aktuell angezeigte Ausschnitt der Leinwand darstellt (Attribute `mPositionInCoordinateSystem` und `mSizeInCoordinateSystem`).

Die abstrakte Basisklasse `Canvas` wurde entworfen, um die Möglichkeit zu haben, auch andere Bibliotheken als *GLUT* zur Visualisierung zu verwenden und entsprechende abgeleitete Klassen zu implementieren.

Die abstrakte Klasse `GlutCanvasPainter` realisiert eine Visualisierung. Sie besitzt ebenfalls eine Methode `init` sowie eine Methode `display`. Die Methode `init` kann von den abgeleiteten Klassen beliebig genutzt werden, beispielsweise um Vorbereitungen für das Zeichnen zu treffen.

Die Methode `display` soll die Visualisierung in das übergebene Fenster (`glutCanvasWindowID`) zeichnen. Die anderen beiden Parameter geben die Position und Größe im Koordinatensystem an, die der aktuelle Ausschnitt des Fensters darstellt. Das Zeichnen muss an diese Werte angepasst werden. Die Methode wird von der Leinwand (Klasse `GlutCanvas`) aufgerufen, auf die gezeichnet werden soll.

Es wurden drei Klassen implementiert, die von der Klasse `GlutCanvasPainter` abgeleitet sind:

- `GlutCanvasVehicleUnitPainter` zur Visualisierung eines Fahrzeuggliedes
- `GlutCanvasGridPainter` zur Visualisierung einer Belegungsmatrix
- `GlutCanvasRRTPainter` zur Visualisierung des Suchbaums

Die drei Klassen werden in den folgenden Abschnitten vorgestellt.

3.5.1 Visualisierung eines Fahrzeuggliedes

Die Visualisierung eines Fahrzeuggliedes (Klasse `VehicleUnit`) wird durch die Klasse `GlutCanvasVehicleUnitPainter` realisiert. Diese ist in Abbildung 3.21 dargestellt.

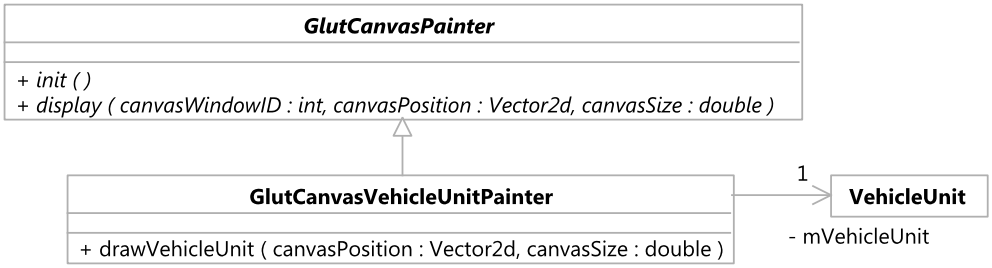

Abbildung 3.21: Die Klasse `GlutCanvasVehicleUnitPainter`.

Die Klasse ist abgeleitet von der abstrakten Basisklasse `GlutCanvasPainter`. Entsprechend implementiert sie die Methoden `init` und `display`.

Das Attribut `mVehicleUnit` zeigt auf das Fahrzeugglied, das gezeichnet werden soll. Dieses wird im Konstruktor übergeben. Im Konstruktor wird außerdem die Methode `init` aufgerufen. In dieser Methode werden lediglich Farbwerte zum Zeichnen gesetzt.

In der Methode `display` wird zunächst das Fenster zum Zeichnen gesetzt (*GLUT*-Methode `glutSetWindow`). Außerdem wird die Farbe zum Zeichnen gesetzt (*GLUT*-Methode `glColor3f`). Anschließend wird die private Methode `drawVehicleUnit` aufgerufen, um das Fahrzeugglied zu zeichnen.

Die Methode `drawVehicleUnit` ermittelt die aktuelle Position und Ausrichtung des Fahrzeuggliedes (Methoden `getAbsolutePosition` und `getAbsoluteOrientation` der Klasse `VehicleUnit`). Außerdem werden die Abmessungen des Fahrzeugglieds abgefragt. Daraus werden die genauen Koordinaten der Eckpunkte und weiterer zu zeichnender Punkte berechnet. Die Methode zeichnet Position, Umrisse und hintere Kupplung des Fahrzeugglieds sowie entweder die Deichsel (vorne) oder den Lenkeinschlag, je nachdem, ob es sich um Anhänger oder Zugfahrzeug handelt.

Die jeweiligen Koordinaten werden vor dem Zeichnen über die Parameter `canvasPosition` und `canvasSize` an den aktuell im Fenster dargestellten Bereich des Koordinatensystems angepasst. Zum Zeichnen werden sie durch die *GLUT*-Funktionalität `glVertex3f` ausgedrückt. Beim Zeichnen werden die von *GLUT* bereitgestellten Formen `GL_POINTS`, `GL_LINES` und `GL_LINE_LOOP` verwendet.

3.5.2 Visualisierung einer Belegungsmatrix

Die Visualisierung einer Belegungsmatrix (Klasse `GridBase`) wird durch die Klasse `GlutCanvasGridPainter` realisiert. Diese ist in Abbildung 3.22 dargestellt.

Die Klasse ist abgeleitet von der abstrakten Basisklasse `GlutCanvasPainter`. Entsprechend implementiert sie die Methoden `init` und `display`.

Das Attribut `mGrid` zeigt auf die Belegungsmatrix, die gezeichnet werden soll. Diese

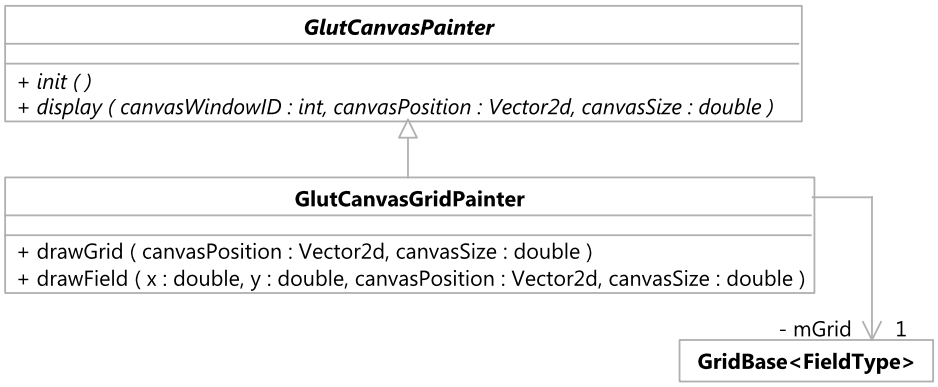

Abbildung 3.22: Die Klasse `GlutCanvasGridPainter`.

wird im Konstruktor übergeben. Zudem wird im Konstruktor die Methode `init` aufgerufen. Diese hat in der aktuellen Implementierung aber keine Funktionalität. Im Konstruktor wird außerdem abhängig vom `GridType` (Attribut `mGridType` der Klasse `GridBase`) die Farbe zum Zeichnen festgelegt. Dadurch können die Belegungsmatrizen von Fahrzeug und Hindernissen unterschiedlich dargestellt werden.

In der Methode `display` wird zunächst das Fenster zum Zeichnen gesetzt (*GLUT*-Methode `glutSetWindow`). Außerdem wird die Farbe zum Zeichnen gesetzt (*GLUT*-Methode `glColor3f`). Anschließend wird die private Methode `drawGrid` aufgerufen, um die Belegungsmatrix zu zeichnen.

Die Methode `drawGrid` kopiert zunächst die interne Belegungsmatrix `mArea` (Typ `Matrix<FieldType,Dynamic,Dynamic>`) des Attributs `mGrid`. Dies ist im Hinblick auf die Synchronisierung notwendig, da die interne Matrix jederzeit geändert werden könnte. Anschließend wird auf der Kopie gearbeitet. Für jedes Feld der Matrix wird überprüft, ob es frei ist (also `mFree` der Klasse `GridBase` entspricht). Ist dies nicht der Fall, wird davon ausgegangen, dass das Feld belegt ist. In diesem Fall wird das Feld durch Aufruf der privaten Methode `drawField` gezeichnet. Dazu werden der Methode die realen Koordinaten des Feldes übergeben.

Die Methode `drawField` zeichnet ein Rechteck in der Rastergröße der Belegungsma-

trix an der übergebenen Position.

Die jeweiligen Koordinaten werden vor dem Zeichnen über die Parameter `canvasPosition` und `canvasSize` an den aktuell im Fenster dargestellten Bereich des Koordinatensystems angepasst. Zum Zeichnen werden sie durch die *GLUT*-Funktionalität `glVertex3f` ausgedrückt. Beim Zeichnen wird die von *GLUT* bereitgestellten Form `GL_QUADS` verwendet.

3.5.3 Visualisierung des Suchbaums

Die Visualisierung des Suchbaums (Klassen `RRT` und `Vertex`) wird durch die Klasse `GlutCanvasRRTPainter` realisiert. Diese ist in Abbildung 3.23 dargestellt.

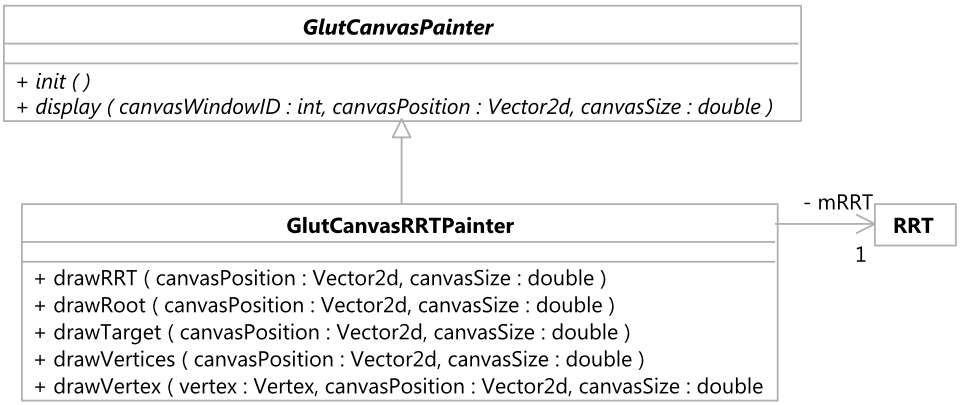

Abbildung 3.23: Die Klasse `GlutCanvasRRTPainter`.

Die Klasse ist abgeleitet von der abstrakten Basisklasse `GlutCanvasPainter`. Entsprechend implementiert sie die Methoden `init` und `display`.

Das Attribut `mRRT` zeigt auf den Suchbaum, der gezeichnet werden soll. Dieser wird im Konstruktor übergeben. Zudem wird im Konstruktor die Methode `init` aufgerufen. In dieser Methode werden lediglich Farbwerte zum Zeichnen gesetzt.

In der Methode `display` wird zunächst das Fenster zum Zeichnen gesetzt (*GLUT*-Methode `glutSetWindow`). Anschließend wird die private Methode `drawRRT` aufgeru-

fen, um den Suchbaum zu zeichnen.

Die Methode `drawRRT` ruft nacheinander die privaten Methoden `drawRoot`, `drawTarget` und `drawVertices` auf.

Die Methode `drawRoot` zeichnet den Wurzel-Knoten (Attribut `mRoot` der Klasse `RRT`). Dieser wird durch Farbe und Größe von den anderen Knoten abgehoben. Dazu werden die *GLUT*-Methoden `glColor3f` und `glPointSize` verwendet. Die Koordinaten ergeben sich aus der Konfiguration des Knotens.

Die Methode `drawTarget` zeichnet das Ziel der Suche (Attribut `mTarget` der Klasse `RRT`). Dieses wird wie der Wurzel-Knoten durch Farbe und Größe von den anderen Knoten abgehoben. Dazu werden dieselben *GLUT*-Methoden wie beim Wurzel-Knoten verwendet. Die Koordinaten entsprechen der Position des Ziels.

Die Methode `drawVertices` zeichnet alle Knoten des Suchbaums (außer den Wurzel-Knoten). Dazu wird ein Objekt der Klasse `RRTVertexIterator` erzeugt. Über den Iterator wird der Suchbaum traversiert und für jeden Knoten die Methode `drawVertex` aufgerufen.

Die Methode `drawVertex` zeichnet sowohl den Knoten selbst als auch eine Verbindung zum Eltern-Knoten. Das Zeichnen des Knotens selbst wird analog zum Zeichnen des Wurzel-Knotens durchgeführt. Die Verbindung zum Eltern-Knoten wird einfach als gerade Linie zwischen den Knoten gezeichnet. Falls die Knoten auf dem Pfad zum Ziel liegen (Attribut `mOnPathToTarget` der Klasse `Vertex` ist `true`), wird die Linie durch Farbe und Stärke hervorgehoben. Dazu wird die *GLUT*-Methode `glLineWidth` verwendet. Falls die entsprechende Bewegung zwischen den Knoten (Attribut `mControl` der Klasse `Vertex`) die Fahrtrichtung rückwärts hat, wird die Linie ebenfalls durch eine andere Farbe gekennzeichnet.

Die jeweiligen Koordinaten werden vor dem Zeichnen über die Parameter `canvasPosition` und `canvasSize` an den aktuell im Fenster dargestellten Bereich des Koordinatensystems angepasst. Zum Zeichnen werden sie durch die *GLUT*-Funktionalität

`glVertex3f` ausgedrückt. Beim Zeichnen werden die von *GLUT* bereitgestellten Formen `GL_POINTS` und `GL_LINES` verwendet.

3.6 Realisierung eines Tools zur Durchführung von Experimenten

Die in den vorherigen Abschnitten des Kapitels 3 vorgestellten Komponenten bilden zusammen eine Funktionsbibliothek. Zur Durchführung einer Pfadplanung müssen die einzelnen Funktionalitäten allerdings noch in einer ausführbaren Anwendung zusammengeführt werden. Zu diesem Zweck wurde ein Tool entwickelt, das erste Experimente ermöglicht. Wie in Abschnitt 3.1 schon angedeutet, handelt es sich dabei lediglich um eine erste Lösung mit überschaubaren Funktionalitäten. Aus zeitlichen Gründen war die Entwicklung einer Anwendung mit umfangreichen Funktionalitäten im Rahmen dieser Masterarbeit leider nicht mehr möglich. Für zukünftige Experimente wird empfohlen, das Tool umzustrukturieren und hinsichtlich der Funktionalität auszubauen. Das Tool ist in der Datei `demo.cpp` im Quellcode der Masterarbeit realisiert.

Das Tool ermöglicht die Durchführung einer Pfadplanung für vordefinierte Szenarios, die im Rahmen der Experimente den Vergleich verschiedener Parametrisierungen des Verfahrens ermöglichen sollen. Ein Szenario beschreibt dabei die Planung auf einer bestimmten Karte mit einem bestimmten Fahrzeug und einer bestimmten Parametrisierung des Verfahrens. Vordefiniert sind 9 Szenarios, basierend auf 2 vordefinierten Fahrzeugen, 3 vordefinierten Karten sowie 6 vordefinierten Parametrisierungen des Verfahrens. Im Rahmen zukünftiger Implementierungen macht es Sinn, die Auswahl von Fahrzeug, Karte und Parametrisierung dynamischer zu gestalten.

Die Implementierung ist auf eine `main`-Methode sowie auf diverse Methoden zum Erzeugen eines Fahrzeuges, zum Einlesen einer Karte oder zum Initialisieren einer Parametrisierung für das Verfahren aufgeteilt.

Die Karte wird jeweils aus einer *PPM*-Datei eingelesen. Mittels der in Abschnitt 3.3.4 vorgestellten Funktionalität wird die Karte in eine entsprechende Belegungsmatrix umgewandelt. Mit der Auswahl der Karte werden gleichzeitig auch die Rastergröße sowie Start und Ziel der Suche festgelegt.

Bei dem Fahrzeug, das in den Szenarien verwendet wird, handelt es sich um die Nachbildung eines Zugfahrzeugs mit Sattelauflieger und Drehschemelanhänger. Die Belegungsmatrizen für das Fahrzeug werden durch Aufruf der Methode `precalculateWrappings` vorberechnet.

Mit der Parametrisierung werden die in Abschnitt 3.4.6 vorgestellten Parameter für die Durchführung einer Suche festgelegt.

Die Auswahl des Szenarios findet über die Konsole statt. Anschließend öffnet sich ein Fenster, in dem die Karte und das Fahrzeug dargestellt werden. Das Fenster und die Visualisierungen sind mit *GLUT* realisiert. Dazu wird in der `main`-Methode zunächst die *GLUT*-Methode `glutInit` aufgerufen. Anschließend wird eine neue Leinwand (Klasse `GlutCanvas`, siehe Abschnitt 3.5) für die Visualisierungen von Karte, Fahrzeug und Suchbaum erzeugt. Auch bei den genannten Visualisierungen wird auf die in Abschnitt 3.5 vorgestellten Klassen zurückgegriffen. Die Karte wird durch einen `GlutCanvasGridPainter` (siehe Abschnitt 3.5.2) visualisiert. Das Fahrzeug wird durch mehrere `GlutCanvasVehicleUnitPainter` (siehe Abschnitt 3.5.1) visualisiert. Zudem werden die Belegungsflächen des Fahrzeugs durch mehrere `GlutCanvasGridPainter` grafisch dargestellt. Zur Darstellung des Suchbaums wird ein `GlutCanvasRRTPainter` (siehe Abschnitt 3.5.3) erzeugt.

Anschließend werden in der `main`-Methode Funktionen zur Bedienung der Anwendung mittels Tastatureingaben definiert. Durch Aufruf der Methoden `glutKeyboardFunc` und `glutSpecialFunc` werden entsprechende Methoden definiert, die die Eingabe bestimmter Tasten behandeln. In den jeweiligen Methoden sind die Aktionen für verschiedene Tastatureingaben definiert. Dort ist beispielsweise definiert, dass mit der Taste `s` die Methode `search` der Klasse `RRT` aufgerufen wird, also die Suche

gestartet wird. Über dieselbe Taste kann die Suche abgebrochen werden (Aufruf der Methode `stopsearch` der Klasse `RRT`). Des Weiteren sind Tasten definiert für die Navigation im Koordinatensystem. Durch Aufruf der Methoden `setZoom` und `setPositionInCoordinateSystem` der Klasse `GlutCanvas` ist dabei zum Beispiel das Zoomen und Scrollen realisiert (siehe Abschnitt 3.5).

Abschließend wird die Methode `display` über die *GLUT*-Methode `glutDisplayFunc` als Visualisierungsfunktion festgelegt. In der Methode `dispay` wird im Wesentlichen die gleichnamige Methode der Klasse `GlutCanvas` aufgerufen, die alle Visualisierungen zeichnen lässt. Durch Aufruf der *GLUT*-Methode `glutMainLoop` wird das Fenster mit den Visualisierungen schließlich erzeugt und bis zum Beenden der Anwendung angezeigt.

Die grafische Ausgabe des Tools nach Durchführung einer Planung ist in Abbildung 3.24 dargestellt.

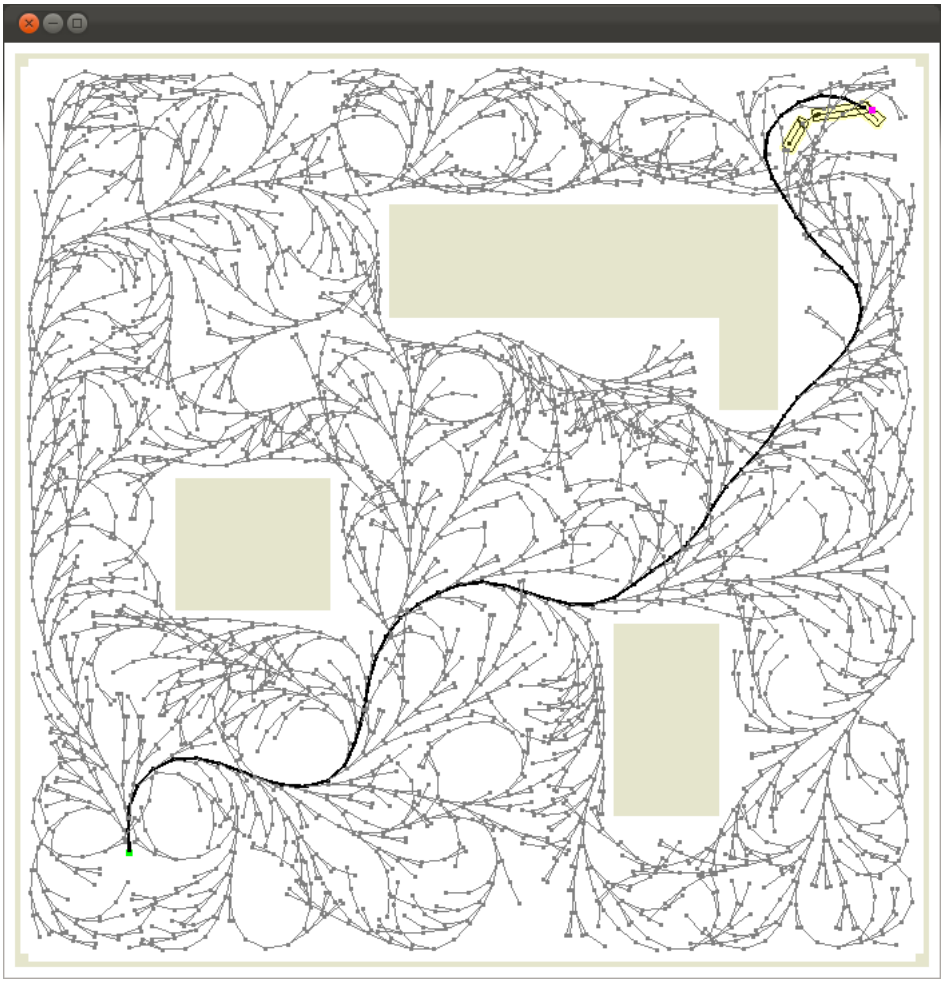

Abbildung 3.24: Tool zum Durchführen von Experimenten.

4 Experimente

Die in diesem Kapitel behandelten Experimente sollen die in den Formulierungen der Ziele beschriebenen Zusammenhänge untersuchen (siehe Abschnitt 1.2, Punkte 4 und 5). Aus zeitlichen Gründen war es im Rahmen dieser Masterarbeit leider nicht mehr möglich, mit systematischer Herangehensweise umfangreiche Experimente durchzuführen. Auf Basis des in Abschnitt 3.6 vorgestellten Tools konnten jedoch erste Experimente durchgeführt und erste qualitative Ergebnisse erzielt werden. Im Rahmen dieses Kapitels werden diese durchgeführten Experimente beschrieben. Eine Auswertung findet im folgenden Kapitel 5 statt.

4.1 Durchgeführte Experimente

Im Rahmen der durchgeführten Experimente wurden folgende Fragestellungen angegangen:

- **Rastergröße**: Wie wirkt sich die Wahl der Rastergröße für die Belegungsmatrizen von Fahrzeug und Hindernissen auf die Performanz der Suche aus?
- **Rückwärtsfahrt**: Wie wirkt sich die Miteinbeziehung der Rückwärtsfahrt auf die Exploration und den entstehenden Suchbaum aus? Welche Vor– und Nachteile sind gegenüber einer Planung mit ausschließlich vorwärts gerichteten Bewegungen zu beobachten?
- **Zielzustand als Sampling-Wert**: Welche Auswirkung hat die in Abschnitt 3.4.4 vorgestellte Sampling-Strategie, bei der mit einer festgelegten Wahrschein-

lichkeit der Zielzustand als Sampling-Wert verwendet wird? Wie wird die Exploration beeinflusst? Wird das Ziel schneller erreicht?

- **Auswahl der Steuervorgabe**: Welche Auswirkungen haben verschiedene Parametrisierungen zur Generierung und Auswahl der Steuervorgaben (Klasse `ControlOptionsOrganizerPV`, siehe Abschnitt 3.4.5)? Welche Distanzen sind für eine Steuervorgabe sinnvoll? Wie wirken sich verschiedene Werte zur Variation des Lenkwinkels auf die Exploration und auf die Pfade aus?

Die Experimente wurde auf zwei Karten durchgeführt. Die Karten mit geringer und hoher Hindernisdichte sind in Abbildung 4.1.

Abbildung 4.1: Karten mit geringer (links) und hoher (rechts) Hindernisdichte.

Das Fahrzeug — erkennbar an der gelben Belegungsfläche — stellt ein Zugfahrzeug mit Sattelauflieger und Drehschemelanhänger dar. Das Ziel der Suche ist jeweils durch den pink gefärbten Punkt dargestellt.

Die in Abschnitt 3.6 genannten vordefinierten Parametrisierungen des Verfahrens wurden im Hinblick auf diese Fragestellungen ausgelegt. Die Experimente dazu wurden auf Basis dieser vordefinierten Parametrisierungen durchgeführt. Es handelt sich dabei jeweils um Abwandlungen einer grundlegenden Parametrisierung. Anhand der

Ergebnisse der grundlegenden Parametrisierung sollen die Auswirkungen der Änderung einzelner Parameter vergleichbar gemacht werden. Im Folgenden werden zunächst diese grundlegende Parametrisierung und das Ergebnis dazu vorgestellt. Im Anschluss daran werden die einzelnen Durchführungen der Suche mit den verschiedenen Parametrisierungen vorgestellt.

4.1.1 Grundlegende Parametrisierung

Die Möglichkeiten zur Parametrisierung wurden in Abschnitt 3.4.6 vorgestellt. Die grundlegende Parametrisierung, die für die durchgeführten Experimente verwendet wurde, ist die Folgende:

- `mDiscardDistance` = 0.25
- `mTargetAsXRandProb` = 0.0
- `mTargetDistanceRange` = 7.5

Die grundlegende Parametrisierung zur Generierung der Steuervorgaben ist Folgende:

- `mControlDistance` = 4.0
- `mMaxSteeringAngleVariation` = 15°
- `mSteeringAngleSpacing` = 3°

Die grundlegende Parametrisierung zur Auswahl aus den generierten Steuervorgaben ist Folgende:

- `mNextSmallestVariationProbability` = 1.0
- `mBackwardsAllowed` = `false`
- `mBackwardsProbability` = 0.0
- `mDirectionChangeAllowed` = `false`

- `mDirectionChangeProbability` = 0.0
- `mMaxNumberOfChildren` = 12

Zudem wurde als Rastergröße für die Belegungsmatrizen von Fahrzeug und Hindernissen jeweils der Wert 0.25 verwendet.

Basierend auf dieser Parametrisierung wurde die Planung ein erstes Mal durchgeführt, um Vergleichswerte für die im Folgenden durchgeführten Experimente zu haben. Der resultierende Suchbaum ist in Abbildung 4.2 dargestellt.

Abbildung 4.2: Ergebnis des Referenz-Experiments.

Bei der dargestellten Suche wurde das Ziel nach 5,8 Sekunden und 1339 Knoten erreicht. Der Pfad zum Ziel (durch die schwarze, breitere Linie hervorgehoben) hat eine Länge von 344, was 86 Steuervorgaben entspricht. Die Anzahl der Steuervorgaben stellt gleichzeitig die Baumtiefe des letzten Knotens dar.

Im Rahmen der durchgeführten Experimente wurden einzelne Parameter angepasst, um deren Auswirkung zu untersuchen. Die Experimente werden in den folgenden Abschnitten vorgestellt.

4.1.2 Experiment „Rastergröße“

Bei diesem Experiment soll untersucht werden, wie sich die Wahl der Rastergröße für die Belegungsmatrizen von Fahrzeug und Hindernissen auf die Performanz der Suche auswirkt.

Durchführung

Dazu wurde einmal mit Rastergröße 0.5 und einmal mit Rastergröße 0.1 geplant. Im Vergleich zur grundlegenden Parametrisierung (siehe Abschnitt 4.1.1) wurde nur die Rastergröße geändert.

Beobachtung

Die Ergebnisse sind in Abbildung 4.3 dargestellt.

Abbildung 4.3: Ergebnisse des Experiments „Rastergröße“ mit Rastergröße 0.5 (links) und 0.1 (rechts).

Mit der Rastergröße 0.5 (links dargestellt) wurde das Ziel nach 4,3 Sekunden und 1247 Knoten erreicht. Der Pfad zum Ziel hat eine Länge von 340, was 85 Steuervorgaben

entspricht.

Mit der Rastergröße 0.1 (rechts dargestellt) wurde das Ziel nach 7,4 Sekunden und 1595 Knoten erreicht. Der Pfad zum Ziel hat eine Länge von 336, was 84 Steuervorgaben entspricht.

Des Weiteren konnte beobachtet werden, dass die grafische Darstellung bei Rastergröße 0.1 deutlich verlangsamt wurde, sodass der Suchbaum nur alle paar Sekunden angezeigt wurde. Bei der Geschwindigkeit der Exploration konnte aber keine nennenswerte Verlangsamung beobachtet werden.

Eine Bewertung der Ergebnisse findet im folgenden Kapitel in Abschnitt 5.1.1 statt.

4.1.3 Experiment „Rückwärtsfahrt“

Bei diesem Experiment soll untersucht werden, wie sich die Miteinbeziehung der Rückwärtsfahrt auf die Exploration und den entstehenden Suchbaum auswirkt. Dabei sollen die Vor– und Nachteile gegenüber einer Planung mit ausschließlich vorwärts gerichteten Bewegungen herausgestellt werden.

Durchführung

Dazu wurden folgende Parameter im Vergleich zur grundlegenden Parametrisierung (siehe Abschnitt 4.1.1) angepasst:

- `mBackwardsAllowed` = `true`
- `mBackwardsProbability` = 0.0
- `mDirectionChangeAllowed` = `true`
- `mDirectionChangeProbability` = 0.1

Die Wahrscheinlichkeit 0.0 für die Rückwärtsfahrt bedeutet dabei, dass diese erst ausprobiert wird, sobald am entsprechenden Knoten keine vorwärts gerichteten Steu-

ervorgaben mehr angeboten werden.

Beobachtung

Das Ergebnis ist in Abbildung 4.4 dargestellt. Zum Vergleich werden die Ergebnisse des in Abschnitt 4.1.1 vorgestellten Referenz-Experiments herangezogen.

Abbildung 4.4: Ergebnisse des Experiments „Rückwärtsfahrt“.

Unter Einbeziehung der Rückwärtsfahrt wurde das Ziel nach 8,9 Sekunden und 1195 Knoten erreicht. Der Pfad zum Ziel hat eine Länge von 432, was 108 Steuervorgaben entspricht.

Des Weiteren konnte beobachtet werden, dass der Suchraum sehr viel langsamer exploriert wurde.

Eine Bewertung der Ergebnisse findet im folgenden Kapitel in Abschnitt 5.1.2 statt.

4.1.4 Experiment „Zielzustand als Sampling-Wert“

Bei diesem Experiment soll untersucht werden, wie sich die in Abschnitt 3.4.4 vorgestellte Sampling-Strategie, bei der mit einer festgelegten Wahrscheinlichkeit der

Zielzustand als Sampling-Wert verwendet wird, auf die Exploration auswirkt. Dabei soll insbesondere festgestellt werden, ob das Ziel schneller erreicht wird und welche Vor– oder Nachteile sie außerdem mit sich bringt.

Durchführung

Dazu wurde im Vergleich zur grundlegenden Parametrisierung (siehe Abschnitt 4.1.1) lediglich der Parameter `mTargetAsXRand` angepasst. Der Wert für diesen Parameter wurde auf 0.2 gesetzt, d.h. mit einer Wahrscheinlichkeit von 20% wird der Zielzustand als Sampling-Wert verwendet.

Beobachtung

Das Ergebnis ist in Abbildung 4.5 dargestellt. Zum Vergleich werden die Ergebnisse des in Abschnitt 4.1.1 vorgestellten Referenz-Experiments herangezogen.

Abbildung 4.5: Ergebnisse des Experiments „Zielzustand als Sampling-Wert“.

Mit der veränderten Sampling-Strategie wurde das Ziel nach 2,4 Sekunden und 947 Knoten erreicht. Der Pfad zum Ziel hat eine Länge von 332, was 83 Steuervorgaben entspricht.

Des Weiteren konnte beobachtet werden, dass das Ziel unmittelbar angesteuert wurde, sobald der Suchbaum in der Nähe des Ziels angekommen war. Dies konnte bei den anderen Parametrisierungen nicht beobachtet werden.

Eine Bewertung der Ergebnisse findet im folgenden Kapitel in Abschnitt 5.1.3 statt.

4.1.5 Experiment „Steuervorgaben“

Bei diesem Experiment soll untersucht werden, wie sich die in Abschnitt 3.4.5 vorgestellte Parametrisierung zur Generierung und Auswahl der Steuervorgaben auf die Exploration auswirkt. Dabei soll herausgefunden werden, welche Distanzen für die Steuervorgaben sinnvoll sind und wie sich verschiedene Werte für die Variation des Lenkwinkels sowohl auf die Exploration als auch auf die Pfade auswirken. Das Experiment wurde auf einer Karte mit geringer Hindernisdichte durchgeführt.

Durchführung

Es wurde mit zwei Parametrisierungen geplant, die sich durch die Parametrisierung zur Generierung der Steuervorgaben unterscheiden.

Die erste Parametrisierung soll kürzere Steuervorgaben und kleinere Lenkwinkeländerungen bewirken. Im Vergleich zur grundlegenden Parametrisierung (siehe Abschnitt 4.1.1) wurden dazu folgende Parameter variiert:

- `mControlDistance` = 2.0
- `mMaxSteeringAngleVariation` = 7.5°
- `mSteeringAngleSpacing` = 1.5°
- `mDiscardDistance` = 0.001

Die zweite Parametrisierung soll längere Steuervorgaben und größere Lenkwinkeländerungen bewirken. Im Vergleich zur grundlegenden Parametrisierung (siehe Ab-

schnitt 4.1.1) wurden dazu dieselben Parameter in folgender Weise variiert:

- `mControlDistance` = 6.0
- `mMaxSteeringAngleVariation` = 6.0°
- `mSteeringAngleSpacing` = 20.0°
- `mDiscardDistance` = 0.3

Beobachtung

Die Ergebnisse sind in Abbildung 4.6 dargestellt.

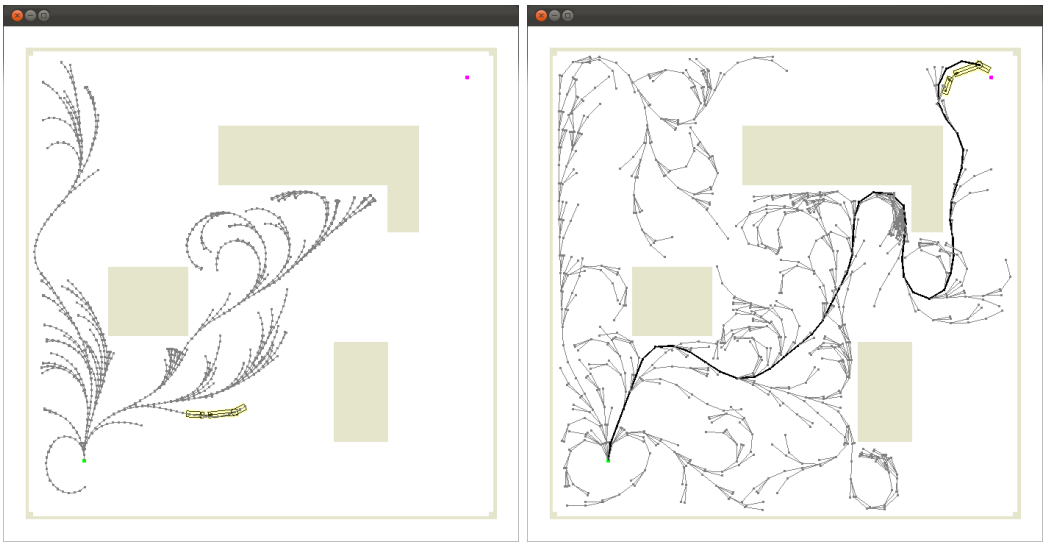

Abbildung 4.6: Ergebnisse des Experiments „Steuervorgaben“ mit kurzen Distanzen und kleinen Lenkwinkeländerungen (links) sowie weiten Distanzen und großen Lenkwinkeländerungen (rechts).

Mit den kurzen Steuervorgaben (Abbildung 4.6 links) wurde die Suche nach über 10 Sekunden manuell abgebrochen. In dieser Zeit wurde das Ziel nicht erreicht. Des Weiteren konnte beobachtet werden, dass der Kartenraum nur sehr langsam exploriert wurde.

Mit den langen Steuervorgaben (Abbildung 4.6 rechts) wurde das Ziel nach 1,2 Se-

kunden und 865 Knoten erreicht. Der Pfad zum Ziel hat eine Länge von 300, was 50 (langen) Steuervorgaben entspricht.

Eine Bewertung der Ergebnisse findet im folgenden Kapitel in Abschnitt 5.1.4 statt.

5 Evaluation

In diesem Kapitel sollen die im vorherigen Kapitel 4 vorgestellten Experimente ausgewertet werden. Dazu soll zunächst auf allgemeine Beobachtungen eingegangen werden. Anschließend sollen die im Einzelnen durchgeführten Experimente ausgewertet werden.

5.1 Auswertung der durchgeführten Experimente

Im Folgenden werden die durchgeführten Experimente ausgewertet. Zunächst soll jedoch eine allgemeine Bewertung der durchgeführten Experimente stattfinden.

Bei der Durchführung der Experimente konnte beobachtet werden, dass das Verfahren — wie zu erwarten — den Kartenraum schnell und gleichmäßig exploriert. Bei mehreren Durchläufen mit derselben Parametrisierung konnte beobachtet werden, dass die grundsätzliche Exploration teilweise ähnlich war, teilweise aber auch sehr unterschiedlich. Erhebliche Unterschiede konnten im Hinblick auf das Erreichen des Ziels festgestellt werden. Das Ziel konnte bei den meisten Durchläufen in wenigen Sekunden erreicht werden. Auf Karten mit hoher Hindernisdichte ist es aber auch mehrfach vorgekommen, dass das Ziel erst nach knapp einer Minute erreicht wurde oder die Suche sogar stagniert hat. In diesem Fall gab es keinen Knoten mehr, der noch Steuervorgaben anbieten konnte.

Des Weiteren konnte beobachtet werden, dass sich viele Knoten des Suchbaums am Rand der Hindernisse konzentrieren. Das ist damit zu erklären, dass auch Sampling-

Werte aus dem Bereich der Hindernisse erzeugt werden und die daraus resultierenden Knoten dann entsprechend am Rand der Hindernisse liegen. Vielleicht kann man diese Erkenntnis nutzen, um eine weitere Sampling-Strategie zu entwickeln, die keine oder weniger Sampling-Werte aus dem Bereich der Hindernisse erzeugt.

5.1.1 Auswertung des Experiments „Rastergröße“

Bei diesem Experiment sollte untersucht werden, wie sich die Wahl der Rastergröße für die Belegungsmatrizen von Fahrzeug und Hindernissen auf die Performanz der Suche auswirkt. Das entsprechende Experiment ist in Abschnitt 4.1.2 beschrieben.

Grundsätzlich konnte festgestellt werden, dass die Rastergröße zwar die grafische Darstellung bremst, nicht jedoch die allgemeine Exploration. Bei der Entwicklung des Raster-basierten Ansatzes im Rahmen dieser Masterarbeit wurden Benchmarks durchgeführt. Dabei konnten auf Basis ähnlich großer Belegungsmatrizen knapp 20000 Kollisionstests pro Sekunde durchgeführten werden. Bei den durchgeführten Experimenten konnten wenige hundert Knoten pro Sekunde erzeugt werden. Es handelt sich dabei zwar lediglich um die erfolgreich durchgeführten Kollisionstests, letztlich kann der Vergleich zwischen den Benchmarks und der im Verfahren erzielten Geschwindigkeit aber so bewertet werden, dass die Kollisionserkennung den *RRT* nicht bremst. Die durchgeführten Experimente zur Rastergröße lassen zudem darauf schließen, dass die Wahl der Rastergröße kaum Auswirkungen auf die Geschwindigkeit hat. Somit sind sehr genaue Rastergrößen möglich.

5.1.2 Auswertung des Experiments „Rückwärtsfahrt“

Bei diesem Experiment sollte untersucht werden, wie sich die Miteinbeziehung der Rückwärtsfahrt auf die Exploration und den entstehenden Suchbaum auswirkt. Dabei sollten die Vor– und Nachteile gegenüber einer Planung mit ausschließlich vorwärts

gerichteten Bewegungen herausgestellt werden. Das entsprechende Experiment ist in Abschnitt 4.1.3 beschrieben.

Von einer Miteinbeziehung der Rückwärtsfahrt erhofft man sich eine bessere Steuerbarkeit an engen, verwinkelten Stellen. Die Rückwärtsfahrt kann außerdem dazu dienen, aus Sackgassen wieder herauszufahren oder Zustände zu erreichen, die durch Vorwärtsfahrt alleine nicht erreichbar sind.

In den Experimenten konnten sich die erhofften Vorteile der Rückwärtsfahrt nicht direkt bestätigen. Zudem konnte beobachtet werden, dass die Exploration deutlich langsamer verläuft, da das Fahrzeug häufig mehrfach auf der Stelle vor und zurück fährt. Aus der Miteinbeziehung der Rückwärtsfahrt resultieren außerdem deutlich schlechtere Pfade: Die Pfade sind aufgrund des Vor– und Zurückfahrens auf der Stelle deutlich länger und aufgrund der Richtungswechsel schwerer zu fahren.

Aus den genannten Gründen erscheint die Miteinbeziehung der Rückwärtsfahrt im Hinblick auf die allgemeine Exploration wenig hilfreich. Allerdings ist sie in bestimmten Situationen sicherlich notwendig, wenn Ziele durch die Vorwärtsfahrt alleine nicht erreicht werden können. Somit wird an dieser Stelle empfohlen, die Rückwärtsfahrt möglichst zu vermeiden und nur in entsprechenden Situationen darauf zurückzugreifen.

5.1.3 Auswertung des Experiments „Zielzustand als Sampling-Wert“

Bei diesem Experiment sollte untersucht werden, wie sich die in Abschnitt 3.4.4 vorgestellte Sampling-Strategie, bei der mit einer festgelegten Wahrscheinlichkeit der Zielzustand als Sampling-Wert verwendet wird, auf die Exploration auswirkt. Dabei sollte insbesondere festgestellt werden, ob das Ziel schneller erreicht wird und welche Vor– oder Nachteile sie außerdem mit sich bringt. Das entsprechende Experiment ist in Abschnitt 4.1.4 beschrieben.

Das Experiment hat im Hinblick auf die genannte Sampling-Strategie gezeigt, dass

das Ziel sehr viel schneller erreicht wird. Sobald der Suchbaum in der Nähe des Ziels angekommen war, wurde das Ziel unmittelbar angesteuert. Allerdings konnten auch einige Nachteile festgestellt werden. So bewirkt die veränderte Sampling-Strategie eine ungleichmäßige Exploration. In diesem Zusammenhang stellt sich die Frage, ob man noch von einem *Rapidly-exploring Random Tree* sprechen kann. Insbesondere konnte eine hohe Konzentration der Knoten an Stellen festgestellt werden, die in der Nähe des Ziels liegen, aber durch Hindernisse vom Ziel abgeschirmt sind. Durch dieses Phänomen wird die Exploration vor allem zu Beginn der Suche etwas gebremst. Insgesamt konnten jedoch bei allen Durchläufen schnellere Ergebnisse erzielt werden.

In jedem Fall ist eine sinnvolle Wahl des Wahrscheinlichkeitswertes für die Wahl des Zielzustands als Sampling-Wert entscheidend. Eine zu groß gewählte Wahrscheinlichkeit führt dazu, dass die Exploration deutlich verlangsamt wird und unter Umständen in Sackgassen hängen bleibt. Zu kleine Wahrscheinlichkeitswerte haben dagegen gar keinen Effekt mehr. Die durchgeführten Experimente legen nahe, dass Werte zwischen 5% und 20% einen sinnvollen Kompromiss darstellen. Unter Umständen könnten Verbesserungen erzielt werden, indem man den Wert während der Suche anpasst. Dies könnte in zukünftigen Experimenten untersucht werden.

5.1.4 Auswertung des Experiments „Steuervorgaben“

Bei diesem Experiment sollte untersucht werden, wie sich die in Abschnitt 3.4.5 vorgestellte Parametrisierung zur Generierung und Auswahl der Steuervorgaben auf die Exploration auswirkt. Dabei sollte herausgefunden werden, welche Distanzen für die Steuervorgaben sinnvoll sind und wie sich verschiedene Werte für die Variation des Lenkwinkels sowohl auf die Exploration als auch auf die Pfade auswirken. Das entsprechende Experiment ist in Abschnitt 4.1.5 beschrieben.

Das Experiment hat gezeigt, dass man durch größere Distanzen und Lenkwinkeländerungen eine deutlich schnellere Exploration und Zielfindung erreicht. Das lässt

vermuten, dass die Steuervorgaben dementsprechend zu wählen sind. Allerdings bringen diese längeren Planungsschritte mehrere Probleme mit sich. Die Distanzen der Steuervorgaben dürfen beispielsweise nicht zu groß gewählt werden, da die Kollisionserkennung sonst keine Kollisionsfreiheit mehr garantieren kann. Die Kollisionserkennung wird nämlich lediglich am Ende einer Bewegung, also an der resultierenden Position, durchgeführt. Auch die Lenkwinkeländerungen können nicht beliebig groß gewählt werden. Eine starke Lenkwinkeländerung ist in der Realität nicht wie geplant umsetzbar. Zudem stellt sie ein Sicherheitsrisiko für den Fahrer und das Fahrzeug dar.

Bezüglich der Generierung und Auswahl der Steuervorgaben reichen die im Rahmen der Arbeit durchgeführten Experimente nicht aus, um aussagekräftige Schlussfolgerungen zu treffen. An dieser Stelle sind weitergehende Experimente notwendig. Insbesondere muss dabei auch der Parameter `mDiscardDistance` näher untersucht werden. Wie am Anfang des Abschnittes 3.4 beschrieben, kann die Wahl dieses Parameters den Aufbau des Baumes blockieren. Insgesamt muss die durch diesen Parameter realisierte Funktionalität überdacht werden. Vielleicht lässt sich die grundsätzliche Idee dahinter auf andere Weise realisieren.

6 Zusammenfassung

Das Verfahren der *Rapidly-exploring Random Trees* konnte erfolgreich realisiert und auf allgemeine Gliederfahrzeuge (*General-n-Trailer*) angewendet werden. Die entwickelte Funktionsbibliothek zur Realisierung des Verfahrens ist modular aufgebaut und einfach erweiterbar. So können im Hinblick auf das Pfadplanungsverfahren weitere Strategien und Heuristiken ohne viel Aufwand ergänzt werden. Zudem kann man auch im Rahmen anderer Anwendungen und Entwicklungen auf die einzelnen Komponenten der Funktionsbibliothek zurückgreifen.

Auf Basis der entwickelten Funktionsbibliothek konnten erste Experimente durchgeführt werden. Damit konnten erste qualitative Ergebnisse im Hinblick auf die Parametrisierung des Verfahrens gewonnen werden. Leider war es aus zeitlichen Gründen im Rahmen dieser Masterarbeit nicht mehr möglich, umfassende Experimente mit systematischer Herangehensweise durchzuführen. Die bisher erzielten Ergebnisse bieten aber eine erste Grundlage für weitere Experimente im Rahmen zukünftiger Forschung.

Literaturverzeichnis

[Altafini 2001] Altafini, C.: Some properties of the general n-trailer. In: *International Journal of Control* 74 (2001), Nr. 4, S. 409–424

[LaValle 1998] LaValle, S. M.: Rapidly-exploring random trees: A new tool for path planning. (1998)

[LaValle u. Kuffner 2000] LaValle, Steven M. ; Kuffner, James J. Jr.: Rapidly-exploring Random Trees: Progress and Prospects / Iowa State University, Ames, IA, USA und University of Tokyo, Bunkyo-ku, Tokyo, Japan. 2000. – Forschungsbericht

[Lindemann u. LaValle 2005] Lindemann, Stephen R. ; LaValle, Steven M.: Current Issues in Sampling-Based Motion Planning / Dept. of Computer Science, University of Illinois. Urbana, IL 61801 USA, 2005. – Forschungsbericht

[Masehian u. Sedighizadeh 2007] Masehian, E. ; Sedighizadeh, D.: Classic and Heuristic Approaches in Robot Motion Planning – A Chronological Review. In: *Engineering and Technology* World Academy of Science, 2007, S. 101–106

[Mostafavi u. a. 2003] Mostafavi, Mir A. ; Gold, Christopher ; Dakowicz, Maciej: Delete and insert operations in Voronoi/Delaunay methods and applications. In: *Computers and Geosciences* Bd. 29. Geomatics Deparment, Laval University, Quebec City, Quebec GI K 7P4, Canada und Department of Geo-Informatics, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong, 2003, S. 523–530

[Schwarz 2009] Schwarz, Christian: *Entwicklung eines Regelungsverfahrens zur*

Pfadverfolgung für ein Modellfahrzeug mit einachsigem Anhänger, Universität Koblenz-Landau, Diplomarbeit, Februar 2009

[Zöbel 2013] Zöbel, Dieter: *Mathematical Modeling of the Kinematics of Vehicles.* 2013. – unveröffentlicht