Analyse und Entwicklung einer HMI zur Visualisierung der Fahrzeugumgebung mit Fokus auf die 3D-Kartendarstellung

Jan Hoppe

Bachelor-Arbeit

Hochschule Hannover
Fakultät IV – Wirtschaft und Informatik
Studiengang B. Sc. Angewandte Informatik

31. 08. 2017

Autor

Jan Hoppe
Matrikelnummer: 1270563
E-Mail: mail@jan-hoppe.de

Erstprüfer

Prof. Dr. Volker Ahlers
Hochschule Hannover
Fakultät IV, Abteilung Informatik
Ricklinger Stadtweg 120
30459 Hannover

Zweitprüfer

Henrik Werner
IAV GmbH
Rockwellstrasse 3
38518 Gifhorn

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die eingereichte Bachelor-Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ort, Datum Unterschrift

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**
2.1 Autonomes Fahren . . . 3
2.1.1 Klassifizierungsstufen . . . 3
2.1.2 Der Stand der Technik und das Straßenverkehrsrecht . . . 4
2.1.3 Navigation autonomer Fahrzeuge . . . 6
2.2 Datenquelle für die Visualisierung der Fahrzeugumgebung . . . 10
2.2.1 Datenquellen für statische Daten . . . 10
2.2.2 Datenquellen für dynamische Daten . . . 12
2.2.3 Nutzen für den Prototyp . . . 12
2.3 HD-Karten am Beispiel Here HD Live Maps . . . 13
2.3.1 DataStore . . . 15
2.3.2 Katalog . . . 15
2.3.3 Layer . . . 15
2.3.4 Partition (oder auch Here-Tile) . . . 15
2.3.5 Quadkey berechnen . . . 16
2.3.6 Here-Tile-ID aus Quadkey berechnen . . . 17

3 Anforderungen **19**
3.1 Funktionale Anforderungen . . . 19
3.1.1 UseCase: Visualisierung betrachten . . . 19
3.1.2 Eigene Position in der Umgebung bestimmen . . . 23
3.1.3 Eigene Orientierung in der Umgebung bestimmen . . . 25
3.2 Nichtfunktionale Anforderungen . . . 27
3.2.1 Primäre Nicht-Funktionale Anforderungen . . . 27
3.2.2 Sekundäre Nicht-Funktionale Anforderungen . . . 29

4 Entwurf I: Architektur **31**
4.1 Komponenten des Prototyps . . . 31
4.2 Verteilung der Komponenten und externe Beziehungen . . . 32

5 Entwurf II: Die Komponenten im Detail **33**
5.1 Environment . . . 33
5.1.1 Schnittstellenspezifikationen . . . 34
5.1.2 Daten-Klassen . . . 36
5.1.3 Interaktionen . . . 40
5.2 GeoReference . . . 46
5.2.1 Distanz berechnen . . . 47

5.2.2 Koordinate konvertieren . . . 48
5.3 GPS . . . 49
5.3.1 Abfrage der Position . . . 50
5.3.2 Abfrage der Orientierung . . . 50
5.4 RoadsMapProvider . . . 51
5.4.1 HereHDProvider . . . 51
5.4.2 HereTileTask . . . 52
5.4.3 Abfrage der Fahrbahngeometrie-Daten . . . 53

6 Implementierung **57**
6.1 Probleme während der Implementierung . . . 57
6.1.1 Here HD Live Map Integration . . . 57
6.1.2 GPS Integration . . . 57
6.1.3 Service-Verbindungstyp . . . 58
6.1.4 Fahrbahngeometrien . . . 58
6.2 Code-Details . . . 60
6.2.1 Projektstruktur . . . 60
6.2.2 Parametrisches einfügen zusätzlicher Koordinaten . . . 61
6.2.3 Service-Implementierung . . . 64
6.2.4 Neue Komponenten einbinden . . . 66

7 Ergebnis **69**
7.1 Funktionale Anforderungen . . . 69
7.2 Nichtfunktionale Anforderungen . . . 69

8 Fazit **71**

A Weiterführende Informationen zu Here HD Live Maps **73**
A.1 Daten über die REST-API abrufen . . . 73
A.1.1 Kataloge . . . 73
A.1.2 Layer . . . 75
A.1.3 Partitionen . . . 77
A.2 Google ProtocolBuffer . . . 78
A.2.1 Schema-Dateien herunterladen . . . 79
A.2.2 Schema-Dateien kompilieren . . . 81
A.3 Die Datenstruktur des Lane-Topology-Layer . . . 81
A.3.1 Ver-/Entschlüsselung von Längen- und Breitengrad . . . 82
A.3.2 Offset-Kodierung von Koordinaten . . . 83

Literaturverzeichnis **85**

1 Einleitung

Autonom fahrende Fahrzeuge gehören aktuell zu den Kernthemen der Automobilindustrie und die Entwicklung der hierfür benötigten Hard- und Softwaresysteme geht stetig voran. Seit einiger Zeit werden bereits Fahrzeuge mit assistierenden Funktionen wie z.B. *Messung des Abstand zum Vordermann und Anpassung bei Unterschreitung des Mindestabstand* und teilautomatisierten Funktionen wie z.B. *automatisches parken* angeboten. Technisch gehen die Möglichkeiten bereits ein Stück weiter, die notwendigen Richtlinien und Gesetze in den verschiedenen Ländern erlauben aber derzeit noch nicht den Einsatz sämtlicher autonomer Funktionen. Speziell Funktionen, welche dem Fahrer die komplette Steuerung des Fahrzeugs abnehmen sind vielerorts noch rechtlich umstritten. Unter anderem besteht aktuell eine Forderung dahingehend, dass der Fahrer immer die volle Kontrolle über das Fahrzeug übernehmen kann, falls er dies für notwendig hält.

Woher weiß der Fahrer aber, dass er die Steuerung wieder übernehmen muss? Im einfachsten Fall erkennt das System, dass es mit einer Fahr-Situation nicht umgehen kann und meldet dies dem Fahrer frühzeitig durch auslösen eines Signals. Was aber, wenn z.B. die Sensoren ein Objekt in der Umgebung nicht erfassen konnten? Das System kann in diesem Fall sein Umfeld nicht richtig bewerten und somit auch nicht auf eine mögliche Gefahr reagieren bzw. dem Fahrer signalisieren. Der Fahrer benötigt somit eine Möglichkeit mit dessen Hilfe er selbstständig erkennen kann, wann er die Kontrolle über das Fahrzeug wieder übernehmen muss.

Die vorliegende Bachelorarbeit widmet sich genau dieser Fragestellung und versucht eine Lösung für dieses Problem aufzuzeigen. Die grundlegende Idee ist hierbei dem Fahrer eine dreidimensionale Karte seiner Umgebung anzuzeigen. Da die Sensoren nur eine begrenzte Reichweite besitzen, werden zusätzlich statische Kartendaten von anderen Quellen für die Visualisierung von Straßen und Gebäuden herangezogen. Die so erstellte Umgebungskarte wird dann, um die vom Fahrzeug erfassten dynamischen Objekte, ergänzt und dem Fahrer präsentiert. Anhand der Umgebungskarte kann der Fahrer dann überwachen, wie das System die Umgebung wahrnimmt. Hierdurch erhält der Fahrer eine Möglichkeit zu überprüfen, ob das System in der Lage ist eine Situation richtig zu bewerten und erlaubt es ihm somit rechtzeitig die Kontrolle über das Fahrzeug zu übernehmen.

Neben dem rein praktischen Zweck kann die Visualisierung der Fahrzeugumgebung auch einen positiven psychologischen Effekt auf einen Fahrer haben, welcher den automatisierten Fahrfunktionen nicht vertraut. Dieser kann sich einstellen, wenn der Fahrer über eine längeren Zeitraum die Visualisierung verfolgt und sieht das alle für den Straßenverkehr relevanten Objekte vom System richtig erkannt werden.

Bereits diese kurze Definition der Fragestellung zieht eine ganze Reihe weiterer Fragen mit sich:

- Welche Probleme können auftreten, wenn man versucht statische Kartendaten mit dynamischen Sensordaten zu verknüpfen?

- Wie können die von den Sensoren gewonnen Daten sinnvoll mit statischen Kartendaten kombiniert werden, um so ein größeres Umfeld für den Fahrer zu visualisieren?
- Liefern statische Kartendaten genug Informationen, um daraus eine dreidimensionale Umgebung zu gestalten?
- Sollten die statischen Karten nicht genug Informationen liefern, ist es dann möglich die fehlenden Informationen aus anderen Quellen zu beziehen?
- Liefern die statischen Datenquellen die Informationen in ausreichender Detailtiefe?
- Wie sind die dynamisch erkannten Objekte zu markieren, damit der Fahrer erkennen kann was tatsächlich erkannt wurde.
- Welche anderen Möglichkeiten existieren, um die Reichweite der Umgebungskarte zu erhöhen bzw. detaillierter darzustellen?

Da die vollständige Ausarbeitung einer Lösung für dieses Problem den Rahmen einer Bachelorarbeit bei weitem übersteigt, konzentriert sich diese Arbeit darauf eine solide Grundbasis zu schaffen. Anstelle eines Systems zur Visualisierung einer vollständigen Umgebungskarte, wird ein prototypisches System entwickelt, welches zunächst ausschließlich statische Objekte (z.B. Häuser und Straßen) dreidimensional visualisiert. Um in späteren Arbeiten auch dynamische Objekte in die Umgebung einfügen zu können, wird das System mit einer Schnittstelle ausgerüstet, welche es erlaubt beliebige Objekte in der Umgebung zu platzieren und anzuzeigen. Neben der reinen Visualisierung der Umgebung soll das System später auch weitere Informationen anzeigen können, wie z.B. Navigationsdaten oder Points of Interest (POI). Daher muss das System flexibel um weitere Use-Cases erweiterbar sein. Da es sich bei der Entwicklung um einen Prototyp handelt, welcher stetig weiterentwickelt werden soll, muss das System wartungsfreundlich gestaltet werden. Dazu wird das System komponentenbasiert strukturiert, so dass einzelne Komponenten leicht überarbeitet oder komplett ausgetauscht werden können.

2 Grundlagen

Dieses Kapitel widmet sich einigen grundlegenden Informationen zum autonomen Fahren und legt keinen Anspruch auf Vollständigkeit. Das Ziel dieses Kapitels ist es somit dem Leser einen schnellen und leichten Einstieg in das Thema zu geben, ohne dabei zu tief in den Details zu versinken.

2.1 Autonomes Fahren

Der Begriff autonomes Fahren ist seit einigen Jahren in aller Munde und wird als die größte Revolution der Fahrzeugentwicklung, seit der Erfindung des Automobils, betrachtet. Aber was hat es mit dem Begriff eigentlich auf sich? Vollständig autonome Fahrzeuge benötigen keinen Fahrer mehr und können vollkommen selbstständig durch den Straßenverkehr navigieren, sowie auf jede Gefahrensituationen reagieren. Dabei spielt es auch keine Rolle, ob sich das Fahrzeug durch den Stadtverkehr oder über die Autobahn bewegt. Der Fahrer, welcher in einem autonomen Fahrzeug nur noch Fahrgast ist, muss lediglich ein Fahrziel auswählen zu welchem das Fahrzeug fahren soll.
Von diesem Entwicklungsstadium ist die aktuelle Technik noch ein Stück entfernt, wobei die Forschung und Entwicklung rasend voran geht. Die ersten Serienreifen autonomen Fahrzeuge werden allerdings nicht vor dem Jahr 2030 erwartet [1]. Bei den derzeit erhältlichen Fahrzeugen handelt es sich dagegen um automatisierte Fahrzeuge. Automatisierte Fahrzeuge können im Gegensatz zu autonomen Fahrzeugen nur in bestimmten Situationen die Kontrolle übernehmen. Die verschiedenen Klassifizierungsstufen der Autonomie werden im folgenden Unterabschnitt erläutert.

2.1.1 Klassifizierungsstufen

Der Verband der Automobilindustrie (VDA) definiert zur Klassifizierung der Entwicklungsphasen für das autonome Fahren ein 6-stufiges System (wie auch in Abbildung 2.1 zu sehen).

Definition 2.1: Klassifizierungsstufen [2]

Stufe 0 (Driver only):
Der Fahrer hat die volle und ausschließliche Kontrolle über das Fahrzeug (Längs*- und Querführung**), kann aber durch das System auf bestimmte Situationen durch Signale aufmerksam gemacht werden. Zum Beispiel, um das Verlassen der Spur oder zu dichtes Auffahren zu signalisieren.

Stufe 1 (Assistenz):
Das System assistiert dem Fahrer in bestimmten Situationen, der Fahrer muss aber immer die Kontrolle über einen Teil der Steuerung übernehmen (Längs- oder Querführung). Der Spurhalteassistent, kann z.B. die Querführung übernehmen um das Fahrzeug in der Spur zu halten, hat aber keinen Einfluss auf die Längsführung.

Stufe 2 (Teilautomatisiert):

Mit der Teilautomatisierung kann der Fahrer die komplette Kontrolle des Fahrzeug (Längs- und Querführung) in bestimmten Situationen an das System übergeben. Als Beispiel sei die Parkautomatik genannt, welche nur bei Stillstand des Fahrzeugs aktiviert werden kann. Der Fahrer muss in dieser Stufe der Automation das Fahrzeug und die Umgebung immer überwachen und falls nötig die Steuerung selbstständig übernehmen.

Stufe 3 (Hochautomatisiert):
Wie Stufe 2, nur dass das System nun selbstständig erkennen kann, ob es mit einer Situation umgehen kann oder nicht. Der Fahrer braucht das Fahrzeug und die Umgebung somit nicht mehr aktiv überwachen, während das System die Kontrolle über die Steuerung hat. Allerdings muss der Fahrer weiterhin innerhalb einer gegebenen Vorwarnzeit die Kontrolle übernehmen können, sollte er vom System dazu aufgefordert werden.

Stufe 4 (Vollautomatisiert):
Das System kann vollständige Fahrten in bestimmten Anwendungsfällen selbstständig durchführen. Als Beispiel dient hier häufig das sogenannte Valet-Parking. Bei dem Valet-Parking sucht sich das Fahrzeug selbstständig einen Parkplatz (z.B. in einem Parkhaus), während der Fahrer sich bereits nicht mehr im Fahrzeug befindet. Die Vollautomatisierten Fahrten sind weiterhin an bestimmte Voraussetzungen gebunden, wie z.B. Straßentyp, Höchstgeschwindigkeit und Fahrumgebung.

Stufe 5 (Autonom - Fahrerlos):
Das Fahrzeug kann vollständig in jeder Situation und auf beliebigen Straßentypen autonom agieren, ein Fahrer ist nicht mehr erforderlich (auch nicht als Backup).

*Längsführung = beschleunigen/bremsen **Querführung = lenken

2.1.2 Der Stand der Technik und das Straßenverkehrsrecht

Aktuelle Serienfahrzeuge verfügen heutzutage höchstens über den Autonomiegrad Stufe 2 (Teilautomatisiert), wobei der Übergang zu Stufe 3 gerade im Gange ist [3]. Dies liegt allerdings nicht nur an den technischen Möglichkeiten, sondern ist auch den rechtlichen Rahmenbedingungen geschuldet [4]. Unter anderem ist Deutschland an den völkerrechtlichen Vertrag, dem *Wiener Übereinkommen über den Straßenverkehr* aus dem Jahre 1968, gebunden [5]. Dieser besagte bis zum Jahr 2016 unter anderem :

> ***ARTIKEL 8***
> ***Lenker (Führer von Tieren)***
> *1. Jedes Fahrzeug und miteinander verbundene Fahrzeuge müssen, wenn sie in Bewegung sind, einen Lenker haben.*
> ...
> *5. Jeder Lenker (Führer von Tieren) muß dauernd sein Fahrzeug beherrschen oder seine Tiere führen können.* [5]

Mit Artikel 8 Absatz 1 wurde zwingend ein Lenker (also ein menschlicher Fahrer) vorgeschrieben und somit widersprachen autonome Fahrzeuge dem geltenden Recht. Artikel 8 Absatz 5 hingegen betont den Umstand, dass der Lenker dauerhaft sein Fahrzeug beherrschen, d.h. Längs- und Querführung

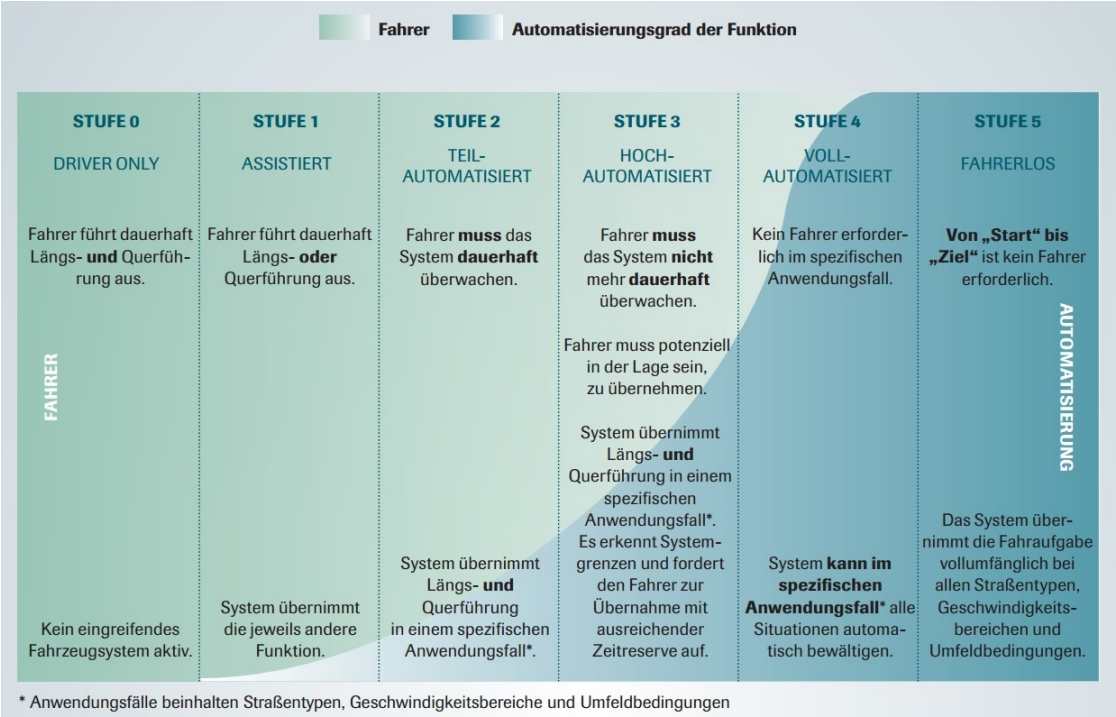

Abbildung 2.1: Übersicht über die Automatisierungsgrade des automatisierten Fahren [2]

kontrollieren muss. Damit waren sämtliche Assistenz- und Automatisierungsfunktionen, welche dem Fahrer die Kontrolle abnehmen (sei es auch nur vorübergehend), ebenfalls nicht erlaubt.
Um dem technischen Fortschritt Folge zu tragen und das assistierte und automatisierte Fahren rechtlich zu ermöglichen, wurde somit im März 2014 ein entsprechender Änderungsantrag [6] zum *Wiener Übereinkommen über den Straßenverkehr* eingebracht, welcher im März 2016 in Kraft getreten ist.

Änderung des Artikels 8:
Nach Absatz 5 wird folgender Absatz 5bis angefügt:
„(5bis) Fahrzeugsysteme, die einen Einfluss auf das Führen des Fahrzeugs haben, gelten als vereinbar mit Absatz 5 und Artikel 13 Absatz 1, wenn sie den Bedingungen für den Bau, den Einbau und die Verwendung nach den internationalen Rechtsinstrumenten betreffend Radfahrzeuge, Ausrüstungsgegenstände und Teile, die in Radfahrzeuge(n) eingebaut und/oder verwendet werden können, entsprechen[1]; Fahrzeugsysteme, die einen Einfluss auf das Führen eines Fahrzeugs haben und die nicht den genannten Bedingungen für den Bau, den Einbau und die Verwendung entsprechen, gelten als vereinbar mit Absatz 5 und Artikel 13 Absatz 1, wenn diese Systeme vom Führer übersteuert oder abgeschaltet werden können.“

Änderung des Artikels 39:
Dem Satz
„(1) Jedes Kraftfahrzeug (Artikel 1 Buchstabe p), jeder Anhänger und alle miteinander verbundenen Fahrzeuge im internationalen Verkehr müssen dem Anhang 5 entsprechen. Sie müssen ferner betriebssicher sein.“

wird folgender Satz angefügt:
„Sind in diese Fahrzeuge Systeme, Teile und Ausrüstungsgegenstände eingebaut, die den Bedingungen für die Konstruktion, den Einbau und die Verwendung nach den in Artikel 8 Absatz 5bis genannten internationalen Rechtsinstrumenten entsprechen, so gelten sie als vereinbar mit Anhang 5.“ [6]

Nach dem Inkrafttreten der Änderungen am Wiener Vertrag, konnte im Mai 2016 ein Gesetzesentwurf [7] eingereicht werden, um diese Änderungen in entsprechendes deutsches Recht zu überführen. Der Gesetzesentwurf wurde unverändert akzeptiert und das Gesetz am 29.09.2016 beschlossen [8]. Erst seit diesem Tag sind sämtliche assistierenden und teilautomatisierten Funktionen auf deutschen Straßen rechtlich erlaubt.
Seit dem 12.05.2017 dürfen auch Fahrzeuge bis zum Autonomiegrad der Stufe 4 (Vollautomatisiert) am Straßenverkehr teilnehmen. Bis diese Fahrzeuge auf den Straßen zu sehen sind wird allerdings noch etwas Zeit vergehen. Zunächst sollen bis etwa 2020 die ersten Fahrzeuge der Stufe 3 (Hochautomatisiert) die Serienreife erlangen, wobei die Termine je nach Hersteller etwas schwanken.

> *Vollautomatisierte Autos könnten bald zum Alltag auf deutschen Straßen gehören. Der Bundesrat hat am 12.05.2017 einem Gesetzesbeschluss des Bundestages vom 30.04.2017 zugestimmt, der die Nutzung dieser Technik regelt. Danach sind künftig auch solche Fahrzeuge zugelassen, bei denen das technische System für eine bestimmte Zeit oder in bestimmten Situationen die Steuerung übernehmen kann. Die letzte Verantwortung verbleibe allerdings beim Menschen.* [9]

Die rechtlichen Regelungen für komplett autonom fahrende Fahrzeuge werden sich vermutlich noch etwas länger hinziehen, da hier der Fahrer unter Umständen keinerlei Einfluss mehr auf das Fahrzeug hat und somit Haftungs- und Ethikfragen stärker in den Vordergrund rücken [10].

Die Abbildung 2.2 zeigt eine Schätzung der VDA für die Einführung verschiedener automatisierter Fahr- und Parkfunktionen. Die erste Einführung einer Funktion auf einer Stufe entspricht auch dem Übergang zu dem jeweiligen Autonomiegrad.

2.1.3 Navigation autonomer Fahrzeuge

Die grundlegende Navigation eines autonomen Fahrzeugs folgt einem einfachen Prinzip. Zuerst muss das autonome Fahrzeug seine exakte Position in der Welt bestimmen und benötigt dann ein hochpräzise Karte mit deren Hilfe das Fahrzeug entlang eines Pfad über die Straßen navigiert werden kann. Dabei scannt die Fahrzeugsensorik die Umgebung kontinuierlich ein und vergleicht die eingescannten Daten mit denen der Karte. Sämtliche Abweichungen zwischen der Karte und dem Scan (z.B. Fußgänger auf der Straße oder andere Fahrzeuge) werden dann als das aktuelle Verkehrsgeschehen interpretiert. Anhand der Abweichungen kann das autonome Fahrzeug dann entscheiden, wie mit der Situation umzugehen ist.
Das hier kurz beschrieben Verfahren nennt sich SLAM (Simultaneous Localization and Mapping). Das S in SLAM bezieht dabei darauf, dass die erfassten Daten (Position und Umgebungsscan) nicht nur einmalig verwendet, sondern auch gleichzeitig genutzt werden, um die Straßenkarte zu aktualisieren bzw. zu verbessern. Das SLAM-Verfahren wird in [10], in Bezug auf das autonome Fahren, etwas genauer erläutert. Neben dem beschrieben Verfahren existieren auch noch weitere Ansätze, welche

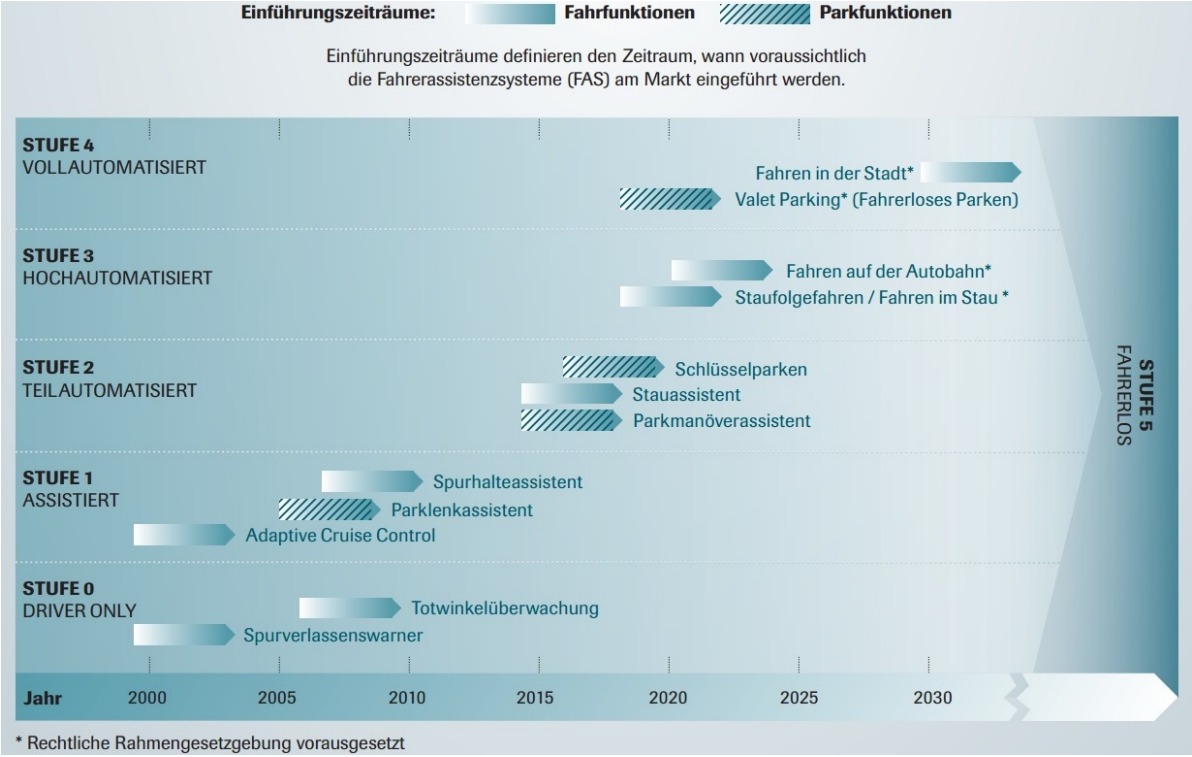

Abbildung 2.2: Übersicht über die Einführung automatisierter Fahr- und Parkfunktionen [2]

verstärkt auf die Sensorik des Fahrzeug und weniger auf die Kartendaten setzen. Das grundlegende Vorgehen bleibt jedoch gleich.

Die Probleme beginnen allerdings schon bei der exakten Ortsbestimmung, aktuelle GPS-Systeme haben eine Genauigkeit im Meterbereich und sind somit für eine exakte Navigation im Straßenverkehr nicht genau genug [11] [10]. Zusätzlich kann das GPS-Signal durch verschiedene Faktoren negativ beeinflusst werden, z.B. Fahrten durch Schluchten bei denen das Signal von den umgebenden Wänden reflektiert und so verfälscht wird oder auch Tunnel bei denen das GPS-Signal gar nicht mehr empfangen werden kann. Daher sucht die Industrie noch immer nach einem geeigneten Verfahren, welches die exakte Ortsbestimmung in allen Situationen ermöglicht.

Definition 2.2: Ortsbestimmungsverfahren

DGPS (Differential Global Positioning System):

Bei dem DGPS werden die GPS-Koordinaten mit Hilfe von Differenzwerten korrigiert. Die Differenzwerte müssen dazu von ortsfesten Referenzstationen regelmäßig ausgesendet werden. Auf diese Weise erhält das GPS neben den eigentlichen Daten zusätzliche Korrekturwerte und kann so eine Position bis auf den Millimeter genau bestimmen. Allerdings ist hierzu ein gut ausgebautes DGPS-Netzwerk mit entsprechend vielen Referenzstationen notwendig [12].

Landmarken:

Eine Landmarke ist ein Objekt welches fest im Raum steht und einen markanten Punkt bietet, in der Schifffahrt werden hierzu z.B. Leuchttürme oder Gebirgszüge verwendet. Zur Ortsbestimmung auf einer Straße können dies z.B. Verkehrsschilder, Ampeln und Fahrbahnmarkierungen, sowie deren genaue Position zueinander sein. Um die genaue Position mittels Landmarken zu bestimmen, wird die Position zunächst mit einem herkömmlichen GPS auf einen Radius von einigen Meter eingegrenzt. Erst danach wird die genaue Position mit Hilfe der Landmarken bestimmt. Hierzu wird wiederum eine Karte benötigt, welche die Landmarken mit deren exakten Positionen verzeichnet. Auch hier wird wieder ein GPS-Signal benötigt und man steht vor demselben Problem wie zuvor. Falls die Karte der Landmarken aber genügend Daten enthält, könnte bereits ein einmaliges GPS-Signal zu Beginn genügen, um anschließend anhand der Geschwindigkeit und der verzeichneten Landmarken unabhängig vom GPS zu navigieren. Das Problem bei diesem Verfahren besteht u.a. darin, genügend feste und unveränderliche Landmarken zu finden und zu katalogisieren, bzw. diese bei Veränderungen immer auf den aktuellen Stand zu halten. Aber selbst wenn diese Problem gelöst werden kann, ist es immer möglich das Landmarken wie z.B. Schilder von anderen Objekten verdeckt werden, man stelle sich dazu einfach eine von starkem Schneefall bedeckte Landschaft vor [13] [14].

Bodenradar:

Bei dem Bodenradarverfahren wird der Boden unterhalb der Straße per Radar eingescannt und die so gewonnenen Daten anschließend mit einer detaillierten Karte des Untergrunds abgeglichen. Auf diese Weise ist es bereits möglich, bei etwa 100km/h, ein Fahrzeug mit einer Genauigkeit von 4-5cm zu lokalisieren. Das Verfahren ist im Prinzip eine Variation des Landmarkenverfahren, nur dass die Landmarken sich unter der Erde befinden und selten ändern (z.B. bei baulichen Maßnahmen). Selbst Schnee und Eis ist bei diesem Verfahren kein Problem, dafür benötigt die Hardware mit Abmessungen von 150 x 60 x 15cm viel Platz und ist nur für größere Fahrzeuge mit genügend Bodenfreiheit geeignet [15][16].

Kann ein Fahrzeug nun seine exakte auf den Dezimeter genaue Position bestimmen, benötigt es als nächstes, Informationen darüber, wie es von einem Punkt A zu einem Punkt B gelangen kann. Diese Informationen müssen ebenfalls wieder auf den Dezimeter genau sein, was zur Folge hat das herkömmliche Karten zu ungenau sind [17]. Die Lösung dieses Problems ist naheliegend und durch eine exakte Vermessung aller Straßen zu bewältigen. Die Neuvermessung ist allerdings mit einem hohen Aufwand und Kosten verbunden. Zudem genügt es nicht, die Straßen nur einmal alle paar Jahre oder Monate neu zu vermessen, sondern die Vermessung muss kontinuierlich durchgeführt werden [18] [19].

Dieser Aufgabe haben sich diverse Navigationsdatenanbieter (z.B. TomTom, Here, Google) schon angenommen, indem sie die benötigten Systeme zur Verwaltung der Kartendaten entwerfen und beginnen mit Daten zu füllen. Die nun entstehenden hochdetaillierten Karten (HD-Karten) umfassen neben dem dezimetergenauen Verlauf einer Straße auch weitere Informationen, wie z.B. Geschwindigkeitsbegrenzungen oder Straßenschilder und Fahrbahnmarkierungen, welche auch als Landmarken verwendet werden [11]. Bis sämtliche Straßen durch Sensorfahrzeuge neu vermessen sind wird noch ein wenig Zeit vergehen und eine kontinuierliche Neuvermessung wird erst Einzug erhalten, wenn genügend Serienfahrzeuge mit entsprechenden Sensoren ausgestattet auf den Straßen unterwegs sind.

Sind die genaue Position und die Route bekannt, kommen die von der Sensorik erfassten Daten zum Tragen. Die Sensorik scannt die gesamte Fahrzeugumgebung kontinuierlich ein und wertet die Daten aus, um Hindernisse (Gegenstände, Personen, andere Fahrzeuge usw.) auf und neben der Fahrbahn zu erkennen. Mit Hilfe der erfassten dynamischen Daten ist das autonome Fahrzeug in der Lage auf die aktuelle Verkehrssituation zu reagieren, z.B. um bei einer roten Ampel das Fahrzeug zu stoppen. Um ein möglichst detailliertes Abbild der Fahrzeugumgebung zu erhalten, verfügt ein autonomes Fahrzeug über ein Reihe verschiedener Sensoren (Ultraschall, Radar, Laserscanner (Lidar), Kamera).

Definition 2.3: Sensorsysteme [20]

Kamera:
Kameras werden in erster Linie zur Erkennung von Straßenschildern, Fahrbahnmarkierungen u.ä. eingesetzt und besitzen eine relativ hohe Reichweite im Vergleich zu den anderen Sensoren. Durch den Einsatz mehrerer Kameras in einem Fahrzeug kann Stereoskopie zum Einsatz kommen, um die Entfernung zu einem beliebigen Punkt im Bild zu messen.

Ultraschall:
Ultraschallsensoren werden zur Erkennung von Objekten im Nahbereich eingesetzt. Dabei wirkt dieser Sensortyp im Bereich von etwa 30cm bis 8m (der genaue Wirkungsbereich ist vom Modell abhängig) und kann dort Objekte bis auf eine Genauigkeit von wenigen Millimetern erkennen.

Radar:
Die Radarsensoren haben die gleiche Aufgabe wie die Ultraschallsensoren, allerdings verfügen diese Sensoren über einen Wirkungsbereich von etwa 50cm bis 200m und decken somit einen wesentlich größeren Bereich ab.

Lidar (Light Detection and Ranging):
Das Lidar ist im Prinzip ein Laserscanner, welcher die Umgebung mit mehreren Laserstrahlen erfasst. Der große Vorteil dieser Technologie ist, dass sie es ermöglicht auch Konturen von Objekten zu erkennen und nicht nur, ob sich ein Objekt in der Umgebung befindet und welchen Abstand dieses hat. Der Wirkungsbereich liegt bei etwa 150m.
Anhand der erfassten Konturpunkte, kann anschließend ein Softwaresystemen Strukturen erkennen und mit diesen dreidimensionale Modelle berechnen. Ein aktuelles Beispiel eines Lidar-Scans zeigt die Abbildung 2.3.

Der in der Abbildung 2.3 dargestellte Lidar-Scan kommt einem Foto zwar sehr nahe, tatsächlich handelt es sich aber um mehrere Millionen einzelne Scanpunkte. Diese bilden eine Punktwolke und bieten noch keinerlei Informationen darüber, um was für ein Objekte es sich handelt (wie z.B. Fahrzeuge, Gebäude, Sctraßenschilder). Zu jedem der Punkte ist die Lage in dem dreidimensional vermessenen Raum und somit auch die Entfernung zum Sensor bekannt.

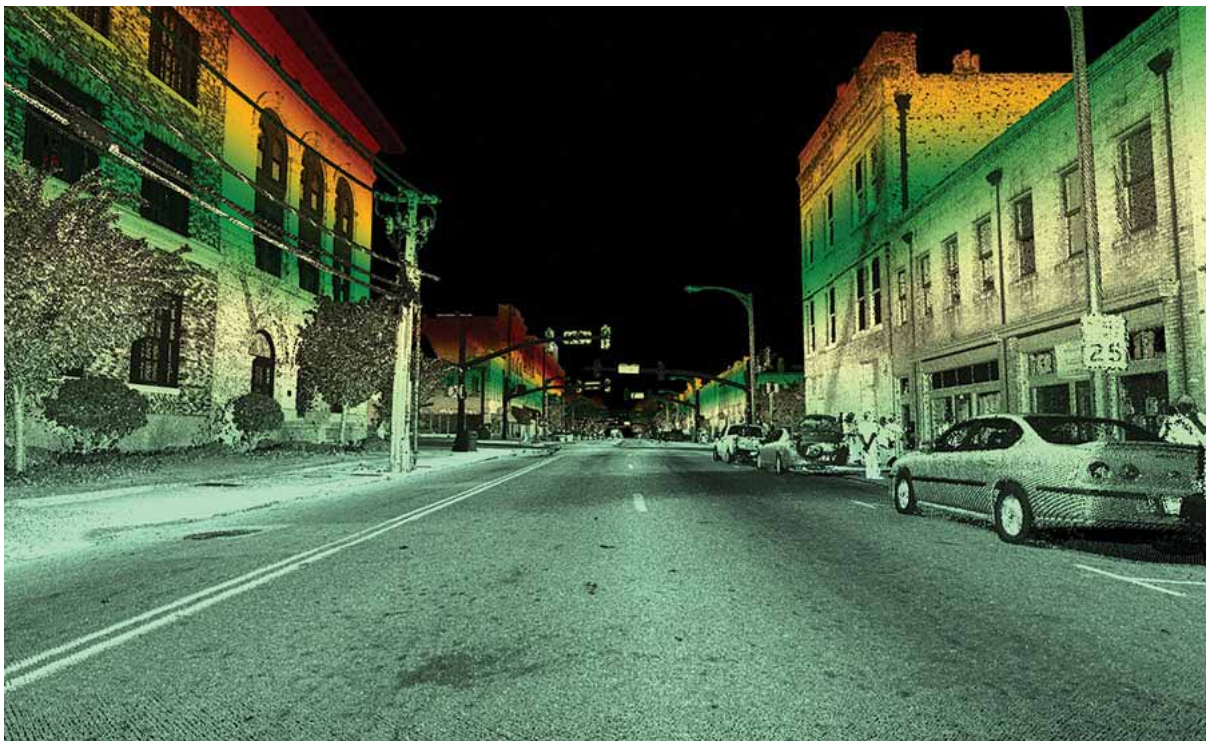

Abbildung 2.3: Aktueller Lidar-Scan [21]

2.2 Datenquelle für die Visualisierung der Fahrzeugumgebung

Ziel dieser Ausarbeitung ist es eine dreidimensionale Umgebung für ein Fahrzeug zu erstellen, welche sich nicht nur auf das von der Fahrzeugsensorik erfasste Umfeld beschränkt, sondern darüber hinausgeht. Die erweiterte Anzeige soll es dem Fahrer ermöglichen, seine eigene Position besser zu lokalisieren. Um eine solche Visualisierung zu erstellen werden 3D-Modelle aller darzustellenden Objekte benötigt, sowie deren exakte Position. Die Frage die sich hier stellt ist, woher die Daten für die 3D-Modelle bezogen werden können. Prinzipiell lassen sich die infrage kommenden Datenquellen in zwei Kategorien einteilen, zum einen in die Datenquellen für statische Daten und zum anderen in die Datenquellen für dynamische Daten.

2.2.1 Datenquellen für statische Daten

Als Quellen für statische Objekte bieten sich nur digital verfügbare Karten an, da diese die Umwelt flächendeckend erfassen und bereits in einem maschinenlesbaren Format zur Verarbeitung vorliegen. Zu den digitalen Karten werden hier vor allem auch sämtliche GIS (Geoinformationssysteme) gezählt, da diese zu den reinen Kartendaten auch weiterführende Informationen zu der Struktur der Kartenelemente enthalten können. In der Praxis unterscheidet sich die Genauigkeit der verschiedenen Karten von Anbieter zu Anbieter, eine genaue Angabe ist daher hier nicht möglich. Zusätzlich erschwert sich eine Angabe der Genauigkeit dadurch, dass die Karten immer weiterentwickelt werden und es so auch von Region zu Region, zu unterschiedlichen Genauigkeiten bei einem Anbieter kommen kann. Die

Genauigkeit herkömmlicher Karten liegt im allgemeinen im Meterbereich (am Beispiel Google Earth [22]). Die Unterschiede werden besonders deutlich, wenn man die hochpräzisen für das autonome Fahren entwickelten HD-Karten, mit den herkömmlichen Karten (hier im weiteren als Standard-Karten bezeichnet) vergleicht. Abgesehen von der Genauigkeit unterscheiden sich die HD-Karten von den Standard-Karten in weiteren Punkten, so dass die beiden Typen in den folgenden Unterabschnitten getrennt betrachtet werden. Bevor die verschiedenen Kartentypen genauer betrachtet werden, erfolgt ein kurzer Exkurs zur Bedeutung der Genauigkeit in der Kartenkunde.

Genauigkeit in der Kartenkunde [23]

In der Kartenkunde existiert der Begriff Genauigkeit in zwei Abstraktionen, der absoluten und der relativen Genauigkeit. Die absolute Genauigkeit gibt an, wie exakt ein Punkt auf einer Karte mit seiner tatsächlichen geodätischen Koordinate übereinstimmt. Die relative Genauigkeit beschreibt hingegen, wie genau die Lage zweier Koordinaten zueinander ist. Das heißt zwei auf einer Karte gemessene Koordinaten, können bei gleicher absoluter Abweichung, eine relative Genauigkeit von einhundert Prozent besitzen.

Standard-Karten

Der Haupteinsatzzweck von Standard-Karten ist es, dem menschlichen Nutzer die Orientierung in einer unbekannten Umgebung zu ermöglichen. Dabei spielt die absolute Genauigkeit gegenüber der relativen Genauigkeit eine eher untergeordnete Rolle. Denn der Mensch orientiert sich stärker an der räumlichen Lage der Objekte, als an deren absoluten Koordinaten. So ist es für den Nutzer einer Straßenkarte eines beliebigen Orts unerheblich, ob die absolute Positionierung gegenüber dem geodätischen Koordinatensystem um einige Meter abweicht. Lediglich die relative Position der Straßen und Gebäude zueinander sind für den Betrachter der Straßenkarte relevant und werden für die Orientierung genutzt.

HD-Karten

Die HD-Karten haben einen anderen Einsatzzweck als die Standard-Karten. Im Gegensatz zu den Standard-Karten, werden HD-Karten für die Orientierung und Navigation von Maschinen entwickelt, wobei die Entwicklung autonomer Fahrzeuge den Hauptantrieb gegeben hat. Da die HD-Karten für Maschinen entwickelt werden, wundert es nicht dass diese auch nur maschinell lesbar sind und derzeit keine grafische Aufbereitung für den Menschen bieten.

> *Es handelt sich um eine reine Maschinenkarte, die für das menschliche Auge nicht lesbar ist, sagt Alex Mangan, Produktmanager bei Here. Sie bündelt mehr Informationen, als ein Fahrer jemals verarbeiten könnte.* [24]

Anders als der Mensch benötigen autonome Fahrzeuge eine möglichst exakte Daten, da diese ihre Position anhand von GPS-Koordinate ermitteln und damit ihre Position auf der Karte bestimmen [25]. Da aktuelle GPS-Geräte noch zu ungenau sind um eine exakte Positionsbestimmung zu bieten, verfügen HD-Karten über eine spezielle Daten-Schicht, welche Informationen zu Landmarken bereitstellt. Mit Hilfe der Landmarken kann das Fahrzeug dann seine auf den Zentimeter genaue Position bestimmen.
Neben den Topologie- und Geometrie-Daten verwalten HD-Karten auch zusätzliche Meta-Informationen, die für das autonome Fahren von Bedeutung sind. Als Beispiel können die erlaubte Geschwindigkeit, Fahrtrichtung der Spur und Fahrbahnmarkierungen genannt werden.

2.2.2 Datenquellen für dynamische Daten

Als Datenquellen für die dynamischen Daten können sämtliche im Fahrzeug verbauten Sensorik-Systeme verwendet werden. Die dynamischen Daten werden in dieser Arbeit nicht weiter betrachtet und sind erst Thema einer späteren Ausarbeitung.

2.2.3 Nutzen für den Prototyp

Die Visualisierung der Fahrzeugumgebung soll dem Fahrer darstellen, welche Informationen das autonome Fahrzeug über die Umgebung besitzt. Da das Fahrzeug auf statische Kartendaten zurückgreift, kennt das System detaillierte Informationen über mehrere Kilometer entfernte Orte. Dies erlaubt es, das Straßennetz und alle weiteren Daten aus dem HD-Kartenmaterial in die Visualisierung mit einfließen zu lassen. Die Größe des zu visualisierenden Bereichs um das Fahrzeug herum, kann dabei theoretisch frei gewählt werden. Dabei ist aber zu beachten, dass die von den Sensoren erfasste Umgebung den Schwerpunkt bilden und das der Fahrer in der Lage sein soll, alle visualisierten Objekte in diesem Bereich zu identifizieren. Der zu wählende Maßstab ist dabei unter anderem von der Größe und Auflösung des Displays des verwendeten Zielgerät abhängig.
Theoretisch ist es möglich neben dem HD-Kartenmaterial auch Daten aus den Standard-Karten in die Visualisierung mit einfließen zu lassen. Hier ist allerdings die absolute Genauigkeit der Standard-Karten ein Problem. Die absolute Genauigkeit der Standard-Karten liegt im Meterbereich. Bei der Überführung der Koordinaten aus einer Standard-Karte in eine HD-Karte, kann es somit zu Verschiebungen von mehreren Metern kommen. Die Skizze in Abbildung 2.4 soll dieses Problem verdeutlichen.

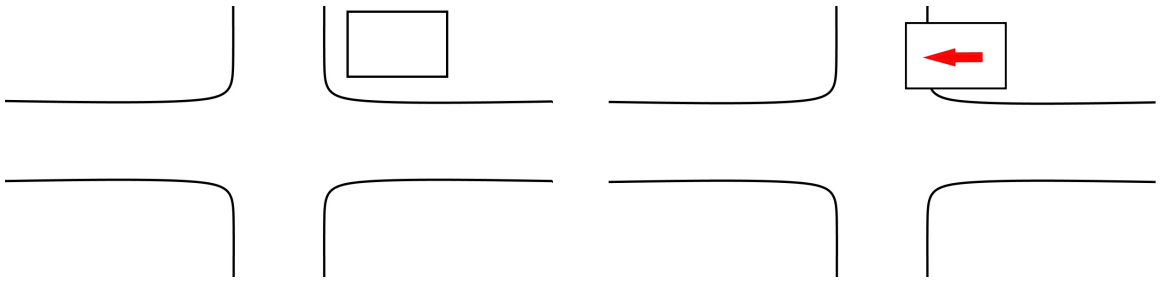

Abbildung 2.4: Links Lage in Standard-Karte, rechts Lage nach Transformation in eine HD-Karte

Damit sind die Standard-Karten, ohne entsprechende Algorithmen für die Transformation der absoluten Koordinaten, unbrauchbar. Ein möglicher Ansatz wäre es, in beiden Karten-Systemen gemeinsame Referenzpunkte zu finden und mit Hilfe dieser zu versuchen, die Objekte anhand der relativen Position zu übertragen. Eine exakte Positionierung in der HD-Karte ist mit dieser Methode aber nicht zu erwarten und wird in dieser Arbeit nicht weiter verfolgt.

2.3 HD-Karten am Beispiel Here HD Live Maps

Anmerkung: *Dieser Abschnitt soll nur einen ersten groben Überblick über Here HD Live Maps liefern, mit einem Hauptfokus auf die in diesem Projekt verwendeten Verfahren und benötigten Informationen. Für einen vollständigen und detaillierteren Einblick sind die Dokumentationen des Anbieters [26] [27] in jeglicher Hinsicht vorzuziehen. Sämtliche Informationen in diesem Abschnitt stammen aus den zuvor genannten Dokumentationen.*

Here HD Live Maps ist ein speziell für das autonome Fahren entwickelter Service zum Abruf von hochgradig detaillierten Fahrbahninformationen. Neben einer auf den Zentimeter genauen Beschreibung der linken und rechten Fahrbahnbegrenzung, sowie deren Verlauf und Lage werden unter anderem auch Metadaten zu der Fahrbahn und den einzelnen Spuren der Fahrbahn (z.B. Anzahl Spuren, Fahrbahnmarkierungen, Tempolimits, Fahrtrichtung) geliefert. Die Abbildung 2.5 zeigt eine Visualisierung der Fahrbahnen mit einem Teil der bereitgestellten Daten.

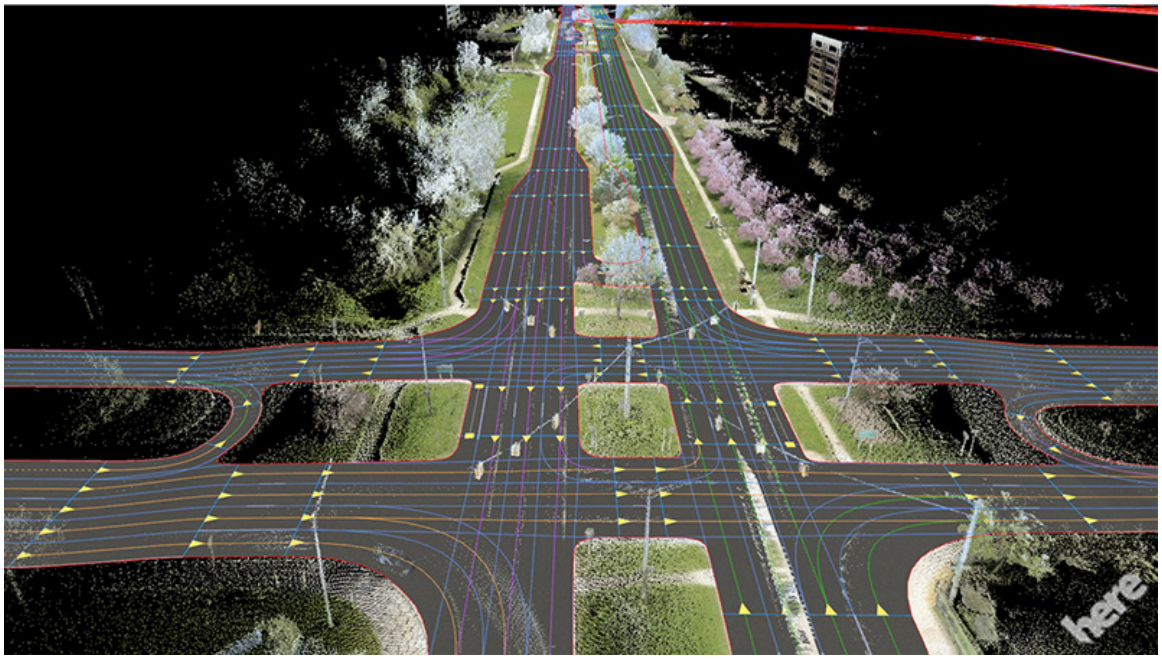

Abbildung 2.5: Die Abbildung zeigt eine Visualisierung der Here HD Live Map-Daten. Die außerhalb der Fahrbahnen dargestellten Objekte sind nicht Bestandteil der Here HD Live Map (Bildquelle: [28]).

Erläuterung der in der Abbildung dargestellten Elemente:

- **Gelbe Dreiecke:** Fahrtrichtung.
- **Gelbe Vierecke:** Start- bzw. Endpunkt einer nicht befahrbaren Spur.
- **Rote Linien:** Linke bzw. rechte Begrenzung der Fahrbahn.
- **Blaue Linien:** Linke bzw. rechte Begrenzung einer Fahrspur.
- **Graue Linien:** Nicht befahrbare Spur.
- **Gelbe, grüne, violette und türkise Linien:** Mitte der jeweilige Fahrspur.

Wie das Wort Live im Namen bereits andeutet, werden die Daten nicht nur einmalig eingescannt und verteilt, sondern jedes mit einem entsprechenden Sensorik-System ausgerüstete Fahrzeug, kann die Daten der Here HD Live Map aktualisieren und so andere Verkehrsteilnehmer über die derzeitige Fahrsituation informieren (z.B. Gegenstände auf der Fahrbahn, Staus, Unfälle, Umleitungen, Glatteis, ...). Here HD Live Maps setzt somit nicht nur auf eine hoch detaillierte Vermessung der Fahrbahnen, sondern baut zusätzlich auf die Schwarmintelligenz der vernetzten Fahrzeuge (in Abbildung 2.6 bildhaft dargestellt).
All diese Informationen werden für die Steuerung und Routenplanung eines autonomen Fahrzeugs bereitgestellt, können aber mit entsprechendem Aufwand auch in die Visualisierung der Fahrzeugumgebung mit einfließen. So kann z.B. ein Hinweis auf einen Unfall angezeigt werden, um den Fahrer, über die Abweichung von einer sonst bekannten Route zu informieren.

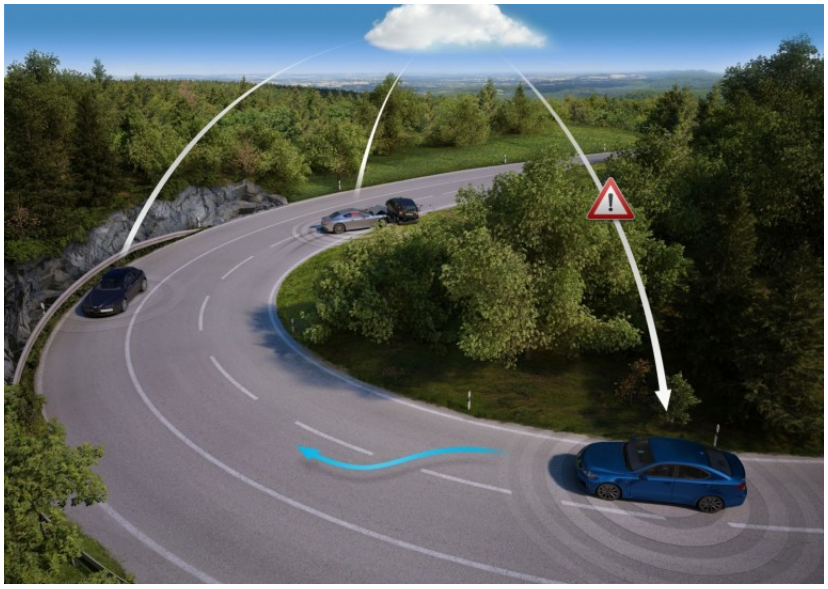

Abbildung 2.6: Visualisierung einer Live Unfallmeldung (Bildquelle: [29]).

Wie bereits eingehend erwähnt, wird Here HD Live Maps speziell für das autonome Fahren entwickelt und enthält somit keine für das autonome Fahren irrelevanten Informationen. Allerdings befindet sich das System derzeit noch im Aufbau und somit kann sich dieser Umstand in der Zukunft jederzeit ändern.

2.3.1 DataStore

Ein DataStore bietet mittels REST-API die verfügbaren Daten zum Abruf an. Die Daten im DataStore werden dabei in verschiedenen Katalogen kategorisiert, siehe dazu die Abbildung 2.7.

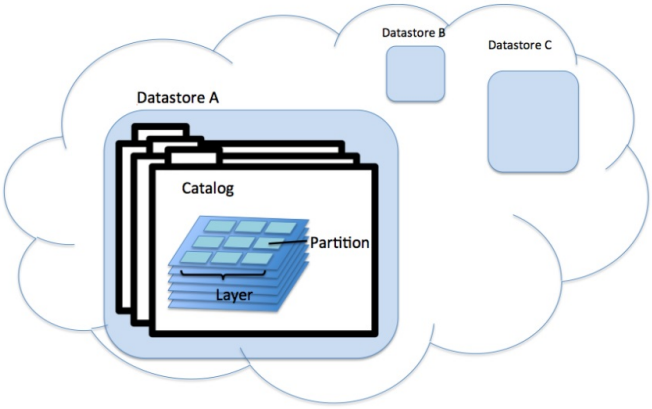

Abbildung 2.7: Graphische Darstellung der Here HD Live Map Datenorganisation im DataStore (Bildquelle: [28]).

2.3.2 Katalog

Jeder Katalog enthält Daten für einen spezifischen Bereich, wie z.B. Live Roads (Sperrungen, Staus, usw.), Live Incidents (aktuelle Unfälle) oder HD Maps (hochgradig detaillierte Fahrbahninformationen).

2.3.3 Layer

Die Daten in jedem Katalog werden in mehrere Layer unterteilt. Im Falle des HD Maps Katalog sind dies u.a. Layer für die strukturelle Beschreibung (z.B. Topology, Geometry) und mehrere Layer für Metadaten (z.B. Geschwindigkeitsbeschränkungen, Fahrbahnmarkierungen). Da es eher unwahrscheinlich ist, dass alle Daten eines Layers gleichzeitig benötigt werden, wird jeder Layer (vor allem aus Gründen hinsichtlich der Performance und des Speicherbedarfs) in kleinere Partitionen unterteilt.

2.3.4 Partition (oder auch Here-Tile)

Eine Partition wird auch als Here-Tile bezeichnet und beinhaltet die spezifischen Daten eines Kartenausschnitts in einer bestimmten Größe, begrenzt durch je zwei Längen- und Breitengrade. Die Größe des Kartenausschnitts ist dabei abhängig vom gewählten Zoomlevel. Mit jedem Zoomlevel wird eine Partition in vier gleichgroße Partitionen unterteilt und somit um Faktor vier herangezoomt.

Die Abbildung 2.8 zeigt links die Partitionierung bei Zoomlevel 1 und rechts die Partitionierung bei Zoomlevel 2. Die Ziffernfolge in den jeweiligen Partitionen bezeichnet man als Quadkeys. Wie zu sehen ist, benötigt man auf Zoomlevel 1 nur einen einstelligen Quadkey und bei Zoomlevel 2 einen zweistelligen Quadkey. Dieses Schema setzt sich bei weiterer Partitionierung fort, d.h. das Zoomlevel

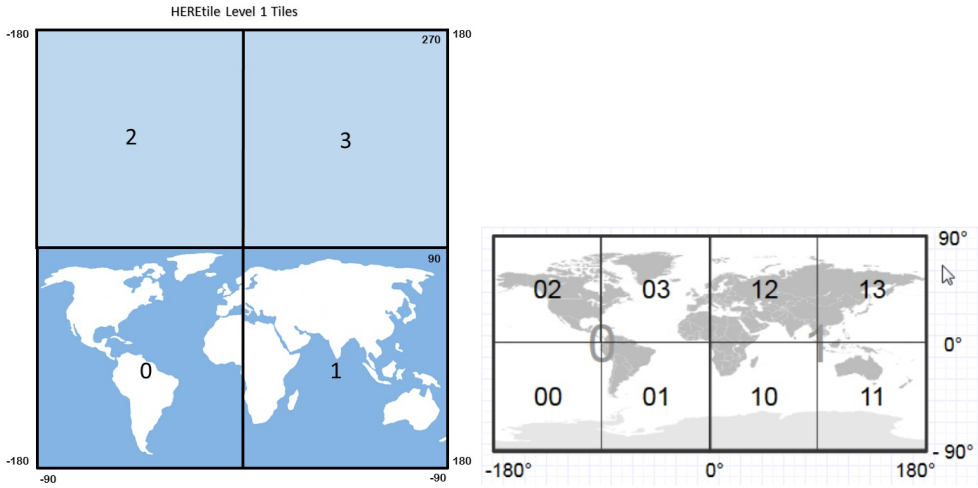

Abbildung 2.8: Links: Weltkarte bei Zoomlevel 1 (Bildquelle: [27]). Rechts: Weltkarte bei Zoomlevel 2 (Bildquelle: [26]).

enspricht der Länge des Quadkeys. Das Quadkey-Schema sieht vor, das jede Partition einen eigenen Key nach folgendem Schema erhält:

- Beginnend bei Zoomlevel 1 wird jeder Partition eine Ziffer zwischen 0 und 3 zugewiesen. Für die Nummerierung gilt immer SW = 0, SO = 1, NW = 2 und NO = 3.
  Anmerkung: *Bei Zoomlevel 1 enthalten die Partitionen 2 und 3 keine Daten, dies gilt folglich auch für dessen Subpartitionen.*
- Betrachtet man die Partition 0 auf Zoomlevel 2, kann man sehen wie die Unterteilung in weitere Subpartition durchgeführt wird. Jede Partition wird einfach in vier gleichgroße Subpartitionen unterteilt und wie zuvor nummeriert (SW nach NO). Da es die Partitionen 0 bis 3 schon gibt, wird zur Unterscheidung der Quadkey der übergeordneten Partition vorangestellt und so die Quadkeys 00, 01, 02 und 03 für die Subpartitionen gebildet.

Hinweis 2.1:

- Die Partitionierung basiert auf **nicht** projizierten WGS84 Längen- und Breitengraden
- Die Partitionen sind nicht quadratisch, sondern es handelt sich um Ausschnitte aus dem WGS84-Elipsoid entlang der jeweiligen Längen- und Breitengrade
- Die Aussage das alle Here-Tiles gleichgroß sind, bezieht sich auf die Angabe in Grad und nicht in Meter
- Aktuell werden Here-Tiles der Here HD Live Map nur mit Zoomlevel 14 angeboten

2.3.5 Quadkey berechnen

Die Bestimmung des Here-Tiles zu einem Längen-/Breitengrad ist nach der oben genannten Vorgehensweise für die automatisierte Verarbeitung eher unpraktisch. Das vorliegende Schema erlaubt allerdings die Berechnung des Here-Tiles zu einer Koordinate in wenigen Schritten.

Beispiel 2.1: Quadkey berechnen

Schritt 1: Größe eines Here-Tile in Grad

Zunächst benötigt man die Größe eines Here-Tiles in Grad für das gewünschte Zoomlevel. Diese lässt sich einfach berechnen, indem man 360° durch die Anzahl der Here-Tiles pro "Zeile" bzw. "Spalte" dividiert.

- $360°/2^{zoomlevel}$ = *Längen-/Breitengrad pro Here-Tile*
- Beispiel für *zoomlevel* = 14:
  Längen-/Breitengrad pro Here-Tile = $360°/2^{14}$ = 0.02197265625°

Schritt 2: X-/Y-Koordinate

Bestimmung der X-/Y-Koordinate. Hierzu betrachtet man die Weltkarte bei Zoomlevel 1 und legt die südwestlichste Koordinate (-180°/-90°) als Punkt (0/0) fest. Somit ist ...

- $X = \lfloor(180° + Längengrad) / Längengrad\ pro\ Here\text{-}Tile\rfloor$
- Beispiel für Längengrad = 13.36937° und Zoomlevel = 14:
  $X = \lfloor(180° + 13.36937)/0.02197265625°\rfloor = \lfloor 8800,45\rfloor = 8800$
- $Y = \lfloor(90° + Breitengrad) / Breitengrad\ pro\ Here\text{-}Tile\rfloor$
- Beispiel für Breitengrad = 52.52507° und Zoomlevel = 14:
  $Y = \lfloor(90° + 52.52507)/0.02197265625°\rfloor = \lfloor 6486.47\rfloor = 6486$

Schritt 3: Konvertierung in Binärdarstellung

Konvertierung der gerundeten X- und Y-Werte in die Binärdarstellung. Die Anzahl der Binärstellen muss dabei dem Zoomlevel entsprechen, d.h. bei Zoomlevel 14 = 14 Stellen. Dazu muss ggf. von links mit Nullen aufgefüllt werden.

- $X = 8800_{10} = 10\ 0010\ 0110\ 0000_2$
- $Y = 6486_{10} = 01\ 1001\ 0101\ 0110_2$

Schritt 4: Binär verschränken

Binärwerte verschränken, beginnend mit Y_1.

- $Y_{31}X_{31}Y_{30}X_{30}Y_{29}X_{29}...Y_1X_1Y_0X_0$
- $0110\ 1000\ 0110\ 0011\ 0110\ 0010\ 1000_2$

Schritt 5: Konvertierung in Vierer-System

Konvertierung des verschränkten Binärwertes ins Vierer-System (= Quadkey).

- $Quadkey(0110\ 1000\ 0110\ 0011\ 0110\ 0010\ 1000_2) = 12\ 20\ 12\ 03\ 12\ 02\ 20_4$

2.3.6 Here-Tile-ID aus Quadkey berechnen

Here-Tile-IDs werden als 32Bit unsigned Integer gespeichert, somit wird noch eine weitere Umrechnung des Quadkeys benötigt, um die Here-Tile-ID zu erhalten.

Beispiel 2.2: Here-Tile-ID bestimmen

Schritt 1: Prefix hinzufügen

Eine 1 dem Quadkey voranstellen.

- $1Quadkey$
- $1\ 12\ 20\ 12\ 03\ 12\ 02\ 20_4$

Schritt 2: Konvertierung in Dezimalsystem

Durch die Konvertierung des neuen Werts aus dem Vierer-System in das Dezimalsystem erhält man nun die Here-Tile-ID.

- *Here-Tile-ID*$(1\ 12\ 20\ 12\ 03\ 12\ 02\ 20_4) = 377894440_{10}$

Weitere Informationen zum Abruf der Daten über die REST-API und über das für die Rückgabe verwendete Google ProtocollBuffer-Format befinden sich in Anhang A.

3 Anforderungen

Der in dieser Arbeit zu entwickelnde Prototyp legt den Fokus auf die Visualisierung statischer Kartenelemente, im speziellen auf die Darstellung der Fahrbahnen. Im Folgenden werden dazu zunächst die funktionalen und nicht funktionalen Anforderungen definiert.

3.1 Funktionale Anforderungen

Die funktionalen Anforderungen werden im Folgenden anhand einiger UseCase-Diagramme veranschaulicht und zusätzlich durch eine textuelle Ausarbeitung der dargestellten UseCases ergänzt.

3.1.1 UseCase: Visualisierung betrachten

Im Folgenden werden die in Abbildung 3.1 dargestellten UseCases beschrieben.

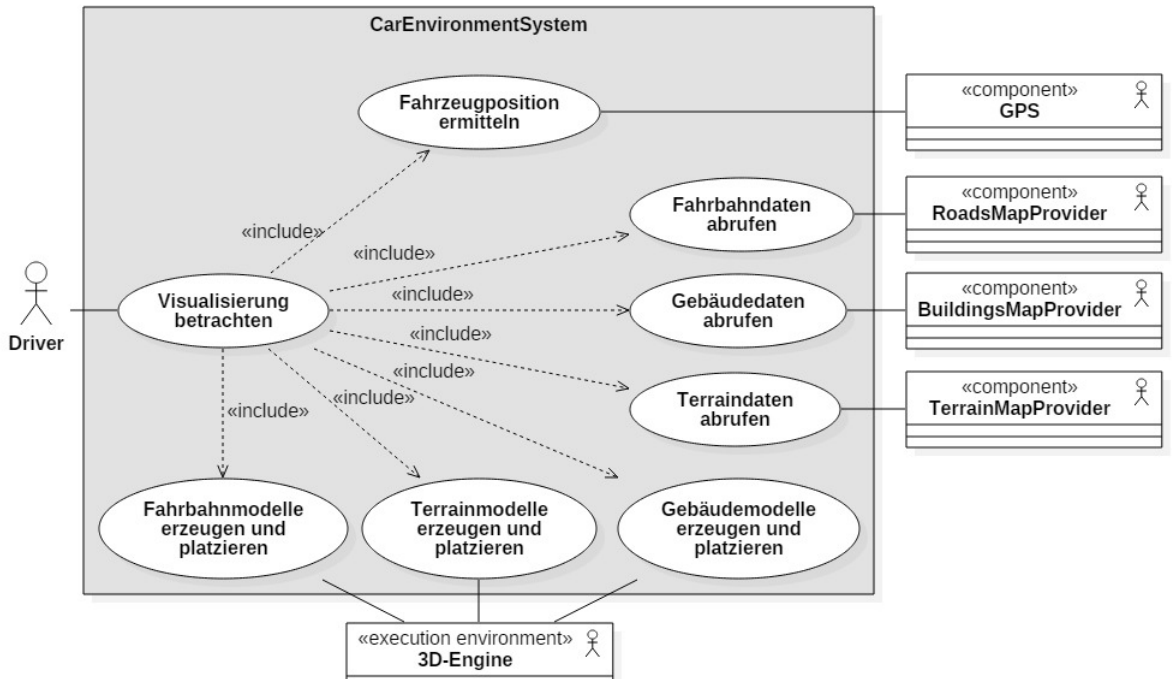

Abbildung 3.1: UseCase: Visualisierung betrachten

Funktionale Anforderung: F1.1
Dem Fahrer wird das visualisierte Fahrzeugumfeld auf einem Display angezeigt. Die Visualisierung ist dabei auf die statischen Objekte (Strassen, Gebäude und Bodenflächen) der Umgebung beschränkt.

UseCase 3.1: Visualisierung betrachten

Aktoren: Driver, GPS, RoadsMapProvider, BuildingsMapProvider, TerrainMapProvider, 3D-Engine

Ziel: Darstellung des Fahrzeugumfelds auf einem Display.

Vorbedingungen:

- Zugriff auf GPS ist möglich
- Zugriff auf alle Kartenanbieter ist möglich oder die Daten der nicht verfügbaren Kartenanbieter liegen offline vor.

Nachbedingungen:

- Alle 3D-Modelle, für den Kartenabschnitt in dem sich die aktuelle Koordinate befindet, wurden erzeugt und in der Szene platziert.

Auslösendes Ereignis: Systemstart oder Aktualisierung des Systems.

Interaktionsfolge:

1. Durch das *Environment-Script* in der 3D-Engine wird die Startroutine aufgerufen bzw. eine Neuberechnung der Umgebung bei der *Environment-Komponente* angestoßen.
2. Die *Environment-Komponente* ruft die aktuelle Position des Fahrzeugs von der *GPS-Komponente* ab. Für Details siehe UseCase **3.2 Fahrzeugposition ermitteln**.
3. Die *Environment-Komponente* ruft die Kartendaten für die aktuelle Position über die *RoadsMapProvider-Komponente, BuildingsMapProvider-Komponente und TerrainMapProvider-Komponente* in beliebiger Reihenfolge ab. Der Abruf der Kartendaten wird in dem UseCase **3.3 Fahrbahndaten abrufen** geschildert, siehe dort auch den Hinweis für die UseCases **Gebäudedaten abrufen** und **Terraindaten abrufen**.
4. Die *Environment-Komponente* erzeugt die 3D-Modelle aus den zuvor abgerufenen Kartendaten und platziert diese in der Szene (in beliebiger Reihenfolge). Der Ablauf der Modellierung und Platzierung der Modelle in der Szene, wird in UseCase **3.4 Fahrbahnmodelle erzeugen und platzieren** beschrieben. Für Informationen zu den UseCases **Gebäudemodelle erzeugen und platzieren** und **Terrainmodelle erzeugen und platzieren**, siehe den Hinweis im UseCase **3.4 Fahrbahnmodelle erzeugen und platzieren**.

Erweiterungen:

2.a Fehler beim Abruf der aktuellen Position.
  2.a1 Die letzte bekannte Position wird verwendet.
  2.a2 Auf das Problem wird in der Visualisierung hingewiesen.

2.b Fehler beim Abruf der aktuellen Position, wenn keine vorherige Position bekannt ist.
  2.b1 Der weitere Ablauf wird unterbrochen, da keine Informationen zur Verfügung stehen um die Umgebung zu visualisieren.
  2.b2 In der Visualisierung wird auf das Problem hingewiesen.

3.a Die aktuelle Koordinate liegt in einem Kartenabschnitt, dessen Fahrbahndaten bereits erfolgreich abgerufen wurden.
  3.a1 Der im UseCase **Fahrbahndaten abrufen** beschriebene Ablauf wird übersprungen.

3.b Die aktuelle Koordinate liegt in einem Kartenabschnitt, dessen Gebäudedaten bereits erfolgreich abgerufen wurden.

3.b1 Der im UseCase **Gebäudedaten abrufen** beschriebene Ablauf wird übersprungen.

3.c Die aktuelle Koordinate liegt in einem Kartenabschnitt, dessen Terraindaten bereits erfolgreich abgerufen wurden.

3.c1 Der im UseCase **Terraindaten abrufen** beschriebene Ablauf wird übersprungen.

4.a Die Fahrbahnmodelle für den aktuellen Kartenabschnitt, wurden bereits erzeugt und platziert.

4.a1 Der im UseCase **Fahrbahnmodelle erzeugen und platzieren** beschriebene Ablauf wird übersprungen.

4.b Die Gebäudemodelle für den aktuellen Kartenabschnitt, wurden bereits erzeugt und platziert.

4.b1 Der im UseCase **Gebäudemodelle erzeugen und platzieren** beschriebene Ablauf wird übersprungen.

4.c Die Terrainmodelle für den aktuellen Kartenabschnitt, wurden bereits erzeugt und platziert.

4.c1 Der im UseCase **Terrainmodelle erzeugen und platzieren** beschriebene Ablauf wird übersprungen.

Funktionale Anforderung F1.2
Die aktuelle Fahrzeugposition wird über ein GPS-System abgerufen.

UseCase 3.2: Fahrzeugposition ermitteln

Aktoren: Driver, GPS
Ziel: Ermittlung der aktuellen Fahrzeugposition.
Vorbedingungen:
- Zugriff auf GPS ist möglich

Nachbedingungen:
- Aktuelle Position (Geokoordinate) ist bekannt.

Auslösendes Ereignis: Systemstart oder Aktualisierung des Systems.
Interaktionsfolge:
1. Die *Environment-Komponente* fragt die aktuelle Position bei der *GPS-Komponente* an.
2. Die *GPS-Komponente* ruft die aktuelle Position von der GPS-Hardware ab.
3. Die *GPS-Komponente* übergibt die aktuelle Position an die *Environment-Komponente*.

Erweiterungen:

2.a Zugriff auf GPS geht verloren.

2.a1 Im Hintergrund wird versucht den Zugriff zurückzuerlangen.

2.a2 Der weitere Ablauf wird unterbrochen und der *Environment-Komponente* wird das Problem signalisiert

Funktionale Anforderung: F1.3
Die Daten für die Konstruktion der Fahrbahngeometrien werden von dem Server des Kartenanbieters abgerufen und anschließend werden die abgerufenen Daten vorverarbeitet. Bei der Vorverarbeitung werden

- die geodätischen Koordinaten in kartesische Koordinaten konvertiert.
- aus den kartesischen Koordinaten wird eine Liste für die Erzeugung der Dreiecksflächen für die 3D-Modelle berechnet.
- zu den kartesischen Koordinaten wird eine Liste der Texturkoordinaten berechnet.

Bei den abzurufenden Daten handelt es sich um den Kartenabschnitt, welche die in F1.2 ermittelte Koordinate beherbergt.

UseCase 3.3: Fahrbahndaten abrufen

Aktoren: Driver, RoadsMapProvider

Ziel: Abruf aller für die Modellierung und Platzierung der Fahrbahnen benötigten Daten, bezogen auf einen Kartenabschnitt. Die Daten werden nach dem Abruf in ein einheitliches auf kartesische Koordinaten basierendes Format umgewandelt, so dass die 3D-Engine diese verarbeiten kann.

Vorbedingungen:

- Zugriff auf den Server des Kartenanbieters ist möglich oder die Daten stehen offline zur Verfügung.
- Die aktuelle Position (Geokoordinate) ist bekannt.

Nachbedingungen:

- Alle für den Kartenabschnitt angefragten Fahrbahndaten stehen zur weiteren Verarbeitung bereit.

Auslösendes Ereignis: Systemstart oder Aktualisierung des Systems.

Interaktionsfolge:

1. Die *Environment-Komponente* fragt die Fahrbahndaten für die aktuelle Koordinate bei der *RoadsMapProvider-Komponente* an.
2. Die *RoadsMapProvider-Komponente* ruft die Fahrbahndaten, für den Kartenabschnitt der aktuellen Koordinate, von dem Server des Kartenanbieter ab.
3. Die *RoadsMapProvider-Komponente* speichert die abgerufenen Daten auf dem lokalen Speicher, so dass diese beim nächsten Abruf offline verfügbar sind.
4. Die *RoadsMapProvider-Komponente* konvertiert die auf Geokoordinaten beruhenden Daten, mit Hilfe der *GeoReference-Komponente*, in ein einheitliches auf kartesische Koordinaten basierendes Format (siehe **5.1.2**). Die Konvertierung erfolgt, damit die *Environment-Komponente* die Daten unabhängig vom verwendeten Kartenanbieter verarbeiten kann.

Erweiterungen:

2.a Die Daten stehen offline zur Verfügung.

2.a1 Die Daten werden aus dem lokalen Speicher gelesen und es erfolgt kein Abruf der Daten über die Server des Kartenanbieter. Da die Daten bereits gespeichert vorliegen, entfällt Schritt 3.

2.b Der Zugriff auf den Server geht verloren

2.b1 Im Hintergrund wird versucht den Zugriff zurückzuerlangen.

2.b2 Der weitere Ablauf wird unterbrochen und der *Environment-Komponente* wird das Problem signalisiert

Hinweis zu den UseCases Gebäudedaten abrufen und Terraindaten abrufen

Der Abruf der Daten für die Gebäude (UseCase **Gebäudedaten abrufen**) und das Terrain (UseCase **Terraindaten abrufen**) erfolgt analog dem Abruf der Fahrbahndaten und wird aus Platzgründen daher nicht in separaten UseCases textuell beschrieben.

Funktionale Anforderung: F1.4
Aus den durch F1.3 bereitgestellten Daten, werden die 3D-Modelle erzeugt und in der Szene platziert.

UseCase 3.4: Fahrbahnmodelle erzeugen und platzieren

Aktoren: Driver, 3D-Engine
Ziel: Die für die Visualisierung benötigten 3D-Modelle erzeugen und in der virtuellen Umgebung platzieren.
Vorbedingungen:
- Die Fahrbahndaten konnten abgerufen werden und liegen in einem auf kartesische Koordinaten basierendem Format vor.

Nachbedingungen:
- Alle 3D-Modelle für die Fahrbahnen konnten erzeugt und in der Szene platziert werden.

Auslösendes Ereignis: Systemstart oder Aktualisierung des Systems.
Interaktionsfolge:
1. Die *Environment-Komponente* erzeugt aus den vorverarbeiteten Fahrbahndaten die 3D-Modelle für die 3D-Engine.
2. Die *Environment-Komponente* platziert die 3D-Modelle in der Szene.

Erweiterungen:
-

Hinweis zu den UseCases Gebäudemodelle erzeugen und platzieren und Terrainmodelle erzeugen und platzieren

Das erzeugen und platzieren der Modelle für die Gebäude (UseCase **Gebäudemodelle erzeugen und platzieren**) und das Terrain (UseCase **Terrainmodelle erzeugen und platzieren**) erfolgt analog dem erzeugen und platzieren der Fahrbahnmodelle und wird aus Platzgründen daher nicht in separaten UseCases textuell beschrieben.

3.1.2 Eigene Position in der Umgebung bestimmen

Im Folgenden werden die in Abbildung 3.2 dargestellten UseCases beschrieben.

Funktionale Anforderung: F2.1
Dem Fahrer wird das Fahrzeugmodell in der Visualisierung angezeigt, wobei die Position des Fahrzeugs auf dem Display immer an der gleichen Stelle erfolgt (die Kamera folgt dem Fahrzeug) und das

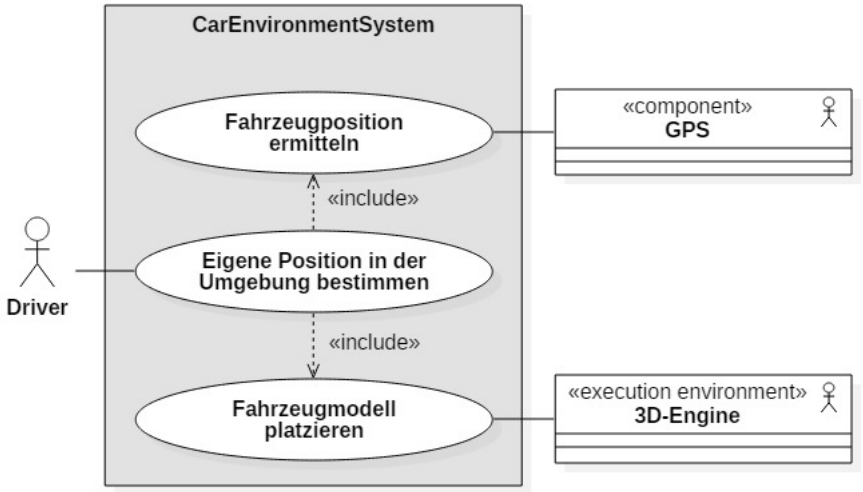

Abbildung 3.2: UseCase: Eigene Position in der Umgebung bestimmen

Fahrzeugmodell entsprechend der realen Koordinate in der visualiserten Umgebung platziert ist.

UseCase 3.5: Eigene Position in der Umgebung bestimmen

Aktoren: Driver, GPS, 3D-Engine

Ziel: Dem Fahrer soll ersichtlich werden, wo genau er sich in der visualisierten Umgebung befindet. Dies wird durch die Platzierung eines 3D-Modells seines Fahrzeugs erreicht.

Vorbedingungen:

- Es ist ein 3D-Modell des Fahrzeugs verfügbar.
- Zugriff auf GPS ist möglich.

Nachbedingungen:

- Das 3D-Modell des Fahrzeugs befindet sich an der aktuellen vom GPS ermittelten Position in der Visualisierung.

Auslösendes Ereignis: Systemstart oder Aktualisierung des Systems.

Interaktionsfolge:

1. Durch das *Environment-Script* in der 3D-Engine wird die Positionierung des Fahrzeugs bei der *Environment-Komponente* angestoßen.
2. Die *Environment-Komponente* ruft die aktuelle Position des Fahrzeugs von der *GPS-Komponente* ab. Für Details siehe UseCase **3.2 Fahrzeugposition ermitteln**.
3. Die *Environment-Komponente* platziert das Fahrzeugmodell in der visualisierten Umgebung bei der aktuellen Position. Der entsprechende Vorgang wird in UseCase **3.6 Fahrzeugmodell platzieren** beschrieben.

Erweiterungen:

2.a Fehler beim Abruf der aktuellen Position.

2.a1 Das Problem wird in der Visualisierung kenntlich gemacht, z.B. indem das Fahrzeug vom Rendervorgang ausgeschlossen wird.

2.a2 Die Ausführung des UseCase **3.6 Fahrzeugmodell platzieren** entfällt, da die Position nicht zur Verfügung steht.

Funktionale Anforderung: F2.2
Das Fahrzeugmodell wird entsprechend der realen Position (ermittelt durch F1.2), in der visualisierten Umgebung platziert.

UseCase 3.6: Fahrzeugmodell platzieren

Aktoren: Driver, 3D-Engine
Ziel: Platzierung des Fahrzeugmodells an der aktuellen Position in der Szene.
Vorbedingungen:
- Es ist ein 3D-Modell des Fahrzeugs verfügbar.
- Die aktuelle Position (Geokoordinate) ist bekannt.

Nachbedingungen:
- Das 3D-Modell des Fahrzeugs befindet sich an der aktuellen vom GPS ermittelten Position in der Visualisierung.

Auslösendes Ereignis: Systemstart oder Aktualisierung des Systems.
Interaktionsfolge:
1. Die *Environment-Komponente* fordert die Positionierung des Fahrzeugs bei der aktuellen Position an, indem eine interne Methode aufgerufen wird.
2. Die *Environment-Komponente* konvertiert mit Hilfe der *GeoReference-Komponente* die Geokoordinate in eine kartesische Koordinate.
3. Das Fahrzeugmodell wird in der Szene, bei der berechneten kartesische Koordinate, platziert.

Erweiterungen:
-

3.1.3 Eigene Orientierung in der Umgebung bestimmen

Im Folgenden werden die in Abbildung 3.3 dargestellten UseCases beschrieben.

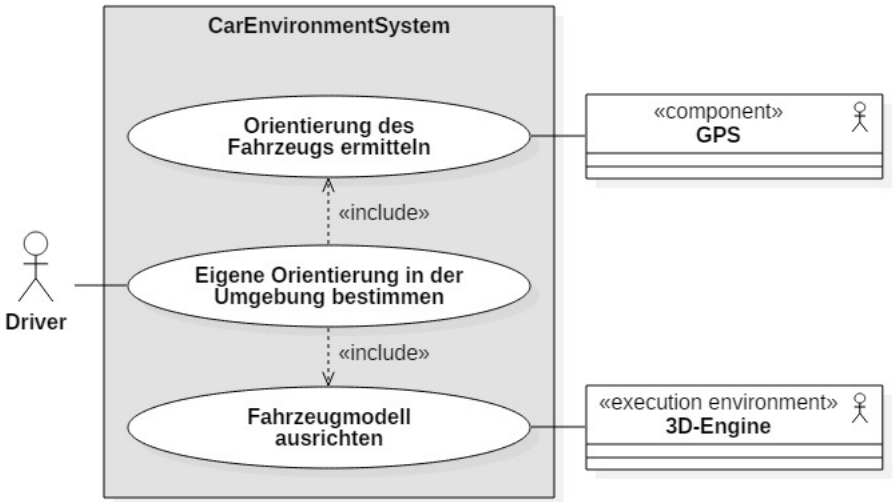

Abbildung 3.3: UseCase: Eigene Orientierung in der Umgebung bestimmen

Funktionale Anforderung: F3.1
Das Fahrzeugmodell wird anhand der Orientierung in der realen Umgebung, in der visualisierten Umgebung ausgerichtet.

UseCase 3.7: Eigene Orientierung in der Umgebung bestimmen

Aktoren: Driver, GPS, 3D-Engine
Ziel: Dem Fahrer soll die Orientierung des Fahrzeugs in der visualisierten Umgebung ersichtlich werden.
Vorbedingungen:
- Es ist ein 3D-Modell des Fahrzeugs verfügbar.
- Zugriff auf GPS ist möglich.

Nachbedingungen:
- Das 3D-Modell des Fahrzeugs ist entsprechend der Blickrichtung, in der visualisierten Umgebung, ausgerichtet.

Auslösendes Ereignis: Systemstart oder Aktualisierung des Systems.
Interaktionsfolge:
1. Durch das *Environment-Script* in der 3D-Engine wird die Ausrichtung des Fahrzeugs bei der *Environment-Komponente* angestoßen.
2. Die *Environment-Komponente* ruft die aktuelle Orientierung des Fahrzeugs von der *GPS-Komponente* ab. Details finden sich in UseCase **3.8 Orientierung des Fahrzeugs ermitteln**.
3. Die *Environment-Komponente* richtet das Fahrzeugmodell anhand der zuvor ermittelten Werte in der Umgebung aus. Die Details sind in UseCase **3.9 Fahrzeugmodell ausrichten** beschrieben.

Erweiterungen:
2.a Fehler beim Abruf der aktuellen Ausrichtung.
2.a1 Das Problem wird in der Visualisierung kenntlich gemacht, z.B. indem Icon über dem Fahrzeug platziert wird.
2.a2 Die Ausführung des UseCase **3.9 Fahrzeugmodell ausrichten** entfällt, da die Ausrichtung nicht bekannt ist.

Funktionale Anforderung: F3.2
Die Orientierung des Fahrzeugs in der realen Umgebung, wird von einem GPS-System abgerufen.

UseCase 3.8: Orientierung des Fahrzeugs ermitteln

Aktoren: Driver, GPS
Ziel: Ermittlung der aktuellen Orientierung des Fahrzeugs.
Vorbedingungen:
- Zugriff auf das GPS ist möglich.

Nachbedingungen:
- Die aktuelle Orientierung des Fahrzeugs ist bekannt.

Auslösendes Ereignis: Systemstart oder Aktualisierung des Systems.
Interaktionsfolge:
1. Die *Environment-Komponente* fragt die aktuelle Ausrichtung bei der *GPS-*

Komponente an.
2. Die *GPS-Komponente* ruft die aktuelle Orientierung von der GPS-Hardware ab.
3. Die *GPS-Komponente* übergibt die aktuelle Orientierung an die *Environment-Komponente*.

Erweiterungen:

2.a Zugriff auf das GPS geht verloren.

2.a1 Im Hintergrund wird versucht den Zugriff zurückzuerlangen.

2.a2 Der *Environment-Komponente* wird das Problem signalisiert.

Funktionale Anforderung: F3.3
Das Fahrzeugmodell wird anhand der in F3.2 ermittelten Orientierung, in der visualisierten Umgebung ausgerichtet.

UseCase 3.9: Fahrzeugmodell ausrichten

Aktoren: Driver, 3D-Engine
Ziel: Ausrichtung des Fahrzeugmodells in der Szene.
Vorbedingungen:
- Es ist ein 3D-Modell des Fahrzeugs verfügbar.
- Die aktuelle Ausrichtung des Fahrzeugs ist bekannt.

Nachbedingungen:
- Die Ausrichtung des 3D-Modells in der visualisierten Umgebung, stimmt mit der vom GPS abgerufenen Orientierung überein.

Auslösendes Ereignis: Systemstart oder Aktualisierung des Systems.
Interaktionsfolge:
1. Die *Environment-Komponente* richtet das Fahrzeugmodell entsprechend der abgerufenen Parameter aus.

Erweiterungen:
-

3.2 Nichtfunktionale Anforderungen

Die Nicht-Funktionalen Anforderungen werden in zwei Klassen unterschiedlicher Priorität unterteilt. Die primären Anforderungen gilt es dabei in jedem Fall zu erfüllen, während von den sekundären Anforderungen, falls notwendig, auch abgewichen werden darf.

3.2.1 Primäre Nicht-Funktionale Anforderungen

Der Prototyp soll durch andere Entwickler weiterentwickelt werden, somit stehen die Anforderungen nach Wartbarkeit und Flexibilität/Erweiterbarkeit an erster Stelle und sind für das System zwingend erforderlich.

Wartbarkeit

Der Prototyp muss ein gute Wartbarkeit aufweisen, damit andere Entwicklern den Code leicht verstehen und gegebenenfalls korrigieren, verbessern und erweitern können.
Um eine gute Wartbarkeit zu erreichen, muss der Code daher nachvollziehbar und vollständig kommentiert werden. Dies bedeutet, dass ...

- **jede Klasse** mit einer Beschreibung des übergeordneten Zwecks beginnt.
- **jeder globalen Variable** eine Beschreibung zu dessen Zweck vorangestellt wird.
- **jeder Methode** eine Beschreibung der Funktion vorangestellt wird. Zusätzlich sind weitere Kommentare innerhalb der Methode zu setzen, falls dadurch die Funktionalität besser nachvollziehbar wird (z.B. bei Berechnungen, Zwischenergebnissen und Teilabschnitten).

Neben der ausführlichen Kommentierung ist darauf zu achten, dass während der Entwicklung nicht mehr benötigter Code umgehend entfernt wird.
Damit Fehler im Code leichter auffindbar werden, wird das zu entwickelnde System in mehrere Komponenten aufgebrochen. Die komponentenbasierte Struktur ermöglicht es, einen Fehler leicht auf eine einzelne Komponente zu beschränken und reduziert somit den zu betreibenden Aufwand bei der Suche.

Nichtfunktionale Anforderung: NF1.1
Der Code wird entsprechend den oben angegebenen Regeln kommentiert.

Nichtfunktionale Anforderung: NF1.2
Toter Code wird umgehend entfernt.

Nichtfunktionale Anforderung: NF1.3
Das System wird komponentenbasiert entwickelt.

Flexibilität/Erweiterbarkeit

Die Entwicklung des Prototyps ist nur der erste Schritt in einer Reihe von Entwicklungsstufen. Daher ist das System so zu gestalten, dass weitere Funktionen auf einfache Weise ergänzt werden können. Aus diesem Grund wird die Hauptkomponente als Framework konzipiert, welche das Zusammenwirken der einzelnen Komponenten orchestriert. Damit die Komponenten leicht ersetzt werden können, wird für jede Komponente eine Schnittstellenspezifikation definiert und über die Hauptkomponente bereitgestellt. Dies ermöglicht es auch, für jede Schnittstellenspezifikation mehrere Komponenten zu schreiben und diese während der Entwicklung gegeneinander zu testen. Da die Hauptkomponente alle Einzelteile zusammenfügt und den Ablauf der Funktionsaufrufe regelt, lassen sich an dieser Stelle mit geringem Aufwand weitere Komponenten (z.B. für die Behandlung der dynamischen Elemente) in das System integrieren.

Nichtfunktionale Anforderung: NF2.1
Die Hauptkomponente wird als Framework konzipiert.

Nichtfunktionale Anforderung: NF2.2
Die Hauptkomponente kennt nur die Schnittstellenspezifikationen der Komponenten RoadsMapProvider, BuildingsMapProvider, TerrainMapProvider sowie GPS und nicht deren konkreten Implementie-

rungen.

Nichtfunktionale Anforderung: NF2.3
Die Hauptkomponente orchestriert den gesamten Ablaufprozess, so dass die Komponenten RoadsMapProvider, BuildingsMapProvider, TerrainMapProvider und GPS voneinander unabhängig sind.

Performance

Um ein ruckelfreies Wiedergabebild zu erzeugen, soll ein Aktualisierungsvorgang nur wenige Millisekunden (< 40ms = 25 Bilder/Sekunde) betragen.

Nichtfunktionale Anforderung: NF3
Ein Aktualisierungsvorgang benötigt nicht mehr als 40ms.

3.2.2 Sekundäre Nicht-Funktionale Anforderungen

Bei den sekundären Nicht-Funktionalen Anforderungen geht es vor allem darum Standards einzuhalten. Unter anderem sind Werkzeuge und Verfahren einzusetzen, welche im Unternehmen bzw. der Abteilung für diese Art von Aufgabe bereits häufiger eingesetzt wurden. Von den Standards kann abgewichen werden, wenn sich herausstellt, dass die Aufgabe mit den Standardmitteln nicht erfüllbar ist.

3D-Engine

Als 3D-Engine (Physics-Engine) wird Unity3D mit der Skriptsprache C# eingesetzt.
Version: 5.6.0f3.

Nichtfunktionale Anforderung: NF4
Als 3D-Engine (Physics-Engine) wird Unity3D in der Version 5.6.0.3f eingesetzt, mit der Skriptsprache C#.

Entwicklungsumgebung

Als Entwicklungsumgebung wird Microsoft Visual Studio verwendet.
Version: 2017 Community

Nichtfunktionale Anforderung: NF5
Als Entwicklungsumgebung wird Microsoft Visual Studio in der Version 2017 Community verwendet.

Programmiersprache

Als Programmiersprache wird C# eingesetzt.

Nichtfunktionale Anforderung: NF6
Als Programmiersprache wird C# eingesetzt.

Kartenanbieter

Als Kartenanbieter für die Fahrbahndaten wird Here HD Live Maps genutzt. Für den Zugriff auf das Kartenmaterial ist bereits eine unternehmensweite Entwickler-Lizenz verfügbar.

Nichtfunktionale Anforderung: NF7
Als Kartenanbieter für die Fahrbahndaten wird Here HD Live Maps genutzt.

4 Entwurf I: Architektur

Aufbauend auf den im vorherigen Kapitel aufgestellten Anforderungen wird als nächstes die Architektur des Prototyps entwickelt.

4.1 Komponenten des Prototyps

Die Abbildung 4.1 zeigt ein Komponentendiagramm, welches die zuvor aufgestellten Anforderungen widerspiegelt. Dabei wird in dem Komponentendiagramm für die Aktoren RoadsMapProvider, BuildingsMapProvider, TerrainMapProvider und GPS jeweils eine eigene gleichnamige Komponente definiert. Diese Komponenten sind das Bindeglied zwischen der jeweiligen externen Ressource und der Hauptkomponente **Environment**. Aufgabe der Komponenten ist der Abruf der benötigten Daten von der externen Ressource, sowie gegebenenfalls der Ausführung notwendiger Vorverarbeitungsschritte.

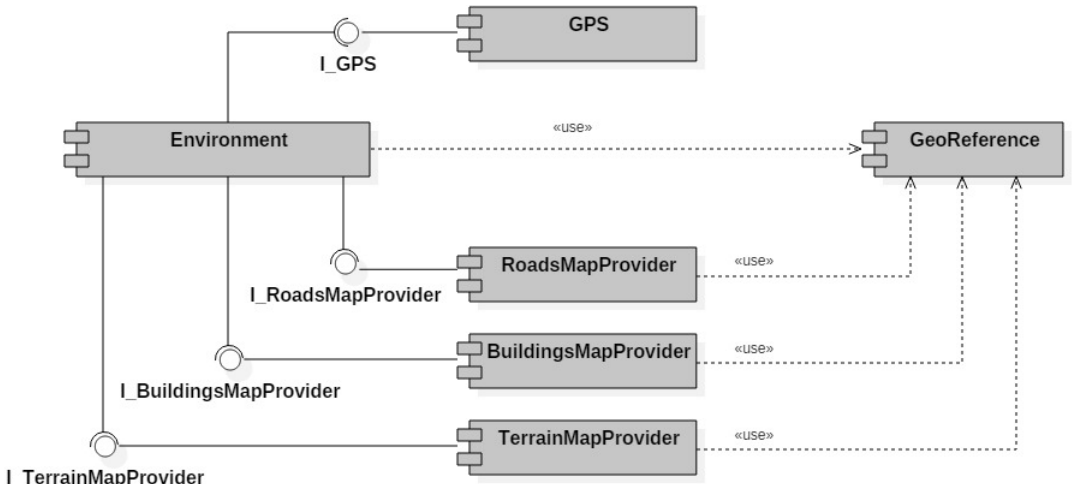

Abbildung 4.1: Komponentendiagramm des Prototyps

Für den Aktor Driver wird keine Komponente definiert, da der Fahrer das System lediglich startet und beendet. In direkte Berührung mit der Software kommt der Fahrer nur durch das Betrachten der Visualisierung und somit ist eine Komponente für Benutzerinteraktionen nicht erforderlich. Der Aktor 3D-Engine erhält ebenfalls keine eigene Komponente, zumindest nicht direkt. Im Kontext der 3D-Engine wird eine Instanz der Hauptkomponente **Environment** erzeugt und konfiguriert, sowie dessen Methoden aufgerufen. Die **Environment-Komponente** führt dann alle weiteren Aufgaben durch, wie die Orchestrierung der anderen Komponenten und das erzeugen und platzieren der 3D-Modelle. Somit kann die Komponente **Environment** als Komponente für die 3D-Engine angesehen werden, wobei diese auch weitere Aufgaben übernimmt.

Hinzu kommt eine neue Komponente, welche nicht in den UseCase-Diagrammen dargestellt wurde. Die neue Komponente trägt die Bezeichnung **GeoReference**. Die **GeoReference-Komponente** wird von mehreren Komponenten für die Umrechnung von Geokoordinaten in kartesische Koordinaten und der Distanzberechnung auf einem Referenzellipsoiden benötigt.

Es möglich das ein Kartenanbieter alle Daten für Straßen, Gebäude und das Terrain zur Verfügung stellt. In diesem Fall kann eine Komponente entwickelt werden, welche die drei Schnittstellen I_RoadsMapProvider, I_BuildingsMapProvider und I_TerrainMapProvider implementiert. Eine Instanz dieser Komponente kann dann gleichzeitig mit den drei zugehörigen Schnittstellen der **Environment-Komponente** verbunden werden, um alle Daten aus einer Quelle zu beziehen.

4.2 Verteilung der Komponenten und externe Beziehungen

Die Komponentenverteilung gestaltet sich unkompliziert. Das komplette zu entwickelnde System läuft auf einer Maschine im Kontext der 3D-Engine. Um die benötigten Daten zu erhalten, werden externe Ressourcen abgefragt. Das Verteilungsdiagramm wird in Abbildung 4.2 dargestellt.

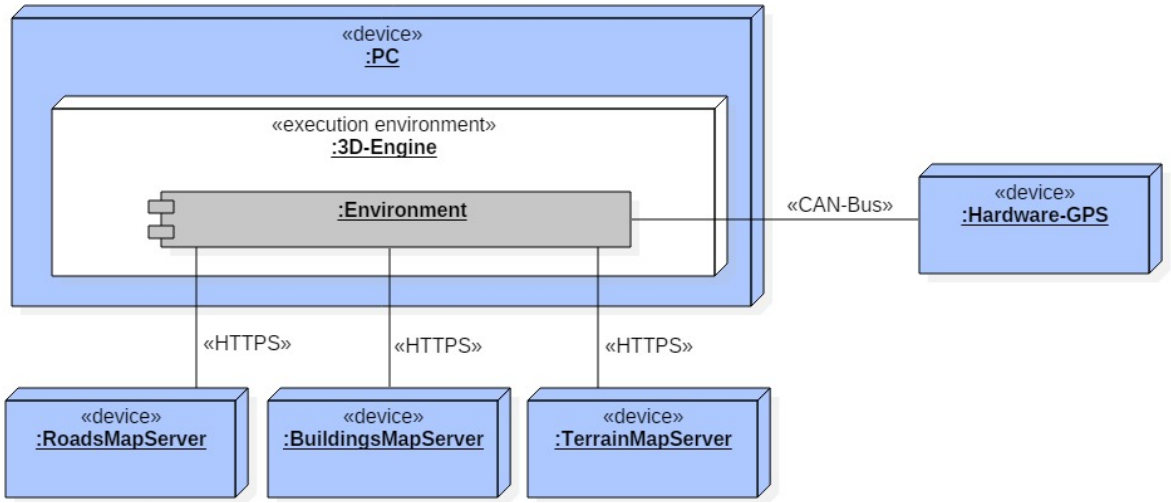

Abbildung 4.2: Verteilungsdiagramm

Die Komponente **Environment** verkörpert an dieser Stelle alle Komponenten des Prototyp, wie er in 4.1 abgebildet ist. Die Verbindungstypen zu den externen Ressourcen können in einer konkreten Implementierung auch abweichen. HTTPS als Verbindungstyp zu den Kartenanbietern wurde hier gewählt, da alle in der Analyse betrachteten Anbieter diesen Verbindungstyp unterstützen und dieser hier später eingesetzt wird. Bezüglich der möglichen Verbindungstypen zum Hardware-GPS wurden keine weiteren Untersuchungen angestellt und daher wurde der in dieser Arbeit verwendete Verbindungstyp CAN-Bus ausgewählt.

5 Entwurf II: Die Komponenten im Detail

In der Abbildung 4.2 wird deutlich gezeigt, dass die komplette Anwendung im Kontext der 3D-Engine ausgeführt wird. Das bedeutet, dass in der Update-Routine der 3D-Engine, die Methoden der **Environment-Komponente** aufgerufen werden. Die Update-Routine der 3D-Engine wird dabei vor jedem Renderingvorgang ausgeführt. Die Konsequenz die sich daraus ergibt ist, dass ein langwieriger Aktualisierungsvorgang der Anwendung den Renderingvorgang der 3D-Engine verzögert und dies eine stockende Wiedergabe oder auch längere Standbilder zur Folge haben kann. Warnungen oder Hinweise können in dieser Zeit ebenfalls nicht angezeigt werden, da der Renderingvorgang erst nach dem Update gestartet wird. Um das Rendering nicht zu verzögern, wird daher an vielen Stellen auf Parallelität gesetzt. Das Erzeugen und Positionieren der 3D-Modelle muss allerdings im Hauptthread ausgeführt werden und kann nicht in parallele Stränge ausgelagert werden.

5.1 Environment

Die Abbildung 5.1 zeigt die Klassen **Environment** und **CacheItem**. Die beiden Klassen stellen die gesamte Funktionalität der Komponente bereit.

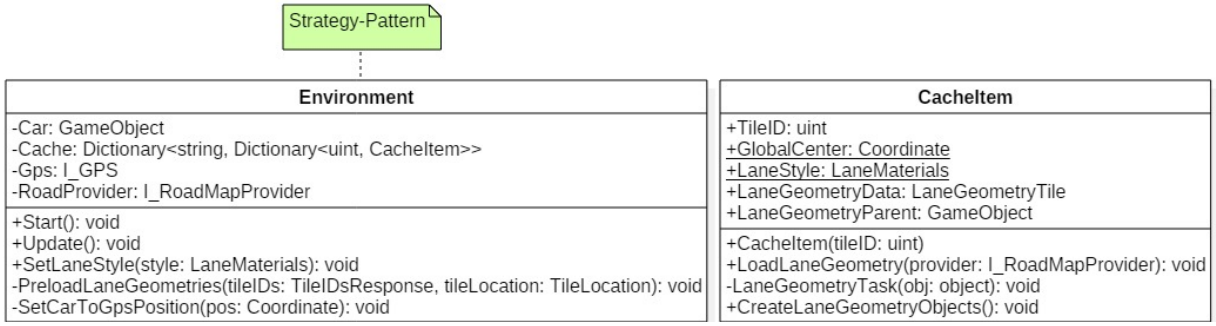

Abbildung 5.1: Klassendiagramm für die Klassen Environment und CacheItem

Die Komponente **Environment** regelt die primären Abläufe des Systems und koordiniert das Zusammenspiel der verschiedenen Komponenten. Diese Aufgabe wird dabei vornehmlich von der gleichnamigen Klasse **Environment** übernommen. Die Tabelle 5.1 zeigt eine Übersicht der Attribute und deren Zweck. Zur Erläuterung der Methoden wird auf den Abschnitt 5.1.3 verwiesen.

Klasse: Environment	
Attribut	Zweck
Car	Fahrzeugmodell

Cache	Cache für 3D-Modelle und die zugehörigen Daten. Die geschachtelte Dictionary-Struktur für das **Cache-Attribut** ist dabei folgendermaßen zu interpretieren. Für jeden Provider existiert ein eigener Cache, welcher über den Providernamen abrufbar ist. In den Cache des Providers ist dann eine **CacheItem-Instanz** über die ID des Kartenabschnitts (TileID) zugreifbar.
Gps	GPS-Komponente welche den Zugriff auf die aktuelle Position ermöglicht.
RoadProvider	Komponente zum Abruf der Fahrbahngeometrien.

Tabelle 5.1: Attribute der Klasse Environment

Die Instanzen der Klasse **CacheItem** stehen stellvertretend für jeweils einen Kartenausschnitt. Jedes **CacheItem** hat neben seiner Funktion als Datenspeicher die Aufgabe, die Geometriedaten über die MapProvider abzurufen und daraus 3D-Modelle zu erzeugen. Auch hier zeigt die nachfolgende Tabelle (Tabelle 5.2) eine Übersicht der Attribute der Klasse **CacheItem** und deren Zweck. Zur Erläuterung der Methoden wird auf den Abschnitt 5.1.3 verwiesen.

Klasse: CacheItem	
Attribut	Zweck
TileID	ID für die Identifikation eines Kartenabschnitts
GlobalCenter	Statische Variable. Gibt an, welche geodätische Koordinate auf den Ursprung des kartesischen Systems abgebildet wird.
LaneGeometryData	Vom Provider gelieferte Daten für den Kartenabschnitt.
LaneGeometryParent	Eltern-Element der 3D-Modelle für den Kartenabschnitt. Alle 3D-Modelle werden mit einem Eltern-Objekt verknüpft, so dass der gesamte Kartenabschnitt leicht verschoben werden kann.
LaneStyle	Statische Variable. Materialien für die Fahrbahngeometrien.

Tabelle 5.2: Attribute der Klasse CacheItem

An dieser Stelle war von Providern in der Mehrzahl die Rede, obwohl nur eine konkrete Implementierung für die Schnittstelle **I_RoadsMapProvider** vorliegt. Der Hintergrund ist, dass die **Environment-Komponente** bereits die Schnittstellenspezifikationen **I_BuildingsMapProvider** und **I_TerrainMapProvider** vorsieht und nutzt. Allerdings existieren noch keine konkreten Implementierungen der Schnittstellen und nur leere Datenklassen für die Kommunikation, weshalb diese Komponenten in der weiteren Betrachtung außer Acht gelassen werden. Die Handhabung der Komponenten in der **Environment-Komponente** erfolgt analog der Nutzung der **RoadsMapProvider-Komoponente**. Der wesentliche Unterschied in der Nutzung der **RoadsMapProvider-Komponente** und den Komponenten **BuildingsMapProvider** und **TerrainMapProvider** ist der zurückgelieferte Datentyp für die Geometrien und daraus resultierend eine abweichende Vorgehensweise bei der Erzeugung der 3D-Modelle.

5.1.1 Schnittstellenspezifikationen

Die **Environment-Komponente** stellt vier Schnittstellenspezifikationen bereit (siehe Abbildung 5.2). Die Schnittstellenspezifikationen müssen von anderen Komponenten implementiert werden, um in die **Environment-Komponente** eingehakt werden zu können.

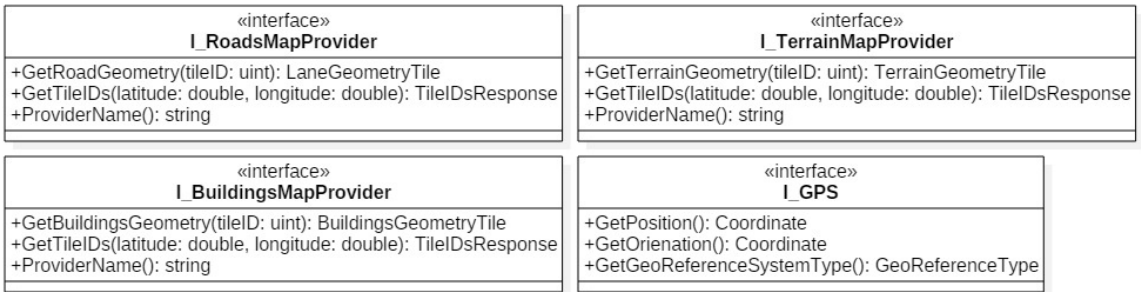

Abbildung 5.2: Klassendiagramm für die Schnittstellenspezifikationen

I_RoadsMapProvider, I_BuildingsMapProvider und I_TerrainMapProvider

Eine Komponente, welche Daten für Fahrbahngeometrien bereitstellt, muss die Schnittstelle **I_RoadsMapProvider** implementieren. Die durch die Schnittstelle vorgegebenen Methoden müssen dazu die in Tabelle 5.3 beschriebenen Aufgaben erfüllen.

Interface: I_RoadsMapProvider	
Methode	Aufgabe
GetTileIDs	Die Methode liefert neun IDs. Jede ID ist für die Identifikation eines Kartenabschnitts bestimmt. Wobei die Kartenabschnitte zu den IDs eine 3x3-Matrix bilden, mit dem Kartenabschnitt der übergebenen Position im Zentrum der Matrix (siehe 5.4).
GetRoadsGeometry	Die Methode ruft den Kartenabschnitt zur übergebenen ID vom Kartenserver ab oder lädt diesen aus einem anderen Speicher. Die abgerufenen Daten müssen dann in Mesh-Daten (Vertices, UVs, Triangles und Material-Typ) konvertiert und in das LaneGeometryTile (siehe Abschnitt 5.1.2) eingepasst werden.
ProviderName	Stellt den Providernamen zur Verfügung. Der Name kann beliebig gewählt werden, ist aber mit einer ID gleichzusetzen.

Tabelle 5.3: Aufgaben der Methoden des Interface I_RoadsMapProvider

Gleiches gilt für die Schnittstellen **I_BuildingsMapProvider** und **TerrainMapProvider**, bei denen jedoch durch die Methoden **GetBuildingsGeometry** bzw. **GetTerrainGeometry** andere Datentypen zurückgegeben werden müssen.

I_GPS

Bei der Implementierung einer konkreten Komponente für die Schnittstelle **I_GPS**, müssen die Methoden, die in Tabelle 5.4 beschriebenen Verhalten implementieren.

Interface: I_GPS	
Methode	Aufgabe
GetPosition	Liefert die aktuelle, durch das GPS ermittelte Koordinate.
GetOrientation	Liefert die aktuelle, durch das GPS ermittelte Orientierung. Wobei 0-Grad nach Norden weist, 90-Grad nach Osten, 180-Grad nach Süden und 270-Grad nach Westen.
GetGeoReferenceSystemType	Liefert das der Koordinate zugrunde liegende geodätische Referenzsystem.

Tabelle 5.4: Aufgaben der Methoden des Interface I_GPS

5.1.2 Daten-Klassen

Die **Environment-Komponente** stellt die in Abbildung 5.3 dargestellten Daten-Klassen, für den Datenaustausch mit anderen Komponenten, zur Verfügung.

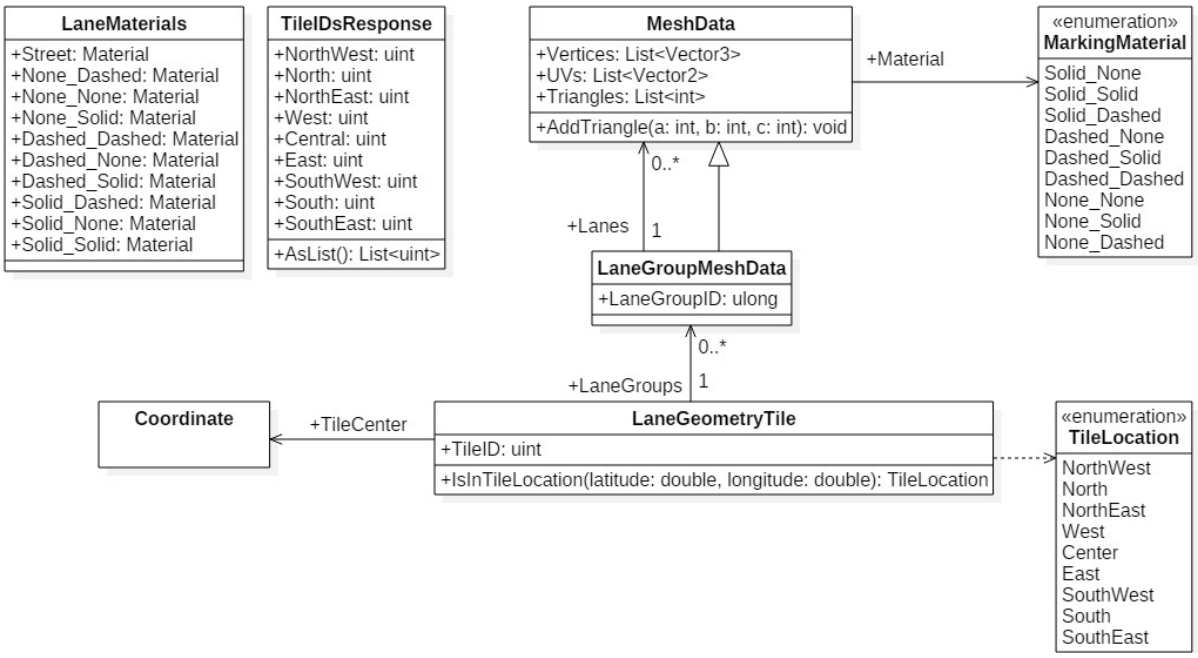

Abbildung 5.3: Klassendiagram der Daten-Klassen der Komponente Environment

Klasse: LaneMaterials

Die Klasse **LaneMaterials** wird von der 3D-Engine genutzt, um der **Environment-Komponente** mitzuteilen, welche Materialien bei der Erzeugung der Fahrbahnmodelle verwendet werden sollen. Die Attributbeschreibungen sind in Tabelle 5.5 zu sehen.

Klasse: LaneMaterials	
Attribut	Zweck
Street	Grundmaterial für die Fahrbahnen.
None_Dashed	Textur für Fahrbahnmarkierungen: Links keine, rechts gestrichelt.
None_None	Textur für Fahrbahnmarkierungen: Links keine, rechts keine.
None_Solid	Textur für Fahrbahnmarkierungen: Links keine, rechts durchgezogen.
Dashed_Dashed	Textur für Fahrbahnmarkierungen: Links gestrichelt, rechts gestrichelt.
Dashed_None	Textur für Fahrbahnmarkierungen: Links gestrichelt, rechts keine.
Dashed_Solid	Textur für Fahrbahnmarkierungen: Links gestrichelt, rechts durchgezogen.
Solid_Dashed	Textur für Fahrbahnmarkierungen: Links durchgezogen, rechts gestrichelt.
Solid_None	Textur für Fahrbahnmarkierungen: Links durchgezogen, rechts keine.
Solid_Solid	Textur für Fahrbahnmarkierungen: Links durchgezogen, rechts durchgezogen.

Tabelle 5.5: Attribute der Klasse LaneMaterials

Enumeration: MarkingMaterial

Die Enumeration **MarkingMaterial** wird benötigt, damit die Komponente **RoadMapProvider** den Fahrbahnen ein Material zuordnen kann, ohne die tatsächlichen Material-Objekte zu kennen. Die Namen der Elemente der Enumeration sind identisch mit den Attributnamen der **LaneMaterials-Klasse**, wodurch eine spätere Zuordnung erleichtert wird.

Klasse: TileIDsResponse

Die Klasse **TileIDsResponse** ist das Rückgabe-Objekt bei der Anfrage nach den IDs für die Kartenabschnitte zu einer Koordinate. Die IDs beschreiben dabei eine 3x3-Matrix von Kartenabschnitten, wie in Abbildung 5.4 zu sehen ist. Die Attribute der Klasse werden in Tabelle 5.6 beschrieben.

Klasse: TileIDsResponse	
Attribut	Zweck
NorthWest	ID des Kartenabschnitts nordwestlich der angefragten Koordinate.
North	ID des Kartenabschnitts nördlich der angefragten Koordinate.
NorthEast	ID des Kartenabschnitts nordöstlich der angefragten Koordinate.
West	ID des Kartenabschnitts westlich der angefragten Koordinate.
Center	ID des Kartenabschnitts in dem sich die angefragte Koordinate befindet.
East	ID des Kartenabschnitts östlich der angefragten Koordinate.
SouthWest	ID des Kartenabschnitts südwestlich der angefragten Koordinate.
South	ID des Kartenabschnitts südlich der angefragten Koordinate.
SouthEast	ID des Kartenabschnitts südöstlich der angefragten Koordinate.

Tabelle 5.6: Attribute der Klasse TileIDsResponse

Enumeration: TileLocation

Mit der Enumeration **TileLocation** wird eine 3x3-Matrix beschrieben, die Elementnamen entsprechen dabei den Attributnamen der Klasse **TileIDsResponse**. Eine Veranschaulichung zeigt Abbildung 5.4.

NW	N	NE
W	C	E
SW	S	SE

Abbildung 5.4: 3x3-Matrix von Kartenabschnitten

Die Enumeration wird hauptsächlich für das vorrausschauende laden neuer Kartenabschnitte verwendet und dort für mehrere Zwecke eingesetzt.

MeshData

Bei der Klasse **MeshData** handelt es sich um die allgemeine Form der Mesh-Daten, für ein zu modellierendes 3D-Objekt. Die Details zu den Attributen werden in Tabelle 5.7 dargestellt.

Klasse: MeshData	
Attribut	Zweck
Vertices	Eine Punktliste zur Beschreibung der äußeren Form des Modells.
UVs	Eine Liste von Texturkoordinaten.
Triangles	Eine Liste von Indizes auf die Liste Vertices. Je drei aufeinanderfolgende Einträge beschreiben ein Dreieck.
Material	Gibt den Typ des Materials für das Fahrbahnmodell an.

Tabelle 5.7: Attribute der Klasse MeshData

Die Methode **AddTriangle** erwartet drei Indizes als Parameter. Jeder der Indizes muss dabei auf einen Vektor in der Vertices-Liste verweisen. Aus den Vektoren, welche durch die Indizes bestimmt wurden, wird dann ein Dreieck konstruiert, dessen Normalenvektor in Richtung der positiven Y-Achse zeigt. Das Dreieck muss mit dem Normalenvektor in Richtung der positiven Y-Achse ausgerichtet werden, da Flächen von der 3D-Engine nur gerendert werden, wenn die Normale in Richtung der Kamera zeigt und diese befindet sich immer über der Fahrbahn (Blickrichtung entlang der negativen Y-Achse). Die Indizes werden dann in der korrekten Reihenfolge in die Vertices-Liste aufgenommen.

LaneGroupMeshData

Die Klasse **LaneGroupMeshData** repräsentiert einen Fahrbahnabschnitt und enthält sowohl die Mesh-Daten für die einzelnen Spuren der Fahrbahnen, als auch für den Fahrbahnabschnitt als Ganzes (LaneGroup). Die Abbildung 5.5 zeigt einen Fahrbahnabschnitt, wie er in der *Here HD Live Map* definiert ist.

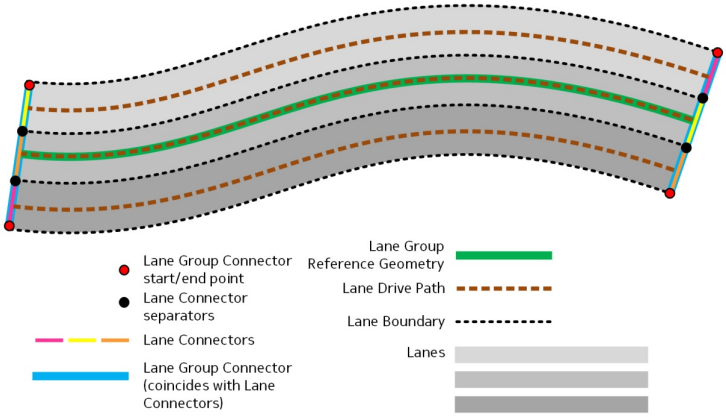

Abbildung 5.5: Repräsentation eines Fahrbahnabschnitts im HereHD-Format [27]

Die Attribute der Klasse **LaneGroupMeshData**werden in der folgenden Tabelle 5.8 näher erläutert.

Klasse: LaneGroupMeshData	
Attribut	Zweck
LaneGroupID	ID des Fahrbahnabschnitts.
Lanes	Eine Liste von Fahrbahnspuren, beschrieben durch MeshData-Instanzen.
Vertices	(Geerbt von MeshData) Punktliste der äußeren Kontur der Fahrbahn.
UVs	(Geerbt von MeshData).
Triangles	(Geerbt von MeshData).
Material	(Geerbt von MeshData).

Tabelle 5.8: Attribute der Klasse LaneGroupMeshData

LaneGeometryTile

Mit der Klasse **LaneGeometryTile** wird ein kompletter Kartenabschnitt beschrieben. D.h. alle Fahrbahnen eines Kartenabschnitts finden sich hier wieder. Die Tabelle 5.9 enthält eine Aufstellung der Attribute und ihrer Bedeutung.

Klasse: LaneGeometryTile	
Attribut	Zweck
TileID	ID des Kartenabschnitts.
LaneGroups	Eine Liste von Fahrbahnen inklusive der einzelnen Spuren, beschrieben durch LaneGroupMeshData-Instanzen.
TileCenter	Koordinate des Zentrums des Kartenabschnitts.

Tabelle 5.9: Attribute der Klasse LaneGeometryTile

Mit der Methode **IsInTileLocation** kann zu einer Koordinate ermittelt werden, in welchem Teilstück des Kartenabschnitts sich die Koordinate befindet. Dazu wird der Kartenabschnitt in eine 3x3-Matrix mit Feldern gleicher Größe unterteilt. Die Rückgabe der Methode wird durch die Enumeration **TileLocation** beschrieben.

5.1.3 Interaktionen

In diesem Abschnitt werden die Funktionsabläufe der Methoden der Klassen **Environment** und **CacheItem** genauer betrachtet und anhand von Sequenzdiagrammen veranschaulicht. Zunächst wird aber gezeigt, wie der Ablauf über ein Script der 3D-Engine zu initialisieren ist.

3D-Engine Start

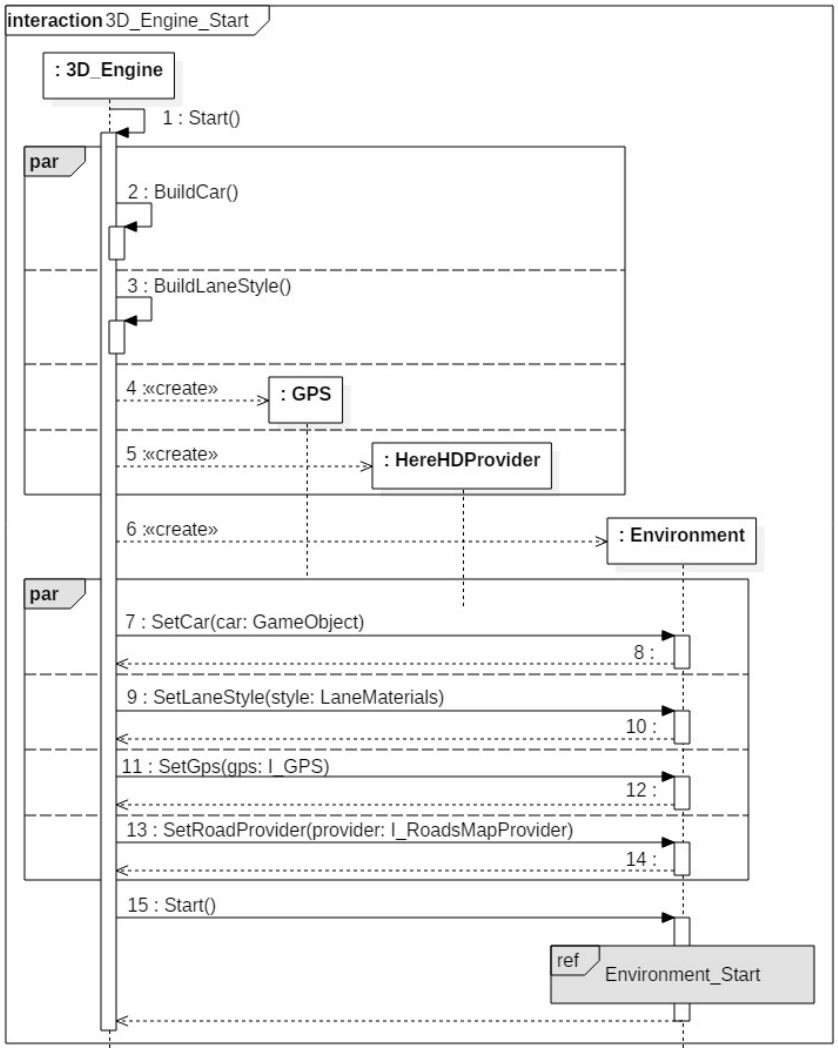

Abbildung 5.6: Sequenzdiagramm: Start-Methode im Script der 3D-Engine

Abbildung 5.6 zeigt den generellen Ablauf für die Erzeugung, Konfiguration und Start der **Environment-Komponente**. Die beiden Methoden **BuildCar** und **BuildLaneStyle** sind hier nur symbolischer Natur. Wie die Objekte letztendlich erzeugt werden, per Script oder über die GUI der 3D-Engine, bleibt dem Entwickler überlassen. Wichtig ist lediglich, dass ein Modell (**GameObject**) für das Fahrzeug und ein Objekt vom Typ **LaneMaterials** für die Texturierung der Fahrbahnen erzeugt wird. Weiterhin muss eine Instanz der **GPS-Komponente** und eine Instanz der **RoadsMapProvider-Komponente** (hier im Beispiel die konkrete Implementierung **HereHDProvider**) erzeugt werden.
Nachdem die oben genannten Objekte erzeugt sind, wird eine Instanz der Klasse **Environment** erzeugt und mit den zuvor erstellten Objekten konfiguriert. Im letzten Schritt wird die Methode **Start** der **Environment-Klasse** aufgerufen, um die Komponente zu initialisieren. Der Ablauf der **Start-Methode** wird in Abschnitt 5.1.3 "Environment Start" beschrieben.

3D-Engine Update

Die **Update-Methode**, in dem Script der 3D-Engine, ist ausschließlich dafür zuständig die **Update-Methode** der **Environment-Komponente** aufzurufen (Abbildung 5.7). Der Ablauf der **Update-Methode** der Komponente **Environment** wird in Abschnitt 5.1.3 "Environment Update" betrachtet.

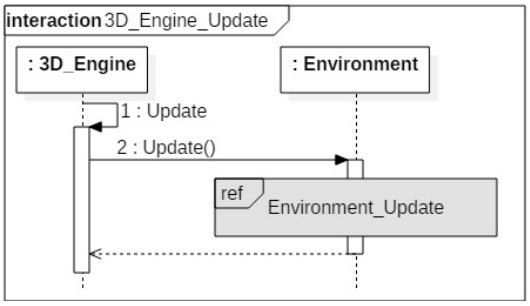

Abbildung 5.7: Sequenzdiagramm: Update-Methode im Script der 3D-Engine

Mit fortschreiten des Projekts können an dieser Stelle weitere Aufrufe, zur Laufzeit, an die **Environment-Komponente** übergeben werden. Zum Beispiel, wenn dem Benutzer ein User-Interface zur Verfügung gestellt wird, um den Stil der Visualisierung auszuwählen oder spezielle Detailansichten anzuzeigen.

Environment Start

Nachdem eine Instanz der **Environment-Klasse** erzeugt und konfiguriert ist, wird mit der Methode **Start** der Initialisierungsvorgang ausgelöst. Der Initialisierungsvorgang ist dafür zuständig, alle Daten für die Visualisierung der aktuellen Umgebung zu laden und daraus 3D-Modelle zu erzeugen. Der wesentliche Unterschied zur **Update-Methode** ist, dass hier auf das laden der Daten gewartet wird.
Wie der Abbildung 5.8 entnommen werden kann, wird zuerst die aktuelle Position bestimmt, um mit dieser anschließend die IDs der zu ladenden Kartenabschnitte zu ermitteln. Für jede der neun IDs (siehe Abschnitt 5.1.2 "Klasse TileIDsResponse") wird dann eine **CacheItem-Instanz** erzeugt und darüber der entsprechende Kartenabschnitt abgerufen. Der Ablauf des **LaneGeometryTasks** wird in Abschnitt

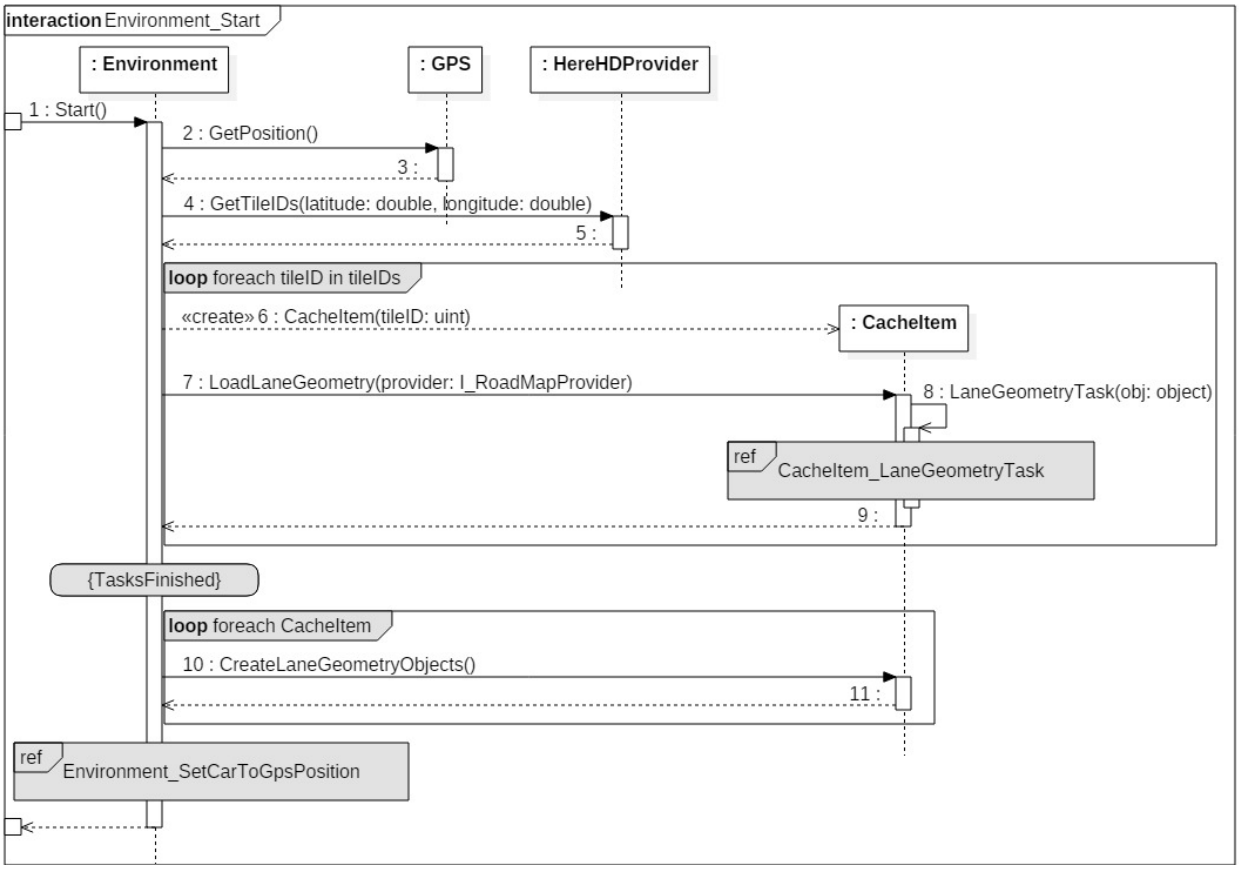

Abbildung 5.8: Sequenzdiagramm: Start-Methode der Komponente Environment

5.1.3 "CacheItem LaneGeometryTask" betrachtet.
Sobald alle **LaneGeometryTasks** vollständig abgearbeitet sind, werden anschließend für alle **CachtItem-Instanzen** die 3D-Modelle erzeugt. Den Abschluss bildet die Positionierung des Fahrzeugmodells in der Umgebung, dies wird in Abschnitt 5.1.3 "Environment SetCarToGpsPosition" beschrieben.

Environment Update

In der **Update-Methode** (Abbildung 5.9) wird geprüft, ob neue Daten geladen und/oder neue 3D-Modelle erzeugt werden müssen. Außerdem wird das Fahrzeugmodel neu positioniert.
Genau wie in der **Start-Methode**, wird in der **Update-Methode**, zunächst die aktuelle Position bestimmt und danach die IDs der Kartenabschnitte ermittelt. Der anschließende optionale Block enthält den Aufruf für die Erzeugung neuer 3D-Modelle und kommt nur zur Ausführung, wenn das Fahrzeug einen neuen Kartenabschnitt betritt und die Daten für den Abschnitt bereits geladen wurden. Bei dem Objekt **item** in der Bedingung handelt es sich um die **CacheItem-Instanz**, welche den Kartenabschnitt im Zentrum der abgerufenen 3x3-Matrix repräsentiert.
Die Methode **PreloadLaneGeometries** prüft im Anschluss, ob neue Daten geladen werden müssen und stößt gegebenenfalls den Vorgang dazu an. Eine Beschreibung der Methode findet sich in Abschnitt

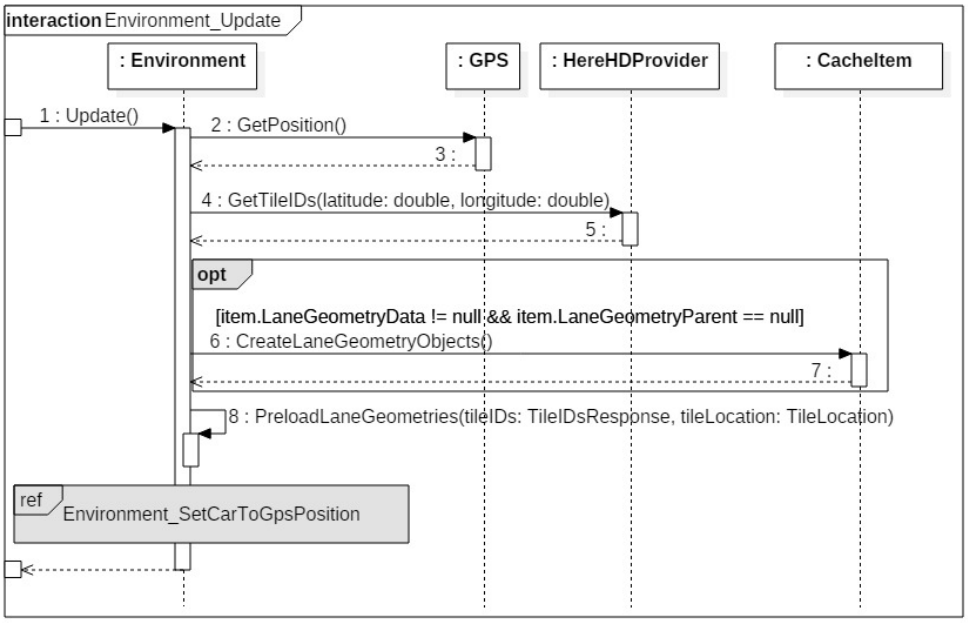

Abbildung 5.9: Sequenzdiagramm: Update-Methode der Komponente Environment

5.1.3 "Environment PreloadLaneGeometries". Die Positionierung des Fahrzeugs wird durch den Aufruf der Methode **SetCarToGpsPosition** angestoßen und in Abschnitt 5.1.3 "Environment SetCarToGps-Position" erläutert.

Hinweis 5.1: Erzeugung neuer 3D-Modelle

Die Erzeugung neuer 3D-Modelle geschieht in der aktuellen Implementierung nur, wenn das Fahrzeug einen neuen Kartenabschnitt betritt. Diese Vorgehensweise führt dazu, dass wenn das Fahrzeug auf einen neuen Kartenabschnitt zusteuert (dessen Fahrbahnen noch nicht generiert wurden), der neue Kartenabschnitt in der Visualisierung erst zu sehen ist, sobald das Fahrzeug die Abschnittsgrenze übertritt. Dieses Verhalten ist für eine Visualisierung, welche dem Nutzer ein sicheres Gefühl geben soll absolut ungeeignet und muss dringend überarbeitet werden.

Environment SetCarToGpsPosition

Die Methode **SetCarToGpsPosition** ist für die Positionierung des Fahrzeugmodells in der visualisierten Umgebung verantwortlich und erhält dazu die aktuelle Koordinate als Parameter. Den Ablauf skizziert die Abbildung 5.10.
Zuerst wird das von der **GPS-Komponente** verwendete geodätische Referenzsystem bestimmt und die aktuelle Orientierung des Fahrzeugs abgefragt. Nachdem alle für die Positionierung benötigten Parameter bekannt sind, wird die aktuelle geodätische Koordinate mit Hilfe der **GeoReference-Komponente** in eine kartesische Koordinate konvertiert. Die Positionierung selbst erfolgt dann über die Methoden

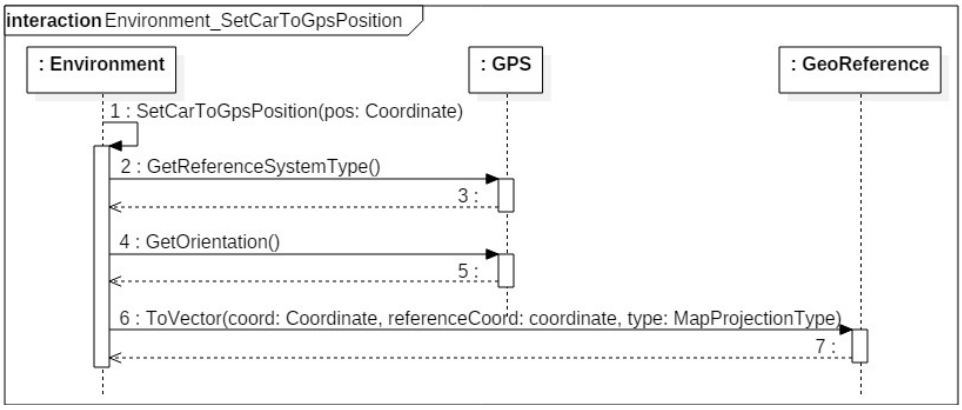

Abbildung 5.10: Sequenzdiagramm: SetCarToGpsPosition-Methode der Komponente Environment

der 3D-Engine, welche hier nicht weiter aufgeführt sind.

Environment PreloadLaneGeometries

Das vorausschauende laden neuer Kartenabschnitte wird mit der Methode **PreloadLaneGeometries** realisiert. Die Methode ist als switch-case realisiert und daher wird an dieser Stelle kein Sequenzdiagramm angeboten und der Ablauf nur textuell beschrieben.
Der Methode werden als Parameter die abgerufenen TileIDs und die Position mitgegeben, auf welchem Unterabschnitt des aktuellen Kartenabschnitts, sich das Fahrzeug derzeit befindet. Der Unterabschnitt wird dabei durch die Enumeration **TileLocation** identifiziert. D.h. die **TileLocation** bezieht sich in dieser Methode nicht auf die TileIDs, sondern auf den in eine 3x3-Matrix unterteilten aktuellen Kartenabschnitt. Anhand der aktuellen Position in dem aktuellen Kartenabschnitt, wird dann geprüft welche Kartenabschnitte zu laden sind. Die Abbildung 5.11 soll zeigen, wie die zu ladenden Kartenabschnitte ausgewählt werden. Das farblich markierte Element in der Mitte wird durch die **TileLocation** identi-

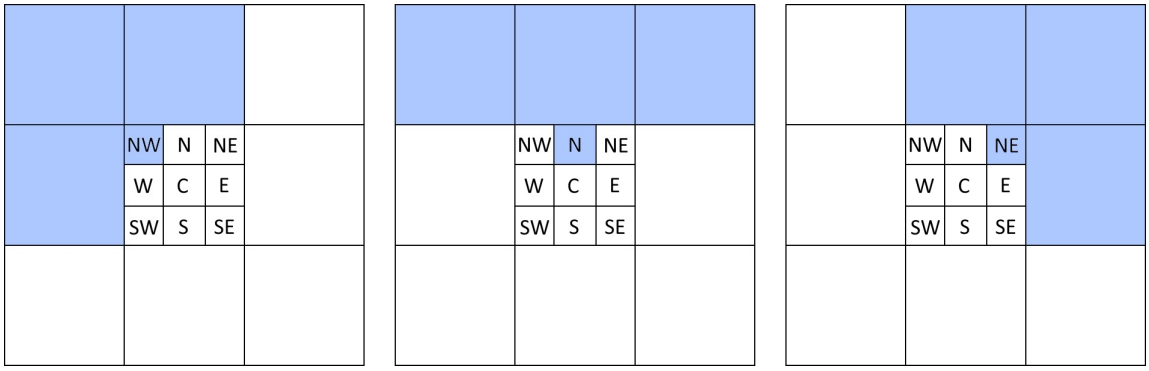

Abbildung 5.11: Vorausschauendes laden von Kartenabschnitten

fiziert und die äußeren farblich markierten Elemente zeigen, welche Kartenabschnitte zu laden sind. Befindet sich das Fahrzeug im Zentrum der aktuellen Karte, wird kein neuer Kartenabschnitt geladen. Das laden neuer Kartenabschnitte erfolgt durch das Erzeugen neuer **CachItem-Instanzen** und dem

anschließenden Aufruf der Methode **LoadLaneGeometry** auf der **CacheItem-Instanz**. Der Aufruf der Methode **LoadLaneGeometry** startet im Wesentlichen nur den asynchronen Task **LaneGeometryTask** (siehe Abschnitt 5.1.3 "CacheItem LaneGeometryTask") und kehrt dann umgehend zurück, so dass die Methode schnell abgearbeitet ist.

CacheItem LaneGeometryTask

Abbildung 5.12 stellt den Ablauf der Methode **LaneGeometryTask** dar. Aufgabe des Tasks ist es, die Fahrbahndaten über die **RoadsMapProvider-Komponente** abzurufen. In der Darstellung wird zur Veranschaulichung die konkrete Komponente **HereHDProvider** verwendet. Zusätzlich wird hier ein kurzer Einblick in den Ablauf der **HereHDProvider-Komponente** gegeben. Dies geschieht, da in den folgenden Abschnitten für die austauschbaren Komponenten auf Aktivitätsdiagramme gesetzt wird. Mit den Aktivitätsdiagrammen kann der allgemeine Ablauf der Komponenten besser erläutert werden, wohingegen die Sequenzdiagramme nur die aktuelle Implementierung wiederspiegeln.

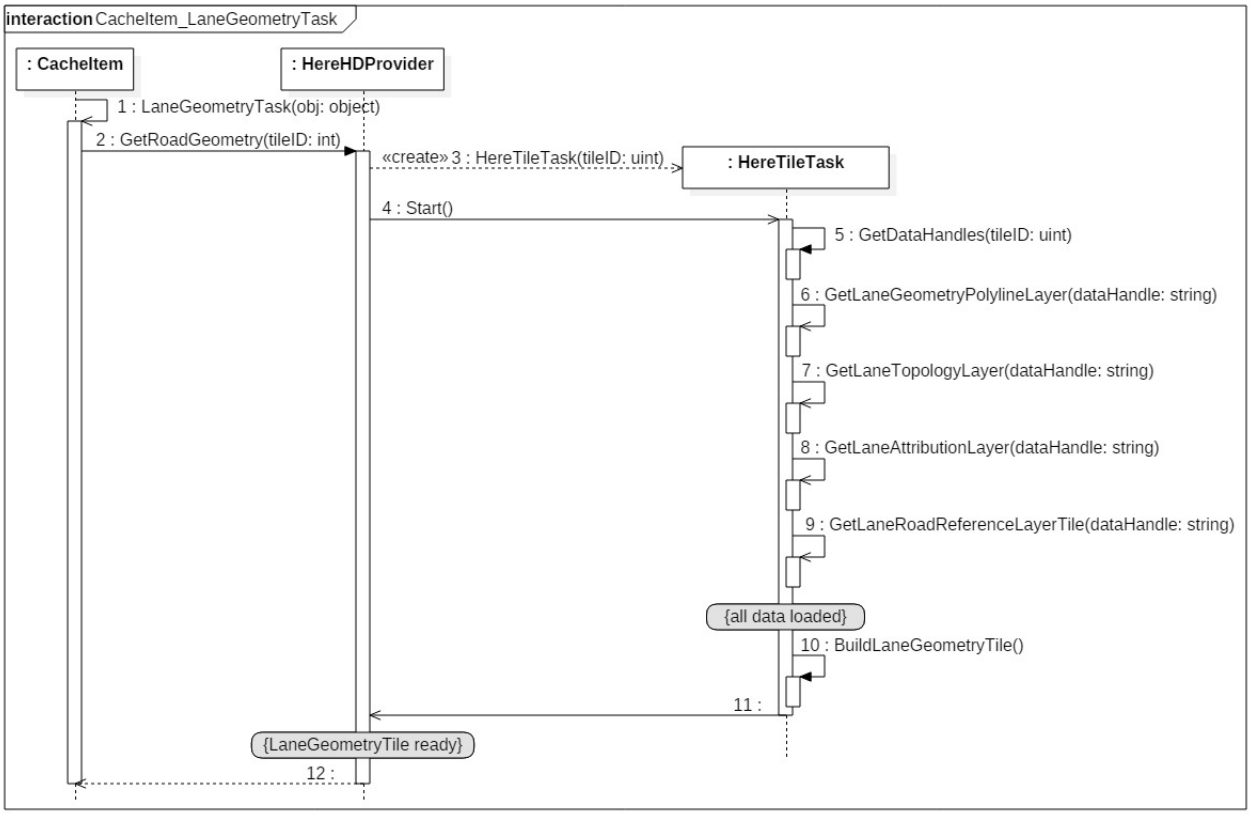

Abbildung 5.12: Sequenzdiagramm: LaneGeometryTask-Methode der Klasse CacheItem

Die asynchrone Methode **LaneGeometryTask** bekommt als Parameter den zu verwendenden **RoadsMapProvider** übergeben und führt lediglich den Aufruf der Methode **GetRoadGeometry** auf den **RoadsMapProvider** aus. Der in diesem Szenario genutzte **HereHDProvider** erzeugt daraufhin eine Instanz der Klasse **HereTileTask** und ruft die asynchrone Methode **Start** auf. In der **Start-Methode**

werden zunächst die DataHandles abgerufen. Die DataHandles sind Identifikatoren für die Layer eines Kartenabschnitts. Mit Hilfe der DataHandles können somit die einzelnen Layer eines Kartenabschnitts von dem Server des Kartenanbieters abgerufen werden, was im Anschluss durch mehrere asynchrone Web-Requests ausgeführt wird. Die Web-Requests werden parallel durchgeführt, damit sich die unter Umständen auftretenden Verzögerungen nicht aufsummieren. Nachdem alle Web-Requests beendet sind, werden die Mesh-Daten für die Fahrbahnen generiert und als **LaneGeometryTile** bereitgestellt.

Hinweis 5.2: Historisch gewachsen

Die **Start-Methode** der Klasse **HereTileTask** benötigt keine Asynchronität, da die Asynchronität bereits durch den **LaneGeometryTask** bereitgestellt wird. Dieser Umstand ist historisch bedingt und wurde noch nicht angepasst.
Die **HereHDProvider-Komponente** wurde vor der **Environment-Kopmponente** implementiert und die Asynchronität eingefügt. Durch spätere Überlegungen während der Entwicklung der **Environment-Komponente**, hat sich gezeigt dass eine Verschiebung der Asynchronität in die **Environment-Komponente** Vorteile bringt. Somit muss sich die konkrete Implementierung der **RoadsMapProvider-Komponente** nicht zwangsläufig um die Asynchronität kümmern, welche von der **Environment-Komponente** benötigt wird.
Die asynchrone Ausführung der **Start-Methode** zeigt keine negativen Auswirkungen, sollte aber schon Aufgrund der Nachvollziehbarkeit bei dem nächsten Refactoring entfernt werden. Das Entfernen der Asynchronität bezieht sich dabei nicht auf die innerhalb der **Start-Methode** ausgeführten parallelen Aufrufe.

5.2 GeoReference

Die **GeoReference-Komponente** ist zuständig für Distanzberechnungen und der Konvertierung von geodätischen Koordinaten in kartesische Koordinaten. Die Berechnung von Distanzen ist in der Geodäsie nicht trivial und abhängig von dem verwendeten Referenzellipsoid. Gleiches gilt für die Konvertierung von geodätischen in kartesische Koordinaten, welche in der Geodäsie als Kartenprojektionsverfahren bekannt sind. Daher wird, wie in der Abbildung 5.13 zu sehen ist, sowohl für die geodätischen Referenzsysteme als auch für die Projektionsverfahren eine Schnittstelle definiert. Die spezifischen Berechnungsverfahren können dann durch konkrete Implementierungen der jeweiligen Schnittstelle realisiert werden. Für die Implementierung der konkreten Klassen wird das Singleton-Pattern verwendet, da keine instanzspezifischen Daten gespeichert werden müssen.

Die statische Klasse **GeoReference** implementiert das Facade-Pattern und dient als zentraler Zugriffspunkt für alle Funktionsaufrufe auf der Komponente. Den Funktionsaufrufen auf der Fassade muss immer der Typ des zu verwendenden geodätischen Referenzsystems bzw. des Projektionsverfahrens mitgegeben werden, da sonst eine Zuordnung nicht möglich ist. Hierfür wird jeweils eine Enumeration zur Verfügung gestellt. Als Datenstruktur stellt die **GeoReference-Komponente** die Klasse **Coordinate** bereit. Die Klasse **Coordinate** speichert neben der Positionsangabe, auf welchem Referenzsystem die Koordinate basiert.

Betrachtet man das Interface **I_MapProjection** sieht man dort noch die Klasse **Vector**. Die Klas-

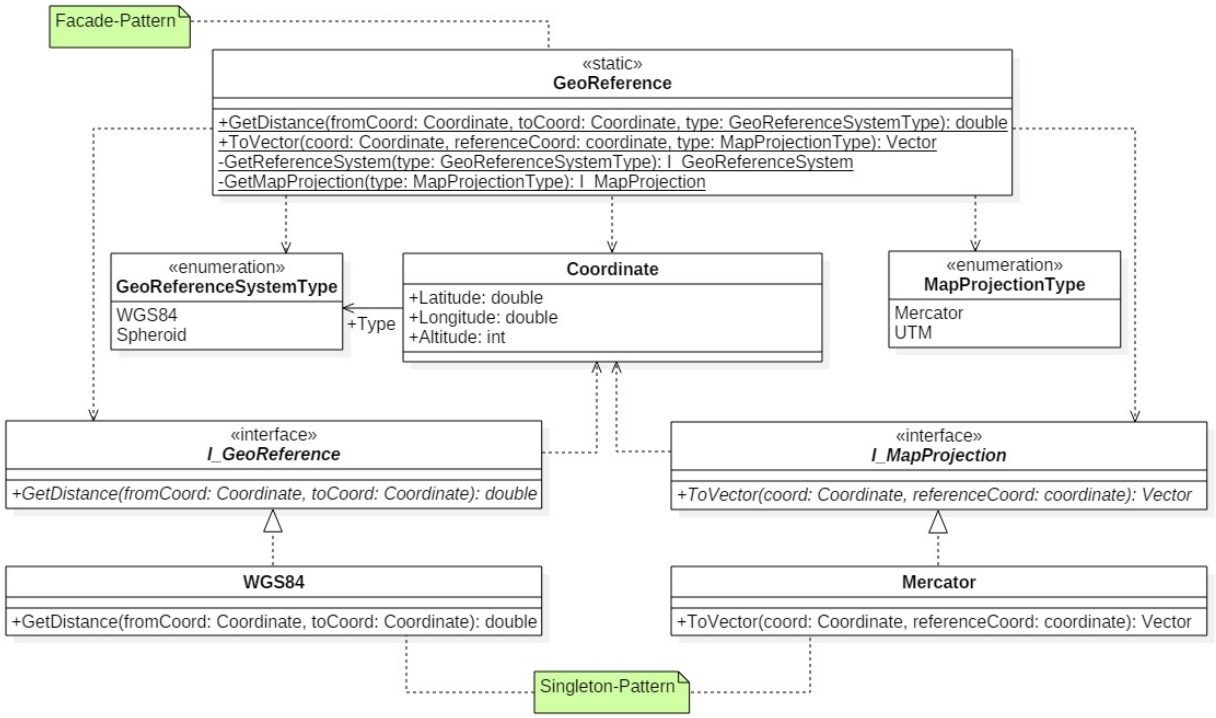

Abbildung 5.13: Klassendiagramm der Komponente GeoReference

se **Vector** wird nicht neu implementiert, sondern es wird die von der 3D-Engine zur Verfügung gestellte Vektoren-Klasse verwendet (im Fall Unity3D ist dies die Klasse**Vector3**).
Die Komponente **GeoReference** wird als Bibliothek bereitgestellt und ist als einzige vollständig unabhängig von allen anderen Komponenten, wodurch diese problemlos in anderen Projekten zum Einsatz kommen kann.

5.2.1 Distanz berechnen

Klasse: GeoReference
Methode: GetDistance(Coordinate, Coordinate, GeoReferenceSystemType)

Die Distanzberechnung zwischen zwei geodätischen Koordinaten benötigt als Eingabeparameter eine Start-Koordinate, eine End-Koordinate und den zu verwendenden geodätischen Referenzsystem-Typ. Die Angabe des geodätischen Referenzsystem-Typs ist erforderlich, damit das richtige geodätische Referenzsystem für die Berechnung ausgewählt werden kann. Anschließend wird die Distanzberechnung mit dem ermittelten geodätischen Referenzsystem durchgeführt. Das Ergebnis der Aktionsfolge ist die Distanz zwischen den Eingabe-Koordinaten in Meter.

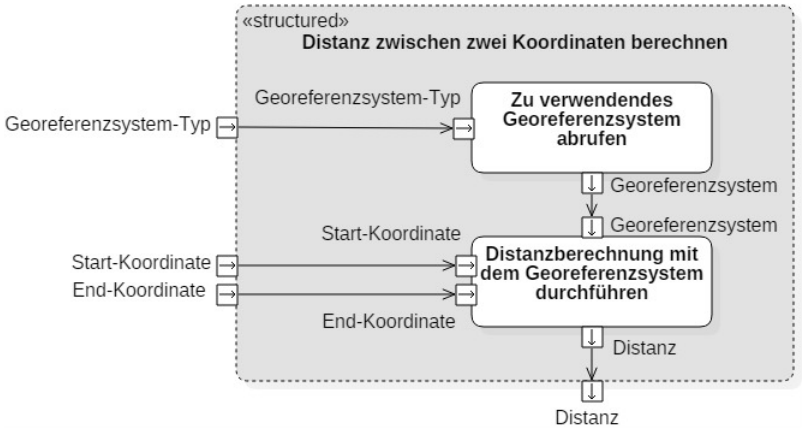

Abbildung 5.14: Aktivitätsdiagramm: Distanzberechnung

Der Ablauf der Distanzberechnung wird in Abbildung 5.14 gezeigt und die Zuordnung der Aktionen im Aktivitätsdiagramm zu den Methoden der konkreten Implementierung findet sich in Tabelle 5.10.

Methodenzuordnung		
Aktion	Klasse	Methode
Zu verwendendes geodätisches Referenzsystem abrufen	GeoReference	GetGeoReferenceSystem
Distanzberechnung mit dem geodätischen Referenzsystem ausführen	z.B. WGS84	GetDistance

Tabelle 5.10: Zuordnung der Aktionen zu den Methoden der Klassen

5.2.2 Koordinate konvertieren

Klasse: GeoReference
Methode: ToVector(Coordinate, Coordinate, MapProjectionType)

Bei der Konvertierung einer geodätischen in eine kartesische Koordinate, werden drei Eingangs-Parameter erwartet. Die Angabe des Kartenprojektions-Typ dient der Auswahl des Projektionsverfahrens und die geodätische Koordinate ist die zu konvertierende Koordinate. Die Referenz-Koordinate gibt den Bezugspunkt für die Konvertierung an. Der Erdmittelpunkt wird durch die Koordinate (0, 0, 0) repräsentiert, es kann aber auch ein anderer Bezugspunkt gewählt werden, z.B. um die Koordinaten bezüglich des Mittelpunkts eines Kartenabschnitts zu bestimmen. Das Ergebnis der Konvertierung ist somit eine kartesische Koordinate mit der Referenz-Koordinate als Ursprung.

Die Abbildung 5.15 stellt den allgemeinen Ablauf der Funktion in Form eines Aktivitätsdiagramms dar, während in der Tabelle 5.11 die Aktionen zu den Methoden der konkreten Implementierung zugeordnet werden.

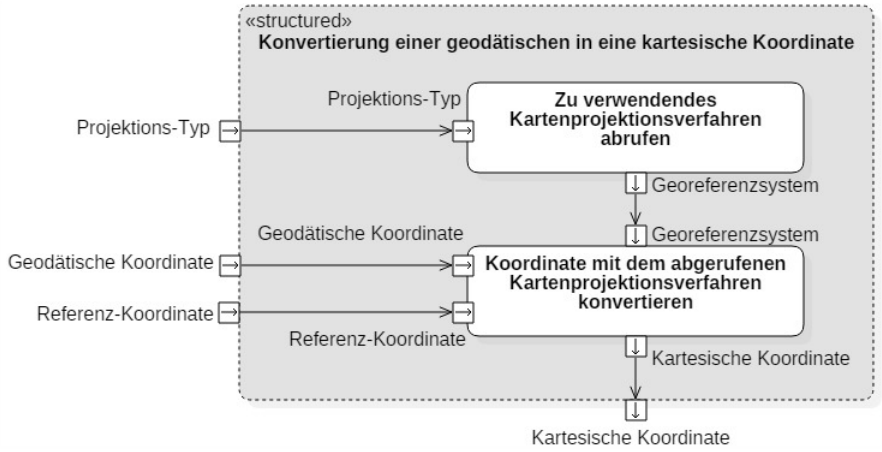

Abbildung 5.15: Aktivitätsdiagramm: Koordinate konvertieren

Methodenzuordnung		
Aktion	Klasse	Methode
Zu verwendendes Kartenprojektionsverfahren abrufen	GeoReference	GetMapProjection
Koordinate mit dem abgerufenen Kartenprojektionsverfahren konvertieren	z.B. Mercator	ToVector

Tabelle 5.11: Zuordnung der Aktionen zu den Methoden der Klassen

5.3 GPS

Aufgabe der **GPS-Komponente** ist das Bereitstellen der aktuellen Position und Orientierung. Um die Komponente einfach zu halten, handelt es sich hier um eine einfache Sammlung von Klassen, die die Schnittstelle **I_GPS** implementieren. Die Schnittstellendefinition wird von der **Environment-Komponente** bereitgestellt, welche eine einheitliche Schnittstelle für das GPS erst erforderlich macht. Das zugehörige Klassendiagramm ist in Abbildung 5.16 zu sehen.

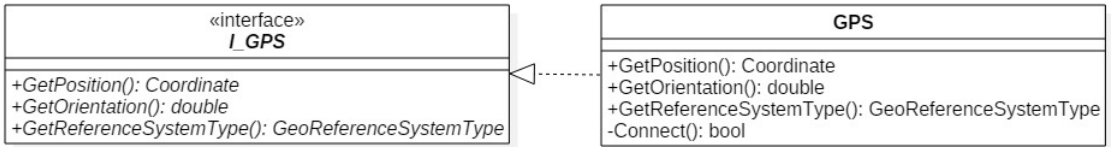

Abbildung 5.16: Klassendiagramm der Komponente GPS

Eine Erläuterung der Schnittstellen-Methoden befindet sich im Abschnitt 5.1. Die Komponente **GPS** wird als Bibliothek bereitgestellt.

5.3.1 Abfrage der Position

Klasse: GPS
Methode: GetPosition

Die Abbildung 5.17 zeigt den Ablauf einer Positionsabfrage an die **GPS-Komponente**. Wird die aktuelle Position bei der Komponente angefragt, prüft diese zunächst den Verbindungsstatus. Besteht eine Verbindung zur GPS-Hardware, dann wird die Koordinate ausgelesen und zurückgeben. Im anderen Fall wird ein Verbindungsfehler gemeldet, während versucht wird die Verbindung in einem parallelen Thread wieder aufzubauen.

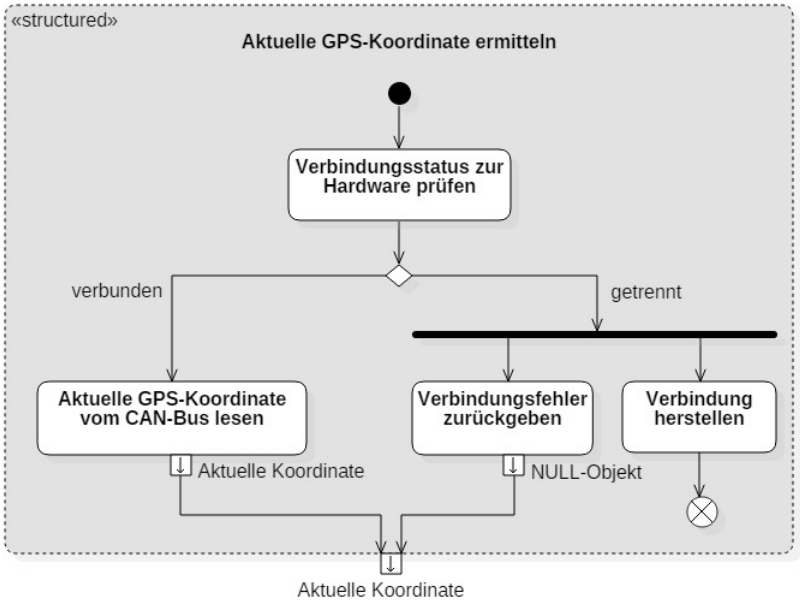

Abbildung 5.17: Aktivitätsdiagramm: Abruf der aktuellen Koordinate

Bei der Herstellung der Verbindung muss darauf geachtet werden, dass nur ein Thread zur Verbindungsherstellung erzeugt wird, ansonsten kann dies negative Folgen für die Leistung des Systems haben. Dies ist im Aktivitätsdiagramm allerdings nicht explizit dargestellt, sondern verbirgt sich mit in der Aktion Verbindung herstellen.

5.3.2 Abfrage der Orientierung

Klasse: GPS
Methode: GetOrientation

Die Abfrage der Orientierung ist fast vollständig identisch mit der Positionsabfrage. Der Ablauf unterscheidet sich ausschließlich durch den abgefragten Wert. Die Abfrage der Orientierung zeigt Abbildung 5.18. Eine weitere Kommentierung wird aufgrund der Ähnlichkeit zur Positionsabfrage ausgelassen.

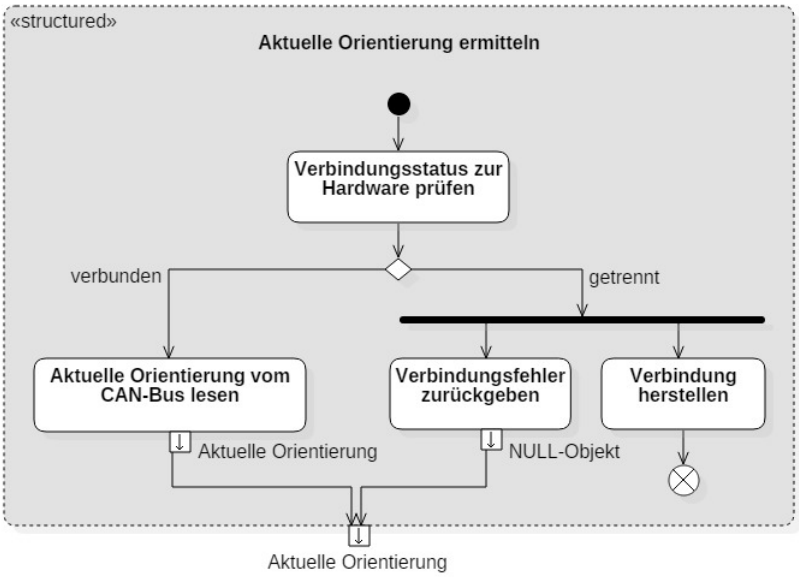

Abbildung 5.18: Aktivitätsdiagramm: Abruf der aktuellen Orientierung

Die Implementierung der konkreten GPS-Klasse ist ein Wrapper, welcher eine bereits bestehende Komponente aus einem abgeschlossenen Projekt verwendet. Die Original-Klasse wurde nur minimal angepasst, um den Anforderungen an die **I_GPS-Schnittstelle** zu genügen. Aus diesem Grund erfolgt hier keine Zuordnung der Aktionen zu Funktionsaufrufen.

5.4 RoadsMapProvider

Die **RoadsMapProvider-Komponente** hat mehrere Aufgaben zu erfüllen. Zunächst müssen die Fahrbahndaten von dem Server des Kartenanbieters heruntergeladen werden. Die heruntergeladenen Daten sind anschließend in Mesh-Daten zu konvertieren. Dies bedeutet, dass die geodätischen Koordinaten in kartesische Koordinaten (Vertices) zu konvertieren sind. Für die Konvertierung der Koordinaten wird die Komponente **GeoReference** genutzt. Außerdem müssen die UVs für die Texturierung berechnet sowie die Polygonflächen (Triangles) bestimmt werden. Damit die **Environment-Komponente** die Mesh-Daten verarbeiten kann, sind die Daten danach in ein homogenes Datenformat zu übertragen. Das homogene Datenformat wird durch die Klasse **LaneGeometryTile** beschrieben und von der Komponente **Environment** bereitgestellt (siehe Abschnitt 5.1.2 "LaneGeometryTile").

Um die Funktionsweise der Komponente zu verdeutlichen, wird hier der konkrete Kartenanbieter *Here* herangezogen. Die Klassenstruktur der Komponente wird in Abbildung 5.19 gezeigt.

5.4.1 HereHDProvider

Eine konkrete Implementierung der **RoadsMapProvider-Komponente** muss die Schnittstelle **I_RoadsMapProvider** implementieren. Die Schnittstellendefinition wird von der **Environment-Komponente** bereitgestellt und die für die Methoden der Schnittstelle zu implementierenden Aufgaben

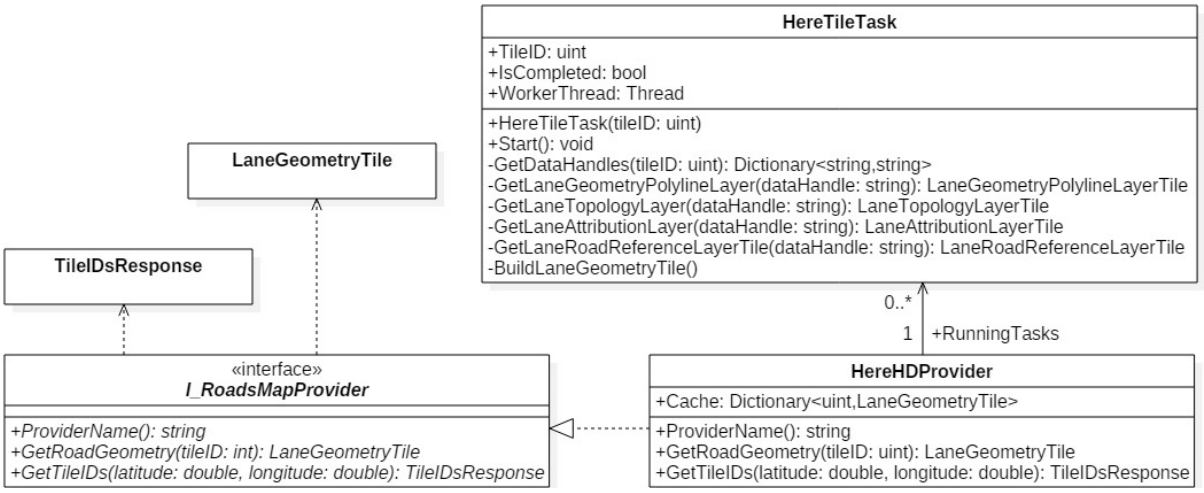

Abbildung 5.19: Klassendiagramm der Komponente RoadsMapProvider am konkreten Beispiel für den Anbieter Here

werden in 5.1.1 "Schnittstellenspezifikationen" beschrieben.

Die Klasse **HereHDProvider** koordiniert die Anfragen für die Kartenabschnitte und hält hierzu zwei Caches bereit. Ein Cache ist für die Verwaltung der aktuell laufenden Tasks, und der zweite ist für die Verwaltung der vollständig abgerufenen und vorverarbeiteten **LaneGeometryTile-Instanzen**. Die Attribute sind noch einmal in der Tabelle 5.12 abgebildet.

Klasse: HereHDProvider	
Attribut	Zweck
RunningTasks	Eine Liste von **HereTileTask-Instanzen**. Wird verwendet um sicherzustellen, dass für jede Anfrage nach einem Kartenabschnitt nur maximal ein Task läuft.
Cache	Ein Dictionary, welches zu jedem vollständig abgerufenen Kartenabschnitt die ID und die vorverarbeitete **LaneGeometryTile-Instanz** vorhält.

Tabelle 5.12: Attribute der Klasse HereHDProvider

Die Methode **ProviderName** liefert in der konkreten Implementierung den String "HereHD" als Namen zurück und die Methode **GetTileIDs** führt den in Beispiel 2.1 und Beispiel 2.2 beschriebenen Algorithmus zur Berechnung der TileIDs durch. Der aufwändigere Ablauf der Methode **GetRoadGeometry** wird in Abschnitt 5.4.3 verallgemeinert anhand eines Aktivitätsdiagramms erläutert.

5.4.2 HereTileTask

Die Klasse **HereTileTask** verrichtet den Großteil der Arbeit und ist zuständig für den Abruf der Daten vom Server des Kartenanbieters, sowie der darauffolgenden Konvertierung der Daten in das homogene Format **LaneGeometryTile**. Die Klasse **LaneGeometryTile** wird ebenfalls in der **Environment-Komponente** definiert und dort erläutert (siehe Abschnitt 5.1.2 "LaneGeometryTile").

Bei der Konstruktion wird der Klasse **HereTileTask** die ID eines Kartenabschnitts (TileID) übergeben. Durch Aufruf der Methode **Start** wird ein Worker-Thread erzeugt, der die Arbeit verrichtet. Der Ablauf ist in der Abbildung 5.12 dargestellt. Ein Kartenabschnitt in der *Here HD Live Map* besitzt derzeit zehn Layer, von denen in der Abbildung lediglich vier abgerufen werden. In der Implementierung werden jedoch alle zehn abgerufen, aus Gründen der Übersichtlichkeit wurde aber auf die Darstellung verzichtet. Tatsächlich verwendet werden derzeit allerdings nur die zwei Layer "LaneGeometryPolyline" und "LaneAttributes", die verbleibenden acht Layer wurden während der Implementierungsphase zu Analysezwecken abgerufen.

Klasse: HereTileTask	
Attribut	Zweck
TileID:	Identifiziert den abzurufenden Kartenabschnitt.
IsCompleted:	Zeigt an, ob der Thread vollständig abgearbeitet wurde.
WorkerThread:	Der für das Abrufen und verarbeiten der Daten zuständige Thread.

Tabelle 5.13: Attribute der Klasse HereTileTask

Die Tabelle 5.13 erläutert die Attribute der Klasse **HereTileTask**.

Die Methode **GetDataHandles** ruft die für die Layer spezifischen Identifikatoren per Web-Request vom Server des Kartenanbieter *Here* ab. Die verbleibenden Get-Methoden nutzen die dataHandles, um jeweils in einem eigenen Thread die Layer-Daten vom Server abzurufen. Sobald alle Layer-Daten abgerufen sind, kann mit der Methode **BuildLaneGeometryTile** die Verarbeitung der Rohdaten in die Mesh-Daten durchgeführt werden.

5.4.3 Abfrage der Fahrbahngeometrie-Daten

Klasse: HereHDProvider
Methode: GetRoadGeometry

Das Aktivitätsdiagramm in Abbildung 5.20 zeigt den Ablauf eines Aufrufs der Methode **GetRoadGeometry** auf eine Instanz der Klasse **HereHDProvider**.
Zunächst benötigt die Methode die Angabe der ID für den zu verarbeitenden Kartenabschnitt (TileID). Anhand der TileID wird dann geprüft, ob die Daten bereits vorverarbeitet im Cache liegen. Ist dies der Fall, können die Daten direkt zurückgeliefert werden. Ansonsten wird geprüft, ob die Daten aus dem lokalen Speicher abgerufen werden können. Liegen die Daten im lokalen Speicher vor, werden diese geladen, dem Cache hinzugefügt und als Ergebnis zurückgeliefert. Sind die Daten auch nicht im lokalen Speicher zu finden, dann werden die Daten vom Server des Kartenanbieter heruntergeladen und der Vorverarbeitungsprozess angestoßen (siehe hierzu Abbildung 5.21). Nachdem die Daten geladen und vorverarbeitet wurden, werden diese auf dem lokalen Speicher für zukünftige Zugriffe abgelegt, dem Cache hinzugefügt und als Ergebnis zurückgeliefert.

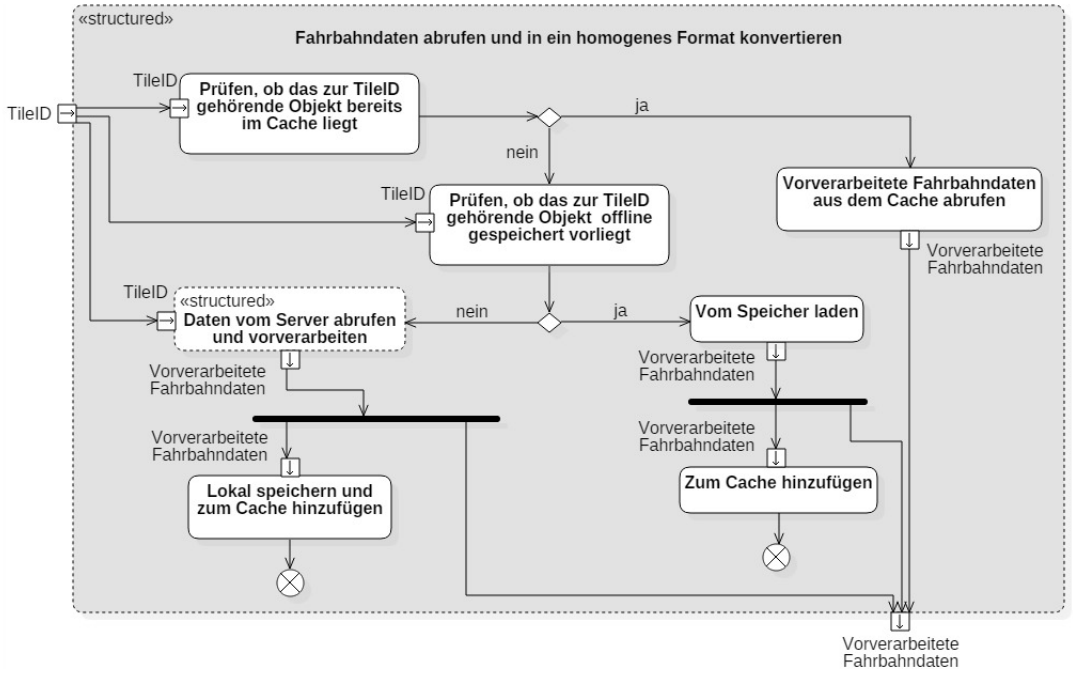

Abbildung 5.20: Aktivitätsdiagramm: Abfrage der Fahrbahngeometrie-Daten

Die Tabelle 5.14 ordnet den Aktionen aus dem Aktivitätsdiagramm in Abbildung 5.20 die korrespondierenden Methoden und Klassen der **HereHDProvider-Komponente** zu.

Methodenzuordnung		
Aktion	Klasse	Methode
Prüfen, ob das zur TileID gehörende Objekte bereits im Cache liegt	HereHDProvider	GetRoadsGeometry
Vorverarbeitete Fahrbahndaten aus dem Cache abrufen	HereHDProvider	GetRoadsGeometry
Prüfen, ob das zur TileID gehörende Objekt offline gespeichert vorliegt	HereTileTask	Konstruktor
Vom Speicher laden	HereTileTask	Konstruktor
Zum Cache hinzufügen	HereHDProvider	GetRoadsGeometry
Daten vom Speicher abrufen und vorverarbeiten	siehe Tabelle 5.15	siehe Tabelle 5.15
Lokal speichern und zum Cache hinzufügen	HereHDProvider	GetRoadsGeometry

Tabelle 5.14: Attribute der Klasse HereTileTask

In Abbildung 5.21 wird die strukturierte Aktion **Daten vom Server abrufen und vorverarbeiten** detailliert dargestellt. Für die Durchführung der Aktion wird die TileID benötigt. Mit der TileID werden dann per Web-Request die dataHandles abgerufen, welche von den folgenden Aktionen benötigt werden, um die Daten der Layer ebenfalls per Web-Request abzurufen. Mit Hilfe der Komponente **GeoReference** (nicht im Diagramm dargestellt) werden die geodätischen Koordinaten des Geometry-Layer in kartesische Koordinaten konvertiert. Nach der Konvertierung der Mesh-Daten, werden diese mit den Attributdaten kombiniert. Die Attributdaten enthalten unter anderem die Informationen zu

den Fahrbahnmarkierungen, welche hier genutzt werden. Die vorverarbeiteten Fahrbahndaten werden als Ergebnis zurückgeliefert. Die vorverarbeiteten Fahrbahndaten entsprechen in der Implementierung **HereHDProvider** der Klasse **LaneGeometryTile**.

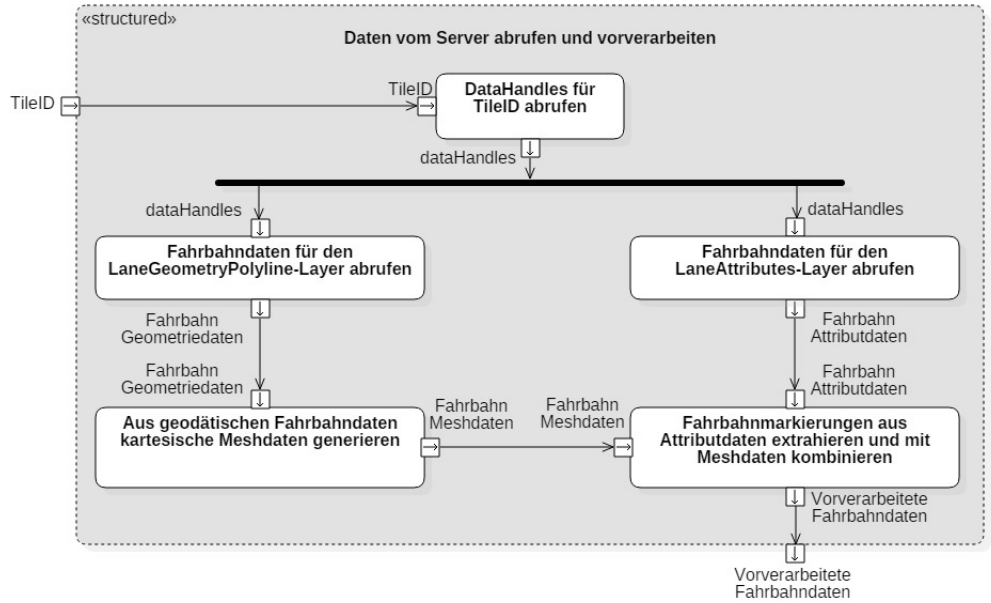

Abbildung 5.21: Aktivitätsdiagramm: Abruf der Fahrbahndaten vom Server und Vorverarbeitung

Die Tabelle 5.15 zeigt die Zuordnung der Aktionen des Aktivitätsdiagramms in Abbildung 5.21 zu den Methoden der Klasse **HereHDProvider**.

Methodenzuordnung		
Aktion	Klasse	Methode
DataHandles für TileID abrufen	HereTileTask	GetDataHandles
Fahrbahndaten für den LaneGeometryPoloyline-Layer abrufen	HereTileTask	GetLaneGeometryPolylineLayer
Fahrbahndaten für den LaneAttributes-Layer abrufen	HereTileTask	GetLaneAttributionLayer
Aus geodätischen Fahrbahndaten kartesische Mesh-Daten generieren	HereTileTask	BuildLaneGeometryTile
Fahrbahnmarkierungen aus Attributdaten extrahieren und mit Mesh-Daten kombinieren	HereTileTask	BuildLaneGemoetryTile

Tabelle 5.15: Attribute der Klasse HereTileTask

6 Implementierung

Zunächst wird in diesem Kapitel auf einige, während der Implementierung aufgetretene, Probleme bzw. Hindernisse eingegangen. Die Entscheidung zuerst auf die Probleme einzugehen ist gefallen, damit die später vorgestellten Code-Fragmente leichter nachzuvollziehen sind.

6.1 Probleme während der Implementierung

6.1.1 Here HD Live Map Integration

Die Layer-Daten für die einzelnen Kartenabschnitte werden von *Here* im ProtocolBuffer-Format (Version 3.0) angeboten. Das NuGet-Package für die ProtocolBuffer der Version 3.0, benötigt in C# eine .NET-Version höher als .NET 3.5. Unity 3D setzt allerdings eine Mono-Implementierung für die Nutzung des .NET-Frameworks ein. Die in Unity 3D verwendete Mono-Implementierung kann derzeit aber nicht mit Funktionen umgehen, welche eine .NET-Version höher als 3.5 benötigen. Somit ist es nicht ohne Umwege möglich, die Layer-Daten direkt über den Kontext der Unity-Engine abzurufen.

Lösung:
Um dieses Problem zu umgehen, wurde ein Service implementiert, welcher die Daten vom Here-Kartenserver abruft und in ein geeignetes Format konvertiert. Da für die weitere Nutzung nur die Daten und nicht die ProtocolBuffer-Methoden benötigt werden, handelt es sich bei dem gewählten Format, um eine 1:1 Kopie der Datenstruktur ohne die Methoden der ProtocolBuffer-Klassen. Aus Sicht des implementierten Systems steht der Service stellvertretend für den Kartenserver. Die zu beachtenden Unterschiede sind, dass die Verbindung zum Service auf TCP/IP aufbaut und die Layer-Daten nicht einzeln, sondern kumuliert (alle zehn Layer) in einem Gesamtpaket ausgeliefert werden.

6.1.2 GPS Integration

Die für die Anbindung des GPS benötigten Bibliotheken verwenden ebenfalls eine .NET-Version höher als .NET 3.5. Somit kann auch das abrufen der GPS-Koordinaten, nicht direkt über den Kontext der Unity-Engine erfolgen.

Lösung:
Auch hier wurde ein Service für die Kommunikation mit dem GPS zwischengeschaltet. Die Daten (Längengrad, Breitengrad und Höhe über dem Referenzellipsoid) werden dabei direkt in den Datentyp **Coordinate** der Komponente **GeoReference** konvertiert und zurückgegeben. Der Zugriff auf den Service erfolgt über TCP/IP.

6.1.3 Service-Verbindungstyp

Zunächst war geplant, die Kommunikation mit den Services, auf Basis der Windows Communication Foundation (kurz: WCF) mit NamedPipes einzurichten. Die WCF ermöglicht eine einfache Einrichtung von Kommunikationskanälen auf verschiedenen Verbindungstypen (TCP/IP, UDP, HTTP, NamedPipes und weitere) und bietet die Möglichkeit den Verbindungstyp mit nur wenigen Schritten zu ändern oder weitere Verbindungstypen parallel anzubieten. Für die WCF wird die .NET-Version 3.5 oder höher benötigt. Wie sich nach einigen Tests und intensiver Recherche herausgestellt hat, unterstützt die von Unity eingesetzte Mono-Version nur die .NET Version 2.0 vollständig und einen Teil der .NET 3.5 Features. Die WCF gehört nicht dazu.

Lösung:
Anstelle der WCF wird der Vorgänger .NET-Remoting (ab .NET 2.0 verfügbar) mit dem Verbindungstyp TCP/IP verwendet.

Die Abbildung 6.1 zeigt das neue Verteilungsdiagramm mit den vorgenommenen Änderungen. Wie dort zu sehen ist, werden die Services auf dem gleichen System ausgeführt, sind aber von dem Rest des Systems entkoppelt.

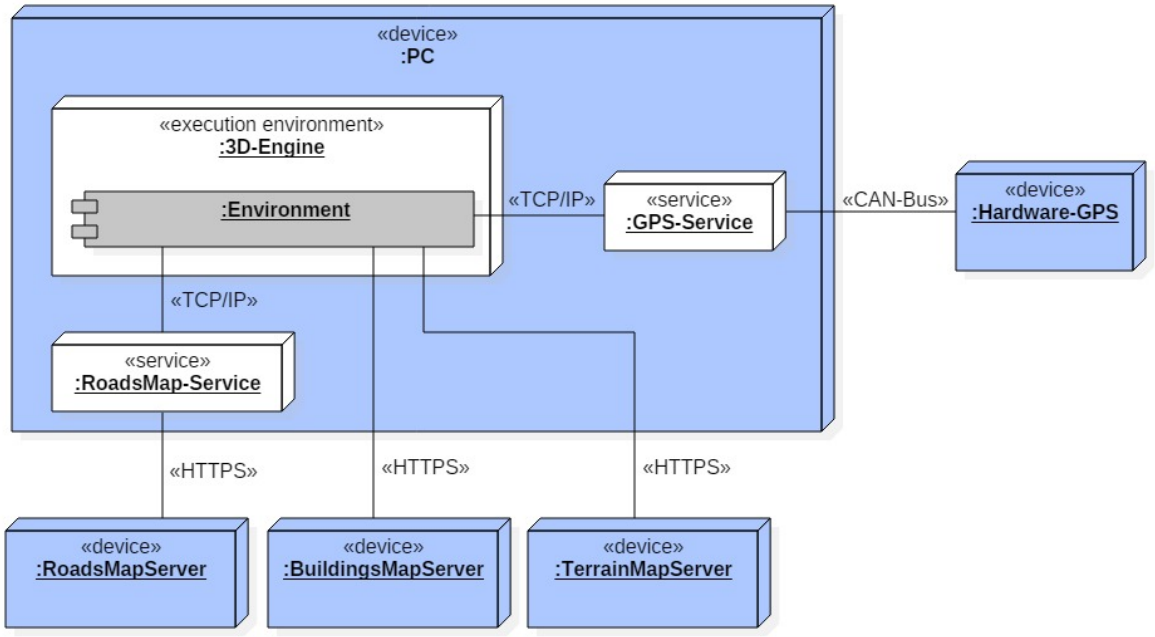

Abbildung 6.1: Verteilungsdiagramm inklusive Services

6.1.4 Fahrbahngeometrien

Der LaneGeometriePolyline-Layer der *Here HD Live Map* liefert zu jeder Boundary (siehe Abbildung 5.5) eine Liste von geodätischen Koordinaten, die in kartesische Koordinaten (Vertices) zu konvertieren

sind. Aus diesen Listen müssen Dreiecksflächen für die Modellierung der Fahrbahnen erzeugt werden. Eine einfache Methode ist es, aus je vier Vertices zwei Dreiecke zu bilden (siehe Abbildung 6.2).

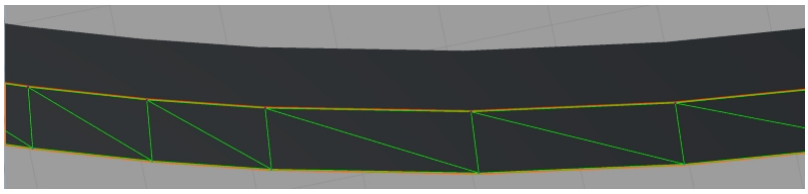

Abbildung 6.2: Je vier Vertices bilden zwei Dreiecke

Dieses Vorgehen funktioniert aber nur dann recht gut, wenn die linke und die rechte Boundary gleichviele Vertices und die Vertices auf beiden Boundaries etwa gleiche parametrische Abstände besitzen. Ansonsten kann es passieren, dass in Kurven Dreiecksflächen aufgespannt werden, welche die Grenzen der Boundaries überschneiden (siehe Abbildung 6.3).

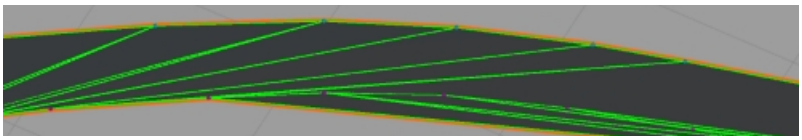

Abbildung 6.3: Vertices liegen innerhalb von Dreiecksflächen

Genau dieses Problem hat sich bei der Verarbeitung der Koordinaten-Listen der *Here HD Live Map* ergeben. Sowohl die Anzahl, als auch die parametrischen Abstände der Vertices zweier Boundaries, können massiv voneinander abweichen.

Lösung:
Beiden Boundaries werden zusätzliche Vertices hinzugefügt, so dass beide Boundaries gleichviele Vertices besitzen und diese auf beiden Boundaries die gleichen parametrischen Abstände vorweisen. Die Skizze in Abbildung 6.4 soll den Vorgang bildlich verdeutlichen. Die blauen Punkte sollen dabei die auf der jeweils anderen Boundary hinzugefügten Vertices darstellen.

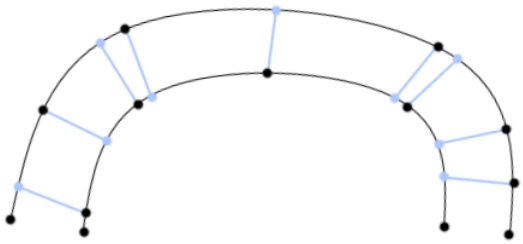

Abbildung 6.4: Parametrisches hinzufügen von Vertices

6.2 Code-Details

Dieser Abschnitt zeigt einige Code-Fragmente und erläutert die gezeigt Funktion oder gibt Aufschluss über die Abweichungen zu den im Kapitel Entwurf dargestellten Abläufen.

6.2.1 Projektstruktur

Wie in Abschnitt 6.1 "Probleme während der Implementierung" erwähnt, sind zu der geplanten Architektur zwei Services hinzugekommen, für dessen Implementierung weitere Projekte notwendig geworden sind. Die komplette Projektstruktur zeigt die Abbildung 6.5

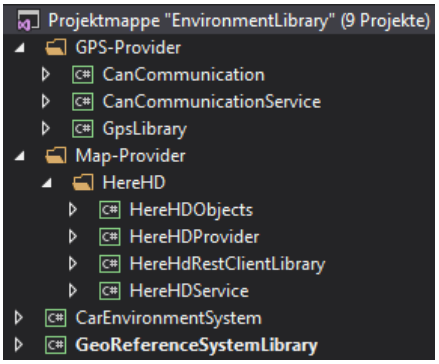

Abbildung 6.5: Projektstruktur der Implementierung

Die Tabelle 6.1 führt die für die Services hinzugefügten Projekte auf und gibt eine kurze Beschreibung zu deren Zweck, sowie eine Zuordnung zu der zugehörigen Komponente.

Services		
Projektname	Service für Komponente	Zweck
CanCommunication	GPS	Führt den Zugriff auf das Hardware-GPS aus und liest die aktuellen Daten über den CAN-Bus ein.
CanCommunicationService	GPS	Der Service selbst, welcher den Zugriff über ein Remoting-Objekt der CanCommunication-Komponente bereitstellt.
HereHdRestClientLibrary	HereHDProvider	Führt den Zugriff auf den Here-Kartenserver durch und konvertiert die Daten in ein .NET 2.0 taugliches Format.
HereHDObjects	HereHDProvider	Stellt die .NET 2.0 tauglichen Klassen, für den Austausch der Daten, zwischen der Komponente HereHDClient und HereHdRestClient bereit. Diese Komponente wird benötigt, da die Klassen in beiden Projekten verfügbar sein müssen und das Projekt HereHdRestClient auf .NET 4.6.1 basiert.

HereHDService	HereHDProvider	Der Service selbst, welcher den Zugriff über ein Remoting-Objekt der HereHdRestClientLibrary-Komponente bereitstellt.

Tabelle 6.1: Zuordnung der zusätzlichen Projekte für die Service und deren Zweck

Neben den zusätzlichen Projekten für die Services fällt auf, das einige Projektbezeichnungen nicht mit den Komponentenbezeichnungen übereinstimmen. Der Grund hierfür ist das die Projektbezeichnugen teilweise sehr lang sind und in dieser Arbeit häufig Erwähnung finden, daher wurden die Namen etwas gekürzt. Eine Zuordnung findet sich in Tabelle 6.2.

Namenszuordnung	
Komponentenname	Projektname
GPS	GpsLibrary
RoadMapProvider	HereHDProvider
Environment	CarEnvironmentSystem
GeoReference	GeoReferenceSystem

Tabelle 6.2: Zuordnung der Komponentennamen zu den Projektnamen

6.2.2 Parametrisches einfügen zusätzlicher Koordinaten

Im Folgenden wird der Code der Methode **AddPointsParametrically** näher erläutert. Mit dieser Methode werden jeder Boundary eines Fahrbahnabschnitts, zusätzliche Koordinaten hinzugefügt, so dass alle Boundaries gleichviele Koordinaten haben und diese parametrisch gleiche Abstände besitzen. Die Methode verwendet dazu zwei bisher unbenannte Klassen, diese werden zunächst beschrieben.

Die Klasse **LineString3dOffset** gehört zu den Datentypen, die von der Komponente **HereHDObjects** bereitgestellt werden und bei denen es sich, um die Konvertierung einer ProtocolBuffer-Klasse der *Here HD Live Map* handelt. Die Attribute der Klasse werden in Tabelle 6.3 erklärt.

Attribute der Klasse LineString3dOffset		
Attribut	Typ	Zweck
Here2DCoordinateDiffs	List<long>	Als long-Werte kodierte Liste von Differenzen zwischen Koordinaten. Der erste Wert der Liste gibt die Differenz zum Zentrum des zugehörigen Kartenabschnitts an. Jeder weitere Wert gibt die Differenz zur jeweils vorhergehenden Koordinate an. Die Werte müssen zunächst per XOR verknüpft und anschließend mit der in Beispiel A.10 beschriebenen Methode entschlüsselt werden.

CmFromWGS84EllipsoidDiffs	List<int>	Abweichung der Höhe vom WGS84-Referenzellipsoid, in Richtung der Normalen, im Punkt der Koordinate. Die Differenzwerte müssen auf gleiche Weise miteinander verknüpft werden, wie die Werte der Here2dCoordinateDiffs-Liste.

Tabelle 6.3: Attribute der Klasse LineString3dOffset

Die Klasse **CoordinateHereHD** erbt von der Klasse **Coordinate** der Komponente **GeoReference** und erweitert diese um Methoden, die für die Dekodierung der als long-Werte gelieferten Koordinaten benötigt werden (siehe Beispiel A.10). Die Klasse **CoordinateHereHD** wird in der Komponente **HereHDProvider** definiert und nur dort genutzt.

Die Abbildung 6.6 zeigt den ersten Ausschnitt des Codes.

```
230 private List<List<CoordinateHereHD>> AddPointsParametically(List<HdExt.Geometry.LineString3dOffset> boundaries, CoordinateHereHD tileCenter)
231 {
232     // Dekodieren
233     List<List<CoordinateHereHD>> boundaryCoordLists =
234         boundaries.Select(boundary => CoordinateHereHD.EncodedOffsetsToListOfCoords(boundary, tileCenter)).ToList();
235
236     // Parametrische Abstände berechnen
237     List<List<double>> parametricPointsLists = new List<List<double>>(); // Hält für jede Boundary eine Liste der parametrischen Abstände
238     foreach (List<CoordinateHereHD> boundaryCoords in boundaryCoordLists)
239     {
240         List<double> coordDistances = new List<double>(); // Entfernung auf dem Referenzellipsoid zu der ersten Koordinate der Boundary
241         coordDistances.Add(0); // Entfernung der ersten Koordinate zur ersten Koordinate ist immer 0
242         for (int i = 1; i < boundaryCoords.Count(); i++)
243         {
244             CoordinateHereHD prev = boundaryCoords[i - 1];
245             CoordinateHereHD actual = boundaryCoords[i];
246             double distance = GeoRef.GetDistance(prev.Latitude, prev.Longitude, actual.Latitude, actual.Longitude, actual.Type);
247             coordDistances.Add(coordDistances[i - 1] + distance);
248         }
249         double inverseLength = 1.0 / coordDistances.Last();
250         List<double> parametrics = coordDistances.Select(dist => inverseLength * dist).ToList();
251         parametrics[parametrics.Count() - 1] = 1.0; // Entfernung der letzten Koordinate zur ersten Koordinate ist immer 1.0
252         parametricPointsLists.Add(parametrics);
253     }
```

Abbildung 6.6: Funktion AddPointsParametrically - Teil 1

Der Code ist folgendermaßen zu interpretieren:

- **Zeile 233-234:** Die verschlüsselten, als Differenzwert-Listen übergebenen, Koordinaten werden entschlüsselt und in der Variablen **boundaryCoordLists** abgelegt.
- **Zeile 237:** Die Variable **parametricPointsLists** wird erzeugt, welche für jede Boundary die Liste der parametrischen Abstände speichert.
- **Zeile 238-253:** In der foreach-Schleife werden die parametrischen Abstände für jede Boundary berechnet und der **parametricPointsLists** hinzugefügt.
  - **Zeile 240-248:** In der for-Schleife wird für jede Koordinate einer Boundary, der Abstand zur ersten Koordinate der Boundary berechnet. Die Berechnung der Distanzen erfolgt mit Hilfe der **GeoReference-Komponente** und wird auf dem entsprechenden Referenzellipsoiden durchgeführt.
  - **Zeile 249-251:** Die zuvor berechneten Distanzen werden werden in parametrische Abstände konvertiert, indem jede Distanz mit der inversen Länge der jeweiligen Boundary multipliziert wird.

- **Zeile 252:** Die Listen der parametrischen Abstände werden der **parametricPointsLists** hinzugefügt.

Mit der **parametricPointsLists** stehen nun für jede Boundary die parametrischen Abstände zur Verfügung. Die einzelnen Listen werden dann miteinander kombiniert, um eine Liste mit allen parametrischen Abständen zu erhalten. Dies ist in der Abbildung 6.7 mit dem zweiten Codausschnitt zu sehen.

```
255  // Eine Liste mit allen parametrischen Abständen erzeugen
256  List<double> combinedParametricPointList = new List<double>();
257  foreach(List<double> list in parametricPointsLists)
258  {
259      combinedParametricPointList.AddRange(list);
260  }
261  combinedParametricPointList.Sort();
262  combinedParametricPointList = combinedParametricPointList.Distinct().ToList();
263
```

Abbildung 6.7: Funktion AddPointsParametrically - Teil 2

Der Code ist folgendermaßen zu interpretieren:

- **Zeile 256:** Es wird die Variable **combinedParametricPointList** erstellt, welche alle parametrischen Abstände speichert.
- **Zeile 257-260:** Die parametrischen Abstände aller Boundaries werden in die **combinedParametricePointList** kopiert.
- **Zeile 261:** Die **combinedParametricePointList** wird sortiert.
- **Zeile 262:** Eventuell vorhandene doppelte Einträge werden aus der **combinedParametricePointList** entfernt.

Als Ergebnis erhält man mit der **combinedParametricePointList** eine sortierte Liste mit allen parametrischen Abständen, ohne doppelte Einträge. Im dritten und letzten Abschnitt werden den Boundaries die zusätzlichen parametrischen Koordinaten hinzugefügt. Der dazugehörige Code ist in Abbildung 6.8 zu sehen.

Der Code ist folgendermaßen zu interpretieren:

- **Zeile 265:** Die Variable **response** für das Rückgabe-Objekt wird erstellt.
- **Zeile 266- 310:** Die for-Schleife wird für jede Boundary (bzw. Liste mit parametrischen Abständen) einmal durchlaufen. Dabei wird für jede Boundary eine neue Liste mit Koordinaten erstellt, welche für jeden parametrischen Punkt der **combinedParametricPointList** eine Koordinate enthält.
  - **Zeile 268:** Die Variable **parametricPoints** wird erstellt und die Liste der parametrischen Punkte, für die aktuell zu verarbeitende Boundary zugewiesen.
  - **Zeile 269:** Die Variable **boundaryCoords** wird erstellt und die Liste der Koordinaten, für die aktuell zu verarbeitende Boundary zugewiesen.
  - **Zeile 270:** Die Variable **responseBoundary** wird erstellt und als neue Boundary für das Rückgabe-Objekt verwendet.
  - **Zeile 273-306:** Die for-Schleife iteriert über alle parametrischen Punkte der aktuellen Boundary. Dabei wird geprüft, ob zwischen dem aktuellen parametrischen Punkt und dem nächsten parametrischen Punkt, weitere parametrische Punkte in der **combinedParametricPointList** zu finden sind. Falls ein Punkt in der **combinedParametricPointList** gefunden wird, der nicht in der **parametricPoints-Liste** existiert, dann wird eine neue Koordinate durch lineare Interpolation berechnet und in die **responseBoundary** eingefügt. Wenn

```
264  // Allen Boundaries neue Koordinaten, basierend auf den parametrischen Abständen hinzufügen
265  List<List<CoordinateHereHD>>  response = new List<List<CoordinateHereHD>>();
266  for (int ppListsIdx = 0; ppListsIdx < parametricPointsLists.Count(); ppListsIdx++)
267  {
268      List<double> parametricPoints = parametricPointsLists[ppListsIdx];
269      List<CoordinateHereHD> boundaryCoords = boundaryCoordLists[ppListsIdx];
270      List<CoordinateHereHD> responseBoundary = new List<CoordinateHereHD>();
271
272      int combinedPpIdx = 0;
273      for (int ppIdx = 0; ppIdx < parametricPoints.Count() - 1; ppIdx++)
274      {
275          double actual = parametricPoints[ppIdx];
276          double next = parametricPoints[ppIdx + 1];
277          if (next == actual) { continue; }
278
279          for (; combinedPpIdx < combinedParametricPointList.Count(); combinedPpIdx++)
280          {
281              double point = combinedParametricPointList[combinedPpIdx];
282
283              if (point == actual)
284              {
285                  responseBoundary.Add(boundaryCoords[ppIdx].Clone());
286              }
287              else if (point > actual && point < next)
288              {
289                  double step =  (point - actual) / (next - actual);
290
291                  double latStep = (boundaryCoords[ppIdx + 1].Latitude - boundaryCoords[ppIdx].Latitude) * step;
292                  double lngStep = (boundaryCoords[ppIdx + 1].Longitude - boundaryCoords[ppIdx].Longitude) * step;
293                  double altStep = (boundaryCoords[ppIdx + 1].Altitude - boundaryCoords[ppIdx].Altitude) * step;
294
295                  var coord = new CoordinateHereHD(tileCenter.Type);
296                  coord.SetGeographic(boundaryCoords[ppIdx].Latitude + latStep,
297                      boundaryCoords[ppIdx].Longitude + lngStep,
298                      Convert.ToInt32(boundaryCoords[ppIdx].Altitude + altStep));
299                  responseBoundary.Add(coord);
300              }
301              else // (point >= next)
302              {
303                  break;
304              }
305          }
306      }
307      responseBoundary.Add(boundaryCoords.Last());
308
309      response.Add(responseBoundary);
310  }
311  return response;
```

Abbildung 6.8: Funktion AddPointsParametrically - Teil 3

der aktuelle parametrische Punkt der **parametricPoints-Liste**, mit dem aktuellen parametrischen Punkt der **combinedParametricPointList** übereinstimmt, dann wird die entsprechende Koordinate aus der Liste **boundaryCoords** zu der **responseBoundary** hinzugefügt.

- **Zeile 307:** Die letzte Koordinate der Boundary wird der **responseBoundary** hinzugefügt.
- **Zeile 308:** Die neue Boundary wird dem Rückgabe-Objekt **response** hinzugefügt.

6.2.3 Service-Implementierung

Als Beispiel für die Implementierung des Service, wird der GPS-Service verwendet. Für die Generierung des Service wurde in C# die Projektvorlage "Windows-Dienst (.NET-Framework)" verwendet. Der notwendige Code reduziert sich damit auf die in Abbildung 6.9 dargestellten Zeilen.
Wie in der Abbildung zu sehen ist, wird zunächst ein Hashtable **props** erzeugt und dort die IP-Adresse und der Port hinzugefügt. Mit der Hashtable **props** wird dann ein **TcpServerChannel** erzeugt und dieser anschließend bei den **ChannelServices** registriert. Durch den Hashtable **props** wird bei der Erzeugung des **TcpServerChannel** angegeben, an welche IP-Adresse und Port dieser sich binden soll. Im letzten Schritt, wird das Remote-Objekt über die **RemotingConfiguration** registriert und somit anderen Applikationen bekanntgegeben. Für die Registrierung wird der Typ des Remote-Objekts, der

```
namespace CanCommunicationService
{
    public partial class Service1 : ServiceBase
    {
        public Service1()
        {
            InitializeComponent();
        }

        protected override void OnStart(string[] args)
        {
            // Serverseitig aktiviertes Remote-Objekt registrieren
            IDictionary props = new Hashtable();
            props["bindTo"] = "127.0.0.1";
            props["port"] = 55555;

            TcpServerChannel channel = new TcpServerChannel(props, new BinaryServerFormatterSinkProvider());
            ChannelServices.RegisterChannel(channel, false);
            RemotingConfiguration.RegisterWellKnownServiceType(typeof(CanCommunication.CanCommunicationFassade), "CanCommunication", WellKnownObjectMode.Singleton);
        }

        protected override void OnStop()
        {
        }
    }
}
```

Abbildung 6.9: Code der Implementierung des GPS-Service (Komponente: CanCommunicationService)

Name unter dem das Remote-Objekt von anderen Applikationen gefunden werden soll und der Objekt-Modus benötigt. Als Objekt-Modus kann zwischen Singleton und SingleCall gewählt werden. Sowohl für den GPS-Service als auch für HereHD-Service wurde sich für den Singleton-Modus entschieden. Im Fall des GPS-Service, würde im SingleCall-Modus für jede Anfrage an das Remote-Objekt, eine neue Verbindung über den CAN-Bus eingerichtet werden und im Fall des HereHD-Service, kann im SingleCall-Modus kein Caching angewendet werden, da jeder Aufruf ein neues Remote-Objekt erzeugen kann.

Den Zugriff auf das Remote-Objekt von der Seite des Clients, erhält man durch die in Abbildung 6.10 gezeigten Zeilen. Der Client muss zunächst ein **TcpChannel-Objekt** erzeugen und dieses eben-

```
// Serverseitiges Remote-Objekt
TcpChannel channel = new TcpChannel();
ChannelServices.RegisterChannel(channel, false);
GpsSystem = (I_GPS)Activator.GetObject(typeof(I_GPS), "tcp://localhost:55555/CanCommunication");
```

Abbildung 6.10: Code für den Zugriff auf das durch den Service bereitgestellte Remote-Objekt

falls über die **ChannelServices** registrieren. Dann kann das Remote-Objekt über die Klasse **Activator** abgerufen werden.

Um den Service nutzen zu können, muss dieser auch installiert werden. Die Installation kann manuell über das Visual Studio Tool "Developer Command Prompt" im Administrator-Modus durchgeführt werden. Dazu muss in das Verzeichnis mit der kompilierten ausführbaren Service-Datei navigiert werden und dort der Befehl `installutil.exe CanCommunicationService.exe` (für die Installation des GPS-Service) ausgeführt werden. Durch den Befehl `installutil.exe /u CanCommunicationService.exe`, kann der Service wieder deinstalliert werden.

6.2.4 Neue Komponenten einbinden

Um neue Komponenten einzubinden, welche von der Komponente **Environment** verwaltet werden sollen, muss der Code der Klasse **Environment** angepasst werden. Das hinzufügen geschieht in drei Schritten:

- Hinzufügen einer neuen Property, welche eine Instanz der neuen Komponente aufnehmen kann.
- Methodenaufrufe der neuen Komponente (über die Instanz der Property), in die Start-Methode der Klasse **Environment** aufnehmen.
- Methodenaufrufe der neuen Komponente (über die Instanz der Property), in die Update-Methode der Klasse **Environment** aufnehmen.

Die Abbildung 6.11 zeigt den aktuellen Code der Update-Methode der Klasse **Environment**.

```
337     /// <summary> Ist für die Aktualisierung der Szene zuständig. Um den Aufrufer nicht lange warten zu lassen wird hier, im ...
350     public void Update()
351     {
352         if (!IsStarted) { return; }
353
354         // Aktuelle Position ermitteln
355         GeoObj.GeographicCoordinate coord = Gps.GetGpsPosition();
356         ActualGpsPosition = coord;
357
358         #region Laden der Kartendaten und Erzeugen der GameObjects
359         Dictionary<uint, CacheItem> cache;
360         if (RoadProvider_ != null) {
361             Objects.TileIDsResponse tileIDs = RoadProvider_.GetTileIDs(coord.Latitude, coord.Longitude);
362             if (tileIDs != null)
363             {
364                 if (!Cache.TryGetValue(RoadProvider_.ProviderName(), out cache))
365                 {
366                     cache = new Dictionary<uint, CacheItem>();
367                     Cache.Add(RoadProvider_.ProviderName(), cache);
368                 }
369
370                 CacheItem item;
371                 if (!cache.TryGetValue(tileIDs.Center, out item))
372                 {
373                     item = new CacheItem(tileIDs.Center);
374                     cache.Add(tileIDs.Center, item);
375                     item.LoadLaneGeometry(RoadProvider_);
376                 }
377
378                 // Falls Kartenabschnitt bereit und Modelle noch nicht erzeugt, dann Modelle erzeugen
379                 if (item.LaneGeometryData != null && item.LaneGeometryParent != null)
380                 {
381                     item.CreateLaneGeometryGameObjects();
382                 }
383
384                 // Laden der Daten, welche voraussichtlich demnächst benötigt werden.
385                 PreloadLaneGeometries(tileIDs, item.LaneGeometryData.IsInQuadrant(coord.Latitude, coord.Longitude));
386             }
387         }
388
389         if (BuildingProvider_ != null)...
417
418         if (TerrainProvider_ != null)...
446         #endregion
447
448         // Fahrzeugmodell platzieren
449         SetCarToGpsPosition(coord);
450     }
```

Abbildung 6.11: Code der Update-Methode der Klasse Environment

In den **Zeilen 360 bis Zeile 387** wird geprüft, ob eine Komponente **RoadsMapProvider** konfiguriert wurde. Anhand einiger Bedingungen wird dann geprüft, ob neue Daten zu laden sind oder Modelle erzeugt werden müssen. Die in der Abbildung ausgeblendeten Code-Abschnitte in den **Zeilen 389 und 418**, sind beinahe identisch mit dem Code in den **Zeilen 360 bis Zeile 387**, nur das die spezifischen Methoden der Komponenten **BuildingsMapProvider** bzw. **TerrainMapProvider** aufgerufen werden. Auf vergleichbare Weise können die Aufrufe für neue Komponenten hinzugefügt werden. Falls die

Daten und Objekte der neuen Komponente auch in das Caching mit aufgenommen werden sollen, sind noch zusätzliche Änderungen in der Klasse **CacheItem** vorzunehmen (z.B. Properties für Daten und Modelle, Methoden zum abrufen der Daten und das Erzeugen der Modelle). Benötigt die neue Komponente Zugriff auf die von anderen Komponenten erzeugten Modelle oder Daten, so können diese aus den **CacheItem-Instanzen** entnommen und in der Start-/Update-Methode an die neue Komponente übergeben werden.

Die Start-Methode der Klasse **Environment** arbeitet nach dem gleichen Prinzip, so dass das hinzufügen neuer Komponenten auf die gleiche Weise wie in der Update-Methode geschieht.

7 Ergebnis

7.1 Funktionale Anforderungen

Übersicht: Erfüllungsgrad der funktionalen Anforderungen		
Ziffer	Stand	Kommentar
F1.1	unvollständig	Erfüllt für die Fahrbahnen, nicht erfüllt für Gebäude und Terrain.
F1.2	erfüllt	-
F1.3	unvollständig	Erfüllt für die Fahrbahnen, nicht erfüllt für Gebäude und Terrain.
F1.4	unvollständig	Erfüllt für die Fahrbahnen, nicht erfüllt für Gebäude und Terrain.
F2.1	erfüllt	-
F2.2	erfüllt	-
F3.1	erfüllt	-
F3.2	erfüllt	-
F3.3	erfüllt	-

Tabelle 7.1: Erfüllungsgrad der funktionalen Anforderungen

Die funktionalen Anforderungen F1.1, F1.3 und F1.4 enthalten die Schritte zum abrufen, erzeugen und anzeigen der Gebäude und des Terrains. Diese Schritte sind aktuell nicht implementiert, wodurch die dazugehörigen funktionalen Anforderungen als unvollständig anzusehen sind (siehe Tabelle 7.1). Das Gesamtsystem des Prototyps ist hingegen lauffähig und kann die Fahrbahnen sowie das Fahrzeug wie gefordert visualisieren. Die Hauptkomponente **Environment** ist auch bereits für die Nutzung der Komponenten **BuildingsMapProvider** und **TerrainMapProvider** vorbereitet.

7.2 Nichtfunktionale Anforderungen

Übersicht: Erfüllungsgrad der nichtfunktionalen Anforderungen		
Ziffer	Stand	Kommentar
NF1.1	erfüllt	Die Transfer-Klassen (1:1 Kopien der ProtocolBuffer-Klassen für Here HD Live Maps) wurden nicht kommentiert, hierzu wird auf die Original-Klassen verwiesen.
NF1.2	erfüllt	-
NF1.3	erfüllt	-
NF2.1	erfüllt	-
NF2.2	erfüllt	-
NF2.3	erfüllt	-

NF3	nicht erfüllt	Bei einem Übertritt in einen neuen Kartenabschnitt kommt es zu Überschreitungen der Vorgabe von 40ms. Das Problem tritt auf, da bei dem Übertritt in einen neuen Kartenabschnitt, alle 3D-Modelle für den neuen Abschnitt in einem Aktualisierungsschritt erzeugt werden. D.h. die Erzeugung der Modelle muss auf mehrere Aktualisierungsschritte aufgeteilt werden.
NF4	erfüllt	-
NF5	erfüllt	-
NF6	erfüllt	-
NF7	erfüllt	-

Tabelle 7.2: Erfüllungsgrad der nichtfunktionalen Anforderungen

Die nichtfunktionale Anforderung NF3 wurde nicht erfüllt (siehe Tabelle 7.2). In der aktuellen Implementierung werden bei einem Übertritt in einen neuen Kartenabschnitt, alle zu dem Kartenabschnitt gehörenden 3D-Modelle erzeugt und platziert. Dies kann bis zu 500ms (bei etwa 2000 Fahrbahnabschnitten) benötigen. Die Vorgabe kann aber eingehalten werden, indem die Modelle vorausschauend erzeugt und platziert werden. Dazu kann am Ende jedes Aktualisierungsvorgangs geprüft werden, ob noch Zeit zur Verfügung steht. Steht noch Zeit zur Verfügung (Aktualisierungsvorgang kürzer als 40ms), dann kann die verbleibende Zeit genutzt werden, um einen Teil der bereits geladenen Modelle zu erzeugen.

8 Fazit

Der zu betreibende Aufwand für die Erstellung eines minimalen Grundsystems ist deutlicher aufwendiger, als zu Beginn erwartet. Dies beginnt bereits mit der Einarbeitung in die verschiedenen Themen:

- Grundlagen des autonomen Fahrens
- Geodäsie/Kartenkunde (Referenzsysteme und Projektionsverfahren)
- Nutzung und Abruf von Kartenmaterial (Dokumentationen der Anbieter)
- Themen aus der Computergrafik
- Verwendung und Grenzen der 3D-Engine

Die Thematik des autonomen Fahrens bildet die Basis des gesamten Projekts. Auch wenn kein Wissen über die komplexeren inneren Abläufe eines autonomen Fahrzeugs benötigt werden, so müssen dennoch die allgemeinen Prinzipien verstanden werden. Ansonsten ist es nicht möglich ein System zu entwickeln, dass dem Fahrer darstellt wie das Fahrzeug die Umgebung wahrnimmt und warum es bestimmte Aktionen ausführt.
Die Geodäsie und Kartenkunde stellen ebenfalls einen wichtigen Baustein dar. Distanzberechnungen müssen immer auf der Oberfläche des Referenzellipsoiden durchgeführt werden, da es sonst zu Ungenauigkeiten kommen kann. Die Nutzung eines Projektionsverfahrens hat sich als unnötig bzw. sogar falsch herausgestellt, auch wenn ein Projektionsverfahren in der Implementierung verwendet wird. Der Grund hierfür ist, dass jedes Projektionsverfahren (wie die Bezeichnung "Projektion" bereits andeutet) eine Verzerrung des realen Bildes zur Folge hat. Das heißt bei der Projektion einer gekrümmten Oberfläche einer Geometrie auf eine Ebene, ist es nicht möglich alle Eigenschaften (Längentreue, Winkeltreue und Flächentreue) zu erhalten [30]. Anstelle des projizierten Kartenausschnitts zu verwenden, macht es mehr Sinn einen Ausschnitt aus dem Referenzellipsoiden zu nehmen. Dieser Umstand wurde während der Ausarbeitung erst zu spät realisiert und daher nicht mehr geändert.
Der Abruf und die Nutzung des HD-Kartenmaterials verlangt eine intensive Einarbeitung in das Thema. Denn die Anbieter der HD-Karten sind bemüht, die zu übertragende Datenmenge möglichst gering zu halten. Dies beschleunigt den Abruf der Daten zwar enorm, hat aber eine deutlich komplexere Datenstruktur zur Folge. Unter anderem wird auf Fließkommazahlen bei der Übertragung gänzlich verzichtet und der Längen- und Breitengrad sogar in einem einzelnen Ganzzahlwert kombiniert verschlüsselt, somit müssen die Werte gegebenenfalls in ihr Fließkommaformat zurückgerechnet oder entschlüsselt werden. Hat man die Dokumentationen des Hersteller nicht sorgfältig studiert, kann dies schnell zu unerwarteten Ergebnissen führen.
Die Grenzen der 3D-Engine (hier: Unity 3D) bzw. die Grenzen der für C# verwendeten Mono-Implementierung führen häufig zu Problemen, wenn man es gewohnt ist mit .NET-Versionen ab 3.5 oder höher zu arbeiten. Eine selbstentwickelte, in Unity eingebundene Bibliothek mit einer Version höher als .NET 3.5, führt in Unity erst während der Ausführung zu einem Fehler.

Der Prototyp kann in seinem derzeitigen Entwicklungsstadium zwar nicht allen geforderten Bedingungen genügen, bildet aber für weitere Projekte eine solide Grundlage. Das erstellte System ist lauffähig und kann die Fahrbahnen mit Seiten- und Mittelstreifen, eines beliebigen Kartenabschnitts der Here HD Live Map visualisieren. Ebenso können Testfahrten simuliert und echte Koordinaten über

ein angeschlossenes GPS eingelesen und verarbeitet werden.
Das System kann leicht um gänzlich neue Komponenten erweitert werden, indem diese zentral über die Hauptkomponente **Environment** hinzugefügt werden. Hierzu muss lediglich eine neue Schnittstelle definiert und die Start- sowie Update-Routine, um die entsprechenden Aufrufe erweitert werden. Das restliche System benötigt keinerlei Anpassungen. Weiterhin lassen sich alle bereits existierenden Komponenten bei Bedarf ohne Probleme ersetzen.

Das System bietet großes Potential und kann, sobald es fertiggestellt ist, vielseitig erweitert werden. Hier einige Beispiele:

- Erweiterung der Visualisierung um eine Fahrzeuginnenansicht, welche es ermöglicht die Umgebungsdaten auf die Scheiben des Fahrzeugs zu projizieren. Der Fahrer wäre dann nicht mehr gefordert, seinen Blick zwischen Display und Scheiben hin und her wandern zu lassen.
- Anzeige von in der Umgebung befindlichen Sehenswürdigkeiten, Restaurants, Tankstellen und ähnlichem (POI - Points of Interest).
- Erweiterung der Visualisierung zu einem Navigationssystem.
- Anzeige der Gefährdungsstufe, so wie diese für ein Objekt von dem autonomen Fahrzeug eingestuft wird. Dies kann zu einer höheren Akzeptanz der Fahrer automatisierter bzw. autonomer Fahrzeuge führen, da diese so sehen können, wie das steuernde System die Lage einstuft.
- Ausbau zu einem Fahrsimulationssystem.

A Weiterführende Informationen zu Here HD Live Maps

Anmerkung: *Dieser Abschnitt soll nur einen ersten groben Überblick über Here HD Live Maps liefern, mit einem Hauptfokus auf die in diesem Projekt verwendeten Verfahren und benötigten Informationen. Für eine vollständigen und detaillierteren Einblick sind die Dokumentationen des Anbieters [26] [27] in jeglicher Hinsicht vorzuziehen.*

A.1 Daten über die REST-API abrufen

Die Url der REST-API hat folgende Struktur:
https://*<Basis Url>*.catalogs.datastore.api.here.com/v0.1/catalog/*<Datentyp>*?app_id=*<APP_ID>*
&app_code=*<APP_CODE><optional: Parameter>*

Bei den grau markierten Teilen der Url handelt es sich um variable Elemente, mit denen eine Anfrage genauer spezifiziert wird. Wie eine spezifische Anfrage zusammengesetzt wird und welche Daten diese zurückliefern, wird in den folgenden Abschnitten erläutert.

Anmerkung: *Die Tokens <APP_ID> und <APP_CODE> erhält man vom Anbieter und werden für die Authentifizierung beim Zugriff auf den DataStore benötigt.*

A.1.1 Kataloge

Die Kataloge werden nach Region getrennt abgerufen. Hierzu existiert für jede Region eine eigene Basis-Url, welche der Dokumentation entnommen werden können.

<Basis-Urls> für die Regionen:

- Westeuropa = here-hdlivemap-weu-2
- Nordamerika = here-hdlivemap-na-2

Beispiel A.1: Anzeige der Kataloge für die Region Westeuropa

Schritt 1: Basis-Url festlegen
<Basis-Url> = here-hdlivemap-weu-2
Schritt 2: Datentyp festlegen
<Datentyp> = configuration
Schritt 3: Url bilden
https://here-hdlivemap-weu-2.catalogs.datastore.api.here.com/v0.1/catalog/configuration?
app_id=*<APP_ID>*&app_code=*<APP_CODE>*

Response (im JSON-Format):


```
layers:
dataCatalogs:
  0:        "hrn:here:datastore:::here-hdmap-ext-weu-1"
  1:        "hrn:here:datastore:::here-hdlm2-hud-weu-2"
  2:        "hrn:here:datastore:::here-hdlm2-liveroadsflow-weu-2"
  3:        "hrn:here:datastore:::here-hdlm2-lrincidents-weu-2"
```


Die Kataloge enthalten die nachstehenden Informationen:

- dataCatalogs[0] = HD-Maps
- dataCatalogs[1] = Humanized Driving (HUD)
- dataCatalogs[2] = Live Roads (aktuelle Verkehrsinformationen)
- dataCatalogs[3] = Live Incidents (aktuelle Unfallmeldungen)

Alle Kataloge werden versioniert gespeichert, somit ist es möglich bei Problemen mit einer Version weiterhin auf eine ältere Version zuzugreifen. Die aktuelle Versionsnummer eines Katalogs kann wie folgt abgerufen werden, diese wird auch später benötigt um die Daten zu einer Partition herunterzuladen.

Beispiel A.2: Anzeige der Version für den Katalog der Region Westeuropa

Schritt 1: Basis-Url festlegen

<Basis-Url> = here-hdlivemap-weu-2

Schritt 2: Datentyp festlegen

<Datentyp> = version

Schritt 3: Url bilden

https://here-hdlivemap-weu-2.catalogs.datastore.api.here.com/v0.1/catalog/version?app_id=*<APP_ID>*&app_code=*<APP_CODE>*

Response (im JSON-Format):

```
versions:
  0:
    version:          1301488
    timestamp:        1495575013975
    timeToLive:       86400
    partitionCounts:
    dependencies:
      0:
        version:      622
        hrn:          "hrn:here:datastore:::here-hdmap-ext-weu-1"
        name:         "here-hdmap-ext-weu-1"
        indirect:     false
      1:
        version:      789678
        hrn:          "hrn:here:datastore:::here-hdlm2-lrincidents-weu-2"
        name:         "here-hdlm2-lrincidents-weu-2"
        indirect:     false
      2:
        version:      53
        hrn:          "hrn:here:datastore:::here-hdlm2-hud-weu-2"
        name:         "here-hdlm2-hud-weu-2"
        indirect:     false
      3:
        version:      1110374
        hrn:          "hrn:here:datastore:::here-hdlm2-liveroadsflow-weu-2"
        name:         "here-hdlm2-liveroadsflow-weu-2"
        indirect:     false
```


Unter dem Attribut "dependencies" befinden sich die vier Kataloge aus dem Beispiel A.1. Jeder der Kataloge besitzt als erstes Attribut die jeweils aktuelle Katalog-Version.

A.1.2 Layer

Alle Kataloge verwalten andere Daten, folglich unterscheiden sich auch die verfügbaren Layer der Kataloge. Ein Layer, der Layer "schema", ist jedoch in jedem Katalog enthalten. Dieser enthält einen Verweis auf die Google ProtocolBuffer Schema-Dateien.

Wie man sich die Layer eines Katalogs anzeigen lassen kann, wird anhand des Beispiels im Anschluss erläutert.

Beispiel A.3: Anzeige der Layer des Katalogs HD-Maps

Schritt 1: Basis-Url festlegen

Als Basis-Url dient der Name des Katalogs, dieser kann

- aus der Katalog-Abfrage im Beispiel A.1 entnommen werden, indem man den Namen aus dem hinteren Teil (alles nach den :::) des Katalog-Strings extrahiert.
  Bsp.: dataCatalogs[0] = "hrn:here:datastore:::here-hdmap-ext-weu-1"
- aus der Versions-Abfrage im Beispiel A.2 entnommen werden. Hier findet man den Namen im jeweiligen name-Attribut der dependency.
  Bsp.: versions[0].dependencies[0].name = "here-hdmap-ext-weu-1"

<Basis-Url> = here-hdmap-ext-weu-1

Schritt 2: Datentyp festlegen

<Datentyp> = configuration

Schritt 3: Url bilden

https://here-hdmap-ext-weu-1.catalogs.datastore.api.here.com/v0.1/catalog/configuration?app_id=*<APP_ID>*&app_code=*<APP_CODE>*

Response (im JSON-Format):

Aus Platzgründen werden die Daten hier allerdings nicht vollständig dargestellt. Unter anderem ist das Attribut "layerProperties" überall ausgeblendet, da in dieser Übersicht nicht weiter darauf eingegangen wird. Diese enthalten z.B. Angaben zum Content-Type, Content-Encoding, Komprimierung und weiteres.


```
▼ layers:
  ▼ 0:
      name:              "schema"
      partitioning:      "generic"
    ▼ dataUrl:           "https://here-hdmap-continental-2.s3.amazonaws.com/here-hdmap-ext-weu-1/"
    ▶ layerProperties:
  ▼ 1:
      name:              "speed-attributes"
      partitioning:      "heretile"
    ▼ dataUrl:           "https://here-hdmap-continental-2.s3.amazonaws.com/here-hdmap-ext-weu-1/"
    ▶ layerProperties:
  ▼ 2:
      name:              "topology-geometry"
      partitioning:      "heretile"
    ▼ dataUrl:           "https://here-hdmap-continental-2.s3.amazonaws.com/here-hdmap-ext-weu-1/"
    ▶ layerProperties:
  ▼ 3:
      name:              "lane-topology"
      partitioning:      "heretile"
    ▼ dataUrl:           "https://here-hdmap-continental-2.s3.amazonaws.com/here-hdmap-ext-weu-1/"
    ▶ layerProperties:
▼ dataCatalogs:
    0:                   "hrn:here:datastore:::here-hdmap-ext-weu-1"
```

Jeder Layer besitzt ein Attribut "dataUrl". Die in diesem Attribut abgelegte Url wird benötigt, um eine Partition aus dem jeweiligen Layer herunterzuladen.

A.1.3 Partitionen

Um eine Partition abzurufen, werden alle zuvor besprochenen Informationen bzw. Verfahren benötigt und zwar angefangen bei der Berechnung der Here-Tile-ID bis zur Ermittlung der "dataUrl" eines Layers. Der Abruf einer Partition ist dabei in zwei Schritte unterteilt. Im ersten Schritt wird ein "dataHandle" ermittelt und im zweiten Schritt wird das "dataHandle" mit dem "dataUrl" verknüpft. Mittels des so gebildeten Urls kann schlussendlich die gewünschte Partition heruntergeladen werden.

Beispiel A.4: Ermitteln des DataHandles für die Partition 377894440 des Layer "lane-topology"

Schritt 1: Basis-Url festlegen

Die Basis-Url wurde bereits mit der Wahl des Katalogs festgelegt (in Beispiel A.3) und kann von dort übernommen werden.

<Basis-Url> = here-hdmap-ext-weu-1

Schritt 2: Datentyp festlegen

<Datentyp> = partitions

Schritt 3: Parameter festlegen

Um eine Partition zu identifizieren, müssen folgende Parameter festgelegt werden.

- version = 622
  Die Version findet man im Versions-Feld des entsprechenden Katalogs (siehe Beispiel A.2), in diesem Fall des Katalogs "HD-Maps"
- layer = lane-topology
  Prinzipiell kann jeder beliebige Layer eines Katalogs ausgewählt werden, als Beispiel wird hier der Layer "lane-topology" gewählt. Dieser enthält 2D-Daten zu den Fahrbahnen.
- partition = 377894440
  Eine Partition wird durch die Here-Tile-ID identifiziert. Hier wurde die ID aus dem Beispiel 2.2 übernommen.

Die Parameter werden wie bei einem Url üblich per &-verknüpft angehängt.

<Parameter> = &version=622&layer=lane-topology&partition=377894440

Schritt 4: Url bilden

https://here-hdmap-ext-weu-1.catalogs.datastore.api.here.com/v0.1/catalog/partitions?app_id=*<APP_ID>*&app_code=*<APP_CODE>*&version=622&layer=lane-topology&partition=377894440

Response (im JSON-Format):

```
▼ partitions:
  ▼ 0:
      version:      621
      partition:    "377894440"
      layer:        "lane-topology"
      dataHandle:   "58713f087302-910b-1af4-8244-c75058b5"
```

Anmerkung: *Benötigt man das "dataHandle" einer Partition nicht nur für einen Layer, sondern für mehrere, dann kann man die weiteren Layer per &-Verknüpfung angeben. Dies würde dann eine Liste mit mehreren Objekten und deren "dataHandles" zurückliefern. Auf die gleiche Weise kann man auch mehrere Partitionen angeben, um so "dataHandles" für mehrere Partitionen gleichzeitig abzurufen. Mit der aktuellen Anfrage erhält man genau ein Objekt mit dem zugehörigen "dataHandle".*

Beispiel A.5: Partition abrufen

Schritt 1: <dataUrl> bestimmen

Die <dataUrl> für den Layer "lane-topology" lautet (siehe Beispiel A.3):
<dataUrl> = https://here-hdmap-continental-2.s3.amazonaws.com/here-hdmap-ext-weu-1/

Schritt 2: <dataHandle> bestimmen:

Der <dataHandle> wurde im vorhergehenden Beispiel A.4 ermittelt und lautet:
<dataHandle> = 58713f087302-910b-1af4-8244-c75058b5

Schritt 3: Url zusammensetzen

<dataUrl><dataHandle> = https://here-hdmap-continental-2.s3.amazonaws.com/here-hdmap-ext-weu-1/58713f087302-910b-1af4-8244-c75058b5

Response (im Google ProtocolBuffer-Format):

Die zurückgelieferten Daten enthalten nun alle Informationen zur gesuchten Partition im gewählten Layer. Diese Daten können allerdings noch nicht direkt ausgewertet werden, sondern liegen im Google ProtocolBuffer-Format vor und müssen noch in das entsprechende Schema umgewandelt werden. Die hierzu benötigten ProtocolBuffer Schema-Dateien können ebenfalls über die REST-API abgerufen werden. Das Google ProtocolBuffer-Format wird in den folgenden Abschnitten etwas näher erläutert.

A.2 Google ProtocolBuffer

Bei Google ProtocolBuffer handelt es sich um ein sprach- und plattformneutrales Format für den Austausch von strukturierten Daten, wie auch z.B. XML. Im Gegensatz zu XML ist das Google ProtocolBuffer-Format nicht für den Menschen lesbar gestaltet, sondern auf Effizienz ausgelegt. D.h. die Datenpakete sind kleiner und können somit schneller übertragen werden.

Da es sich bei den Google ProtocolBuffer um ein strukturiertes Format handelt, muss dieses Format zunächst in irgendeiner Weise beschrieben werden, hierfür werden die bereits erwähnten Schema-Dateien verwendet. Die Schema-Dateien enthalten die Baupläne zur Beschreibung der programmiersprachlichen Objekte und können mittels eines von Google bereitgestellten Compilers in das jeweilige Sprachformat (z.B. C++, C#, Java, Python) kompiliert werden.

Anmerkung: *Auf eine ausgiebige Einführung in das Google ProtocolBuffer-Format wird hier verzichtet und stattdessen auf die offizielle Dokumentation verwiesen:* `https://developers.google.com/protocol-buffers/docs/proto3`

A.2.1 Schema-Dateien herunterladen

Das herunterladen der Schema-Dateien erfolgt analog dem herunterladen der Partitionsdaten. Wie im Beispiel A.3 zu sehen ist, existiert in jedem Katalog ein Layer "schema". Der Layer "schema" wird ebenfalls in Partitionen unterteilt, wobei die Partitionen des Layer "schema" die Schema-Dateien für die verschiedenen Layer des Katalogs speichert. Um die Schema-Dateien herunterzuladen, benötigt man somit wieder die "dataUrl" und ein "dataHandle".

Beispiel A.6: Ermitteln des DataHandles für die Schema-Dateien

Schritt 1: Basis-Url festlegen

Die Basis-Url wurde bereits mit der Wahl des Katalogs im Beispiel A.3 festgelegt und kann von dort übernommen werden.

<Basis-Url> = here-hdmap-ext-weu-1

Schritt 2: Datentyp festlegen

<Datentyp> = partitions

Schritt 3: Parameter festlegen

layer = schema

Schritt 4: Url bilden

https://here-hdmap-ext-weu-1.catalogs.datastore.api.here.com/v0.1/catalog/partitions?app_id=*<APP_ID>*&app_code=*<APP_CODE>*&layer=schema

Response (JSON-Format):

```
partitions:
  0:
    version:      448
    partition:    "hdmap-lane-topology-bundle"
    layer:        "schema"
    dataHandle:   "3dd390dd52ea-c8ab-6304-4a37-5d661c85"
  1:
    version:      448
    partition:    "hdmap-speed-attributes-bundle"
    layer:        "schema"
    dataHandle:   "9eb9b0e1f2b4-1bd8-cef4-066f-0cd23ebd"
  2:
    version:      448
    partition:    "hdmap-topology-geometry-bundle"
    layer:        "schema"
    dataHandle:   "343b26cb0ed3-47ca-6384-d591-38af1e71"
```


Über das Attribut "partition" können die Schema-Dateien einem Layer zugeordnet werden, wobei der in "partition" angegebene Layer-Name mit einem Pre- und Postfix versehen ist:

- Prefix = hdmap-
- Postfix = -bundle

Beispiel A.7: Schema-Dateien abrufen

Schritt 1: *<dataUrl>* bestimmen

Die *<dataUrl>* für den Layer "lane-topology" lautet (siehe Beispiel A.3):
<dataUrl> = https://here-hdmap-continental-2.s3.amazonaws.com/here-hdmap-ext-weu-1/

Schritt 2: *<dataHandle>* bestimmen

Der *<dataHandle>* wurde im vorhergehenden Beispiel A.6 ermittelt und lautet für den Layer "lane-topology":
<dataHandle> = 3dd390dd52ea-c8ab-6304-4a37-5d661c85

Schritt 3: Url zusammensetzen

<dataUrl><dataHandle> = https://here-hdmap-continental-2.s3.amazonaws.com/here-hdmap-ext-weu-1/3dd390dd52ea-c8ab-6304-4a37-5d661c85

Response (ZIP-Datei):

Als Rückgabe erhält man eine ZIP-Datei mit den ProtocolBuffer Schema-Dateien (Dateityp: *.proto), welche nun noch in das gewünschte Format kompiliert werden müssen.

A.2.2 Schema-Dateien kompilieren

Den benötigten Compiler findet man zum Download unter https://developers.google.com/protocol-buffers/docs/downloads.

Das kompilieren selbst erfolgt über die Kommandozeile.

Beispiel A.8: Schema-Dateien kompilieren

Das kompilieren kann mit folgendem Befehl angestoßen werden:
protoc --csharp_out=*<Zielordner>* *<Quellordner>*<*Schema-Datei*>

Mit dem Attribut "--csharp_out" wird das Ausgabeformat festgelegt, in diesem Fall C#-Klassendateien. Der Parameter *<Zielordner>* gibt den Zielordner für die kompilierte Datei an und zwar relativ vom aktuellen Ordner. Die Parameter *<Quellordner>* und *<Schema-Datei>* identifizieren zunächst den Ordner in welchem sich die .proto-Datei befindet, gefolgt vom Namen der .proto-Datei (inklusive Endung .proto). Der Ordnerpfad ist ebenfalls relativ vom aktuellen Ordner aus anzugeben.

Beispiel: protoc --csharp_out=compile com\here\pb\hdmap\external\v1\geometry\geometry.proto

```
D:\HereHD>dir
 Datenträger in Laufwerk D: ist DATA
 Volumeseriennummer: 0CF7-E22C

 Verzeichnis von D:\HereHD

29.05.2017  11:30    <DIR>          .
29.05.2017  11:30    <DIR>          ..
29.05.2017  11:29    <DIR>          com
29.05.2017  11:29    <DIR>          compiled
27.01.2017  19:20         4.157.440 protoc.exe
               1 Datei(en),      4.157.440 Bytes
               4 Verzeichnis(se), 283.990.491.136 Bytes frei

D:\HereHD>protoc --csharp_out=compiled com\here\pb\hdmap\external\v1\geometry\geometry.proto

D:\HereHD>_
```

Für detailliertere Informationen zum kompilieren der Schema-Dateien sei hier wieder auf die entsprechende Dokumentation verwiesen https://developers.google.com/protocol-buffers/.

A.3 Die Datenstruktur des Lane-Topology-Layer

In diesem Abschnitt werden detailliertere Informationen zur Datenstruktur des Layer "lane-topology" aus dem Katalog "here-hdmap-ext-weu-1" gegeben. Der Layer "lane-topology" verwaltet die Informationen zur Beschreibung der Topologie der Fahrbahnen, dies beinhaltet:

- den Verlauf der äußeren Begrenzungen der Straßen
- Anzahl der Spuren der Straße
- Verlauf der linken und rechten Begrenzung zu jeder Spur

Bevor die eigentliche Datenstruktur des Layers besprochen wird, werden zunächst ein paar allgemeine Verfahren erörtert, welche von allgemeiner Relevanz für alle Datenstrukturen sind.

Anmerkung: *Für eine vollständige Übersicht sei erneut auf die Dokumentation [27] des Anbieters verwiesen.*

A.3.1 Ver-/Entschlüsselung von Längen- und Breitengrad

Um Koordinaten ressourcenschonend speichern und übertragen zu können, werden die Längen- und Breitengrade in einem einzelnen long-Wert (bzw. 64Bit-Integer) verschlüsselt. Als Verschlüsselungsmethode wird ein Morton-Code verwendet, dieser basiert auf ganzzahligen Werten. D.h. zunächst müssen bei der Verschlüsselung der Breiten- und Längengrad in Integer-Werte umgewandelt werden. Der Längengrad wird dabei mit 32Bit dargestellt und der Breitengrad mit 31Bit. Der Breitengrad wird mit 1Bit geringerer Genauigkeit gespeichert, da dieser im Bereich -90° bis 90° liegt und somit nur einen halb so großen Bereich wie der Längengrad besitzt, welcher den Bereich -180° bis 180° abdeckt. Da es allerdings keinen 31Bit-Integer gibt, wird das höchstwertige Bit beim Breitengrad immer auf 0 gesetzt. Die Genauigkeit liegt somit bei $360°/2^{32} = 180°/2^{31} = 0.00000008381903171539306640625°$ dies entspricht in etwa einem 1cm auf dem Äquator/Meridian des WGS84-Ellipsoid.

Beispiel A.9: Koordinaten verschlüsseln

Schritt 1: Längen- und Breitengrad als 32-Bit Integer darstellen

Als Beispiel dienen wieder die Werte Längengrad = 13.36937° und Breitengrad = 52.52507° aus dem Beispiel 2.1.

- Formel: $Längengrad_{Integer} = \lfloor Längengrad/(360°/2^{32}) \rfloor$
  $Längengrad_{Integer} = \lfloor 13.36937°/(360°/2^{32}) \rfloor = \lfloor 159502796.995 \rfloor = 159502796$
- Formel: $Breitengrad_{Integer} = \lfloor Breitengrad/(180°/2^{31}) \rfloor$
  $Breitengrad_{Integer} = \lfloor 52.52507°/(180°/2^{31}) \rfloor = \lfloor 626648494.084 \rfloor = 626648494$

Schritt 2: Konvertierung in Binärdarstellung

Das höchstwertige Bit des Breitengrad (erstes Bit von links) wird immer auf 0 gesetzt, wenn dieses eine 1 enthält.

- $Längengrad_{Integer}(159502796_{10}) = 0000\ 1001\ 1000\ 0001\ 1101\ 0001\ 1100\ 1100_2$
- $Breitengrad_{Integer}(626648494_{10}) = 0010\ 0101\ 0101\ 1001\ 1110\ 0101\ 1010\ 1110_2$

Schritt 3: Binär verschränken

Die im vorherigen Schritt berechneten Binärwerte müssen nun ineinander verschränkt werden, beginnend mit dem höchstwertigen Bit des Breitengrad (B), gefolgt von dem höchstwertigen Bit des Längengrad (L).

- $B_{31}L_{31}B_{30}L_{30}B_{29}L_{29}...B_1L_1B_0L_0$
- $00001000\ 01100011\ 01100010\ 10000011\ 11111001\ 00100011\ 11011000\ 11111000_2$

Schritt 4: Konvertierung ins Dezimalsystem

Den Dezimalwert, welchen man nach der Konvertierung erhält, wird als 64Bit-Integer interpretiert und repräsentiert den MortonCode.

- $Koordinate_{Encoded} = 604435093957892344_{10}$

Beispiel A.10: Koordinaten entschlüsseln

Das Dekodieren erfolgt durch das ausführen der inversen Funktionen zu den einzelnen Schritten in umgekehrter Reihenfolge.

Schritt 1: Konvertieren in Binärdarstellung

Die 64Bit-Kodierte Koordinate aus der Dezimaldarstellung in die Binärdarstellung konvertieren.

- $Koordinate_{Encoded} = 604435093957892344_{10}$
- $00001000\,01100011\,01100010\,10000011\,11111001\,00100011\,11011000\,11111000_2$

Schritt 2: Binäre Verschränkung rückgängig machen

Den 32Bit Längen- und Breitengrad aus der 64Bit-Kodierten Koordinate extrahieren.

- Den $Längengrad_{Integer}$ erhält man, indem man alle Binärstellen mit $Index\%2 == 0$ extrahiert und die Reihenfolge der Binärstellen beibehält.
- $Längengrad_{Integer} = 0000\,1001\,1000\,0001\,1101\,0001\,1100\,1100_2$
- Den $Breitengrad_{Integer}$ erhält man, indem man alle Binärstellen mit $Index\%2 == 1$ extrahiert und die Reihenfolge der Binärstellen beibehält.
- $Breitengrad_{Integer} = 0010\,0101\,0101\,1001\,1110\,0101\,1010\,1110_2$

Schritt 3: Konvertieren in Dezimaldarstellung

Die aktuell in der Binärdarstellung vorliegenden Werte für den $Längengrad_{Integer}$ und den $Breitengrad_{Integer}$ in die Dezimaldarstellung konvertieren.

- $Längengrad_{Integer}(0000\,1001\,1000\,0001\,1101\,0001\,1100\,1100_2) = 159502796_{10}$
- $Breitengrad_{Integer}(0010\,0101\,0101\,1001\,1110\,0101\,1010\,1110_2) = 626648494_{10}$

Schritt 4: Längen- und Breitengrad in die üblich Grad-Darstellung zurückrechnen

Die benötigten Formeln können durch einfache Umformung der Formeln aus Schritt 1 des Beispiel A.9 erlangt werden, wobei das Abrunden nicht rückgängig gemacht werden kann. Durch das Abrunden beim verschlüsseln entsteht eine geringe Abweichung vom Originalwert, diese ist aber kleiner als 1cm und somit vernachlässigbar.

- Ausgangsformel: $Längengrad_{Integer} = Längengrad/(360°/2^{32}) \qquad |*(360°/2^{32})$
  $Längengrad = Längengrad_{Integer} * (360°/2^{32})$
  $Längengrad = 159502796_{10} * (360°/2^{32}) = 13.36936992°$
- Die Umformung der Formel für den Breitengrad verläuft analog.
  $Breitengrad = 626648494_{10} * (180°/2^{31}) = 52.525069993°$

Schritt 5: Prüfen ob Breitengrad > 90° Das höchstwertige Bit des Breitengrad wird bei der Verschlüsselung auf 0 gesetzt. Da das höchstwertige Bit bei einem Integer aber als Vorzeichen-Bit verwendet wird, sind nun alle Breitengrade positiv. Daher muss noch geprüft werden, ob der berechnete Breitengrad größer als 90° ist. Ist dies der Fall, muss nun noch 180° vom berechneten Breitengrad abgezogen werden.

A.3.2 Offset-Kodierung von Koordinaten

Jede Datenstruktur zur Beschreibung einer Partition besitzt ein Attribut, welches die Koordinate des Mittelpunkts der Partition angibt. Alle weiteren Koordinaten der Datenstruktur werden durch eine Offset-Kodierung in Abhängigkeit zum Mittelpunkt angegeben. Für Arrays mit Koordinaten gilt:

- Die Koordinate bei Index 0 wird in Abhängigkeit zum Partitionsmittelpunkt angegeben

- Alle weiteren Koordinaten werden in Abhängigkeit zu ihrem Vorgänger angegeben

Die Offset-Kodierung erfolgt durch eine bitweise XOR-Verknüpfung der kodierten Koordinaten. Durch dieses Verfahren sollen möglichst viele 0en produziert werden, welche eine bessere Komprimierung bei der Übertragung ermöglichen.

Beispiel A.11: Dekodierung einer Koordinate aus einem Array

Dekodierung einer Koordinate bei Index 3 in einem Array.

Schritt 1: XOR-Verknüpfung

Um die kodierte Koordinate bei Index 3 zu erhalten, muss eine bitweise XOR-Verknüpfung des Partitionsmittelpunkts und der Koordinaten aus dem Array (von Index 0 bis Index 3) durchgeführt werden.

Mittelpunkt	:	0110 1100
XOR Array[0]	:	0000 0001
XOR Array[1]	:	0000 1001
XOR Array[2]	:	0010 0001
XOR Array[3]	:	0100 1000
Ergebnis	:	0000 1101

Anmerkung: *Die Binärwerte dienen nur zur Verdeutlichung und entstammen keinem realen Beipsiel.*

Schritt 2: Koordinate entschlüsseln

Die Entschlüsselung der Koordinate wird nun mit dem im Beispiel A.10 beschriebenen Verfahren durchgeführt.

Literaturverzeichnis

[1] Marcus Erbrich: *Experte über Autonomes Fahren: Durchbruch nicht vor 2030*. Automobilwoche, 2017. http://www.automobilwoche.de/article/20170204/NACHRICHTEN/170209931/experte-ueber-autonomes-fahren-durchbruch-nicht-vor-, besucht: 29.08.2017.

[2] Verband der Automobilindustrie e. V.: *Automatisierung: Von Fahrerassistenzsystemen zum automatisierten Fahren*. VDA Magazin, (03. September), 2015. https://www.vda.de/de/services/Publikationen/automatisierung.html.

[3] Schaal, S.: *Audi A8: Audis skurriles Ablenkungsmanöver*. Wirtschafts Woche, 2017. http://www.wiwo.de/unternehmen/auto/audi-a8-beim-autonomen-fahren-kann-der-a8-mehr-als-erlaubt/20045816-2.html, besucht: 29.08.2017.

[4] Fasse, S.: *Recht bremst autonomes Fahren in Europa aus*. VDI-Nachrichten, 2014. http://www.vdi-nachrichten.com/Technik-Wirtschaft/Recht-bremst-autonomes-Fahren-in-Europa, besucht: 29.08.2017.

[5] *Übereinkommen über den Straßenverkehr*. Bundeskanzleramt der Republik Österreich, 2017. https://www.ris.bka.gv.at/GeltendeFassung.wxe?Abfrage=Bundesnormen&Gesetzesnummer=10011542, besucht: 29.08.2017.

[6] *Entwurf eines Gesetzes zur Änderung der Artikel 8 und 39 des Übereinkommens vom 8. November 1968 über den Straßenverkehr*. Bundesrat, 2016. https://www.bundesrat.de/SharedDocs/beratungsvorgaenge/2016/0201-0300/0243-16.html, besucht: 29.08.2017.

[7] *Änderung des Wiener Übereinkommens vom 8. November 1968 über den Straßenverkehr*. Deutscher Bundestag, 2016. https://www.bundestag.de/blob/478076/841f44a098f22c40b9ac44adf6f973cb/wd-2-131-16-pdf-data.pdf, besucht: 29.08.2017.

[8] *Gesetz zur Änderung der Artikel 8 und 39 des Übereinkommens vom 8. November 1968 über den Straßenverkehr*. Bundesrat, 2016. https://www.bundesrat.de/SharedDocs/beratungsvorgaenge/2016/0501-0600/0584-16.html, besucht: 29.08.2017.

[9] *Bundesrat gibt grünes Licht für automatisierte Autos*. beck-aktuell, 2017. https://rsw.beck.de/aktuell/meldung/bundesrat-gibt-gruenes-licht-fuer-automatisierte-autos, besucht: 09.08.2017.

[10] Lenz, B.: *Autonomes Fahren: Technische, rechtliche und gesellschaftliche Aspekte*. Springer, s.l., 2015, ISBN 978-3-662-45854-9.

[11] Eisele, K. H.: *Die Vermessung der automobilen Welt.*, 2014. https://www.mercedes-benz.com/de/mercedes-benz/next/automation/die-vermessung-der-automobilen-welt/, besucht: 16.08.2017.

[12] *Differential Global Positioning System*, 06.08.2017. https://de.wikipedia.org/w/index.php?oldid=166890164, besucht: 29.08.2017.

[13] *Landmarke*, 09.08.2017. https://www.colliseum.net/wiki/Landmarke, besucht: 29.08.2017.

[14] *Landmarke*, 06.08.2017. https://de.wikipedia.org/w/index.php?oldid=166545718, besucht: 29.08.2017.

[15] Stanley, B.: *Localizing Ground-Penetrating Radar: Innovative ground-penetrating radar that maps underground geological features provides autonomous vehicles with real-time localization in all-weather conditions.* TechNote, 2016. https://ll.mit.edu/publications/technotes/TechNote_LGPR.pdf, besucht: 29.08.2017.

[16] Sokolov, D. A. J. und h. online: *Bodenradar für selbstfahrende Autos horcht unter die Straße*. heise online, 2016. https://www.heise.de/newsticker/meldung/Bodenradar-fuer-selbstfahrende-Autos-horcht-unter-die-Strasse-3273941.html, besucht: 09.08.2017.

[17] Haupt, H.: *Autonomes Fahren: Fahrerlos, aber mit guter Karte*. ZEIT ONLINE, 2015. http://www.zeit.de/mobilitaet/2015-07/navigation-autonomes-fahren, besucht: 29.08.2017.

[18] Schaal, S.: *Nokia Here: Warum Autobauer Milliarden für Karten ausgeben*. Wirtschafts Woche, 2015. http://www.wiwo.de/unternehmen/auto/nokia-here-warum-autobauer-milliarden-fuer-karten-ausgeben/12137572.html, besucht: 29.08.2017.

[19] *Orientierung bei Wind und Wetter: Radarkarten für das autonome Fahren*. Focus Online, 2017. http://www.focus.de/auto/news/radarkarten-fuer-das-autonome-fahren-orientierung-bei-wind-und-wetter_id_7224211.html, besucht: 16.08.2017.

[20] Rügheimer, H.: *Forschung fürs autonome Fahren: Software macht Sensoren besser*. Intelligente Welt, 2016. http://intelligente-welt.de/forschung-fuers-autonome-fahren-sensoren/, besucht: 09.08.2017.

[21] King, V.: *The Latest in LiDAR*. Point Of Beginning, 2017. http://www.pobonline.com/articles/100757-the-latest-in-lidar, besucht: 09.08.2017.

[22] El-Ashmawy, K. L.: *Investigation of the Accuracy of Google Earth Elevation Data*. Artificial Satellites, 51(3):53, 2016, ISSN 2083-6104.

[23] Bollmann, J. (Hrsg.): *Lexikon der Kartographie und Geomatik: In zwei Bänden*. Spektrum, Akad. Verl., Heidelberg [u.a.], 2005, ISBN 3827416728.

[24] Hoberg, F.: *Warum die Welt dringend neu vermessen werden muss*. PS Welt, 2016. https://www.welt.de/motor/article157555441/Warum-die-Welt-dringend-neu-vermessen-werden-muss.html, besucht: 29.08.2017.

[25] Rabel, D. I. D.: *Digitale Karten als Wegweiser für das autonome Fahren: Präzise, dynamisch und selbstheilend*. ATZextra, 2016(3), 2017.

[26] *DataStore API DataStore API: Developer's Guide*. HERE Global B.V., 2.0 Aufl., 2016.

[27] *HD Live Map: Developer Guide*. HERE Global B.V., 3.0 Aufl., 2017.

[28] Rabel, D. I. D.: *How HERE HD Live Map paved the way for autonomous cars in 2016 - HERE 360*. 13.12.2016. http://360.here.com/2016/12/13/how-here-hd-live-map-paved-the-way-for-autonomous-cars-in-2016/, besucht: 29.08.2017.

[29] Rubel, B.: *HERE: Audi, BMW und Mercedes-Benz vernetzen ihre Autos in Echtzeit*. Mobile Geeks, 2016. https://www.mobilegeeks.de/artikel/here-maps/, besucht: 29.08.2017.

[30] Lange, N.: *Geoinformatik: In Theorie und Praxis*. Springer, Berlin and Heidelberg, 3., vollst. überarb. u. akt. aufl. 2013 Aufl., 2013, ISBN 978-3-642-34806-8.