Inhaltsverzeichnis

Abbildungsverzeichnis

Formelverzeichnis

1. Einleitung

Die Datenübertragung von der Erde zum Mars kann bis zu 26 Minuten dauern (vgl. Carsten u. a. 2009, S.2), welches ein Problem für die Erforschung mit den Rovern ist. Bei einem Fernsteuerungsversuch müsste man mit diesen großen Verzögerungen rechnen und könnte somit nur mit sehr vorsichtigen und langsamen Manövern den riesigen roten Planeten erforschen. Zur Lösung dieses Problems, können sogenannte autonome Navigationsprogramme helfen, welche bei allen aktuellen Rover Missionen der NASA verwendet werden. Dabei ist vor allem ein fehlerfreier Ablauf wichtig, damit die zeit- und kostenaufwändigen Rover nicht beschädigt werden.

In dieser Arbeit werden im ersten Abschnitt die Gründe, Problemstellungen und Methoden zur Entwicklung autonomer Navigationssoftware erläutert. Der zweite Abschnitt befasst sich mit deren Funktionsweise bei verschiedenen Rovern und dessen Weiterentwicklung über den Missionen hinweg. Als Nächstes wird der Entstehungsprozess meiner Simulation in der dritten Sektion dokumentiert, welche die Funktionsweisen in einer Praxisumgebung darstellt. Und zum Schluss wird abschließend Perspektiven für die Zukunft gegeben.

2. Sinnhaftigkeit der Navigationssoftware

Es ergibt nur Sinn an einer Software zu forschen, wenn dies einen Nutzen mit sich bringt und ein zuvor ungelöstes Problem löst. So ist es auch bei der Navigationssoftware, welche versucht, die Forschung auf dem Mars so effizient wie möglich zu gestalten, was besonders bei den begrenzten Energie Ressourcen der Rover wichtig ist (vgl. Laubach, Sharon, 1999, S.25). Wichtig ist dabei auch ein reibungsloser Ablauf, welcher am ehesten mit Regeln bei der Entwicklung garantiert werden kann. Und um ungewollte Verhaltensweisen zu vermeiden, sollte vor dem Einsatz der Software ausgiebig getestet werden.

2.1 Problemstellung der autonomen Software

Bei Mars Missionen gilt es, möglichst viel in kürzester Zeit zu erforschen. Daran gehindert wird man aber durch technische Gegebenheiten.

Einmal wären da die oben genannten Probleme der Datenübertragung (vgl. Carsten u. a. 2009, S.2). Durch eine große Verzögerung, kann man entweder nur kleinschrittige und vorsichtige Manöver durchführen oder es riskieren, den kostenaufwändigen Rover zu beschädigen, oder gar zu verlieren.

Hierzu kommt noch die Limitation der Kommunikation des Deep Space Networks (vgl. Maimone, Mark, 2017, 07:32-07:49). Bei einem Versuch der Echtzeit Steuerung, müssten die Bilddaten vom Rover zur Erde übertragen werden, damit ein Mensch die nächsten Schritte planen kann. Diese Daten zu den nächsten Schritten müssten dann an den Rover gesendet werden, damit er diese ausführt. Da aber das Deep Space Network auch mit allen anderen Missionen geteilt wird (vgl. Laubach, Sharon, 1999, S.26), gibt es die Beschränkungen der Kommunikationsmöglichkeit von einmal am Tag (vgl. Maimone, Mark, 2017, 06:38-06:50), welche nicht eingehalten werden könnte.

Dies würde aber außer Acht lassen, dass nicht zu jeder Zeit eine Verbindung zum Rover möglich ist. Wenn das Raumfahrzeug keine direkte Sicht auf die Erde hat, wird ein Verbindungsaufbau zum Rover verhindert (vgl. Laubach, Sharon, 1999, S.25). Bei der Fernsteuerung wäre die mobile Forschungseinheit in diesen Zeiträumen nicht in der Lage, Befehle zu erhalten und auszuführen, und damit nicht einsatzfähig sein. Eine langsame & verzögerte Übertragung, gebündelt mit einer begrenzten Kommunikationsmöglichkeit zum Mars, macht es somit nicht ausführbar, die Rover in Echtzeit zu steuern, weswegen eine autonome Navigationssoftware eine große Bedeutung bei der Erforschung des roten Planeten hat.

2.2 Herausforderungen bei der Entwicklung der Software

Die Herausforderungen ändern sich über den Missionen hinweg, doch es gibt auch viele Gemeinsamkeiten auf welche bei allen Operationen geachtet werden muss.

Einmal wäre da das unbekannte Terrain, welches die Rover durchqueren müssen. Die gesamte Oberfläche des Mars ist größtenteils nicht bekannt (vgl. Morrison, Jack/ Nguyen, Tam, 1996, S.1), weswegen kein vorher berechneter und programmierter Weg für die Rover erstellt werden kann. Bei der Operation kann es zu neuen Hindernissen kommen, oder das mobile Oberflächenraumfahrzeug könnte an dem Terrain ausrutschen und die tatsächliche Position verändern (vgl. Laubach, Sharon, 1999, S.25). Somit muss die Software in verschiedenen unvorhersehbaren Situationen einen passenden Weg berechnen und den Rover nach diesen Berechnungen bewegen (vgl. Morrison, Jack/Nguyen, Tam, 1996, S.1).

Dazu kommen noch die Gegebenheiten der Hardware, die von Rover zu Rover unterschiedlich sind (siehe Tabelle 1). Am größten war diese Hürde bei der ersten Rover Mission, dem Pathfinder Rover oder Sojourner. Der dortige On-board Computer hat nur einen Thread mit 0,5 MB an Speicherplatz und verwendet Solarenergie am Tag und Energie einer nicht aufladbaren Batterie in der Nacht (vgl. Harrison, Reid u. a., 1995, S.3).

Aufgrund der knappen Energiereserven und niedriger Performance beim Multitasking, kann er nur eine Hauptfunktion, wie das Fahren, Lenken oder die Kommunikation, gleichzeitig bewerkstelligen (vgl. Harrison, Reid u. a., 1995, S.3) (vgl. Morrison, Jack/Nguyen, Tam, 1996, S.2).

Als Letztes gibt es noch die Verantwortung, die die Programmierer bei der Entwicklung der Software haben. Eine solche Mission, bei der ein Rover auf einen fremden Planeten geschickt wird, ist fest geplant mit vielen Investitionen, um das Projekt zu verwirklichen. Würde jedoch ein Programmierfehler aufkommen, könnten damit die Chance über den Mars mehr zu erfahren verschwinden und damit auch die Investitionen, wie ebenfalls das Vertrauen in die Organisation (vgl. Gerard J. Holzmann, 2014).

2.3 Techniken zur Fehlervermeidung

Um solche Fehler zu reduzieren, werden Strategien und Regeln aufgestellt.

Einmal ist da das "level of compliance (LOC)" (Gerard J. Holzmann, 2014, Reducing Risk) System, welches je nach LOC des Codes für verschieden wichtige Software zugelassen ist. Beim ersten Level oder "LOC-1" (Gerard J. Holzmann, 2014, Reducing Risk) wird darauf geachtet, dass der Code nicht auf zusätzliche "compiler-specific extensions" (Gerard J. Holzmann, 2014, Reducing Risk) angewiesen ist, also dass keine Extras verwendet werden, um überhaupt den Code zu kompilieren. Dazu muss es noch den "compiler"(Gerard J. Holzmann, 2014, Reducing Risk) und einen "static source code analyzer" (Gerard J. Holzmann, 2014, Reducing Risk) ohne Warnungen bestehen (Statische Source Code Analysator überprüfen den Source Code auf Schwächen, welche zu Schwachstellen führen könnten). Eine wichtige Regel bei "LOC-3" (Gerard J. Holzmann, 2014, Reducing Risk) ist die Verwendung von "assertions" (Gerard J. Holzmann, 2014, Reducing Risk), also eine Behauptung, von der man ausgeht, dass sie an einem bestimmten Punkt im Programm richtig ist. Die minimale Assertionsdichte sollte 2 % betragen und bei einer falschen Assertion das System in einen sicheren Zustand gebracht werden, während das Programm überprüft wird und später weitergeführt werden kann.

Bei allem Code, der kritisch für die Missionen ist, sollte mindestens "LOC-4" (Gerard J. Holzmann, 2014) erreicht werden. Auf diesem Level wird die Verwendung vom "C Preprocessor" (Gerard J. Holzmann, 2014, Reducing Risk) begrenzt.

Die letzten Level "LOC-5 and LOC-6" (Gerard J. Holzmann, 2014, Reducing Risk) sind das Ziel für sicherheitskritische und "human-rated" (Gerard J. Holzmann, 2014, Reducing Risk)

Software. Hier werden noch die restlichen Regeln der “MISRA C coding guidelines” (Gerard J. Holzmann, 2014, Reducing Risk) eingeführt.

Doch selbst bei den strengsten Regeln ist es möglich, dass bestimmte Prozesse ungewolltes Verhalten aufzeigen. Um dieses noch rechtzeitig vor der Mission zu identifizieren, sind Testungen die letzte Hürde.

2.4 Testungen zur Erkennung letzter Mängel

Um möglichst sicherzustellen, dass alle Funktionen der Rover einwandfrei laufen, muss der Rover unter verschiedenen Bedingungen getestet werden und bei unerwünschtem Verhalten überarbeitet werden. Hierbei werden alle verschiedenen Komponenten getestet.

Um die Navigationssoftware in verschiedenen mars-ähnlichen Situationen zu testen, verwendet man Testumgebungen auf einem mars-analogen Terrain. Um erstmal möglichst mars-nahe Landschaft zu erstellen, werden Bilder und andere Messungen von anderen Missionen, wie dem Viking Lander, verwendet, woraus das Mooresche Model abgeleitet wurde (vgl. Harrison, Reid u. a., 1995, S. 20). Dieses kann die Frequenz und Größe der Steine vorhersagen, welche an einem Standort auf dem Mars vorzufinden sind (vgl. Harrison, Reid u. a., 1995, S. 20), womit ein Testbereich für den Pathfinder Rover mit den Maßen 4x12m angefertigt wurde (vgl. Harrison, Reid u. a., 1995, S. 6). Als Test-Rover wird ein analog zum Sojourner gebauter Rover verwendet. Die einzigen Unterschiede sind hier das größere Gewicht, da es Gummiräder statt Metallräder verwendet, und das Fehlen einiger Sensoren (vgl. Harrison, Reid u. a., 1995, S. 5-6). Für die Mars Exploration Rovers wurde ebenfalls ein Testbereich mit einer 9 Meter Breite und 22 Meter Länge angefertigt, so wie ein Rover namens “Surface System TestBed (SSTB)” (Carsten, Joseph u. a., 2009, S. 15). Dieser weicht von den Opportunity und Spirit Rovern ab, da dieser keine Solaranlage besitzt und stattdessen die Energieversorgung mit einem Kabel erfolgt. Auch ein Teil der Elektronik ist in einem sterilen Raum und wird ebenfalls mithilfe eines Kabels zum Rover verbunden (Carsten, Joseph u. a., 2009, S. 15).

So können in einem kleinen, mars-analogen Raum verschiedene Tests durchgeführt werden, doch für längere Strecken wird der Rover im Freien getestet. Dafür verwendet NASA ein Gebiet namens Arroyo Seco, welches neben dem Jet Propulsion Laboratory liegt (vgl. Harrison, Reid u. a., 1995, S. 6). Diese Teststätten werden hauptsächlich verwendet, um die Leistung der einzelnen Navigationsfunktionen zu bestimmen (vgl. Harrison, Reid u. a., 1995, S. 6).

Auch wird ein PC-Emulator verwendet, welcher es ermöglicht das Programm direkt an den UNIX Arbeitsstationen laufen zu lassen und so nicht immer am Prototyp Rover zu testen. (vgl. Morrison, Jack/Nguyen, Tam, 1996, S.6)

Um dann auch die Schnittstellen zu anderen Komponenten wie dem Lander, also der Landungseinheit des Rovers, zu überprüfen wird ein Simulator eingesetzt. Dieser kann auf einem PC oder Laptop als auch auf den UNIX Arbeitsstationen laufen, und simuliert die Kommunikation zwischen den Rover und den Lander. Über den Kommunikationskanal zum Radio Empfänger des Rovers, können beispielsweise Operationsbefehle gesendet werden und so das Verhalten getestet werden. (vgl. Morrison, Jack/Nguyen, Tam, 1996, S.6-7)

Mit diesen verschiedenen Einrichtungen und Methoden kann die Software in verschiedenen Situationen getestet und dann je nach Ergebnis überarbeitet und verbessert werden.

3. Entwicklung der autonomen Software

Die Navigationssoftware für Rover wurde nicht mit einem Mal fertig geplant und dann umgesetzt. Eine Software durchläuft meistens mehrere Versionen wobei immer wieder iteriert und getestet wird. Es werden immer wieder neue Funktionen ausprobiert und andere wiederum gestrichen. So auch bei der Navigationssoftware, deren Entwicklung man besonders an den verschiedenen Mars Missionen sehen kann. Aber auch während der Operationen gab es Neuerungen, die erst nachträglich zum Rover gesendet und verwendet wurden.

Hier wird die Entwicklung der verschiedenen Versionen, als auch deren Funktionsweisen beschrieben.

3.1 Mars Pathfinder Mission

Die erste Mars Mission war eine besondere Herausforderung für die Entwickler, da bis zu diesem Zeitpunkt, dem 4. Juli 1997, noch kein anderer Rover auf dem Mars gelandet und eingesetzt wurde. Es war auch das ursprüngliche Missionsziel, einen Rover auf den roten Planeten zu bringen. (vgl. NASA, o. J., Mars Pathfinder)

Wegen der limitierten Elektronikleistung (siehe Tabelle 1), den Missionszielen und aufgrund des kurzen Entwicklungszyklus, wurde die Software Architektur von vier zentralen Eigenschaften motiviert: “Reliability” (Morrison, Jack/Nguyen, Tam, 1996, S.1), “Flexibility” (Morrison, Jack/Nguyen, Tam, 1996, S.1), “Simplicity” (Morrison, Jack/Nguyen, Tam, 1996, S.1) und “Visibility” (Morrison, Jack/Nguyen, Tam, 1996, S.2). Reliability soll garantiert

sein, da bei einem Fehler kein menschliches Eingreifen möglich ist. Flexibility, heißt sich den Gegebenheiten der Hardware und der Umgebung anzupassen. Simplicity hat die Vorteile, dass die Lösungen zuverlässig, flexibel und schnell zu entwickeln sind, sowie leicht zu testen sind. Visibility ist auf die Zustände der Elektronikkomponenten und des Programms bezogen, welche möglichst oft übermittelt werden sollen. (vgl. Morrison, Jack/Nguyen, Tam, 1996, S.1-2)

Um für den Rover ein Ziel festzulegen, muss man mithilfe von Stereo Bilder Wegpunkte festlegen. Die Funktionsweise der autonomen Navigation, um ein Wegpunkt zu erreichen, erfolgt mit einer einfachen Kontrollschleife. Der Mars Pathfinder Rover fährt ein Reifenradius von 6,5 cm (vgl. Harrison, Reid u. a., 1995, S.16) bei 0,67 cm/s (vgl. Harrison, Reid u. a., 1995, S.19), hält dann an und scannt die Umgebung. Dies geschieht mit Infrarotsensoren und CCD Kameras, welche eine Karte mit den ungefähren Höhen des Terrains erstellen (vgl. Morrison, Jack/Nguyen, Tam, 1996, S.2). Falls ein Hindernis identifiziert wird, dreht sich der Rover um seinen Mittelpunkt in die Richtung, die einen kleineren Drehwinkel besitzt oder zum Wegpunkt, wenn die Winkel gleich sind. Kommt kurz darauf ein weiteres Objekt in den Weg, dreht er sich in dieselbe Richtung weiter, um ein hin und her Drehen zu vermeiden (vgl. Morrison, Jack/Nguyen, Tam, 1996, S.3). Wenn der Weg frei ist, fährt der Sojourner erst ein paar Segmente gerade aus und nimmt dann wieder in einem Bogen den Kurs zum Wegpunkt auf. Dieser Zyklus kann durch eine davor festgelegte Zeitüberschreitung, Sensoranzeige außerhalb der sicheren Grenzen, unmittelbare Nähe des Landers oder einen physischen Kontakt unterbrochen werden (vgl. Morrison, Jack/Nguyen, Tam, 1996, S.2). Bei diesen Vorkommnissen, wird die Route zum Wegpunkt abgebrochen. Wurde davor aber spezifisch aufgetragen, der Rover solle eigenständig fahren, fährt dieser erst zurück, dreht sich, und versucht es erneut (vgl. Morrison, Jack/Nguyen, Tam, 1996, S.3).

Es gibt noch die Möglichkeit sich für eine bestimmte Distanz durch schmälere Wege zu "zwängen" (Morrison, Jack/Nguyen, Tam, 1996, S.3), wobei man dafür nicht mehr den Platz einberechnet, um sich auf der Stelle zu drehen. Wenn der Sojourner doch auf ein Hindernis stößt, so fährt er zurück und sucht nach einer Alternative. Ein weiterer Modus der Operation ist "Rock finding" (Morrison, Jack/Nguyen, Tam, 1996, S.3), wo bei einem außergewöhnlichen Felsen der Mars Pathfinder Rover zum Stein zentriert wird und ihn mit dem Spektrometer scannt. Falls als Erstes das Ziel erreicht wird, sucht er in einer Spirale nach einem Gestein. (vgl. Morrison, Jack/Nguyen, Tam, 1996, S.3)

3.2 Mars Exploration Rover (MER)

Der Sprung vom Mars Pathfinder Rover zu dem Mars Exploration Rover bringt die meisten Änderungen und neue Features zur autonomen Navigation (AutoNav) hervor, damit gibt es aber auch viele Funktionen, welche es nicht in die Mars Exploration Rover geschafft haben. Einige dieser Funktionen kamen jedoch im Laufe der Mission mit hinzu, wie z. B. der "D* Planner" (Maimone, Mark, 2021, 46:43-47:07), welcher zum globalen Wegplanen verwendet wird.

3.2.1 Lokales Wegplanen mit RoverBug für Rocky 7

Das System auf dem "Rocky 7" (Baumgartner, Eric u. a., 2000, S.3) unterscheidet sich von dem, welches final bei Opportunity und Spirit verwendet wird. Damit ist dies ein Beispiel für ein nicht verwendetes Feature, welches in Betracht gezogen wurde, aber durch den später beschriebenen Wegplaner ersetz wurde. Der Test Rover hat die Fähigkeit, Wissenschaftszielorte und das Ziel festzulegen, wobei von einem Programm namens "CASPER (Continuous Activity Scheduling, Planning, Execution, and Replanning)" (Balaram, J. u. a., 2000, S.2) Zwischenziele angelegt werden, die sich je nach neuen Informationen dynamisch an eine mögliche Route anpassen. Wenn es bei den Eingabebedingungen der Wissenschaftszielorte, den Zuständen des Rovers oder der Position zum Konflikt kommt, wird der Plan mithilfe des "iterative repair" (Daun, Brian u. a., 1994, S.1) Algorithmus angepasst, indem ein oder mehrere Modifikationen am Plan gemacht werden. CASPER hat auch die Möglichkeit auf eine globale Karte, theoretisch generiert von Bildern des Landers, zuzugreifen und damit einen effizienteren Weg zu planen. (vgl. Balaram, J. u. a., 2000, S.2) Durch das dynamische Neuplanen auf unerwartete Ereignisse, kann der Operator Befehle auf einem höheren Level senden, wo bei dem Mars Pathfinder Rover eine genaue und arbeitsintensivere Beschreibung der Ausführung gegeben werden musste. (vgl. Balaram, J. u. a., 2000, S.1) (vgl. Baumgartner, Eric u. a., 2000, S.3) Während der Entwicklung haben sich die Algorithmen, die für das Pathplanning verwendet werden, verändert. Für den limitierten Sichtbereich oder "wedge" (Laubach, Sharon, 1999, S.76) wurde der "Wedgebug" (Laubach, Sharon, 1999, S.71) auf der Basis von "TangentBug" (Laubach, Sharon, 1999, S.56) erstellt. Dabei wird die eingeschränkte Prozessorleistung berücksichtigt, weswegen es zu Optimierungen kommt, wie die Berechnung der kleinsten Nummer an Scans, die zu erheben sind (vgl. Balaram, J. u. a., 2000, S.2). "RoverBug" (Laubach, Sharon, 1999, S.122) ist die Implementation von Wedgebug auf dem Rocky7 Test Rover. Hierbei werden weitere Probleme spezifisch an den Rover

angepasst, wie zum Beispiel, dass ein Rover kein Punkt ist, wie in TangentBug und Wedgebug modelliert (vgl. Laubach, Sharon, 1999, S.121). Der RoverBug Planer verwendet dabei nur die lokalen Sensoren des Rovers, wobei im Sichtfeld ein "local tangent graph, or LTG" (Laubach, Sharon, 1999, S.46) erstellt wird. Der LTG beinhaltet alle Verbindungen zwischen dem Startpunkt, den Hindernisgrenzen und dem nächsten Punkt zum Ziel (vgl. Laubach, Sharon, 1999, S.75) (siehe Abbildung 1). Dabei werden von der Höhenkarte Hindernisse abgelesen und in "convex hulls" (Baumgartner, Eric u. a., 2000, S.3) umhüllt, welche die Größe des Rovers und eine Sicherheitsdistanz im Fall von einer veränderten Position miteinberechnen (vgl. Laubach, Sharon, 1999, S.128).

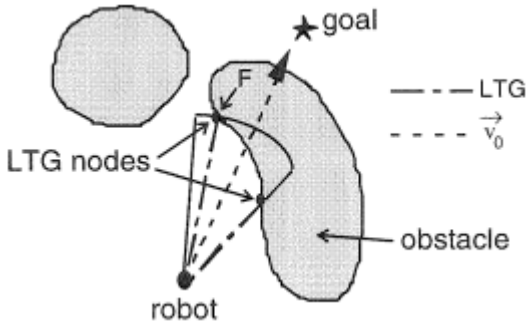

Abbildung 1: Local Tangent Graph

(vgl. Laubach, Sharon, 1999. S.76)

Je nach Situation geht er in einen von zwei Operationsmodi über. Einmal gibt es "motion-to-goal" (Baumgartner, Eric u. a., 2000, S.3), welcher standardmäßig eingesetzt wird. Das Ziel ist es näher an den festgelegten Wegpunkt zu kommen, das heißt die Distanz zwischen Rover und Endpunkt zu verringern. Dabei kann entweder kein Hindernis im Weg sein und damit der Rover direkt zum Punkt im Blickfeld mit der geringsten Distanz zum Zielpunkt fahren ("direct mode") (Laubach, Sharon, 1999, S.79) oder an einer Grenze eines Hindernisses entlang manövrieren, solange die Distanz zum Wegpunkt abnimmt ("sliding mode") (Laubach, Sharon, 1999, S.79). Wenn keine Route in Sicht ist, die näher zum Ziel kommt, so schaltet RoverBug auf den Modus "boundary-following" (Laubach, Sharon, 1999, S.88) um. Hierbei scannt der Rover das blockierende Objekt von der aktuellen Position aus, um die effizientere Richtung vom boundary-following zu bestimmen (vgl. Laubach, Sharon, 1999, S.88). Dann folgt der Rover der Grenze des Hindernisses, bis entweder wieder ein Näherkommen des Wegpunktes möglich ist und zurück zu motion-to-goal wechselt oder bis eine Runde um das Objekt gedreht wurde und das Ziel als unerreichbar deklariert wird. (vgl. Balaram, J. u. a., 2000, S.4)

3.2.2 Lokales Wegplanen mit GESTALT auf den Mars Exploration Rover

Für das lokale Navigationssystem auf den Mars Exploration Rovers, wurde statt dem RoverBug Algorithmus ein neuer Algorithmus mit dem Namen “Grid-based Estimation of Surface Traversability Applied to Local Terrain (GESTALT)” (Goldberg, Steven u. a., 2002, S.1) entwickelt. Zum Festlegen eines Wegpunktes wird entweder eine X, Y und Z Koordinate bereitgestellt oder ein bestimmtes Merkmal in der Umgebung festgelegt und daraus dynamisch die Position ermittelt. Der Algorithmus versucht dann mithilfe von verschiedenen Sensor-Daten die effizienteste und sicherste Route zu bestimmen, welche dann an die Rädermotoren gesendet wird. (vgl. Goldberg, Steven u. a., 2002, S.4) Dies geschieht mit einem “navigation cycle” (Goldberg, Steven u. a., 2002, S.4), welcher als erstes überprüft, ob der Rover an seinem Ziel angekommen ist. Da aber der Rover kein Punkt ist, wird die aktuelle Position als eine 5 Meter Scheibe mit einem gewissen Toleranzbereich modelliert (vgl. Maimone, Mark, 2021, 16:20-16:44).

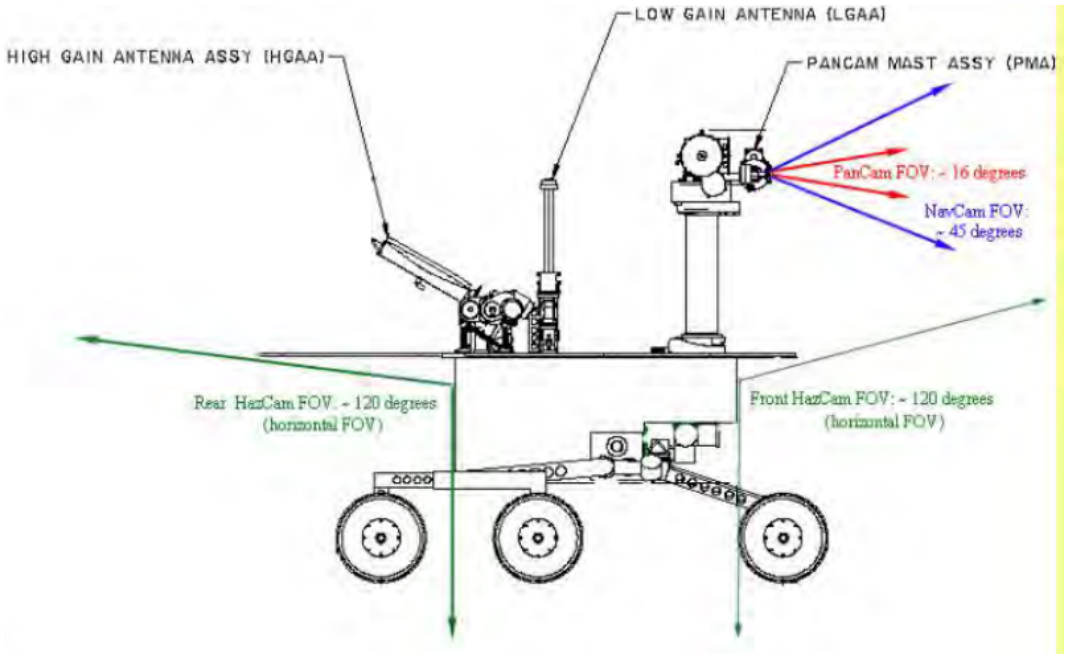

Abbildung 2: Mars Exploration Rover Kamera Positionen

(vgl. Maimone, Mark, 2021, Folie 8)

Falls das nicht zutrifft, so scannt der Mars Exploration Rover die Umgebung mit den Kameras auf dem Mast, genannt “PanCam” und “NavCam” (Maimone, Mark, 2021, 9:49) und den “HazCam” (Maimone, Mark, 2021, 9:49), die unmittelbar vor und hinter dem Rover scannen. (siehe Abbildung 2) Allein mit den HazCams kann man zweimal dessen Länge, also etwa 3,2 Meter, vorausschauen (vgl. Maimone, Mark, 2021, 10:07-10:17). Diese Bilder werden mit dem Stereo Vision Algorithmus evaluiert, indem erst die Pixelqualität von 1024 × 1024 auf 256 × 256 reduziert wird, um die Berechnung schneller auszuführen (vgl.

Goldberg, Steven u. a., 2002, S.2). Dann wird die Differenz zwischen den Pixeln im linken und rechten Bild überprüft, um die Entfernung zu berechnet. Um sicherzustellen, dass diese Daten auch stimmen, werden verschiedene Filtern verwendet (vgl. Biesiadecki, Jeffrey/Maimone, Mark, 2006, S.8). Einer von denen ist der "Left/Right Line of Sight filter" (Goldberg, Steven u. a., 2002, S.3), welcher die Differenz von rechten zum linken Bild als auch vom linken zum rechten Bild berechnet und bei keiner Übereinstimmung der Differenz, diese Daten nicht weiterverwendet.

Daraus werden die Koordinaten abgeleitet und zur Erstellung einer zweidimensionalen lokalen Karte verwendet, welche aus Zellen jeweils mit der Größe eines "mechanically-determined obstacle" (Biesiadecki, Jeffrey/Maimone, Mark, 2006, S.7) von "0.2m × 0.2m" (Goldberg, Steven u. a., 2002, S.6) besteht. Nur Informationen in einem 5 Meter Radius werden in der Karte behalten (vgl. Biesiadecki, Jeffrey/Maimone, Mark, 2006, S.7). Dabei wird jedes Quadrat auf seine "Traversability" (Goldberg, Steven u. a., 2002, S.6) beurteilt, indem ein "goodness" (Goldberg, Steven u. a., 2002, S.7) Wert von 0 bis 255 errechnet wird. Dieses Maß wird von den Arten an Hindernissen, die in einem Areal von der Größe des Rovers zu finden sind, beeinflusst. Die erste Art ist der "Step Hazard" (Goldberg, Steven u. a., 2002, S.7), welcher vernachlässigt wird, falls der maximale Höhenunterschied zwischen zwei Zellen (e) in einer Zone(vgl. Biesiadecki, Jeffrey/Maimone, Mark, 2006, S.9f) von der Größe der "Warm Electronics Box (WEB)" (Biesiadecki, Jeffrey/Maimone, Mark, 2006, S.7), kleiner als 1/3 von der überwindbaren Höhe (h) ist. Sonst wird der Wert mit "goodness 127 * (1 – min(1, e/h))" (Goldberg, Steven u. a., 2002, S.7) errechnet. Auch wird auf "Roughness Hazard" (Goldberg, Steven u. a., 2002, S.7) überprüft, wobei der Restbereich an "planar fit r" (Goldberg, Steven u. a., 2002, S.7), also ebenen Boden, mit der "roughness fraction * 1/3 h" (Goldberg, Steven u. a., 2002, S.7) verglichen wird und bei einem kleineren Wert, wie beim Step Hazard nicht beachtet wird. Falls der Wert größer oder gleich ist, so beträgt die goodness "255 * (1 – min(1, roughness fraction * r/h))" (Goldberg, Steven u. a., 2002, S.7). Dann wird noch die Neigung im "Pitch Hazard" (Goldberg, Steven u. a., 2002, S.7) beurteilt, wo bei einer Steigung s vom ebenen Boden der goodness Wert mit "255 * (1 – min(1, s/max pitch angle))" (Goldberg, Steven u. a., 2002, S.7) berechnet wird. Zum Schluss wird noch der "Border Hazard" (Goldberg, Steven u. a., 2002, S.7) evaluiert, wo der goodness Wert mit "border goodness" (Goldberg, Steven u. a., 2002, S.7) gleichgesetzt wird, falls eine Zelle an einer Zelle mit unbekanntem Wert grenzt. Bei der goodness Berechnung mit unbekannten Daten, wird nur ein Teil des Rovers modelliert. Dabei kommt es aber vor, dass sich Objekte genau an der Grenze der Goodness Map befinden und somit ausgewichen werden (vgl.

Biesiadecki, Jeffrey/Maimone, Mark, 2006, S.9). Wenn aber im nächsten Schritt das Hindernis exakt außerhalb der lokalen Karte ist und damit nicht miteinberechnet wird, kann es dazu kommen, dass sich der Rover zu diesem hazard wieder hinbewegt (vgl. Biesiadecki, Jeffrey/Maimone, Mark, 2006, S.9). Zur Lösung dieses Problems werden "Layered Goodness Maps" (Biesiadecki, Jeffrey/Maimone, Mark, 2006, S.9) verwendet. Hierbei werden goodness values für Radien von einer Zelle bis zu sieben Zellen berechnet (vgl. Biesiadecki, Jeffrey/Maimone, Mark, 2006, S.7). Der finale goodness Wert ist die kleinste Zahl von den Ergebnissen der Hazard Berechnungen. (vgl. Goldberg, Steven u. a., 2002, S.7) (vgl. Biesiadecki, Jeffrey/Maimone, Mark, 2006, S.9)

Daraus wird dann die effizienteste Kurve genommen, welches mit "arc votes" (Goldberg, Steven u. a., 2002, S.7) bestimmt wird. Die erste Abstimmung sind die "Hazard Arc votes" (Goldberg, Steven u. a., 2002, S.7), welche mit einer gewichteten Summe die goodness Werte aller Kurven berechnen, wobei nähere Zellen stärker gewichtet werden als weiter entfernte. Die zweite Gewichtung sind die "Waypoint Arc votes" (Goldberg, Steven u. a., 2002, S.7), welche den besten goodness Wert von 255 der Kurve geben, die am nächsten zum Wegpunkt kommt. Dieser Wert bildet den Hochpunkt der "gaussian curve" (Goldberg, Steven u. a., 2002, S.7), welche zu den restlichen Bögen angewandt wird. Bei den Kurven, die rückwärts gehen, ist der Hochpunkt statt 255 der Wert 128 (vgl. Goldberg, Steven u. a., 2002, S.7). Zuletzt erfolgen die "Steering bias votes" (Carsten, Joseph u. a., 2009, S.3). Hier werden niedrige Werte vergeben bei Wegen, die eine höhere Zeit zum Drehen der Räder aus der aktuellen Orientierung benötigen (vgl. Carsten, Joseph u. a., 2009, S.3). Diese Votes werden dann zu einem goodness Wert je Pfad zusammengefügt. Falls einer der Werte unter einem Grenzwert liegen, so wird der kleinere Wert genommen, sonst wird eine gewichtete Summe errechnet mit den "certainties" (Goldberg, Steven u. a., 2002, S.7) der goodness Werte als Gewicht. Während der Fahrt werden keine Bilder zur weiteren Terrain-Analyse erhoben (vgl. Goldberg, Steven u. a., 2002, S.4). Am Ende wird dann noch die Position des Rovers berichtigt, falls es zu Rutschen kam. (vgl. Goldberg, Steven u. a., 2002, S.4)

Diese Vorgehensweise bringt den Rover in den meisten Fällen zum Ziel, doch gibt es hier Ausnahmen. Wie z. B. am 108 sol (Mars Tag) von Spirit, konnte der Rover sich nicht um ein Hindernis-Feld bewegen, auf dessen anderer Seite sich der Wegpunkt befand. Dadurch, dass der Waypoint Arc vote Spirit näher zum Ziel bewegen will, aber der Hazard Arc vote das Areal vom Feld der Steine nicht zulässt, kommt es dazu, dass der Spirit Rover nicht zum Ziel kommt. (vgl. Carsten, Joseph u. a., 2009, S.4)

3.2.3 Globales Wegplanen mit dem Field D* Algorithmus

Um dieses Problem zu vermeiden, wird ein weiterer Planer, der "Field D* algorithm" (Carsten, Joseph u. a., 2009, S.5), zusätzlich für die Wege verwendet, welcher alle verfügbaren Informationen miteinbezieht. Also wird des Weiteren eine lokale Karte nur mit Daten in einem 5 Meter Radius verwendet, als auch eine globale Rasterkarte mit allen erhobenen Daten des Terrains, welche fixiert zum Terrain ist (vgl. Carsten, Joseph u. a., 2009, S.6). Da nun alles zur Planung eines Pfades verwendet wird, können globale Feldplaner effizientere Routen finden, was mit mehr Daten der Umgebung weiter verbessert wird. (vgl. Carsten, Joseph u. a., 2009, S.5)

Statt mit einer Goodness Map arbeitet der Algorithmus mit einer "Cost Map" (Carsten, Joseph u. a., 2009, S.6), welche die Kosten für den Aufwand zur Überwindung einer Zelle darstellt. Um die Cost Map mit der Goodness Map zu aktualisieren, wird als erstes ermittelt, wo sich die lokale Karte in der größeren globalen Karte befindet. (vgl. Carsten, Joseph u. a., 2009, S.6) Damit auch jeder Wert zugeordnet werden kann, müssen die Zellen der verschiedenen Karten dieselbe Größe und selben Kanten besitzen. Das Problem wird bei der Erstellung der Karten beseitigt. Für den Cost Wert wird der goodness Wert einer Zelle invertiert, außer bei Zellen mit unbekannten Werten und mit sehr niedrigen goodness. (vgl. Carsten, Joseph u. a., 2009, S.7) Bei nicht bekannten Daten, wird ein zuvor bestimmter Betrag zugewiesen. Dieser ist nicht zu hoch, was die bereits erkundeten Bereiche bevorzugen würde, und auch nicht zu niedrig, was nur zur Neuerkundung führen würde. (vgl. Carsten, Joseph u. a., 2009, S.6) Bei sehr niedrigem goodness wird ein spezieller Cost Wert zugewiesen. Dieser soll Hindernisse darstellen, wodurch kein Weg geplant wird. (vgl. Carsten, Joseph u. a., 2009, S.7)

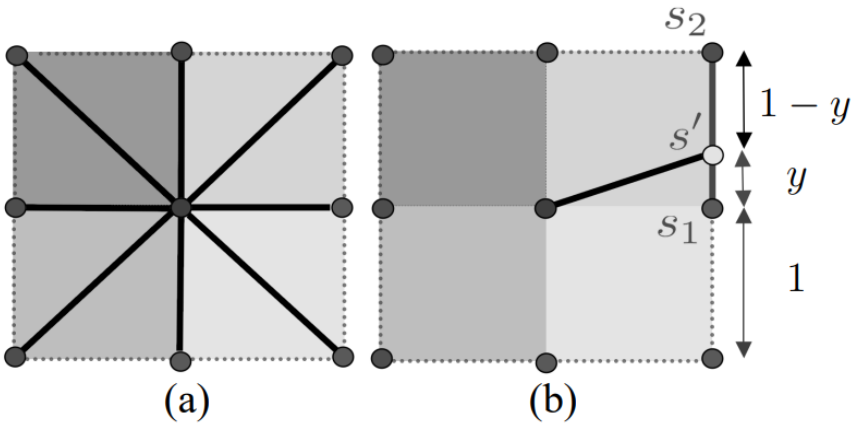

Abbildung 3: Traditionelle Rasterkarte (a) und mit linearer Interpolation (b)

(vgl. Carsten Joseph u. a., 2009, S.7)

In traditionellen Rasterkarten ist es nur möglich von Ecke zu Ecke zu planen (siehe Abbildung 3a), was zu unnötigen Drehungen und damit zu nicht optimalen Routen führt. Field D* kann statt nur durch die Ecken auch zu den Kanten der Zellen planen (siehe Abbildung 3b). Um die Kosten anzunähern, wird "linear interpolation" (Carsten, Joseph u. a., 2009, S.5) verwendet (siehe Formel 1). Hierbei werden die Kosten zum Knoten s' ungefähr berechnet, wobei s_1 und s_2 als bekannt vorausgesetzt werden. Der Wert y ist die Distanz zwischen s' und s_1 in der Längeneinheit der Zellen. (vgl. Carsten, Joseph u. a., 2009, S.5)

Formel 1: Lineare Interpolation zur ungefähren Berechnung von den Kosten

(vgl. Carsten, Joseph u. a., 2009, S.5)

$$PathCost(s') \approx y \cdot PathCost(s_2) + (1 - y) \cdot PathCost(s_2)$$

Um die Route der autonomen Navigation von GESTALT zu beeinflussen, werden zusätzliche Votes der Bögen aus diesem System eingeführt. Dafür wird von jedem Ende der Kurven die Überwindungskosten zum Ziel berechnet und der mit dem niedrigsten Wert bevorzugt. (vgl. Carsten, Joseph u. a., 2009, S.7) Es werden zwei neue Abstimmungen pro Route erzeugt. Einmal v_{scale} (siehe Formel 2a), was die verschiedenen Kosten auf einer linearen Bandbreite von Vote Werten zuweist. Und dann noch v_{close} (siehe Formel 2b), was bei weiten Entfernungen zum Ziel hohe Werte annimmt, da c_{min} fast so groß wie c_{max} ist. Bei immer kleineren Entfernungen, wird der Unterschied zwischen c_{min} und c_{max} größer und führt dazu, dass der Vote zwischen dem Pfad mit den niedrigsten Kosten und dem mit den höchsten Kosten weiter auseinander gehen. (Carsten, Joseph u. a., 2009, S.8)

Formel 2: Berechnungen der Votes

(vgl. Carsten, Joseph u. a., 2009, S.8)

$$a) \qquad v_{scale_i} = v_{max} \cdot \frac{c_{max} - c_i}{c_{max} - c_{min}}$$

$$b) \qquad v_{close_i} = v_{max} \cdot \frac{c_{min}}{c_i}$$

So werden die Abstimmungen der Kurven mit den Votes von GESTALT kombiniert und führen zu robusterer autonomer Navigation. (Carsten, Joseph u. a., 2009, S.8)

3.3 Mars Science Laboratory (MSL)

Der Mars Science Laboratory Rover Curiosity, hat im Gegensatz zu den anderen Rovern keine Änderungen der autonomen Navigationssoftware und verwendet dieselbe Software wie seine Vorgänger. Da aber Curiosity größer als die MER ist, müssen vier Bilder zur Terrain-Analyse aufgenommen werden (siehe Tabelle 1).

3.4 Mars 2020 Mission: Neuerungen autonomer Navigation

Die aktuelle Mars 2020 Mission von NASA und der Perseverance Rover haben im Kontrast zu Curiosity viele weitere Optimierungen zur autonomen Navigationssoftware der MER vorgenommen.

Ein präventives Feature des Programms der Mars Exploration Rover ist, dass der Rover als eine 5 Meter Scheibe simuliert wird und damit konservativ den Hindernissen ausweicht. Der Perseverance Rover wird nicht mehr so dargestellt, sondern wird jeder einzelne Reifen und die Höhe des Körpers modelliert. Dies ermöglicht dem Rover sich über kleine Hindernisse hinweg zu bewegen (siehe Abbildung 4). (vgl. Maimone, Mark, 2021, 51:18-51:46) (vgl. Jet Propulsion Laboratory, o. J., Enhancement 1: Straddling)

Eine weitere Erweiterung, ist das neue lokale Planen mit einem Kurvenbaum (siehe Abbildung 5), statt nur einzelnen Kurven. Hiermit ist es möglich weiter im Voraus zu planen und den effizientesten Weg durch diesen Baum zu wählen. (vgl. Maimone, Mark, 2021, 50:40-51:00) Dazu kommt noch der globale Planer, welcher von jedem Endpunkt der Routen im Baum die Kosten berechnet. (vgl. Maimone, Mark, 2021, 51:06)

4. Simulation der Navigationssoftware

Die Simulation konzentriert sich darauf, den obigen beschriebenen Algorithmus des Mars Pathfinder Rovers in einer Testumgebung zu demonstrieren. Hierfür wird die Game- und Simulation-Engine Unity verwendet, welche das Rendern der 3D-Modelle bewerkstelligt und viele vorgefertigten Methoden, unter anderem zur Bewegung, bereitstellt. Als Programmiersprache wird C# verwendet.

4.1 Vorbereitung zur Erstellung des Programms

Um sich hauptsächlich mit der autonomen Navigation zu befassen, werden die Stereo Bilderfassung und Bearbeitung, sowie die Abweichung der tatsächlichen Position mit dem vermuteten Ort, vereinfacht bzw. weggelassen. Dabei wird angenommen, dass der Rover ein Sichtfeld von 1 Meter vom Mittelpunkt nach vorne besitzt und die nötigen Daten für die autonome Navigation erfasst hat. Auch wird vereinfacht angenommen, dass der Sojourner sich bei der Erkennung eines Hindernisses, immer nach rechts dreht.

Zu Beginn wird ein Demonstrationsareal mit einer mars-ähnlichen Landschaft und Gesteine von verschiedenen Größen und Häufigkeit eingerichtet, worauf ein 3D Modell des Mars Pathfinder Rovers platziert wird (siehe Abbildung 6).

Unity spezifische Komponente auf den Rover, darunter ein “Capsule Collider” für den Sichtbereich (siehe Abbildung 7) und mein C# Skript MarsPathfinderRover.cs (siehe Source Code Anhang), werden hinzugefügt, um die Verhaltensweise des Modells zu programmieren. Die Bewegung der Kamera in der Simulation und die Logik von einem Mausklick eine Koordinate des Bodens zu bekommen, sind in den Dateien CameraMovement.cs und Waypoint.cs programmiert, welche hier nicht weiter beschrieben werden.[1] Das Verhalten des Wegpunktes auf dem Bildschirm angezeigt zu werden (siehe Abbildung 8) wird ebenfalls in Waypoint.cs geregelt.
Einige Funktionen werden noch Tasten der Tastatur zugeordnet. W, A, S und D zum Bewegen der Kamera nach vorne, links, hinten und rechts, R für den Reset der Kamera, F zum Neuladen der Testumgebung, T um den aktuellen Wegpunkt zu terminieren und zum Schluss noch Escape zum Schließen des Programms.

4.2 Beschreibung der Funktionsweise des Source Codes

In der Region der Attribute (gekennzeichnet mit `#region` bis `#endregion`), werden die Konstanten (erkennbar an dem Schlüsselwort `const`), wie z. B. die Geschwindigkeit von Sojourner von 6,7 cm/s (vgl. Harrison, Reid u. a., 1995, S.19), aber auch Variablen, die den aktuellen Zustand widerspiegeln, so als Beispiel `Hazards` was die Anzahl an Hindernissen im Sichtfeld anzeigt (siehe Source Code Anhang). In der privaten Methode `Start()` werden zur Initialisierung zwei Methoden aufgerufen. Die Erste (`ParallelToGround()`) orientiert den Rover parallel zum Boden und die Zweite (`StartCoroutine(ScanningCoroutine())`) startet das Scannen des Terrains. Als Nächstes kommt die Hauptmethode `Update()`, was die Kontrollschleife des Mars Pathfinder Rovers ist. Erst wird überprüft, ob es Wegpunkte gibt, ohne diese das Programm nicht läuft. Kommt es zu einem Wegpunkt, durch einen linken Mausklick in der Testumgebung, wird abgefragt, obgleich der Rover fahren darf und keine Hindernisse vor ihm sind. Ist dies nicht der Fall, rotiert er auf der Stelle bis er einen hindernisfreien Sichtbereich erreicht (`RotateInPlace()`). Darf Sojourner fahren, so wird erst das Erreichen des Wegpunktes geprüft und in dem Ereignis der Wegpunkt entfernt (`Waypoint.Instance.Waypoints.Dequeue()`). Sonst fährt der Rover vorwärts (`MoveForward()`), rotiert dann zum Wegpunkt (`RotateTowardsWaypoint()`) und richtet sich wieder parallel zum Boden. Wenn die Distanz ohne scannen kleiner gleich null ist,

[1] Auf GitHub sind alle Programm Dateien unter „Assets>Scripts“: https://github.com/OsokaOiv/Mars-Rover-Simulation

so wird das Scannen gestartet. Das Scannen wird mit den zwei Methoden geregelt, namens `OnTriggerEnter(Collider other)` und `OnTriggerExit(Collider other)`, welche automatisch aufgerufen werden, falls ein Stein in bzw. aus dem Sichtfeld kommt und dementsprechend die `Hazards` um eins erhöht bzw. verringert. Genauere Beschreibung der Methoden sind in den Kommentaren (gekennzeichnet mit `//`) des Source Codes (siehe Source Code Anhang).

4.3 Ergebnisse des Algorithmus in der Testumgebung

Bei dem Versuch Sojourner in der Testumgebung[2] auf verschiedene Wegpunkte fahren zulassen, erreicht er in den meisten Fällen das Ziel (siehe Abbildung 8). Doch gibt es Grenzen, die mit diesem einfachen Algorithmus zu erreichen sind, wie z. B., wenn der Wegpunkt in die Mitte eines großen Objektes platziert wird, kommt der Rover nie an und umkreist das Hindernis wiederholt (siehe Abbildung 9).

5. Zukünftige Aussichten

Dem Weg, der die autonome Navigation der Rover überwunden hat, zeigt, dass die Software einige Iterationen durchlaufen ist, bevor es zu dem aktuell verwendeten Programm des Perseverance Rover gekommen ist. Wenn wir davon ausgehen, wie die Hardware immer leistungsfähiger wird und mehr Speicher besitzen wird, so kann die Navigation immer mehr Variablen miteinbeziehen und somit effizientere Routen für die kommenden Rover planen. Mit der steigenden Leistungsfähigkeit wird es bald auch möglich sein ohne anzuhalten autonom in Echtzeit zu fahren und so viel mehr Orte des Mars erforschen. So wird es in Zukunft nicht mehr nötig sein, auf Fahrten ohne Autonomie zurückzugreifen.

Dazu könnte es zu Analysen des Bodens kommen, also auf was für ein Terrain der Rover fahren wird und damit eine Gefahr des Steckenbleibens oder Ausrutschens (vgl. Maimone, Mark, 2021, 53:07-54:54).

[2] Die Simulationssoftware ist hier fertig kompiliert und einsatzbereit für Windows unter dem Passwort „MarsRoverSim“ geschützt: https://osoka.itch.io/mars-pathfinder-rover-simulation

6. Literaturverzeichnis

6.1 Papers

- Balaram, J. u. a.: Enhanced Mars Rover Navigation Techniques, IEEE International Conference on Robotics and Automation (ICRA), San Francisco CA, April 24-28 2000.
- Baumgartner, Eric u. a.: Technology Development and Testing for Enhanced Mars Rover Sample Return Operations, IEEE Aerospace Conference, Big Sky, Montana, March 18-25, 2000.
- Biesiadecki, Jeffrey/Maimone, Mark: The Mars Exploration Rover surface mobility flight software: Driving ambition. In 2006 IEEE Aerospace Conference Proceedings, Big Sky, MT, 2006.
- Carsten, Joseph u. a.: Global Planning on the Mars Exploration Rovers: Software Integration and Surface Testing, Journal of Field Robotics, 26(4), 2009.
- Daun, Brian u. a.: Scheduling and Rescheduling with Iterative Repair. In J. Zweben and M. Fox, editors, Intelligent Scheduling, pages 241-256. Morgan Kaufman, 1994.
- Goldberg, Steven u. a.: L. Stereo Vision and Rover Navigation Software for Planetary Exploration. IEEE Aerospace Conference, Big Sky, Montana, March 2002.
- Harrison, Reid u. a.: Mars Microrover Navigation: Performance Evaluation and Enhancement, Autonomous Robots Journal, Special Issue on Autonomous Vehicles for Planetary Exploration, 2(4), 1995.
- Laubach, Sharon: Theory and Experiments in Autonomous Sensor Based Motion Planning with Applications for Flight Planetary Microrovers. PhD thesis, California Institute of Technology, May 1999.
- Morrison, Jack/Nguyen, Tam: On-Board Software for the Mars Pathfinder Microrover, John Hopkins University, Applied Physics Laboratory, Laurel, Maryland, 1996.

6.2 Websites

- Gerard J. Holzmann (2014): Mars Code, https://cacm.acm.org/magazines/2014/2/171689-mars-code/fulltext?mobile=false (Stand: 08.11.2021)

- Jet Propulsion Laboratory (Hrsg., o. J.): Enhanced Autonav for Mars 2020 Rover: Introduction, https://trs.jpl.nasa.gov/bitstream/handle/2014/48231/CL%2317-3124.pdf (Stand: 08.11.2021)
- Maimone, Mark (03.05.2017): The Evolution of Autonomous Capabilities on NASAs Mars Rovers, Southern California Robotics Symposium 2017, [YouTube] https://www.youtube.com/watch?v=u4-4x8GhE6Y (Stand: 08.11.2021)
- Maimone, Mark (15.09.2021): Evolution of Mars Rover Surface Navigation Autonomy, NASA Engineering and Safety Center (NESC) Academy, Guidance Navigation and Control Webcast, https://nescacademy.nasa.gov/video/62a838b3d8a54a9095dffbba69da76701d (Stand: 08.11.2021)
- NASA (Hrsg., o. J.): Mars Pathfinder, https://mars.nasa.gov/mars-exploration/missions/pathfinder/ (Stand: 08.11.2021)

Anhang

Abbildungen

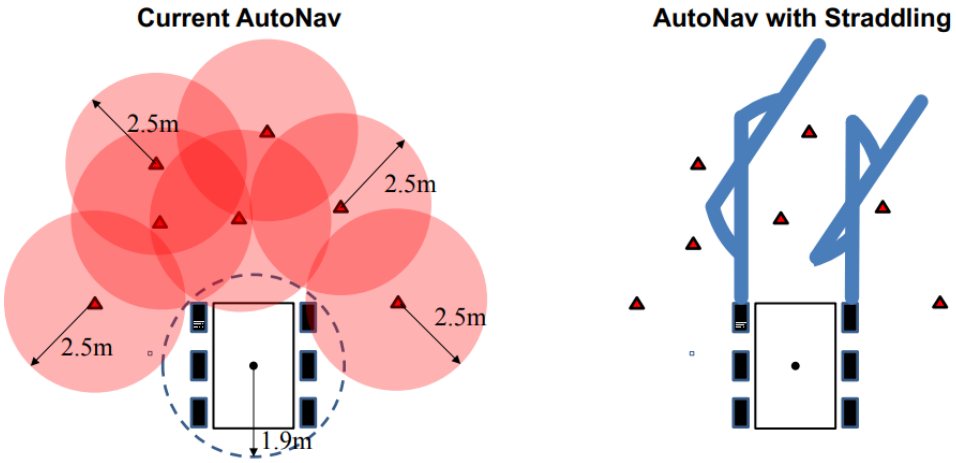

Abbildung 4: Links konservative Herangehensweise von MER und rechts Bewegung über kleine Hindernisse hinweg bei Mars 2020

(vgl. Jet Propulsion Laboratory, o. J, Enhancement 1: Straddling)

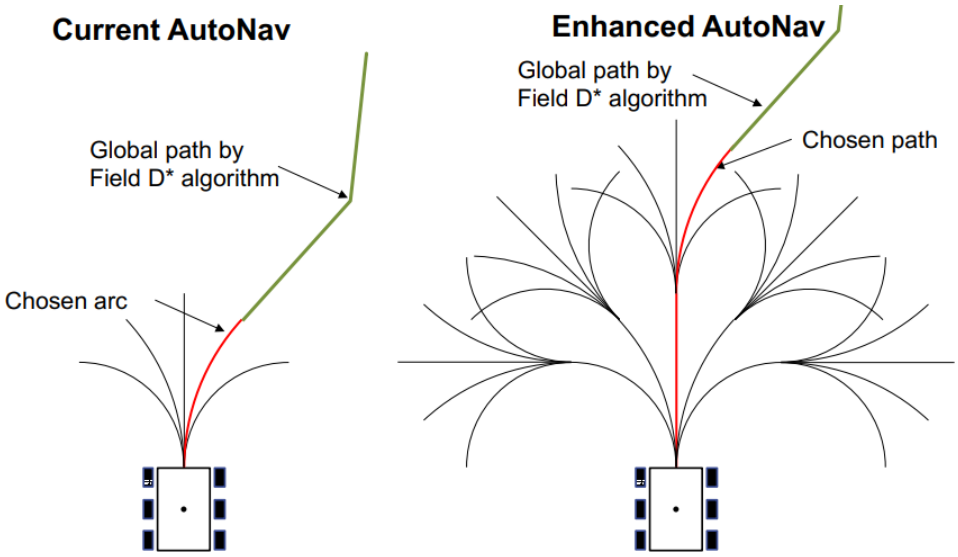

Abbildung 5: Beim MER linke Seite nur mit Kurven und bei Mars 2020 mit Kurvenbaum

(vgl. Jet Propulsion Laboratory, o. J, Enhancement 3: Planning with tree)

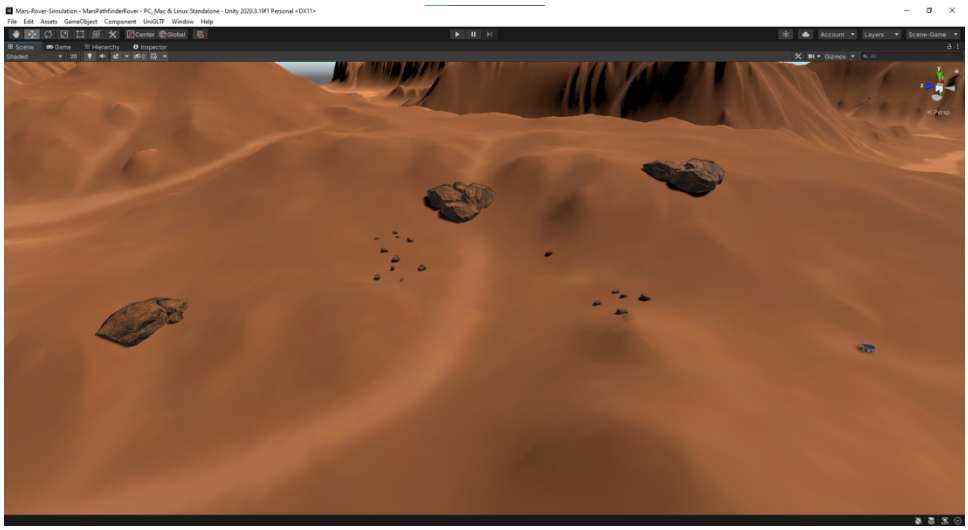

Abbildung 6: Szene der Testumgebung in Unity

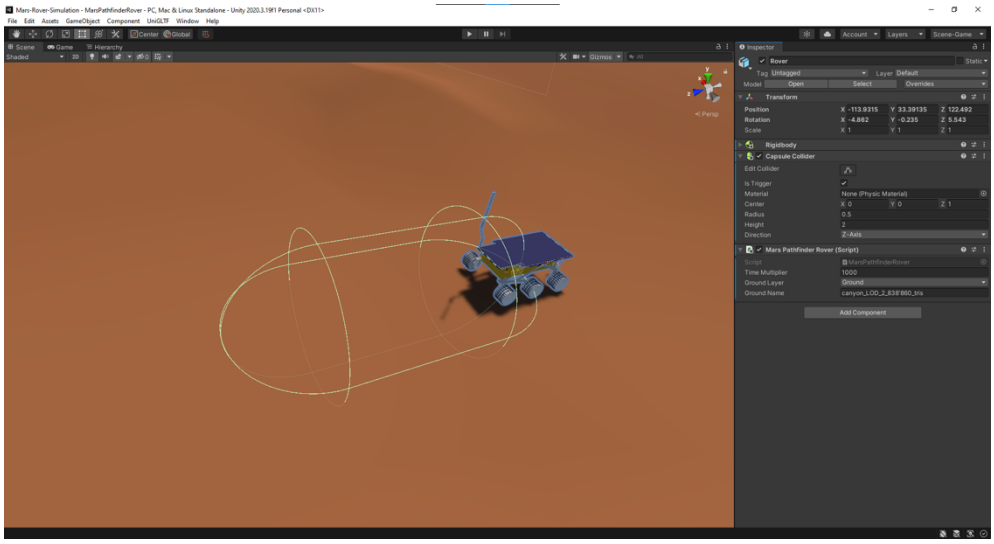

Abbildung 7: Sichtbereich als ein Capsule Collider

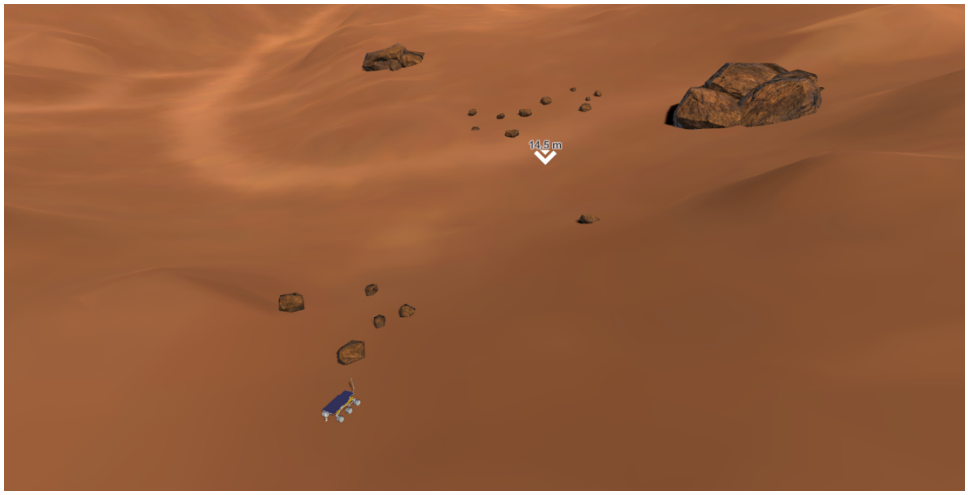

Abbildung 8: Laufende Simulation in der Testumgebung mit Wegpunkt als Pfeil

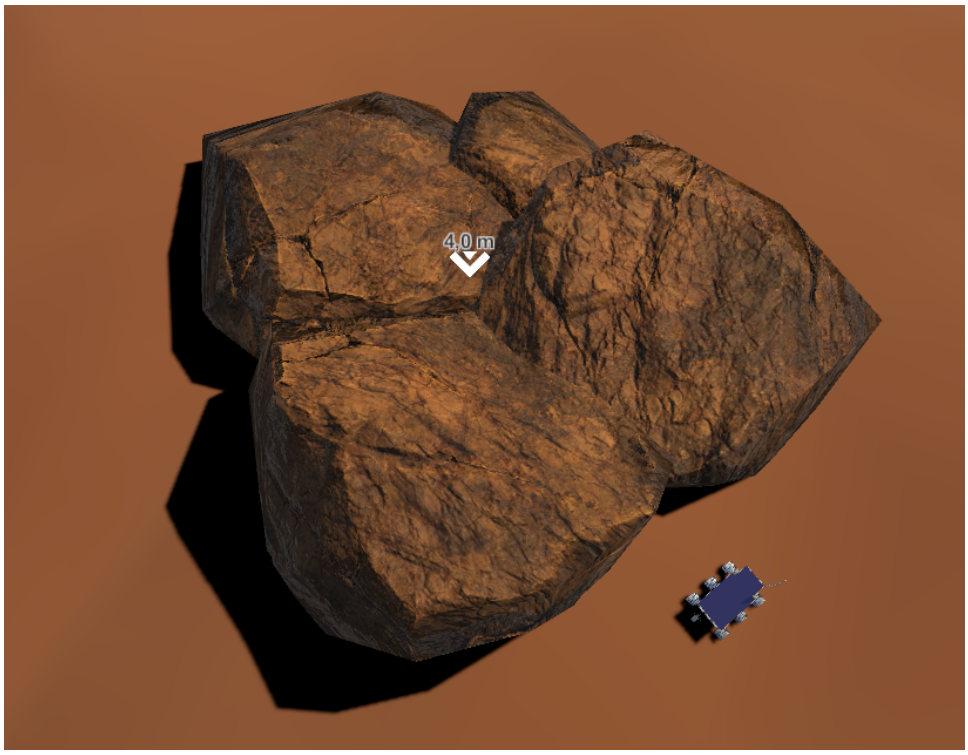

Abbildung 9: Sojourner kreist wiederholt um ein großes Hindernis

Tabelle

Tabelle 1: Evolution von Rover Hardware

(In Anlehnung an Maimone, Mark, 2021, Folie 13)

	Sojourner 1997	MER 2004	MSL 2012	M2020 2021
CPU	80C85	BAE RAD6000	BAE RAD750	BAE RAD750 (x2) FPGA Xilinx Virtex5QV
MHz	2	20	133	133 (x2) + FPGA (22M disp/s)
RAM(Mbytes)	0.56	128	128+512	128 (x2) + 512 (x2)
Non-volatile storage (Mbytes)	0.17	256 flash	4,096 flash	4,096 + 3,072 flash
Image Pairs/Step	1	1-2	4	1
Stereo Pixels processed per step	20	10,000 - 50,000	40,000 - 200,000	FPGA: 240,000 – 1,200,000
AutoNav Pause/Step	?	~120 s	~120 s	< 14 s

Source Code Anhang von MarsPathfinderRover.cs:

```
using System.Collections;
using UnityEngine; // Bibliotheken mit vorgefertigten Methoden

public class MarsPathfinderRover : MonoBehaviour {
  #region Attribute
```

```
public int TimeMultiplier = 100; // Zeitmultiplikator
[SerializeField] private LayerMask GroundLayer;
[SerializeField] private string GroundName;
private const float MovementSpeed = .0067f; // 0,67 cm/s
private const float SegmentLength = .065f; // 6,5 cm
private const float RoverDiskRadius = 1f; // 1 m
private const float ScanningTime = 120; // 120 s
private const float RoverHeightAboveGround = .05f;
private const float MaxTurnAngle = 1;
private const float RotationSpeed = .2f;
private float DistanceToDrive = 0;
private int Hazards = 0; // Anzahl Hindernisse im Sichtfeld
// Wird temporär auf false gesetzt, wenn ein Hindernis im Weg ist
private bool ShouldTurnToWaypoint = true;
#endregion
// Wird zur Initialisierung aufgerufen
private void Start() {
  ParallelToGround(); // Rover parallel zum Boden orientieren
  StartCoroutine(ScanningCoroutine());
}

// Die Kontrollschleife wird wiederholt aufgerufen
private void Update() {
  // Wenn es mehr als 0 Wegpunkte gibt
  if (Waypoint.Instance.Waypoints.Count > 0) {
    // Wenn der ohne Scannen fahren kann
    if (DistanceToDrive > 0 && Hazards == 0) {
      // Wenn der Wegpunkt erreicht wurde
      if (WaypointReached(Waypoint.Instance.Waypoints.Peek())) {
        Waypoint.Instance.Waypoints.Dequeue(); // Wegpunkt entfernen
        return; // Kontrollschleife unterbrechen
      }
      MoveForward(); // Vorwärts fahren
      if (ShouldTurnToWaypoint)
        RotateTowardsWaypoint(Waypoint.Instance.Waypoints.Peek());
      ParallelToGround();
      // Wenn ein Segment abgefahren wurde dann scannen
      if (DistanceToDrive <= 0) {
        StartCoroutine(ScanningCoroutine());
      }
    } else if (Hazards > 0) {
      RotateInPlace();
    }
  }
}

// Simuliert Scanzeit & beendet Scan bei keinen Hindernissen
IEnumerator ScanningCoroutine() {
  yield return new WaitForSeconds(ScanningTime / TimeMultiplier);
  if (Hazards == 0) { NoHazardInView(); }
  else { StartCoroutine(ScanningCoroutine()); }
}
#region Bewegung und Rotation zum Wegpunkt
// Bewegt den Rover nach vorne
private void MoveForward() {
  transform.Translate(Vector3.forward * Time.deltaTime *
    TimeMultiplier * MovementSpeed);
  DistanceToDrive -= Time.deltaTime * TimeMultiplier *
    MovementSpeed;
}
```

```
// Orientiert den Rover parallel zum Boden
private void ParallelToGround() {
  // Checkt den Boden direkt unter dem Rover mit einem Ray
  Ray ray = new Ray(transform.position, Vector3.down);
  if (Physics.Raycast(ray, out RaycastHit raycastHit,
    float.MaxValue, GroundLayer)) {
    transform.position = new Vector3(transform.position.x,
      raycastHit.point.y + RoverHeightAboveGround,
      transform.position.z);
    transform.rotation = Quaternion.FromToRotation(transform.up,
      raycastHit.normal) * transform.rotation; // Parallel zum Boden
  }
}

// Rotiert den Rover zum Wegpunkt
private void RotateTowardsWaypoint(Vector3 waypoint) {
  // Berechnung vom Vektor von der Rover Position zum Wegpunkt
  Vector3 vectorToWaypoint = waypoint - transform.position;

  // Konvertieren von 3D zu 2D Koordinate der Vektoren
  Vector2 vectorToWaypoint2D = new Vector2(vectorToWaypoint.x,
    vectorToWaypoint.z);
  Vector2 forward2D = new Vector2 (transform.forward.x,
    transform.forward.z);

  // Berechnung des Winkels zwischen den Vektoren
  float radians = Mathf.Acos(Vector2.Dot(
    vectorToWaypoint2D.normalized, forward2D.normalized));
  float angle = radians * Mathf.Rad2Deg;

  // Maximale Drehung von MaxTurnAngle
  angle = angle < MaxTurnAngle ? angle : MaxTurnAngle;

  // Beurteilung ob Drehung nach rechts oder links erfolgen soll
  Vector2 right2D=new Vector2(transform.right.x,transform.right.z);
  Vector2 left2D = -right2D;
  if (Mathf.Acos(Vector2.Dot(vectorToWaypoint2D.normalized,
    left2D.normalized))*Mathf.Rad2Deg <
    Mathf.Acos(Vector2.Dot(vectorToWaypoint2D.normalized,
    right2D.normalized)) * Mathf.Rad2Deg)
    { angle = -angle; }

  // Rotation um die y-Achse (Achse nach oben) des Rovers
  transform.Rotate(Vector3.up, angle * Time.deltaTime *
    TimeMultiplier * RotationSpeed);
}

// Gibt true zurück wenn der Wegpunkt erreicht wurde
private bool WaypointReached(Vector3 waypoint) {
  // Wenn die x-Koordinate vom Wegpunkt nicht in der Scheibe ist
  if (waypoint.x <= transform.position.x - RoverDiskRadius ||
    waypoint.x >= transform.position.x + RoverDiskRadius)
    { return false; }
  // Wenn die z-Koordinate vom Wegpunkt nicht in der Scheibe ist
  else if (waypoint.z <= transform.position.z - RoverDiskRadius ||
    waypoint.z >= transform.position.z + RoverDiskRadius)
    { return false; }
  else { return true; }
}
#endregion
#region Hazard Avoidance
```

```
  // Wird aufgerufen, wenn ein Hindernis in das Sichtfeld kommt
  private void OnTriggerEnter(Collider other) {
    if (other.gameObject.name == GroundName) { return; }
    ShouldTurnToWaypoint = false;
    Hazards++;
  }

  // Wird aufgerufen, wenn ein Hindernis das Sichtfeld verlässt
  private void OnTriggerExit(Collider other)
  {
    if (other.gameObject.name == GroundName) { return; }
    Hazards--;
  }

  // Stellt die Distanz ohne Scannen ein
  private void NoHazardInView() {
    DistanceToDrive = SegmentLength;
    ShouldTurnToWaypoint = true;
  }

  // Auf der Stelle nach rechts drehen
  private void RotateInPlace() { transform.Rotate(Vector3.up,
    MaxTurnAngle * Time.deltaTime * TimeMultiplier * RotationSpeed);
}
  #endregion
}
```