TECHNISCHE HOCHSCHULE KÖLN

CAMPUS GUMMERSBACH

Institut für Informatik

Praxisprojektarbeit Informatik

Verwendung von Google Tango für die autonome Roboter-Navigation

vorgelegt von

Jannis Möller

Prüfer: Dr. rer. nat., Dipl.-Inf. Heinrich Klocke

Betreuer: M.Sc. Alex Maier

Juli 2017

Technology
Arts Sciences
TH Köln

Praxisprojektarbeit

Titel: Verwendung von Google Tango für die autonome Roboter-Navigation

Gutachter:

1. Prof. Dr. rer. nat., Dipl.-Inf. Heinrich Klocke
2. M.Sc. Alex Maier

Zusammenfassung: Diese Arbeit stellt die Ergebnisse der praktischen Implementation einer Robotersteuerung auf einem Google Tango Smartphone vor. Sie gibt eine Einführung in Google Tango, sowie das Occupancy-Octree-Framework OctoMap und stellt die Hürden der Implementierung dar. In weiteren Teilen der Arbeit werden der Entwurf des Roboters sowie die Algorithmen zu Navigationsplanung dokumentiert. Außerdem wird ein detaillierter Einblick in den Aufbau und die Funktionsweise der Smartphone-Applikation gewährt. Zuletzt wird noch ein Algorithmus zur Segmentierung der Umgebung kurz vorgestellt der in zukünftigen Projekten auch implementiert werden könnte.

Stichwörter: Google Tango, Roboter, Autonom, Erkundung, OctoMap

Datum: 17. Juli 2017

Inhalt

Einleitung

Autonome Agenten finden heutzutage in allen möglichen Umgebungen und Aufgabenfeldern Anwendung. Die prominentesten liegen z. B. bei Staubsauger- und Rasenmährobotern, autonomen PKWs und LKWs, aber auch bei Anwendungsgebieten, die für den Menschen zu gefährlich sind, wie z. B. im Rettungsbereich die Katastrophenortserkundung.

Die Probleme, die dabei zu lösen sind, umfassen Wahrnehmung und Verständnis der unbekannten Umgebung, Navigation in dieser sowie Interaktion mit ihr.

Google Tango (früher Project Tango) ist ein von Google entwickeltes Projekt, das es Smartphones ermöglichen soll ihre Umgebung besser wahrnehmen und dadurch stärker in die Interaktion mit ihnen einbinden zu können. Hauptaugenmerk ist dabei die Entwicklung von Augmented Reality Anwendungen, aber die zusätzlichen Sensoren und die damit verbundene Software lassen das System auch für Anwendungen in der Robotik reizvoll erscheinen.

In dieser Arbeit sollte die Technologie Google Tango eingesetzt werden um ein Smartphone zur Steuerung eines Roboters mit dem Ziel der autonomen Erforschung der Umgebung zu verwenden. Die dabei gesammelten Ergebnisse, Erkenntnisse und Erfahrungen wurden in diesem Dokument festgehalten.

Gummersbach, 17.07.2017

1 Google Tango

Google Tango ist ein von Google entwickeltes API zur Entwicklung von umgebungsverarbeitenden Android-Applikationen. Es ist für die Programmiersprachen Java, C und die Gameengine Unity verfügbar.

Die öffentliche Dokumentation der APIs findet sich unter https://developers.google.com/tango/apis/overview.

Neben der Dokumentation gibt es von Google auch ein GitHub-Repository mit verschiedenen Beispielanwendungen von Google Tango. Für das Java API sind diese Beispiele unter https://github.com/googlesamples/tango-examples-java zu finden.

Ich habe mich für diese Arbeit für das Java API entschieden, da die Unity Version sehr stark auf Spiele und Simulationen zugeschnitten ist und die C Version für mich einen sehr viel höheren Aufwand in der Entwicklung bedeutet hätte, da ich mit der Entwicklung in C bzw. C++ nur wenig vertraut war. Mit Java habe ich durch mein bisheriges Studium wesentlich mehr Erfahrung.

1.1 Überblick der Konzepte

Ein großer Teil der Projektarbeit bestand darin, sich mit den Konzepten und Funktionen des APIs vertraut zu machen. Im nächsten Abschnitt werden die drei Kernkonzepte von Google Tango, wie sie in [1] genannt werden, vorgestellt.

1.1.1 Motion Tracking

Das Motion-Tracking erlaubt es dem Smartphone seine Pose im Raum, also seine Position und seine Orientierung wahrzunehmen und zu verfolgen. Somit kann in Augmented Reality Anwendungen z. B. eine virtuelle Kamera gesteuert werden. Der Nutzer muss dann für die Bewegung im virtuellen Raum genau die gleiche Distanz auch in der Realität zurücklegen.

Das Motion-Tracking alleine gewährt dem Smartphone allerdings noch kein Wissen über die Beschaffenheit der Umgebung.

Die Implementierung erfolgt über einen nicht näher spezifizierten Visual-Inertial-Odometrie Algorithmus. Das heißt, das Smartphone nutzt sowohl die Daten die es von der IMU (Inertiale Messeinheit) bekommt als auch den Image-Stream der Weitwinkel-Farbkamera. Auf diesen Bildern wird ein Feature Tracking durchgeführt und so die Änderung der Position im dreidimensionalen Raum eingeschätzt.

Bei der einfachen Form des Motion Trackings kann das Smartphone eine bereits besuchte Umgebung nicht wiedererkennen und sich so auch nicht, wenn nötig korrigieren (Relokalisierung und Loop-Closure), sondern sammelt über Zeit einen immer größer werdenden Fehler in der geschätzten Position. Die Lösung dafür ist das nächste Konzept.

Für weitere Informationen zu Motion Tracking ist die offizielle Dokumentation unter https://developers.google.com/tango/overview/motion-tracking zu finden.

1.1.2 Area Learning

Das Konzept des Area Learnings baut stark auf dem des Motion Trackings auf. Es bietet neben dem normalen Tracking die Funktionalität bereits besuchtes Gebiet wiederzuerkennen. Damit ist es in der Lage den über die Zeit gesammelten Fehler, auch Drift genannt, zu korrigieren und so die Genauigkeit der Lokalisierung zu verbessern und zu stabilisieren. Außerdem besteht die Möglichkeit ein zuvor gelerntes Gebiet zu laden und sich darin zu lokalisieren. Diese Technik ist vor allem in vor dem Anwendungszeitpunkt bekannter Umgebung von Nutzen, in denen sich der User lokalisieren soll. Ein möglicher Anwendungszweck ist die Verwendung für die Indoor-Navigation in Gebäuden.

In welcher Form genau diese gelernten Umgebungen abgespeichert werden, ist nicht weiter spezifiziert. Es muss sich allerdings um eine sehr speicher-effiziente Darstellung handeln, da die Dateien nicht besonders groß sind.

Bei dem Area Learning gibt es vor allem zwei verschiedene Einstellungen zu beachten:

1. Soll die aktuelle Umgebung gelernt werden
2. Soll sich gegen eine gespeicherte Umgebung lokalisiert werden.

Wenn nur die erste Option aktiv ist, wird die aktuelle Umgebung gelernt und kann, solange kein Tracking Verlust aufgetreten ist, abgespeichert werden.

Ist nur die zweite Option aktiv wird die Anwendung versuchen sich in der bekannten Umgebung zu lokalisieren, diese allerdings nicht weiter lernen, auch nicht, wenn der gelernte Bereich verlassen wird.

Wenn beide Option zusammen aktiv sind wird zunächst versucht sich gegen die gespeicherte Umgebung zu lokalisieren und wenn das erfolgreich war, wird die Umgebungsbeschreibung mit weiteren Details ergänzt. In der aktuellen Version des Tango SDKs gibt es leider einige Probleme in diesem für die Robotik eigentlich gut geeigneten Modes, mehr dazu unter Abschnitt 1.3.

Zwei gelernte und gespeicherte Umgebungen nachträglich zu einer einzigen zusammen zu führen ist leider noch nicht möglich.

Für weitere Informationen zum Area Learning ist die offizielle Dokumentation unter https://developers.google.com/tango/overview/area-learning zu finden.

1.1.3 Depth Perception

Die Wahrnehmung der Tiefe erlaubt es der Anwendung die Beschaffenheit der Umgebung wahrzunehmen. Für Augmented Reality Anwendungen bedeutet dies, dass virtuelle Objekte realistisch in der Umgebung platziert werden und sogar durch reale Gegenstände verdeckt dargestellt werden können. Außerdem bietet sie die Möglichkeit die Umgebung zu vermessen und nicht nur wie beim Motion Tracking die zurückgelegte Strecke des Gerätes selbst.

Für die Navigationsplanung in der Robotik ist dieses „Konzept“ besonders interessant, da hiermit die Umgebung kartografiert werden kann.

Die Wahrnehmung der Tiefe wird von dem Tango SDK ausschließlich in Form einer Point Cloud in dem lokalen Koordinatensystem der Tiefen-Kamera angeboten, es besteht aber die Möglichkeit mithilfe der Point Cloud nachträglich eine Art Tiefenbild zu berechnen.

Für weitere Informationen zur Depth Perception ist die offizielle Dokumentation unter https://developers.google.com/tango/overview/depth-perception zu finden.

1.2 Eingesetzte Hardware

Die Software ist stark an Hardware-Voraussetzungen geknüpft, wodurch zum Zeitpunkt der Projektarbeit nur ein auf dem Markt verfügbares Smartphone (Phablet) diese Technik unterstützt:

Das Lenovo Phab 2 Pro
(seit Juni 2017 auch das Asus ZenFone AR, allerdings mit sehr geringer Verfügbarkeit)

Zusätzlich zu der heutzutage in fast jedem Smartphone verbauten Farb-Kamera und einer inertialen Messeinheit (inertial measurement unit, IMU) bieten Tango-fähige Smartphones folgende zusätzliche Komponenten:

1.2.1 Weitwinkel-Kamera

Für das robuste visuelle Tracking der Bewegung ist es erforderlich zwischen den verschiedenen Frames der Kamera einen möglichst großen Überlappungsbereich zu haben um möglichst viele visuelle Features einfangen zu können. Daher wird für das Motion Tracking eine Fisheye-Kamera mit einem großen Aufnahmewinkel vorausgesetzt.

1.2.2 Tiefenbild-Kamera

Um dem Smartphone die Möglichkeit zu geben nicht nur Farbe, sondern auch Tiefe wahrzunehmen wird in allen Tango-fähigen Geräten eine Tiefen-Kamera verbaut. Im Falle des Phab 2 Pro ist es ein Sensor der „REAL3™“-Serie des Herstellers Infineon, der mit 224 x 172 Pixeln bei 5 Frames pro Sekunde auflöst. Der Sensor funktioniert nach dem Prinzip der „Time of Flight“-Kameras, d. h. ein Infrarotblitz wird erzeugt, von der Umgebung zurückgeworfen und jeder Pixel löst einzeln aus, sobald das Licht ihn erreicht. Durch das Messen der Zeit, die zwischen dem Aussenden des Lichtes und dem Eintreffen in den bestimmten Pixel vergeht, kann mithilfe der Konstante der Lichtgeschwindigkeit die Entfernung berechnet werden. Es handelt sich hierbei also um das gleiche Prinzip wie auch bei Sonar, Radar und Lidar mit dem Unterschied, dass nicht nur eine Messung geradeaus vor dem Sensor, sondern ca. 38k parallel verarbeitet werden. Die genauen Spezifikationen zu dem Sensor können unter [2] nachgelesen werden.

1.3 Problematiken bei Verwendung in der Robotik

Der ursprüngliche und eigentlicher Einsatzzweck Google Tango liegt in der Entwicklung von Augmented Reality Applikationen und nicht in der Robotik, daher ist das API auch nicht darauf zugeschnitten. An manchen Stellen wird das besonders deutlich:

Motion Tracking:

Das System ist relativ stark von guten Lichtverhältnissen abhängig, in Low Light Szenarien kommt es häufig zu starkem Positions-Drift und Tracking-Verlust. Außerdem ist das Tracking auf Umgebungen angewiesen, die viele gut unterscheidbare visuelle Features beinhalten. So kam es auch bei Tests in den kahlen Fluren des Gebäudes der Technischen Hochschule häufig zu Problemen.

Durch die geringe Höhe, in der das Smartphone im späteren Anwendungsszenario angebracht ist, nimmt der meist Feature-arme Boden einen großen Teil des Sichtfelds ein. Außerdem erschweren Vibrationen die durch die Bewegung des Bots entstehen das Tracking weiter.

Area Learning:

Wie bereits in Abschnitt 1.1.2 erwähnt funktioniert der Area Learning Mode mit beiden Optionen (Learning und Localizing) nicht zuverlässig. Tatsächlich funktioniert das Lokalisieren in diesem Modus so schlecht, dass er für die Anwendung in diesem Projekt ungeeignet ist. In Tests musste ich teilweise einige Minuten warten, bis das Gerät sich lokalisiert hatte und konnte dann nur sehr vorsichtige Bewegungen machen, ohne dass das Tracking verloren ging.

Intern wird für die Relokalisierung bei gleichzeitigem Lernen scheinbar eine andere Pipeline verwendet als bei der Relokalisierung ohne weiteres Lernen. Aus welchem Grund das so ist, ist nicht ersichtlich und auch nicht dokumentiert, aber vielleicht ist das Learning besonders rechenintensiv und kann daher nur mit einer geringen Framerate durchgeführt werden. Für zukünftige Updates lässt sich hoffen, dass diese Probleme von Google behoben werden.

Aus diesen Gründen ist dieser Modus und damit das Speichern der sogenannten Area Description Files momentan nicht implementiert. Möchte man es jedoch implementieren muss man beachten, dass es grundsätzlich nur möglich ist, während das Gerät sich gegen die bisher erstellte Umgebung lokalisiert hat. Um die Area Description File an einem beliebigen Ort speichern zu können werden zwei Schritte benötigt:

1. Die aktuelle Area Descripting muss gespeichert werden, der Ort ist dabei nicht frei wählbar. Man erhält aber einen Universally Unique Identifier (UUID) der ADF.
2. Die gespeicherte ADF muss an den gewünschten Ort exportiert werden. Als Parameter nimmt diese Funktion den UUID.

Um eine bestehende exportierte ADF laden zu können werden entsprechend die umgekehrten Schritte benötigt.

Außer diesem Area Learning Modus gibt es aber noch einen weiteren experimentellen Driftkorrektur Modus. Dieser verwendet eine identische Tracking-Technik wie das Area Learning, erstellt jedoch eine dichtere Umgebungsbeschreibung, und ist daher etwas robuster gegenüber Tracking-Verlust. Leider lässt sich die dabei erstellte Umgebungsrepräsentation nicht speichern.

Da dies aktuell der beste Kompromiss gegenüber dem starken Drift im Motion Tracking Modus ist, ist es vorerst der als Standard ausgewählte Modus.

Bei allen Tracking Modi ist es wichtig das Gerät am Anfang eine kurze Zeit lang nach dem Verbinden der Tango Services still zu halten, damit die inertialen Sensoren richtig konfiguriert werden können, da sonst für einige Sekunden ein starker Drift der Position auftreten kann.

Tiefenwahrnehmung:

Durch die auf Infrarotlicht basierende Technik ist die Tiefenwahrnehmung extrem anfällig gegenüber starken externen Infrarotlichtquellen, welche in der Realität vor allem direkter und indirekter Sonnenschein sind. Teilweise treten auch in Gebäuden durch einfallendes Tageslicht Artefakte in der Tiefenwahrnehmung auf.

Außerdem besteht das Problem, dass die als Point Clouds vorliegenden Messungen einem recht starken Rauschen unterliegen. Dadurch, dass keine direkte Möglichkeit besteht, an das Bild der Tiefenkamera zu gelangen, ist es auch unnötig schwer die Messungen zusätzlich zu filtern.

Möchte man dies tun gibt es die Möglichkeit über die statische Methode „`upsampleImageBilateral`“ der Klasse TangoSupport durch Übergabe einer Pointcloud und dem dazugehörigen Farbbild-ImageBuffer sowie zwei weiterer Parameter ein auf die Bildebene projiziertes und mit einem bilateralen Filter gefiltertes Tiefenbild zu erhalten. Den zu der PointCloud passenden Farbbild-ImageBuffer muss man dazu selbst über einen Listener holen und zwischenspeichern.

Dieses Tiefenbild ist vor allem für Augmented Reality Anwendungen gedacht, die Berechnungen im Screenspace durchführen wollen. Daher gibt es auch keine passende Methode, um dieses Bild wieder in eine PointCloud zu transformieren und so musste die Funktion manuell implementiert werden. Bei beiden Transformationen geht Präzision in der Messung verloren.

In Tests ergab sich allerdings, dass diese Filter-Methode mehr Artefakte verursachte als sie filterte. Das Rauschen auf Flächen war zwar eliminiert, aber die Anzahl der fehlerhaften Messungen bei denen Punkte direkt vor der Kamera gemessen wurden erhöhte sich drastisch. „Abbruch-Kanten“ von Objekten hatten auf einmal eine Verbindung zu den im Hintergrund liegenden Flächen – ein Ergebnis der Interpoleration, aber keineswegs erwünscht.

Weiterhin war die Performance durch das Upscaling auf die Größe des Farbbildes (1920x1080) sehr schlecht (ca. 300 ms pro PointCloud), sodass das Filtern nicht nur eine Latenz zwischen eigentlicher Messung und Einfügen der Messung in das Modell der Umgebung hinzugefügt hätte, sondern tatsächlich die Frequenz der PointCloud Aufnahmen von 5 FPS nicht gehalten werden könnte.

2 Repräsentation der physikalischen Welt

Die Repräsentation der physikalischen Welt bildet die Grundlage für die Verarbeitung der Sensor Messungen sowie die Navigations-Planung. Daher werden an sie viele verschiedene Anforderungen gestellt:

- Die Umgebung sollte möglichst präzise dargestellt werden können
- Messungen sollten mit möglichst geringen Rechenaufwand eingefügt werden können
- Da der verfügbare Speicher auf dem Phab 2 Pro mit 2 GB zwar nicht klein, aber auch nicht übermäßig groß bemessen ist, sollte möglichst wenig Speicher benötigt werden
- Operationen zur Navigation bzw. Kollisionsvermeidung sollten darauf ausgeführt werden können.

2.1 OctoMap

Die Wahl fiel für dieses Projekt auf das in C++ geschriebene probabilistische Octree Framework Octomap. Der Source Code, ein wissenschaftliches Paper sowie auch die Dokumentation sind unter https://octomap.github.io/ zu finden.

2.1.1 Konzept und Funktionsweise

OctoMap bietet für viele verschiedene OcTree-Anwendungszwecke eine gute Basis-Implementation, wurde aber besonders für die Repräsentation von Raum konzipiert.

a) Probabilistische Repräsentation

Die Klasse OccupancyOcTreeBase und damit alle abgeleiteten Klassen basieren auf dem Modell einer probabilistischen dreidimensionalen Gridmap zur räumlichen Darstellung von freien und besetzten Volumen. Dieses Prinzip ermöglicht es sowohl dynamische Änderungen der Umgebung zu unterstützen als auch Unsicherheiten in den Sensormessungen auszugleichen. Die Wahrscheinlichkeiten werden dabei in logarithmischer Form gespeichert, was den Vorteil hat, dass Updates auf simple Additionen reduziert und somit sehr effizient durchgeführt werden können [3].

b) Speicherung als Octree

Gegenüber gewöhnlichen Gridmaps, die für jede Zelle Speicher benötigen, ob sie besetzt ist oder nicht, bieten Sparse Octrees eine enorme Speicher-Ersparnis gegen einen geringen Nachteil in der Zugriffszeit.

Hat ein Knoten acht gleiche Kinder, das heißt Kinder mit den gleichen stabilen logarithmischen Odds, können diese in dem Vaterknoten zusammengefasst und gelöscht werden. Genau so wird auch unbekannter Raum zusammengefasst, sodass dieser überhaupt keinen Speicherplatz einnimmt.

Die Zugriffszeit auf einen zufälligen Knoten ist durch die Begrenzung der Tiefe des Baumes in diesem Framework auf 16 immer noch konstant und liegt bei $O(\log(16))$.

Die Begrenzung der Tiefe ermöglicht ein sehr effizientes System zur Adressierung der einzelnen Knoten über Schlüssel (OcTreeKey). Daher ist eine Änderung der maximalen Tiefe nicht ohne größeren Aufwand möglich, im Gegenzug wird aber das Errechnen von durch Rays betroffene Knoten vereinfacht, was wiederum das Einfügen neuer Messungen effizienter macht. Mit dieser Begrenzung sind immer noch $2^{16^3} = 65535^3$ verschiedene Zellen darstellbar.

Ein weiterer Vorteil von Octrees ist es, dass Anfragen auf eine bestimmte Tiefe begrenzt und so beispielsweise sehr effiziente Visualisierungen implementiert werden können. Dafür müssen bei dem Update der log-Odds eines Knotens stets auch alle Elternknoten aktualisiert werden, um deren Kinder korrekt zusammenzufassen.

Zusätzlich zu dem log-Odds Wert wird in jedem Knoten ein Pointer zu einem Array der Kinder gespeichert. Aus diesen Informationen können die drei verschiedenen Zustände gelesen werden:

1. Ist der log-Odds Wert über oder unter einem bestimmten Threshhold ist das Volumen besetzt oder frei
2. Zeigt der Zeiger des Kind-Arrays auf NULL so ist der Knoten ein Blattknoten
3. Zeigt ein Zeiger in dem Kind-Array auf NULL, so ist das entsprechende Teilvolumen unbekannter Raum.

c) Einfügen einer Pointcloud-Messung

Bevor eine Pointcloud von Google Tango in den Octree eingefügt werden kann, müssen die Koordinaten der Punkte erst transformiert werden. Sie befinden sich grundsätzlich immer im lokalen Koordinatensystem der Tiefenkamera. Google Tango bietet für die Transformation einer Pointcloud eine geeignete Hilfsmethode, die allerdings nicht besonders performant implementiert ist. Der einzige für die eigene Implementation nötige Schritt ist eine einfache Matrixmultiplikation mit der Transformationsmatrix, die man von Google Tango erhält. Aktuell ist diese Transformation in C++ implementiert, noch effizienter könnte sie aber implementiert werden, wenn speziell für die Same Instruction Multiple Data (SIMD) Verarbeitung konzipierte Prozessorteile bzw. Befehlssätze verwendet würden. Auf Android gibt es mit NEON eine Möglichkeit das Ganze auf der CPU und mit RenderScript auf der GPU umzusetzen. Da die Transformation in der eigenen C++ Implementation aber nicht besonders viel Rechenzeit beansprucht wurde davon abgesehen.

Das Einfügen einer Pointcloud selbst wird in zwei Schritte unterteilt. Zunächst werden im ersten Schritt die Schlüssel aller betroffenen Knoten berechnet und dann im zweiten Schritt diese Schlüssel iteriert und das Update auf den entsprechenden Knoten ausgeführt. Das Updaten der Knoten ist dabei wiederum aufgeteilt nach Knoten, die in dieser Messung freiem Raum entsprechen und Knoten, die besetztem Raum entsprechen.

Die Berechnung der Schlüssel der besetzten Knoten ist sehr simpel, da sie sich direkt aus den Punkten der Pointcloud ergeben. Die Schlüssel werden lediglich in eine Hashmap eingefügt, um Doppelungen beim Updaten zu vermeiden. Die Berechnung der Schlüssel der

Knoten, die in dieser Messung dem freien Raum entsprechen ist etwas komplexer, sie müssen erst über eine Art Raytracing berechnet werden. Dazu wird für jeden Punkt in der Point Cloud ein Ray von dem Ursprung der Messung, also der Sensor Position zu dem jeweiligen Punkt erzeugt. Dieser Ray wird dann rasterisiert und die extrahierten Schlüssel in eine zweite Hashmap eingefügt.

Eine Option für bessere Performance beim Einfügen ist die Diskretisierung der Punkte der Pointcloud. Sie basiert auf der Annahme, dass die betroffenen Zellen zweier Rays sich nur gering voneinander unterscheiden, wenn Start und Endpunkt der Rays jeweils in den gleichen Zellen liegen. Daher können Punkte der Point Cloud die den gleichen Schlüssel wie ein schon untersuchter anderer Punkt haben ignoriert werden.

OctoMap bietet über die Kompilieroptionen die Unterstützung des Multithreading Frameworks OpenMP. Dieses Framework basiert vor allem darauf Schleifendurchläufe auf mehrere Threads aufzuteilen und zu parallelisieren. Es kommt z. B. in der Methode zur Berechnung der Schlüssel der freien Knoten zum Einsatz, da das Rasterisieren der Rays problemlos parallel laufen kann. Nur für das Einfügen der Schlüssel in die HashMap müssen die Threads synchronisiert werden. Der Einsatz dieser Parallelisierung ist aber nicht für alle Schleifen geeignet, da ein gewisser Overhead mit der Erzeugung der Threads und dem Verteilen der Arbeit verbunden ist.

d) OctoVis

OctoMap bietet von sich aus für die im Framework implementierten Octree-Klassen das Visualisierungstool OctoVis. Es verwendet OpenGL, um die Räume der Knoten des Baumes als Würfel darzustellen. Mit der Taste F kann der freie Raum als transparente grüne Würfel dargestellt werden und mit der Taste S können Gitternetz-Linien ein und ausgeblendet werden, die die unterschiedlich tiefen Detailstufen der Knoten gut visualisieren.

Außerdem gibt es die Möglichkeit die Visualisierung auf eine bestimmte Tiefe zu begrenzen. OctoVis nutzt dabei die Eigenschaft des Octrees die in 2.1.1 b) erklärt wurde.

Durch die Speicherung weiterer Informationen in den Knoten (siehe nächster Abschnitt) ist OctoVis leider nicht mehr kompatibel mit meiner erweiterten Octree-Klasse.

2.1.2 Anpassungen

a) Integration

Die Integration als Bibliothek auf dem von OctoMap vorgesehenen Wege durch ausführen der CMAKE Skripte ist in Kombination mit Android Studio nicht gelungen. Das liegt daran, dass auf diese Weise nicht verschiedene Versionen der Bibliothek für die unterschiedlichen Android Prozessor Architekturen kompiliert wurden und sie deswegen auch nicht in die Android Installationsdatei (APK) gepackt wurden.

Eine weitere Möglichkeit bestand darin einen bereits vorhandenen Java Wrapper zu verwenden. Joctomap [4] war der einzige, der sich finden ließ, scheiterte aber leider schon bei der Maven Installation, da eine Dependency fehlte und nicht mehr auf dem entsprechenden Server verfügbar war.

Die Lösung dieser Probleme brachte nach Einlesen in CMAKE die manuelle Modifikation der CMAKE Skripte. Es handelt sich nun um stark vereinfachte Versionen der Originalskripte.

Im Nachhinein erwies sich das als gute Entscheidung, da dies die Erweiterung erleichterte.

b) Erweiterung

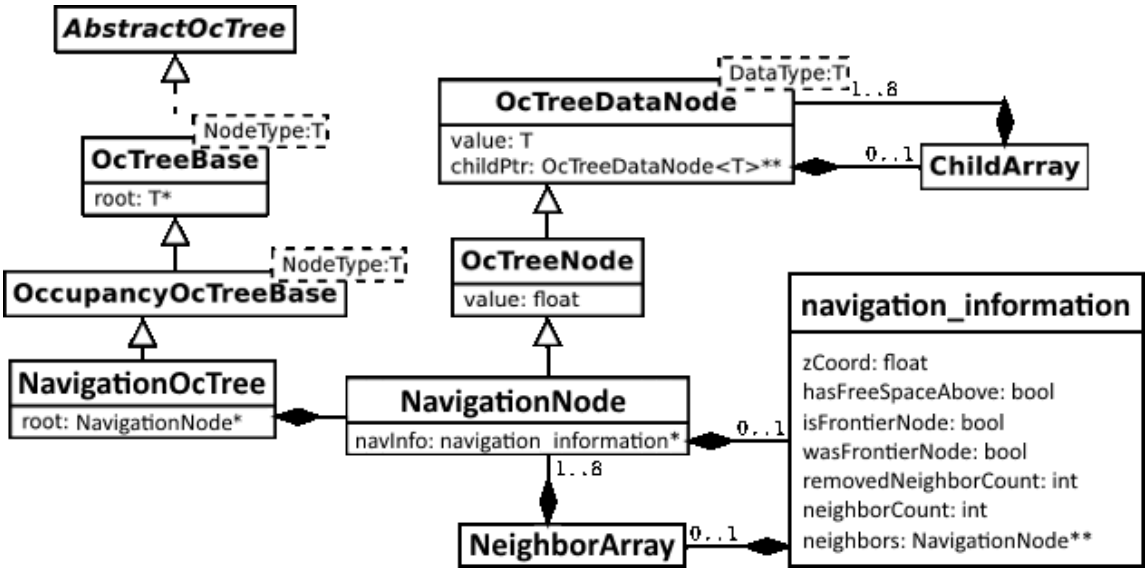

Abbildung 1: Überblick der Vererbung und Komposition

Die Klasse NavigationOcTree erbt von der Kasse OccupancyOcTreeBase und erweitert und überschreibt vor allem die Methoden dieser Klasse.

Die Klasse NavigationNode erweitert die Klasse OcTreeNode um einige Informationen die zu dem Struct „navigation_information“ zusammengefasst sind und nur dann Speicher reservieren, wenn es sich um einen besetzten Knoten handelt.

Durch die erweiterten Zusatzinformationen ergeben sich auch für den Status der Knoten neue Möglichkeiten der Interpretation:

- Pointer der navigation_information zeigt auf NULL: Knoten ist freier Raum
- zCoord: Ermöglicht genaueren Höhenvergleich zwischen Nachbarn, als es die Auflösung des OcTrees zulässt
- hasFreeSpaceAbove: Knoten hat über sich genug freien Platz für den Bot
- isFrontierNode: Knoten liegt an der Grenze zu unbekannten Raum
- wasFrontierNode und removedNeighborCount werden benötigt, um einen zellulären Automaten simulieren zu können
- neighborCount != 0: Knoten hat mindestens einen weiteren erreichbaren Nachbarn, der auch genug freien Platz über sich hat. Wenn diese Bedingungen nach einem Update der Navigierbarkeit und vor dem Einfügen einer weiteren Messung gegeben ist, kann sich der Bot kollisionsfrei auf dieser Position befinden.

Eine genauere Beschreibung des Algorithmus zur Findung navigierbare Knoten wird in Abschnitt 3.2.1 gegeben.

Um die Navigationsplanung zu vereinfachen werden besetzte Zellen, die gleiche Kinder eines Vaterknotens sind, nicht mehr zusammengefasst. Somit muss nicht mit verschiedenen Größen von Zellen gerechnet werden und die Erzeugung eines Adjazenzgrafen mit einer festen Anzahl von Nachbarn je Knoten ist möglich. Der Nachteil in der Speichereffizienz ist zu verkraften, da die Sensormessungen in der Regel sowieso nur Oberflächen und keine Volumina umfassen und die Ersparnisse beim Zusammenfassen besetzter Knoten daher nicht allzu groß ausfallen.

Iteration des Octrees

Da das Iterieren des Octrees für verschiedene Teilfunktionen in der Anwendung wichtig und performancekritisch war, habe ich eine eigene Implementation geschrieben. OctoMap selbst bietet eine syntaktisch sehr schöne und simple Möglichkeit über einen Iterator, dazu muss aber zuerst eine Datenstruktur mit Zeigern zu allen Knoten des Baumes gefüllt werden. Meine Implementation führt die Funktionalität direkt bei dem rekursiven Durchlaufen des Baumes aus. In Tests der Funktionalität zum Hinzufügen der Knotenkoordinaten zu einem Buffer für das spätere Rendering in OpenGL konnte so eine zeitliche Verbesserung in der Größenordnung des Zwei- bis Zehnfachen festgestellt werden.

Das Iterieren eines bestimmten Teilbereichs des Baumes, der durch eine Boundingbox spezifiziert wird, konnte ich auf die gleiche Weise effizienter implementieren. Zusätzlich werden dabei aber die Grenzen des Volumens das der aktuelle Knoten umfasst überprüft. Da diese Überprüfung relativ rechenintensiv ist, wird sie nur ausgeführt, wenn der Knoten nur teilweise mit der Bounding Box überlappt. Ist ein Knoten komplett in der bounding box eingeschlossen so können alle Kinder rekursiv ohne Überprüfung iteriert werden. Dieser besondere Fall wird in der Implementation in OctoMap nicht beachtet.

Finden eines Nachbarknotens

Um das Problem der Feststellung der Navigierbarkeit der Map zu lösen habe ich einen sehr effizienten Algorithmus zum Finden eines räumlich nahen Nachbarknotens in implementiert.

Er basiert darauf das eine Zelle in einem Octree eindeutig durch ein Key beschrieben werden kann, der die Position so codiert, dass die binäre Ziffer einer bestimmten Tiefe an Stelle n in dem Key bei einer 0 der unteren Zelle und bei einer 1 der oberen Zelle entspricht (bzw. links/rechts und vorne/hinten). Diese Keys kann man dann dazu verwenden, einen bestimmten Knoten in dem Baum zu adressieren.

Möchte man nun einen Knoten weiter in eine bestimmte Richtung springen addiert bzw. subtrahiert man 1 um die Anzahl der Tiefe bitweise nach links verschoben zu der gewünschten Achse und erhält so den Schlüssel der neuen Zelle.

Um nicht jedes Mal die 16 Schritte von der Wurzel bis zu einem Blattknoten laufen zu müssen wird ein Stack mit den traversierten Knoten aufgebaut. Mit dem bitweisen XOR können nun die Differenzen zwischen dem ursprünglichen Schlüssel und dem Schlüssel der neuen Zelle berechnet werden. Anschließend müssen die führenden Nullen gezählt werden, um so die Stelle in dem Stack bestimmen zu können an der beide Zellen einen gemeinsamen Vater haben und ab welcher der Stack zu dem Nachbarknoten neu aufgebaut werden muss. Die Implementierung des Stacks muss daher auch die Möglichkeit bieten über den Index auf die Elemente zugreifen zu können. In dem Sinne handelt es sich nicht mehr um einen reinen Stack und umgesetzt wurde es als „array“ der Standardbibliothek, einem C++ Wrapper eines nativen Arrays, der zusätzlich Indexüberprüfung bietet.

Problematik des Speicherns

Die Basis-Implementierung aller Octrees bietet die Möglichkeit die Knoten des Baumes rekursiv zu speichern. Dazu werden alle Knoten mittels einer Tiefe-zuerst-Iteration besucht und jeder Knoten schreibt dabei seine Informationen sequenziell in den Outstream. Ein Problem ergibt sich dadurch bei der Speicherung der für die Navigation wichtigen Nachbarn des Knotens, denn diese stellen eine zirkuläre Abhängigkeit dar. Ohne das Speichersystem des Octrees komplett zu überarbeiten, den Knoten IDs zu geben und zusätzlich eine Adjazens-Darstellung mit diesen IDs zu speichern, ist es also nicht möglich die kompletten Informationen einer NavigationNode zu speichern. Daher wird aktuell davon abgesehen und nach jedem Laden eines NavigationOcTrees einmal für den gesamten Baum die Navigierbarkeit und die Nachbarschaften der Knoten neu bestimmt.

2.2 Alternativen

Wegen der teilweise problematischen Performance beim Einfügen der Point Clouds habe ich mich während des Projektes immer wieder nach Alternativen umgesehen.

2.2.1 Ringpuffer

Unter anderem in [5] vorgestellt bieten dreidimensionale Ringpuffer eine Alternative zu Octrees die vor allem den Zeitaufwand für das bei Octrees teure Einfügen neuer Messungen senken kann. Für diese Anwendung sind sie jedoch nicht geeignet, da der extrem stark ansteigende Speicherbedarf es nicht zulässt größere Gebiete bei einer akzeptablen Auflösung zu erfassen und somit das Mapping unmöglich macht. Sie sind vor allem für Anwendungszwecke geeignet, bei der nur die unmittelbare Umgebung entscheidend und Geschwindigkeit in der Verarbeitung sehr wichtig ist.

Ein weiteres Problem ist der bei einigen Operationen (z. B. dem Traversieren aller Knoten) auch vergleichsweise hohe Rechenaufwand, da gleich belegte Zellen anders, als bei Sparse Octrees nicht zusammengefasst werden können.

Ein Ansatz um diesen Problemen zu entgehen wäre es ein Hybrid aus Octree und Ringbuffer zu erstellen. Der Ringpuffer würde dann nur die Umgebung in Sensorreichweite abdecken. Bei diesem Ansatz tritt allerdings das Problem der Synchronisierung zwischen Octree

und Puffer auf. Außerdem ist die Navigationsplanung bei einer solchen Datenstruktur wesentlich komplexer.

2.2.2 3D Distance Maps

3D Distance Maps (auch Signed Distance Fields) sind deine Darstellung die auch auf einem Raster basiert, allerdings für jeden Punkt im Raster nur die kürzeste Entfernung zu dem nächsten Hindernis speichert. So können sehr effizient Kollisionsabfragen durchgeführt werden. Wie in einem von Armin Hornung gehaltenen Talk [6] erwähnt, wurde für OctoMap auch an einem solchen System gearbeitet. Der Stand dieses Teilprojekts ist jedoch unklar, da keinerlei Dokumentation gefunden werden konnte.

Für die Weiterführung dieses Projektes wäre dies aber ein möglicher Bereich weiterer Forschung.

3 Navigation

3.1 Mobile Plattform

3.1.1 Design

Für die Realisierung einer mobilen Plattform wurde das Lego-EV3 System gewählt, da ich damit durch den Vorgänger NXT aus dem Modul Künstliche Intelligenz größtenteils vertraut war.

a) Anforderungen

Bei dem Entwerfen eines geeigneten Bots waren einige Anforderungen zu beachten:

Befestigung des Phablets:

- Das Phab2Pro ist relativ groß und schwer, daher muss es sicher befestigt werden
- Die Kameras dürfen nicht verdeckt sein und das weite Sichtfeld der Fisheye-Kamera sollte möglichst frei sein
- Das Display sollte möglichst frei bleiben, um die Bedienung im befestigten Zustand zu ermöglichen

Bewegungseigenschaften:

- Der Bot soll mindestens geradeaus fahren und sich auf der Stelle drehen können. Am besten ist dafür ein Differential-Antrieb geeignet

Weitere Anforderungen:

- Das Phablet sollte möglichst gut vor Kollisionen geschützt sein. Das bedingt, dass der Bot ein gutes Verhältnis von Grundfläche zu Schwerpunkthöhe hat und das Phablet nicht überstehend befestigt ist

- Der Bot und damit das Phablet dürfen nicht zu stark vibrieren, da die Kamera sonst nur unscharfe Bilder aufnehmen kann und die Genauigkeit des Trackings leidet
- Der Brick muss einfach bedienbar bleiben
- Der Akku des Bots muss austauschbar bleiben

b) Designfindungsprozess:

Testdesign Nummer 1:

Nachgebaut wurde das Kettenfahrzeug "Tank" aus dem EV3 Expansion Set.

Beobachtungen:

- Schwerpunkt sehr niedrig, seitliche Stabilität sehr gut, "frontale" Stabilität gut
- Akku nicht austauschbar, WiFi-Dongle nicht anschließbar
- Vibration sehr stark mit "Noppen-Spikes", ohne diese immer noch stark
- Gripp mit "Noppen-Spikes" gut, ohne diese ist auf glatten Oberflächen kein Gripp vorhanden

Fazit:

- Aufgrund der Vibration und anderen Einschränkungen eher schlecht geeignet

Testdesign Nummer 2:

Im zweiten Testdesign wurden 2 der sehr großen Räder in Kombination mit einem Rollerball verwendet.

Besonderes Augenmerk wurde dabei auf folgende Eigenschaften gelegt:

- Stabilität der Radbefestigung und der gesamten Konstruktion
- Offener Zugang zum Akku, zum Ladeanschluss, zu den USB- und MicroSD-Steckplätzen, sowie den Sensor- und Motorports und der Bedienoberfläche
- Bedienoberfläche nach oben gerichtet und aus der gleichen Richtung bedienbar, wie später das montierte Phablet
- Schwerpunkt tief und etwas weiter nach hinten verlagert, um später das Gewicht des Phablets auszugleichen
- Größtmögliche Bodenfreiheit

Beobachtung des Testlaufs:

Die Vibrationen sind weniger stark, aber immer noch vorhanden. Die Auswirkungen auf das Tracking müssen untersucht und evtl. die "Aufhängung" des Phablets mit Dämpfern ausstattet werden.

Später im Projekt wurde dieses Design noch einmal verbessert, dabei wurde das Ziel verfolgt den Bot kompakter zu gestalten und die Verwindungssteifigkeit zu verbessern.

Die finalen Maße sind:

- Radstand Außenkante der Reifen: 22,4 cm
- Radstand Innenkante der Reifen: 15 cm
- Länge: 20 cm
- Gesamthöhe: 22 cm
- Maximaler Radius bei Rotation um eigene Achse: 18 cm
- Höhe Kamera: 19 cm

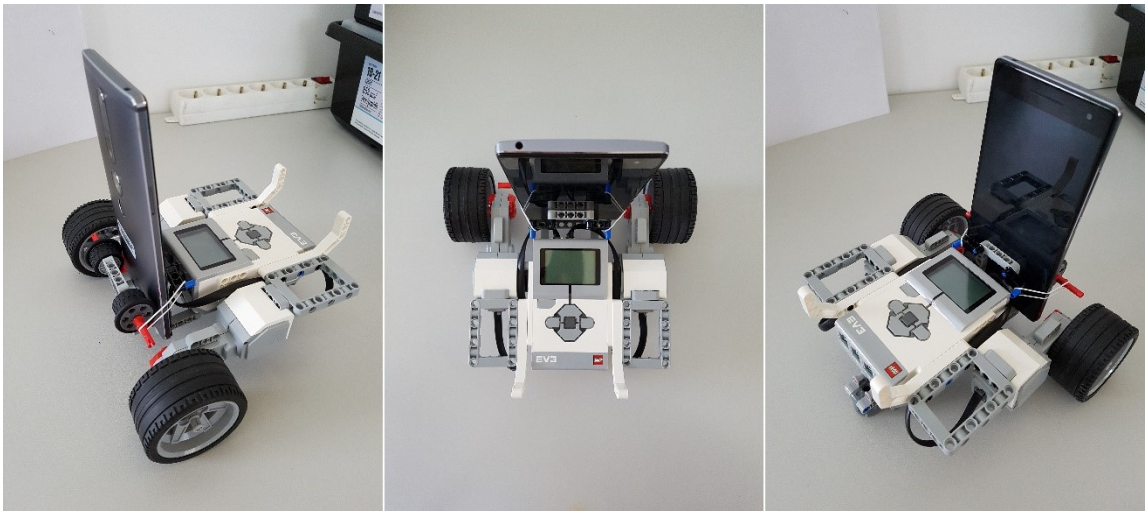

Abbildung 2: Finales Bot-Design

Es wurde die Möglichkeit in Betracht gezogen als „Notaus-Schalter“ einen frontal angebrachten Ultraschallsensor zu verwenden, mit dem der Bot softwareseitig im Notfall verhindert, dass er gegen ein zu nahes Hindernis fährt. Diese Idee wurde aber letzten Endes verworfen und softwareseitig auf dem Phablet umgesetzt.

3.1.2 Steuerung

Voraussetzungen:

- Smartphone-seitig: Hinzufügen der "ev3classes.jar" zu dem Android Projekt
- Bot-seitig: Bootfähige Installation von LeJOS auf einer SD-Karte installieren

Die Bot-Steuerung wird über eine WiFi-Verbindung realisiert. Dazu muss der Nutzer auf dem Smartphone einen mobilen WiFi-Hotspot öffnen und den Bot mit diesem verbinden.

In der Android App wird über die von der LeJOS-Bibliothek zur Verfügung gestellte Klasse „RemoteRequestEV3“ eine Remote-Verbindung hergestellt, über die z. B. die Motoren direkt angesteuert werden können.

In der Klasse BotController wurde eine einfache Abstraktion der Motor-Steuerbefehle implementiert. Sie bietet die Befehle „forward“, „backward“, „turnLeft“, „turnRight“ und „stop“ und ähnelt stark den verschiedenen Piloten-Implementierungen in LeJOS. Bei Experimen-

ten mit der über das Remote-Interface angebotenen Piloten-Klasse ist es jedoch teilweise zu sehr merkwürdigem Verhalten gekommen.

Problem dabei war es, dass ein Puffer die gesendeten Kommandos aufnimmt und in der Ankunftsreihenfolge speichert. Die Geschwindigkeit, mit der die Befehle abgearbeitet werden ist häufig zu niedrig gewesen und so kam es dazu, dass die Latenz zwischen senden und ausführen eines Befehls immer größer wurde und Befehle teilweise sogar verloren gingen. Für die Fernsteuerung eines Roboters durch einen Menschen ist diese Implementierung wahrscheinlich völlig ausreichen, für ein auf externem Feedback basierendes Steuerungssystem sind das aber sehr schlechte Voraussetzungen. Diesem Problem wurde begegnet, indem der letzte Befehl gecached wird und nur dann ein neuer Befehl gesendet wird, wenn dieser sich von dem letzten unterscheidet.

3.2 Planung

3.2.1 Umgebungsverarbeitung

Drei Dimensionen verkomplizieren die Navigationsplanung ungemein. Einige Ansätze basieren darauf viele verschiedene Wege auf Umsetzbarkeit zu überprüfen und falls nötig anzupassen. Diese Ansätze sind besonders dann geeignet, wenn das Fahrzeug in der Änderung der Richtung auf bestimmte Weise eingeschränkt ist, oder ein bestimmter Pfad von Beginn an vorgesehen ist, der möglichst genau eingehalten werden soll.

Der Ansatz, der in dieser Arbeit verfolgt wurde war es die dreidimensionale Umgebung, die in Form eines Octrees vorliegt auf eine zwei dimensionale Darstellung zu vereinfachen, da sich der Bot sowieso nur auf der Bodenebene bewegen kann. Ein kompletter Schnitt durch den Octree mit anschließendem Extrahieren eines 2D-Arrays kam dabei nicht infrage, da Steigungen und somit Änderungen in der Höhe des Bodens berücksichtigt werden sollten.

Bei der Recherche, wie schon bestehende Systeme dieses Problem lösen, bin ich auf die Flexible Collision Library (FCL) gestoßen, die die Kollisionserkennung auf einem Octree des OctoMap Frameworks unterstützt. Bei weiterer Nachforschung hat sich allerdings herausgestellt, dass diese Bibliothek hauptsächlich zur Kollisionsabfrage zwischen Geometrie in Form von Primitiven oder Meshes dient und nur ein kleiner Teil tatsächlich für die Kollisionsabfrage in einem Octree zuständig ist. Außerdem würde eine Kollisionsabfrage alleine nicht das Problem lösen alle navigierbaren Positionen auf effiziente Art und Weise zu finden.

Die letzten Endes verwendete Methode basiert auf der Vereinfachung der Roboterform auf einen aufrecht stehenden Zylinder. Damit ein den Boden repräsentierender Knoten navigierbar ist müssen dann zwei Dinge gegeben sein:

1. Über ihm muss genug als frei bekannter Raum gegeben sein
2. Um ihn herum müssen genug Nachbarn vorhanden sein die auch Bedingung 1 erfüllen und zu denen zusätzlich kein zu großer Höhenunterschied besteht.

Der Algorithmus der diese Bedingungen berücksichtigt und einen Adjazenzgrafen der traversierbaren Knoten aufbaut, ist in drei Hauptteile aufgeteilt:

1. Von allen besetzten Knoten aus werden schrittweise die oberhalb liegenden Nachbarn untersucht und so festgestellt, ob sich über dem Boden genügen freier Raum befindet. Zur Verbesserung der Effizienz wird hierbei die Methode zum Finden eines Nachbarknotens, die in Abschnitt 2.1.2b) vorgestellt wurde, verwendet. Besitzt ein Knoten genügend Freiraum, wird er als solcher markiert.
2. Für alle im vorherigen Schritt markierten Knoten werden nun die 8 potenziellen direkten Nachbarn untersucht. Um dabei auch schräge Untergründe und damit Sprünge in der Raster-ähnlichen Struktur eines Octrees zu unterstüutzen müssen neben dem auf der gleichen Ebene liegenden Nachbarknoten auch der darüber liegende und darunterliegende Knoten berücksichtigt werden. Die Reihenfolge verläuft dabei von oben nach unten, da die Iteration so nach dem Finden des ersten besetzen Knotens auf jeden Fall abgebrochen werden kann. Handelt es sich bei einem gefundenen Nachbarn auch um einen Knoten mit genügen Freiraum über sich, wird dieser mit dem aktuellen Knoten bidirektional als Nachbar verknüpft. Handelt es sich bei der theoretischen Position des Nachbarn um freien Raum, wird der aktuelle Knoten als Frontier-Knoten markiert.
3. Um den Radius des „Bot-Colliders“ zu berücksichtigen werden im dritten und letzten Schritt wieder alle Knoten durchlaufen und dabei der zuvor entstandene Adjazenzgraf aus gegenseitigen Nachbarn in mehreren Iterationen um die an der Grenze (nicht Frontier) liegenden Knoten reduziert. Die Anzahl der benötigten Iterationen lässt sich mit der Division des Radius durch die Auflösung des Octrees berechnen. Die Reduktion selbst funktioniert wie eine Regel eines zellulären Automaten: Wenn die Zahl der Nachbarn unter einer festgelegten Zahl (z. B. <7) liegt, wird der Knoten von seinen Nachbarn getrennt und verliert so seine Traversierbarkeitseigenschaft, gibt aber seine Frontiereigenschaft weiter, sofern vorhanden. Um die Funktionalität eines zellulären Automaten trotz der seriellen Iteration der Knoten korrekt umsetzen zu können, werden zwei Hilfsvariablen („removedNeighborCount“ und „wasFrontierNode“) benötigt, die zwischen den Iterationen wieder zurückgesetzt werden müssen.

Um all diese Schritte möglichst effizient ausführen zu können werden sie nur in einer bestimmten Bounding-Box durchgeführt. Zwischen den Updates der Navigierbarkeit wird der Raum in dem die durch Einfügen von PointClouds aktualisierten Knoten liegen in Form der minimalen und maximalen Koordinaten bzw. Schlüssel überwacht und gespeichert. So bleibt der rechenzeitliche Aufwand auch bei anwachsender Kartengröße in kontrollierbarem Rahmen.

3.2.2 Pfadplanung

Für die Planung des Erkundens wurden verschiedene Methoden in Betracht gezogen. Das große Problem dabei ist, dass die Umgebung zur Zeit des Planens noch nicht vollständig bekannt ist und damit keine perfekte Traversierung der Umgebung sichergestellt werden kann.

a) Erste Ansätze waren:

„Auf gut Glück“ in zufällige Richtungen fahren, eine Strategie die bei vielen Staubsauger und Rasenmährobotern zum Einsatz kommt.

Problematik: Es gibt keine Garantie, dass die Umgebung in endlicher Zeit komplett erforscht wird und durch mehrfaches Besuchen der gleichen Stellen ist diese Methode sehr ineffizient.

„Außenkanten abfahren“, dann Innenraum füllen

Vorteile: Nach initialem Abfahren der Kanten ist die maximale Ausbreitung der Map bekannt

Problematik: „Innenraum“ nicht immer endlich, bzw. in angemessenen oder gewünschten Umfang endlich

Vergleich: Tiefensuche bei Bäumen/Graphen

„Sternsuche“, sternförmiges erweitern des bekannten Raumes

Vorteile: man behält ein stark/dicht zusammenhängendes bekanntes Gebiet

Problematik: möglicherweise ineffizient, da Positionen evtl. sehr oft mehrfach besucht werden

Vergleich: Breitensuche bei Bäumen/Graphen

b) Clustering von Frontier-Knoten

Nach einiger Recherche wurde dann eine Strategie gewählt die unter anderem auch in [7] angewandt wurde. Bei dieser werden die an der Grenze zu unbekannten Gebiet liegenden Felder, sogenannte Frontier-Felder, zu Clustern zusammengefasst und als Navigationsziele bewertet und angesteuert.

Um die Frontier-Knoten zu Clustern zusammenzufassen werden alle Blattknoten iteriert. Nur die Knoten bei denen die Attribute „isFrontierNode“ und „hasFreeSpaceAbove“ wahr sind, sind dabei tatsächlich Knoten von Interesse. Für jeden dieser Knoten wird die Distanz zu allen bisher bekannten Clustern berechnet und das räumlich am nächsten gelegene Cluster ermittelt. Gibt es kein Cluster oder ist die Distanz größer als die maximale Reichweite des Clusters, so wird ein neues Cluster angelegt, ansonsten wird der Knoten zu diesem Cluster hinzugefügt. Ein Cluster speichert neben der maximalen Reichweite nur die gewichtete zentrale Position und die Anzahl aller enthaltenen Knoten. Da für jeden Knoten alle Cluster durchlaufen werden müssen und die tatsächliche Distanz zu den Clustern nicht von Interesse ist, wird der Vergleich der Distanzen im Quadrat ausgeführt. So kann die rechenintensive Berechnung der Quadratwurzel vermieden werden.

c) Wegfindung zu den Clustern

Für das Berechnen der Wege zu den einzelnen Clustern wurde ein A-Stern Algorithmus implementiert. Die meisten verfügbaren Implementationen operieren auf Adjazenzmatrizen

oder auf der Darstellung des Graphen in Form einer begrenzten zweidimensionalen Gridmap. Sie kamen daher für diese Anwendung nicht infrage.

Für die eigene Implementation wurden im Hinblick auf Performance für die Closed-List ein assoziatives Datenfeld in Form der Implementation der „unordered_map“ der Standardbibliothek und für die Openlist eine Kombination aus einer unordered_map und einem Binärheap gewählt. So erhält man für den zufälligen Zugriff auf ein Element bei beiden Listen die konstante Laufzeit des Hashing-Verfahrens und hat mit dem binären Heap der Open-List trotzdem eine Implementation einer Priority-Queue.

Für die Hashfunktion der unordered_map[s] wurde die von OctoMap implementierte Key-Hashfunktion genutzt. In der OpenList dient die assoziative Map nur dazu die Positionen der Nodes in dem Array das dem Binärheap zugrunde liegt zu speichern. Um diese Position bei Umstrukturieren des Heaps aktualisieren zu können hat jede Node der Open-List einen Pointer zu der entsprechenden Speicheradresse in der Map.

Nachdem der ansonsten klassische Ablauf des A-Stern Algorithmus beendet und ein Pfad gefunden wurde, erhält man eine Liste an Knoten die jeweils direkt nebeneinanderliegen. Um die Navigation entlang dieser Knoten für den Bot zu erleichtern, werden Knotenabfolgen die ausschließlich in die gleiche Richtung verlaufen kollabiert, sodass nur noch Start und Endknoten eines solchen Teilstücks vorliegen und an jedem Knoten des Pfades ein Richtungswechsel stattfindet.

d) Auswahl des nächsten Weges anhand von 3 Kriterien:

Nachdem nun die Wege zu allen Clustern berechnet wurden, muss einer dieser Wege als nächstes Navigationsziel ausgewählt werden. Die folgenden Kriterien werden dabei beachtet:

- Länge des Weges
- Winkel der benötigten Drehung, um den Pfad zu beginnen
- Anzahl der Nodes in dem zugehörigen Cluster.

Diese drei Kriterien werden normalisiert, mit unterschiedlichen Faktoren gewichtet und summiert. Nach dem daraus errechnetem Wert wird ein Ranking aufgestellt und der Pfad, der am besten abschneidet als nächstes Ziel ausgewählt.

3.3 Ausführung

Die Steuerung des Bots durch den Navigator wurde mithilfe eines einfachen Behaviour-Systems implementiert. Jedes Behaviour erbt von der abstrakten Klasse BotBehaviour und muss neben dem Code, der zur Ausführung erforderlich ist, auch eine Möglichkeit bieten die Ausführung zu pausieren und sie vorzeitig abzubrechen.

Im Folgenden sind die drei Behaviour erklärt:

3.3.1 Halt

Das HaltBehaviour dient dazu die Bewegung des Bots für einen bestimmten Zeitraum lang zu pausieren. Da es den Bot nicht bewegt, läuft die Ausführung dieses Behaviours auch weiter, wenn der Bot z. B. über das User-Interface angehalten werden soll.

3.3.2 LookAround

Das LookAroundBehaviour hat die Aufgabe den Bot eine 360° Rechts-Drehung vollführen zu lassen, um die komplette Umgebung um sich herum zu erfassen ohne die Position zu verändern. Es wird außer zu Beginn immer dann eingesetzt, wenn der Navigationsplaner keinen neuen Pfad mehr finden kann. Dieses und das FollowPathBehaviour bedienen sich zur Durchführung der benötigten Vektormathematik der Bibliothek „javax.vecmath“.

Bei diesem Behaviour wird der Winkel zwischen der letzten Blickrichtung und der aktuellen aufsummiert und sobald 360° bzw. 2π erreicht sind der Bot gestoppt und das Behaviour beendet.

3.3.3 FollowPath

Das komplexeste der drei Behaviour ist eine Implementation eines einfachen Pathfollowing-Algorithmus. Es basiert auf der abwechselnden, möglichst präzisen Ausrichtung des Bots auf den aktuellen Wegpunkt durch Rotation um die eigene Achse und dem vorwärtsbewegen bis die Abweichung im Winkel einen bestimmten Wert überschritten hat. Diese eher ineffiziente Steuermethode musste gewählt werden, da sie mit wenigen Steuerbefehlen funktioniert und die Remote-Verbindung zu dem Bot so nicht überfordert wird (siehe Abschnitt 3.1.2).

Ist ein Wegpunkt erreicht, wird der nächste ausgewählt und angesteuert. Kommt der Bot an das Ende des Pfades, wird das Finden eines neuen Pfades angefordert und das Behaviour beendet.

Sollte der Bot wegen falscher Planung z. B. durch eine fehlerhafte Positionsangabe durch Google Tango zu nah vor ein Hindernis fahren, wird der Zusammenstoß durch eine auf der aktuellsten PointCloud ausgeführte Kollisionserkennung verhindert und das aktuelle Verhalten auf das LookAroundBehaviour gewechselt. Damit die Kollisionserkennung positiv ausfällt, muss ein bestimmter Schwellwert an Punkten die in der Bounding-Box des Bots liegen überschritten werden. In Tests hat sich ein Wert von 0.4% von maximalen 60000 Punkten einer PointCloud als robust gegenüber Störungen aber trotzdem sensitiv gegenüber dem Ernstfall erwiesen. Der Schwachpunkt bei dieser Methode sind die Ränder der Wahrnehmung der Tiefenkamera. Daher kann es immer noch vorkommen, dass der Bot mit einem Rad an einem Hindernis hängen bleibt.

4 Applikation

Die Anwendung wurde in Android Studio entwickelt. Durch das Plugin NDK (Native Development Kit) ist es möglich auch Teile des Programms in C bzw. C++ zu schreiben und diese dann über ein JNI (Java Native Interface) in die App zu integrieren.

Als Version Controll System wurde Git genutzt. Während der Entwicklung wurde das Projekt in einem privaten Repository auf BitBucket gehalten und zur Abgabe und Veröffentlichung inklusive der Dokumentation und allen Materials auf GitHub hochgeladen.

4.1 Architektur und Pipeline

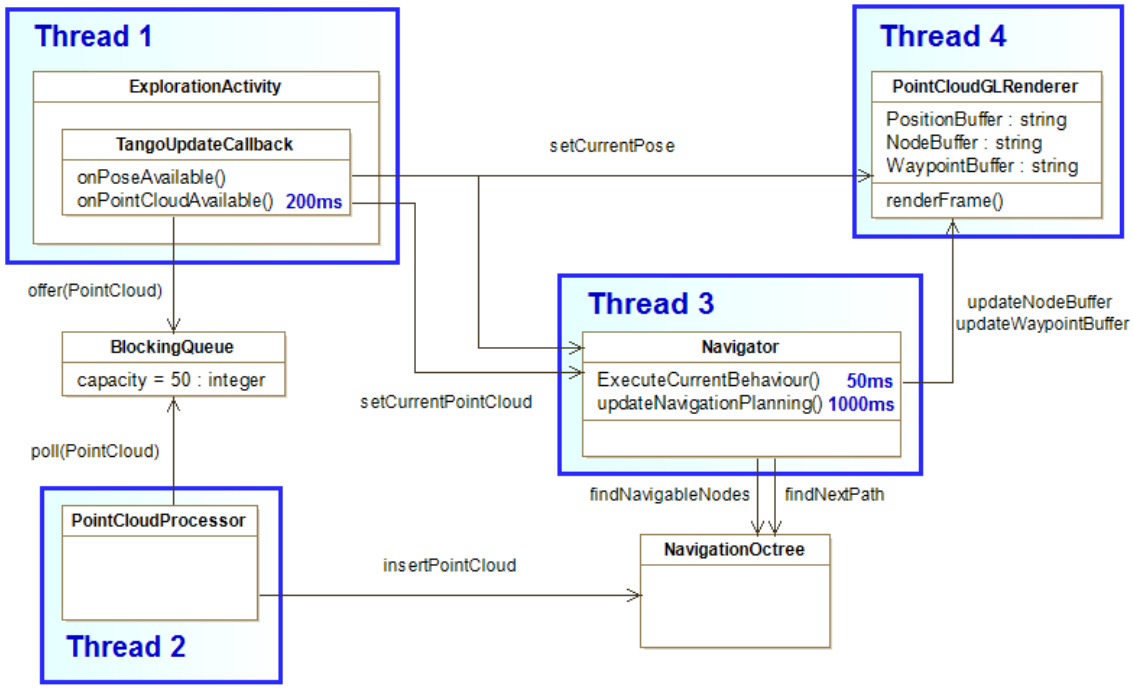

Abbildung 3: Pipeline (blaue Zeitangaben geben die Häufigkeit zeitgesteuerter „Events“ an)

Die verschiedenen Teile der Anwendung sind auf unterschiedliche Threads aufgeteilt, da es wichtig ist, dass sie unabhängig voneinander laufen und sich nicht gegenseitig blockieren.

Thread 1 beinhaltet die anonyme Klasse, die das TangoUpdateCallback Interface implementiert. Dieser Thread darf auf keinen Fall blockiert werden, da das Callback von einem Tango-Thread selbst aus aufgerufen wird. Ein Blockieren oder Durchführen rechenintensiver Aufgaben in den Callbackmethoden hätte zur Folge, dass Tango-interne Funktionen wie das Motion Tracking nicht mehr richtig funktionieren. Daher dient diese Klasse nur dazu die Daten, also Posen und PointClouds an die anderen Threads weiterzugeben.

Thread 2 dient alleine dem Einfügen der neuen PointCloud-Daten in den Octree. Zwischen Thread 1 und Thread 2 befindet sich eine BlockingQueue als Mittel der Kommunikation der PointCloud-Daten. Da das Einfügen von neuen PointCloud Messungen durchaus die Zeitspanne von ca. 200 ms zwischen zwei PointCloud-Updates überschreiten kann dient sie als Puffer und Synchronisationswerkzeug. Ist die Queue leer, blockiert sie den PointCloudPro-

zessor wenn dieser versucht ein PointCloud-Daten Objekt zu anzufordern. Ist die Queue voll, blockiert sie aber nicht den Callback-Thread, sondern ignoriert das angebotene PC-Objekt. Dies sollte aber in der Realität nicht vorkommen, da vorher bei vollzulaufen drohender Queue der Bot pausiert und so keine neuen Messungen mehr berücksichtigt werden.

Thread 3 besteht intern tatsächlich aus zwei verschiedenen Threads, die hier jedoch der Einfachheit halber zusammengefasst werden. Die beiden Zuständigkeiten entsprechen den in den Abschnitten 3.2 und 3.3 beschriebenen Aufgaben. Um diese Aufgaben durchführen zu können benötigt der Navigator Zugriff auf die aktuellste Position sowie die aktuellste PointCloud. Erstere um die Planung der Navigation durchführen und die Einhaltung der Ausführung überprüfen zu können und letztere als Echtzeit-Kollisionserkennungssystem. Außerdem braucht der für die Planung zuständige Thread des Navigators natürlich Zugriff auf den Octree. Da sowohl der PointCloudProzessor als auch der Navigator um diesen Zugriff kämpfen, muss eben dieser Zugriff über einen Lock synchronisiert werden. Der Lock wird dazu auf dem ExplorationData Objekt ausgeführt. Hier befindet sich auch der größte Bottleneck der Anwendung, da der jeweils andere Prozess erst warten muss, bis er den Lock bekommt und sowohl die Operationen zum Einfügen der PointClouds als auch die Operationen zur Navigationsplanung recht viel Zeit beanspruchen.

Thread 4 dient der Visualisierung des aktuellen Zustands des Octrees, des Positionsverlaufs und des aktuell geplanten Pfades. Die Informationen befinden sich in den drei entsprechenden Floatbuffern, die von außen von den CallbackUpdates und dem Navigator manipuliert werden.

4.2 Dateiformat und Inhalte der Metadaten

Die im Folgenden beschriebene Funktionalität ist in der Klasse „ExplorationDataLoader“ implementiert.

Um die Explorations zu speichern und später visualisieren zu können, müssen zusätzlich zu dem OcTree einige zusätzliche Daten gespeichert werden. Um diese zusammenhängend verschieben und versenden zu können werden die einzelnen Dateien zu einem Zip-Archiv zusammengepackt und komprimiert. Da die Verwendung der dafür von dem Java Development Kit zur Verfügung gestellten Klassen Probleme bereitet hat, wird jetzt die Bibliothek „zt-zip“ [8] dazu verwendet.

Eine Exploration-Datei umfasst folgende Inhalte:

- **Octree**: Binär gespeicherte Version des erzeugten Octrees
- **ADF**: Gespeichertes und exportiertes Ergebnis des Area Learnings des Tango SDKs (im Moment eine leere Datei, siehe Abschnitt 1.3)
- **PositionHistory**: Chronologische Liste der Positionen, an denen sich das Gerät bzw. der Bot befunden hat als JSON. Um den benötigten Speicherplatz gering zu halten werden neue Positionen nur eingefügt, sobald eine bestimmte Strecke zurückgelegt wurde.

- **MetaData**: Metadaten über die „Aufnahme-Session“ als JSON
    - **Timestamp**: Beginn der Exploration
    - **Maxrange**: Maximale Reichweite, für die Sensorinformationen eingefügt werden sollen (siehe Realtime Performance)
    - **Minrange**: Alle Punkte, die näher an der Kamera waren wurden ignoriert
    - **Bot Properties**: Alle Eigenschaften des Bots, die in den Einstellungen der Applikation vorgenommen wurden, außer der IP-Adresse

Um die Metadaten und den Positionsverlauf in einem einfach lesbaren Format abzuspeichern wurde der Weg über einen JSON Parser gewählt. Dazu wird die JSON Library Gson [9] die von Google zur Verfügung gestellt verwendet.

4.3 Automatisierte Unit Tests

4.3.1 Java

Für Java Code bietet das Android Studio mit der Integration von JUnit ein hilfreiches Tool, um die Funktionalität geschriebener Methoden zu testen und auf Dauer sicherzustellen. In Android Studio werden die Tests in die Packages „test“ und „androidTest“ unterteilt. Wie die Namen vermuten lassen unterscheiden sie sich durch den Ort der Durchführung. Die Tests des Packages „test“ werden auf dem aktuellen Computer ausgeführt und die Tests des Packages „androidTest“ auf dem angeschlossenen Android-Gerät. So lassen sich neben einfachen Tests wie dem „GeometryHelperUnitTest“, der einige mathematische Hilfs-Funktionen testet, auch komplexere und von der Installationsumgebung abhängige Testfälle formulieren. „OcTreeJNITest“ dient dazu die korrekte Installation des Octomap-Frameworks sowie die Funktionalität der erweiternden Klassen und Methoden zu überprüfen und „Bot-ControllerTest“ ist eine Testklasse, die den Verbindungsaufbau zwischen dem Smartphone und dem EV3 Bot sowie die grundlegenden Navigationskommandos kontrolliert.

4.3.2 C++

Für C++ Code ist es in Android Studio nicht so einfach automatisierte Tests anzulegen und durchzuführen, aber es gibt z. B. die Methode „assert(…)“ der Bibliothek „assert.h“. Sie hat die gleiche Funktion wie ein „assertTrue(…)“ in JUnit und wirft einen Error wenn die Bedingung nicht wahr ist. Da diese Überprüfungen natürlich auch Rechenzeit beanspruchen, werden sie vom Compiler nur übersetzt, wenn das Debug-Flag gesetzt ist.

Für die Prüfung der Integrität der erwarteten Zustände der Knoten zum Beispiel sind diese Statements extrem hilfreich, sowie auch beim Auffinden von Fehlern im Code. Die normalen Fehlermeldungen unter Verwendung des NDKs sind sehr oft nicht besonders aufschlussreich.

4.4 Tour durch das User-Interface

4.4.1 Einstellungen

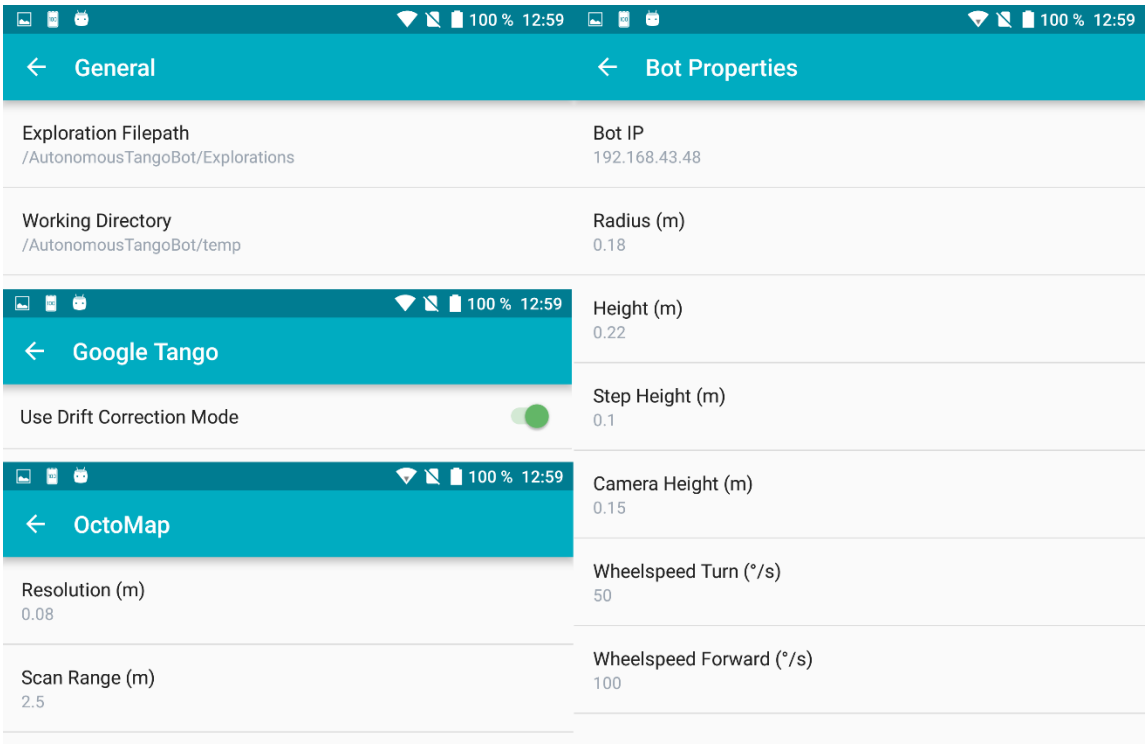

Abbildung 4: Einstellungen

In den, aus dem Hauptmenü oben rechts und aus dem New Exploration Menü über den Butten in der Mitte erreichbaren Einstellungen der Anwendung lassen sich verschiedene Parameter persistent speichern. Die verschiedenen Settings sind nach Zugehörigkeit in vier verschiedene Gebiete unterteilt:

1. Unter „General“ finden sich die Einstellungen die den Speicherort der Exploration-Dateien sowie den temporären Speicherort zum Packen der Dateien angeben
2. Unter der Gruppe „Google Tango“ lässt sich der Drift-Korrektur-Modus ein- und ausschalten
3. Unter der Überschrift „Octomap“ findet man die Parameter zur Einstellung der Auflösung des OcTrees und zur Begrenzung der maximalen Scan-Reichweite
4. Und unter der letzten Kategorie „Bot Properties“ sind all die Eigenschaften gesammelt, die den Bot beschreiben und so Einfluss auf die Navigationsplanung und Ausführung haben.

4.4.2 Neue Exploration

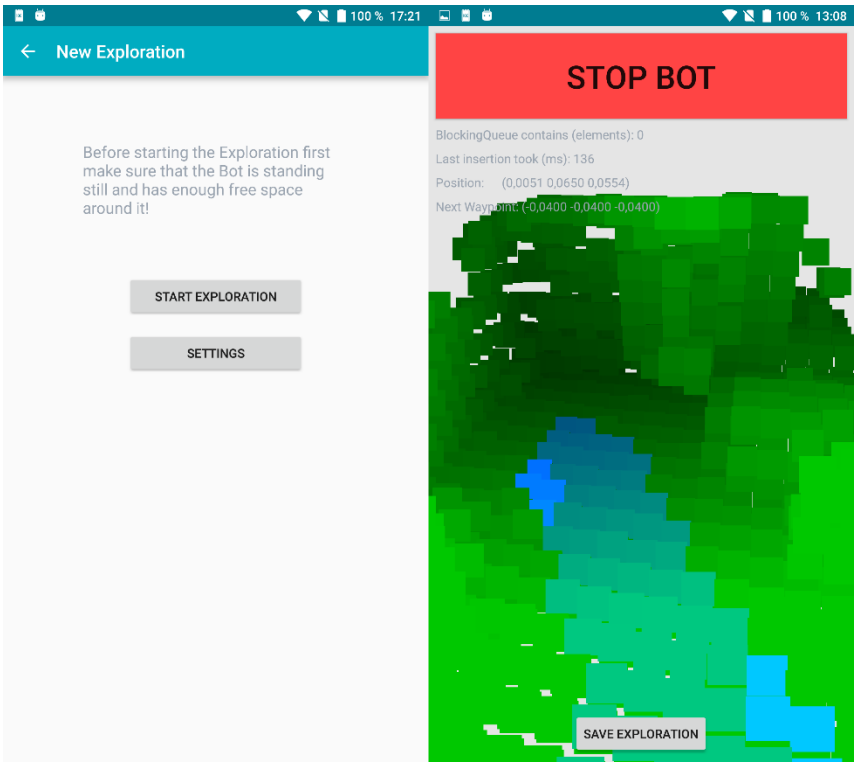

Abbildung 5: Starten einer neuen Exploration

Auf der linken Seite von Abbildung 5: Starten einer neuen Exploration ist das New Exploration Menü mit den Hinweisen zur ordnungsgemäßen Benutzung und den Buttons zum Starten und zum Ändern der Einstellungen zu sehen.

Auf der rechten Seite befindet sich das Interface, das während einer Exploration zu sehen ist. Ganz oben ist ein großer Stop-Button platziert, der die Bewegung des Bots sofort pausieren kann. Darunter sind einige hilfreiche Debug-Informationen zu finden. Sie umfassen:

- Den aktuellen Füllstand der BlockingQueue
- Die Zeit, die das Einfügen der letzten PointCloud gedauert hat
- Die aktuelle Position des Bots
- Und den als nächstes anzusteuernden Wegpunkt.

Im Hintergrund ist eine in OpenGL implementierte Visualisierung des NavigationOcTrees zu sehen. Für sie wurden die zwei Klassen PointCloudGLSurfaceView und PointCloudGLRenderer von der Klasse GLSurfaceView und dem Interface GLSurfaceView.Renderer abgeleitet bzw. implementiert.

Es musste für das korrekte Darstellen des Baumes und der Bewegung der Position zunächst die korrekte Matrix-Multiplikations-Kette herausgefunden werden. Das Hindernis dabei war es, dass die Parameter der Methode „Matrix.multiplyMM()“ von rechts nach links zu lesen waren, um die in der mathematischen Notation von links nach rechts notierte Operation durchzuführen. Am Ende ist es so gelungen, die reale Kameraposition und -orientierung als Pose der virtuellen Kamera zu verwenden.

Um den OcTree effizient darstellen zu können, mussten angepasste Vertex-Shader geschrieben werden. Um diese nicht als String in dem Quellcode fest programmieren zu müssen, habe ich zusätzlich einen kleinen ShaderLoader implementiert, der den Shader Quellcode aus einer Textdatei liest. Dabei war es wichtig hinter jeder eingelesenen Zeile den Escape-Charakter für neue Zeile einzufügen, da sonst Zeilenkommentare bis an das Ende der Datei galten.

Die einzelnen besetzten Knoten des Octrees werden in dieser Visualisierung als GL_POINTS (Quadrate) und nicht als kleine Würfel, wie in OctoVis, dargestellt. Das liegt vor allem daran, dass für eine solche komplexe grafische Darstellung das System nicht geeignet ist und es während der Exploration unnötig belastet würde. Über das Variieren der Größe und Farbe der Punkte kann jedoch trotzdem ein recht guter Eindruck der räumlichen Beschaffenheit des Baumes vermittelt werden.

Um den richtigen Faktor für die Darstellung der Größe der einzelnen Knoten zu erhalten, muss in dem Shader eine perspektivische Division durchgeführt werden. So erhält man zunächst die normalized device coordinates (NDC) [10]. Dazu werden einfach die X-, Y- und Z-Komponente des, in den Clip-Space transformierten Punktes durch die W-Komponente geteilt. Die neu erhaltene Z-Komponente gibt dann einen Wert von 0 bis 1 an, bei dem 0 einem Punkt an der Near-Plane der Kamera und 1 einem Punkt an der Far-Plane der Kamera entspricht. Dieser Wert ist aber nicht linear und kann, wenn man ihn von 1 subtrahiert als Faktor für die Größe der darzustellenden Quadrate verwendet werden.

Die Farbe interpoliert über eine vorher programmatisch angefertigte Look-Up-Textur. In der Ferne wird sie immer dunkler, auf mittlere Distanz sind es verschiedene Grün-Schattierungen und in der Nähe durchläuft sie mehrere Gelb-, Orange- und Rottöne.

Außerdem spiegelt die Farbe noch weitere Informationen über die Knoten dar. So entsprechen türkise Quadrate navigierbaren Knoten und hellblaue unter diesen navigierbaren Knoten diejenigen, die die Frontier-Eigenschaft besitzen.

Schwarz und Magenta sollten nicht auftreten, da sie Fehler in der Integrität der Knotenzustände darstellen und zu Debug-Zwecken eingefügt wurden.

Der Positionsverlauf sowie der geplante Weg werden als GL_LINE_STRIP (Durch Linien verbundene Vertices) dargestellt. Rot repräsentiert den Verlauf und Blau den geplanten Weg.

Um das gesamte Render-System der Knoten effizienter zu gestalten, könnte, falls es in der Zukunft nötig wäre, ein auch auf Octrees aufbauendes Chunk-System verwendet werden, bei dem ähnlich wie bei dem Navigation-Update nur die Chunks erneuert werden deren Knoten aktualisiert wurden. Da jedem Chunk dann ein VertexBuffer und dementsprechend auch ein eigener Drawcall zugeordnet sind, wird es immer ein Kompromiss zwischen Drawcalls und Update-Aufwand des Buffers bleiben.

4.4.3 Laden einer Exploration

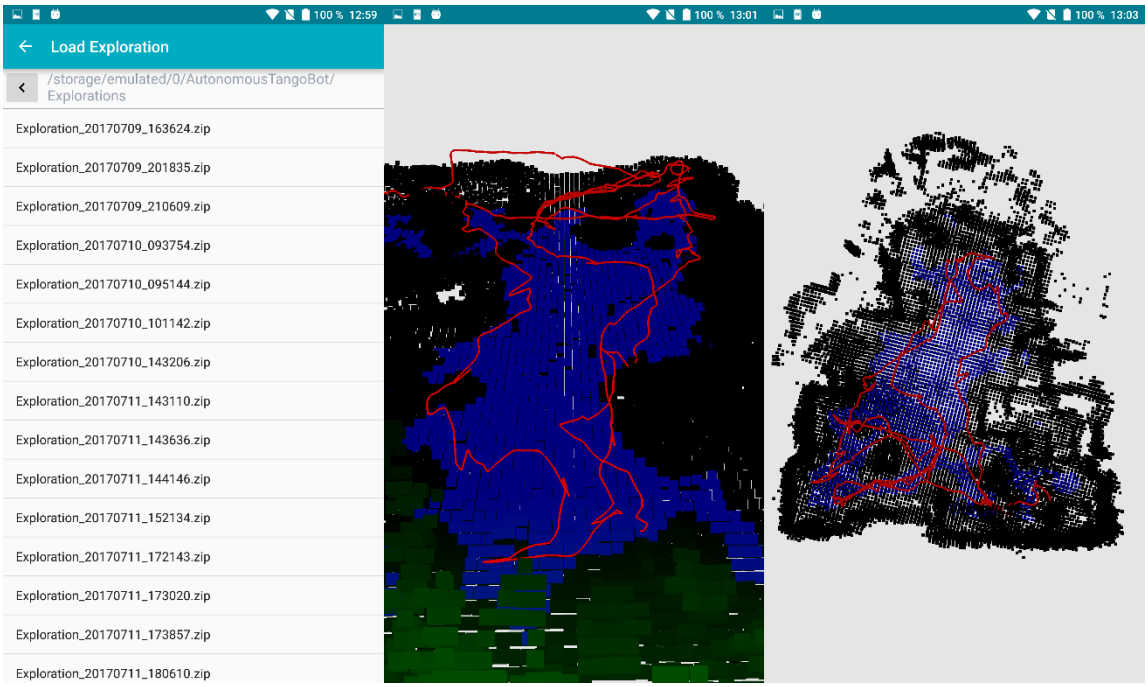

Abbildung 6: Laden gespeicherter Explorations

Auf der linken Seite der Abbildung 7 ist ein selbst implementierter File Chooser für Exploration-Dateien zu sehen, der nur Ordner und Zip-Dateien anzeigt. Mit ihm ist es möglich durch das lokale Dateisystem zu navigieren und Exploration-Dateien von beliebigen Orten zu laden.

In der Mitte und auf der rechten Seite ist die gleiche Visualisierung zu sehen, wie sie auch während der Exploration verwendet wird. Dadurch, dass man in dieser Visualisierung aber nicht mehr an die Position des Gerätes gebunden ist, sondern mit Touch-Gesten die Kamera frei bewegen kann, kann man sich einen viel besseren Überblick verschaffen und zum Beispiel die Bewegungsmuster anhand des rot dargestellten Positionsverlaufs analysieren.

Um die Kamera zu schwenken, muss lediglich mit einem Finger in die entsprechende Richtung über den Bildschirm gestrichen werden. Dabei wird sichergestellt, dass unten auch immer unten bleibt. Um die Position der Kamera zu verändern, werden zwei Finger benötigt. Durch streichen von oben nach unten bewegt sich die Kamera vorwärts, durch streichen von unten nach oben rückwärts und durch streichen von links nach rechts bzw. rechts nach links bewegt sie sich nach links bzw. nach rechts.

4.5 Performancebeobachtungen

Dem Lenovo Phab2Pro stehen nur recht stark begrenzte Hardware-Ressourcen zur Verfügung [11]:

- CPU Octa-core (4x1.8 GHz Cortex-A72 & 4x1.4 GHz Cortex-A53)
- GPU Adreno 510
- RAM 4GB, ca. 2 davon verfügbar.

Daher ist es alles andere als leicht gewesen geeignete Parameter für die Anwendung und vor allem für OctoMap zu finden, unter denen das System noch effektiv in Echtzeit laufen konnte. Die Bedingung dafür war es, dass die Zeit zum Einfügen im Durchschnitt die 200 ms des „onPointCloudAvailable“-Updateintervalls nicht überschreiten durfte. Wollte man eine feine Auflösung musste man die Scan-Reichweite recht stark begrenzen, hier ergaben sich Werte von 0.05 m für die Auflösung und 2 m für die maximale Reichweite. Wenn man bereit war eine etwas weniger feine Auflösung zu verwenden, konnte man bei 0.1 m Auflösung fast die durch den Sensor technisch bedingte maximale Reichweite des Scans von ca. 4.5 m erreichen.

Wie gut Parameter funktionieren hängt auch stark von der Umgebung ab. Befindet man sich in einem großen Raum, so liegen die Punkte der PointCloud weiter auseinander und lassen sich weniger gut diskretisieren. Befinden sich in der Umgebung viele Infrarotlicht-schluckende Oberflächen, so umfassen auch die PointCloud-Messungen weniger Daten und das Einfügen verläuft schneller.

Um die Datenmenge weiter zu reduzieren wurden viele Versuche unternommen. Der effektivste ist es letztendlich gewesen, vor dem Einfügen der PointCloud die Veränderung der Pose seit dem letzten PointCloud-Update zu überprüfen, und die neue PointCloud nur einzufügen, wenn der Unterschied einen gewissen Schwellwert überschritten hat.

5 Objekt Segmentierung

Die ursprünglich geplante Implementierung der Objekt Segmentierung würde, anders als anfangs angenommen den Zeitrahmen der Praxisprojektarbeit stark überschreiten. Geplant war es, die in [12] vorgestellte Methode zu implementieren.

Von dem Autor wird auf der Internetseite der Technischen Universität München [13] eine auf Ubuntu vorkompilierte Version als Bibliothek angeboten. Sie eignet sich für die Verwendung für dieses Projekt leider nicht, da zum einen einige weitere Third-Party Bibliotheken vorausgesetzt werden und zum anderen, weil diese Bibliothek ihr eigenes SLAM System implementiert und daher als Eingabe nur die einzelnen Frames einer RGB-D Kamera annimmt. Würde dieses System mit Tango zusammen verwendet werden, würde die Positionseinschätzung beider Systeme schnell asynchron werden und die als Octree vorliegende physikalische Umgebungsrepräsentation nicht zu der aus dieser Bibliothek extrahierten semantischen Segmentierung passen.

Für eine eigene Implementierung fehlt aufgrund der Tatsache, dass der Source Code nicht frei verfügbar ist, die Arbeit stark auf anderen wissenschaftlichen Arbeiten aufbaut und weil Details zur Implementierung in der eben genannten wissenschaftlichen Ausarbeitung nicht genauer beschrieben werden, die Zeit.

Dennoch wird in dem folgenden Abschnitt ein grober Überblick über den Algorithmus gegeben. Der Algorithmus umfasst im Original sowohl eine Implementation eines SLAM Algorithmus als auch einen Teil zur Segmentierung. Hier wird nur der letztere Teil beschrieben.

5.1 Funktionsweise

Die grundsätzliche Idee bei diesem Algorithmus basiert darauf, die Segmentierung und das Update des globalen Modells komplett im Image-Space des RGB-D Bildes durchzuführen. Es soll so möglich sein diese Operation in konstanter Zeit durchzuführen, unabhängig von der Größe des gesamten Modells.

5.1.1 Globales Modell

Das globale Modell besteht aus einem ungeordneten Set von sogenannten Surfels. Surfels sind Objekte im 3D Raum, die eine Oberfläche repräsentieren. Gespeichert werden sie unter Abgabe der folgenden Attribute: Koordinate des Punktes, Normalenvektor der Oberfläche, Reichweite (vorzustellen als Radius einer Kreisfläche), sowie ein Timestamps, ein Confidence Wert und zuletzt eine ID des zugehörigen Objektes.

5.1.2 Einfügen einer neuen Messung

Soll nun ein vorliegender Frame einer RGB-D Kamera eingefügt werden, wird zunächst unter Verwendung des SLAM Algorithmus die Pose der Kamera geschätzt.

Aus dem Tiefenbild wird dann durch einfache Transformation eine Vertexmap erstellt. Mithilfe der Vertexmap kann wiederum eine Normalmap erstellt werden.

Diese Normalmap wird im nächsten Schritt verwendet, um konvexe zusammenhängende Flächen zu markieren. Der Algorithmus nutzt nämlich die Annahme aus, dass Objekte realen Leben vorrangig aus konvexen Formen bestehen. Bereiche der Normalmap die konkaven Flächen mit einer bestimmten Krümmung entsprechen werden als Zwischenräume der Objekte markiert.

Ist dieser Schritt abgeschlossen, werden die Pixel zusammenhängender Flächen mit der gleichen ID versehen.

Um die neue Messung in das globale Modell integrieren zu können müssen die betroffenen Surfels des globalen Modells zuerst in den Image-Space der aktuellen Messung projiziert werden. Ist dies geschehen, kann die Zugehörigkeit der Flächen der neuen Messung zu denen des globalen Modells anhand der Normalen und Positionen berechnet werden [12 Sec. III B]. Bringt dieser Zugehörigkeitsvergleich einen ausreichend höhen Überlappungsbereich einzelner Flächen hervor, so werden diese als ein und dieselbe Fläche angenommen. Um Robustheit gegenüber Artefakten in der Messung zu erreichen, müssen die Flächen die Mindestgröße einer bestimmten Pixelanzahl besitzen.

Der letzte Schritt vor dem Updaten des globalen Modells dient dazu Segmente zusammenzuführen, die zu dem gleichen Objekt gehören. Während dem Segmentationsprozess kann es dazu kommen, dass eine Fläche, verdeckt durch ein anderes Objekt, zunächst zwei oder mehr Teilsegmenten zu geordnet wird. Dieser Schritt erfasst nun ähnlich wie im letzten Schritt die überlappende Fläche und fügt so die verschiedenen Segmente zu einem Zusammenhängenden zusammen [12 Sec. III C].

In dem letzten Schritt wird das globale Modell aktualisiert. Es direkt mit den aus den vorherigen Schritten gewonnenen Werten an den entsprechenden Stellen zu ersetzen, wäre zu

anfällig für Fehler in den Messungen. Daher wird hier, ähnlich wie bei OctoMap mit einem Confidence-basierten System gearbeitet. Stimmt eine Messung mit dem Modell überein, wird die Confidence erhört, stimmt sie nicht überein, wird sie verringert. Wird sie mehrmals nacheinander verringert und fällt dabei auf 0, kann davon ausgegangen werden, dass der Punkt, der sich an dieser Stelle befindet, verändert hat und daher durch die aktuelle Messung ersetzt werden kann [12 Sec. III D].

Zusammenfassung und Ausblick

In dieser Arbeit wurde experimentell die Implementation einer Robotersteuerung zur autonomen Erkundung mithilfe der Technologie Google Tango durchgeführt. Zur räumlichen Repräsentation wurde das umfangreiche Framework OctoMap integriert und eine Architektur und Verarbeitungspipeline darum herum aufgebaut. Daneben wurde eine mobile Roboterplattform, basierend auf dem Lego Mindstorms EV3 Set, sowie deren Steuerung durch ein Behaviour-System konzipiert und umgesetzt. Zur Planung der Navigation wurden verschiedene Herangehensweisen betrachtet und eine davon implementiert. Um das Ganze grafisch ansprechend zu gestalten und ein besseres Verständnis der Vorgänge zu ermöglichen, wurde eine visuelle Darstellung der Octrees umgesetzt und in die Applikation integriert. Zuletzt wurde noch ein Algorithmus zur iterativen Segmentation der Umgebung vorgestellt, für dessen Implementation am Ende leider die Zeit gefehlt hat.

Dieses Projekt stellt meinen ersten Kontakt mit der Programmiersprache C++, der Shader-Sprache GLSL und der Konfigurationssprache CMAKE dar. Außerdem war es das erste Mal, dass ich eine Applikation in Android Studio entwickelt habe und das erste Mal, dass ich die Frameworks OpenGL ES 2.0, Google Tango und OctoMap verwendet habe. Ich habe mir also während des Projektes sehr viel neues fachliches Wissen aneignen können.

Vor allem aber hat es mir geholfen, die Grenzen meiner eigenen Arbeitsleistung in einem so umfangreichen Projekt aufzuzeigen und ermöglicht mir so in Zukunft hoffentlich den zeitlichen Aufwand einer Arbeit besser einschätzen zu können.

Für die Weiterführung der Arbeit würde sich z. B. anbieten das Thema der alternativen Repräsentationen der Umgebung zu vertiefen und dort besonders die Darstellung anhand von Distance Fields untersuchen. Aber auch die Planung einer effizienten Traversierung der Umgebung bietet viel Spielraum für weitere Forschung. Nicht zuletzt ist auch das Themengebiet der semantischen Umgebungsverarbeitung und -erkennung ein großes und interessantes Forschungsgebiet.

Anhang

Weitere Screenshots des User Interfaces sowie Videos von einigen Testläufen befinden sich zusammen mit dem Source Code der Anwendung in einem Repository unter https://github.com/jannismoeller/AutonomousTangoBot

Abbildungsverzeichnis

Quellenverzeichnis

[1] (18.05.2017): Tango Concepts | Tango | Google Developers. https://developers.google.com/tango/overview/concepts. Abgerufen am 15.07.2017.

[2] infineon: Product Brief REAL3™ image sensor family. 3D depth sensing based on Time-of-Flight. https://www.infineon.com/dgdl/Infineon-REAL3+Image+Sensor+Family-PB-v01_00-EN.PDF?fileId=5546d462518ffd850151a0afc2302a58.

[3] Hornung, A, Wurm, KM, Bennewitz, M, Stachniss, C, Burgard, W (2013): OctoMap. An efficient probabilistic 3D mapping framework based on octrees. Autonomous Robots, 34(3):189–206.

[4] gonzalezsieira/joctomap. https://github.com/gonzalezsieira/joctomap. Abgerufen am 15.07.2017.

[5] Usenko, V, Stumberg, Lv, Pangercic, A, Cremers, D: Real-Time Trajectory Replanning for MAVs using Uniform B-splines and 3D Circular Buffer.

[6] RosCon2013 Track1 Day1 04 Armin Hornung 3D Mapping with OctoMap. https://www.youtube.com/watch?v=RRP29VnY8go&feature=youtu.be&t=38m18s. Abgerufen am 05.07.2017.

[7] Thomas Wohlfahrt (2015): Exploration for autonomous 3D voxel mapping of static indoor environments with depth cameras and 2D odometry. http://elib.dlr.de/96335/.

[8] zeroturnaround/zt-zip. https://github.com/zeroturnaround/zt-zip. Abgerufen am 16.07.2017.

[9] google/gson. https://github.com/google/gson. Abgerufen am 16.07.2017.

[10] (06.04.2017): Vertex Post-Processing - OpenGL Wiki. https://www.khronos.org/opengl/wiki/Vertex_Post-Processing#Perspective_divide. Abgerufen am 17.07.2017.

[11] Lenovo Phab2 Pro - Full phone specifications. http://www.gsmarena.com/lenovo_phab2_pro-8145.php. Abgerufen am 15.07.2017.

[12] Tateno, K, Tombari, F, Navab, N (2015): Real-time and scalable incremental segmentation on dense SLAM. In: Burgard, W (Hrsg), *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). Sept. 28, 2015 - Oct. 2, 2015, Hamburg, Germany.* IEEE, Piscataway, NJ.

[13] ProjectInSeg. http://campar.in.tum.de/Chair/ProjectInSeg. Abgerufen am 05.07.2017.

TH Köln
Gustav-Heinemann-Ufer 54
50968 Köln
www.th-koeln.de

Technology
Arts Sciences
TH Köln