Reaktive Bewegungsplanung auf 3D-Sensordaten mittels GPU-basierter Kollisionserkennung

Untersuchung von hochgradig parallelen Algorithmen für mobile Serviceroboter

Zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
(Dr.-Ing.)

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte
DISSERTATION

von
Dipl.-Inform. Andreas Hermann

Tag der mündlichen Prüfung:	12. Juli 2018
1. Referent:	Prof. Dr.-Ing. Rüdiger Dillmann
2. Referent:	Prof. Dr.-Ing. Torsten Kröger

Danksagung

Diese Arbeit ist meiner wundervollen Frau Ümmüye gewidmet. Ohne ihre unendliche Unterstützung wäre es für mich undenkbar gewesen, meine Familie und die vorliegende Arbeit erfolgreich unter den einen berühmten Hut zu bekommen.
Yildizima teşekkürler!

Die folgenden Inhalte entstanden in den Jahren 2009 bis 2017 während meiner Tätigkeit als wissenschaftlicher Mitarbeiter in der Abteilung Interaktive Diagnose- und Servicesysteme am Forschungszentrum Informatik in Karlsruhe. Während der gesamten Zeit konnte ich wichtige Erfahrungen sammeln und mich in einer breiten Vielzahl von Themengebieten weiterbilden. Diese Möglichkeiten möchte ich nicht missen. Herrn Prof. Dr.-Ing. Rüdiger Dillmann danke ich besonders für die Anregung zu dieser Arbeit, die wissenschaftliche Förderung, die stets vorhandene Diskussionsbereitschaft und für die Übernahme des Hauptreferates. Für die freundliche Übernahme des Korreferates gebührt mein Dank Herrn Dr.-Ing. Torsten Kröger. Ein besonders großer Dank steht Zhixing Xue zu, für die Einführung in die wissenschaftliche Arbeit während meiner ersten Jahre am FZI und die wichtige Erkenntnis, dass *alle nur mit Wasser kochen*. Für viele wertvolle Gespräche, die zum Gelingen dieser Arbeit beigetragen haben, danke ich auch Prof. Dr.-Ing. J. Marius Zöllner. Arne Rönnau danke ich dafür, dass er als Abteilungsleiter die Freiräume zur Promotion geschaffen hat und keine Mühen scheute, das Arbeitsklima für seine Kollegen so positiv wie möglich zu gestalten. Ebenso fand ich in ihm einen großen Unterstützer der Voxel-Technologien. Weiterhin wären die praktischen Versuche dieser Arbeit ohne eine Reihe von herausragenden Studenten nicht möglich gewesen. Hier möchte ich Sebastian Klemm, Jian Sun, Jörg Bauer, Florian Drews, Matthias Wagner, Klaus Fischnaller, Felix Mauch, Herbert Pietrzyk und Christian Jülg meinen großen Dank aussprechen. Meinen IDS und TKS Kollegen danke ich für eine tolle Zusammenarbeit, den Zusammenhalt in stressigen Zeiten sowie für die angeregten Gespräche. Ganz besonders Danken möchte ich Sebastian Klemm für die gute Freundschaft, sowie Georg Heppner als stetige Quelle guter Laune. Meinen ehemaligen Büronachbarn Steffen Rühl und Marc Zofka gilt darüber hinaus der Dank für die stets gute Stimmung und den regelmäßigen Austausch von zuckerhaltigen Nahrungsmitteln.

Für ihre ununterbrochene Unterstützung in allen Belangen des Lebens gilt mein ganz persönlicher Dank meinen tollen Eltern Eva und Hans-Peter Hermann. Sie geben mir die Liebe und den Rückhalt, den man sich nur wünschen kann und ermöglichten es mir, meine Träume umzusetzen. Derselbe innige Dank gilt auch der besten Schwester der Welt: Svenja.

Kurzfassung

In der vorliegenden Arbeit wird die Anwendung von **hoch-parallelen Algorithmen** zur Auswertung von 3D-Punktwolken in der **Robotik** untersucht. Die Zielstellung ist es, essentielle Berechnungen der **Kollisionsprüfung und Planung** so weit zu beschleunigen, dass Roboter ein **reaktives Verhalten** erreichen und somit in einer **dynamischen, teils unbekannten Umgebung** eingesetzt werden können. Da die präsentierten Lösungen direkt auf **3D-Sensordaten** der Umgebung arbeiten, reduzieren sie die Abhängigkeit auf a priori bekannte geometrische Modelle, was das Einsatzgebiet erweitert. Die untersuchten Aufgabenstellungen decken alle drei Teilbereiche des traditionellen *Sense-Plan-Act*-Zyklus ab und reichen von Bewegungsprädiktion, inverser Kinematik, über Greif- und Trajektorienplanung bis zur Ausführungsüberwachung.

Aktuelle 3D-Kameras liefern detaillierte geometrische Umweltmodelle in Form von Punktwolken, welche jedoch hohe Anforderungen an den Datendurchsatz der verarbeitenden Algorithmen stellen. Um hier die benötigte Effizienz zu erreichen, liegt der erste Beitrag dieser Arbeit in der Entwicklung unterschiedlicher **diskretisierender Datenstrukturen auf Voxelbasis**, die eine **hoch parallele Interpretation** der Sensordaten auf **Grafikprozessoren** ermöglichen. Hervorzuheben ist hierbei der umgesetzte, lastbalancierte Octree.

Ausgehend von spezialisierten Techniken zur Kollisionsprüfung werden weitere Beiträge durch Softwarekomponenten geleistet, die es erlauben, unterschiedliche Planungs- und Überwachungsaufgaben zur Ausführungszeit (also während der Bewegung) zu lösen. Ein Kernaspekt dabei ist die Verwendung von **dichten Swept-Volumen** zur volumetrischen Darstellung von Bewegungstrajektorien. Im Gegensatz zur weit verbreiteten Dreiecksnetzdarstellung sind diese durch eine Voxeldarstellung sehr effizient zu erstellen und auf Kollisionen zu evaluieren.

Die im Rahmen dieser Arbeit entwickelten Lösungen wurden in einer OpenSource Softwarebibliothek zusammengeführt und sehr ausführlich in unterschiedlichen Szenarien praktisch evaluiert. Neben den Evaluationsergebnissen sind auch die dafür vom Autor entwickelten **Robotersysteme** Teil dieser Arbeit.

Inhaltsverzeichnis

Akronyme

AABB	Axis Aligned Bound Box.
API	Application Programming Interface.
BVH	Bounding-Volume-Hierarchie.
CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture.
DOF	Degree of freedom.
EDT	Euklidische Distanz Transformation.
EKF	Erweitertes Kalman Filter.
FCL	Flexible Collision Library.
FPS	Frames per Second.
GNAT	Geometric Near-neighbor Access Tree.
GP-GPU	General-purpose Graphics Processing Unit.
GPU	Graphics Processing Unit.
HAL	Hardware Abstraction Layer.
HPC	High-Performance Computing.
ICP	Iterative Closest Point.
MRK	Mensch-Roboter-Kollaboration.
OMPL	Open Motion Planning Library.
OOBB	Object Oriented Bound Box.
OpenGL	Open Graphics Library.
OpenMP	Open Multi-Processing Library.
PCL	Point Cloud Library.
PSO	Partikelschwarmoptimierung.
RANSAC	Random sample consensus.
RGBD	Red, Green, Blue, Depth.
ROS	Robot Operating System.
SBPL	Search-based Planning Library.
SLAM	Simultaneous Localization and Mapping.
SSV-ID	Sub-Swept-Volumen-Identifikator.
TCP	Tool Center Point.
TSDF	Truncated Signed Distance Function.
URDF	Unified Robot Description Format.
VBO	Vertex-Buffer Object.

Glossar

Adero (Advanced Dexterous Robot) ist ein mobiler, zweiarmiger Roboter, der vom Autor zu Testzwecken als Vorgänger von Immp entwickelt wurde.

Anytime Algorithmus ist ein iterativer, approximierender Algorithmus, der nach jedem Durchlauf valide Ergebnisse bereitstellt, und somit jederzeit abgebrochen werden kann. Mit zunehmenden Iterationen nähert sich das Ergebnis dem Optimum.

Baum-Invariante beschreibt den eindeutigen, gültigen Zustand eines gewurzelten Baumes, in dem alle inneren Knoten den ihren Kindknoten entsprechenden, zusammengefassten Zustand besitzen. Siehe Definition 14.

Block ist eine Einheit aus CUDA *Threads*, die synchron einen *Kernel* ausführt und sich einen gemeinsamen Speicher teilt.

CUDA ist eine Programmiertechnik mit einer gleichnamigen Laufzeitumgebung, mit der Algorithmen für GPUs compiliert und auf ihnen ausgeführt werden können. CUDA wird von Nvidia exklusiv für Grafikkarten der eigenen Marke entwickelt.

CUDA-Thread führt einen *Kernel* auf einer Recheneinheit (Core) der GP-GPU aus. CUDA Threads werden in *Blöcken* gestartet, in denen sie synchron ablaufen und einen gemeinsamen Speicher nutzen.

Device ist die physische und logische Einheit aus GP-GPU, ihrem Speicher, und allen weiteren Komponenten, die auf einer Grafikkarte verbaut sind. Da das *Device* nicht eigenständig nutzbar ist, benötigt es einen *Host*, also ein Computersystem, in dem es läuft.

Endeffektor ist das Werkzeug am Ende eines robotischen Armes, mit dem Objekte gegriffen oder manipuliert werden.

Erweitertes Kalman Filter ist eine nichtlineare Erweiterung des Kalman Filters, die es erlaubt, nichtlineare Systemmodelle zu schätzen.

Euklidische Distanz Transformation beschreibt eine Rechenvorschrift und ein Muster zur Anwendung auf die Elemente einer Datenstruktur. Dient zur Berechnung der euklidische Distanz zu einem anderen Element der Datenstruktur.

GP-GPU ist ein Grafikprozessor, der für generische Berechnungen genutzt werden kann.

GPU-Voxels ist die im Verlauf dieser Dissertation entwickelte Open Source Software"=Bibliothek, die hoch parallelisierte Algorithmen und Datenstrukturen zur Arbeit mit Voxeln zur Verfügung stellt.

Grid besteht aus mehreren CUDA Blöcken, die ohne Synchronisierung auf einer GP-GPU ausgeführt werden.

Grid-Stride-Loop sind `for`-Schleifen in CUDA *Kerneln*, die Arbeit im Kernel serialisieren, falls zu wenige *Threads* für eine vollständige Parallelisierung gestartet wurden. Siehe Algorithmus 1.

HoLLiE (House of Living Labs intelligent Escort) ist ein mobiler Roboter mit anthropomatischem Torso, der vom Autor entwickelt und in dieser Arbeit als Testplattform verwendet wurde.

Host ist das Computer-System, welches eine GP-GPU beinhaltet, und diese als untergeordnete Einheit (*Device*) nutzt, um CUDA Berechnungen auszuführen.

Immp (Industrial mobile manipulation plattform) ist ein mobiler, zweiarmiger Roboter, der vom Autor entwickelt und in dieser Arbeit als Testplattform verwendet wurde. Der Roboter verfügt über eine leistungsstarke on-board GP-GPU.

Kernel bezeichnet eine in sich abgeschlossenen Funktion, die in einem CUDA-Kontext auf der GP-GPU zur Ausführung gebracht wird. Die Parallelisierung des Kernels mit mehreren *Threads* kann über Aufrufparameter gesteuert werden.

Kinematische Konfiguration beschreibt die Anordnung aller beweglichen Achsen eines Roboters zueinander und somit seine Beweglichkeit. Nicht zu verwechseln mit: Roboterkonfiguration.

KinFu ist die OpenSource Implementierung von Kinect Fusion, einem Algorithmus zur Rekonstruktion von Oberflächenmodellen aus 3D-Punktwolken, die mit bewegten Sensoren aufgenommen werden.

Memory Coalescing beschreibt ein Speicher-Zugriffsmuster, bei dem mehrere Threads auf im Speicher hintereinander liegende Elemente zugreifen und somit den Speicherbus / Cache effizient nutzen .

Morton-Code ist die Linearisierung einer n-dimensionalen Adressierung, die bei einem n-ären Baum indirekt den Weg von der Wurzel zu einem Blatt beschreibt..

Octree ist eine 8-äre Baum-Datenstruktur mit einer einzelnen Wurzel. Jeder Knoten im Baum weist somit entweder acht direkte Nachfolger oder keine Nachfolger (Blattknoten) auf.

OpenGL ist eine Plattform- und Programmiersprachen-übergreifende Programmierschnittstelle zur Darstellung von 2D und 3D-Szenen. GPUs bieten eine Hardwarebeschleunigung für OpenGL .

Partikelschwarmoptimierung ist eine biologisch motivierte Herangehensweise um nichtlineare Optimierungsprobleme zu lösen. Ähnlich dem Schwarmverhalten in der Natur wird dabei eine Population von potentiellen Lösungen (Partikel) iterativ durch den Suchraum bewegt und über eine Bewertungsfunktion beurteilt.

Prefixsumme ist ein Verarbeitungsprimitiv der Parallelverarbeitung, bei dem über eine Folge von Eingabedaten iteriert wird und sukzessive ihre Partialsummen (oder das Ergebnis anderer binärer Operationen) gebildet werden. Auch *Scan* genannt. (s. Anhang).

Provider ist das Haupt-Programm in GPU-Voxels, das Berechnungen durchführt, und dessen Daten durch den Visualizer angezeigt werden können.

Raycasting beschreibt Verfahren zur Abtastung eines simulierten Licht- oder Sichtstrahles, um synthetische Bilder zu generieren oder Freiräume zu bestimmen.

Reduktion ist ein Verarbeitungsprimitiv der Parallelverarbeitung, bei dem eine Folge von Eingabedaten paarweise rekursiv zu einem einzigen Ergebniswert zusammengefasst wird (s. Anhang).

RGBD-Kamera ist eine Kombination aus zwei Sensoren in einem Gehäuse: Durch die gleichzeitige Auswertung der Daten einer Farbkamera (RGB: Red Green Blue) und eine Tiefenkamera (D: Depth) können eingefärbte 2,5 dimensionale Abbildungen einer Szene erstellt werden.

Roboterkonfiguration beschreibt den Zustand aller beweglichen Achsen eines Roboters und somit seine *Pose*. Nicht zu verwechseln mit: Kinematische Konfiguration.

Sense-Plan-Act-Zyklus beschreibt den typischen Ablauf von Roboterhandlungen: Sensorielle Wahrnehmung und Analyse der Daten. Basierend darauf eine Planung von Aktionen und letztendlich die Ausführung der Pläne. Die Ausführung geschieht somit *blind*.

Sub-Swept-Volumen-Identifikator ist ein Bitmuster, das in Voxeln gespeichert wird, um ihre Zugehörigkeit zu einer oder mehreren Entitäten zu kennzeichnen.

Swept-Volumen entspricht dem aufintegrierte Volumen im Raum, das von einem Objekt durch seine Bewegung überstrichen wird.

Vertex-Buffer Object ist ein Speicherbereich der in OpenGL genutzt wird, um Eckpunkte und andere Informationen über mehrere Zeichenaufrufe hinweg zu speichern. VBOs dienen zur Entkopplung der Daten-Bereitstellung und dem eigentlichen Zeichnen. Im GPU-Voxels-Visualizer liegen VBOs im CUDA Shared Memory.

Visual Servoing ist ein Kamera gestützter Regelungsprozess, bei dem ein Endeffektor relativ zu einem detektierten Objekt positioniert wird.

Visualizer ist ein eigenständiges Programm zu Visualisierung von Daten in GPU-Voxels. Es benötigt einen laufenden Provider, dessen Daten interpretiert werden.

Voxel bezeichnet ein kubisches Volumen im dreidimensionalen Raum, das die kleinste Einheit der in dieser Arbeit verwendeten Raumpartitionierung darstellt. Ein Voxel kann unterschiedliche Zustände annehmen und einer oder mehreren Entitäten zugeordnet werden. Er ist das Pendant des zweidimensionalen *Pixel*.

Voxel-Bedeutung entspricht der semantischen Interpretation von Voxeldaten. Hauptsächlich als Bitmuster in Bitvektor-Voxeln gespeichert, oder als Wahrscheinlichkeitsgrenze bei probabilisitischen Voxeln definiert.

Voxeltyp bezeichnet die Art der Implementierung eines Voxels. Je nach Typ können unterschiedliche Informationen pro Voxel gespeichert werden.

Voxelumwandlung bezeichnet das Einfügen einer Punktwolke in eine Voxel"=Datenstruktur, wobei die Belegtheitsinformationen der Voxel, in welchen die Punkte liegen, aktualisiert werden.

Warp bestehend aus 32 Threads. Ist die Menge an Threads, die ein Prozessorkern der GPU gleichzeitig ausführt.

Symbolverzeichnis

P bezeichnet eine Punktwolke, die aus den Punkten p_i besteht.

$\boxdot_{\ddagger}(V)$ ist der Operator, der einen Voxel V durch die Operation $\ddagger$ aktualisiert.

q beziffert die Anzahl an parallel ablaufenden Threads innerhalb eines Kernels.

l_{Voxel} beschreibt die Kantenlänge eines Voxel.

Ψ bezeichnet den Zustand eines Voxels, der über die voxeltypspezifische Nutzdaten repräsentiert wird.

$\ddagger$ ist der Operator, der bei einer Aktualisierung auf die Nutzdaten Ψ eines Voxels angewendet wird.

$\boxplus(M, P)$ ist der Operator, mit dem die Voxel-Datenstruktur M durch die Punktwolke P aktualisiert wird. Verwendet den $\boxdot_{\ddagger}(V)$-Operator.

$\blacksquare(V)$ ist der Operator, der als wahr ausgewertet wird, wenn der Voxel V als belegt zu interpretieren ist. Kann weitere, vom Voxeltyp abhängige Parameter aufweisen.

$\&$ ist der Operator, der zwei Voxel mittels $\blacksquare(V)$ auf Belegtheit überprüft und der als wahr ausgewertet wird, wenn beide Eingabevoxel belegt sind. Der Operator ist je nach Voxel-Datentyp und Semantik unterschiedlich implementiert und parametrisiert.

$\|$ ist der Operator, der zwei Voxel mittels $\blacksquare(V)$ auf Belegtheit überprüft und der als wahr ausgewertet wird, wenn einer der Eingabe-Voxel belegt ist. Der Operator ist je nach Voxel-Datentyp und Semantik unterschiedlich implementiert und parametrisiert. Im Octree bestimmt er den Status eines Elternknotens aus den Kindknoten.

$\cap$ ist der Operator, der zwei Voxel-Datenstrukturen überlagert und welcher die Menge der Voxel findet, die nach einem gegebenen Belegtheitskriterium in beiden Datenstrukturen belegt sind. Dafür werden paarweise alle Voxel mittels dem $\&$-Operator verglichen.

$\cup$ ist der Operator, der die Vereinigung zweier Punktwolken oder Voxel-Datenstrukturen (mittels $\|$ -Operator) bildet. Doppelte Einträge werden dabei entfernt.

V_O ist die Menge aller Voxel, die von einem Objekt O geschnitten werden.

Δt_{sv} ist die Zeitdauer, die einem Sub-Swept-Volumen-Identifikator (SSV-ID) entspricht.

$SE(2)$ ist eine spezielle euklidische Gruppe in der Ebene, die Translationen und Rotationen enthält (3 DOF).

$SE(3)$ ist eine spezielle euklidische Gruppe im dreidimensionalen Raum, die Translationen und Rotationen enthält (6 DOF).

1. Einführung

Nicht erst seit der Prägung des Begriffs ***„Roboter“*** durch Čapec streben Menschen nach einer intelligenten und universellen Maschine, die sie in möglichst allen Belangen des Lebens unterstützen kann. Ein unbändiger Enthusiasmus und Erfindungsreichtum führten zu immer gewagteren Vorhersagen, zu was und vor allem wann Roboter dazu in der Lage sein sollten. Doch der Durchbruch ist leider auch zu Zeiten des ***Deep Learnings*** noch immer nicht erreicht. Daher soll an dieser Stelle auf Zukunftsprognosen verzichtet und stattdessen die aktuellen Themen der praktischen Robotik kurz beschrieben werden, die diese Arbeit motivieren:

Das Einsatzspektrum von Robotern erweitert sich stetig. Während die Politik in einer ***Industrie 4.0*** [1] die Zukunft der (dann individualisierten) Produktion sieht, in der Maschinen untereinander und mit ihren Produkten in einem *Internet der Dinge* kommunizieren, spricht die Industrie momentan erst von der ***Dritten Revolution in der Robotik*** [2]. Diese bringt die ***mobile Manipulation*** in die Produktionsanlagen und soll so die nicht gewinnschöpfenden 25% der Produktionszeit, die aktuell für Materialfluss benötigt werden, eklatant verkürzen, indem eine Bearbeitung im Materialfluss möglich wird (frei nach Peter Klüger, KUKA Roboter GmbH, Juli 2015). Dafür müssen sich die Maschinen in ihrem Umfeld frei bewegen können, was insbesondere eine kollisionsfreie Bewegungsplanung in mehr oder minder unstrukturierter Umgebung erfordert. In der Forschung wird daher bereits seit vielen Jahren die vierte Revolution vorbereitet, welche die kognitiven Fähigkeiten der Roboter auf eine neue Ebene heben soll. Auch hier gilt es, die Herausforderungen, die bei der Verarbeitung der immensen Datenmengen entstehen, zu bewältigen.

Kollisionsfreie Bewegungsplanung und die für eine ***Arbeitsraumüberwachung*** erforderliche effiziente Datenverarbeitung bilden die Grundlage für eine sichere ***Koexistenz*** von Menschen und Robotern und erlaubt deren zaunlosen Betrieb. Doch erst die Kombination aus Leichtbaurobotik und inherenter bzw. intrinsischer Nachgiebigkeit ermöglicht eine sichere ***Kooperation*** zwischen Mensch und Maschine.

Die nun angestrebte industrielle und somit auch robotische Revolution zielt jedoch auf eine echte ***Kollaboration*** zwischen Mensch und Roboter, für die noch einige essentielle Technologien fehlen und zu welcher diese Arbeitet einen Beitrag leisten möchte.

[1] Vorangegangen waren die erste industrielle Revolution mit der Einführung dampfbetriebener, mechanischer Produktionsanlagen gegen Ende des 18. Jahrhunderts. Mit der Wende zum 20. Jahrhundert folgte die zweite industrielle Revolution, die eine Massenproduktion von Gütern mit Hilfe elektrischer Energie und Arbeitsteilung (Fordismus, Taylorismus) erlaubte. Ab Mitte der 70er Jahre und bis heute andauernd sorgt der Einsatz von Elektronik und IT in der Automatisierung von Produktionsprozessen für die dritte industrielle Revolution.

[2] Eine erste Revolution war die eigentlich Einführung der Robotik in der Automatisierungstechnik. Diese ging mit dem Einsatz von komplexer Sensorik in einer zweiten Revolution in die sichere, sensitive und adaptive Automatisierung über.

(a) Koexistenz (b) Kooperation (c) Kollaboration

Abb. 1.1.: Stufen der Zusammenarbeit zwischen Mensch und Roboter.

Klassische Sicherheitssysteme wie Lichtgitter, Laserscanner, kapazitive Annäherungssensoren, taktile Böden und Kraft- / Momentensensoren beschränken die Produktivität von Robotern enorm, da sie zum einen sehr große Schutzbereiche um die Maschinen erfordern und zum anderen, im Falle einer Verletzung dieser Bereiche, einen Stillstand des Roboters auslösen. Aus diesem Grund stoppen Roboter in Szenarien der Mensch-Roboter-Kollaboration (MRK) meist erst kontaktbasiert. Dies ist möglich, da sich die Maschinen so langsamen bewegen, dass sie bei einem physischen Kontakt mit einer Person schnell und sicher abgebremst werden können. Ein Kontakt mit dem Roboter ist jedoch weder angenehm für den Menschen noch ist dieser effizient, da er zu einem kompletten Stillstand und Wiederanlauf führt. Die Ansätze und Techniken aus der *DIN EN ISO 10218-1* und *DIN EN ISO 10218-2*, die die Anforderungen für inhärent sichere, kollaborative Industrieroboter festgelegen und die grundlegenden Gefährdungen und Risiken beschreiben, sind im Hinblick auf ihre Effizienz unzureichend.

(a) Optisch (b) Kapazitiv (c) Haptisch

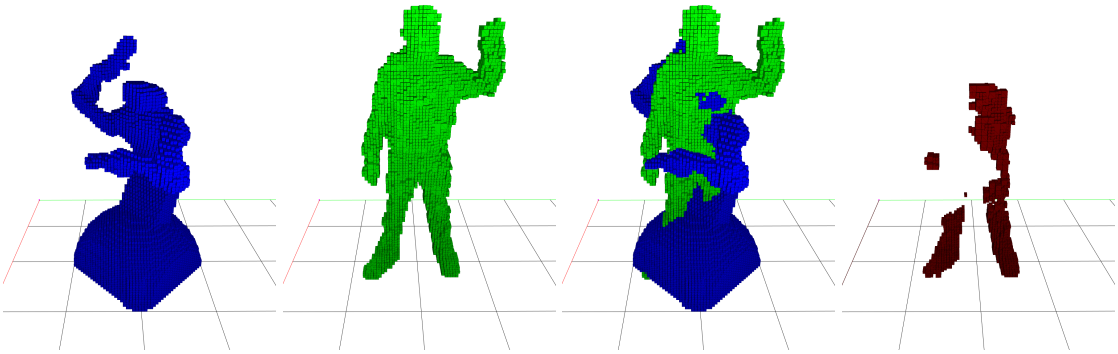
Abb. 1.2.: Aktuelle Kollisionserkennungsverfahren

Um eine ***höhere Effizienz*** in der im Bereich der MRK zu erreichen, sollten ***Kollisionen*** nicht nur erkannt, sondern ***antizipiert*** werden und so eine sinnvolle Behandlung ermöglichen. Für eine solche proaktive Kollisionsvermeidung ist eine ***visuelle Perzeption zur Planung und Überwachung*** von kollisionsfreien Bahnen ein vielversprechender Ansatz. Daher sind in dieser Arbeit ausschließlich Kollisionserkennungsverfahren von Interesse, die mit Daten aus einer visuellen 3D-Perzeption arbeiten. Nur sie erlauben es, Hindernisse bereits aus der Distanz detailliert und mit hoher Bildrate zu erkennen, um frühzeitig auf sie reagieren zu können, und somit physische Kollisionen zu verhindern. Dies macht den Unterschied zwischen einem einfachen Industrieroboter und dem kollaborierenden Assistenten (CoBot) aus und kann den Wunsch nach sicheren, geteilten Mensch-Roboter-Arbeitsräumen erfüllen.

Durch ***reaktive Verhalten***, die geplante Bewegungen in ***dynamischen Umgebungen*** anpassen können, steigt nicht nur die ***Effizienz*** eines Systems, sondern auch das vermittelte ***Sicherheitsgefühl***. Die somit geschaffene höhere ***Akzeptanz*** erhöht letztendlich die ***Ergonomie*** eines geteilten Arbeitsplatzes für den Mitarbeiter und sollte ein langfristiges Ziel der Robotik darstellen.

1.1. Kurzfassung

Um in einer teilweise unbekannten Umgebung eine reaktive Planung auf Basis von 3D-Sensordaten zu ermöglichen, wird in dieser Arbeit die *GPU-Voxels* Softwarebibliothek entworfen. Sie basiert auf einer Kollisionsdetektion durch die hoch parallele Überlagerung zweier Voxel-Datenstrukturen im Graphics Processing Unit (GPU) Speicher. Der klassische Fall ist in Abb. 1.3 zu sehen: Dabei enthält eine der Datenstrukturen das Umweltmodell aus 3D-Sensordaten, während die zweite ein Egomodell des Roboters und seiner Bewegungen beinhaltet. Eine leere Schnittmenge als Ergebnis der Überlagerung bedeutet dabei Kollisionsfreiheit, wohingegen eine nicht leere Menge das in Kollision liegende Volumen repräsentiert. Unterschiedliche Datenstrukturen und darauf abgestimmte Methoden zur Bestimmung ihrer Schnittmengen bzw. Befüllung werden untersucht. Aufbauend auf der GPU-Kollisionsdetektion können dann unterschiedliche, spezialisierte Algorithmen zur Bewegungsplanung entwickelt werden.



(a) Egomodell (b) Umweltmodell (c) = (a) ∪ (b) (d) = (a) ∩ (b)

Abb. 1.3.: Zwei Voxel-Datenstrukturen und das Ergebnis des Vereinigungs- und Schnitt-Operators zur Kollisionsdetektion.

1.2. Begriffsbildung

Zu besseren Einordnung der vorliegenden Arbeit sollen zunächst die wichtigsten Begrifflichkeiten definiert werden:

Begriffsbildung Kollisionsdetektion: Flexibel einsetzbare Roboter müssen in der Lage sein, sich kollisionsfrei in ihrer Umwelt zu bewegen. Um Kollisionen verhindern zu können, müssen diese zunächst auf Basis von simulierten Situationen erkannt werden.

Hierbei werden Momentaufnahmen einer Szene, bestehend aus Umwelt und Roboter miteinander überlagert. Die Kollisionserkennung findet also auf statischen Daten statt und kann folglich auch nur Aussagen über einzelne Zeitpunkte bzw. Konstellationen in der Szene liefern. Dabei können auch Umweltdaten aus Sensoren berücksichtigt werden.

Definition 1. Die **Kollisionserkennung** oder **Kollisionsdetektion** ermittelt, ob sich zwei Entitäten zur selben Zeit am selben Ort befinden und somit eine Kollision vorliegt. Kollisionen können auch Eigenkollisionen einer Entität mit sich selbst beinhalten. In dieser Arbeit geschieht die Kollisionserkennung innerhalb von Simulationen und bezieht sich somit nicht auf die Detektion physischer Kontakte.

Begriffsbildung Kollisionsvermeidung: Da Roboter ihre auszuführenden Bewegungen im Voraus exakt kennen, kann die Simulation der Kollisionserkennung auch zukünftige Zustände des Roboters auf Zusammenstöße mit dem aktuellen Zustand der Umwelt überprüfen. Somit lassen sich bereits vor der Ausführung Probleme mit statischen Hindernissen erkennen und vermeiden, indem der Roboter rechtzeitig gestoppt wird. Neben der binären Entscheidung über eine Kollision können zusätzlich auch Distanzen zu Objekten oder Kollisionswahrscheinlichkeiten betrachtet werden, die bei der Vermeidung von Kollisionen helfen.

Definition 2. Bewegt sich ein Roboter in einer veränderlichen Umgebung oder ist sein Umweltwissen nicht vollständig, sollte er über Strategien zur **Kollisionsvermeidung** verfügen. Sie ermöglichen es, auf Basis eines Modells der eigenen Bewegungsbahn (Egotrajektorie) noch vor Eintritt einer Kollision Gegenmaßnahmen einzuleiten. Zur Kollisionsvermeidung zählt auch die Minimierung der Kollisionswahrscheinlichkeit, beispielsweise über die Maximierung des Abstandes zu Hindernissen.

Begriffsbildung Bewegungsplanung: Durch die Evaluierung vieler hypothetischer Zustände oder Bewegungen kann ein Roboter eine Bewegungsabfolge generieren, die ihn an Hindernissen vorbei und zu einem konkreten Ziel führen.

Definition 3. In der Robotik stützt sich die **Bewegungsplanung** auf die Kollisionsdetektion, um simulierte Bewegungen zu evaluieren. Das Ziel ist es, einen Agenten von einem Start-Zustand in einen Ziel-Zustand zu überführen, ohne dabei Kollisionen hervorzurufen. Mehr als 90% der Berechnungszeit für die Planung einer Bewegung in realistischen Umgebungen sind dabei der Kollisionsprüfung geschuldet [46]. Bewegungen können weiterhin nach unterschiedlichen *weichen* Kriterien optimiert werden, wobei jedoch immer das *harte* Kriterium der Kollisionsfreiheit gewährleistet sein muss.

Begriffsbildung Kollisionsprädiktion: Da sich Bewegungsplanung und Kollisionserkennung zunächst auf statische Momentaufnahmen der Umwelt stützen, erweist sich dieses Vorgehen in dynamischen Szenen jedoch nur bedingt als geeignet, da Kollisionen erst erkannt werden, wenn sich Hindernisse bereits in der Bewegungsbahn des Roboters befinden. Verfügt der Roboter jedoch über Informationen zu seiner eigenen Dynamik und zu derjenigen des Hindernisses, kann er Kollisionen vorhersagen, die mit einer gewissen Wahrscheinlichkeit zu einem späteren Zeitpunkt auftreten werden und proaktiv darauf eingehen.

Definition 4. Wurde eine Bewegung eines Hindernisses über eine gewisse Zeit beobachtet, kann eine **Kollisionsprädiktion** stattfinden. Diese detektiert und verfolgt dynamische Entitäten und nutzt ein Bewegungsmodell, um ihre zukünftige Trajektorie vorherzusagen. Überschneidet sich die geplante Egotrajektorie mit der vorhergesagten Hindernistrajektorie, kann eine Kollision prädiziert werden.

Begriffsbildung reaktives Verhalten: Weiterhin ist es in dynamischen Umgebungen unabdingbar, schnell zu planen, beziehungsweise Pläne dynamisch anzupassen, da diese in einem typischen Sense-Plan-Act System sonst korrumpiert sein können, bevor sie überhaupt zur Ausführung gebracht werden. Rein reaktive Systeme hingegen können keine global optimierten Lösungen liefern und sich in lokalen Minima verfangen. Daher werden Planer meist mit reaktiven Ansätzen kombiniert. Je weiter dabei die Grenze zwischen deliberativer Planung und reaktiver Ausführung in Richtung einer onlinefähigen Planung verschoben wird, desto deterministischer fallen die Ergebnisse aus.

Definition 5. Sind die Verfahren der Kollisionserkennung und Kollisionsvermeidung an die Dynamik der Umgebung angepasst, kann ein Roboter ein **reaktives Verhalten** bei gleichzeitig vollem Determinismus zeigen. Im optimalen Fall ist seine Planung schnell genug, um mit Veränderungen in der Umwelt umzugehen, ohne dafür seine Bewegung zu stoppen.

1.3. Zielsetzung und Problemstellung

Mit den definierten Begriffen kann nun die Zielsetzung dieser Arbeit beschrieben werden:

Forschungsziele. In der Arbeit sollen **Planungsysteme** entwickelt werden, welche es erlauben, einen Roboter in einer **dynamischen, teils unbekannten Umgebung** einzusetzen. Voraussetzung hierfür ist eine **hoch performante Kollisionsprüfung** auf Basis von 3D-Sensordaten, die dem Roboter eine **reaktive Planung** ermöglicht. Dafür sollen ermittelte Bewegungspläne antizipierend auf Kollision mit dynamischen Hindernissen überwacht werden. Bei Bedarf sind die Pläne an wechselnde Umgebungsbedingungen zu **adaptieren**, indem kritische Teile **umgeplant** werden, oder auf **alternative Pläne** ausgewichen wird, ohne dafür die Ausführung zu stoppen.

Aus diesen Zielen lassen sich mehrere Problemstellungen ableiten: Zunächst wird eine passende Perzeption benötigt, um die Umwelt wahrzunehmen. Diese muss zwangsläufig dreidimensionale Daten liefern, da sich Mensch und Roboter im Raum bewegen. Auch die Rate, mit der diese Informationen aktualisiert werden, muss adäquat sein, um Änderungen der Umwelt rechtzeitig erkennen zu können. Eine Sensorik, die diese Anforderungen erfüllt, ist in Form moderner 3D-Kameras vorhanden und kann daher als gegeben angenommen werden. Die zu verarbeitende Informationsmenge aus der 3D-Perzeption ist jedoch so umfangreich, dass ihre sequentielle Verarbeitung auf aktueller Hardware an Durchsatzgrenzen stößt. Daher verwendet diese Arbeit Parallelprozessoren mit einem sehr hohem Parallelisierungsgrad und beschäftigt sich folglich mit den Fragen:

Forschungsfrage 1. Welche Algorithmen der Verarbeitungskette zur Interpretation von 3D-Daten können effizient parallel ablaufen?

Weiterhin müssen die gewonnenen Daten für ihre Interpretation in eine Repräsentation umgewandelt werden, die den Anforderungen unterschiedlicher Kollisionsprüfungsverfahren gerecht wird, und die einen parallelen Zugriff ermöglicht. Hierfür hat sich die Klasse der raumpartitionierenden Repräsentationen als geeignet erwiesen, deren Vor- und Nachteile gegenüber etablierten Modellen auf Basis von Dreiecksnetzen aufgezeigt werden sollen.

Forschungsfrage 2. Welche Vor- und Nachteile haben raumpartitionierende Repräsentationen gegenüber Oberflächenmodellen bei der Verarbeitung von 3D-Daten zur Kollisionserkennung?

Nach der Beantwortung dieser Frage sollen die spezifischen Eigenschaften mehrerer konkreter Datenstrukturen untersucht werden, um Ego- und Umwelt-Modell je nach Anwendungsszenario passend abbilden zu können. Hierbei müssen Speicherverbrauch gegen Berechnungsgeschwindigkeit entsprechend der zu modellierenden Daten aufgewogen werden. Weiterhin muss die Datenstruktur es ermöglichen, **örtliche, zeitliche und semantische Informationen** zu speichern.

Forschungsfrage 3. Welche Datenstrukturen, die einen parallelen Zugriff erlauben, eignen sich zur Speicherung dynamischer raumpartitionierender 3D-Repräsentationen? Welche Vorteile können bei der Kollisionserkennung in unterschiedlichen Anwendungszenarien aus ihren spezifischen Eigenschaften gezogen werden?

Aufbauend auf den Datenstrukturen und dazu passenden Algorithmen sollen schließlich Planungsverfahren entwickelt werden:

Forschungsfrage 4. Welche Verfahren zur Bewegungsplanung können von einer hochparallelen Kollisionserkennung profitieren? Welche Vorteile entstehen durch die Verwendung von Swept-Volumen zur Abbildung von Bewegungen?

Am Ende der Arbeit wird die Praxisrelevanz der entwickelten Lösungen hinterfragt. Die Evaluation von General-purpose Graphics Processing Unit (GP-GPU)s in der Robotik wird dabei an zahlreichen Beispielen durchgeführt, die in Abb. 1.4 in einen typischen Sense-Plan-Act-Zyklus eingeordnet sind.

Forschungsfrage 5. Erlaubt der Einsatz der entwickelten hochparallelen Verfahren die Planung von Roboterbewegungen in dynamischen Umgebungen? Auf welche anderen typischen Robotik-Probleme lassen sich die Ergebnisse dieser Arbeit anwenden?

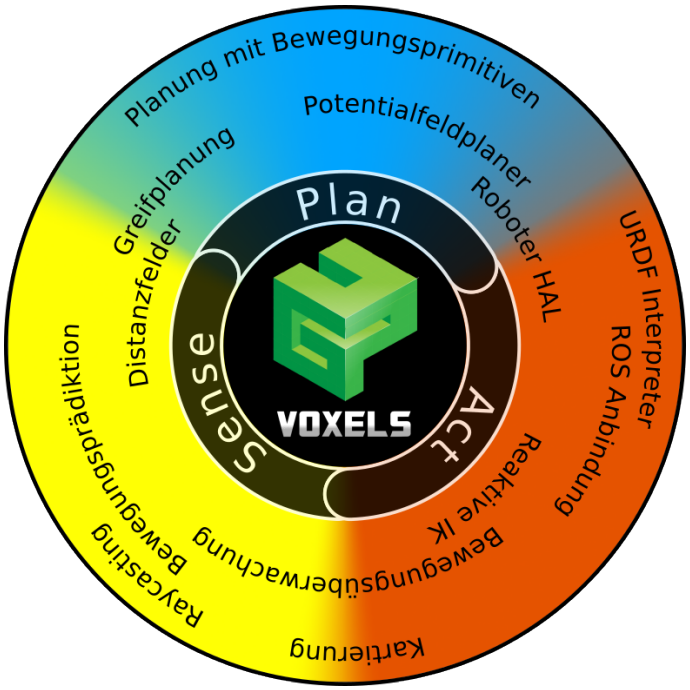


Abb. 1.4.: Einordnung der bearbeiten Themengebiete in den Sense-Plan-Act-Zyklus. GPU-Voxels stellt die Kernalgorithmen, Datenstrukturen und Werkzeuge wie eine Visualisierung für alle Aufgaben bereit.

1.4. Einordnung und Wissenschaftlicher Beitrag

Die Generierung von kollisionsfreien Bewegungen ist in der Robotik seit jeher eine fundamentale Herausforderung. Entsprechend groß ist das Feld verwandter Arbeiten in den Bereichen Kollisionsprüfung, Planung und reaktiver Verfahren, auf die im folgenden Kapitel eingegangen wird. Während die Entwicklung der Planungsverfahren rasante Fortschritte durchlaufen hat, ergaben sich bei der Modellierung der Umwelt und den darauf aufbauenden Verfahren zur Kollisionserkennung wesentlich weniger Änderungen. Der dominierende Anteil an Kollisionsprüfungsverfahren arbeitet auf Dreiecksnetzen und Hierarchien aus Hüllkörpern, deren Generierung aus Sensor-Punktwolken einen erheblichen Berechnungsaufwand erfordert. Somit werden diese Verfahren bevorzugt im Zusammenhang mit a priori bekannten Modellen eingesetzt, deren Geometrien statisch sind und die zur Laufzeit lediglich ihre Pose ändern.

Im Gegenzug herrscht ein Defizit bei der Generierung von Bewegungen in Umgebungen, die erst während der Aktivität des Roboters sensorisch erfasst werden: Bei der Verarbeitung von Punktwolkendaten mit Hilfe von **diskretisierenden Datenstrukturen** reduzieren viele Verfahren aus dem aktuellen Stand der Technik die Problemstellung auf 2 oder 2,5 Dimensionen, um hochauflösende Sensordaten schritthaltend verarbeiten zu können.

Verfahren, die dagegen **dreidimensionale Belegtheitskarten** erstellen, müssen die anfallenden Datenmengen entweder in ihrer räumlichen oder zeitlichen Auflösung beschränken, um echtzeitfähig zu sein. Daraus leitet sich der direkte Bedarf nach neuen, parallelisierten Herangehensweisen ab, um nicht nur in robotischen Anwendungen performanter mit hochauflösenden Punktwolken zu arbeiten und Kollisionen zwischen ihnen zu bestimmen. Zwar ist die Verarbeitung von a priori gegebenen **Volumendaten auf Parallelprozessoren** weit verbreitet, allerdings existieren nur sehr wenige Arbeiten, die speichereffiziente, volumetrische Modelle zur Laufzeit aufbauen, da die benötigten dynamischen Datenstrukturen eine Herausforderung auf solcher Hardware darstellen. Weiterhin existieren auch nur wenige Arbeiten, die Swept-Volumen heranziehen, um die Planung und Ausführungsüberwachung zu verbessern, da die Generierung von Sweeps auf Basis von Dreiecksnetzen sehr rechenintensiv ist, und sich daher nicht für Echtzeitverarbeitung eignet. Ein weiterer Schwachpunkt aktueller Planungsverfahren ist die Nutzung unterschiedlicher Modelle für die **Trajektorienplanung** und die **Ausführungsüberwachung**. Um hier fehleranfällige Redundanzen zu vermeiden, sollte nur ein einziges einheitliches Modell in allen drei Phasen des Sense-Plan-Act-Zyklus zum Tragen kommen.

Die vorliegende Arbeit setzt an allen gelisteten Problemen an, indem sie eine Voxelmodellierung auf GP-GPUs realisiert, die sowohl für die Trajektorienplanung als auch für die Ausführungsüberwachung nutzbar ist. Weiterhin wandelt sie etablierte Algorithmen der Robotik, insbesondere Planungsverfahren so ab, dass sie das volle Potential der parallelisierten Kollisionsprüfung ausschöpfen. Alle weiteren abgedeckten Themengebiete finden sich in Abb. 1.4.

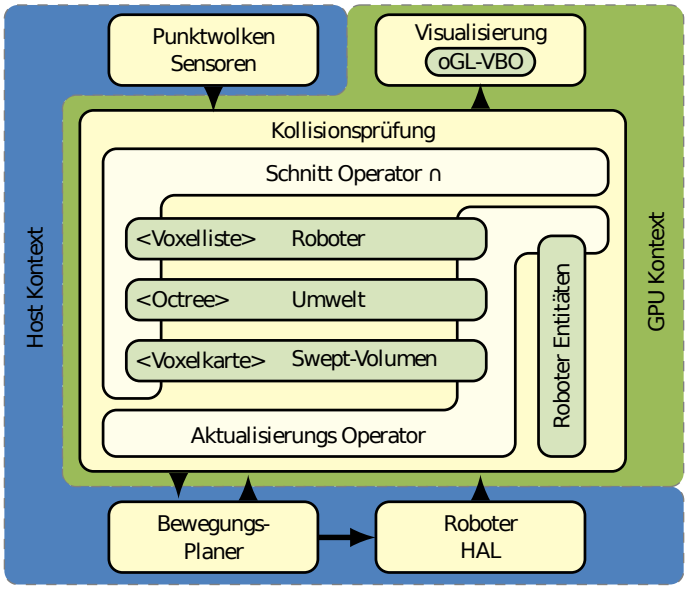


Abb. 1.5.: Übersicht über die grundlegenden Software-Module, aufgeteilt in GPU (grün) und Host (blau) Komponenten. Daten, die im GPU-RAM liegen sind hellgrün dargestellt.

Bei einer Einordnung anhand der NASREM-Architektur [35] bewegt sich die Arbeit auf der mittleren Ebene: Unterhalb eines Missionsplaners und oberhalb der ausführenden Schichten. Die Schwerpunkte liegen auf den beiden Säulen der Sensorik und des Weltmodells, wie auch in Abb. 1.5 zu erkennen ist. Symbolische Planung, Regelungsverfahren

und verhaltensbasierte Ansätze sind nicht Teil der Arbeit.

Der wissenschaftliche Beitrag stützt sich auf die Verknüpfung von Volumenrepräsentationen, die bereits seit mehreren Jahrzehnten untersucht werden, mit modernen Methoden der heterogenen Parallelverarbeitung auf CPUs und GPUs. Ziel ist die schritthaltende Kollisionsprüfung und Bewegungsplanung, die anhand von drei Teilzielen evaluiert wird (vgl. Begriffsbildung):

1. Zunächst ist die Parallelisierbarkeit verschiedener Datenstrukturen und ihre Eignung zur Modellierung unterschiedlicher Entitäten der Bewegungsplanung zu prüfen. Hierfür wird eine Unabhängigkeit zwischen den einzelnen Zellen der diskretisierenden Modelle angenommen, um diese datenparallel verarbeiten zu können. Aufbauend darauf soll eine **Kollisionsprüfung** von Momentaufnahmen durchgeführt werden. Der Erfolg wird anhand der Laufzeit der Verfahren gemessen und mit etablierten Algorithmen verglichen. Es soll mindestens eine Verarbeitungsrate erreicht werden, die der Bildrate aktueller 3D-Sensoren entspricht.

2. In einem zweiten Schritt soll eine **Kollisionsvermeidung** in Form einer Bewegungsplanung umgesetzt werden. Ziel ist es, die geplanten Robotertrajektorien während der Ausführung kontinuierlich auf Kollision mit Momentaufnahmen einer dynamischen Umwelt zu prüfen. Es soll ein überwachter Korridor aus Swept-Volumen entstehen, in dem sich der Roboter sicher bewegen kann. Untersucht werden daher Planungsverfahren, die nicht nur einzelne Posen, sondern ganze Bewegungsabläufe nutzen, um aus diesen komplexe, kollisionsfreie Bahnen zu synthetisieren. Zur Evaluation werden verwandte Arbeiten herangezogen und zahlreiche Experimente durchgeführt.

3. Um die Planung zu einer **Kollisionsprädiktion** zu erweitern, werden schließlich nicht nur die Eigenbewegungen, sondern auch die Trajektorien dynamischer Hindernisse als Swept-Volumen modelliert. Dies erlaubt proaktiv auf dynamische Umgebungen zu reagieren. Um die Unsicherheiten der Prädiktion abzubilden werden weiterhin Distanzfelder auf Voxelkarten definiert, über die ein variabler Sicherheitsabstand um Hindernisse herum gewahrt werden kann. Somit ist neben einer exakten Kollisionsberechnung auch die Distanzberechnung zwischen zwei Modellen umgesetzt, die für zahlreiche Planungsalgorithmen benötigt wird. Beispielsweise können Distanzkarten als Potentialfeld interpretiert werden, um entlang dessen Pfade zu planen. Auch hier dienen vorhandene Arbeiten praktische Versuche zur Bewertung der erzielten Ergebnisse.

Technisch soll der Einsatz von General Purpose **GPUs für die Robotik** vorangebracht werden, die eine Echtzeitverarbeitung von umfangreichen Sensordaten erlauben und somit reaktivere Verhalten ermöglichen. Hierfür wird untersucht, in wieweit sich traditionelle Verfahren der Robotik auf GPUs übertragen lassen, da hier der Einsatz einer dynamischen Speicherverwaltung äußerst negative Auswirkungen auf den Datendurchsatz hat. Ein sehr praktischer Beitrag ist die Zusammenstellung und Veröffentlichung aller Softwarekomponenten in Form einer **Open-Source Bibliothek**, die in das sehr stark verbreitete *Robot Operation System* (ROS) integrierbar ist.

1.5. Aufbau der Arbeit

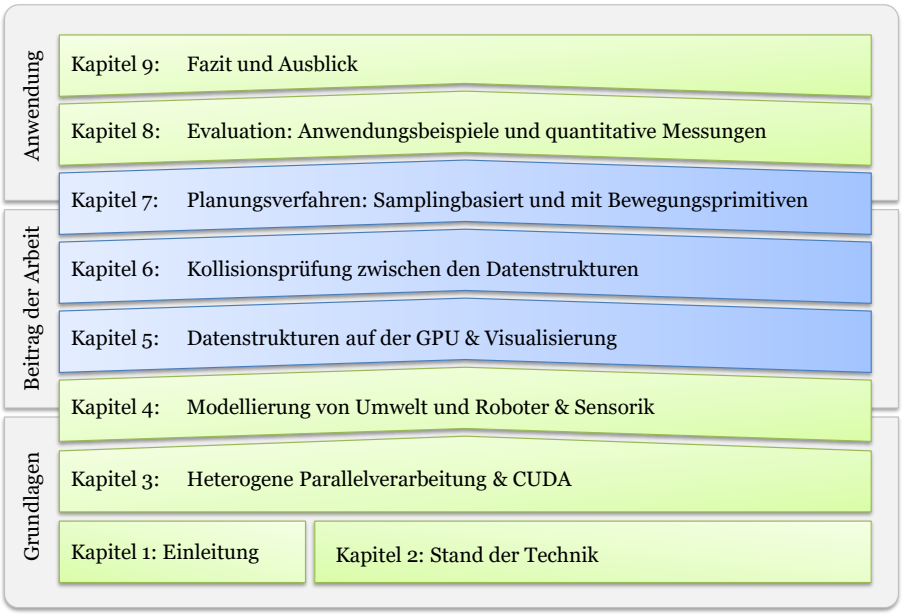


Abb. 1.6.: Übersicht über die aufeinander aufbauenden Kapitel.

Nach dieser Einführung und der Vorstellung des aktuellen Forschungsstandes in **Kapitel 2** folgt der weitere Aufbau der Arbeit den logischen Abhängigkeiten der umgesetzten Verarbeitungskette. Zu Beginn werden in **Kapitel 3** die Grundlagen der heterogenen Datenverarbeitung eingeführt und die Prinzipien der Programmierung in CUDA erläutert. **Kapitel 4** behandelt dann das Thema Sensorik und die unterschiedlichen Modelle die aus den Sensordaten gewonnen werden können. Den Schwerpunkt bilden die verwendeten Voxelmodelle zur Darstellung der Umwelt und des Roboters, sowie Swept-Volumen zur Repräsentation von Bewegungen. Auch die Prädiktion von Bewegungen auf Basis der Sensordaten findet sich in diesem Kapitel. Daran schließt sich in **Kapitel 5** die Implementierung von Datenstrukturen an, mit denen die Voxelmodelle auf der GPU effizient repräsentiert werden können. Detailliert wird hierbei auf den GPU-Octree eingegangen, da dieser wissenschaftlich und praktisch von großer Bedeutung ist. Aufgrund der implementierungstechnischen Nähe ist hier auch das Vorgehen zur Visualisierung aller Datenstrukturen erörtert, die über einen geteilten Speicher erfolgt.

Nachdem alle nötigen Bestandteile definiert sind, kann der Kern der Arbeit folgen: **Kapitel 6** setzt sich mit der eigentlichen parallelen Kollisionsprüfung zwischen den Datenstrukturen auseinander, auf denen die restlichen Kapitel aufbauen. Insbesondere **Kapitel 7**, das mehrere Planungsverfahren untersucht, die von den Vorteilen der GPU-basierten Kollisionsprüfung profitieren.

Die umfangreiche experimentelle Evaluation in **Kapitel 8** belegt dann die praktische Einsetzbarkeit der entwickelten Verfahren in unterschiedlichen Szenarien und Anwendungen. Ein abschließendes Fazit bzw. einen Ausblick gibt **Kapitel 9**. Im Anhang sind letztendlich mathematische Definitionen und technische Details der Implementierung zusammengestellt.

2. Stand der Technik

Wie bereits beschrieben, stellt die Bahnplanung und somit auch die Kollisionsprüfung ein fundamentales Problem der Robotik dar. Aus den anfänglichen Fragestellungen zur Bestimmung von Trajektorien für punktförmige Roboter in zweidimensionalen Umgebungen [97] wurden schnell Planungssysteme, die eine Vielzahl von Freiheitsgraden in Kombination mit einer dreidimensionalen Umgebung handhaben konnten [116]. Dies ist eine Voraussetzung für die mobile Manipulation, also die Kombination eines Manipulationsroboters mit einer mobilen Plattform, die bereits seit beinahe dreißig Jahren erforscht wird [170]. Ziel eines mobilen Roboters ist die Erweiterung seines Arbeitsraumes und eine Steigerung der Flexibilität, die durch die Mobilität gewonnen wird. Diese bringt jedoch auch neue Schwierigkeiten für die Planung mit sich, da der Arbeitsraum eines mobilen Roboters im Allgemeinen nicht komplett einsehbar ist, weshalb auch Hindernisse oder Probleme eventuell erst während der Ausführung einer Aufgabe erkannt werden können. Roboter, die in einer unstrukturierten, dynamischen Umwelt agieren sollen, müssen somit nicht nur auf bekannten Modellen planen, sondern auch auf Sensordaten ihrer Umgebung – einerseits, um Kollisionen zu vermeiden, andererseits aber auch, um physikalische Interaktionen planen zu können.

Erste Systeme zur mobilen Manipulation bearbeiteten die Planung in zwei unabhängigen Teilaufgaben: Zunächst wurde ein Pfad für die mobile Plattform gesucht, um dann in nächster Nähe eines Zielobjektes die eigentlich Manipulationsaufgabe des Roboterarms zu planen. Folgende Arbeiten versuchten die beiden Teilprobleme immer enger zu koppeln, bis schließlich erste Planer leistungsfähig genug waren, um alle Freiheitsgrade gleichzeitig zu berücksichtigen und zu planen.

Die Ansätze folgten zunächst dem klassischen deliberativen Sense-Plan-Act-Zyklus einer NASREM Architektur [35] und konnten einmal berechnete Pläne nur sehr eingeschränkt an neu gewonnene Umweltinformationen anpassen. Bedingt durch die Planung in einem hochdimensionalen Zustandsraum mit zahlreichen Randbedingungen und über größere Distanzen hinweg, lagen zwischen dem Eintreffen eines Auftrages und dem Start der Ausführung durch den Roboter lange Stillstandszeiten. Wurde dann während der Ausführung ein Problem festgestellt, musste der Roboter gestoppt werden und der Planungsprozess begann von neuem. Ein Paradigmenwechsel zur reaktiven Robotik löste dieses Problem auf lokaler Ebene in der direkten Umgebung des Roboters. Eine globale Planung war damit jedoch nicht mehr möglich, was in komplexeren Problemstellungen zu inkonsistenten Entscheidungen und damit zum Scheitern führen konnte. Daher finden sich in modernen Systemen meist Kombinationen aus globalen Planern mit reaktiven Komponenten, die bei der Ausführung in gewissem Maße von einer geplanten Trajektorie abweichen können, um Hindernissen auszuweichen (*Elastic Bands and Strips* [165, 52]). Zur Verkürzung der Berechnungszeiten verflechten neuere Ansätze Planungs- und Ausführungsvorgänge (*Interleaved Planning and Execution* [153]). Darüber hinaus existieren Planer, welche ihre Ziele und Zwischenziele abstrakter und somit variabler definieren

(*Task-Space-Regions* [45]) können, was zur Ausführungszeit mehr Spielraum zur Anpassung eines Planes lässt. Kapitel 7 stellt die wichtigsten Klassen von Planungsalgorithmen ausführlicher vor. Alle Verfahren profitieren von einer immer weiter gesteigerten Rechenkapazität und erlauben somit die Lösung komplexer Planungsprobleme zur Ausführungszeit. Auch die vorliegende Arbeit nutzt moderne parallele Hardware in Form von GPUs und hat sich zum Ziel gesetzt, die Planung global konsistenter Lösungen so weit zu beschleunigen, dass in vielen Szenarien auf eine reaktive Komponente verzichtet werden kann. Dabei nimmt die Umweltmodellierung und die darauf aufbauende Kollisionsprüfung eine Schlüsselfunktion ein.

Erste Kollisionsprüfungen für den 3D-Raum basierten auf einer Diskretisierung des Arbeitsraumvolumens in gleichmäßige würfelförmige Abschnitte, so genannte ***Voxel***. Diese fanden seit den 80er Jahren einen Einsatz für die grafische Darstellung von Volumenobjekten, insbesondere für Bildgebende Verfahren der Medizin. Vorreiter war hier Kaufman und Bakalash [111, 112]. Bereits 1989 konnte eine hardwarebeschleunigte Echtzeit-Kollisionsdetektion auf Basis von Voxeln durch Duffy et al. [72] umgesetzt werden. Belegen in einer Voxelrepräsentation mehrere Entitäten denselben Voxel, so liegt eine Kollision zwischen ihnen vor [105]. Für Planungsaufgaben konnten darüber hinaus Potentialfelder um belegte Voxel erzeugt werden, die einen Sicherheitsabstand für Agenten definierten. So war Kitamura bereits 1995 in der Lage, eine Trajektorie für einen beliebig geformten Roboter durch eine 3D-Umgebung zu berechnen [120].

Aufkommende Anforderungen nach realistischeren Kollisionstests, die auch die physikalische Interaktion von Modellen simulieren konnten, verdrängten ab 1990 die Voxelverfahren. Hierfür wurde auf eine Oberflächennetz-Modellierung von Objekten mittels Dreiecken zurückgegriffen, wie sie aus der Computergrafik bekannt war. Um dabei nicht jedes einzelne Dreieck berücksichtigen zu müssen, wurden die Modelle zusätzlich in Hüllkörper-Geometrien eingeschlossen, die als geschlossener mathematischer Ausdruck darstellbar sind. Somit lässt sich schnell berechnen, ob sich diese einfachen geometrischen Primitive überlappen und somit eine potenzielle Kollision vorliegt. Nur in diesem Fall sind alle Dreiecke innerhalb des Primitivs detailliert zu überprüfen. Werden mehrere Ebenen dieser Diskretisierung ineinander geschachtelt, spricht man von Bounding-Volume-Hierarchies (BVHs). An diesen Techniken änderte sich über mehrere Jahre nichts grundlegendes, allerdings konnten sie extrem beschleunigt werden, indem sie teilweise auf GP-GPUs portiert wurde. Durch die Verfügbarkeit dieser leistungsfähigen Parallelprozessoren entstanden viele Arbeiten auf dem Gebiet der Kollisionserkennung, die in Kapitel 6 detaillierter vorgestellt werden. Die meisten dieser Verfahren sind jedoch nicht darauf ausgelegt, Punktwolken aus 3D-Kameras zu verarbeiten. Um dennoch mit Sensordaten arbeiten zu können, ist es ein verbreiteter Ansatz, die Messpunkte mit hohem Aufwand zu triangulieren und das entstehende Oberflächennetz genau wie andere, a priori bekannte, Modelle handzuhaben. Dies schränkt die erreichbare Wiederholungsrate der Kollisionsprüfung bzw. den Detailgrad ihrer Modelle ein und erfordert starke Prämissen für den Umgang mit Verschattungen und Sichtkanten. Einen probabilistische Umsetzung, bei der die Durchdringung zweier Punktwolken als Klassifikationsproblem modelliert wird, verfolgt die FCL Kollisionserkennungsbibliothek [157].

Die Verbindung von Punktwolkendaten mit kubischen Volumen in Form des Voxmap-Pointshell Algorithmus von McNeely [142] stellt 1999 ein erstes Revival der Voxeltechnologien dar. Seine Lösung, die zunächst für ein haptisches Rendering von CAD Da-

ten entwickelt wurde, konnte ab 2007 auch teilweise in die Robotik übertragen werden [171, 177]. Das Verfahren erlaubt schnelle und vor allem zeitkonstante Kollisionsprüfungsintervalle, womit im Gegensatz zu Dreiecksnetz-Modellen eine Echtzeitfähigkeit nachweisbar ist.

Auch in der Welt der Spiele wurden Voxel ab 2008 wieder ein populäres Thema, nachdem id-Software erste Techologiedemonstrationen seiner Voxelengine veröffentlichte [155], die eine verformbare Umwelt erlaubte. Aufgrund technischer Herausforderungen dauerte es jedoch bis 2016, bis das erste erfolgreiche Spiel (*DOOM*) mit einer Voxel-Engine (*id tech 6*) vermarktet wurde.

Um das stark eingeschränkte repräsentierbare Arbeitsvolumen von Voxelkarten zu erweitern, folgten viele Arbeiten, die sich auf die Datenstruktur des Octree stützten. Wichtigster Vertreter einer Octree-basierten Punktwolkenverarbeitung auf der Central Processing Unit (CPU) ist die OctoMap Bibliothek [104], die jedoch nur zweitrangig für eine Kollisionsdetektion nutzbar ist [103, 95].

Andere aktuelle Arbeiten im Bereich der Robotik, die ebenfalls das Ziel eines reaktiven Verhaltens auf Basis von Sensordaten verfolgen, sind optimierende Ansätze. Hierbei maximieren Flacco et al. in [80] oder Kaldestad et al. in [109] die Distanzen dynamischen Hindernismessungen aus Tiefensensoren (siehe Abb. 2.1).

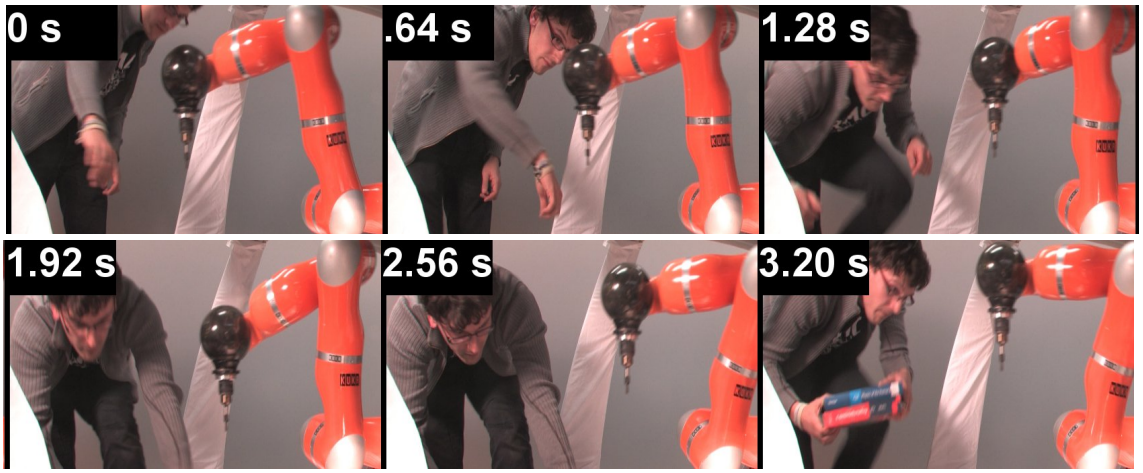


Abb. 2.1.: Reaktives Verhalten zur Vermeidung von Kollisionen: Der Roboter weicht dem Menschen aus. Bild aus [109].

Andererseits gibt es Ansätze, die direkt vergleichbar mit der vorliegenden Arbeit sind und auf einer schnellen Kollisionsdetektion basieren. So beschreibt Gibson in [87] eine Voxel-Kollisionsdetektion, die pro Voxelkarte zwei Datenstrukturen nutzt: Zum einen ein Feld aus Zeigern, die entweder Null sind (freier Raum), oder auf einen Voxel in einem weiteren Feld aus Voxel-Nutzdaten zeigen. Somit müssen bei einer Kollisionsprüfung lediglich Prüfungen auf Null durchgeführt werden, während Voxel dennoch beliebig komplizierte Nutzdaten speichern können. Diese Implementierung ist auf der GPU nicht zielführend, da der Nutzdatenspeicher dynamisch verwaltet werden muss.

Grundlegende Übereinstimmungen bestehen auch mit den Arbeiten von He und Kaufmann [98], die ebenfalls volumetrische Repräsentationen von Umwelt und Roboter in

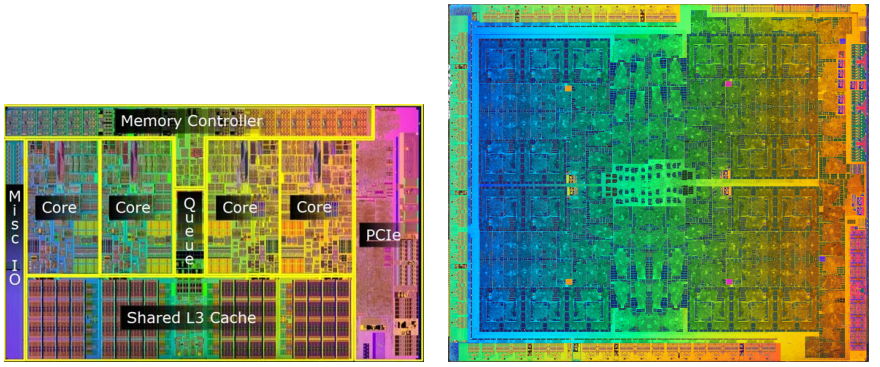
Form von Kugelbäumen nutzen, um darüber Distanzfelder zu erzeugen und hierarchische Kollisionstests durchführen. Von besonderem Interesse sind weiterhin Arbeiten, die GPUs zur parallelen Verarbeitung von Sensordaten nutzen, da diese bei größeren Eingabedaten seriellen Ansätzen weit überlegen sind. Bedkowski et al. haben dies in [47] für typische Aufgaben wie Normalen- oder Histogrammberechnung bereits gezeigt. Abgesehen davon existiert umfangreiche theoretische und praktische Arbeit im Bereich heterogener Parallelverarbeitung [58], die bisher hauptsächlich in den Gebieten der Computergrafik und des Machine Learnings eingesetzt wird, jedoch selten in der Robotik verwendet wird. Hier schlägt diese Arbeit eine Brücke und verbindet typische Vorgehensweisen der robotischen Pfadplanung mit moderner Parallelisierung.

Nur sehr wenige der vorgestellten Verfahren befinden sich im kommerziellen Einsatz. Verfügbare Lösungen sind die interaktive Anzeige von Daten bildgebender Verfahren in der Medizin, das Rendern fotorealistischer Szenarien in der Computergrafik, die Montageplanung in CAD Programmen oder Computerspiele. Bei allen handelt es sich jedoch um rein virtuelle Anwendungen. Nach aktuellem Kenntnisstand des Autors existieren keine industriellen Anwendungen, bei denen Voxel- oder GPU-Techniken mit live 3D-Daten aus Sensoren verbunden werden. Begründet wird dies meist mit dem Fehlen von sicherheitszertifizierten 3D-Kameras, auch wenn diese bereits seit langem auf der Agenda vieler Anbieter von Sicherheitstechnik stehen. Roboterhersteller hingegen planen bereits, ihre Steuerungen mit GPUs auszustatten, wobei diese eher zur Ausführung und zum Training neuronaler Netze dienen sollen.

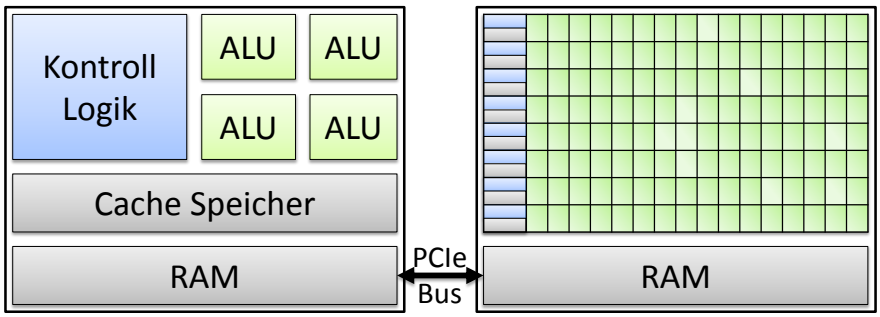
Neben diesem groben Überblick über den Stand der Technik, der sich weitgehend in die Taxonomie aus Abb. 2.2 gliedern lässt, gehen auch die einzelnen Kapitel jeweils detaillierter auf verwandte Arbeiten ein.

Abb. 2.2.: Taxonomie zur Einordnung der kollisionsfreien Bahnplanung in verwandte Themengebiete.

3. Heterogene Parallelverarbeitung



Dye



Schema

Bezeichnung	**CPU** **Intel Core i7 Broadwell**	**GPU** **Nvidia GP104 Pascal**
Rechenkerne	4	2560
Max. Threads	8	2048
Rechenleistung	$\sim$ 220 GFlops	SP: 8,873, DP: 277 GFlops
Takt	2,9 GHz	1,607 GHz
L2 Chache	4×256 kb	2048 kb
Transistoren	$\sim$ 1,8 Mrd. (ohne GPU)	7,2 Mrd.
Die Größe	$\sim 123\,mm^2$ (ohne GPU)	$314\,mm^2$
Strukturgröße	14 nm	16 nm
Abwärme	47 Watt	150 Watt

Tab. 3.1.: Vergleich einiger technischer Daten einer aktuellen CPU und GPU[1].

Nachdem über Jahrzehnte die Berechnungsleistung von Prozessoren durch höhere Takt-

[1]Bildquellen: `http://hothardware.com` und `https://www.flickr.com/photos/130561288@N04/29111683364/`,
Technische Daten: `https://www.microway.com/`, `https://en.wikichip.org/`, `https://de.wikipedia.org/wiki/Nvidia-Geforce-10-Serie`

frequenzen, Befehlssatzerweiterungen, optimierte Caching-Taktiken und erweiterte Sprungvorhersagen verbessert wurde, war 2005 eine weitere Steigerung ökonomisch nicht mehr vertretbar. Ab diesem Moment hielten Mehrkern-Prozessoren Einzug in Workstations und PCs, um dem Abwärmeproblem bei gleichzeitiger Durchsatzsteigerung zu begegnen. Somit war Parallelverarbeitung nicht mehr nur im High-Performance Computing (HPC)-Bereich anzutreffen, sondern auch in Einzelplatzcomputern.

Bereits seit einigen Jahren zuvor wuchs die Leistungsfähigkeit und der Befehlssatz von Grafikprozessoren, so dass deren Shader-Einheiten immer mehr Funktionen der CPU übernehmen konnten. Ab ca. 2001 wurden GPUs dann erstmals auch für nicht-grafische Aufgaben genutzt, die stark parallelisierbar ablaufen können (z.B. Matrix-Multiplikationen). Um den Aufwand der dafür benötigten Shader-Programmierung in OpenGL zu reduzieren, entwickelte Nvidia sein proprietäres CUDA Framework[2], dessen erste Version im Juni 2007 freigegeben wurde. Im Dezember 2008 folgte dann das von einem Konsortium spezifizierte, offene und freie OpenCL[3], welches auch auf nicht-Nvidia Grafikkarten ausführbar ist.

Seit diesem Zeitpunkt steigt die Relevanz der *heterogenen Datenverarbeitung* kontinuierlich, da sie die Vorteile von CPUs und GPUs kombiniert, die in Tab. 3.1 in Zahlen gefasst sind. Sehr unterschiedliche Anwendungsbereiche (Finanzwesen, Wetter-Vorhersage [145], geologische Simulationen [34], usw.) profitieren von dieser Entwicklung. Einer breiten Öffentlichkeit ist GPU-fokussierte Datenverarbeitung spätestens seit dem Einsatz neuronaler Netze bekannt, welche medienwirksam durch GO-Spiele [188] oder Google's Deep-Dream [140] vermarktet werden.

Auch die vorliegende Arbeit stützt sich auf die Parallelverarbeitung als eine ihrer Kerntechnologie. Dieses Kapitel stellt daher Prinzipien aktueller Hard- und Software vor und führt grundlegende Begriffe ein, bevor dann Paradigmen der Programmierung von GPUs vorgestellt werden, die auch in GPU-Voxels verfolgt wurden.

3.1. Grundlagen

Parallelisierung geschieht auf zwei Ebenen: Ein Problem muss einerseits so auf Algorithmen abgebildet werden, dass seine Teilprobleme gleichzeitig bearbeitet und später zu einer konsistenten Lösung zusammengeführt werden können. Andererseits muss Hardware zur Verfügung stehen, die die Teilaufgaben nicht nur parallel abarbeiten kann, sondern auch Funktionen zur Kommunikation und Synchronisation der einzelnen Recheneinheiten zur Verfügung stellt, um die Arbeit zu koordinieren. Zur Bewertung unterschiedlicher Hard- und Software-Kombinationen, die im Folgenden betrachtet werden, dienen folgende Metriken:

3.1.1. Flynn's Taxonomie

Die Informatik teilt die vielfältigen Hardwarearchitekturen zur Datenverarbeitung in Flynn's Taxonomie [81] ein:

[2] CUDA: `https://developer.nvidia.com/about-cuda`
[3] OpenCL: `https://www.khronos.org/opencl/`

Definition 6. Die Leistung einer Berechnungsarchitektur bestimmt sich über drei Kriterien: **Bandbreite** (Datenmenge pro Zeit), **Durchsatz** (Berechnungen pro Zeit) und **Latenz** (Zeitdauer vom Start bis zum Ende einer Berechnung). Die Relevanz der einzelnen Kriterien unterscheidet sich je nach Problemstellung.

Single Instruction, Single Data stream (SISD) Dies beschreibt die klassische Arbeitsweise von Computern mit einem Einkern-Prozessor, die nach der *Von-Neumann-* oder der *Harvard-Architektur* aufgebaut sind. Um Daten aus einem Vektor der Länge n zu verarbeiten, muss hierbei sequenziell n mal auf den Speicher zugegriffen und n Instruktionen müssen geladen werden, um n Berechnungsschritte auszuführen.

Single Instruction, Multiple Data streams (SIMD) Computer mit Vektorprozessoren sind in der Lage, dieselbe Instruktion gleichzeitig auf m Daten anzuwenden. Stellt ein Speicherzugriff m Daten zur Verfügung, können diese also in einem einzelnen Berechnungsschritt abgearbeitet werden. Moderne Desktop-Prozessoren enthalten neben einer SISD Berechnungseinheit oft weitere SIMD Einheiten um spezielle Operationen datenparallel und somit effizienter abzuarbeiten.

Multiple Instruction, Single Data stream (MISD) Hierbei werden unterschiedliche Operationen innerhalb eines Berechnungsschritts auf ein Datum angewendet. In der Praxis ist dies, außer bei redundant ausgelegten Pipeline-Computern, selten anzutreffen.

Multiple Instruction, Multiple Data streams (MIMD) Es können gleichzeitig unterschiedliche Operationen auf mehrere Datenströme angewendet werden. Dies wird meist durch die Kopplung mehrerer Prozessoren oder Prozessorkerne erreicht. Moderne Desktop-Prozessoren fallen in diese Kategorie, da sie simultan mit mehreren Threads rechnen können, während sich die Threads über einen geteilten Speicher synchronisieren und Informationen austauschen.

Die Parallelisierung bei der MIMD Ausführung kann weiter unterschieden werden:

Single Program, Multiple Data (SPMD) Um große Datenmengen parallel mit demselben Algorithmus zu verarbeiten, kann dieser mehrfach auf unterschiedliche Teilmengen der Daten angewendet werden. Derselbe Algorithmus läuft dabei (im Gegensatz zu SIMD) auf mehreren autonomen Berechnungseinheiten. Diese Art der Parallelisierung ist bei der Nutzung von Mehrkernprozessoren sehr stark verbreitet und stellt bspw. die Grundlage von *Open Multi-Processing Library (OpenMP)* [4] dar.

Multiple Program, Multiple Data (MPMD) Hier ist die Kopplung der Recheneinheiten noch losgelöster, da unterschiedliche Algorithmen auf unterschiedliche Daten angewendet werden. Dies ist der Fall, wenn bei verteiltem Rechnen eine Management-Entität die eigentlichen Berechnungen koordiniert an weitere Prozessoren auslagert und ihre Ergebnisse sammelt.

[4] OpenMP ist ein Application Programming Interface (API) für die einfache parallele Programmierung mittels geteiltem Speicher in C / C++ und Fortran: `http://www.openmp.org/`

In Flynn's Taxonomie wären GPUs zwischen SIMD und MIMD einzuordnen, da die Shader-Recheneinheiten zwar zunächst alle dieselben Instruktionen laden (Kernel-Code) und diese nach dem SPDM-Prinzip auf unterschiedliche Daten anwenden. Jedoch können Threads dann zur Laufzeit individuelle Entscheidungen in den Programmverzweigungen treffen. Nvidia ordnet daher seine Hardwarearchitektur nicht direkt in dieses Schema ein, sondern prägt den Begriff SIMT [154], der wie folgt definiert ist:

Definition 7. Die Hardwarearchitektur einer GPU arbeitet nach dem **Single Instruction, Multiple Threads (SIMT)** Prinzip. Der Name verdeutlicht, dass einem Programmierer trotz paralleler Logik sämtliche Optionen der Programmverzweigung zur Verfügung stehen (im Gegensatz zu SIMD). Allerdings geschieht im Gegensatz zu MIMD eine pro Thread individuell ablaufende Programmausführung auf Kosten der Laufzeit (vgl. Abb. 3.5).

3.1.2. Parallelisierung auf Aufgaben- und Datenbasis

Einer der Gründe für die Vielzahl an Klassen in Flynn's Taxonomie ist der Unterschied in den Anforderungen, die verschiedene Problemklassen an die Hardware stellen. Hier kann zwischen *Aufgaben-* und *Datenparallelisierung* unterschieden werden. Im einen Fall sind viele unabhängige Aufgaben auszuführen, die somit parallel auf verschiedenen Recheneinheiten ablaufen können. Da dabei jede Einheit sequentiell arbeitet, bieten sich MPMD Architekturen an. Im anderen Fall sind nur wenige Aufgaben auf einer großen Menge an Daten anzuwenden. Die Parallelisierung erstreckt sich somit über die Daten und eine SIMD Architektur ist von Vorteil.

Feststellung 1. GP-GPUs eignen sich vorrangig für datenparallele Aufgaben. Ihre Berechnungseinheiten sind im Vergleich zu CPUs wesentlich leichtgewichtiger aufgebaut, und somit nicht auf die Optimierung von komplexen Kontrollflüssen (keine Sprungvorhersage) ausgelegt, sondern auf eine hohe Bandbreite bei einfacher Kontrolllogik. CPUs hingegen unterstützen einen komplexen, unvorhersehbaren aber sequentiellen Kontrollfluss. Sie weisen kürzere Latenzen und höheren Durchsatz auf.

Die Aufteilung der Daten auf parallel laufende Prozesse muss zum Speicherinterface der Hardware passen und ist somit entscheidend für die Bandbreite eines Algorithmus. Unterschiedliche Muster und Granularitäten der Partitionierung sind in Abb. 3.1 dargestellt. Zu sehen sind *zyklische Muster* (jeder Prozess bearbeitet mehrere, weit verteilte Daten) und *blockweise Muster* (jeder Prozess bearbeitet einen zusammenhängenden Bereich im Speicher). Weist eine Problemstellung hohe Datenlokalität auf, ist es von Vorteil, diese Daten kompakt im Speicher abzulegen, um ein gutes Caching Verhalten zu erreichen. Diese Problematik wird in Unterabschnitt 3.2.3 detailliert beschrieben.

3.1.3. Programmsynchronisation: Daten- und Ressourcenabhängigkeit

Losgelöst von der verwendeten Hardware ist die Parallelisierbarkeit eines Problems zunächst durch seine interne *Datenabhängigkeit* bestimmt. So gibt es in jedem Programm Berechnungen, die von den Ergebnissen vorheriger Schritte abhängig sind und daher nicht

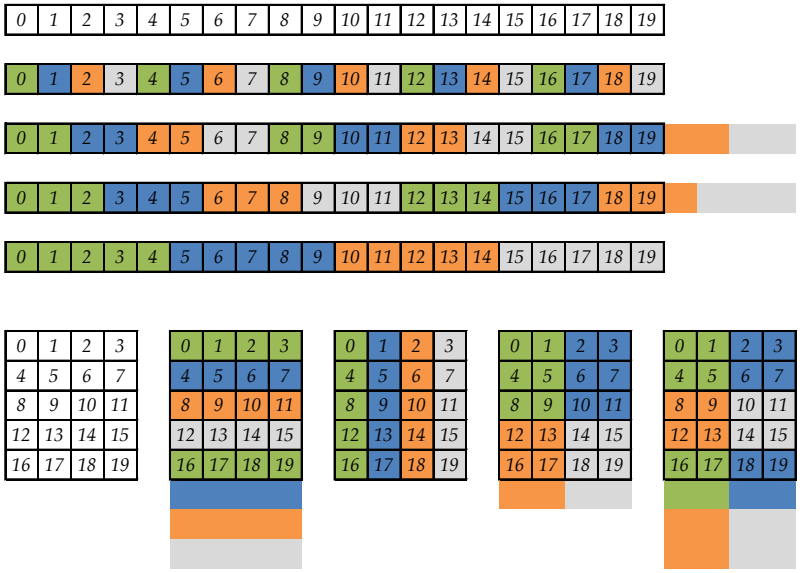


Abb. 3.1.: Unterschiedliche Möglichkeiten, ein Datenfeld aus 20 Einträgen mit vier Prozessen (farblich gekennzeichnet) zu bearbeiten. Da auf einer GPU die Threads eines Blocks gleich lang laufen, entsteht bei einigen Aufteilungen ein Leerlauf einzelner Threads.

parallel zu diesen ausführbar sind (siehe Abb. 3.2). Auch kleinste Operationen können eine Abhängigkeit bedeuten: Sollen beispielsweise mehrere Prozesse einen gemeinsamen Ergebniszähler inkrementieren, kann dies nicht gleichzeitig geschehen. Solche Punkte in der Berechnungslogik zu identifizieren ist eine Voraussetzung für die Parallelisierung der unabhängigen Programmteile.

Ähnlich verhält es sich mit Ressourcenabhängigkeiten zwischen mehreren Prozessen. Müssen diese mit einer exklusiven Ressource arbeiten, beispielsweise dem Speicherbus, muss der Zugriff serialisiert geschehen. Auch hier liegt es am Programmierer, das Zugriffsmuster möglichst so zu gestalten, dass Wartezeiten minimiert werden.

In beiden Fällen ist eine Synchronisierung unter den Prozessen unabdingbar, um kritische Wettläufe zu unterbinden und eine deterministische Ausführung zu gewährleisten. Ihr konkreter Aufwand hängt von der Hardwarearchitektur ab, da die Synchronisation eine bidirektionale Kommunikation beschreibt: Ein Multiprozessor mit gemeinsam genutztem Speicher bildet die Kommunikation sehr effizient darauf ab, während sich ein verteiltes System über eine externe Verbindung austauschen muss. In beiden Fällen stellt die Synchronisation jedoch eine, bezüglich der Ausführungszeit, teure Operation dar, die nur bei Notwendigkeit eingesetzt werden sollte.

3.1.4. Multithreading

Um die Wartezeiten von Threads, die aufgrund von Daten- oder Ressourcenabhängigkeiten blockiert sind, effizient zu nutzen, greifen CPUs und GPUs auf Multithreading zurück. Dabei werden mehrere Prozesse abwechselnd auf der Hardware ausgeführt und

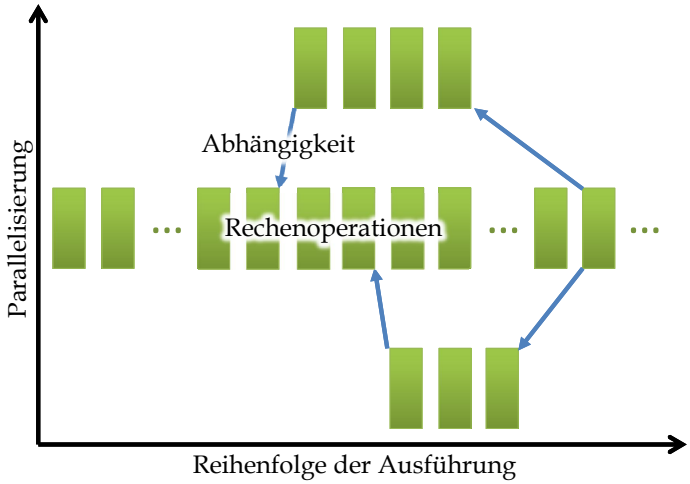


Abb. 3.2.: Parallelisierung und Datenabhängigkeit. Blaue Pfeile symbolisieren Datenabhängigkeiten einzelner Programmteile.

Feststellung 2. Das Verhältnis aus unabhängigen und abhängigen Programmanteilen entscheidet maßgeblich, ob und wieweit sich die Aufteilung eines Problems auf mehrere parallele Prozesse rentiert. Das Amdahlsche Gesetz [36] beschränkt dabei die maximal erreichbare Beschleunigung durch parallele Ausführung.

so eine weitere virtuelle Parallelisierungsebene oberhalb einer echt-parallelen Ausführung geschaffen.

CPU-Threads stellen dabei komplexe Einheiten dar, deren Wechsel aufwendig ist und das Betriebssystem hochgradig involviert. Der Aufwand zur Konsistenz-Sicherung aller Cache-Hierarchien erfordert dabei einen entsprechend langwierigen Prozess. Im Gegensatz dazu sind GPU-Threads sehr leichtgewichtig. Durch die schlanken Cache-Hierarchien auf der GPU ist ein Wechsel sehr schnell und einfach möglich. Daher unterscheiden sich die Strategien beider Architekturen stark: Während eine CPU auf die Verwaltung von nur wenigen Threads (2-16) per Hardware ausgelegt ist, kann eine GPU pro Multiprozessor über 1000 Threads vorhalten und bei Bedarf schnell zur Ausführung bringen.

Feststellung 3. Um den Speicherbus optimal auszulasten, sollte die **Auslastung (Occupancy)** einer GPU mit wartenden Threads wesentlich höher ausfallen, als bei einer CPU. Durch schnelle Threadwechsel können somit Wartezeiten aufgrund von Ressourcenkonflikten effizient überdeckt werden. Faktoren, die Zahl der Threads limitieren, sind unter anderem die begrenzte Anzahl an Registern und die Größe des geteilten Speichers.

3.1.5. Zusammenfassung

Bereits mit einem einzigen regulären Computer haben Entwickler heute die Möglichkeit, die Vorteile zweier grundlegend verschiedener Hardwarearchitekturen komfortabel zu

kombinieren: Aktuelle CPUs können vier bis acht komplexe Programme parallel abarbeiten, wobei diese untereinander über Interprozesskommunikation effizient synchronisierbar sind und auf umfangreiche Speicherhierarchieen zugreifen können. Zusätzlich stehen für die datenparallele Verarbeitung GPUs bereit, die mit ihrer hohen Zahl an Verarbeitungseinheiten einfache Algorithmen massiv parallel ausführen können. Da CPUs und GPUs über einen breiten PCI Bus Daten austauschen können, lassen sich die Aufgaben je nach ihren Anforderungsprofilen auf beide Architekturen verteilen.

Feststellung 4. Für das Design von effizienten Algorithmen ist eine genaue Kenntnis über das Verarbeitungsprinzip der Zielhardware essentiell. Da sich GPU und CPU in ihrer Art der Parallelisierung grundlegend unterscheiden, können CPU Algorithmen, die auf SPMD Verarbeitung optimiert sind – abhängig von ihrem Datenmodell – gar nicht, oder nur mit großem Aufwand, zu GPU geeigneten SIMT Algorithmen portiert werden.

Auch GPU-Voxels nutzt heterogene Datenverarbeitung, indem oftmals die eigentlichen Berechnungen auf der GPU stattfinden, die Ergebnisse jedoch erst auf der CPU zusammengeführt und final ausgewertet werden. Für die Programmierung solcher kombinierter Anwendungen wird in dieser Arbeit das CUDA Framework genutzt. Dieses wird im Folgenden in Bezug auf weitere GPU-spezifische Kriterien, die sich auf die Effizienz der Parallelisierung auswirken, vorgestellt.

3.2. CUDA Praxis

Compute Unified Device Architecture (CUDA) stellt eine Reihe von Werkzeugen dar, die es erlauben, Algorithmen für ein heterogenes System aus einer CPU und einer oder mehreren Nvidia-GP-GPUs in unterschiedlichen Programmiersprachen zu erstellen und auszuführen. Dabei ist es dem Programmierer überlassen, welche Teile er für die CPU und welche er für die GPU implementiert. Die theoretisch erreichbaren Datenraten aus Abb. 3.4 und Berechnungsdurchsätze aus Abb. 3.3 lassen vermuten, dass GPUs in datenintensiven Anwendungen heutige CPUs um ein Vielfaches überbieten. Auch wenn Lee et al. zeigen, dass vielfach reklamierte Leistungssteigerungen mehrerer Größenordnungen nicht realistisch sind, so können dennoch Beschleunigungen bis zu einem Faktor von sieben erreicht werden [133]. Im Folgenden Kapitel sollen nun die Besonderheiten beschrieben werden, die beim Design von Algorithmen zu beachten sind, um GPUs mit ihrer Vielzahl von Rechenkernen (aktuell bis zu 2688) optimal auszulasten, um so diese Beschleunigungen zu erreichen.

Die wichtigsten Bestandteile von CUDA sind die API, die Laufzeitumgebung und der *nvcc* Compiler, die in ihrem Zusammenspiel von hardwarespezifischen Eigenschaften der GPUs abstrahieren und die Erzeugung von portablen Programmen ermöglichen. Dafür generiert der Compiler einen Meta-Code (PTX), der erst durch den Treiber in Binär-Code umgewandelt und dann ausgeführt wird. Ergänzt werden diese Kernkomponenten durch anwendungsspezifische Bibliotheken, zum Teil von Drittanbietern (bspw. Thrust[5],

[5]Thrust: `https://thrust.github.io/`

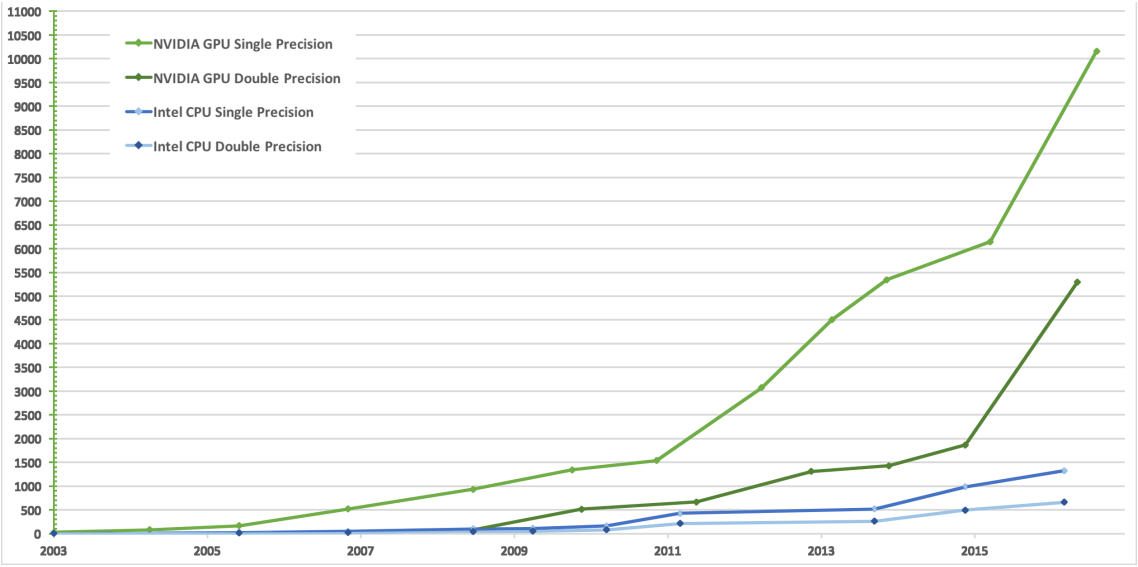


Abb. 3.3.: Vergleich des theoretisch möglichen Durchsatzes [GFLOP/s] von Intel CPUs und Nvidia GPUs über die Jahre 2003 bis 2016. Entnommen aus [154].

ArrayFire[6], CUB[7], NPP[8]), die gängige Problemstellungen komfortabel abdecken.

Die Verfügbarkeit dieser Bibliotheken, die optimale Unterstützung der Hardware und die ausgereiften Optimierungswerkzeuge führten für diese Arbeit zur Entscheidung, das proprietäre CUDA dem offenen OpenCL vorzuziehen.

3.2.1. CUDA-Kernel

Da eine GP-GPU eine nicht autonom lauffähige Komponente in einem Computersystem darstellt, spricht man von einem *Device*, das in einem *Host*-System läuft. Im weiteren Verlauf werden die Begriffe *Device* und GP-GPU äquivalent verwendet, genauso wie *Host* und CPU. Um Code auf der GP-GPU ausführen zu können, muss dieser bei CUDA in Form eines *Kernels* vorliegen, der vom Host Code aufgerufen wird. Wie im Beispielcode in Algorithmus 1 zu sehen, ist die Syntax eines Kernels identisch zu einer `void` C-Funktion, die um das Schlüsselwort `__global__` bzw. `__device__` erweitert wurde. Die übergebenen Funktionsparameter werden beim Methodenaufruf auf die Grafikkarte kopiert und stehen jedem Thread zur Verfügung. Beim Aufruf muss über gesonderte Parameter in ⋘ ... ⋙-Schreibweise die Anzahl der Ausführungsblöcke und der Threads pro Block angegeben werden, um durch sie den Parallelisierungsgrad zu bestimmen. Hierzu mehr im nächsten Abschnitt. Zudem benötigt ein Kernel Zeiger auf seine Nutz- und Ausgabe-Datenstrukturen, um mit diesen arbeiten zu können. Über die Block- und Thread-IDs eines jeden Threads können einzelne Kernel-Instanzen unterschiedliche Datenpartitionen bearbeiten (vgl. Abb. 3.1).

[6] ArrayFire: `https://github.com/arrayfire/arrayfire`
[7] CUB: `http://nvlabs.github.io/cub/`
[8] NPP: `https://developer.nvidia.com/npp`

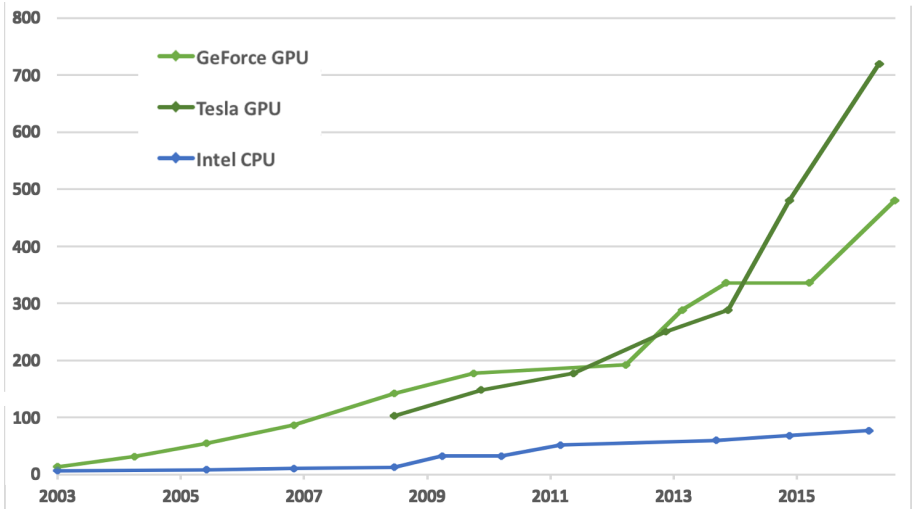
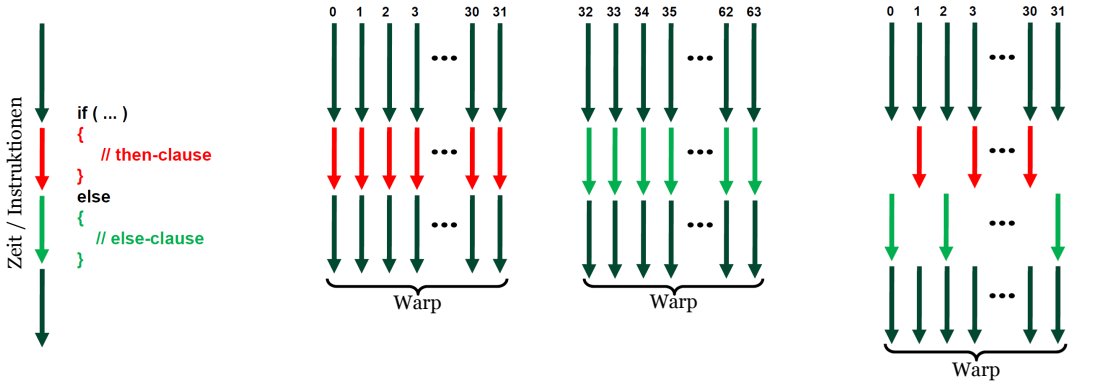


Abb. 3.4.: Vergleich der theoretisch möglichen Speicherbandbreite [GB/s] von Intel CPUs und Nvidia GPUs über die Jahre 2003 bis 2016. Entnommen aus [154].



(a) Thread Code (b) Gleicher Instruktionsfluss pro Warp (c) Thread Divergenz

Abb. 3.5.: Erhöhte Laufzeit durch Divergenz der Threads eines Warps

Bedingt durch das oben beschriebene SIMT-Prinzip der Hardwarearchitektur führen alle (innerhalb eines Warps) parallel ablaufenden Instanzen eines Kernels (Threads) ihre Befehle synchron aus. Divergenzen im Programmfluss (siehe Abb. 3.5), wie zum Beispiel Threadspezifisch ausgewertete `if-else`-Verzweigungen, oder verschieden lange `for`-Schleifen, führen zu einer Serialisierung der Codeabschnitte und zu empfindlichen Performance-Einbrüchen, da Wartezeiten in den Threads entstehen, die für sie irrelevante Codeabschnitte nicht ausführen. Verzweigungen innerhalb des Kernel-Codes sollten also möglichst vermieden werden, bzw. wie durch Han und Abdelrahman in [94] beschrieben, durch *iteration delaying* oder *branch distribution* geschickt angeordnet werden.

Algorithmus 1 Beispielkernel mit Grid-Stride-Loops und aufrufender Host-Code.

```
1  __global__
2  void VecAddition(int n, float* A, float* B, res* C) {
3     for (int i = blockIdx.x * blockDim.x + threadIdx.x;
4        i < n;
5        i += blockDim.x * gridDim.x)
6     {
7        C[i] = A[i] + B[i];
8     }
9  }

11 int main() {
12    ...
13    // Kernelaufruf mit 4 * 512 Threads und 4046 Eingabe-Elementen
14    VecAddition<<<4, 512>>>(4046, A, B, C);
15    ...
16 }
```

3.2.2. Grids, Blöcke, Warps und Threads in CUDA

Die Ausführung eines Kernels erfolgt durch eine Vielzahl an *Threads*, die gemeinsam durch den Kernel-Aufruf auf dem Device gestartet werden. Eine GP-GPU besteht aus mehreren Streaming-Multiprozessoren, die jeweils eine Vielzahl von *Warps* zu je 32 Threads simultan mit einem geteilten Steuerwerk bearbeiten können. Da im Normalfall deutlich mehr als 32 Threads zu starten sind, lassen diese sich zur Arbeitsaufteilung und zu Schedulingzwecken in *Blöcken* arrangieren. Die Blöcke eines Kernels bilden wiederum ein *Grid*. Diese in Abb. 3.6 gezeigte Hierarchie bestimmt auch die Synchronisation und Kommunikation zwischen den Threads, da nicht alle Speicherebenen einer GPU gleichermaßen für alle Threads sichtbar sind (siehe Tab. 3.2 und Abb. 3.8). Jeder Block muss somit eine unabhängige Einheit darstellen, die für ihre Ausführung keinen Zugriff auf andere Blöcke benötigt. Dies erlaubt die Skalierung der Parallelität in CUDA: Je mehr Multiprozessoren zur Verfügung stehen, desto mehr Blöcke laufen gleichzeitig. Während die Threads eines Blocks immer auf demselben GPU-Prozessorkern ausgeführt werden, können sich die Blöcke eines Grids auch über mehrere Prozessoren verteilen oder sequentiell zur Ausführung gebracht werden. Je nach Anwendungsfall lassen sich sowohl Threads als auch Blöcke ein-, zwei-, oder dreidimensional gestalten. So sind aktuell über 30 000 Threads auf einer GP-GPU verwaltbar, wobei jeder Thread seine Block- und Thread-ID kennt. Das effizienteste Verhältnis zwischen Threads-pro-Block und Blöcken-pro-Grid ist von zahlreichen Parametern abhängig und schwer analytisch zu bestimmen. Daher wird die Auslastung einer GPU meist empirisch mittels konkreter Benchmarks optimiert, wie in der Evaluation in Abschnitt 8.1 beschrieben ist.

Weiterhin muss beim Design eines Kernels auf die im Folgenden beschriebene CUDA Speicherhierarchie geachtet werden, um effiziente Inter-Thread und Inter-Block Kommunikation bzw. Synchronisation zu bewerkstelligen.

3.2.3. Speicherarchitektur

Die Speicherarchitektur einer CUDA Grafikkarte, die in Abb. 3.8 gezeigt ist, weist, wie auch das Host-System, eine Hierarchie auf, verfügt aber lediglich über sehr einfache Ca-

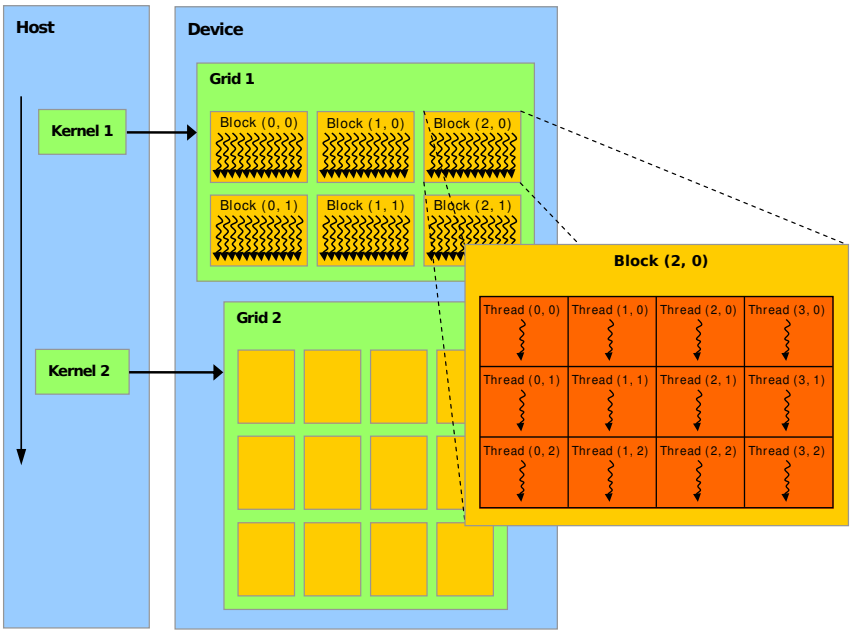


Abb. 3.6.: CUDA Kernels werden durch eine Menge von Threads ausgeführt, die in (mehrdimensionalen) Blöcken organisiert sind. Blöcke wiederum bilden (mehrdimensionalen) Grids. Adaptiert nach [154].

ching-Mechanismen. Einzelne Threads arbeiten auf eigenen Registern, die in der Hardware sehr schnell umschaltbar sind, wenn ein Threadwechsel durchgeführt wird. Weiterhin verfügt jeder Thread über *lokalen Speicher* im RAM, der als Heap oder Register-Auslagerung genutzt wird. Die Threads eines Blocks teilen sich einen gemeinsamen Speicherbereich (*Shared*), über den sie Daten austauschen können.

Shared Memory und Datenzugriffsmuster

Jeder Kern eines GPU-Multiprozessors verfügt neben Caches für Texturen und konstanten Speicher über einen Speicherbereich (siehe Abb. 3.9), der von allen Threads eines Blocks gelesen und geschrieben werden kann. Neben der Inter-Thread-Kommunikation kann dieser auch als selbstverwalteter L1 Cache dienen, um Daten aus dem GPU-RAM performant zu puffern. Dieser geteilte Speicher ist ein bis zu 48 kByte großer allozierter Bereich des L1 Caches, der jedoch zwischen allen Blöcke aufgeteilt werden muss. Daher ist bei jedem Kernelaufruf die benötigte Größe über einen Kernel-Parameter zu spezifizieren. Da sich der Speicher auf 32 Speicherbänke verteilt, müssen sich alle 32 Threads eines Warps (nicht eines Blocks) 32 Zugriffswege teilen, die jeweils 1 Wort breit sind. Ein Wort entspricht 4 Bytes und fasst somit beispielsweise ein `float` oder einen `int32`. Die zuständige Speicherbank für das Lesen eines Bytes B lässt sich über $(Addr._B \mod 4) \mod 32$ bestimmen. Greifen die Threads des Warps auf Speicherbereiche in unterschiedlichen Bänken zu (egal, in welcher Permutation), geschieht dies voll parallel. Wenn mehrere Threads dieselbe Adresse lesen, erzeugt dies einen Broad- oder Multicast und schränkt die Parallelisierung somit nicht ein. Benötigen jedoch mehrere Threads Speicheradressen aus derselben Bank, können diese nur sequentiell abgerufen werden. In diesem Fall be-

steht ein *Bank-Konflikt*, der die Leistung unter Umständen einschränkt. Beispiele dazu finden sich in Abb. 3.7. Auch wenn die durch Bank-Konflikte verursachten Latenzen in vielen Fällen durch ein Scheduling anderer Warps abgefangen werden, sollten Konflikte durch eine passend gewählte Datenanordnung, notfalls durch zusätzliche Padding-Bytes, vermieden werden.

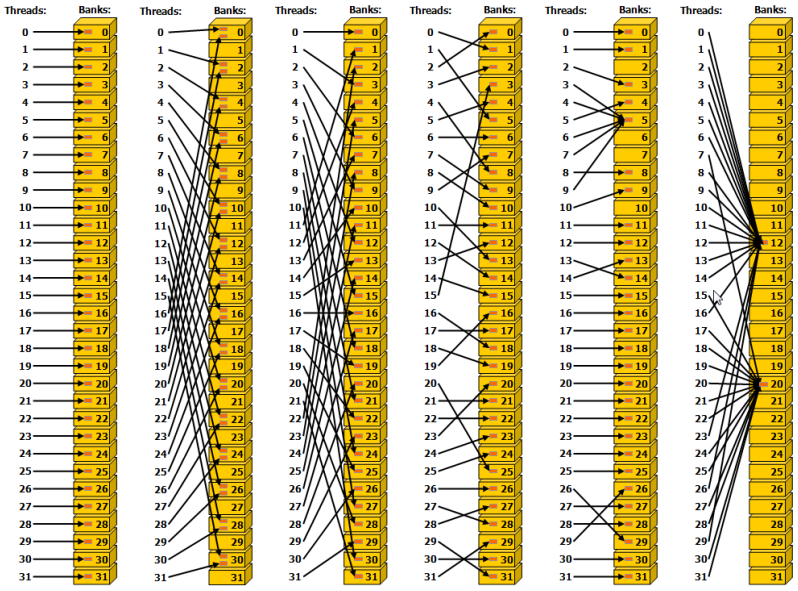


Abb. 3.7.: Zugriffsmuster auf Speicherbänke des geteilten Speichers mit 4 Byte Wörtern. Vlnr: a) Linear mit Schrittgröße 1 (keine Konflikte). b) Linear mit Schrittgröße 2 (zweifacher Bank-Konflikt). c) Linear mit Schrittgröße 3 (keine Konflikte). d) Randomisiert ohne Bank-Konflikte (keine Konflikte). e) Randomisiert mit teilweisem Broadcast (keine Konflikte). Broadcast von zwei Elementen (keine Konflikte). Entnommen aus [154]).

Datenzugriffsmuster auf globalen Speicher

Feststellung 5. Im Vergleich zur Speicherverwaltung des Host-Systems ist eine GPU um Faktor 30-40 mal langsamer (`malloc()` vs. `cudaMalloc()`). Dieser von Boyer et al. in [50] beschriebene Faktor deckt sich mit selbst durchgeführten Experimenten. Daher eignen sich dynamische Datenstrukturen nur sehr bedingt für eine Portierung auf GP-GPUs.

Ein Zugriff auf den GPU RAM weist eine Latenz zwischen 200 und 400 Prozessorzyklen auf, falls die gesuchten Werte noch nicht im Cache vorliegen. Da dies im Vergleich zu einem Zugriff auf den geteilten Speicher (10 Zyklen Latenz) sehr teuer ist, sollten auch hier die Zugriffsmuster optimiert werden, um die Bandbreite eines Kernels zu maximieren. Entscheidend ist dafür eine Kenntnis über den Nvidia Speicherbus. Dieser erlaubt eine lineare Adressierung und überträgt Daten ausschließlich in 32 oder 128 Byte großen Abschnitten (ohne / mit Caching). Benötigen Threads nur einzelne Bytes aus dem Speicher, und liegen diese verteilt im RAM, so müssen dennoch für jeden Zugriff mindestens 32 Bytes transferiert werden, auch wenn ein Großteil der Daten nicht verwendet wird.

Greifen jedoch mehrere Threads eines Warps auf zusammenhängende Speicherbereiche innerhalb eines solchen Abschnittes zu, so muss dieser nur einmal übertragen werden. Ein positives Beispiel für dieses so genannte *Memory Coalescing* ist in Abb. 3.10 links zu sehen. Im zweiten, ungünstig ausgerichteten Fall rechts tritt ein Mehraufwand von 25% bzw. 100% auf (ohne / mit Caching).

In der Praxis ist ein sequentielles Zugriffsmuster am einfachsten zu erreichen, indem Threads mit fortlaufender ID auf aufeinander folgende Speicheradressen zugreifen, die möglichst eng zusammen liegen und an Wort-Adressen ausgerichtet sind. Dieses Prinzip wird in der Kernel-Programmierung durch so genannte *Grid-Stride-Loops* durchgesetzt, deren Schrittgröße der Anzahl an Threads pro Block entspricht (siehe Algorithmus 1). Dabei muss sichergestellt sein, dass bei zu breiter Parallelisierung (mehr Threads als Daten) keine ungültigen Speicherzugriffe geschehen. Werden zusammengesetzte Datenstrukturen verwendet, sind des weiteren *Structures of Arrays* vor einem *Array of Structures* zu bevorzugen, wie die Codebeispiele in Abschnitt A.4 des Anhangs verdeutlichen.

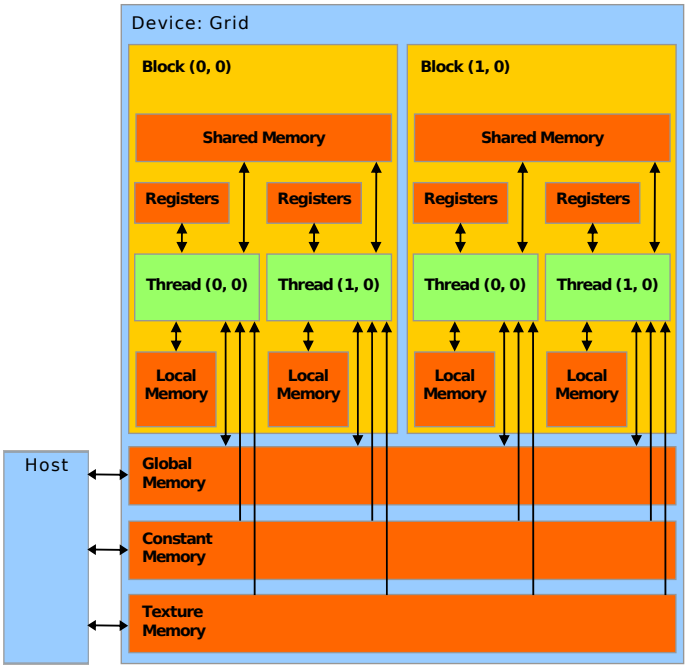


Abb. 3.8.: Blockdiagramm der CUDA-Speicherarchitektur. Entnommen aus [154].

Textur- und konstanter Speicher

Eine GPU bietet zwei weitere Zugriffswege auf ihren RAM an, die in vielen Anwendungsfällen einen höheren Datendurchsatz und niedrigere Latenzen aufweisen, als der kanonische Zugriff auf globalen Speicher. Dies ist zum einen ein *konstanter Speicher*, der von der Host-Seite aus beschrieben und von Device-Seite lediglich gelesen wird, wodurch sich Cache-Inkonsistenzen beim Schreiben zugunsten der Geschwindigkeit eliminieren lassen. Gleiches gilt für den *Textur-Speicher*, der im Allgemeinen ebenfalls in Kerneln nur lesbar gemappt wird. Er bietet weitere Zusatzfunktionen, wie z.B. hardwarebeschleunigte Interpolation und Normalisierung zwischen Einträgen. Außerdem kann

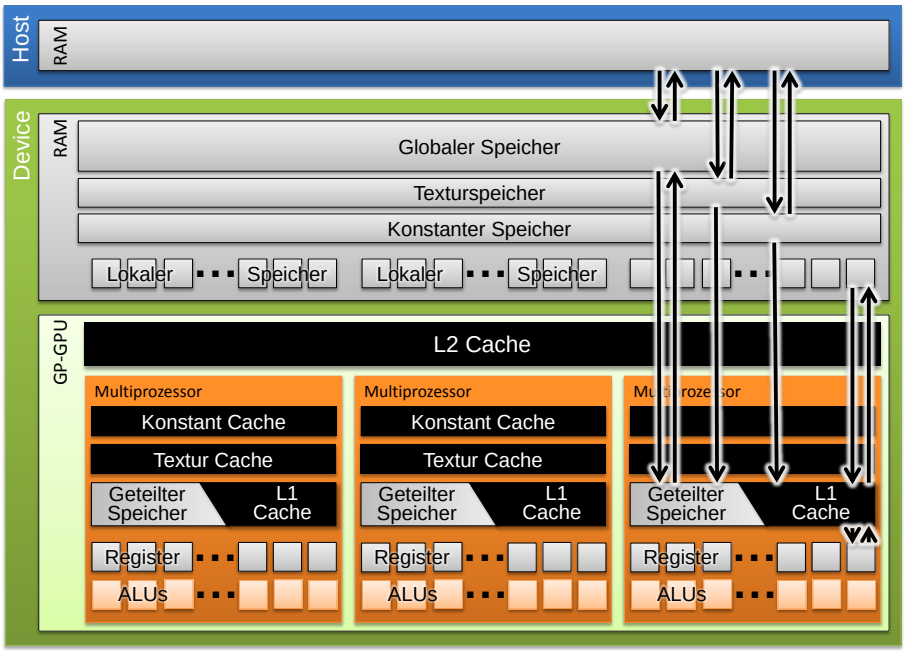


Abb. 3.9.: Physische Aufteilung des Speichers (grau) auf RAM und Prozessor-Speicher. Der Datentransfer ist durch Pfeile rechts dargestellt. Es ist ersichtlich, dass der lokale Speicher zwar Thread-lokal ist, aber im RAM liegt, während nur der geteilte Speicher direkt im GPU-Chip residiert. Caches sind schwarz dargestellt.

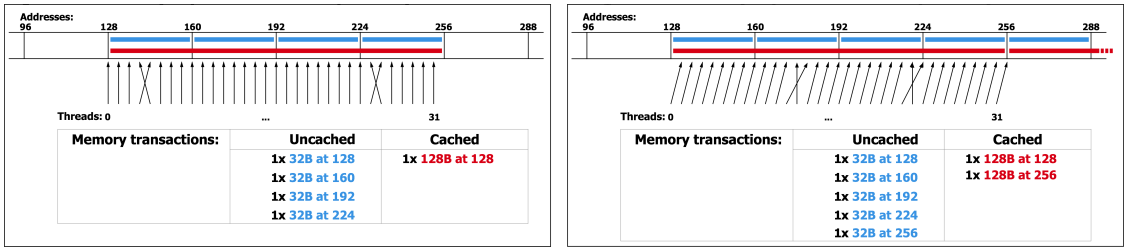
ein Zugriff auf Einträge außerhalb eines Textur-Arrays wieder in das Array umgeleitet werden. Beide Zugriffsarten verlaufen gecached, wobei bei konstantem Speicher jeder Speicherzugriff eines halben Warps eine Transaktion darstellt und somit nur effizient ist, wenn alle Threads an derselben Adresse lesen. Die Verwendung beider Speicherspezialisierungen erfordert sehr detailliertes Wissen zur Beurteilung der Zweckmäßigkeit und einen höheren Programmieraufwand, da ein Zugriff nur mittels Bindings möglich ist.

3.2.4. CUDA Intrinsics

Die CUDA-API stellt dem Programmierer viele Befehle zur Verfügung, die komplexe Funktionen hardwarenah umsetzen und dabei wesentlich effizienter ablaufen, als eine manuelle Implementierung. Dazu zählen sowohl Funktionen, die innerhalb eines Warps Ergebnisse zusammenführen (*Voting*), aber auch mathematische Funktionen (*Bitcounting*) oder Synchronisationsbarrieren für ganze Threadblöcke. Einige Funktionsbeispiele, die in GPU-Voxels konsequent eingesetzt wurden, finden sich im Anhang in Abschnitt A.2.

3.2.5. Weitere Konzepte der Parallelverarbeitung

Neben den beschriebenen, sehr hardwarenahen Optimierungstechniken, existieren auch allgemeingültigere, abstraktere Programmiermuster zur Effizienzsteigerung von parallelen Algorithmen. So sollte es immer das Ziel sein, eine Aufgabe mit geringem Aufwand in möglichst gleich große Teilaufgaben zu zerlegen, die sich in unabhängigen Paketen



(a) Ausgerichteter Speicherzugriff (sequentiell und nicht-sequentiell)

(b) Nicht ausgerichteter Speicherzugriff (sequentiell und nicht-sequentiell)

Abb. 3.10.: Beispiel für den Speicherzugriff eines Warps mit und ohne *Memory-Coalescing* und die benötigten Bus-Übertragungen. Jeder Thread greift auf ein 4 Byte-Wort zu. Grafik aus [154].

Speicher	Latenz	Zugriff	Sichtbarkeit	Lebensdauer
Global	200-400	R/W	Alle Threads + CPU	Programm
Constant	*10*	R	Alle Threads + CPU	Programm
Texture	200-400	R	Alle Threads + CPU	Programm
Local	200-400	R/W	Thread	Thread
Shared	*10*	R/W	Block	Block
Register	0	R/W	Thread	Thread

Tab. 3.2.: Übersicht der wichtigsten Eigenschaften der unterschiedlichen Speichertypen. Bei der Latenz von Constant Memory und Shared Memory ist keine zuverlässige Angabe möglich, sondern nur eine pessimistische Abschätzung. Daten aus [22].

Feststellung 6. Bedingt durch die parallele Nutzung der lokalen / globalen Speicherbusse und die beschränkten Möglichkeiten des CUDA-Compilers zur Zugriffsoptimierung, ist eine genaue Planung der Speicherzugriffsmuster durch den Programmierer bei der Implementierung eines GPU-Algorithmus von wesentlich größerer Bedeutung, als bei einer CPU-Implementierung.

parallel bearbeiten lassen. Diese Pakete formen dann CUDA-Blöcke, in welchen wiederum eine feingranulare Parallelisierung auf Thread-Ebene stattfindet. Hier ist dann aufgrund der SIMT Hardwareeigenschaften eine datenparallele Verarbeitung der Schlüssel, wofür ein genaues Verständnis der Datenabhängigkeiten vorliegen muss. Auf Blockebene stehen für eine effiziente Kommunikation und Synchronisation die genannten *Intrinsics* zur Verfügung. Abschließend sind die Ergebnisse mittels *Compaction* bzw. *Reduktion* wieder zusammenzuführen, wobei eine Abwägung zwischen Datentransfer und Berechnungsaufwand aufzustellen ist. So kann es bei hoher Komplexität zielführend sein, den letzten Schritt auf dem Host auszuführen. Konkrete Code-Beispiele finden sich in Abschnitt A.5 des Anhangs.

Kontextwechsel und CUDA Streams

Host und Device stellen zwei eigenständige Systeme mit eigener Speicherhierarchie dar, die untereinander ausschließlich über den PCIe Bus gekoppelt sind. Da dieser Bus, verglichen mit den jeweiligen Speicherhierarchien, eine vergleichsweise niedrige Datenübertragungsrate aufweist [65], empfiehlt es sich, Algorithmen zu entwerfen, die einen möglichst geringen Datenaustausch zwischen GPU und CPU erfordern. Aus demselben Grund sollte bei einem Vergleich der Berechnungszeiten unterschiedlicher GPU-Algorithmen, wie von Gregg et al. in *Where is the data?* [91] angemahnt, immer angegeben werden, ob die gemessenen Zeiten die Datenübertragung beinhalten.

Es hat sich bewährt, Verarbeitungsketten zu implementieren, deren Eingabedaten einmalig auf das Device kopiert und dort bei ihrer Verarbeitung entsprechend verändert werden. Lediglich das Endergebnis sollte zurück auf den Host kopiert werden müssen. Techniken, wie *Unified Memory* oder Thrust's Zuweisungsoperator, verbergen den nötigen Speichertransfer vor dem Programmierer und verleiten so zu einem feingranularen Kopieren von Daten, was jedoch zu Lasten der Laufzeit geht.

Aktuelle GP-GPU Generationen erlauben noch eine weitere Stufe der Parallelisierung, indem Operationen auf so genannten *CUDA-Streams* aufgeteilt werden. Unter bestimmten Umständen sind diese gleichzeitig ausführbar, zum Beispiel wenn ein Stream ausschließlich Kernel mit Berechnungen auf einem Teil des GPU-Speichers ausführt, während ein zweiter Stream Daten in oder aus einem anderen GPU-Speicherbereich von oder auf den Host kopiert. CUDA-Streams kommen in dieser Arbeit nicht zum Einsatz, stellen aber eine potentielle Leistungssteigerung dar, wenn unterschiedliche Voxelkarten aus unterschiedlichen Datenquellen befüllt werden.

Abgesehen vom Datenaustausch geschieht jedoch auch das Starten eines Kernels auf dem Device nicht latenzfrei. Daher ist unabhängig von der Verwendung von Streams bei jeder Operation abzuwiegen, ob der zusätzliche Aufwand für einen Kontextwechsel gerechtfertigt ist, oder ob die Aufgabe nicht in kürzerer Zeit direkt auf dem Host ausführbar ist.

Lastbalancierung

Bei einigen Aufgabentypen ist der Berechnungsaufwand einzelner Arbeitspakete im Vorfeld nicht bestimmbar, wodurch eine gleichmäßige Aufteilung auf CUDA Kernel nicht gewährleistet werden kann. In solchen Fällen ist es zur Laufzeit wichtig, zyklisch oder anhand eines Kriteriums zur Auslastungsbestimmung eine Neuverteilung der Aufgaben durchzuführen, um brachliegende Rechenkerne zu vermeiden. Dieses Prinzip der Lastbalancierung wurde beispielsweise von Steinberger et al. im *Whippletree Load-Balancer* generisch implementiert [190]. Auch die GPU Kollisionsprüfung *gProximity* setzt Lastbalancierung ein [130]. Der dort verwendete Ansatz inspirierte die Octree-Implementierung dieser Arbeit, wurde dabei jedoch um eine probabilistische Komponente erweitert, die zum Ziel hat, Threads mit ähnlichen Laufzeiten in Blöcken zusammenzufassen. Dafür nutzt sie eine Bewertungsfunktion, die auf die auszuführenden Arbeitselemente angewendet wird. Details dazu finden sich in Abschnitt 5.5.2. Weitere Arbeiten zum Thema stammen aus dem High-Performance-Computing Bereich (Chen et al. [57]) und skalieren

das Prinzip auch über mehrere GPUs [169]. Vergleichbar ist auch das *Work Stealing* von Cederman et al. aus [55].

3.3. Fazit

Das Kapitel stellte die heterogene Parallelverabeitung mit CUDA auf Basis von GPUs und CPUs vor. Verbreitete Konzepte wurden beschrieben und praktisch motiviert. Dabei wurde deutlich, dass für eine effiziente Umsetzung von parallelen Algorithmen sehr genaue Kenntnisse über die Datenabhängigkeiten innerhalb der Problemstellung, als auch über die Zielhardware nötig sind. So wurde die Entscheidung, welche Algorithmen für welche Zielplattform umgesetzt werden, vorrangig anhand der benötigten Dynamik ihrer Datenstrukturen getroffen. Planungsverfahren, die zur Laufzeit Graphen aufbauen und regelmäßig umorganisieren, residieren auf dem Host, während bspw. die Sensordatenvorverarbeitung mit konstanter Nutzdatengröße auf der GPU stattfindet. Die gesammelten Erkenntnisse werden in den folgenden Kapiteln für die Implementierung von GPU-Voxels herangezogen.

4. Perzeption und Modellierung

Um sich in einer veränderlichen Umwelt zurechtzufinden, wird die Fähigkeit zur Wahrnehmung des Umfelds benötigt. Der Mensch kombiniert hierfür unterbewusst und intuitiv seine sechs Sinne. In der Robotik stehen dagegen meist nur wenige und wesentlich eingeschränktere „Sinne“ zur Verfügung, ebenso ist ihre Fusion (noch) nicht selbstverständlich. Da die entwickelten Techniken zur Kollisionserkennung auf visuellen Daten beruhen, beschränkt sich auch diese Arbeit auf die visuelle Perzeption der Umwelt. Im Hinblick darauf beleuchtet dieses Kapitel daher zunächst passende Sensoren, bevor dann unterschiedliche Umwelt- und Egomodellierungen analysiert werden. Abschließend wird ein Verfahren zur Bewegungsprädiktion sowie eine Simulation zur Generierung von Punktwolkendaten vorgestellt.

4.1. Visuelle Sensorik

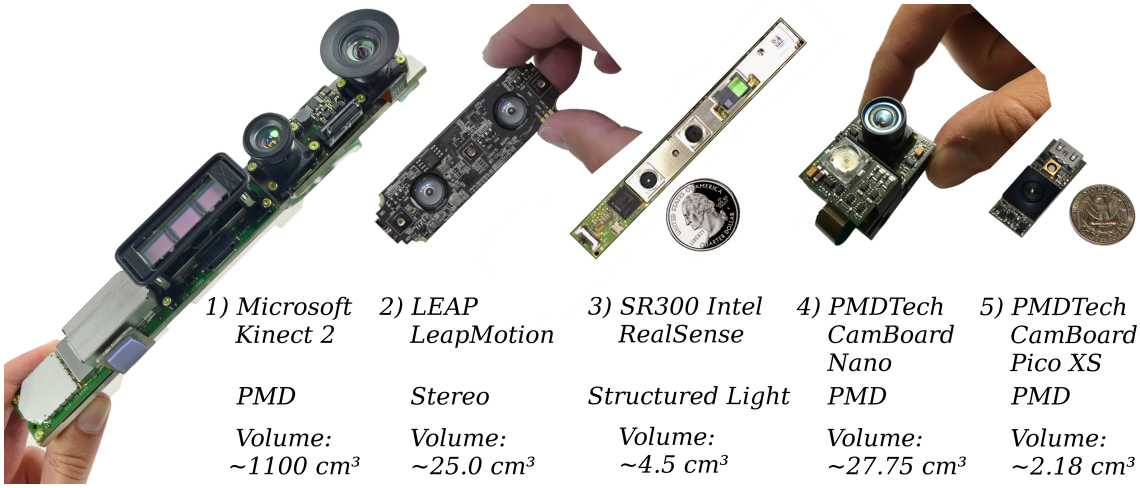


Abb. 4.1.: Vergleich des Funktionsprinzips und des Bauraumes unterschiedlicher Tiefenkameras. Bild veröffentlicht in [8].

Sensoren dienen im Allgemeinen dazu, eine physikalische Größe in ein elektrisch messbares und interpretierbares Signal umzuwandeln. Dabei wird zwischen *externe* und *internen* Sensoren unterschieden. Erstere ermöglichen es einem Roboter, seine Umwelt wahrzunehmen. Im Gegensatz dazu dienen *internen* Sensoren zur Erfassung des Roboterzustandes, beispielsweise seiner Gelenkwinkel. Sie spielen eine Rolle bei der Aktualisierung des weiter unten beschriebenen Robotermodells und werden hier nicht näher betrachtet. Der Fokus liegt dagegen auf visuellen Sensoren, da diese berührungslos messen und eine große Bandbreite und Datenvolumen an Informationen bereitstellen können. Anderen

Messprinzipien, wie z.B. akustische oder taktile Sensoren, eignen sich nur in Sonderfällen für die Kollisionserkennung, da sie entweder eine zu geringe örtliche Auflösung aufweisen oder nicht alle relevanten Hindernisse detektieren können.

Klassische visuelle Sensoren sind Intensitätskameras (Graustufen- oder Farbkameras), deren Daten sich durch vielfältige Verfahren interpretieren lassen, um beispielsweise Gegenstände oder Personen zu klassifizieren. Bedingt durch das Messprinzip, das eine Projektion auf eine Bildebene erfordert, sind diese Kameras jedoch nicht in der Lage, Tiefeninformationen der erfassten Szene zu messen. Da sich diese Arbeit jedoch mit der Kollisionserkennung im 3D-Raum befasst, werden folglich Sensoren benötigt, welche Distanzen messbar machen. Hierbei unterscheidet man zwischen *passiven* und *aktiven* Sensoren, je nachdem ob zusätzliche Energie emittiert wird oder der Sensor ausschließlich mit Umgebungslicht arbeitet.

Eine Klasse von aktiven Tiefensensoren bildet LIDAR (Light Detection And Ranging): Hier tastet ein einzelner Laserstrahl die Umgebung ab, indem er mittels mechanischer Vorrichtungen um zwei Achsen rotiert wird. Die Laufzeit, des von den Oberflächen zurückgeworfenen Lichtes, kann gemessen und somit die Distanz bestimmt werden. Bedingt durch die Abtastbewegung vergeht eine gewisse Zeit, bis die Szene vermessen wurde (je nach Auflösung bis zu mehreren Sekunden). Daher ist LIDAR nur bedingt für dynamische Szenen geeignet.

Relevanter ist hingegen die Klasse der Sensoren, die in einer einzigen Aufnahme ein ganzes Feld aus Messwerten erzeugen. Geräte dieser Art waren noch bis vor einigen Jahren sehr hochpreisig und wiesen ein großes Bauvolumen auf. Inzwischen sind jedoch Kameras aller drei folgenden Messprinzipien in kompakter Bauweise (vgl. Abb. 4.1) auch für Privatanwender erschwinglich:

- **Stereokameras** bestehen aus zwei konventionellen, passiven Intensitätskameras, die nach biologischem Vorbild Punkte in der Szene triangulieren. Auch wenn beide Kameras hochauflösende Bilder liefern, kann die Menge der vermessenen Punkte in der Szene stark schwanken, da eine Triangulation nur mit kontrastreichen Strukturen in den Bildern funktioniert.
- **Strukturiertes Licht** aus einem Musterprojektor ersetzt hier eine der Stereokameras und erzeugt künstliche Texturen (örtliche Modulation) auf der Szene, die mit der Kamera detektierbar sind. Aus dem bekannten Abstand zwischen Projektor und Sensor kann auf die Distanz der erkannten Muster geschlossen werden.
- **PMD Sensoren** (Photonic Mixture Devices) arbeiten nach dem *Time of Flight* Prinzip. Sie senden sinusförmig moduliertes Licht (zeitliche Modulation) aus und messen die Phasenverschiebung der Reflexionen. Die Sensoren sind, wie andere Kameras auch, im CMOS Verfahren herstellbar.

Als Messergebnis erzeugen alle drei Kameraklassen zunächst 10 bis 30 mal pro Sekunde ein Tiefenbild, also eine 2D-Matrix aus Distanzwerten. Mit Hilfe der intrinsischen Kalibrierungsparameter der Kameraoptik lassen sich daraus 3D-Punktwolken berechnen. Anzumerken sei jedoch, dass zwar von 3D-Sensoren gesprochen wird, es sich tatsächlich aber um 2,5D Sensoren handelt, da die visuelle Messtechnik nicht hinter oder in Objekte blicken kann. Deshalb existiert keine echte bijektive Abbildung der 3D-Szene auf das Messergebnis, da nicht jeder Punkt im Raum, der im Sichtfeld der Kamera liegt, eine

Entsprechung in der Aufnahme findet. Somit kann ein einzelnes Tiefenbild nicht dazu verwendet werden, das Volumen eines Objektes zu bestimmen. Weiterhin kann es durch Verschattungen zu Diskontinuitäten in den Messdaten kommen. Beides ist in späteren Verarbeitungsschritten zu berücksichtigen.

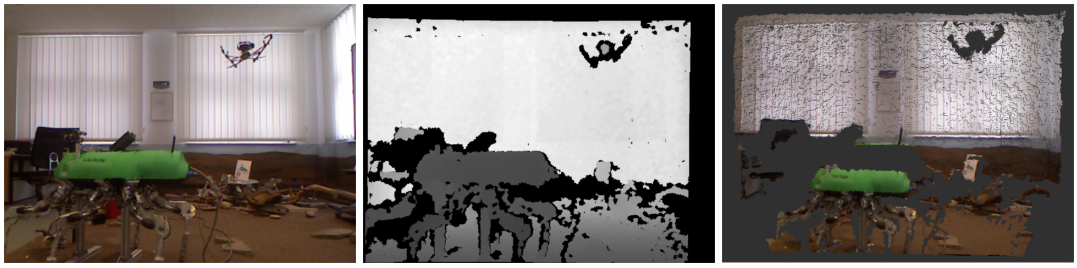
4.1.1. Registrierung von Farb- und Tiefendaten

Da Tiefenbilder an sich keine Farb- oder Intensitätsdaten enthalten, basieren viele Kameras, die für den Privatgebrauch hergestellt werden, auf der Kombination eines Tiefensensors mit einer konventionellen RGB-Kamera. Sie stellen also so genannte Red, Green, Blue, Depth (RGBD)-Kameras dar. Um eine Zuordnung zwischen den Tiefen- und Farbwerten zu schaffen (siehe Abb. 4.2), müssen beide Kameras gegeneinander kalibriert werden. Da sich die beiden Sensoren auf derselben horizontalen Achse befinden, stimmen ihre vertikalen Komponenten überein. Die vertikale Zuordnung von Messpunkten kann nach [189] wie folgt bestimmt werden:

$$u_{\text{RGB}} = \vec{K}_{\text{RGB}} \cdot dis\left(\vec{R}_{\text{RGB}}\left(\vec{x}_{\text{IR}} - \vec{c}_{\text{RGB}}, \vec{k}_{\text{RGB}}\right)\right) \tag{4.1}$$

Hierbei steht u_{RGB} für die Komponente im Farbbild, die dem Pixel $\vec{x}_{\text{IR}}$ zugeordnet wird. Als bekannt vorausgesetzt wird die intrinsische Kalibriermatrix $\vec{K}_{\text{RGB}}$ der Farbkamera, die extrinsische Kalibrierung der Farbkamera gegenüber der IR-Kamera über $\vec{R}_{\text{RGB}}$ und $\vec{c}_{\text{RGB}}$, und ihre Verzerrungsparameter $\vec{k}_{\text{RGB}}$. Die optische Entzerrung geschieht über die Funktion $dis()$.

Eine genaue Zuordnung der Farb- und Tiefeninformationen ist in dieser Arbeit für Bewegungs-Prädiktion aus Abschnitt 4.6 wichtig.



(a) RGB-Bild (b) Registriertes Tiefenbild (c) Eingefärbte Punktwolke

Abb. 4.2.: Beispiel der Komponenten einer RGBD-Aufnahme aus [27]. Die 3D-Struktur der Flugdrohne ist in der gegebenen Distanz nicht mehr aufzulösen. Schwarz dargestellte Pixel repräsentieren Stellen, an denen keine Tiefenwerte vorliegen. In der gewählten Darstellungsperspektive (die nicht der Kamera-Perspektive entspricht) sind die Abschattungen, welche aufgrund der 2,5D-Datenstruktur entstehen, deutlich sichtbar.

4.1.2. Untersuchte Tiefenkameras

Im Folgenden sollen konkrete Kameras kurz vorgestellt werden, die in dieser Arbeit untersucht und verwendet wurden. Einige der Sensoren wurden aus dem aktuellen Bedarf

Bildauflösung:	640×480	Tiefenmessbereich:	$0,8$ - 4 m
Bildwiederholfrequenz:	$\sim 30\frac{1}{s}$	Pixelgröße bei 2 m:	$3,4 \times 3,6$ mm
Tiefenauflösung bei 2 m:	12 mm	Öffnungswinkel:	$57\,° \leftrightarrow \quad 43\,° \updownarrow$

Tab. 4.1.: Hardwarespezifikation des ersten Kinect-Modells aus [37].

heraus entwickelt, Verbraucherelektronik mit integrierter Gestenerkennung auszustatten. Diese Tiefenkameras sind für den Nahbereich optimiert und weisen eine sehr kleine Bauform auf. Somit eignen sie sich für den Einbau in oder nahe am Endeffektor, was z.B. in der Greifplanung von Vorteil ist, wie in Unterabschnitt 7.2.7 beschrieben. Andere Sensoren entstammen der Spieleindustrie, sowie der Automatisierungstechnik.

Kinect Dieser von Microsoft seit 2010 für den Spielemarkt vertriebene Sensor machte die 3D-Datenerhebung in der Robotik zu einem Routineproblem. Es handelt sich um eine aktive RGBD-Kamera, die mit einem Infrarot-Musterprojektor arbeitet und verhältnismäßig hoch auflösende Tiefenbilder erzeugt. Die technischen Einschränkungen (siehe Tab. 4.1) der Kinect sind in den meisten Robotikanwendungen hinnehmbar. So funktioniert der Sensor nicht in hellem Sonnenlicht, was ihn für den Einsatz im Freien unbrauchbar macht. Der Messbereich ist auf 0,8 m bis ca. 4 m beschränkt und bei der Nutzung mehrerer Sensoren sollten sich deren Sichtfelder nicht überlappen, da die projizierten Muster nicht kameraindividuell sind. Durch einen integrierten Prozessor (zur Weiterverarbeitung der Kameradaten zu einem artikulierten Skelettmodell) weist die Kamera einen großen Bauraum auf.

Asus Xtion Ein Nachbau der Kinect ist von Asus verfügbar. Diese Kamera zeichnet sich durch ihre kleine Bauform (keine komplexe Datenverarbeitung in der Kamera) und ihren geringeren Stromverbrauch aus (siehe Abb. 4.1). Allerdings strahlt die Infrarotlichtquelle weniger intensiv als bei der Kinect, weshalb die maximal messbaren Distanzen nur bis zu 3 m verlässlich sind.

Intel RealSense Intel nutzt in dieser Kamera eine neue, dem DLP-Prinzip (Digital Light Processing) ähnliche, Projektionstechnik, bei der ein oszillierender Mikrospiegel eine Laserlinie ablenkt. Wie auch bei der Kinect wird das damit erzeugte Muster von einer versetzt angebrachten Infrarot-Kamera aufgenommen und zu einem Tiefenbild umgerechnet. Der Sensor ist in zwei Ausführungen für den Nah- und Fernbereich erhältlich. In dieser Arbeit wurde eine RealSense SR300 für die Greifplanung (siehe Abschnitt 8.10) im Nahbereich verwendet. Der sehr kleine Sensor erzeugt 640×480 Bildpunkte und misst Distanzen zwischen 0,11 und 1,2 m.

PMD Nano und Basler TOF Kamera Beide Kameras basieren auf dem oben beschriebenen TOF Prinzip, unterscheiden sich jedoch stark in ihrem Bauvolumen. Während die PMD Nano nur wenige Kubikzentimeter groß ist, misst die Basler TOF ca. 15×8 cm. Bedingt durch die miniaturisierte Lichtquelle reicht der Messbereich der Nano auch nur bis ca. 2 m. Dieser ist bei der Basler TOF Kamera mit 0 m bis 13 m bei einer Auflösung von

640×480 Pixeln und einer Bildrate von 20 FPS wesentlich größer. Negativ fiel bei Tests mit beiden Kameras allerdings ihr Signal/Rausch-Verhältnis und eine Kissenverzerrung der Distanzwerte auf, weshalb sie hier in Experimenten nicht eingesetzt wurden.

Leap 3D Die kleine Leap 3D-Stereo-Kamera arbeitet mit einer zusätzlichen Infrarot Beleuchtung und einer Fischaugen-Stereooptik. Sie wurde entwickelt, um die Finger eines Benutzers zu detektieren und somit als alternatives Eingabegerät zu fungieren. Versuche, die Bilder der beiden Kameras manuell zu einem Disparitätsbild zu fusionieren, um daraus Punktwolken zu berechnen, scheiterten an der intrinsischen Kalibrierung. Da diese bedingt durch die Fischaugenoptik nicht gelang, war auch dieser Sensor nicht nutzbar.

4.1.3. Sensordatenverarbeitung

Nachdem unterschiedliche, passende Sensoren ermittelt wurden, erläutert dieser Abschnitt die in Abb. 4.3 gezeigte Verarbeitungskette, mit der Kameradaten in Voxel umgewandelt werden. Das Ausgangsmaterial, das alle verwendeten 3D-Kameras liefern, sind Tiefenbilder, also 2D-Felder aus Distanzinformationen. Diese werden auf die GPU kopiert, da alle weiteren Schritte sehr gut parallel ausführbar sind und alle Ergebnisse direkt auf der GPU verbleiben können. Zunächst werden die Tiefenbilder als Graustufenbild interpretiert, wodurch sich regulärer Bildverarbeitungsalgorithmen als erste Filterschritte anwenden lassen (Min/Max-, Ausreißer-, Rauschfilterung, zur Entfernung ungültiger Messwerte). Die vorverarbeiteten Daten werden dann mit Hilfe der intrinsischen Kameraparameter zu einer Punktwolke im 3D-Raum projiziert und über die extrinsischen Parameter in das Weltkoordinatensystem transformiert. In einem letzten Schritt wird die Punktwolke anhand des verwendeten Voxelrasters diskretisiert und je nach genutzter Datenstruktur anhand ihres Morton-Codes sortiert. Zuletzt müssen dann die Voxel, in welche die Messpunkte fallen, entsprechend des Voxeltyps (siehe Abschnitt 5.1) angepasst werden. Dieser Prozess der Voxelumwandlung wird in Abschnitt 4.3 noch detailliert erläutert.

Die verwendeten 3D-Sensoren können lediglich Messpunkte auf sichtbaren Oberflächen der Umwelt erzeugen. Basierend auf der einfachen Annahme, dass das Volumen im Sichtkegel, der sich zwischen Sensor und den detektierten Objekten befindet, freier Raum sein muss (da die Messung sonst nicht in ihrer Form möglich gewesen wäre), kann dennoch zwischen unbekanntem und freiem Raum unterschieden werden. Ein Verfahren zur Berechnung des Freiraumes ist in Unterabschnitt 4.3.1 dargelegt.

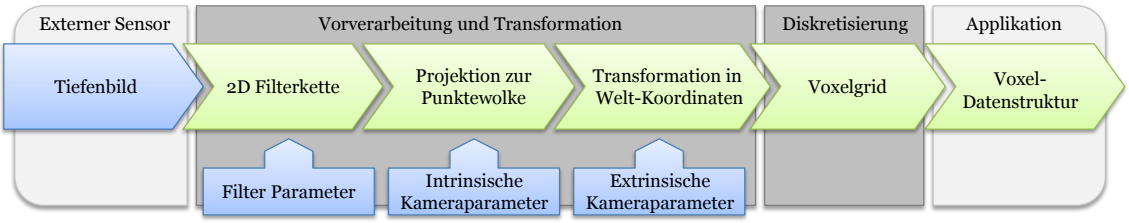


Abb. 4.3.: Verarbeitungskette zur Umwandlung von Tiefenbildern in Voxel. Grün dargestellte Schritte laufen auf der GPU ab.

4.1.4. Sensormodell

Eine exakte probabilistische Modellierung des Sensormodells der unterschiedlichen verwendeten Tiefenkameras liegt außerhalb der Möglichkeiten dieser Arbeit. Alternativ wurde ein sehr einfaches Sensormodell verwendet, das lediglich die Entfernung der Messungen berücksichtigt. Dieses Modell ist allgemein genug, um für alle genutzten Klassen von Sensoren praktikabel zu sein.

Als Rechtfertigung wurden die Untersuchungen von Khoshelham et al. [117] herangezogen, in denen der Kinect bescheinigt wird, dass ihr Messrauschen quadratisch mit der gemessenen Entfernung zunimmt und bei $5\,\mathrm{m}$ bereits $4\,\mathrm{cm}$ beträgt. Weiterhin sinkt die messbare Tiefenauflösung über die Distanz, so dass die Kinect am Ende ihres Messbereiches nur noch in $7\,\mathrm{cm}$ Schritten auflöst. Daher wurden bei der Verwendung einer Kinect alle Messwerte jenseits von $3\,\mathrm{m}$ verworfen, womit auch die zu erwartenden Fehler unterhalb der gängigen Voxelauflösung von 2–$4\,\mathrm{cm}$ liegt. Ähnlich wurde bei PMD Kameras vorgegangen, deren Messbereich noch wesentlich restriktiver eingeschränkt wurde.

Nguyen et al. stellen in [151] fest, dass auch das Messrauschen linear mit der lateralen Distanz zur Hauptachse des Sensors zunimmt, was in dieser Arbeit jedoch nicht berücksichtigt ist. Messfehler aufgrund des Winkels, unter dem eine Oberfläche vom Sensor gesehen wird, spielen laut Nguyen erst ab $70°$ eine Rolle. Da noch steilere Messungen jedoch meist direkt vom Sensor verworfen werden, wurden auch diese nicht im Sensormodell beachtet. Einen absoluten Versatz der Messungen, der abhängig von der Temperatur der Kamera ist, stellen Choo et al. in [60] fest, weshalb sie eine 30-minütige Aufwärmphase empfehlen, bevor mit relevanten Messungen begonnen wird.

Weitere Fehlerquellen, die bei allen Sensoren beobachtet wurden, sind so genannte *Jump-Edges*, die an Kanten von Objekten auftreten, hinter welchen weitere sichtbare Oberflächen liegen. In diesem Fall generieren die Kameras häufig fehlerhafte Messpunkte hinter der Kante, entlang des Sichtlinie des Sensors. Da in dieser Arbeit jedoch von bewegten Sensoren ausgegangen wird, treten solche geisterhaften Messpunkte örtlich stark verteilt auf. Statistisch und praktisch betrachtet, werden sie daher verlässlich von korrekten Messungen überschrieben.

Auch die Anfälligkeit der Sensoren gegen Fremdlicht, insbesondere Sonnenlicht ist problematisch. Sie wurde in den Versuchen durch angepasste Umgebungsbedingungen vermieden.

Ausgehend von den Messdaten soll nun ein passendes Umweltmodell zu ihrer Aggregation und Weiterverarbeitung gefunden werden.

4.2. Umweltmodell

Grundlegender Bestandteil eines Robotersystems ist sein Umweltmodell. Dieses erfüllt sehr unterschiedliche Aufgaben, von der Selbstlokalisierung, über die Bewegungsplanung, bis zur Verwaltung von detektierten Objekten. Da entsprechend vielfältige Umsetzungen existieren, soll die folgende Taxonomie aus der Robotikvorlesung von Prof. Dillmann einen strukturierten Vergleich ermöglichen. Sie unterscheidet die Kategorien:

- **Abstraktionsniveau**: Geometrisch, topologisch, semantisch
- **Umweltbedingung**: Statisch, dynamisch, bekannt, unbekannt
- **Operationsraum**: 2D, 2,5D, 3D
- **Informationsgehalt**: Pfade, Freiraum, Objekte
- **Anwendungsgebiet**: Gemischt, strukturiert, unstrukturiert
- **Art der Modellierung**: Exakt, approximierend

Da nicht alle Inhalte sinnvoll in einem Modell vorgehalten werden können, lassen sich auch mehrere Modelle kombinieren, was allerdings ein Konsistenzproblem in sich birgt. Oftmals wird ein Umweltmodell aus Sensordaten mit einem abstrakten Objektmodell aus bekannten Entitäten und ihren Posen im Raum kombiniert. Diese Arbeit hingegen unterscheidet nicht zwischen zwei Modellen, sondern annotiert Objektinformationen direkt in den zugehörigen Teilen des Umweltmodells und vermeidet so potentielle Inkonsistenzen.

Einige weit verbreitete Arten der Modellierung sollen nun kurz beschrieben und in die gegebenen Kategorien eingeordnet werden. In direktem Zusammenhang mit dem Modell stehen auch die möglichen Verfahren zur Kollisionsdetektion, auf die in Abschnitt 6.1 eingegangen wird. Da für eine Kollisionserkennung immer geometrische Modelle benötigt werden, sind hier keine rein topologischen oder semantischen Ansätze berücksichtigt, wie beispielsweise Datenbanken aus abstrakten Objektinformationen.

4.2.1. Oberflächen beschreibende Modelle

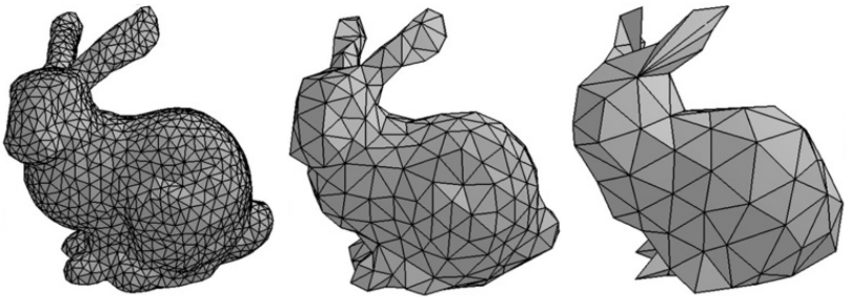
Die verbreitetste Art der 3D-Modellierung stammt aus dem Bereich der Computergrafik und beschränkt sich auf die geometrische Beschreibung von Oberflächen. Liegt exaktes Wissen über die zu modellierenden Geometrien vor, können ihre Flächen mittels funktionaler Methoden (bspw. Polynome, NURBS) repräsentiert und mit beliebiger Genauigkeit interpretiert werden. Häufiger anzutreffen sind jedoch approximative Methoden, bei denen beliebige Formen durch Polygon-Netze angenähert werden, wie in Abb. 4.4a zu sehen ist. Die kleinsten Einheiten der Netze sind fast immer Dreiecke, da diese planar und konvex sind, was ihre Berechnung und Darstellung vereinfacht. Ein Vorteil dieser Modellierung ist die hohe Speichereffizienz, die durch einen örtlich variablen Detaillierungsgrad (Dreiecksgröße) erreicht wird. So lassen sich innerhalb eines Modells einheitliche Flächen mit wenig Dreiecken darstellen, während detailreiche Regionen feiner aufgelöst werden können. Alternativ lassen sich unterschiedlich fein strukturierte Versionen derselben Oberfläche erzeugen (*Level of Detail*, LOD), um je nach Anforderung dynamisch zwischen diesen zu wechseln [85].

Weiterhin lassen sich Normalenvektoren bestimmen, anhand derer die Ausrichtung der Fläche definiert wird, um Unterscheidungen wie *innerhalb / außerhalb* zu ermöglichen. Diese Oberflächennormalen sind auch zur Berechnung physikalischer Interaktion, wie Kraftübertragung oder Reibung, nötig.

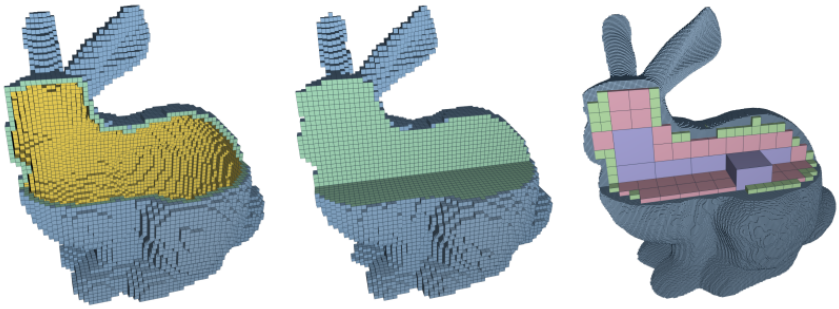
Mesh-basierte Repräsentationen sind sehr gut für die Darstellung von strukturierten, a priori bekannten Modellen geeignet. Es existieren zahllose Programme zur Erzeugung und Animierung von Oberflächen-Geometrien und für deren Darstellung. Sind die Modelle jedoch zur Laufzeit aus Sensordaten zu rekonstruieren, müssen die Punktwolken

mittels Triangulierungsverfahren wie z.b. Poisson-Meshing oder Marching-Cubes in Dreiecksnetze umgewandelt werden. Diese Tesselierung weist einen hohen Rechenaufwand auf, der mit jeder Sensoraufnahme anfällt und kubisch ($\mathcal{O}(n^3)$) mit der Anzahl an Messpunkten skaliert [61]. Daher können selbst parallelisierte Verfahren, die auf GP-GPUs ablaufen, aktuell nur ca. 10 Mio. Punkte pro Sekunde [163] verarbeiten, was gerade der Datenrate einer einzelnen Kinect-Kamera entspricht ($307\,200$ Punkte mit 25 Hz). Da die genannten Algorithmen eine szenenabhängige Parametrierung benötigen, um Annahmen zur Sichtbarkeit und Zusammengehörigkeit von Messungen zu treffen, kann es zu Fehlinterpretationen kommen, die die Dreiecksnetze für eine Kollisionsprüfung unbrauchbar machen können. Weiterhin ist die Repräsentation von Pfaden oder Bewegungen mittels Swept-Volumen aus Dreiecksnetzen sehr rechenaufwendig, wie in Abschnitt 4.5 gezeigt wird.

Der größte Nachteil von Oberflächenmodellen liegt jedoch darin, dass Freiräume oder unbekannte Regionen nicht oder nur sehr umständlich darstellbar sind.



(a) Dreiecksnetzmodell mit unterschiedlichen Auflösungen



(b) Voxel- und Octree-Modell

Abb. 4.4.: Stanford Bunny in unterschiedlichen Modellierungen [185].

4.2.2. Zusammengesetzte Primitive und generative Beschreibungen

Eine andere Klasse der Modellierung nutzt parametrisierte Primitive und aus ihnen erzeugte Vereinigungen, um Objekte zu beschreiben. Weit verbreitet im CAD-Umfeld ist das *Constructive Solid Geometry (CSG)*-Verfahren, das mittels Kugeln, Quadern oder Zylindern und Operatoren wie Schnitt, Vereinigung oder Differenz auch komplexe Geometriene darstellen kann. Die formelbasierte Beschreibung garantiert korrekte Modelle, die

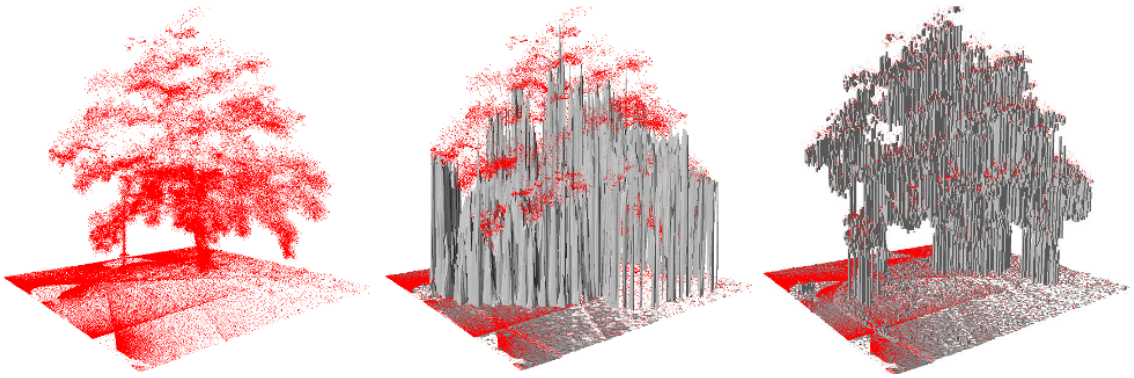


Abb. 4.5.: Unterschiedliche 2,5D Repräsentationen: V.l.n.r.: Eingabedaten als 3D-Punktwolke, Höhenkarte, Multi-Level-Höhenkarte. Adaptiert aus [203].

als Baumstruktur speicherbar sind. In der Robotik bietet sich CSG an, um Greifaufgaben zu planen. Wurden für alle Grundkörper passende Griffe vorberechnet, muss zur Laufzeit lediglich eine Approximation des zu greifenden Objektes aus den bekannten Körpern gefunden werden, um einen Griff auszuwählen [74].

Einen Spezialfall stellen so genannte Superquadriken dar. Diese geometrischen Körper werden über Formeln beschrieben, deren Parametrisierung die Gestalt grundlegend, aber stetig verändert. So kann eine einzelne Superquadrik sowohl Zylinder, Kugeln oder Quader approximieren. Auch dies wird zur Greifplanung genutzt, indem Griffe definiert werden, die sich adaptiv mit den Superquadrik Parametern verhalten. Auch hier ist zur Laufzeit lediglich die beste Approximation des Zielobjektes durch einen Superquadrik-Parametersatz zu finden [63].

Eine weitere Art der generativen Modellierung ist die Erzeugung von Rotationsflächen oder Sweeps, um komplexe Oberflächen zu beschreiben. Dafür werden Kurven oder Volumen rotiert bzw. entlang einer weiteren Kurve verschoben, um das entstehende Integral zu generieren. Die in dieser Arbeit mehrfach eingesetzten Swept-Volumen folgen einer ähnlichen Vorgehensweise. Sie sind in Abschnitt 4.5 erläutert.

Keines dieser Verfahren eignet sich jedoch zur Repräsentation von Sensordaten. Hierfür wäre mit jeder neuen Punktwolke ein komplexes Optimierungsproblem zu lösen, das bestimmt, welches Primitiv in welcher Parametrierung die Anordnung der Messpunkte am besten beschreibt.

4.2.3. Raumpartitionierende Modelle

Sollen nicht nur Oberflächen, sondern Volumendaten repräsentiert werden, bietet sich die Nutzung von raumpartitionierenden Modellen an.

Definition 8. Eine **raumpartionierende Repräsentation** teilt den darzustellenden Raum in adressierbare, disjunkte Zellen auf, die als Container für sortierte Daten dienen, oder denen Eigenschaften wie *belegt* / *frei* zugesprochen werden. Die Partitionierung kann in einem gleichförmigen Schema oder flexibel erfolgen.

Klassische Beispiele für eine flexible Partitionierung im $\mathbb{R}^3$ sind *Voronoi-Diagramme*, der *Binary Space Partitioning Tree* [84] oder *Octalbäume (Octrees)* [143]. Werden die umschließenden Geometrien rekursiv zur Unterteilung genutzt, entstehen *Bounding-Volume-Hierarchies (BVHs)* [62], auf welche im Zusammenhang mit der Kollisionsprüfung in Kapitel 6 noch genauer eingegangen wird. Weiterhin lassen sich BVHs mit Oberflächennetzen in einer hybrider Darstellungen kombinieren, bei der Oberflächenmodelle zusätzlich in grobe Hüllkörper unterteilt werden [85].

Im Gegensatz zu einer Abtastung eines darzustellenden Volumens an diskreten Punkten, bei der der Raum zwischen den abgetasteten Koordinaten undefiniert ist, repräsentiert eine einzelne Zelle eines raumpartinionierenden Modells die Menge an Information innerhalb des abgedeckten Teilvolumens. Entweder indem alle Daten, die in die Zelle fallen, in ihr gespeichert werden, oder indem alle Datenpunkte über ein probabilistisches Modell miteinander verrechnet werden.

Definition 9. Ein Spezialfall der Partitionierung ist die gleichförmige Unterteilung eines Volumens in kubische Einheiten, so genannten **Voxel** (das dreidimensionale Pendant eines Pixels). Eine Menge aus adressierbaren Voxeln, die ein zusammenhängendes, quaderförmiges Volumen bilden, wird in dieser Arbeit als **Voxel-Datenstruktur** bezeichnet.

Diese Modellierung ist geometrischer Natur, da Voxel die Lage und Dimension von Objekten im dreidimensionalen Operationsraum beschreiben können. Semantisch kann im einfachen Fall zwischen unbekanntem, freiem und belegtem Volumen unterschieden werden, im Falle von annotierten Voxeln lassen sich Volumen auch unterschiedlichen Entitäten zuordnen. Die Voxelmodellierung eignet sich ebenso für strukturierte Anwendungsgebiete mit Umweltinformationen, die z.B. aus geometrischen Modellen gewonnen werden, als auch zur Repräsentation von Sensordaten eines unstrukturierten Gebietes, wie bspw. der freien Natur. Weiterhin sind alle anfangs gelisteten Umweltbedingungen abgedeckt, da durch annotierte Voxel auch unvollständiges Wissen explizit modellierbar ist und die Verarbeitungsgeschwindigkeit dynamischen Situationen des untersuchten Anwendungsgebietes gerecht wird. Die Generierung einer Voxel-Datenstruktur aus einer Punktwolke erfolgt sehr effizient in $\mathcal{O}(n)$ anhand einer Diskretisierung der einzelnen Punktkoordinaten und der Zusammenfassung aller Messungen innerhalb einer Zelle. Spezifische Verfahren hierzu werden in Kapitel 5 vorgestellt. Gleiches gilt auch für die mehrfache Umwandlung einer bewegten Punktwolke, was zur einfachen Generierung von Swept-Volumen genutzt werden kann. Der Informationsgehalt umfasst also neben Objekten und Freiraum auch Pfade dynamischer Objekte.

Problematisch ist jedoch der Speicherverbrauch einer kanonischen Voxel-Datenstruktur, da sie nicht nur belegte, sondern auch freie Volumen explizit repräsentiert. Daher existieren zahlreiche Ansätze, die auf Kosten der Laufzeit eine Kompression der Daten durchführen. Möglich ist das Zusammenfassen gleichförmiger Volumen mittels Octrees (siehe Abschnitt 5.5), oder das Verwerfen von nicht relevanten Voxeln (siehe Voxellisten in Abschnitt 5.4).

Eine andere verbreitete Möglichkeit ist eine Dimensionsreduktion, wie sie in Abb. 4.5 zu sehen ist. Hier werden in einer 2D-Datenstruktur lediglich eine oder mehrere Höheninformationen gespeichert, um die Eingabedaten bestmöglich abzubilden. Semantische oder probabilistische Informationen sind dabei jedoch nicht sinnvoll repräsentierbar.

4.2.4. Truncated Signed Distance Functions (TSDFs)

Das Kinect-Fusion Verfahren [150] zur 3D-Modellierung von unstrukturierten Umgebungen aus Tiefenaufnahmen nutzt eine weitere Art der geometrischen Modellierung: Oberflächen werden hier mittels *Truncated Signed Distance Functions (TSDFs)* repräsentiert. Die grundlegende Datenstruktur ist auch hier eine Voxelkarte, deren Voxel jedoch keine Belegtheit, sondern den Abstand zu der ihnen am nächsten liegenden Oberfläche speichern. Voxel, die den Wert Null enthalten, repräsentieren also eine Oberfläche. Durch die Verwendung der TSDF bewegen sich die Distanzwerte im negativen Bereich, wenn sie hinter einer Fläche liegen, bzw. im positiven Bereich, wenn sie vor einer der Flächen liegen, und erreichen bereits ab einer geringen euklidischen Distanz einen Betrag von Eins. Die Vorteile dieser Modellierung liegen in der effizienten Integration aufeinanderfolgender Sensoraufnahmen von statischen Szenen, bei gleichzeitiger Minimierung des Messrauschens. Problematisch ist der hohe Berechnungsaufwand, der mit dem darstellbaren Volumen wächst: Die kontinuierliche Generierung eines Modells aus den Daten einer Kinect-Kamera lastet eine moderne GPU bereits aus, auch wenn das Volumen auf 512^3 Voxel beschränkt wird. Daher wurden TSDFs für die Kollisionserkennung nicht weiter in Betracht gezogen. Eine spätere Unterstützung wäre jedoch einfach möglich.

4.2.5. Auswahl der geeignetsten Modellierung

Eine Modellierung von Sensordaten durch Primitive, generative Beschreibungen oder TSDFs scheidet aus den genannten Gründen aus. Bei den verbleibenden Techniken decken diskretisierende Modelle alle für eine Bewegungsplanung relevanten Punkte aus der Taxonomie ab. Weiterhin ergeben sich durch ihre Verwendung mehrere Vorteile gegenüber einer Modellierung mittels Dreiecksnetzen:

- Punktwolken lassen sich wesentlich effizienter auf Voxel-Datenstrukturen abbilden ($\mathcal{O}(n)$), als auf Oberflächennetze ($\mathcal{O}(n^3)$).
- Eine Datenfusion über die Zeit oder aus mehreren Datenquellen ist inhärent innerhalb der Voxel möglich.
- Freiraum und unbekannter Raum kann explizit modelliert werden, was für eine Bewegungsplanung sehr vorteilhaft ist.
- Swept-Volumen können ohne zusätzlichen Reduktionsaufwand generiert und direkt gespeichert werden.
- Durch die Annahme einer statistischen Unabhängigkeit zwischen einzelnen Voxeln können diese einfach parallel bearbeitet werden.

Die Einschränkungen, die sich bei Voxelmodellen durch nicht vorhandene Oberflächennormalen ergeben, sind in der Bewegungsplanung nicht relevant. Prinzipbedingte Diskretisierungsfehler lassen sich außerdem zur Ausführungszeit beispielsweise durch eine Regelung des Systems (bspw. Impedanzregelung) ausgleichen. Somit ist Forschungsfrage 2 beantwortet und es kann, auch im Hinblick auf die Evaluierung in Abschnitt 8.4, die Feststellung 7 getroffen werden:

Feststellung 7. Zusammenfassend eignet sich die Voxelmodellierung am besten für die in dieser Arbeit verfolgten Ziele einer Kollisionsdetektion auf Basis von 3D-Punktwolken, weshalb sie als Grundlage in GPU-Voxels gewählt wurde.

4.3. Voxelumwandlung

Das Aktualisieren von Voxel-Datenstrukturen durch Punktwolken ist ein grundlegender Arbeitsschritt, der hier formalisiert werden soll.

Definition 10. Als **Voxelumwandlung** wird das Eintragen einer Punktwolke P in eine Voxel-Datenstruktur M bezeichnet, das mit dem Operator $\boxplus(M, P)$ beschrieben wird. Hierfür sind die Voxel, in welche die Punkte fallen, entsprechend zu aktualisieren. Um sie zu bestimmen, müssen die Koordinaten der Punkte mit dem Raster der Datenstruktur, das der Voxelkantenlänge l_{Voxel} entspricht, diskretisiert werden. Je nach Dichte der Punktwolke können bei der Diskretisierung Voxel übersprungen werden, was zu Löchern im belegten Volumen führt. Um auch unter einer beliebigen Rotation der Punktwolke solche Abtastfehler zu vermeiden, muss für den maximalen Abstand Δ_p zwischen einzelnen Messpunkten $p \in P$ gelten: $\Delta < \frac{l}{\sqrt{2}}$*. Der maximale Diskretisierungsfehler durch die Voxelumwandlung liegt bei $\frac{1}{2}\sqrt{3}l$.

*Kantenlänge des minimalen Würfels, dessen Ecken außerhalb eines Voxels liegen

Gegeben sei: Eine Voxel-Datenstruktur M und eine Punktwolke P. Dabei sei ein Voxel $V = (a, b, c, \Psi)$ das Tupel seiner Koordinaten $a, b, c \in \mathbb{Z}$ und seinem Zustand Ψ (Ψ kann je nach Voxeltyp unterschiedliche Wertebereiche aufweisen). Ein Punkt $p \in P : p = (x, y, z)$ sei das Tupel seiner Koordinaten $x, y, z \in \mathbb{R}$.

Weiterhin existiere ein Operator $\boxdot_{\ddagger}(V)$, der den Zustand Ψ eines gegebenen Voxels V entsprechend einer vom Voxeltypen abhängigen Funktion $\ddagger$ ändert.

Mit diesen Voraussetzungen lässt sich der Operator $\boxplus(M, P)$ definieren, der den Zustand einer Voxel-Datenstruktur M aktualisiert, wenn eine Punktwolke P in diese eingetragen wird:

$$\begin{aligned} \boxplus(M, P) &:= \bigcup \boxdot_{\ddagger}(V_j)\ ;\ \forall p_j \in P \text{ und } V_j \in M \\ &\text{wobei } V_j = \Big(\begin{pmatrix} \lfloor x_j / l_{Voxel} \rfloor \\ \lfloor y_j / l_{Voxel} \rfloor \\ \lfloor z_j / l_{Voxel} \rfloor \end{pmatrix}^T, \Psi_j \Big) \text{ mit } (x_j, y_j, z_j) = p_j \end{aligned} \tag{4.2}$$

Unter der Annahme einer konstanten Laufzeit für $\boxdot_{\ddagger}(V)$ ist der Aufwand der Voxelumwandlung direkt durch die Parallelisierbarkeit der Zieldatenstruktur bestimmt, da zwischen den Eingabedaten keine Abhängigkeiten bestehen.

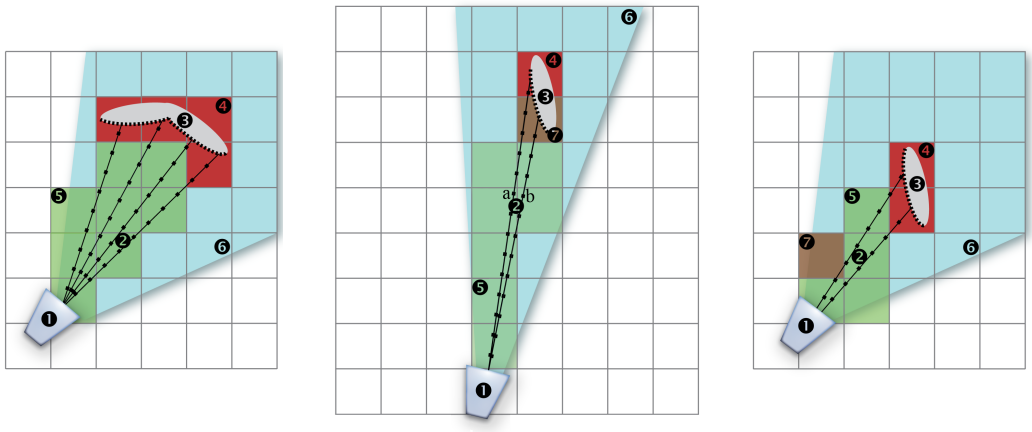
4.3.1. Freiraumbestimmung

Aufgrund der genutzten visuellen Messprinzipien können die verwendeten 3D-Sensoren nur Messpunkte auf den sichtbaren Oberflächen der Umwelt erzeugen. Um dennoch eine Unterscheidung zwischen unbekanntem und freiem Raum zu ermöglichen, kann die vereinfachte Annahme getroffen werden, dass das Volumen im Sichtkegel zwischen dem Sensor und den detektierten Objekten freier Raum sein muss, da die Messung sonst so in ihrer Form nicht möglich gewesen wäre. Basierend auf dieser Annahme können die freien Voxel mittels einem Raycasting Verfahren bestimmt werden, welches seinen Ursprung in der Computergrafik hat. Dort wird es verwendet, um den Schnittpunkt der Sichtstrahlen einer Kamera mit virtuellen Objekten zu bestimmen. Ganz ähnlich wird in dieser Arbeit eine Abwandlung des *Bresenham Algorithmus* [51] eingesetzt, um die Menge F der Voxel zu berechnen, die in einer Voxel-Datenstruktur auf der Strecke zwischen dem Sensorursprung und einem Messpunkt liegen (vgl. Abb. 4.6a). Dabei sind jedoch zwei Beobachtungen zu berücksichtigen, die sich aus dem pyramidenförmigen Sichtfeld des Sensors ergeben: 1) Die Menge F enthält viele Voxel mehrfach, da die Voxel, die nahe am Sensor liegen, von den Strahlen vieler weiter entfernter Messpunkte geschnitten werden (das Verhältnis steigt quadratisch mit der Distanz h zum Sensor). 2) Die Anzahl $|F|$ der freien Voxel übersteigt die Anzahl der belegten Voxel $|O|$ bei einer Objektdistanz h von 300 cm und einer Voxelkantenlänge von 1 cm bereits um zwei Größenordnungen (da $|F| = h/3 \cdot |O|$). Bei 0,3 Mio. Messpunkten der Kinect wären dies 30 Mio. Freiraumvoxel.

Um die große Menge an Voxeln, die sich aus Beobachtung 2) ergeben, effizient zu ermitteln, läuft das Raycasting parallel für alle Messstrahlen. Beobachtung 1) zeigt jedoch, dass dabei mehrere Threads gleichzeitig auf dieselben Voxel zugreifen können. Aus Effizienzgründen wurde dennoch auf eine Serialisierung mittels atomarer Operationen verzichtet, auch wenn im Falle von probabilistischen Voxeln das mehrfache Dekrementieren der Belegtheitswahrscheinlichkeit das korrekte Verhalten wäre. Um Fehler zu minimieren, wurde die Strahlenverfolgung invertiert und läuft ausgehend vom Messpunkt in Richtung Kamera, wodurch die Strahlen unterschiedlich lang sind, und die Threads somit die kameranahen Voxel asynchron traversieren. Weiterhin stellt die Speicher-Koherenz sicher, dass jeder betroffene Voxel mindestens einmal als frei markiert wird. Praktische Tests haben letztendlich bestätigt, dass die verbleibenden Fehler so selten auftreten, dass sie vernachlässigbar sind.

Um ein nachträgliches Überschreiben von belegten Voxeln durch das Raycasting zu vermeiden (vgl. Abb. 4.6b), werden zunächst alle Freiraumvoxel markiert, und erst danach die Hindernisse eingetragen. Abtastfehler, wie sie in Abb. 4.6c dargestellt sind, können zwar nicht ausgeschlossen werden, sie stellen jedoch auch kein Problem dar, da sie nur in einem sehr kleinen Randbereich des Sichtkegels auftreten.

Die letztendliche Implementierung des Raycastings unterscheidet sich zwischen Voxelkarte und Octree: Während bei einer Voxelkarte die Markierung direkt ausgeführt wird (setzen der Voxeleigenschaften im Falle einer binären Voxelkarte, bzw. aktualisieren der Belegtheitswahrscheinlichkeiten im Falle von probabilistischen Voxeln), wird beim Octree noch ein Zwischenschritt eingeführt. Auch dort dienen Voxelkarten aufgrund ihres effizienteren wahlfreien Zugriffes als Datenstruktur für das Raycasting. Ist dieses abgeschlossen, lassen sich bereits auf Basis der Voxelkarten zusammenhängede Freiräume



(a) Positivbeispiel (b) Falschdeklaration (c) Abtastfehler

Abb. 4.6.: Freiraumberechnung vereinfacht für den zweidimensionalen Fall. Legende: 1) Sensor, 2) Strecke |Sensor-Messpunkt|, 3) Hindernis, 4) Belegte Zelle, 5) Freies Feld, 6) Sichtkegel, 7) Fehlerhafte Markierung

zusammenfassen, bevor die Ergebnisse in den Octree übertragen werden. Weiterhin ist es damit möglich, den Freiraum in einer gröberen Auflösung zu bearbeiten und somit Berechnungsaufwand einzusparen, ohne die Auflösung der Hindernisvoxel zu beeinträchtigen (siehe Abb. 4.7).

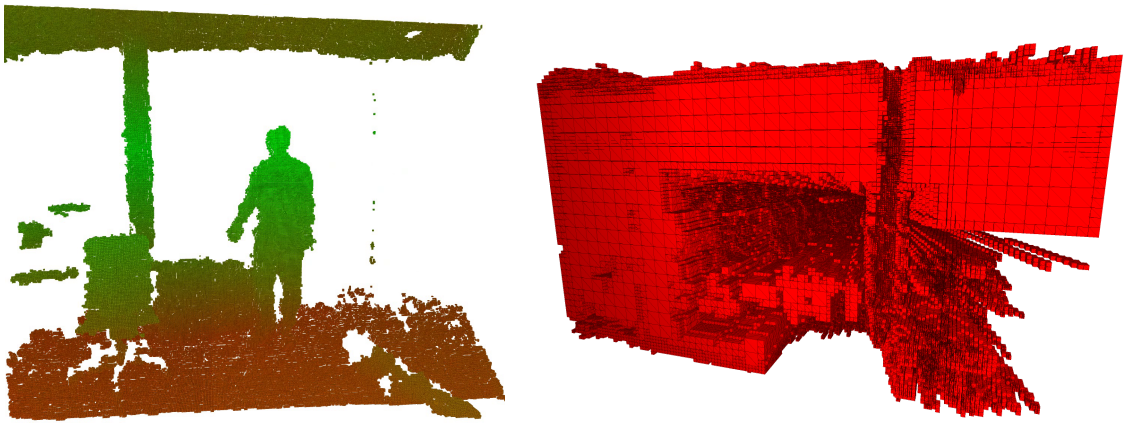
Ein Beispiel für das Raycasting ist in Abb. 4.7 gezeigt. Hierbei wird das Umweltmodell aufgebaut, während die Kamera ein Objekt umkreist.

4.4. Roboter-Modell

Ebenso wie das Umweltmodell dient auch das Egomodell eines Roboters vielen unterschiedlichen Zwecken, wie beispielsweise der Zustands- oder Fehlerdiagnose. Für die Kollisionsdetektion ist jedoch ausschließlich ein animiertes geometrisches Modell relevant. Um dieses effizient verarbeiten zu können, soll es ebenso wie die Umwelt über eine diskretisierende Voxel-Datenstruktur repräsentiert werden. Die folgenden Abschnitte beschreiben die Generierung und Aktualisierung eines Voxel-Egomodells.

4.4.1. Artikulierte Robotermodelle

Das Modell eines beweglichen Roboters muss die Geometrie und die kinematische Konfiguration eines Mehrkörpersystems aus rigiden Teilen beschreiben können, um die relativen Bewegungen der einzelnen Roboterglieder im $SE(3)$ eindeutig abzubilden [125]. Dafür wird meist auf so genannte Szenengraphen zurückgegriffen. Diese stellen baumartige Strukturen mit verketteten geometrischen Transformationen dar, die ausgehend von einem Welt-Koordinatensystem bis zu den einzelnen Elementen des Roboters reichen. Jede Stelle im Baum repräsentiert ein Koordinatensystem, mit dem ein geometrisches Modell verknüpft werden kann. Zur Laufzeit lassen sich die Transformationen durch



(a) Eigabeszene aus Sicht der Kamera (ca. 200 kVoxel)

(b) Freiraum aus entgegengesetzter Richtung (ca. 5 MVoxel)

Abb. 4.7.: Beispiel der Freiraumberechnung mittels Raycasting. Der Freiraum wird im Octree zu größeren Voxeln zusammengefasst.

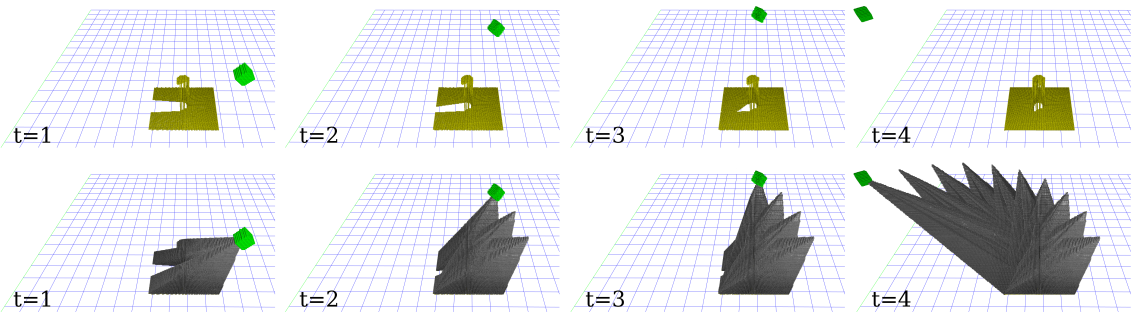


Abb. 4.8.: Aufbau des Umweltmodells aus zusammengeführten Punktwolken: Während der Sensor (grüner Würfel) um das Objekt wandert (gelber Zylinder auf Bodenebene) berechnet ein Raycasting Algorithmus den tatsächlichen Freiraum (graue Pyramiden). Alle nicht einsehbaren / verschatteten Regionen verbleiben als unbekannt modelliert.

Gelenkwinkel oder andere Sensormessungen anpassen, womit sich auch die Modelle im Raum bewegen. Zwei verbreitete Konventionen, um diese Transformationsketten mathematisch zu definieren, sind in GPU-Voxels umgesetzt worden:

Denavit Hartenberg (DH) Konvention: Unter Nutzung der DH-Konvention [68] kann für jedes Robotergelenk n aus den vier geometrischen Parametern θ_n, d_n, a_n und α_n eine homogene Matrix T bestimmt werden, die die Anordnung des n-ten Rotations- oder Schubgelenks gegenüber seiner Basis $n-1$ beschreibt.

$$
\begin{aligned}
{}^{n-1}T_n &= \mathrm{Rot}(z_{n-1}, \theta_n) \cdot \mathrm{Trans}(z_{n-1}, d_n) \cdot \mathrm{Trans}(x_n, a_n) \cdot \mathrm{Rot}(x_n, \alpha_n) \\
&= \begin{pmatrix} \cos\theta_n & -\sin\theta_n \cos\alpha_n & \sin\theta_n \sin\alpha_n & a_n \cos\theta_n \\ \sin\theta_n & \cos\theta_n \cos\alpha_n & -\cos\theta_n \sin\alpha_n & a_n \sin\theta_n \\ 0 & \sin\alpha_n & \cos\alpha_n & d_n \\ 0 & 0 & 0 & 1 \end{pmatrix}
\end{aligned} \tag{4.3}
$$

Durch die Multiplikation aller n Matrizen entsteht eine Kette, die die Transformationen von der Roboterbasis bis hin zum Endeffektor beschreibt. Kürzere Teilketten können verwendet werden, um die Pose der Roboterglieder zu berechnen. Durch die Änderung von θ_n bei Rotationsgelenken bzw. α_n bei Schubgelenken kann die Kinematik und somit die Geometrie des Roboters bewegt werden.

Unified Robot Description Format (URDF): Eine redundantere, aber intuitivere Möglichkeit ist es, die Transformationsmatrizen direkt über die Art, die Verschiebung und die Rotation der Glieder zueinander zu bestimmen [124]. Dieses Format der kinematischen Beschreibung findet sich in der Unified Robot Description Format (URDF)[1] Modellierung, die im Robot Operating System (ROS) verwendet wird. Auch hier wird letztendlich eine Transformationskette aufgebaut, über welche die Lage der einzelnen Glieder je nach Roboterpose bestimmt werden kann.

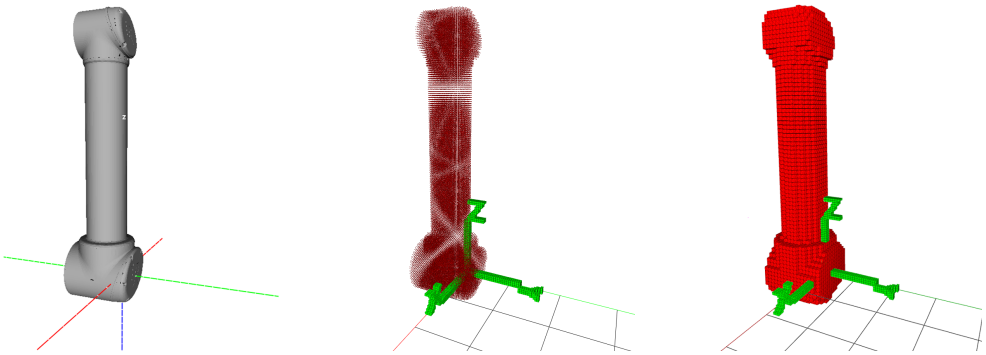
4.4.2. Voxelmodelle

Ist die Transformationskette bekannt, kann damit jeder rigide Roboterbestandteil an seine Pose bewegt werden. Während bei einer Modellierung mittels Oberflächen jedes Dreieck nach seiner Transformation direkt auf Kollision prüfbar ist, muss bei einer Voxelmodellierung ein Zwischenschritt stattfinden, da hier nicht die Modelle selber, sondern zwei Voxel-Datenstrukturen miteinander überlagert werden.

Um ein Voxelmodell eines Roboters zu erhalten, sind zunächst vorhandene CAD Oberflächenmodelle der einzelnen Glieder des Roboters in dichte 3D-Punktwolken umzuwandeln (vgl. Abb. 4.9). Dieser Prozess entspricht einer Abtastung der Oberflächenmodelle mit einem dreidimensionalen Gitter aus Messpunkten, wobei alle Punkte gespeichert werden, die auf oder innerhalb des Oberflächenmodells liegen. Hierbei muss die Auflösung der Abtastung so hoch gewählt werden, dass bei der Voxelumwandlung der Punktwolke in beliebiger Rotation keine Löcher im Modell entstehen. Dieser Diskretisierungsfehler wird vermieden, wenn gemäß Definition 10 für die Abtastdistanz Δ gilt: $\Delta < \frac{\text{Voxel-Seitenlänge}}{\sqrt{2}}$. In dieser Arbeit wurde für die Abtastung das Werkzeug *binvox*[2] eingesetzt [152]. Weiterhin muss der Ursprung des Koordinatensystems der Punktwolke im Rotationspunkt des Gelenkes liegen, welches das betrachtete Körperglied bewegt.

Die Punktwolken aller Körperteile werden einmalig untransformiert in den konstanten Speicher der GPU geladen und verbleiben dort schreibgeschützt. Verändert sich zur

[1] URDF: `http://wiki.ros.org/urdf`
[2] binvox: `http://www.patrickmin.com/binvox/`



(a) CAD-Oberflächenmodell des Roboter Oberarmes (b) Als dichte Punktwolke abgetastetes Modell (c) In Ausgangslage in Voxel umgewandelte Punktwolke

Abb. 4.9.: Voxelumwandlung eines einzelnen Roboter-Gliedes am Beispiel des Oberarmes.

Laufzeit die Position des Roboters oder seine Gelenkwinkel, muss eine Kopie jeder Punktwolke gemäß der kinematischen Kette transformiert, und in eine Voxel-Datenstruktur eingetragen werden. Wird der Roboter nur achsparallel verschoben, kann auf das rechenintensive Transformieren der einzelnen Punktwolken verzichtet werden und statt dessen auf das Verfahren zur Translation mittels Basisversatz aus Unterabschnitt 5.3.1 zurückgegriffen werden.

4.4.3. Selbstausblendung und Eigenkollisionen

Soll der Arbeitsraum eines Roboters visuell auf Kollisionen hin überwacht werden, so kann nicht vermieden werden, dass neben der Umwelt auch der Roboter von den Sensoren erfasst wird. Um hierbei nicht fälschlicherweise Eigenkollisionen zu detektieren, müssen die Messpunkte, die den Roboter repräsentieren, aus den Aufnahmen entfernt werden. Hierfür wurden zwei unterschiedliche Verfahren genutzt, die eine exakte Kalibrierung zwischen Kamera und Roboter voraussetzen:

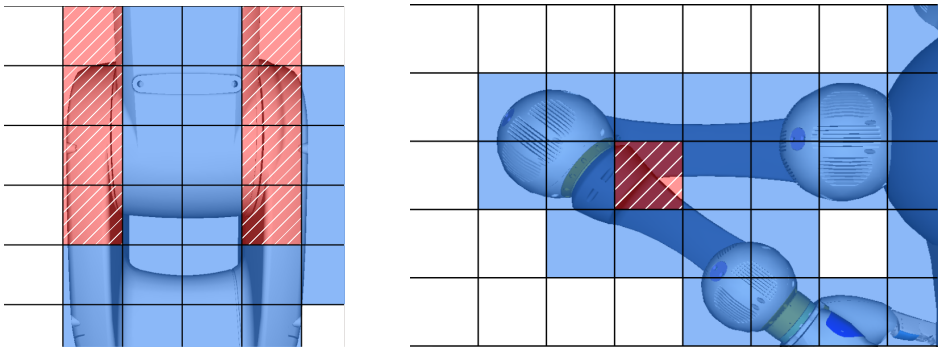
In ROS existiert hierfür das *Realtime URDF Filter*-Paket[3]. Wie der Name vermuten lässt, filtert dieses aus dem Datenstrom einer 3D-Kamera basierend auf einem URDF-Modell des Roboters die problematischen Punkte heraus. Dafür rendert ein OpenGL-Programm live das geometrische Modell des Roboters aus der Perspektive der Tiefenkamera und vergleicht die Tiefendaten der Sensorwerte mit den Z-Puffer-Ergebnissen des Renderings. Messpunkte, die ähnliche Entfernungswerte wie der Z-Puffer aufweisen, werden entfernt. Das Resultat ist der Datenstrom aus Tiefeninformationen, die um den Roboter bereinigt wurden. Der URDF Filter wurde erfolgreich für die Evaluation der Bewegungsprädiktion aus Abschnitt 4.6 genutzt.

Wird der original Datenstrom aus der Kamera nicht benötigt, da es ausreicht, Roboter-Voxel auszufiltern, kann der Aufwand des OpenGL-Renderings vermieden werden. Dazu wurde ein Verfahren umgesetzt, das rein auf GPU-Voxels und dem darin vorhandenen

[3]Realtime URDF Filter: `http://wiki.ros.org/realtime_urdf_filter`

Robotermodell basiert. Als Voraussetzung müssen die Modell- und Sensordaten in unterschiedlichen Datenstrukturen gehalten werden: Die in Voxel umgewandelten Sensordaten liegen in einer Umweltkarte vor, während sich das Robotermodell in einer weiteren Datenstruktur befindet. Somit kann das animierte Robotermodell von den Sensorvoxeln subtrahiert werden. Die verbleibenden Daten können dann herangezogen werden, um sie gegen eine dritte Datenstruktur auf Kollisionen zu prüfen, bspw. um darin Bewegungen zu planen.

Echte Eigenkollisionen werden separat während des Eintragens einer Roboterpose in die Voxel-Datenstruktur behandelt. Da die Elemente der Kinematik hierbei sequentiell abgearbeitet werden, kann vor dem Eintragen eines Modellpunktes geprüft werden, ob der Zielvoxel bereits belegt ist. In diesem Fall liegt eine Eigenkollision vor. Aufgrund von Diskretisierungsfehlern tritt dieser Fall jedoch an fast jedem Robotergelenk auf, da die Geometrien hier sehr eng aneinander liegen (siehe Abb. 4.10). Somit ist es notwendig, Paare von Entitäten im Voraus durch Expertenwissen von einer Kollisionsprüfung auszuschließen, wenn diese rein kinematisch nicht kollidieren können. Eine Identifizierbarkeit einzelner Entitäten kann über Bitvektor-Voxel gewährleistet werden, die in Unterabschnitt 5.1.4 vorgestellt werden.



(a) Gabelung im Fuß eines Industrie-Roboters-Gelenks

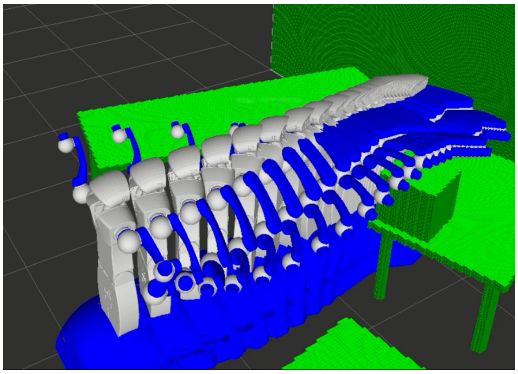
(b) Eng aneinander liegende Geometrien

Abb. 4.10.: Beispiele für potentielle Falschdetektion von Eigenkollisionen, die durch manuell modellierte Kollisionspaare auszuschließen sind.

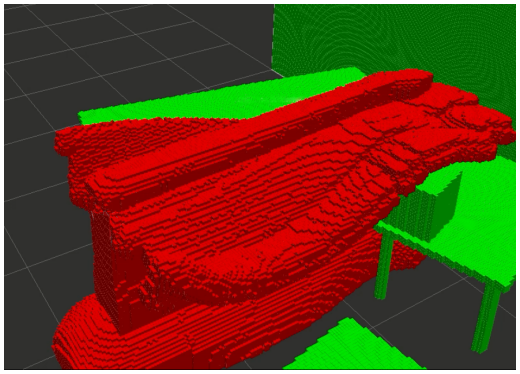
4.5. Swept-Volumen

Da sowohl bei der Bewegungsplanung, als auch bei der Überwachung von Bewegungen, meist nicht nur einzelne, unabhängige Posen auf Kollisionen geprüft werden müssen, sondern Abfolgen von zusammenhängenden Bewegungen, ist es von Vorteil, diese effizient repräsentieren und evaluieren zu können. Ein Beispiel einer solchen Repräsentation ist in Abb. 4.11 zu sehen.

In der Praxis wird zur Berechnung die Bewegung des Objektes an diskreten Zeitpunkten t_i abgetastet, woraus sich ein Approximationsfehler ergibt. Wird die zeitliche Auflösung Δ_t zu grob gewählt, können im entstehenden Swept-Volumen Lücken wie in Abb. 4.12



(a) Grob abgetastete Bewegung des Oberflächenmodells



(b) Fein abgetastetes Voxel Swept-Volumen

Abb. 4.11.: Swept-Volumen einer Ganzkörperbewegung des Roboters HoLLiE

Definition 11. Ein **Swept-Volumen** beschreibt das aufintegrierte Volumen V_{Swept} im Raum, das von einem Objekt mit Volumen V_O durch seine Bewegung s im Zeitraum $t_1 - t_0$ überstrichen wird.
$V_{\text{Swept}} = \int_{t_0}^{t_1} V_O \cdot s(t)\,\mathrm{d}t.$

entstehen, die bei einer Kollisionsprüfung zu einer Nichtdetektion von Kollisionen führen können. Wird hingegen die Abtastfrequenz zu hoch angesetzt, führt dies zu einem hohen Ressourcenverbrauch bei der Erstellung des Volumens.

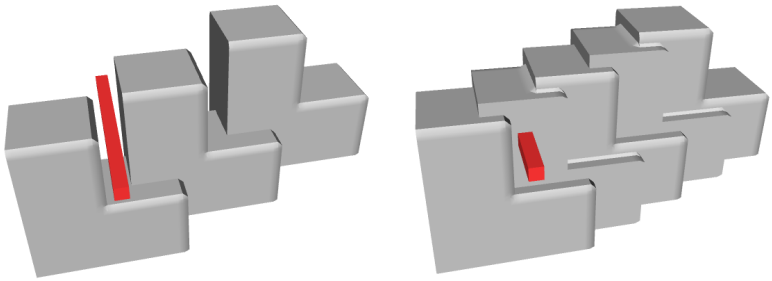


Abb. 4.12.: Unterabtastung (links) und ausreichende Abtastung (rechts) bei der Generierung eines Swept-Volumen zur Kollisionsprüfung mit rotem Objekt. Aus [38].

Bei einer Oberflächennetzbasierten Darstellung des Sweeps werden entweder Instanzen des Objektes an den Abtastzeitpunkten erstellt und einzeln verarbeitet, oder die Momentaufnahmen werden mittels Boolscher Operationen miteinander verschmolzen und als ein einzelnes Modell gesehen. Im ersten Fall bleiben bei einer Kollisionsprüfung die Bestandteile und somit die Zeitschritte identifizierbar, jedoch steigt der Speicher- und Rechenaufwand linear mit der Anzahl der Abtastschritte. Im zweiten Fall müssen im Inneren liegende Oberflächen aufwendig entfernt werden [33]. Algorithmen für die Verschmelzung basieren meist auf *Marching Intersections* und weisen bei n Dreiecken eine Laufzeit von $\mathcal{O}(n^3)$ auf [119, 195, 85]. In beiden Fällen wird dennoch lediglich die Oberfläche und nicht das Volumen des Sweeps betrachtet.

Bei einer voxelbasierten Repräsentation entfallen diese Probleme, da hier das durch die

Voxel diskretisierte Volumen erzeugt wird. Überlappen sich die Momentaufnahmen, stellt das mehrfache kennzeichnen belegter Voxel keinen besonderen Aufwand dar und benötigt auch keinen zusätzlichen Speicher. Durch eine besondere Repräsentation der Voxel (siehe Bitvektor-Voxel in Unterabschnitt 5.1.4), die die Swept-Volumen-Implementierung in dieser Arbeit nutzt, ist es möglich, Segmente innerhalb des Gesamtvolumens identifizierbar zu halten, und somit eine Zuordnung zum Abtastzeitpunkt zu speichern. Somit lässt sich die Generierung des Volumens als Abfolge von Operationen zur Voxelumwandlung $\boxplus(M, P(t_i))$ beschreiben, wobei die Punktwolke $P(t_i)$ animiert ist und die Aktualisierungsfunktion $\boxdot_{\ddagger_i}(V)$ einen Operator $\ddagger_i$ aufweist, der vom Abtastzeitpunk t_i abhängt. Da der Speicher pro Bitvektor-Voxel stark beschränkt ist, können in der aktuellen Implementierung lediglich $i \leq 250$ Zeitintervalle unterschieden werden. Somit bestimmt sich das minimale, identifizierbare Zeitintervall Δ_{t_B} direkt aus dem abzubildenden Zeitraum: $\Delta_{t_B} = T_{\text{gesamt}}/250$. Dieses Intervall ist unabhängig von der Abtastdichte Δ_t, was bedeutet, dass bis zu b Abtastschritte dieselbe ID aufweisen können: $b = \Delta_{t_B}/\Delta_t$.

Der zeitliche Verlauf einer Armbewegung ist in Abb. 4.13 durch die Farben von Grün ($i = 0$) bis Magenta dargestellt. Da die Gelenkwinkeländerung des Armes in der linken Bildhälfte größer ist, als auf der rechten Seite, und somit eine längere Bewegungsdauer aufweist, werden hier intensivere Magenta-Töne erreicht.

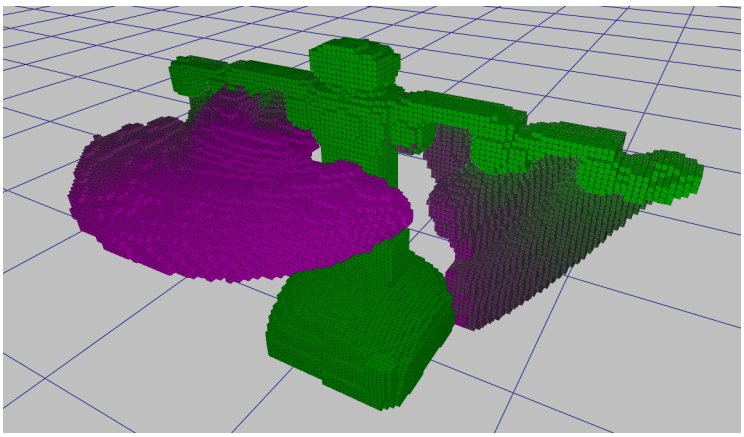


Abb. 4.13.: Swept-Volumen einer Roboterbewegung. Aufsteigende Sub-Swept-Volumen-Identifikatoren (SSV-IDs)) sind durch den Farbverlauf von Grün nach Magenta visualisiert.

Eine alternative Abtaststrategie wäre es, den Moment der Abtastung über die zurückgelegte Distanz Δ_s und nicht über feste Zeitintervalle Δ_t zu definieren. Somit könnten Lücken im Swept-Volumen optimal vermieden werden. Da Δ_s jedoch sowohl Gelenkwinkel als auch kartesische Distanzen ausdrücken müsste, wäre die Zuordnung des Abtastzeitpunktes zu Voxeleigenschaften unverhältnismäßig komplex. Daher wurde dieser Ansatz hier nicht verfolgt.

4.6. Bewegungsprädiktion

Um einem Roboter ein vorausschauendes Verhalten (siehe Definition 4) zu ermöglichen, muss dieser die Bewegung in einer Szene zunächst erkennen und korrekt interpretie-

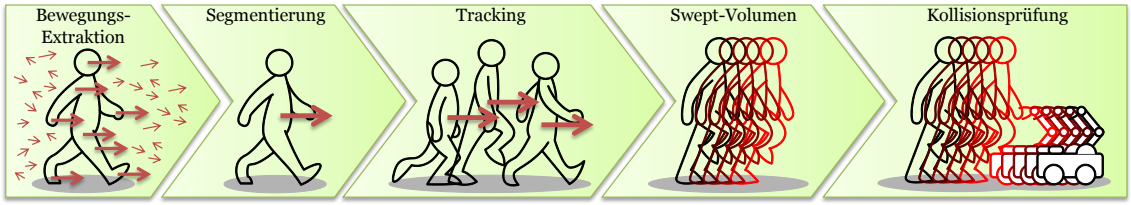


Abb. 4.14.: Schritte der Bewegungssegmentierung und Prädiktion[4].

ren, um dann Vorhersagen treffen zu können. Die Prädiktion kann dann in Voxelmodelle umgewandelt werden, um sie in der Kollisionsdetektion und der Bewegungsplanung zu berücksichtigen (vgl. Abb. 4.14). Der Fokus dieser Arbeit liegt auf der weiterführenden Nutzung der Bewegungsschätzungen zur Kollisionsprädiktion, weswegen die Prädiktionskette nicht erschöpfend untersucht und implementiert wurde.

In diesem Abschnitt wird eine vierstufige Verarbeitungskette beschrieben, die die genannten Aufgaben löst und die in der Masterarbeit von Felix Mauch [27] evaluiert wurde:

1. Erkennung und Segmentierung aller Objekte (gegenüber ihrer Umgebung), deren Bewegung geschätzt werden soll. Bei der Segmentierung von RGBD-Daten kann hierfür neben den Farbinformationen die Tiefenkomponente verwendet werden, um ein Objekt gegenüber einem Hintergrund freizustellen.
2. Verfolgung der Objekte über einen gewissen Zeitraum, um ihren Bewegungsverlauf zu bestimmen.
3. Schätzung von Bewegungshypothesen für einen bestimmten Zeithorizont mit Hilfe eines a priori Bewegungsmodells.
4. Voxel-Rendering der Bewegung aus dem segmentierten Objekt entlang seiner Bewegungshypothese.

Die ersten drei Schritte werden unter dem Begriff *Tracking* zusammengefasst. Klassische Verfahren, die heute bereits beispielsweise im Automotive-Umfeld [82] zum Tracking von Verkehrsteilnehmern genutzt werden, sind auf die Vorhersage einzelner starrer Objekte ausgelegt. Hierbei genügt es, den Mittelpunkt oder den Schwerpunkt eines zusammenhängenden Bereiches in den Sensordaten zu verfolgen und dessen Bahn vorherzusagen. In anderen Fällen werden geometrische Modelle für die Segmentierung genutzt [39, 59], um die Ergebnisse der Detektion robuster und die Vorhersagen genauer zu machen. Da hierfür Objektmodelle nötig sind, können die Verfahren nicht auf unbekannte Objekte angewendet werden. Für diese Anwendung existieren modellfreie Ansätze, die mit wenigen, dafür aber markanteren Merkmalspunkten pro Objekt arbeiten [197], wobei die Herausforderung dann in der Bestimmung und der Extraktion geeigneter Merkmale besteht.

Solche modellfreien Verfahren bilden auch die Grundlage zur Verfolgung artikulierter oder deformierbarer Objekte, wie beispielsweise den menschlichen Körper. Hierbei erfolgt das Tracking, bzw. die Vorhersage per Pixel oder im 3D-Fall per Voxel und die Segmentierung bildet erst den zweiten Schritt der Verarbeitungskette.

[4] Illustration der Laufbewegung von Charles Leon

Ein verbreiteter Ansatz dieser Kategorie zur Analyse von Bewegungen in Videoaufnahmen ist der so genannte *Bildfluss* oder *optische Fluss*, der einzelne Pixel über eine Bildsequenz hinweg verfolgt, und ihnen somit einen zweidimensionalen Bewegungsvektor zuordnen kann [86]. Wird das Verfahren um die dritte Dimension erweitert, spricht man von *Szenenfluss*. Dieser wurde erstmals in [200] eingeführt und in mehreren Arbeiten auf unterschiedliche Weisen berechnet:

- Statistische Optimierungsverfahren: *RGBD-Flow* [99, 202], *Dense Semi-Rigid Scene Flow* [166]
- Partikelfilter: *Scene Particles* [93]
- Bewertungsfunktionen mit Hin- und Rücktransformation: *Sphere Flow* [101]
- Expectation Maximization Algorithmen: *Dense Ridgid-Body Motion Segmentation and Estimation* [192]

Eine Visualisierung von Ergebnissen der ersten drei Kategorien ist in Abb. 4.15 zu sehen. Aus der vierten Kategorie war zum Zeitpunkt der Untersuchung noch keine Implementierung verfügbar.

(a) RGB-Bild t_0 (b) RGB-Bild $t_1 = t_0 + 1$ (c) Farblegende des Bewegungsvektors

(d) SphereFlow (e) RGBD-Flow (f) DSR Scene Flow (g) Scene Particles

Abb. 4.15.: Visualisierung der Ergebnisse der untersuchten Szenenfluss-Algorithmen für das Bildpaar a) und b). *Sphere Flow* mit aussagekräftigstem Szenenfluss, inhomogene aber korrekte Ergebnisse aus *RGBD-Flow* und *Dense Semi-Rigid Scene Flow*. Fehldetektion im Hintergrund aus *Scene Particles*. Gegenüberstellung entnommen aus [101].

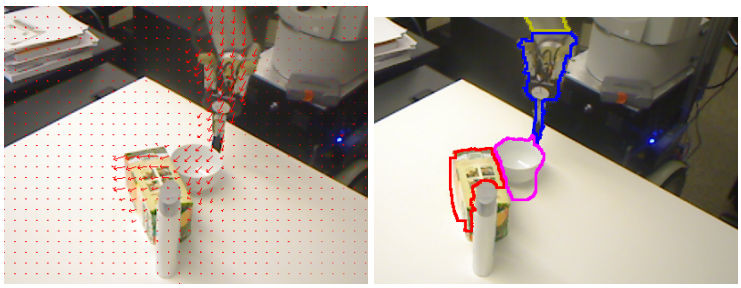
Um den am besten für die Integration in GPU-Voxels geeigneten Algorithmus zu finden, wurden in der Masterarbeit von Mauch [27] alle genannten Verfahren untersucht und diejenigen praktisch evaluiert, von denen eine Implementierung zur Verfügung stand. Das Fazit der Arbeit stellt sich wie folgt dar: *Sphere Flow* musste wegen seiner hohen Laufzeit von der Verwendung ausgeschlossen werden, da die gesamte Verarbeitungskette mehrere Bilder pro Sekunde verarbeiten sollte. Das *Scene-Particles*-Verfahren litt unter hohen Fehlerraten, da es nicht vorhandene Bewegungen in Bildregionen detektierte, welche kurz zuvor durch das eigentlich bewegte Objekt noch verdeckt waren. Bei den

verbleibenden, zueinander ähnlichen Verfahren *RGBD-Flow* und *Dense Semi-Rigid Scene Flow* unterschieden sich die quantitativen und qualitativen Ergebnisse nur wenig. Daher wurde letztendlich der *RGBD-Flow*-Ansatz weiter verfolgt, da von diesem bereits eine GPU-optimierten Implementierung vorlag. Im Folgenden soll dessen implementierte, optimierte Version beschrieben werden.

4.6.1. Vorverarbeitung

In einem ersten Filterschritt gilt es, die Messpunkte, die den Roboter darstellen, aus den Aufnahmen zu entfernen. Da neben den Tiefendaten auch die Farbwerte relevant sind, wird dafür der *Realtime URDF Filter* aus Unterabschnitt 4.4.3 eingesetzt. Der zweite Filterschritt besteht dann aus einer zeitlichen Glättung über mehrere Bilder, um das Grundrauschen des verwendeten Kinect Sensors zu kompensieren (vgl. Unterabschnitt 4.1.4). Für die betrachteten Anwendungsfälle wurde dafür empirisch eine geeignete Fenstergröße von fünf aufeinander folgenden Aufnahmen ermittelt. Die Filterung ist unproblematisch, da die Framerate der Kamera weitaus höher ist, als die der folgenden Verarbeitungskette. Auch Unschärfeeffekte sind nicht zu erwarten, da die Kamera statisch positioniert ist, und die betrachteten Objektbewegungen von geringer Geschwindigkeit sind. In einem letzten Vorverarbeitungsschritt kann die Kameraauflösung von 640×480 auf 160×120 beschränkt werden, da alle Objekte eine gewisse Minimalgröße aufweisen und so auch bei niedriger Auflösung gut erkennbar sind. Diese Datenreduktion ist notwendig, um die Laufzeiten aller nachfolgenden Berechnungen echtzeitfähig zu halten.

4.6.2. RGBD-Flow



(a) Vektorfeld des 3D-Szenenflusses (b) Objektsegmentierung

Abb. 4.16.: Mittels RGBD-Flow berechneter Szenenfluss der durch den Roboterarm verursachten Objektbewegungen. Die unterschiedlichen Bewegungsrichtungen werden zur Objektsegmentierung genutzt. Verfahren und Grafiken aus [99].

Das RGBD-Flow-Verfahren aus [99] basiert auf der Optimierung der Energiefunktion aus Gleichung 4.4, die über alle Pixel $\vec{x} = (x, y)$ des Bildes I gebildet wird:

$$E\left(\vec{u}(\vec{x})\right) = \int_{\vec{x} \in I} \left(E_C\left(\vec{u}(\vec{x})\right) + E_Z\left(\vec{u}(\vec{x})\right) + \alpha E_S\left(\vec{u}(\vec{x})\right) + \gamma E_B\left(\vec{u}(\vec{x})\right)\right) d\vec{x} \tag{4.4}$$

$\vec{u}(\vec{x}) \in \mathbb{R}^3$ beschreibt hier den 3D-Szenenfluss im Pixel $\vec{x}$.

Die beiden ersten Komponenten, aus denen sich die Funktion zusammensetzt, maximieren die *Konsistenz im Farbbild* $E_C\left(\vec{u}(\vec{x})\right)$ und die *Konsistenz im Tiefenbild* $E_Z\left(\vec{u}(\vec{x})\right)$, da davon ausgegangen wird, dass ein Pixel seine Farbe während einer Bewegung nicht verändert bzw. die Position im Raum sich nur entsprechend der Schätzung bewegt. $E_S\left(\vec{u}(\vec{x})\right)$ dient der *Regularisierung*, indem hohe Gradienten im Fluss-Vektorfeld unterdrückt werden, um somit Mehrdeutigkeiten im Farbbild-Fluss zu vermeiden. $E_B\left(\vec{u}(\vec{x})\right)$ ist ein *Balancierungsterm*, der entsprechend der Entfernung des Messpunktes mit Hilfe der Kamerabrennweite die Einheiten Meter und Pixel aneinander angleicht. Somit können $E_C\left(\vec{u}(\vec{x})\right)$ und $E_Z\left(\vec{u}(\vec{x})\right)$ verrechnet werden.

Um Lösungen für dieses nichtlineare Gleichungssystem zu finden, wird das Fixpunktverfahren eingesetzt. Dabei wird das System schrittweise linear approximiert und dann das Gleichungssystem jedes Schrittes mittels *Successive Over-Relaxation-Verfahren* (SOR) optimiert. Weiterhin läuft das Lösungsverfahren auf einer Bildpyramide mit ansteigender Bildauflösung ab, so dass die Lösung der Fixpunktiteration einer niedrigen Auflösung als Startwert der Berechnung für die nächsthöhere Auflösung dienen kann.

Die Parameter und Abbruchkriterien dieses Lösungsverfahrens wurden in der Abschlussarbeit von Mauch [27] so weit optimiert, dass die Laufzeit des Algorithmus eine schritthaltende Verarbeitung der Kameradaten mit ca. 10 Frames per Second (FPS) ermöglicht.

4.6.3. Segmentierung bewegter Objekte

Abb. 4.17.: Körpersegmentierung anhand der Bewegungen.

Zur einfacheren weiteren Verarbeitung wird das Ergebnis-Vektorfeld des RGBD-Flow-Verfahrens in eine konsistent metrische Darstellung umgewandelt. Ähnlich zum Balancierungsterm E_B aus Gleichung 4.4 müssen hierfür Bewegungsangaben in Pixeln mit Hilfe des Kameramodells in Meter umgerechnet werden. Im Ergebnis liegt zu jedem 3D-Punkt der Punktwolke ein Farbwert und ein 3D-Bewegungsvektor vor. Dieser muss lediglich durch die Differenz der Aufnahmezeitpunkte dividiert werden, um seine Bewegungsgeschwindigkeit zu erhalten. Über eine Filterung anhand einer Minimalgeschwindigkeit werden nun alle Punkte entfernt, die sich nicht, oder nur sehr langsam bewegen.

Um letztendlich die Objekte in der verbleibenden Punktwolke zu segmentieren, wird im Gegensatz zum Originalverfahren aus [99] kein Random sample consensus (RANSAC)-Ansatz angewandt, sondern ein performanteres Region-Growing. Dessen Ähnlichkeitsmetrik berücksichtigt neben den Bewegungsgeschwindigkeiten auch räumliche Distanzen, um ausgehend von einem Startpunkt benachbarte Punkte zu clustern.

Jedes gefundene Cluster repräsentiert somit ein Objekt, dessen Bewegung aus den Eigenschaften aller seiner 3D-Punkte zu ermitteln ist. Um ohne weitere Vorkenntnisse oder Annahmen eine Objektposition festlegen zu können, wird diese als Schwerpunkt aller Messungen definiert. Diese Definition ist jedoch stark von der Betrachtungsperspektive und Verdeckungen abhängig. Daraus entstehende Sprünge oder Schwankungen müssen in der weiteren Verarbeitung berücksichtigt werden.

Für die Bestimmung einer Objektbewegung wird die Bewegungsrichtung als Mittelwert aller Punkte definiert. Für die Bewegungsgeschwindigkeit ist eine Mittelung jedoch nicht ohne weiteres möglich, da verfahrensbedingt die Geschwindigkeiten nur am Rand eines Objektes korrekt berechnet werden und zum Inneren des Clusters abfallen (in Gleichung 4.4 präferiert E_S niedrige Gradienten im Szenenfluss, während fehlende Kontraste innerhalb eines Objektes E_C und E_Z ansteigen lassen, weshalb kleinere Bewegungen bevorzugt werden). Daher wird die Geschwindigkeit bei weiteren Berechnungen nicht direkt verwendet, sondern wie im nächsten Abschnitt beschrieben, über ein Erweitertes Kalman Filter (EKF) geschätzt.

4.6.4. Tracking

Zur Verfolgung der Detektionen über die Zeit wird pro Objekt ein EKF instantiiert. Dabei entscheidet ein Ähnlichkeitsmaß (aus Position und Richtung) darüber, ob für eine Detektion bereits ein Filter existiert, welcher aktualisiert werden muss, oder ob ein neuer Filter anzulegen ist. EKF Instanzen, die zu lange keine Aktualisierung erfahren, werden gelöscht.

In der Implementierung für diese Arbeit wurde ausschließlich ein lineares Bewegungsmodell angenommen. Das Zustandsraummodell der Filter weist dabei folgende sieben Dimensionen auf: Position und Richtung der Bewegung im 3D-Raum und eine skalare Geschwindigkeit. Diese getrennte Betrachtung von Geschwindigkeit und Bewegungsrichtung erlaubt die Berücksichtigung der unterschiedlichen Verlässlichkeit der Messungen.

Der Kalman-Prädiktionsschritt, der mit jeder Aktualisierung des Filters ausgeführt wird, lässt sich direkt aus dem einfachen Bewegungsmodell ableiten. Eine Besonderheit liegt lediglich in der Nichtbetrachtung des Messwertes der Geschwindigkeit, da dieser wenig zuverlässig ist. Zustandsraum, Prozess- und Messmodell werden sonst als fehlerlos modelliert. Für das Mess- und Modellrauschen wird die Unabhängigkeit der einzelnen Dimensionen von Position und Richtung angenommen, ihre Genauigkeit bzw. Unsicherheit soll in allen drei Dimensionen identisch sein.

(a) RGB Anteil der Eingabedaten (b) Segmentierungsergebnis (c) Swept-Volumen der Prädiktion

Abb. 4.18.: Beispiel des Trackings und der Prädiktion von zwei Personen.

4.6.5. Prädiktion in Form eines Swept-Volumens

Die Prädiktion jedes Filters liefert letztendlich eine geschätzte Bewegungsrichtung und Geschwindigkeit für jede Objektpunktewolke. Ausgehend von einer bekannten Kameraperspektive kann daraus jedes Cluster entlang seines Bewegungsvektors schrittweise transformiert werden. Die Schrittlänge ergibt sich aus der geschätzten Geschwindigkeit und einem festen Zeitraster. Wird die transformierte Punktwolke in jedem Schritt in eine Voxelliste (vgl. Abschnitt 5.4) eingetragen, entsteht somit ein Swept-Volumen der anzunehmenden Bewegung. Listen werden hier einer Voxelkarte vorgezogen, da eine Karte nur sehr dünn besetzt wäre. Weiterhin wird das Swept-Volumen in jeder Iteration gelöscht und neu aufgebaut, wofür der geringe Speicherbedarf der Liste von Vorteil ist.

Die Liste wiederum besteht aus Bitvektor-Voxeln (siehe Unterabschnitt 5.1.4), um jeden Zeitschritt mit einer inkrementierten SSV-ID kennzeichnen zu können. Somit ist bei einer Kollisionsprüfung nachvollziehbar, zu welchem Zeitpunkt diese auftreten würde. Um Bitvektor-Voxel zur Repräsentation von zeitabhängiger Belegtheit im Raum nutzen zu können, wurde eine Codierung gewählt, in der jedes Bit einer konstanten Zeitdauer entspricht. Somit können Zeitpunkte (relativ zu einer bekannten Startzeit) direkt auf SSV-IDs abgebildet werden, um so die räumliche Ausdehnung einer Bewegung zu diskreten Zeitschritten darzustellen.

4.6.6. Unscharfe Kollisionsprüfung

Um den geschätzten Bewegungsschlauch auf Kollisionen mit dem Swept-Volumen der Robotertrajektorie zu prüfen, muss neben der örtlichen auch die zeitliche Überschneidung berücksichtigt werden. Um hierbei leichte Fehler in der prädizierten Geschwindigkeit ausgleichen zu können, kommt ein besonderes Verfahren zum Einsatz, das nicht eine exakte zeitliche Übereinstimmung prüft, sondern ein Zeitfenster aus mehreren SSV-IDs. Zusätzlich muss die Zeitdifferenz zwischen Roboter- und Hindernisrepräsentation kontinuierlich auf alle Voxel des Roboter-Swept-Volumens addiert werden, bevor eine korrekte Kollisionsprüfung möglich ist. Die Addition einer Zeitspanne t entspricht dem bitweisen Verschieben der SSV-IDs um $t/\Delta t_{Sv}$ Bits. Für den nötigen $\ll$-Operator ergeben sich durch die Größe des Bitvektors dieselben Herausforderungen wie bei der zeitlich gefensterten Kollisionsprüfung (siehe Abschnitt 6.2.1), weshalb die Implementierung denselben Code nutzt.

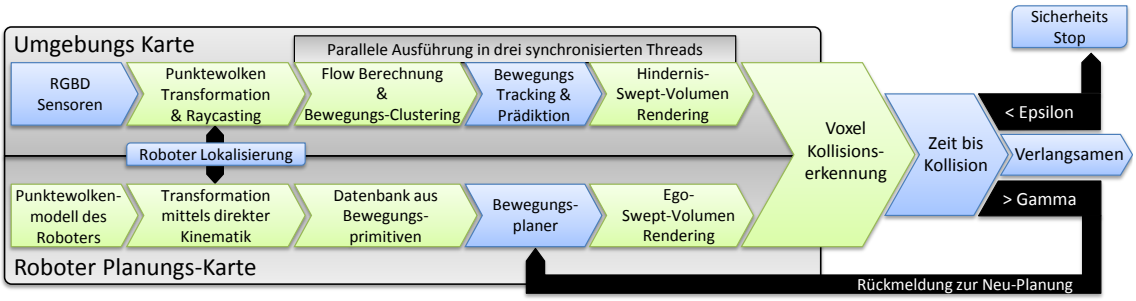


Abb. 4.19.: Der Programmablauf lässt sich in zwei Stränge einteilen: Die Verarbeitung der Umweltinformationen und die Planung mit dem Robotermodell. Blau gefärbte Abschnitte werden auf dem Host ausgeführt, grüne auf der GP-GPU. Nach der Detektion einer Kollision ergeben sich je nach verbleibender Zeit drei Möglichkeiten: Not-Stop, Neuplanung, Verlangsamung.

4.6.7. Implementierung

Das Gesamtprogramm ist auf Host-Seite in vier asynchron laufende Threads aufgeteilt. Während ein Thread die Kameradaten vorverarbeitet und glättet, arbeitet die Szenenflussberechnung parallel dazu auf dem letzten Kameraframe. Hierbei wird zunächst der CUDA-Code zur Flussberechnung aufgerufen, das berechnete Vektorfeld segmentiert und die ermittelten Objekthypothesen getrackt. Auf diese Ergebnisse wartet ein dritter Thread, der das Rendern der Swept-Volu-men auf der GPU übernimmt. Ein Haupt-Thread verwaltet die beschrieben Arbeiten, löst die Kollisionsprüfung aus, und entscheidet je nach Ergebnis über die Reaktion. Optional kann zusätzlich ein fünfter Thread der GPU-Voxels-Visualisierung (siehe Abschnitt 5.7) die Prädiktionen und Sensorwerte für den Nutzer darstellen.

4.6.8. Kamerabewegung

Die bisher aufgezeigte Verarbeitungskette setzt eine statische Kamerapose voraus. Ist jedoch, wie bei einem mobilen System üblich, die Kamera auf dem Roboter montiert, so muss die Eigenbewegung in der Berechnung berücksichtigt werden. Würde man ausschließlich das Kamerabild auswerten, wäre bei der Detektion einer Bewegung unklar, ob diese durch ein dynamisches Objekt oder durch die Änderung der Kameraperspektive verursacht wurde.

Um die Eigenbewegung zu neutralisieren, muss nach der Berechnung des Szeneflusses von den Bewegungsvektoren jedes Clusters ein Kompensationsvektor subtrahiert werden. Rotiert der Roboter bzw. die Kamera, muss dabei pro Cluster (anhand dessen Lage bezüglich des Roboterkoordinatensystems) bestimmt werden, wie sich eine Rotation auf die Abbildung der Cluster auswirkt (siehe Abb. 4.20).

Das Ergebnis der Kompensation ist direkt abhängig von der Qualität der Daten zur Eigenbewegung und deren zeitlichen Synchronisierung gegenüber den Sensordaten. Ist kein externes Trackingsystem zur Überwachung der Kamerapose vorhanden, muss sich

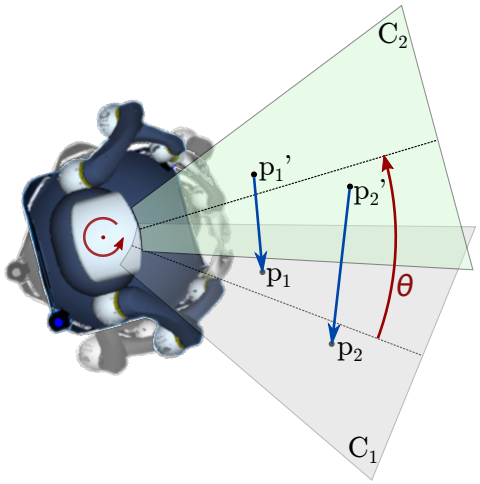


Abb. 4.20.: Eine Rotation des Roboters um θ wirkt sich je nach Distanz unterschiedlich auf wahrgenommene Objekte aus.

das Verfahren auf die interne Sensorik des Roboters stützen. Da insbesondere bei rotatorischen Bewegungen jedoch bereits kleinste Abweichungen zu einer Über- oder Unterkompensation führen, wurde für die Evaluierung kein bewegter Sensor verwendet. In den Versuchen kam statt dessen ein statischer RGBD-Sensor zum Einsatz. Als Lösung für weiterführende Arbeiten wird der Einsatz von visueller Odometrie vorgeschlagen, da hierbei derselbe Sensor genutzt wird und somit keine Synchronisation zur Eigenbewegungsmessung benötigt wird.

4.6.9. Zusammenfassung

In diesem Abschnitt wurden Verfahren vorgestellt, die aus einer zeitlichen Abfolge von RGBD-Daten zusammenhängende Volumen bestimmen können, die sich gemeinsam bewegen. Durch eine Schätzung ihrer Bewegungsvektoren können kurzzeitige Vorhersagen erstellt und in die Zukunft projiziert werden, um daraus den zukünftig belegten Raum in Form von Swept-Volumen zu generieren. Mit Hilfe dieser prädizierten Bewegungsschläuche ist eine Kollisionsdetektion in der Lage, nicht nur auf Momentaufnahmen der Umwelt, sondern auf der zu erwarteten Situation wahrscheinlich auftretende Kollisionen zu bestimmen. Die Annahme eines linearen Bewegungsmodells mit konstanten Geschwindigkeiten schränkt die Genauigkeit der Prädiktionen stark ein. Dies kann jedoch in weiterführenden Arbeiten durch die Verwendung komplexerer Modelle verbessert werden.

Anwendungen, die die Prädiktion zur Vermeidung von Zusammenstößen nutzen, werden in Abschnitt 8.8 ausführlich evaluiert.

4.7. Simulierte Umgebung

Viele Szenarien und Algorithmen können anhand von aufgezeichneten Tiefendaten getestet werden. Sollen jedoch reaktive Verhalten oder dynamische Planungsalgorithmen untersucht werden, ist eine Simulation von sich mit dem Roboter bewegenden 3D-Kameras unabdingbar. Nur so kann ein dynamischer Informationshorizont geschaffen werden. Daher wurde unabhängig von GPU-Voxels ein rudimentärer Simulator implementiert, der für mehrere virtuelle Sensoren Punktwolken innerhalb deren begrenzten Sichtfeldes und Reichweite generiert. Auf Basis von zwei Höhenkarten (Graustufenbildern) erlaubt es der Simulator, 2,5D Hindernisse von Boden und Decke ausgehend zu generieren und so auch Überhänge zu simulieren. Eingesetzt wird dafür ein Raycasting, das Schichtweise die Höhenprofile abtastet und an Hindernissen Messpunkte generiert. Diese Erzeugung von Punktwolken stellt zwar eine sehr gut parallelisierbare Aufgabe dar, dennoch wurde die Software explizit nur für die CPU umgesetzt, um nicht in Konkurenz zu GPU-Voxels auf der GPU ausgeführt zu werden. Eine Parallelisierung mit Hilfe von OpenMP erlaubt aber auch auf der CPU die gleichzeitige Simulation von bis zu vier Kinect-Aufnahmen mit realistischen Wiederholraten von 20 Hz.

Die Erzeugung künstlicher Punktwolken kam zur ersten Evaluation von allen in Kapitel 8 beschriebenen Szenarien zum Einsatz. Beispiele simulierter Umgebungen finden sich in Abschnitt 5.7, Abb. 5.23 oder Abschnitt 8.9, Abb. 8.35.

4.8. Fazit

Das Kapitel schafft die Voraussetzung für den Umgang mit Messdaten der Umwelt. Dafür wurden zunächst passende Datenquellen untersucht und ein adäquates Sensormodell begründet. Im Anschluss wurden mögliche Modellierungen der Umwelt verglichen, um die Vorteile einer Voxelrepräsentation herauszustellen und Forschungsfrage 2 zu beantworten. Ebenso konnte ein animiertes Egomodell über Voxel umgesetzt werden, in dessen Rahmen auch der Begriff der Swept-Volumen definiert wurde.

Im Verlauf des Kapitels konnten mehrere Verarbeitungsschritte identifiziert werden, die sich sehr gut für eine Parallelisierung auf der GPU eignen: Bereits die Vorfilterung und Projektion von Distanzbildern zu Punktwolken ist effizient parallelisierbar. Ebenso die eigentliche Voxelumwandlung – sowohl von Umweltinformationen (inklusive der simultan ablaufende Freiraumbestimmung mittels Raycasting), als auch von animierten Robotermodellen. Darüber hinaus wurde eine Bewegungsprädiktion für dynamische Hindernisse entwickelt, die eine komplexe Verarbeitungskette aus mehreren Algorithmen auf der GPU berechnet. Zusammenfassend konnte somit Forschungsfrage 1 umfangreich und positiv beantwortet werden, was zu Feststellung 8 führt. Dieses Aussage wird auch

Feststellung 8. Alle für die Kollisionsprüfung relevanten Teilgebiete der Sensordatenverarbeitung und Modellierung lassen sich effizient datenparallel bearbeiten.

durch die praktischen Versuche in Kapitel 8 unterstrichen.

5. Voxel-Datenstrukturen auf der GPU

Im Rahmen dieser Dissertation wurde seit Mitte 2012 an einer Softwarebibliothek gearbeitet, welche das Ziel hat, möglichst vielfältige Robotikanwendungen, die in den vorausgehende Kapiteln beleuchtet wurden, durch den Einsatz von GPU-Technologie zu verbessern oder sogar erst zu ermöglichen. Wie in Kapitel 4 dargelegt, ist der Grundgedanke der Bibliothek die Vermeidung von Dreiecksnetzen zur Repräsentation von Umwelt und Roboter, wie es im aktuellen Stand der Forschung üblich ist, da diese nur über Umwege aus Punktwolkendaten erstellbar sind. Statt dessen wurden mehrere diskretisierende Datenstrukturen mit unterschiedlichen Eigenschaften und Vorzügen implementiert, um die 3D-Daten aufzunehmen, welche bei Kollisionstests und zur Bewegungsplanung benötigt werden. Diese Datenstrukturen speichern wiederum unterschiedliche Voxeltypen, die die eigentlichen anfallenden Nutzdaten Ψ beinhalten. Zusätzlich zu den Datenstrukturen wurden Algorithmen entwickelt, die den Operator zur Voxelumwandlung $\boxplus(M, P)$ aus Definition 10 individuell für jede Datenstruktur umsetzen, um diese mit Daten zu befüllen. Die Architektur der Datentypen und Algorithmen wurde auf bestmögliche Parallelisierbarkeit ausgelegt, indem ihre Implementierung den Paradigmen aus Kapitel 3 folgt. Die Parallelisierung in CUDA erfolgt, wenn nicht explizit anders erwähnt, grundsätzlich datenparallel auf Voxel- bzw. Punkte-Ebene.

Im Folgenden sollen zunächst die Voxeltypen, die Datenstrukturen und ihre Eigenschaften und darauf aufbauend die Algorithmen zur Auswertung erläutert werden. Das spätere Kapitel zur Kollisionserkennung stützt sich wiederum auf diese Auswertungen.

5.1. Voxeltypen

Je nach Anwendungszenario gilt es, den 3D-Raum mit unterschiedlich umfangreichen Daten zu annotieren. Daher wurden vier Voxeltypen implementiert, um unterschiedliche Informationen repräsentieren zu können. Mit Ausnahme des einfachsten Typs wurde die Speichergröße der Voxel dabei passend zur Cache-Architektur der verwendeten Hardware gewählt.

Definition 12. Die verfügbaren **Voxeltypen** unterscheiden sich durch die in ihnen speicherbaren Nutzdaten Ψ. Dies können Belegtheitswahrscheinlichkeiten, Distanzen oder Zugehörigkeiten sein. Jeder Voxeltyp weist unterschiedliche Aktualisierungsoperatoren $\boxdot_{\ddagger}(V)$ auf, die seinen Zustand entsprechend neuer Eingabedaten ändern. Weiterhin existieren $\blacksquare$-Operatoren, die die spezifischen Informationen im Zuge einer Kollisionsdetektion interpretieren, um eine Aussage zu treffen, ob ein Voxel belegt ist. Nicht zu verwechseln sind Voxeltypen mit der *Voxel-Bedeutung* der Bitvektor-Voxel (siehe Definition 13).

Diese vier Typen werden nun detailliert beschrieben.

5.1.1. Deterministische Voxel

Diese Art der Voxel weisen den kleinsten Speicherverbrauch auf, da sie nur drei Zustände kennen und daher mit 2 Bit codiert werden können: *frei*, *belegt* und *unbekannt*. Sie sind zu bevorzugen, wenn eindeutige Belegtheitsinformationen abzubilden sind, beispielsweise zur Repräsentation des Roboter-Egomodells. Für die Aktualisierung deterministischer Voxel stehen genau zwei Operatoren zur Verfügung, die einen Voxel V durch eine Messung zum Zeitpunkt t entweder als *belegt* oder *frei* markieren:

$$\boxdot_+(V) := \Psi_t = \quad belegt \quad \forall \Psi_{t-1} \in [unbekannt, frei, belegt] \tag{5.1}$$

$$\boxdot_-(V) := \Psi_t = \quad frei \quad \forall \Psi_{t-1} \in [unbekannt, frei, belegt] \tag{5.2}$$

Ein einmal aktualisierter Voxel kann folglich seinen Zustand nicht mehr in *unbekannt* ändern, sondern nur noch seine Belegtheit wechseln.

Der Belegtheitsoperator ■ ist offensichtlich definiert als:

$$\blacksquare(V) := \begin{cases} 1 & : \Psi_V = belegt \\ 0 & : \text{sonst} \end{cases} \tag{5.3}$$

5.1.2. Probabilistische Voxel

Um gegenüber Sensorrauschen bzw. kurzzeitigen Ereignissen robuster zu sein, speichern probabilistische Voxel eine Belegtheitswahrscheinlichkeit. Arbeiten aus dem Bereich der robotischen Kartierung verfolgen für die Aktualisierung $\boxdot_\pm(V)$ meist einen Weg, der auf eine langfristige Stabilität des Zustandes Ψ ausgelegt ist. Moravec und Elfes [146] nutzen dafür zwei Zählvariablen $\in \mathbb{Z}$ pro Zelle und ermitteln die Belegtheitswahrscheinlichkeit $p(belegt)$ als Quotienten aus der Anzahl von Messungen, die ein Hindernis in der Zelle anzeigen und der Anzahl, mit der die Zelle als frei gesehen wurde, also sie von einem Messstrahl passiert wurde:

$$p_V(belegt) = \frac{|\boxdot_+(V)|}{|\boxdot_+(V)| + |\boxdot_-(V)|} \quad \text{und entsprechend} \quad p_V(frei) = 1 - p_V(belegt) \tag{5.4}$$

Da hierbei kein Sensormodell berücksichtigt wird, wurde in dieser Arbeit der probabilistische Ansatz verfolgt, den unter anderem Thrun et al. in [196] vorstellen und der mit einer Variablen pro Voxel auskommt. Hierbei ist die Belegung einer Zelle als Schätzung eines binären Zustandes formuliert und durch einen binären Bayes-Filter berechnet. Dieser kann eingesetzt werden, wenn aus einer Folge von Sensormessungen auf eine binäre Zustandsvariable geschlossen werden soll. Als Log-Odd dargestellt (Definition siehe Abschnitt A.1 im Anhang), ergibt sich die folgende einfache Formel zur Aktualisierung der Belegtheitswahrscheinlichkeit Ψ einer Zelle, gegeben einer Messung z_t aus einem inversen Sensormodell:

$$\boxdot_\pm(V, z_t) := \Psi_t = \Psi_{t-1} + log\frac{p(belegt|z_t)}{1 - p(belegt|z_t)} - log\frac{p(belegt)}{1 - p(belegt)} \tag{5.5}$$

Die Verlässlichkeit einer Messung ergibt sich aus der probabilistischen Modellierung des eingesetzten Sensors (siehe Unterabschnitt 4.1.4, Sensormodelle).

Für die betrachteten Szenarien ist es wichtig, Voxel als Hindernis anzusehen, auch wenn sie nur sehr kurzzeitig als belegt gemessen wurden. Dabei genügt es, lediglich einen kurzen Zeithorizont auszuwerten, so dass die Belegtheitswahrscheinlichkeit p pro Voxel speichereffizient in Form eines ganzzahlingen Wertes aus $\mathbb{Z} \in [-128, 127]$ in Ψ gespeichert werden kann. Dem Wert -128 kommt die besondere Bedeutung der *Unbekanntheit* zu, mit dem zunächst alle Voxel initialisiert werden. Volumen dieses Wertes wurden also nie von einem Sensor erfasst und identifizieren Verschattungen, oder nicht explorierte Bereiche. Andere negative Werte stellen eine Wahrscheinlichkeit der *Nicht-Belegtheit* dar, positive Werte die Wahrscheinlichkeit der *Belegtheit*. Bei einem Voxel mit $\Psi = 0$ ist die Wahrscheinlichkeit der Belegtheit $p_{belegt}(V) = 0,5$. Da der Log-Odd aus Gleichung 5.5 einen Wertebereich von $\pm\infty$ aufweist, muss dieser zunächst beschnitten, skaliert und dann diskretisiert werden, um ihn auf den Wertebereich von Ψ abzubilden. Das relevante Intervall lässt sich aus der Reaktionszeit Δ_t (ab der die Belegtheitswahrscheinlichkeit sich um 0,5 geändert hat), der Frequenz f, mit der Sensordaten verarbeitet werden und der maximalen Änderung des Log-Odds pro Berechnungsschritt Δ_p aus dem Sensormodell ermitteln.

$$\Psi_{max} = \Delta_p \cdot f \cdot \Delta_t \tag{5.6}$$

So ergibt sich unter der Annahme repräsentativer Daten ein Wertebereich von $\pm 14,31$ ($f = 25\,\text{FPS}$, $\Delta_t = 0{,}5\,\text{s}$, $\Delta_l = log\frac{0,9}{0,1} \simeq 0,954$. Skaliert man dies auf den verfügbaren Wertebereich von ± 127, ergibt sich eine Auflösung von $0,113$ für Ψ, womit sich Wahrscheinlichkeitsinkremente von $0,565$ darstellen lassen. Dies ist für den angestrebten Zeithorizont von wenigen Sekunden und dem angenommenen einfachen Sensormodell mehr als ausreichend.

Der Belegtheitsoperator $\blacksquare$ für probabilistische Voxel wird über einen Grenzwert ε parametriert, der bestimmt, ab welcher Wahrscheinlichkeit ein Voxel als belegt angesehen wird:

$$\blacksquare(V, \varepsilon) := \begin{cases} 1 & : \Psi_V > \varepsilon \\ 0 & : \text{sonst} \end{cases} \tag{5.7}$$

5.1.3. Distanz-Voxel

Distanz-Voxel werden zur Berechnung und Darstellung von Distanzfeldern (vgl. Abschnitt 5.6) eingesetzt. Sie speichern in Ψ die Voxeladresse des ihnen am nächsten liegenden, belegten Voxels. Die Distanz zu diesem Hindernis kann entweder zur Laufzeit aus der eigenen und der Hindernisvoxelposition abgeleitet werden, oder zugunsten der Rechenzeit, zusätzlich im Voxel gespeichert werden. Durch die Speicherung des Hindernisses - und nicht alleine dessen Entfernung - ist es einigen, der in Abschnitt 5.6 vorgestellten Algorithmen möglich, propagierte Distanzen exakt zu berechnen, wohingegen anderenfalls nur eine Abschätzung möglich wäre.

Zwei Datentypen stehen zur Verfügung:

- **32 Bit Voxel:** Speichert lediglich die Position seines nächstgelegenen Hindernisses in Form von drei 10 Bit Integer Werten. Erlaubt sind somit Distanzfelder von maximal $1023 \times 1023 \times 1023$ Voxeln (3×1 Bit sind reserviert für uninitialisierte Voxelkoordinaten).
- **128 Bit Voxel:** Speichert Hinderniskoordinaten in 3×32 Bit. Zusätzlich ist die Distanz zum Hindernis in den verbleibenden 32 Bit gespeichert, so dass diese nicht wiederholt aus den Koordinaten berechnet werden muss.

Wird im Voxel nicht der Distanzwert ($\in \mathbb{R}$) sondern sein Quadrat ($\in \mathbb{Z}$) gespeichert, so kommt die eigentliche Distanzberechnung fast ohne Fließkomma-Wurzeloperationen aus [134]. Da die Algorithmen dann hauptsächlich durch die Speicherbandbreite limitiert sind, wirkt sich die Verwendung des 32 Bit Voxels dennoch wesentlich drastischer zu Gunsten der Laufzeiten aus, als das Zwischenspeichern der Distanzwerte in 128 Bit Voxeln.

Der Aktualisierungsoperator $\boxdot_{\pm}(V)$ entspricht in seiner ersten Stufe dem Operator der deterministischen Voxel. Sinnvoll kann er jedoch nur in Kombination mit $\boxplus(M, P)$ angewendet werden, da nach jedem Einfügen einer neuen Punktwolke die Distanzberechnungen durchzuführen sind.

Der Belegtheitsoperator $\blacksquare$ erlaubt die Berücksichtigung eines Kollisionsradius. Liegt innerhalb dessen ein Voxel, gilt der angefragte Voxel als belegt:

$$\blacksquare(V, r) := \begin{cases} 1 & : \Psi_V < r \\ 0 & : \text{sonst} \end{cases} \tag{5.8}$$

5.1.4. Bitvektor-Voxel

Bitvektor-Voxel erlauben die Codierung diskreter Zustände oder Zugehörigkeiten. Dafür besteht Ψ aus einem Bitvektor, dessen Bits die Bedeutungen aus Abb. 5.1 zugeordnet sind.

Definition 13. Das im Bitvektor gespeicherte Muster beschreibt die **Voxel-Bedeutung** des Bitvektor-Voxels. Über sie können Zeitpunkte (Voxel ist zum Zeitschritt n belegt) oder Zugehörigkeiten zu Entitäten (bspw. Roboter, statisches / dynamisches Hindernis, Bewegungsplan n, ...) repräsentiert werden. Nicht zu verwechseln mit Voxeltypen (siehe Definition 12).

Die Länge des Bitvektors kann zur Übersetzungszeit definiert werden und geht direkt in den Speicherverbrauch pro Voxel ein. In allen Experimenten in dieser Arbeit wurde eine Bitvektorlänge von 32 Byte pro Voxel gewählt, so dass ein Voxel bis zu 256 identifizierbare Zustände gleichzeitig annehmen kann. Sechs der Bits sind allgemeiner Natur, die verbleibenden 250 Bits stehen für SSV-IDs oder identifizierbare Entitäten zur Verfügung.

Um die Bits der Voxel zu setzen, wurden mehrere Aktualisierungsoperationen $\boxdot_{\ddagger}(V, \Phi)$ implementiert, die Ψ über einen binären Operator $\ddagger$ mit dem zusätzlichen Bitvektor Φ verknüpfen. Als Operator $\ddagger$ stehen `AND`, `OR`, `NOT` und `NOR` zur Verfügung:

$$\boxdot_{\ddagger}(V, \Phi) := \Psi_t = \Psi_{t-1} \ddagger \Phi \tag{5.9}$$

Somit kann $\boxdot_{+}(V, \Phi)$ mittels $\boxdot_{\text{OR}}(V, \Phi)$ umgesetzt werden, wobei in Φ das Bit 1 gesetzt sein muss und optional ein oder mehrere Bits ≥ 4.

Auch der Belegtheitsoperator $\blacksquare$ benötigt einen zweiten Bitvektor Φ, der angibt, welcher Zustand relevant ist. Ist eines oder mehrere der Bits aus Φ und die gleichwertigen Bit in Ψ gesetzt, ist der Voxel belegt:

$$\blacksquare(V, \Phi) := \bigvee_{n=4}^{254} \Psi_{V,n} \wedge \Phi_n \tag{5.10}$$

Spielt die Voxelbedeutung keine Rolle, genügt es, das Belegtheitsbit 1 abzuprüfen:

$$\blacksquare(V) := \Psi_{V,1} \tag{5.11}$$

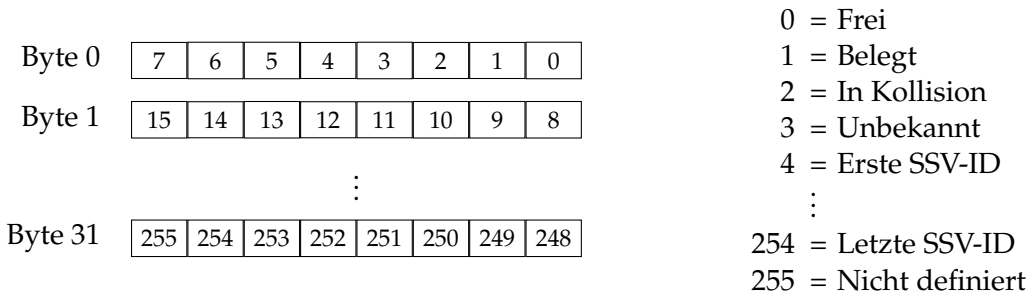


Abb. 5.1.: Aufbau eines Bitvektors in GPU-Voxels: 32 Bytes in einem Array ergeben eine Bit-Maske mit 256 Einträgen, von denen die ersten vier und das letzte Bit eine besondere Bedeutung besitzen.

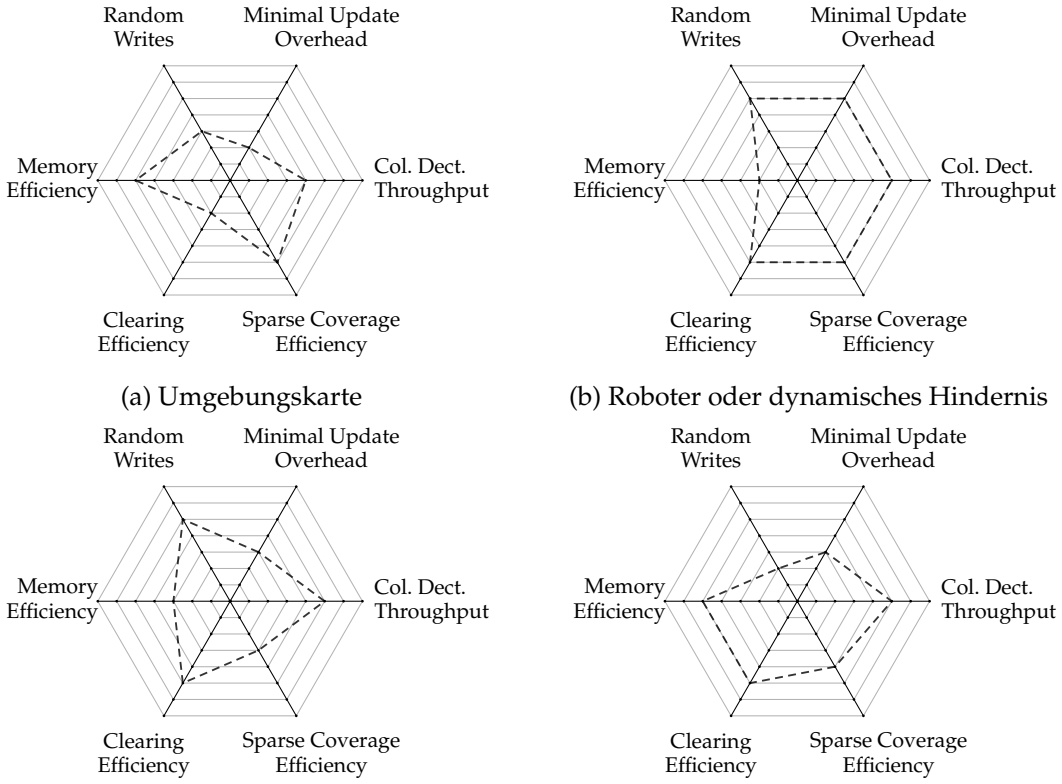
5.2. Anforderungsanalyse Datenstrukturen

In der Informatik existieren eine Vielzahl von potentiellen Datenstrukturen, in welchen die vorgestellten Voxeltypen arrangiert werden könnten. Von diesen, in der imperativen Programmierung typischen Strukturen sind in dieser Arbeit jedoch lediglich drei relevant: Felder fester Größe, Felder variabler Größe und Bäume. Auf die Umsetzung von anderen Datenstrukturen mit variabler Größe (Verkettete Listen, Halden) wurde aufgrund der Einschränkungen der CUDA Speicherverwaltung verzichtet. Da neben der Position im Raum keine andere Ordnung auf Voxeln benötigt wurde, sind auch Hashtabellen oder Vorrang-Warteschlangen nicht von Bedeutung, ebenso Graphen mit bi- oder multidirektionalen Verknüpfungen.

Um eine fundierte Zuordnung zwischen diesen Strukturen und vier typischen Anwendungsfällen zu schaffen, wurden anhand von sechs charakteristischen Eigenschaften Profile der genutzten Datenquellen aufgestellt. Dafür wurden in Abb. 5.2 die folgenden Kriterien qualitativ bewertet:

- Wahlfreie Schreibzugriffe
- Hoher Anteil von unbekannten / freien Voxeln
- Selektives Löschen vorgehaltener Daten
- Spärliche Datendichte
- Hochfrequente Kollisionsprüfung (sequentielles Lesen)
- Häufige abschnittsweise Aktualisierung

Im Fazit des Kapitels werden diese Profile mit den Leistungsmerkmalen der im Folgenden vorgestellten Datenstrukutren verglichen: Voxelkarten (Felder fester Größe), Voxellisten (Felder variabler Größe) und Voxel-Octrees (Baumstrukturen).



(a) Umgebungskarte

(b) Roboter oder dynamisches Hindernis

(c) Swept-Volumen

(d) Bewegungsprimitive

Abb. 5.2.: Anforderungen unterschiedlicher Datenquellen bei der Planung

5.3. Voxelkarten

Eine Voxelkarte stellt eine bijektive Abbildung des diskretisierten dreidimensionalen Raumes $\mathbb{N}_0^3$ auf ein konstantes eindimensionales Feld an Adresse B im Speicher der GPU dar. Unter der Annahme einer statistischen Unabhängigkeit zwischen den Voxeln erlaubt die Datenstruktur das parallele Schreiben und Lesen beliebig vieler Voxel. Eine datenparallele Verarbeitung des Speicherinhaltes ist somit über Grid-Stride-Loops sehr effizient möglich (vgl. Kapitel 3). Weiterhin ist ein wahlfreier Zugriff auf beliebige Voxel in $\mathcal{O}(1)$ möglich. Besitzt der abgebildete Raum die Dimensionen $(dim_x, dim_y, dim_z)^T$ besteht das

Feld aus $n = dim_x \cdot dim_y \cdot dim_z$ Voxeln, deren Speicher einmalig und zusammenhängend zu allozieren ist. Daher ist der Speicherverbrauch konstant, da er direkt über die Dimensionen der Voxelkarte und der Speichergröße pro Voxel Mem_{Voxel} bestimmt ist und nicht vom Grad der Belegtheit abhängt (ein belegter Voxel verbraucht dieselbe Speichermenge, wie ein freier oder unbekannter Voxel). Um die Menge der belegten oder freien Voxel zu bestimmen, muss die gesamte Karte mit linearem Aufwand durchlaufen werden. Sind dafür q Threads verfügbar, liegt der Aufwand für den Schnittoperator $\cap$ und den Vereinigungsopertator $\cup$ bei $\mathcal{O}(n/q)$.

Für die Voxelumwandlung $\boxplus$ von Datenpunkten mit Fließkommakoordinaten $p = (x, y, z)^T,\ x, y, z \in \mathbb{R}$ werden diese in jeder Dimension mit der Voxelseitenlänge l_{Voxel} diskretisiert und zeilen- / ebenenweise nach dem Schema

$$addr(x, y, z) = B + (\lfloor \frac{z}{l_{Voxel}} \rfloor \cdot dim_x \cdot dim_y + \lfloor \frac{y}{l_{Voxel}} \rfloor \cdot dim_x) + \lfloor \frac{x}{l_{Voxel}} \rfloor) \cdot Mem_{Voxel} \tag{5.12}$$

auf die Adresse $addr$ im Speicher abgebildet (vgl. Gleichung 4.2 zur Voxelumwandlung).

Entsprechend kann die geometrische Mitte des Voxels mit der Adresse $addr$ bestimmt werden:

$$\begin{aligned} I_z &= \lfloor \frac{addr - B}{dim_x \cdot dim_y} \rfloor \\ I_y &= \lfloor \frac{addr - B - (dim_x \cdot dim_y \cdot I_z)}{dim_x} \rfloor \\ I_x &= addr - B - (dim_x \cdot dim_y \cdot I_z) - (dim_x \cdot I_y) \end{aligned} \tag{5.13}$$

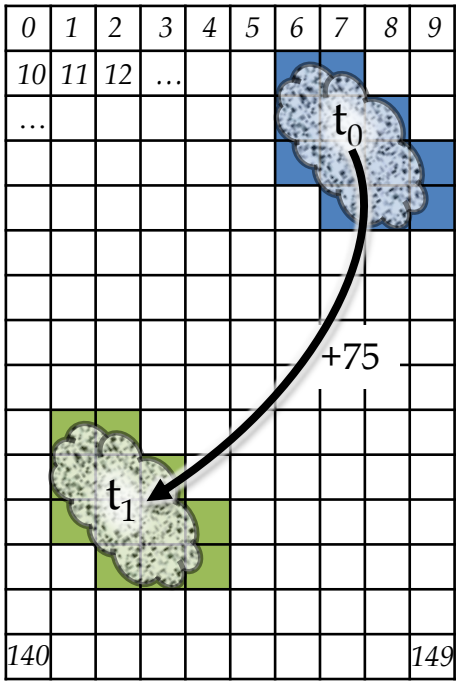
$$\vec{C_i} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} I_x \\ I_y \\ I_z \end{pmatrix} + \frac{1}{2} \cdot \begin{pmatrix} l_{Voxel} \\ l_{Voxel} \\ l_{Voxel} \end{pmatrix} \tag{5.14}$$

5.3.1. Translation mittels Basisversatz

Weist die Punktwolke eines mobilen Objektes eine rein translatorische Bewegung auf, so muss sie nicht mehrfach in Voxel umgewandelt werden. In diesem Fall ist es ausreichend, die Punkte einmalig in eine Voxelkarte oder Voxelliste einzutragen, und die Translation über einen Versatz der Basisadresse, wie in Gleichung 5.12, abzubilden. So kann der Versatz in der Implementierung des $\cap$- oder $\cup$-Operators auf die Basisadresse B der ortsfesten Voxelkarte addiert werden, um so die Voxel gegenüber der zweiten Datenstruktur des mobilen Objektes virtuell zu verschieben, bevor die $\&$ bzw. $\|$ Operationen angewendet werden (vgl. Abb. 5.3).

Da lediglich ganzzahlige Additionen ausgeführt werden müssen, ist eine empfindliche Perfomance-Steigerung gegenüber einer matrixbasierten, geometrischen Transformation der Punktwolke mit 28 Fließkommaoperationen pro Datenpunkt und dem erneuten Eintragen in die Datenstruktur gegeben.

Diese Technik wird in den Anwendungsfällen der Pfadplanung mit Rotations-Swept-Volu-men (Unterabschnitt 7.2.2), der Planung mit Bewegungsprimitiven (Unterabschnitt 7.2.3) und der Greifplanung (Unterabschnitt 7.2.7) eingesetzt.



(a) Addressierungsschema der Voxelliste

Voxeladressen		
t_0	Offset	t_1
16		91
17		92
26		101
27		102
28		103
36	+75	111
37		112
38		113
39		114
47		122
48		123
49		124

(b) Voxelliste vor und nach Translation

Abb. 5.3.: Translation der Voxelliste einer umgewandelten Punktwolke zwischen t_0 und t_1. Alle Voxeladressen der Liste werden um den Versatz inkrementiert. Dies erspart die geometrische Transformation der Punktwolke.

5.3.2. Voxelkarten mit mehrstufiger Auflösung

Ähnlich einer Bildpyramide in der 2D-Datenverarbeitung kann auch bei Voxelkarten eine bedarfsgesteuerte, schrittweise Verfeinerung der Verarbeitung realisiert werden, während die Vorteile des effizienten Datenzugriffs nutzbar sind. Dafür müssen mehrere Kopien der maximal auflösenden Ausgangs-Voxelkarte M_{Maxres} angelegt werden, die schrittweise gröber diskretisieren. Der Speicheraufwand einer Voxelkarte ist zwar, im Vergleich mit den anderen Datenstrukturen, prinzipbedingt groß, jedoch reduziert er sich mit jeder Halbierung der Auflösung um den Faktor acht. So kann der Speicherbedarf $Mem_{Multires}$ einer N-stufigen Kartenpyramide mit folgender Formel berechnet werden:

$$Mem_{Multires} = \sum_{n=0}^{N-1} \frac{\#Voxel_{Maxres}}{8^n} \cdot Mem_{Voxel} \tag{5.15}$$

An einem Beispiel mit realistischen Werten sieht man, dass der zusätzlich benötigte Speicher für eine Pyramide mit $N = 4$ Ebenen im Vergleich mit der original Karte sehr gering ist: Die Karte M_{Maxres} von der ausgegangen wird, soll $5{,}12\,\mathrm{m} \times 5{,}12\,\mathrm{m} \times 2{,}56\,\mathrm{m}$ mit $1\,\mathrm{cm}^3$ Voxeln abdecken. Benötigt ein Voxel dabei $1\,\mathrm{B}$ Speicher, ergibt sich ein Speicherbedarf von $512\,\mathrm{B} \times 512\,\mathrm{B} \times 256\,\mathrm{B} = 64\,\mathrm{MiB}$ für die erste Etage, und lediglich $\frac{64\,\mathrm{MiB}}{8} + \frac{64\,\mathrm{MiB}}{8^2} + \frac{64\,\mathrm{MiB}}{8^3} = 9{,}125\,\mathrm{MiB}$ für die weiteren drei Etagen der Pyramide, bzw. $Mem_{Multires} = 73{,}125\,\mathrm{MiB}$ für die gesamte Pyramide. Ähnlich wie der Speicherverbrauch sinkt auch die Zeit für

Kollisionsprüfungen mit der Voxelanzahl. Daher ist auch hier der entstehende Zusatzaufwand für mehrfache Prüfungen, verglichen mit der erzielbaren Beschleunigung, vernachlässigbar. Das hierarchische Verfahren, das in Abb. 5.4 skizziert ist, lässt sich auch mit Voxellisten umsetzen. Zeitmessungen dazu finden sich in Abschnitt 8.2.

In der praktischen Umsetzung werden die Zwischenstufen der Pyramide nicht genutzt und nur auf einer maximalen und minimalen Diskretisierung gearbeitet, da dies in der Evaluation zum höchsten Zeitgewinn führte.

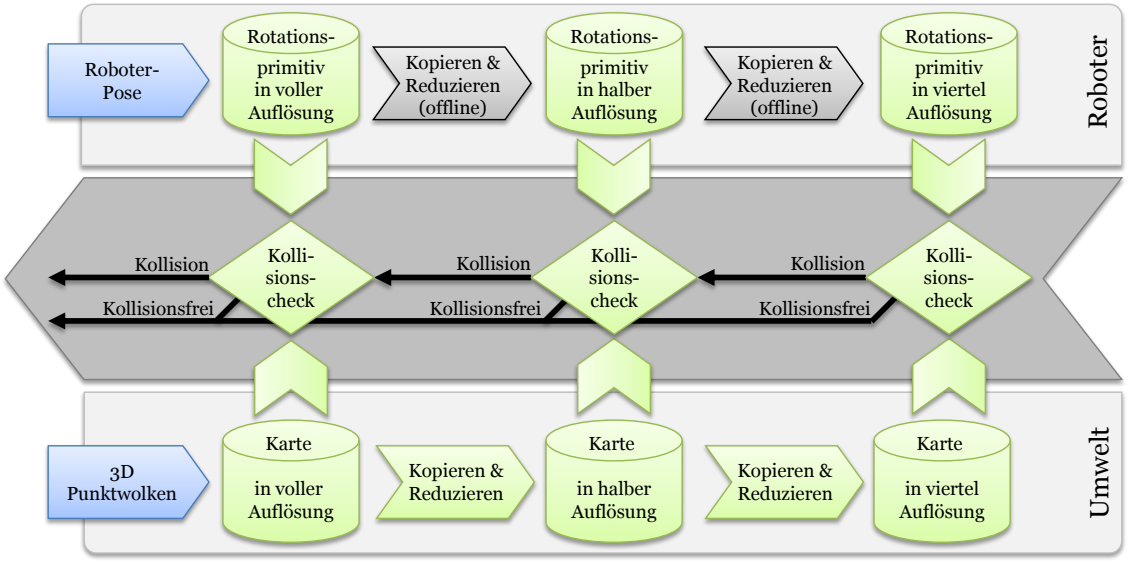


Abb. 5.4.: Hierarchische, bedarfsgesteuerte Kollisionsprüfung. Das Roboter-Modell wird offline in der Auflösung reduziert, während die Umweltdaten kontinuierlich komprimiert werden. Grüne Komponenten sind in CUDA implementiert.

5.4. Voxelliste

Auch eine Voxelliste ist prinzipiell ein eindimensionales Feld im GPU-Speicher. Jedoch werden in ihr nur Voxel eines bestimmten Typs vorgehalten. Somit lässt sich beispielsweise gezielt nur der belegte Raum speichern, womit der Speicherverbrauch für dünn besetzte Umgebungen gegenüber einer Voxelkarte drastisch reduziert wird. Effizienter als bei einer Voxelkarte ist dadurch auch das lineare Durchlaufen der Datenstruktur, da hierbei nicht über irrelevante Voxel iteriert werden muss, was die Laufzeit des $\cup$-Operators für Kollisionsprüfungen entsprechend beschleunigt. Ein Zugriff auf einzelne Voxel über geometrische Koordinaten ist hingegen nur mit Hilfe einer Suche möglich. Um den Suchaufwand zu minimieren, sind die Einträge der Liste anhand ihrer Voxeladresse sortiert und dedupliziert. Diese beiden Eigenschaften müssen bei der Verschmelzung zweier Listen durch den $\cap$-Operator oder bei der Voxelumwandlung einer dichten Punktwolke mit dem $\boxplus$-Operator aufrechterhalten werden.

Um eventuelle Duplikate aus der Liste zu entfernen, müssen alle Einträge mittels eines *Radix Sort* nach ihrer Voxeladresse aufsteigend geordnet werden. In der Folge liegen mehrfach existierende Voxel benachbart im Speicher. Anschließend läuft eine Pre-

fix-Summe rückwärts über die Nutzdaten und wendet dabei einen Voxeltypspezifischen Vereinigungsoperator auf Voxel an, die dieselbe Adresse aufweisen. Durch die rückwärts ablaufende Bearbeitung enthält somit der erste von mehreren ortsgleichen Voxeln die kombinierten Nutzdaten (Bitvektoren oder Belegtheitswahrscheinlichkeiten) der Dubletten. Daraufhin kann erneut vorwärts über die Liste gelaufen werden, um alle Duplikate zu löschen. Voxellisten weisen entsprechend einen hohen Aufwand bei ihrem Aufbau auf, der lineare Zugriff ist hingegen sehr effizient.

Je nachdem ob eine Voxelliste in Kombination mit einem Octree oder einer Voxelkarte einsetzt werden soll, handelt es sich bei den gespeicherten Voxeladressen um virtuelle Adressen in der zugehörigen Voxelkarte (nach Gleichung 5.12), oder um Morton-Codes im entsprechenden Octree. Die Adressierung ist um eine Basisadresse der Datenstruktur bereinigt und beginnt somit bei Null.

Neben den Adressen speichert die Voxelliste auch die geometrischen Koordinaten und die Nutzdaten der Voxel. Da die Implementierung in *Thrust* umgesetzt ist, werden alle Daten, entsprechend des Structure-of-Arrays-Prinzips, in drei gleich langen Listen gespeichert. Die Einträge am selben Listenplatz repräsentieren zusammen einen Voxel, weshalb in Thurst so genannte Zip-Iteratoren den Zugriff auf zusammengesetzte Elemente erleichtern. Ändert sich die Anzahl der Voxel in der Liste, so muss der Speicher aller drei Listen neu alloziert werden. Auch ein Umsortieren der Voxel muss auf allen Listen gleichermaßen angewendet werden. Die redundante Speicherung von Voxeladressen und geometrischen Koordinaten wurde aus Performancegründen gewählt. Da bei typischen Anwendungsszenarien die Listen kurz sind, fällt der zusätzliche Speicherverbrauch gering aus.

Der $\boxplus$-Operator zur Voxelumwandlung entspricht dem einer Voxelkarte. Er generiert zunächst für jeden Punkt einen eigenen Voxel. Fallen durch die Diskretisierung mehrere Punkte in denselben Voxel, so werden die mehrfachen Einträge bei der Sortierung zusammengefasst. Aufgrund dieses Aufwandes sollte $\boxplus$ nicht für hochfrequente Aktualisierungen eingesetzt werden.

5.5. Octree

Bei der geometrischen Modellierung von Volumendaten mittels Voxelkarten stellt der Speicher schnell eine Grenze für die mögliche Auflösung, oder das abzubildende Volumen dar, unabhängig davon, zu welchem Grad dieses Volumen belegt ist. Daher ist eine Repräsentation mit variabler Auflösung erstrebenswert, in der sich gleichförmige Bereiche (bspw. der Freiraum) speichereffizient zusammenfassen lassen. Eine weit verbreitete Datenstruktur, die diese Anforderung umsetzt, ist der Octree (eine baumförmige Struktur, deren Knoten je acht Verzweigungen aufweisen, siehe Abb. 5.5b). Bei einer geometrischen Interpretation (siehe Abb. 5.5a) repräsentiert der *Wurzelknoten* auf der höchsten Ebene einen Kubus, der alle abzubildenden Daten enthält. Er wird rekursiv in acht gleich große Kuben unterteilt, bis jeder Kubus nur noch gleichartige Daten enthält, oder eine vorgegebene Maximaltiefe erreicht ist, die bis zu n Einträge aufweist. Diese Baumstruktur und die auf ihr definierte Ordnung erlaubt eine schnelle Suche nach Elementen in $\mathcal{O}(\log n)$.

Während der Aufbau eines Octrees aus Datenpunkten in einer seriellen Implementierung durch eine Rekursion gut lösbar ist, bestehen für eine GPU-Implementierung durch den a priori unbekannten Speicher- und Laufzeitaufwand zwei Herausforderungen: Zunächst ist das bedarfsgesteuerte, inkrementelle Allozieren von Speicher während der Rekursion nach Feststellung 5 ineffizient. Weiterhin ist der Arbeitsaufwand von der Verteilung der Eingabedaten abhängig und somit nicht direkt parallelisierbar. Die folgenden Abschnitte beschreiben eine Herangehensweise, die eine effiziente heterogene Parallelisierung dennoch ermöglicht.

5.5.1. Stand der Technik

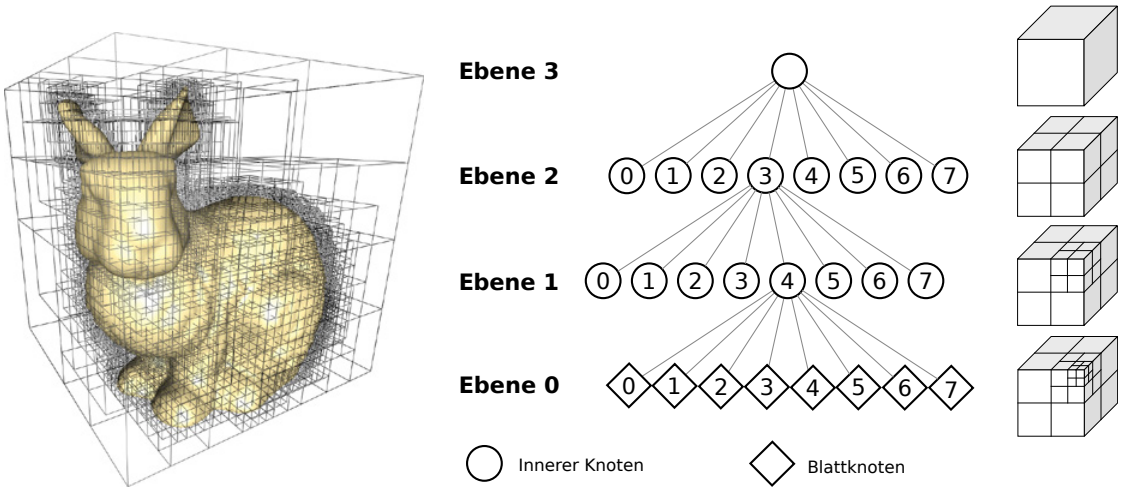
Die ersten Arbeiten zur Implementierung von Octrees wurden bereits 1980 von Meagher in [143] und Jackins [105] veröffentlicht. Während der Fokus dieser ersten Arbeiten noch auf der effizienten Manipulation und Darstellung der Datenstrukturen lag, folgten bereits 1984 erste Veröffentlichungen zur kollisionsfreien Bahnplanung für einen Manipulator [79] auf Octree-Basis, die dann 1986 von Herman um Rotations-Swept-Volumen erweitert [100] wurden. Auch gab es Versuche, Distanzinformationen in Bäumen zu speichern, um Kollisionsanfragen noch schneller zu beantworten [107]. In den 1990er Jahren wurden dann jedoch vermehrt Dreiecksnetzmodelle zur Darstellung und zur Planung mit 3D-Modellen verwendet und die Fortschritte der Octrees stagnierten.

Klassische Octree-Implementierungen speichern pro internem Knoten acht Zeiger auf Kindknoten, was bei kleinen Nutzdaten einen großen Zusatzaufwand an Speicher bedeutet. Eine Alternative dazu sind mittels Hashfunktion linear gespeicherte Bäume. Hier werden ausschließlich belegte Knoten vorgehalten, wobei jeder Knoten den Morton-Code seiner Position zusammen mit einem Bitvektor speichert, der angibt, welche seiner Kindknoten existieren. Über diese Informationen lassen sich Eltern- und Kindknoten adressieren. Auch die Tiefe im Baum und daraus die Größe des abgedeckten Volumens ist implizit repräsentiert. Die eigentliche Speicheradresse muss letztendlich über eine Hashfunktion aus dem Morton-Code abgeleitet werden. Diese Art der Bäume tauschen Speichereffizienz gegen Laufzeitaufwand ein und kommen daher für diese Arbeit nicht in Frage.

Eine relevante, modere CPU-Implementierung ist der äußerst speichereffiziente Octree von Borrman et al. [49], der je acht Kindknoten mit nur einem einzigen Zeiger adressiert. Die Implementierung ist jedoch nicht für dynamische Szenen ausgelegt und differenziert nicht zwischen freiem und unbekanntem Raum. Dies ist hingegen bei *Octomap* möglich, einer weit verbeitete CPU-Bibliothek, die Octrees mit beliebigen Nutzdaten erlaubt [104]. Die Implementierung ist jedoch weder speichereffizient noch besonders performant, was somit auch für das darauf aufbauende *ROS Collider* Paket gilt, das Kollisionsprüfungen mit dem Octree ermöglicht [95].

Die einzigen relevanten GPU-Octrees stammen aus der Spielewelt [155] oder der Computergrafik [185]. Bei ihnen handelt es sich jedoch um statische Datenstrukturen, die zur Laufzeit partiell geladen und wieder freigegeben werden (*Out-of-core* Aufbau), womit sie sich nicht für die bearbeiteten Problemstellungen eignen. Eine Arbeit, die hingegen den Aufbau eines Octrees auf der GPU durchführt, stammt von Karras [110]. Dieser verzichtet dabei jedoch auf die Propagierung von Knotenzuständen entlang des Baumes in

Richtung Wurzel, womit ein effizienter Abstieg im Baum, der für die Kollisionsprüfung unabdingbar ist, nicht gegeben ist. Eine weitere Arbeit, die sich mit der parallelen Manipulation von dynamischen Baumstrukturen auseinandersetzt, stammt von Chaudhary [56]. Hier liegt der Fokus auf der effizienten Umsetzung von boolschen Operationen, jedoch wird eine Hardware vorausgesetzt, die eine dynamische Allokation von Recheneinheiten erlaubt. Da dies ist bei GPUs nur mit hohen Latenzen möglich ist, ist der Ansatz hier nicht nutzbar. Somit ist keine vergleichbare Arbeit bekannt, die einen effizienten, parallelen Aufbau und eine parallelisierte Traversierung unterstützt, wie sie der im folgenden beschriebene dynamische GPU-Octree umsetzt.



(a) Überlagerung eines Oberflächenmodells mit einer Octree-Partitionierung [162].

(b) Baumstruktur der rekursiven Unterteilung des Wurzelknotens auf Ebene 3 bis zu Blattknoten auf Ebene 0.

Abb. 5.5.: Beispiele zur Verdeutlichung des Octree-Prinzips.

Definition 14. Die **Baum-Invariante** beschreibt den eindeutigen, gültigen Zustand eines gewurzelten Baumes, in dem alle inneren Knoten den ihren Kindknoten entsprechenden, zusammengefassten Zustand besitzen (*frei*, *belegt*, *teilweise belegt*, *unbekannt*). Die Invariante muss nach jeder Änderung am Baum geprüft und bei Bedarf wiederhergestellt werden. Ein parallelisiertes Verfahren dazu wird in Abschnitt 5.5.2 vorgestellt.

5.5.2. Umsetzung

In den folgenden Abschnitten werden Vorgehen beschrieben, die in der Octree-Implementierung von Florian Drews im Rahmen seiner Masterarbeit [22] umgesetzt wurden. Dafür sollen zunächst einige Begriffe und Grundlagen definiert werden. In dieser Arbeit wächst der Baum von oben nach unten. Seine *Wurzel* liegt somit auf der höchsten Ebene und die *Blattknoten* (Knoten ohne weitere Kindknoten) auf Ebene 0. Die Blattknoten weisen das kleinste adressierbare Volumen auf und bestimmen somit die räumliche Auflösung des Octrees. Zwischen Wurzel und Blattknoten liegen *innere Knoten*. Sie können jedoch, im Gegensatz zur üblichen Definition, auch ohne Kindknoten existieren. Alle Octree-Knoten, deren Größe sich über ihre Ebene im Baum definiert, werden auch als Voxel

bezeichnet. Die Kindknoten eines inneren Voxels sind eindeutig von 0 bis 3 nummeriert und über 40 Bit-Zeiger erreichbar (erlaubt 1 TB große Bäume). Die Implementierung ist auf Datensparsamkeit ausgelegt, weshalb implizite Daten, wie der Morton-Code eines Voxels, nicht in den Nutzdaten gespeichert sind. Auch die Zeiger auf die Kindknoten sind auf einen einzigen reduziert, da die Kinder in einem Array angeordnet sind und ihre Adressen somit durch Zeigerarithmetik bestimmbar sind (siehe Abb. 5.6). Dies ist durch eine, später beschriebene, konservative Speicherallokation möglich und verbessert den Zugriff bei der Parallelverarbeitung durch Memory Coalescing. Ein weiteres 8 Bit großes Statuswort pro Voxel enthält Verwaltungsinformationen, beispielsweise ob sein Subbaum zu aktualisieren ist.

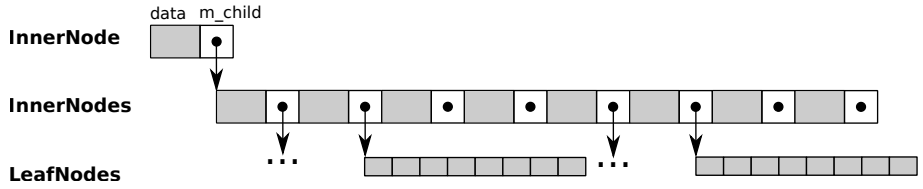


Abb. 5.6.: Speicherlayout von inneren Knoten und Blattknoten. Grafik aus [22].

Blattknoten können als deterministische oder probabilistische Voxel implementiert sein (siehe Abb. A.6 in Anhang A). Eine Umsetzung mit Bitvektor-Voxeln ist möglich, wurde bisher aber nicht benötigt. Im determinsistischen Fall müssen innere Knoten alle Zustände (*belegt*, *frei* und *unbekannt* auch gleichzeitig aufweisen können, um ihre Unterbäume korrekt zu repräsentieren. Der Speicherbedarf stellt sich wie in Tab. 5.1 dar.

Typ	**Deterministisch**		**Probabilistisch**	
	LeafNode	**InnerNode**	**LeafNodeProb**	**InnerNodeProb**
Größe [Byte]	1	8	2	8

Tab. 5.1.: Speicherbedarf der verschiedenen Knotentypen des Octrees

Speicherverwaltung

Ein Octree stellt eine dynamische Datenstruktur dar, da bei seinem sukzessiven Aufbau aus Sensordaten im Vorfeld nicht bekannt ist, wie viele Knoten benötigt werden. Ihr Speicher ist fortlaufend anzulegen, oder beim Zusammenfassen von Kindknoten zu verwerfen. Daher sind Strategien zu entwickeln, die diesen Widerspruch der dynamischen Speicherallokation aus Feststellung 5 möglichst auflösen.

Naheliegend ist zunächst die Zusammenfassung vieler kleiner Speicherreservierungen einzelner Knoten zu größeren Einheiten, die auf einmal angefordert werden. Neben der Zeitersparnis würde dies auch zu zusammenhängenden Speicherbereichen führen, was parallele Berechnungen durch Memory Coalescing beschleunigt. Allerdings könnten somit auch Speicherfreigaben nicht mehr auf Knotenbasis, sondern nur in denselben großen Einheiten erfolgen, was ohne eine Octree-spezifische Speicherverwaltung zu Fragmentierung führt. Da die Umsetzung einer eigenen Verwaltung aufgrund des hohen Aufwandes und der durch sie eingeführten zusätzlichen Latenzen nicht zielführend ist, wird eine andere Strategie verfolgt:

Ein regelmäßiger Neuaufbau des Octrees erlaubt es, zunächst zu viel Speicher zu reservieren und eine Fragmentierung während der Konstruktion des Baumes in Kauf zu nehmen. Nachdem der tatsächliche Speicheraufwand ermittelt ist, kann der komplette Baum zusammenhängend kopiert werden, um den fragmentierten Speicher freizugeben. Dieser Kompromiss zur Umsetzung einer dynamischen Datenstruktur ist in Abb. 5.7 gezeigt. Ausgelöst wird der Neuaufbau durch das Überschreiten von Grenzwerten des Speicherverbrauchs.

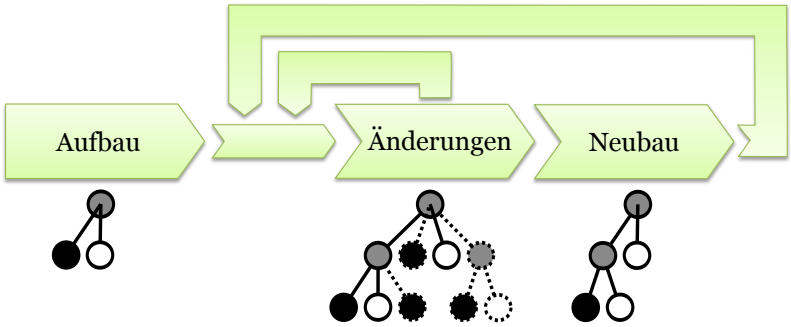
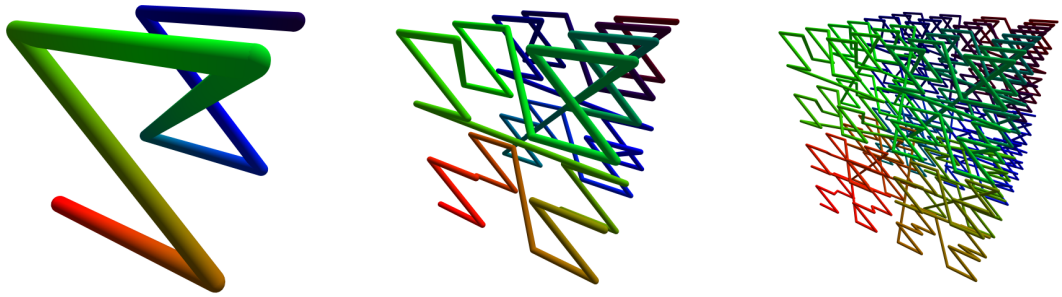


Abb. 5.7.: Zyklischer Neuaufbau des Octrees als Kompromiss zur Umsetzung einer dynamischen Datenstruktur auf der GPU.

Die finale Umsetzung des Octrees reserviert Speicher jedoch nicht auf Knotenebene, sondern vorausschauend in Blöcken von acht Voxeln, was der hohen Wahrscheinlichkeit geschuldet ist, dass Knoten meist mehr als einen Kindknoten aufweisen. Somit besitzt ein innerer Knoten in dieser Implementierung entweder keinen oder gleich acht Kindknoten, die dann im Speicher direkt nebeneinander liegen (mit den bereits beschrieben Vorteilen des Memory Coalescings).

Adressierung über Morton-Codes



(a) 2x2x2 Unterteilungen (b) 4x4x4 Unterteilungen (c) 8x8x8 Unterteilungen

Abb. 5.8.: Z-Kurve der 3D-Morton-Adressierung in drei Rekursionsschritten. Illustration von Asger Hoedt[1].

Um ausgehend von 3D-Koordinaten in einem Octree den zugehörigen Knoten zu finden, muss ein Abstieg im Baum erfolgen, der in jedem Knoten die Abstiegsrichtung durch drei Größer-/Kleiner-Vergleiche mit den Koordinaten bestimmt. Eine andere Art der Adressierung ist die Verwendung von Morton-Codes, wie sie in Abb. 5.8 dargestellt sind. Ihre

genaue Definition, sowie weitere Beispiele finden sich in Abschnitt A.3. Diese Codes lassen sich aus den Koordinaten ableiten und beschreiben implizit den Pfad von der Wurzel bis zum gesuchten Voxel in Form der zu verfolgenden Kindknoten. Durch diese Eigenschaft kann weiterhin sehr effizient der kleinste gemeinsame Elternknoten zweier Voxel bestimmt werden, was bei der parallelisierten Traversierung des Octrees hilfreich ist. Außerdem können Morton-Codes genutzt werden, um zu ermitteln, ob ein Voxel in den durch zwei 3D-Punkte aufgespannten Quader fällt und somit die Kollisionsdetektion beschleunigen. Diese Funktionen werden im Folgenden noch aufgegriffen.

Umgesetzt wurden in dieser Arbeit Morton-Codes mit $60\,\mathrm{Bit}$ Breite, womit pro Koordinate $2^{20}\,\mathrm{Bit}$ (also $\hat{=}1\,048\,576\,\mathrm{Voxel}$) adressierbar sind. Dies entspricht bei $1\,\mathrm{cm}^3$ Voxelgröße einem abgedeckten Raum von $\sim 1153\,\mathrm{km}^3$.

Aufbau

Beim Aufbau eines Octrees muss zwischen unterschiedlichen Voraussetzungen der Ausgangsdaten unterschieden werden. Diese können geometrisch sortiert oder unsortiert vorliegen. Weiterhin könnten die Ausgangsdaten unverarbeitet den verfügbaren Speicher überschreiten, was eine schrittweise Verarbeitung erzwingt (*Out-of-core* Aufbau). Hier soll von unsortierten Punktwolken aus einem oder mehreren 3D-Sensoren ausgegangen werden, deren Größe einen *In-core* Aufbau erlaubt.

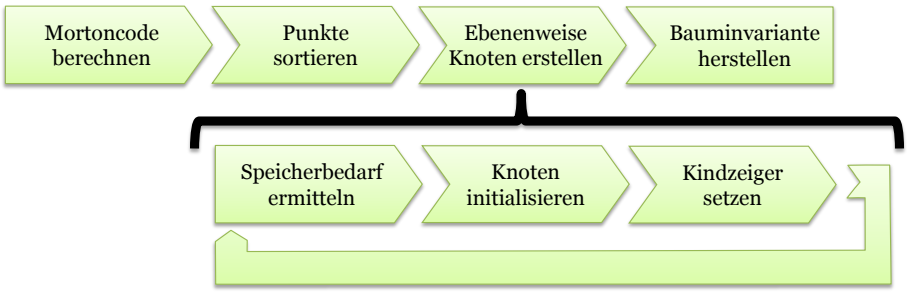


Abb. 5.9.: Octree Aufbau aus einer unsortierten Punktwolke in vier Schritten.

Die Erstellung des Octrees erfolgt nach den vier, in Abb. 5.9 gezeigten, Schritten:

1. Zunächst sind die Morton-Codes aller Eingabedaten zu berechnen. Dabei findet automatisch eine Diskretisierung mit der maximal unterstützten Auflösung statt. Dieser Schritt kann problemlos parallelisiert erfolgen, da keine Abhängigkeiten im Prozess oder in den Daten vorliegen.

2. Liegen die Codes vor, werden sie für eine effizientere parallele Verarbeitung sortiert. Dies geschieht mittels parallelisiertem Radix-Sort (siehe Unterabschnitt A.5.3).

3. Der wichtigste Schritt ist dann der ebenenweise Aufbau des Baumes, ausgehend von den Blättern in Richtung der Wurzel. Hierfür sind drei Teilschritte auszuführen:

[1] Blog von Asger Hoedt: `http://asgerhoedt.dk/?p=276`

a) Um die benötigte Anzahl an Elternknoten für alle belegten Kinder zu ermitteln, werden die Präfixe aller Kinder-Morton-Codes auf Gleichheit untersucht und ungleiche Präfixe gezählt. Da die Codes sortiert sind, ist es ausreichend, benachbarte Einträge zu vergleichen und das binäre Ergebnis in einem Array zu speichern. Das Array kann dann über eine parallele Reduktion (siehe Unterabschnitt A.5.2) mit einem Zähloperator zusammengefasst werden.
b) Mit der nun bekannten Anzahl an benötigten Elternknoten wird der Speicher für alle ihre Kindknoten reserviert, in dem dann Knoten mit Initialwerten angelegt werden. Wie bereits beschrieben, sind grundsätzlich alle acht Kindknoten zu reservieren, auch wenn sie nicht alle benötigt werden.
c) Der letzte Schritt unterscheidet sich zwischen Blattebene und inneren Ebenen. Im Falle der Blattebene müssen die Knoten, die tatsächlich belegt sind (also Daten mit ihrem Morton-Code vorhanden sind), entsprechend markiert werden. Im Falle einer inneren Ebene ist der Zeiger auf den ersten der acht Kindknoten zu setzen. Dieser Zeiger wurde bereits im vorhergehenden Zählschritt aus einer Präfix-Summenberechnung (siehe Unterabschnitt A.5.1) abgeleitet.

4. Im letzten Schritt sind die Zustände der inneren Knoten in Abhängigkeit ihrer Kinder zu setzen, um die Baum-Invariante herzustellen. Details hierzu folgen in Abschnitt 5.5.2.

Da aus Gründen der Speichereffizienz die Morton-Codes der Knoten nicht explizit vorgehalten werden, müssen diese während dem Aufbau des Baumes temporär gespeichert werden, um im nächsten Durchgang als Berechnungsgrundlage der nächsten Schicht bereitzustehen.

Die folgenden Formeln werden beim Aufbau des Octrees genutzt, um die Knotenpositionen und die Zeiger auf Kindknoten zu berechnen:

$$c_k^l := \begin{cases} 1, & \text{falls } k = 0 \\ 1, & \text{falls } \mathit{prefix}^l(m^l[k-1]) \neq \mathit{prefix}^l(m^l[k]) \\ 0, & \text{sonst} \end{cases} \tag{5.16}$$

$$\mathit{offset}^l_{\mathit{parent}}(i) := \sum_{k=0}^{i} c_k^l - 1 \tag{5.17}$$

$$\mathit{offset}^l_{\mathit{node}}(i) := 8 \cdot \mathit{offset}^l_{\mathit{parent}}(i) + \mathit{child}^l(m^l[i]) \tag{5.18}$$

$$\mathit{offset}^l_{\mathit{child}}(i) := 8 \cdot i \tag{5.19}$$

Als Eingabedaten dienen n temporär gespeicherte Morton-Codes m^l der Blätter ($l = 0$) bzw. der vorher bearbeiteten Knotenebene l. Ausgehend vom i-ten Morton-Code $m^l[i]$ kann die Position des zu ihm gehörenden Knotens im Feld aus Knoten N^l der Ebene l mittels $\mathit{offset}^l_{\mathit{node}}(i)$ berechnet werden. Somit steht sein erster Kindknoten an der Position $\mathit{offset}^l_{\mathit{child}}(i)$ des Feldes N^{l-1} der Ebene $l-1$. Dabei ist $0 \leq \mathit{child}^l(m^l[k]) < 8$ die Nummer des Kindknotens. c_k^l indiziert, ob Morton-Code $m^l[k]$ der kleinste Kindknoten unter den Knoten mit gleichem Elternknoten darstellt. Ist dies der Fall (also wenn für Morton-Code $m^l[i]$ gilt: $c_i^l = 1$), dann wird dieser Knoten für die Erstellung der nächsten

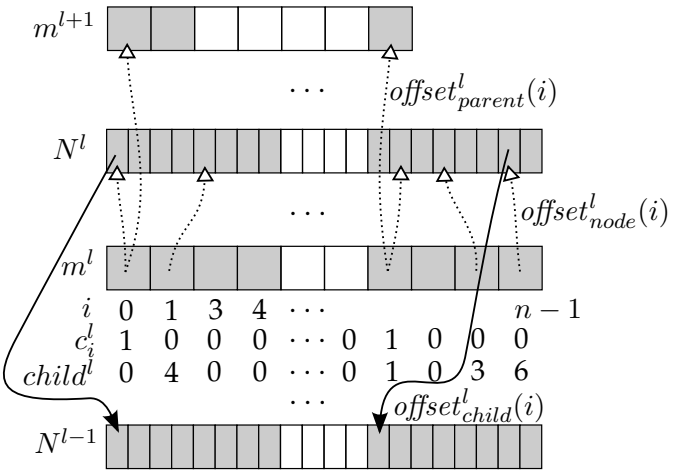


Abb. 5.10.: Illustration der Gleichung 5.16 zur Berechnung von Knotenpositionen (gestrichelte Linien) und Zeigern auf Kindknoten (durchgezogene Linien).

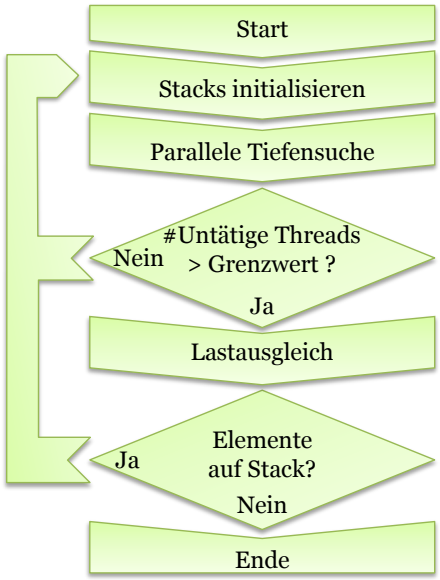


Abb. 5.11.: Paralleles traversieren des Baumes mit heuristischem Lastausgleich.

Ebene benötigt, und daher in das temporären Feld m^{l+1} der Größe $offset^l_{parent}(n-1)+1$ an Stelle $offset^l_{parent}(i)$ kopiert.

In Abb. 5.10 sind die Zusammenhänge anhand eines Beispiels skizziert. Grau hervorgehoben sind Blöcke von Knoten, die den gleichen Elternknoten besitzen.

Propagieren von Statusinformationen: Baum-Invariante herstellen

Zur Herstellung der Baum-Invariante wird der Baum komplett traversiert, um dabei Informationen durch alle Ebenen zu propagieren. Knoten, die dabei einen *ungültigen* oder *nicht initialisierten* Zustand aufweisen, erben den Zustand ihres nächsten gültigen Elternknotens. Da bei der Erstellung des Baumes alle Knoten, die nicht explizit als *belegt* markiert wurden, zunächst *nicht initialisiert* sind, entscheidet letztendlich der Wurzelknoten über ihren Zustand. In einer sensoriell erfassten Umgebung sollte daher die Baumwurzel als *unbekannt* markiert sein, wohingegen sie in einer vollständig bekannten Umwelt (z.B. durch ein geometrisches a priori Modell) mit dem Zustand *frei* starten kann. In einem zweiten Durchlauf (von unten nach oben) wird der Wurzelzustand dann auf seinen wahren Wert aktualisiert.

Dieser Mechanismus kann auch dazu verwendet werden, größere kubische Volumen im Baum effizient zu verändern. Dazu muss lediglich der Elternknoten des Teilbaumes vor der Herstellung der Invariante angepasst werden, um alle Kindknoten automatisch zu verändern.

Hochparalleles Traversieren des Baumes mit Lastausgleich

Nicht nur zur Durchsetzung der Baum-Invariante, sondern auch für viele weitere, im Folgenden vorgestellten Operationen, ist das Traversieren des Octrees eine grundlegende Funktion. Die dafür verwendete Tiefen- oder Breitensuche ist in einer sequentiellen Implementierung problemlos umzusetzen, während die Datenabhängigkeit beider Suchverfahren eine geradlinige Parallelisierung jedoch verhindert: Der Arbeitsaufwand für das Ablaufen eines dünn besetzten Baumes lässt sich ohne genaue Kenntnis der Baumstruktur nicht in gleiche Arbeitspakete aufteilen. Eine Tiefensuche, die parallel auf unterschiedlichen Teilbäumen abläuft, hat abhängig von der Baumstruktur sehr unterschiedliche Laufzeiten, da manche Teilbäume früher enden, als andere. Eine Breitensuche dagegen, die einzelne Ebenen parallel abarbeiten könnte, benötigt neben einer größeren Menge an Speicher (zum Zwischenspeichern aller Voxel der Ebene) auch eine Synchronisation aller Threads vor jeder neuen Ebene, was eine hohe Latenz einführt.

Um diese wichtige Funktion dennoch gewinnbringend zu parallelisieren, wurde eine Tiefensuche umgesetzt, die mit Hilfe eines bedarfsgesteuerten Lastausgleichs für eine gleichmäßige Umverteilung der Arbeitspakete sorgt. Das Verfahren orientiert sich an der Arbeit von Lauterbach et al. [130] zur Traversierung der dort verwendeten BVHs, wurde aber um eine Heuristik zur Aufwandsabschätzung erweitert. Der Lastausgleich wird ausgelöst, wenn die Anzahl untätiger Threads einen Grenzwert überschreitet. Das Flussdiagramm in Abb. 5.11 skizziert den Algorithmus.

Ein unausgeglichener Lastzustand, wie er in Abb. 5.12 links gezeigt ist, definiert sich über unterschiedlich gefüllte Arbeitsstapel der laufenden Threads. Ziel des Lastausgleiches ist es, den Aufwand der Elemente aller Arbeitsstapel gleichmäßig auf die Threads umzuverteilen.

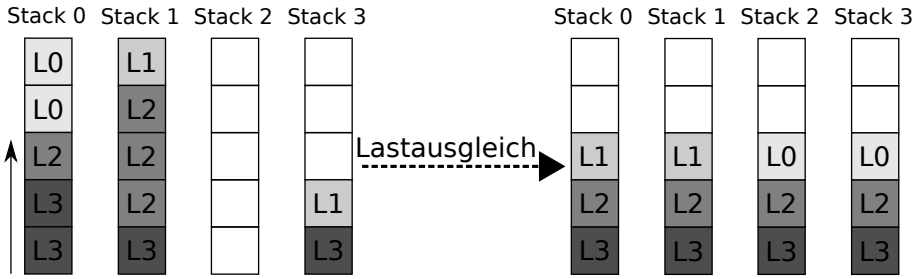


Abb. 5.12.: Lastausgleich unter Berücksichtigung des geschätzten Arbeitsaufwandes.

Da jedes Element den Wurzelknoten unterschiedlich tiefer Teilbäume repräsentieren kann, wäre eine Aufteilung anhand der Anzahl der Elemente nicht ausreichend. Daher bewertet die implementierte Lastverteilung die Arbeitselemente über eine Heuristik, und ordnet Elemente mit ähnlichem Aufwand auf gleicher Höhe im Arbeitsstapel an. Das Ergebnis ist in Abb. 5.12 rechts zu sehen. Die Heuristik basiert auf der Annahme, dass Teilbaum-Wurzelknoten, die auf einer höheren Ebene im Gesamtbaum liegen, potentiell weiter absteigen müssen und daher mehr Arbeitsaufwand ($maxArbeit$) verursachen:

$$Knoten\ n, m \mid Ebene(n) > Ebene(m) \implies maxArbeit(n) > maxArbeit(m) \tag{5.20}$$

Das Resultat erfordert durchschnittlich weniger Lastausgleichsschritte und sorgt für einen höheren Parallelisierungsgrad. Voraussetzung ist jedoch, dass die Ebene jedes Arbeitselements bekannt ist, und jeder Arbeitsstapel die darüber definierte *Arbeitsstapel-Invariante* erfüllt:

$$S[n], \quad \forall\, i \in [1, n) \mid Ebene(S[i-1]) \geq Ebene(S[i]) \tag{5.21}$$

Ist diese Gleichung erfüllt, können über Prä- und Postfixsummen sehr effizient die Positionen der sortierten Elemente in den Arbeitsstapeln berechnet werden. Einzelne CUDA Blöcke sind dabei jeweils für die Analyse eines Arbeitsstapels zuständig, wobei die Etagen des Stapels über die Threads pro Block parallel bearbeitet werden. Nach der Zähl- und Sortierphase können die Elemente parallel an ihre neuen Positionen geschrieben werden und erfüllen dann noch immer die Arbeitsstapel-Invariante. Der Pseudocode dieses parallelisierten Lastausgleiches findet sich in Abschnitt A.7, Algorithmus 5. Über kleine, problemspezifische Anpassungen kann die balancierte Tiefensuche für viele unterschiedliche Aufgaben im Octree eingesetzt werden.

Erweiterung eines bestehenden Octrees

Nach der Vorstellung der parallelen Traversierung sollen nun Techniken zur Erweiterung eines bestehenden Octrees beleuchtet werden, da diese über die Leistung der Datenstruktur in dynamischen Umgebungen entscheidet. Die Erweiterung ist die komplementäre Ergänzung zum bereits beschriebenen zyklischen Neuaufbau (siehe Abb. 5.7): Einerseits benötigt sie weniger Rechenzeit und weitaus weniger temporären Speicher, jedoch steigt durch sie die Speicherfragmentierung, da es bei der Erweiterung nicht möglich ist, Speicherbereiche wiederzuverwenden oder freizugeben, auch wenn Knoten zusammengefasst werden. Vorausgesetzt wird, dass die Menge der einzufügenden Voxel alle von derselben Größe bzw. Ebene sind, diese nach ihrem Morton-Code sortiert vorliegen und keine Voxel denselben Morton-Code aufweisen. Das Einfügen neuer Daten geschieht in zwei Schritten: Zunächst werden neue Knoten erstellt und über Kindzeiger in den bestehenden Baum eingehängt. Daraufhin muss die Baum-Invariante wiederhergestellt werden, wobei auch Knoten gleichen Zustands verschmolzen werden.

Der erste Schritt zur Erweiterung der Baumstruktur gliedert sich in dieselben Teilschritte wie der Aufbau eines neuen Octrees, der bereits in Abschnitt 5.5.2 dargelegt wurde, wobei jedoch die bestehende Baumstruktur zu berücksichtigen ist:

1. **Speicherbedarf ermitteln:** Für jeden neu hinzuzufügenden Blattknoten ist der Baum zu traversieren, um zu bestimmen, ob fehlende innere Knoten einzufügen sind. Da unterschiedliche Voxel dieselben Elternknoten aufweisen können, muss bei einer Parallelisierung des Zählens ein doppeltes Traversieren von inneren Voxeln verhindert werden. Um die ineffiziente, naive Verwendung von atomaren Mutexes pro Knoten zu vermeiden, wurde dafür ein Verfahren umgesetzt, welches die Serialisierung über die Sortierung der Voxel nach ihren Morton-Codes löst. Durch diese ist sichergestellt, dass Voxel mit demselben Elternknoten nebeneinander liegen und jeder Voxel durch Prüfung seines Nachbarn feststellen kann, ob er der Kindknoten mit der kleinsten ID ist. Wenn nun jeweils ausschließlich die Knoten mit den kleinsten IDs bei der Traversierung fortschreiten und dabei das Zählen übernehmen, können keine Doppelungen auftreten. Die Zählung der benötigten Knoten (und somit

auch die Prüfung auf die kleinste Kind-ID) geschieht pro Ebene über eine Präfixsumme. Bei ihrer Ausführung wird gleichzeitig der als letztes erreichte, bereits im Baum existierende Knoten und dessen Ebene gespeichert. Diese Information wird für das Setzen der Kindzeiger im dritten Schritt benötigt und müsste anderenfalls durch ein zusätzliches Traversieren des Baumes erneut bestimmt werden.

2. **Speicher reservieren und Knoten initialisieren:** Nachdem die Anzahl der inneren Knoten sowie der Blattknoten bekannt ist, kann deren benötigter Speicher auf einmal reserviert werden. Alle neuen Knoten werden zunächst als undefiniert markiert.

3. **Knotenzustand und Kindzeiger setzen:** Im letzten Schritt müssen die neuen Knoten in den bestehenden Baum gehängt und dieser stellenweise aktualisiert werden. Auch hier wird die Sortierung der Voxel genutzt, um jeweils nur den Knoten mit der kleinsten Kind-ID beim Setzen der Kindzeiger zu verfolgen, da diese sonst auch mehrfach überschrieben würden. Die Knoten im bestehenden Baum, die durch die Erweiterung erstmals Kindknoten erhalten, sind bereits aus der Traversierung im ersten Schritt bekannt: Sie waren die letzten erreichten Knoten beim Abstieg in Richtung eines neu erzeugten Blattknotens. Ihr Zustand lässt sich mit dem Zustand der neuen Kindknoten verrechnen und entsprechend aktualisieren.

Nach der Ausführung dieser drei Teilschritte sind alle neuen Knoten in den Baum eingefügt. Es ist jedoch sehr wahrscheinlich, dass es durch die neuen Kindknoten zu einer Verletzung der Baum-Invariante gekommen ist, weshalb diese nachträglich wiederhergestellt werden muss. Verfahren dazu werden im nächsten Abschnitt aufgezeigt.

Baum-Invariante durchsetzen

Zum besseren Verständnis soll zunächst ein naiver, sequentieller Ansatz dargelegt werden. Dieser Algorithmus iteriert über die, als bekannt vorausgesetzte, Menge der geänderten Voxel. Für jeden wird der Baum in Richtung Wurzel abgelaufen, um dabei jeden inneren Knoten in den Zustand zu versetzen, der sich aus der $\|$ -Verknüpfung seiner acht Kindknoten ergibt. Die Kosten C des nötigen Berechnungsaufwandes müssen über die Baumtiefe d und der Menge der geänderten Knoten m auf $C \approx m \cdot d \cdot 8$ abgeschätzt werden, da keinerlei Synergien genutzt werden und das Verfahren wegen dem gleichzeitigen Zugriff auf innere Knoten nicht gut parallelisierbar ist.

Gewünscht wäre es hingegen, den gemeinsamen Pfad mehrerer Knoten nur einmalig abzulaufen, womit sich die Kosten auf $C \approx 8/7 \cdot m + 8 \cdot (d - log_8(m))$ (vgl. Gleichung 6.8 zur Aufwandsabschätzung der Kollisionsprüfung) reduzieren ließen. Hierbei entsteht jedoch offensichtlich eine Datenabhängigkeit, da vorausgesetzt wird, dass alle auszuwertenden Kindknoten bereits aktualisiert wurden, bevor höhere innere Knoten bearbeitet werden können. Um damit umgehen zu können, wurde die bereits definierte Tiefensuche mit Arbeitsstapel zu einer *eingeschränkten Zwei-Phasen-Tiefensuche mit Lastausgleich (Load-Balancing Propagate)* erweitert, die den Baum parallelisiert einmal von oben nach unten und danach von unten nach oben traversiert. Auf dem Weg in Richtung Blattknoten werden noch nicht initialisierte Knoten auf den Zustand ihrer Elternknoten gesetzt (*Abstieg*), während auf dem Rückweg von den Blättern zur Wurzel die Kindknoten zusammengefasst und der Status ihres Elternknotens aktualisiert wird (*Aufstieg*). Wie bereits erklärt,

können einzelne Threads bei der Traversierung ungleiche Arbeitslast erzeugen, insbesondere durch die Expansion von Knoten, die vorher voll belegt oder komplett frei waren, und daher keine Kindknoten aufwiesen. Das Ausstatten mit Kindknoten stellt gegenüber der Aktualisierung vorhandener Knoten bei der Traversierung einen deutlichen Mehraufwand dar. Daher ist die Lastbalancierung ein sehr wichtiger Bestandteil. Örtlich eingeschränkt ist das Verfahren insofern, dass es gezielt nur die Regionen des Baumes abläuft, die von einer Aktualisierung betroffen sein können. Somit wird eine hohe Datenlokalität erreicht und gegenüber dem Ablaufen des kompletten Baumes viel Zeit gespart. Die relevanten Regionen ergeben sich aus dem maximalen Sichtfeld der auszuwertenden Sensoren. Die Elemente der Zwei-Phasen-Tiefensuche, die auf dem Arbeitsstapel liegen, sind so gewählt, dass sie klein sind und gleichzeitig das Memory Coalescing optimieren. Für die parallele Abarbeitung führt jeder CUDA Block eine Tiefensuche mit eigenem Arbeitsstapel aus. Da die Verteilung des Arbeitsaufwands aber nicht a priori bekannt ist, müssen die Elemente der Stapel zur Laufzeit zwischen den Threads aufgeteilt werden. Hier kommt die Technik zur dynamischen Lastverteilung, die in Abschnitt 5.5.2 bzw. Algorithmus 5 im Anhang bereits für die Kollisionsprüfung entwickelt wurde, zum Einsatz. Der komplette Ablauf kann in Abschnitt A.7 in Algorithmus 7 nachvollzogen werden. Eine Besonderheit der Lastbalancierung hierbei ist eine potentielle Blockade des parallelen Algorithmus, der durch Abhängigkeiten zwischen *Aufstiegs*-Arbeitselementen auf den Stapeln entstehen kann: Sind die Stapel zu groß zur simultanen Abarbeitung aller Elemente und sind genau die Elemente inaktiv, auf welche die aktiven Threads warten, führt dies zum Stillstand. Daher verfügen die Arbeitselemente über eine Boolsche Variable, die anzeigt, ob der zugehörige Thread Berechnungsfortschritte erzielen kann. Über die Variable kann bei der Balancierung entschieden werden, ob alle Threads auf Ressourcen warten und in diesem Fall der Algorithmus neu zu starten ist.

Extrahieren von Voxeldaten

Sollen die Daten des Octrees ohne Kenntnis der Baum-Datenstruktur extern weiterverarbeitet werden, bietet es sich an, diese in Form einer Liste aus Voxeln zu exportieren. Jeder Voxel speichert dabei seine kartesische Position und Kantenlänge. Dafür wird der Baum mit der bereits vorgestellten lastbalancierten Tiefensuche abgelaufen und jeder Blattknoten bzw. jeder innere Knoten, der keine Kinder aufweist, kopiert, wobei sein Morton-Code in kartesische Koordinaten umgerechnet wird. Da die Octree-Knoten ihren Morton-Code nicht explizit speichern, wird dieser beim Abstieg im Baum sukzessive aus dem Code seines Elternknotens generiert (siehe Gleichung A.4 im Anhang). Soll nicht der komplette Baum kopiert werden, lässt sich die Suche anhand örtlicher Grenzen, der maximalen Detailstufe (Abstiegstiefe) oder dem Knotenzustand beschränken. Diese Einschränkungen werden beispielsweise für die Visualisierung aus Abschnitt 5.7 genutzt, um nur den Ausschnitt des Octrees in der benötigten Auflösung zu extrahieren, der sich im Sichtfeld der virtuellen Kamera befindet.

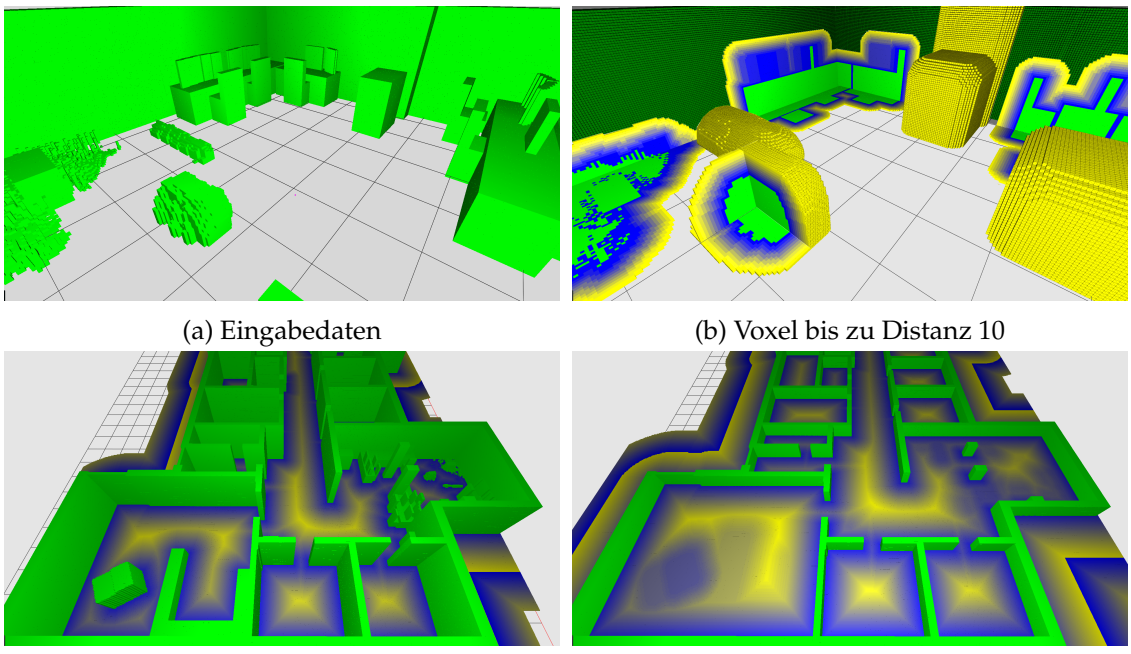
Da vor der Traversierung des Baumes nicht bekannt ist, wie viele Knoten zu kopieren sind, kann der Speicher für die Voxelliste nicht im Voraus reserviert werden. Daher stehen zwei Strategien zur Verfügung: Entweder kann der benötigte Speicherplatz großzügig abgeschätzt und iterativ erweitert werden, falls die Extraktion mehr Platz benötigt, oder der Baum muss zweimal traversiert werden, um zunächst die benötigten Voxel zu

zählen, bevor diese extrahiert werden. Für die Visualisierung wurde das erste Verfahren gewählt, da hier eine unvollständige Extraktion keine sicherheitsrelevanten Probleme verursacht (vgl. Abschnitt 5.7.1 zum Thema Visualisierung).

Zusammenfassung

Die vorgestellte, sehr speichereffiziente Implementierung eines Octrees nutzt unterschiedliche Strategien, um die dynamische Natur dieser Datenstruktur so weit zu kaschieren, dass eine Parallelisierung des Aufbaus und der Traversierung des Baumes auf der GPU möglich werden. Dafür wurden zwei grundlegende Techniken realisiert: Zum einen eine hochgradig an die Zielhardware angepasste Technik des probabilistischen Lastausgleichs zur Arbeitsverteilung zwischen CUDA-Blöcken. Zum anderen eine präventive Überallokation von Speicher in Kombination mit einem bedarfsgesteuerten Neuaufbau der Datenstruktur zur Vermeidung von Speicherfragmentierung. Die Unterstützung von deterministischen und probabilistischen Voxeln im Baum deckt sowohl die Roboter- als auch die Umweltmodellierung ab, während in beiden Fällen zwischen belegten, freien und unbekannten Volumen unterschieden werden kann. Quantitative Details zur Leistungsfähigkeit finden sich in der Evaluation in Abschnitt 8.3.

5.6. Distanzkarten



(a) Eingabedaten

(b) Voxel bis zu Distanz 10

(c) Voronoi-Regionen in Bodenebene

(d) Voronoi-Regionen in 2 m Höhe

Abb. 5.13.: Teilweise angeschnittenes Distanzfeld einer Laborumgebung. Kombination mehrerer Schnitte durch den Raum.

Neben den Planungsansätzen aus Kapitel 7 existiert in der Robotik der weit verbreitete Ansatz zur Navigation anhand von Distanzkarten mit so genannten Potentialfeld-Planern. Hierbei wird in einer diskretisierten Datenstruktur in jeder Zelle die Distanz zum nächstgelegenen Hindernis gespeichert (siehe Abb. 5.13). Diese Distanzen können invertiert und als abstoßende Potentiale interpretiert werden, die den Roboter von Hindernissen fernhalten. Übt das Ziel gleichzeitig eine anziehende Kraft auf den Roboter aus, kann dieser sich, wie in Abb. 5.14 gezeigt, entlang des Gradienten im kombinierten Vektorfeld sicher ins Ziel bewegen. Da zusätzlich zur Kollisionserkennung auch beliebige Annäherungen detektierbar sind, reicht das Anwendungsgebiet von Distanzkarten über die reine kollisionsfreie Bewegungsplanung hinaus, bis hin zur Freiraumoptimierung.

Unterschiedliche Verfahren zur Berechnung von Distanzfeldern werden im folgenden untersucht.

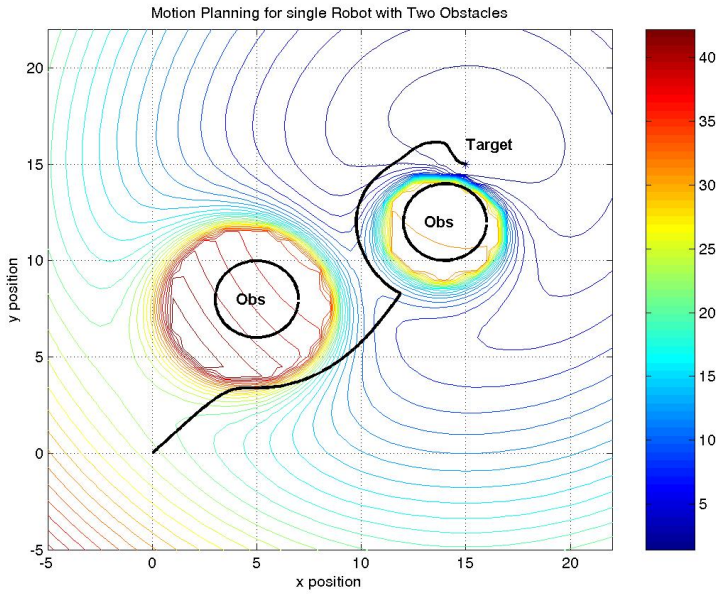


Abb. 5.14.: Beispiel eines Pfades entlang des Gradienten in einem kombinierten Potentialfeld aus abstoßenden Kräften der Hindernisse und einer anziehenden Kraft des Zieles (Grafik aus [132]).

5.6.1. Zielstellung

Aufgrund von begrenzter Rechenkapazität wurden Distanzkarten, deren Berechnungsaufwand sehr hoch ist, in vorhergehenden Arbeiten meist nur zweidimensional erstellt und ausgewertet [172]. In diesem Kapitel soll, basierend auf einer Parallelisierung und dem damit einhergehenden Performancegewinn, die Berechnung hochauflösender, dreidimensionaler Distanzfelder mit einer hohen Aktualisierungsrate ermöglicht werden. Dies erlaubt die Umsetzung von online-fähigen Potentialfeldplanern auf Basis von live Punktwolken einer dynamischen Umgebung. Weiterhin wird untersucht, ob es zielführend ist, bei Änderungen in der Umwelt nur betroffene Ausschnitte der Distanzkarte neu zu berechnen, oder die gesamte Karte neu zu erstellen.

Vorausgesetzt wird eine exakte Lokalisierung des Sensors bzw. des Roboters bspw. durch Simultaneous Localization and Mapping (SLAM). Wäre dieses nicht gegeben, müssten zeitlich aufeinander folgende Aufnahmen mittels Iterative Closest Point (ICP) o.ä. zunächst aneinander ausgerichtet werden. Als Vorverarbeitungsschritt zur Filterung und Ausdünnung der Sensordaten werden die aufgenommenen Punktwolken zunächst in einer probabilistischen Voxelkarte aggregiert, um dann mit ihren belegten Voxeln die Distanzberechnung durchzuführen.

5.6.2. Verwandte Arbeiten

Bei der Berechnung von Distanzkarten können unterschiedliche Metriken eingesetzt werden. Für die Robotik ist lediglich die euklidische Distanz von Bedeutung, weshalb in dieser Arbeit ausschließlich Euklidische Distanz Transformationen (EDTs) betrachtet werden. Diese lassen sich wiederum in approximierende und exakte Verfahren aufteilen. Speichern die Algorithmen in jedem Eintrag ihrer Datenstruktur nicht nur die Distanz des nächstgelegen Hindernisses, sondern auch dessen Position, dann entsprechen die generierten Karten einem Voronoi-Diagramm.

Die Arbeiten von Jones [106] und Fabbri [77] geben eine Übersicht über 2D- und 3D-Distanztransformationen, von denen die meisten sequentiell angelegt sind. Sie verwenden entweder Wellenfronten, die sich von Hindernissen aus kreisförmig ausbreitende oder mehrere lineare Abtastungen der kompletten Datenstruktur, um eine EDT anzuwenden.

Neben diesen sequentiellen Ansätzen existieren jedoch auch solche, die eine Parallelisierung auf GP-GPUs erlauben und daher hier von Interesse sind. Hierzu gehören der *Jump Flooding Algorithm* (JFA) [173, 174], der *Parallel-Banding-Algorithm* (PBA) [54], der *Schneider, Kraus und Westermann Algorithm* (SKW) [182] und der *Fast Hierarchical Algorithm* (FHA) [66]. Aufgrund ihrer Vorteile wurden von diesen vier Ansätzen PBA und JFA weitergehend untersucht: PBA als schnellster Kandidat für große Eingabedaten (SKW wurde in [54] langsamer als PBA getestet) und JFA für kleinere Karten (da er eine einfache Struktur und weniger Verwaltungsaufwand aufweist [54]). Weiterhin setzt PBA als einziger der vier Algorithmen eine exakte Berechnung der EDT um, während FHA und JFA Approximationsfehler aufweisen, die in [66] verglichen werden. Die GP-GPU optimierte Variante des SKW aus [182] basiert auf der Vektor Distanz Transformation von Danielsson [179] und weist daher wie diese eine obere Fehlerschranke der berechneten Distanzen von 0.091 Voxeln auf.

Alle gelisteten Ansätze erfordern eine Neuberechnung des kompletten Distanzfeldes, sobald Änderungen in den Ausgangsdaten auftreten. Somit erscheint ihre Anwendung zunächst bevorzugt in statischen Fällen sinnvoll. Im Gegensatz dazu ist bspw. der *Brushfire Algorithmus* aus [127] in der Lage, in dynamischen Szenen gezielt nur die Bereiche des Distanzfeldes zu aktualisieren, in denen sich Änderungen auch auswirken. Dennoch ermöglicht die hochparallele Berechnung von „statischen“ Ansätzen eine vielfach effizientere Datenverarbeitung, weshalb sie in den, in dieser Arbeit betrachteten Anwendungsfällen, die „dynamischen“ Ansätze auch in bewegten Szenen ausstechen. Letztere eignen sich aufgrund von inhärenten Datenabhängigkeiten nicht für eine parallelisierte Implementierung auf der GPU. Weiterhin bestehen ihre Eingabedaten aus der Menge

der geänderten Voxel, welche in vielen Fällen nur mit dem zusätzlichem Aufwand einer Differenzberechnung zu ermitteln sind.

Die letzte betrachtete Kategorie von EDTs beschreibt Octrees, welche mit Distanzinformationen angereichert sind. Hierzu gehört die Arbeit von Jung [107, 108], die sich auf statische Szenen konzentriert und den Aufbau der Datenstrukturen als einen Offline-Prozess mit beliebiger Laufzeit interpretiert. Die eigentliche Online-Kollisionsprüfung auf *Octree Distance Maps* lässt sich so auf CPUs um bis zu 40% beschleunigen, während sie im Verglich zu regulären Octrees nur ca. 25% mehr Speicher benötigen. In dynamischen Szenen kann die zeitliche Einsparung den Aufwand für den zyklischen Aufbau der Datenstrukturen jedoch nicht aufwiegen, weshalb der Ansatz hier nicht verfolgt wird.

5.6.3. Umsetzung

Unterschiedliche Ansätze zur parallelisierten Berechnung von Distanzfeldern wurden in der Masterarbeit von Christian Jülg [24] realisiert und verglichen. Dieser Abschnitt stützt sich auf seine Ergebnisse.

Kanonische, exakte Euklidische Distanz Transformation

Die einfachste Herangehensweise zur Berechnung der Distanzen in einer Menge Voxel V zu einer Menge an belegten Hindernisvoxeln O ist durch die folgende Minimumssuche beschrieben:

$$\forall v \in V : \underset{o \in O}{\operatorname{argmin}} \parallel v - o \parallel \tag{5.22}$$

Dieser naive Ansatz ist in seiner Berechnungskomplexität $\mathcal{O}(n \cdot m)$ linear abhängig von der Anzahl der Voxel $n = |V|$ und der Anzahl der Hindernisse $m = |O|$, weshalb er lediglich zur Überprüfung anderer Algorithmen herangezogen wurde. Bei einer Szene aus 16 Mio. Voxeln, von welchen $67\,625$ belegt waren, lag der erreichte Berechnungsdurchsatz auf der GPU bei $877\,000\,\mathrm{Voxel/s}$.

Brushfire

Hier wird die Ausbreitung von Wellenfronten imitiert, die von neu entdeckten Hindernissen ausgehen. Jeder Voxel, durch den die Welle läuft, wird mit Distanzinformationen beschrieben. Da dies pro Voxel nur einmal geschieht, ist der Algorithmus bezüglich seiner Schreiboperationen optimal. Um Approximationsfehler ähnlich derer der CDT zu vermeiden, kann beim Schreiben auch der Ursprung der Welle in jedem Voxel gespeichert werden, um daraus bei Bedarf die exakte Distanz berechnen zu können. Der Algorithmus nutzt eine Min-Heap-Datenstruktur, um die Kandidaten, durch welche die Welle läuft, in der Reihenfolge entsprechend ihrer Hindernisdistanz abzuarbeiten. Die Berechnungskomplexität liegt bei $\mathcal{O}(n * logn)$, wobei $n = |V|$ die Menge der Voxel beschreibt. Der logarithmische Anteil rührt von der Suche im Min-Heap und liegt normalerweise

weit unter $logn$, da die Anzahl der Hindernisse weitaus kleiner ist, als die Anzahl der Voxel in der Karte.

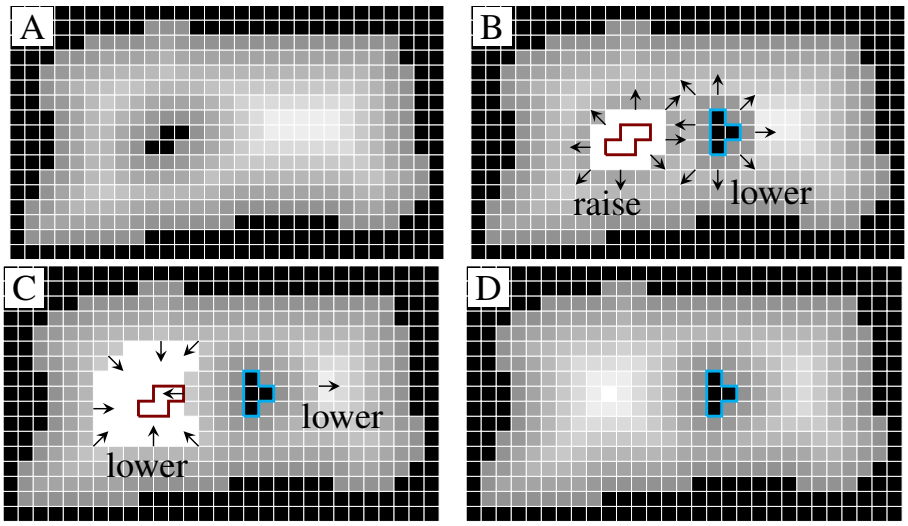


Abb. 5.15.: 2D-Distanzfelder vor (A) und nach einem Update (D). Das neu hinzugekommene, blau umrandete Hindernis löst eine Welle aus, die die Distanzwerte der umliegenden Voxel verringert, während die Welle um das verschwundene, rot umrandete Hindernis zunächst Distanzen löscht, bevor diese von den umliegenden Feldern neu beschrieben werden. Grafik aus [127].

Da die Welle in Voxeln verebbt, die bereits kleinere Distanzen enthalten, muss der Algorithmus beim Einfügen neuer Hindernisse nicht die komplette Karte ablaufen (siehe Abb. 5.15).

Die Berechnungen zur Ausbreitung der Wellen sind in CUDA nur schlecht parallelisierbar, da in jedem Voxel Entscheidungen getroffen werden, die direkt von umliegenden Voxeln abhängig sind, was eine Vielzahl an Synchronisationen zwischen den Threads erfordert. Weiterhin ist der Kontrollfluss einzelner Threads von der Position ihrer bearbeiteten Voxel abhängig, was Laufzeitdivergenzen verursacht und einen kohärenten Speicherzugriff verhindert (vgl. Kapitel 3).

Eine serialisierte Implementierung des Brushfire Algorithmus von Lau und Sprunk ist als Erweiterung des ROS-Paketes OctoMap [104] verfügbar. Hier werden kontinuierlich 3D-Sensordaten in eine Octree-Datenstruktur eingefügt und die Differenzen zwischen zwei Momentaufnahmen des Octrees als Startpunkte der Wellenausbreitung genutzt [128].

Fast Marching Method

In diesem sehr generischer Ansatz der Wellenausbreitung wird die Ausbreitungsgeschwindigkeit F der Wellen als konstant angenommen, weshalb die Ankunftszeit der Welle in einem Voxel direkt in die Distanz zum Wellenursprung umgerechnet werden kann [106, 186]. Vergleichbar mit Brushfire kann auch hier mit einer Menge von Startpunkten gearbeitet werden. Allerdings tragen die Wellen keine Information über die Koordinaten

ihres Ursprungs mit sich, wodurch die berechneten Distanzen Rundungsfehler enthalten und lediglich eine Annäherung der euklidischen Distanz darstellen. Erweiterungen wie FMMHA reduzieren diese Fehler, können ihn aber prinzipbedingt nicht vermeiden [106].

Auch wenn sich eine Parallelisierung aus den im vorherigen Abschnitt genannten Gründen schwierig gestaltet, konnten mehrere Arbeiten einen hohen Performancegewinn durch die Ausführung auf verteilten Supercomputer-Systemen erzielen [122, 206]. So dauert die Berechnung eines Distanzfeldes aus 1024^3 Voxeln auf 65536 Rechenknoten lediglich 0,5 Sekunden (≈ 2 GVoxel/s) bzw. 10 Sekunden auf 256 Rechenknoten (≈ 100 MVoxel/s).

Jump Flood Algorithmus

Dieser Algorithmus zeichnet sich durch seine niedrige Berechnungskomplexität von $\mathcal{O}(N \log n)$ für $N = n^3 = |V|$ Voxel aus [174]. Diese wird erreicht, da die dreidimensionale Datenstruktur nur $\log n$ mal abgetastet werden muss, wie in Abb. 5.16b zu sehen ist. Bei jeder Abtastung gibt ein Voxel seine Hindernisinformationen an bis zu 26 Nachbarvoxel in der Schrittweite k weiter. Da dabei sowohl der Kontrollfluss, als auch die Speicherzugriffsmuster nicht von der Anzahl oder Verteilung der Hindernisse abhängen, eignet sich JFA sehr gut für eine parallele GP-GPU Implementierung.

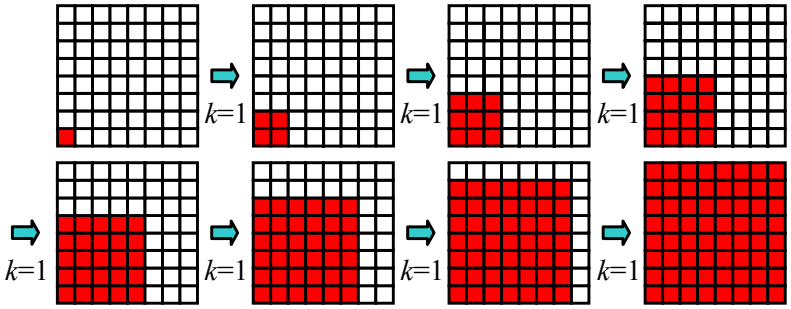
Allerdings liefert der Algorithmus lediglich eine Annäherung an die exakten euklidischen Hindernisabstände und in besonderen Fällen können Hindernisse in sehr spitzen Voronoi-Regionen übersehen werden, da diese durch Diskretisierungsfehler ihren Zusammenhang verlieren [173]. Diese Fehler können durch zusätzliche Abtastschritte minimiert werden. In der JFA Basisversion lässt sich pro Voxel nur die Information eines Hindernisses weitergeben, was zwangsläufig zu einem Informationsverlust führt, wenn mehrere Hindernisse gleichzeitig in der Datenstruktur propagiert werden. Ein naheliegender Kompromiss ist es daher, pro Voxel mehr als nur eine Hindernisinformation zu speichern.

Die JFA Implementierung in dieser Arbeit nutzt Double-Buffering, so dass neue Hindernisdaten geschrieben werden können, während die Distanzberechnung auf den vorhergehenden Daten abläuft.

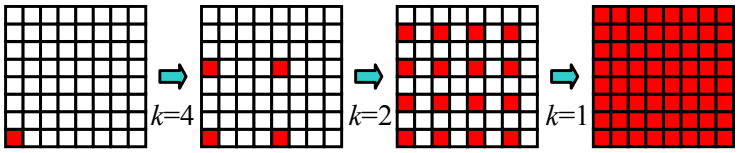
Parallel Banding Algorithmus

Als einer der wenigen parallelen Algorithmen liefert PBA nicht nur Annäherungen, sondern exakte euklidische Distanzinformationen für jeden Voxel. Alle drei Phasen dieses Algorithmus wurden explizit für eine Ausführung auf GP-GPUs entworfen und folgen demselben Grundprinzip: Reduktion der Problemdimensionalität zu Gunsten der Parallelisierbarkeit [54]. Hierfür werden die Eingabedaten in so genannte Bänder aufgeteilt, die in Phase 1 und 2 zunächst unabhängig voneinander bearbeitet werden können, bevor sie in Phase 3 zusammengefasst werden. Der Programmfluss, in dem sich Phase 2 und 3 im dreidimensionalen Fall einmal wiederholen, ist in Abb. 5.17 zu sehen.

Phase 1 verbreitet Hindernisinformationen entlang der Z-Achse, der erste Durchlauf von Phase 2 entlang der Y-Achse. Da die Inhalte der Voxelkarte im Speicher nach ihrer X-



(a) Flood Fill mit fester Schrittgröße $k = 1$



(b) Jump Flooding mit sich halbierender Schrittgröße k

Abb. 5.16.: Vergleich von Strategien zur Informationsverbreitung (aus [174]).

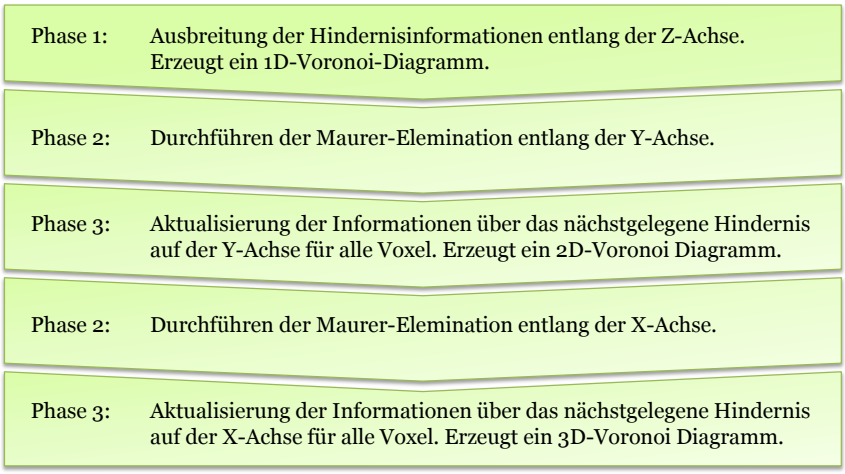
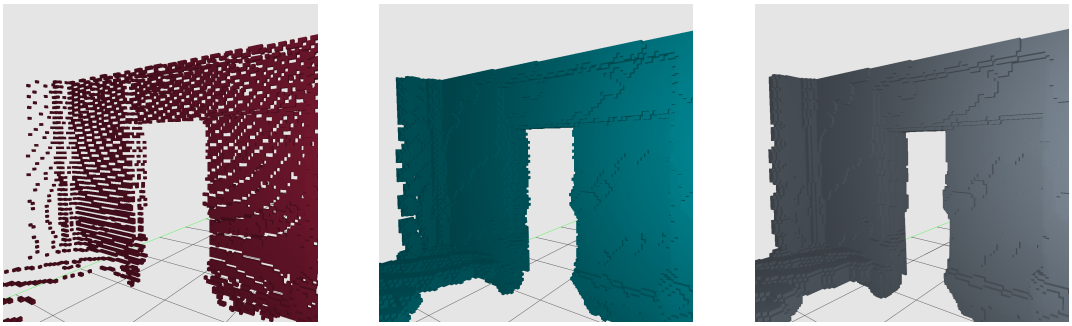


Abb. 5.17.: Vereinfachter Ablauf des Parallel-Banding-Algorithmus.

Koordinate abgelegt sind, werden die Threads eines Warps den Bändern zugeordnet, die Nachbarn bezüglich der X-Achse sind. Somit greifen Threads und Warps auf fortlaufenden Speicher zu, was durch Memory Coalescing für einen optimalen Speicherdurchsatz sorgt. Um diesen Vorteil auch in der zweiten Runde von Phase 2 und 3 nutzen zu können, die entlang anderer Achsen ausgerichtet sind, findet zunächst eine XY-Transformation mittels schnellem geteiltem Speicher statt. Die *Maurer-Elimination* der zweiten Phase beschreibt den Algorithmus zur exakten euklidischen Distanztransformation in linearer Laufzeit aus [141]. Sie wird genutzt, um zu entscheiden, welche Punkte Einfluss auf ein Band haben. Drei Parameter m_1, m_2 und m_3 bestimmen den Parallelisierungsgrad von PBA. Während m_1 und m_2 die Anzahl Bänder in Phase 1 und 2 beschreiben, steht $m3$ für die Anzahl der gleichzeitig auf Voronoi-Zugehörigkeit geprüften Voxel in Phase 3. Der Einfluss der Parameter wird in Abschnitt 8.9, Abb. 8.33 bewertet.

Die Berechnungskomplexität des gesamten Ablaufes ist lediglich linear abhängig von der Anzahl der betrachteten Voxel ($\mathcal{O}(n)$), da PBA die Eigenschaft eindimensionaler Voronoi-Diagramme ausnutzt, die besagt, dass jeder Punkt nur durch einen einzigen anderen Punkt auf derselben Achse beeinflusst wird. Der Ablauf zum Aufbau eines 2D Voronoi Diagramms mittels PBA ist in einem Video[2] zur Publikation von Cao et al. [54] gut nachvollziehbar.



(a) Eingabedaten (b) Voxel bis zu Distanz 3 (c) Voxel bis zu Distanz 5

Abb. 5.18.: Spärlich abgetastete Punktwolke einer Wand mit Tür. Die Distanzberechnung funktioniert robust und schließt Lücken.

Die PBA Implementierung dieser Arbeit folgt in der Programmlogik der Referenzimplementierung von Cao et al. [54]. Inhaltliche Änderungen bereinigen zum einen Programmierfehler, die die Ergebnisse der Phase 2 bei Bändern mit mehr als einem Pixel Breite zerstören konnten. Zum anderen greift die Implementierung dieser Arbeit nicht auf Texturspeicher, sondern auf generellen CUDA Speicher zurück. Darüber hinaus wurde der Code zum besseren Verständnis restrukturiert. Ein technisch interessantes Detail der Umsetzung ist die doppelte Verwendung der Voxel-Datenstruktur: Zunächst werden in ihr alle belegten Hindernisvoxel eingetragen, während am Ende jeder Voxel die Koordinaten seines nächstgelegenen Hindernisses enthält. Zur Laufzeit von PBA können Voxel jedoch temporär auch die Zeiger einer doppelt-verlinkten Liste auf nächstgelegene Hindernisse aufnehmen, da PBA die Kartendimensionen sequentiell bearbeitet. In den bereits ausgewerteten Koordinatenkomponenten (X-Koordinate) eines Voxels kann dann der Index (Y-Koordinate) eines Hindernisvoxels abgelegt werden. Da PBA zeilen- und spaltenweise vorgeht, müssen beide Voxel dieselbe X-Koordinate aufweisen, womit die Position des verlinkten Hindernisvoxels eindeutig bestimmt ist. Sind X- und Y-Komponenten ausgewertet, gilt dieses Vorgehen auch für die Z-Komponente.

5.6.4. Zusammenfassung und Vergleich

Es wurden unterschiedliche Verfahren zur Berechnung euklidischer Distanztransformationen untersucht, umgesetzt und evaluiert. Als erfolgreichster Ansatz ist daraus der Parallel Banding Algorithmus hervorgegangen, mit dem auf modernen GP-GPUs auch für große 3D-Karten mit hoher Wiederholungsrate die Distanzen zu allen in der Karte vorhanden Hindernissen berechnet werden können. Wie in Tab. 5.2 zu sehen, liegt der dabei

[2] `http://www.comp.nus.edu.sg/~tants/pba_files/pba.mov`

Verfahren	Aufwand	Durchsatz [MVoxel/sek]
Kanonisch	$\mathcal{O}(n \cdot m)$	0,9
Brushfire	$\mathcal{O}(n \cdot \log n)$	163,5
FMM	$\mathcal{O}(n \cdot \log n)$	2000,0
JFA	$\mathcal{O}(n \cdot \log n)$	75,0
PBA (original)	$\mathcal{O}(n)$	2093,0
PBA (GPU-Voxels)	$\mathcal{O}(n)$	1325,0

Tab. 5.2.: Vergleich unterschiedlicher Verfahren zur Berechnung von Distanzfeldern.

erreichte Datendurchsatz bei bis zu 1,3 GVoxel pro Sekunde und lässt CPU-basierte Verfahren somit weit hinter sich, auch wenn diese gezielt nur die Kartenteile analysieren, in denen sich geänderte Hindernisse befinden.

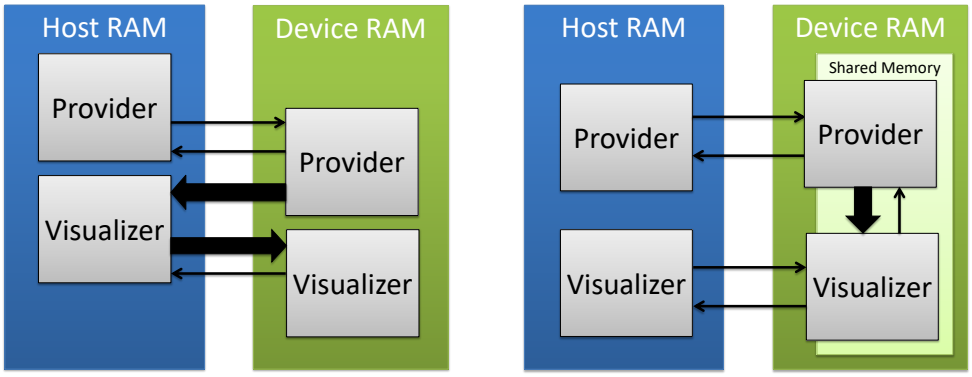
Wie in Abschnitt 5.6.3 erwähnt, erreicht das CPU-basierte dynamische Brushfire-Verfahren auf 3D-Karten aus $12\,\mathrm{MVoxel}$ einen Datendurchsatz, der $163{,}5\,\mathrm{MVoxel/s}$ bei einem „statischen“ Verfahren entspricht. Dabei werden jedoch nur Distanzen bis zu einer Entfernung von 10 Voxeln berechnet. Dagegen erreicht der in GPU-Voxels implementierte PBA bis zu $1{,}3\,\mathrm{GVoxel/s}$ auf wesentlich größeren Karten, mit mehr Hindernissen und bei unbeschränkter Entfernung. Die Geschwindigkeitsvorteile von PBA steigen sogar noch weiter, wenn Brushfire größere Distanzen berechnen müsste.

Gregg und Hazelwood hinterfragen in [91] bei Vergleichen von CPU- und GPU-Laufzeiten den zusätzlichen Aufwand der Datenübertragung zwischen Host und Device. Im Falle von PBA verbleibt die Distanzkarte auf der GPU, so dass lediglich eine Menge von Hindernisdaten zu übertragen ist. Da deren Datenmenge um mehrere Größenordnungen kleiner als die eigentliche Karte ausfällt, ist der Aufwand vernachlässigbar. So beträgt die Zeit der Datenübertragung von $67\,625$ Hindernissen inklusive der Initialisierung einer 256^3 Voxel großen Karte ca. $1{,}8\,\mathrm{ms}$.

Aufbauend auf diesen Ergebnissen wurden zwei sehr unterschiedliche Robotikanwendungen implementiert, die in Abschnitt 8.9 evaluiert werden.

5.7. Visualisierung

Sowohl bei der Entwicklung und Validierung von Algorithmen, als auch während der Ausführung derselben ist es sehr hilfreich, über eine Visualisierung der Ein- und Ausgabedaten zu verfügen. Im Bezug auf GPU-Voxels bedeutet dies, dass ein Nutzer sowohl die Punktwolken der Sensoren, als auch die Voxel (und ihre Eigenschaften) aus mehreren Voxel-Datenstrukturen betrachten kann. Da es sich hierbei um räumliche Daten handelt, ist eine dreidimensionale Darstellung unerlässlich, welche hier wegen der guten Interoperabilität mit CUDA über Open Graphics Library (OpenGL) realisiert wird. Weiterhin ist es wichtig, dem Benutzer eine Kontrolle über die Menge und Art der dargestellten Daten zu erlauben, so dass dieser zu Gunsten der Übersichtlichkeit eine aufgabenspezifische Darstellung umsetzen kann.



(a) Übertragung der Ergebnisse auf den Host und zurück zur GPU

(b) Aufbereitung der Ergebnisse innerhalb des GPU-Speichers.

Abb. 5.19.: Prinzipieller Datenfluss zur Visualisierung von Ergebnissen aus GP-GPU-Berechnungen. Die Dicke der Pfeile symbolisiert die übertragene Datenmenge.

Da Voxel eine sehr einfache Würfelgeometrie aufweisen und ihre Eigenschaften, beziehungsweise Zugehörigkeiten, über Farben darstellbar sind, besteht die Herausforderung nicht in der Art der Darstellung, sondern in der Menge der darzustellenden Voxel und somit in der geforderten Effizienz. Im Gegensatz zu weit verbreiteten Grafik- und Spiele-Engines, wie *OGRE*[3] oder *Cryengine*[4], benötigt die Visualisierung für Voxel keinen komplexen Szenengraphen mit aufwendigen Licht- oder Textureffekten. Erforderlich ist vielmehr der schnelle Umgang mit großen Mengen an dynamischen Daten, die die Voxelszenen ausmachen.

Eine erste Implementierung einer Visualisierung basierte darauf, Kopien aller darzustellenden Informationen, die auf dem *Device* nach einer Voxelberechnung vorlagen, auf den *Host* zu kopieren, dort in OpenGL Strukturen umzuwandeln und diese dann erneut auf die GPU zu transferieren, um sie anzuzeigen (siehe Abb. 5.19a). Wie in Abschnitt 3.2.5 beschrieben, stellt dieses mehrfache Kopieren offenkundig ein Performanceproblem dar.

Daher wurde eine Lösung entwickelt, bei der die Daten auf der GPU verbleiben und dort für ihre Visualisierung aufbereitet werden (siehe Abb. 5.19b). Da CUDA und OpenGL dafür einen gemeinsam genutzten Speicher auf dem Device verwenden, spricht man von einem *Shared Memory*-Ansatz. Die Nutzdaten werden somit zwischen dem *Provider*, also dem eigentlichen Programm und dem *Visualizer* geteilt. Eine Synchronisation geschieht mittels Interprozesskommunikation über den Host. Dieser Ansatz wurde in der Bachelorarbeit [32] von Matthias Wagner erfolgreich umgesetzt und über die Dauer der Dissertation ständig weiterentwickelt. Ein Klassendiagramm der Implementierung findet sich in Abschnitt A.8 in Abb. A.9. Alle Voxelgrafiken in diesem Dokument, mit Ausnahme derer in Unterabschnitt 7.2.2, wurden mit dieser Visualisierung erstellt.

[3] `http://www.ogre3d.org/`
[4] `http://cryengine.com/`

Funktionsumfang

Hier sollen zunächst die wichtigsten Funktionen aufgelistet werden, die umgesetzt wurden: Die Visualisierung ist als eigenständiges Programm lauffähig. Sie kann somit unabhängig von einem Provider gestartet und beendet werden, um keine GPU-Ressourcen zu belegen, wenn keine Visualisierung benötigt wird. Aktualisierungen der Datenstrukturen werden gezielt über Nachrichten des Providers ausgelöst, um unnötige Speicherzugriffe einzusparen. Zusätzlich kann der maximale Speicherverbrauch beschränkt werden, um dem Provider stets eine definierte Laufzeitumgebung zu gewähren.

Alle in GPU-Voxels verfügbaren Datenstrukturen sind (auch mehrfach) darstellbar, wobei verschiedene Voxeltypen bzw. SSV-IDs beliebig wählbare Farben erhalten. Zur Verbesserung der Darstellung kann sich die Farbe einer Datenstruktur entlang einer Achse des Koordinatensystems ändern, um beispielsweise die Höhe eines Voxels über dem Boden besser einzuschätzen. Zur Verbesserung der Tiefenwahrnehmung wird die Szene zusätzlich zu ambientem Licht aus Richtung der Kamera beleuchtet, deren Perspektive per Maus und Tastatur in mehreren Modi anpassbar ist. Neben den Voxeln sind 3D-Punktemengen und geometrische Hilfsobjekte (Kugel, Quader) darstellbar.

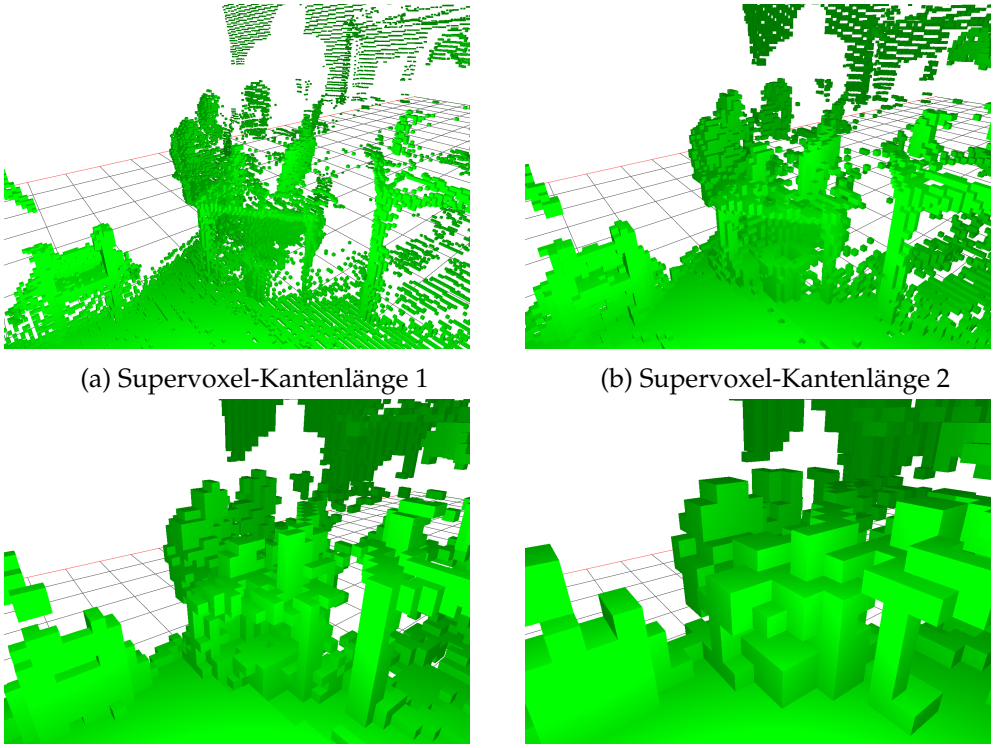
Zu Gunsten der Bildwiederholrate und des Speicherverbrauchs lassen sich Voxel zu Supervoxeln zusammenfassen. Dies reduziert die sichtbare Auflösung und somit die Anzahl der zu zeichnenden Dreiecke. Weiterhin kann der Nutzer das zu zeichnende Volumen einschränken, was es auch ermöglicht, Schnittflächen zu visualisieren. Wird ein Voxel angeklickt, erhält der Benutzer Informationen zum diesem (zugehörige Datenstruktur, Position, Ausdehnung, Status). Zur Verbesserung des Überblicks sind alle dargestellten Informationen separat an und abschaltbar. Alle Einstellungen lassen sich in Konfigurationsdateien speichern.

5.7.1. Geometriegenerierung aus Voxeldaten

Die Darstellung von Geometrien in OpenGL erfolgt über Dreicksnetze, deren Eckpunkte in einem Vertex-Buffer Object (VBO) im GPU-Speicher abgelegt sein müssen. Für ihre Erzeugung wurden Geometrie-Kernel in CUDA implementiert, die für jeden darzustellenden Voxel 36 Eckpunkte (6 Flächen aus je 2 Dreiecken) aus den 3D-Koordinaten der Voxel generieren und im VBO speichern (siehe Abb. 5.21). Zur Steigerung der Effizienz werden alle Voxel derselben Farbe auf einmal gezeichnet und müssen daher aufeinanderfolgend im Speicher abgelegt sein. Da jede Voxel-Datenstruktur im Normalfall nur zu einem gewissen Grad belegt ist, variiert die Größe des benötigten geteilten Speichers. Weiterhin variieren die Anteile der unterschiedlichen Voxel-Bedeutungen (bekannt, unbekannt, SSV-IDs, usw.), die in einzelne Abschnitte des VBOs zu kopieren sind. Im Folgenden wird ein Ansatz entwickelt, der die Voxeldaten effizient in den VBO überträgt.

Speicherabschätzung und Hysterese

Um alle potentiell möglichen, anteiligen Zusammensetzungen von Voxel-Bedeutungen naiv und ohne Vorberechnungen direkt in den VBO kopieren zu können, müsste für jede



(a) Supervoxel-Kantenlänge 1

(b) Supervoxel-Kantenlänge 2

(c) Supervoxel-Kantenlänge 4

(d) Supervoxel-Kantenlänge 8

Abb. 5.20.: Benchmarkszene dargestellt mit unterschiedlicher Supervoxelgröße. Bildraten siehe Abb. 8.11. Die Punktwolke wurde mit einem rotierendem Laserscanner aufgenommen.

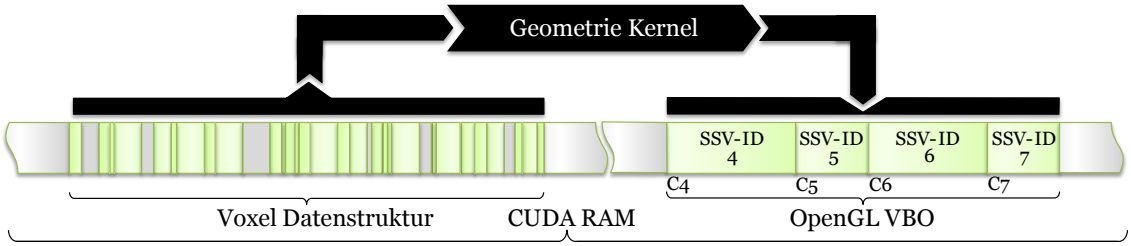


Abb. 5.21.: CUDA Kernel zum Erzeugen von Geometrie-Daten aus Voxeln und anschließender sortierter Ablage im OpenGL Vertex-Buffer Object.

Voxel-Bedeutung der Speicher einer kompletten Datenstruktur vorgehalten werden. Um diese eklatante Überallokation von Speicher zu vermeiden, liegt es nahe, die Voxel-Datenstruktur zunächst zu traversieren, um den Belegtheitsgrad pro Voxel-Bedeutung zu ermitteln. Anschließend lässt sich ein passend dimensionierter VBO inklusive Zeigern auf die Einzelabschnitte anlegen. Während sich der Zählschritt problemlos parallelisieren lässt, benötigt das eigentliche Kopieren der Daten in den Puffer eine Synchronisation zwischen parallel arbeitenden Threads. Diese müssen über geteilte Zählvariablen die Schreibzeiger im Puffer inkrementieren, sobald sie einen belegten Voxel geschrieben haben. Als Alternative zum Engpass einer Synchronisation werden in dieser Arbeit Präfixsummen für jeden Datentyp generiert (siehe Unterabschnitt A.5.1), um die Zieladressen

der Kopieroperationen jedes Threads zu bestimmen.

Um weiterhin das doppelte Traversieren der Datenstruktur (zählen, kopieren) für jedes zu zeichnende Bild zu vermeiden, wurde eine dynamische Speicherverwaltung umgesetzt, die jeweils um ein Bild versetzt auf geänderte Speicheranforderungen reagieren kann. Hierfür wird der VBO initial mit einer gewissen Größe angelegt und gleichmäßig unter allen Voxel-Bedeutungen aufgeteilt. Vor jedem Zeichnen wird er dann, entsprechend der Prefixsummen, mit Voxeldaten befüllt und dabei festgestellt, ob das Fassungsvermögen der Abschnitte den zu zeichnenden Voxelmengen entspricht. Falls nicht, werden die Abschnitte für das nächste Bild um einen definierten prozentualen Anteil vergrößert oder verkleinert, so dass es zu einer möglichst stabilen Hysterese kommt.

Octree

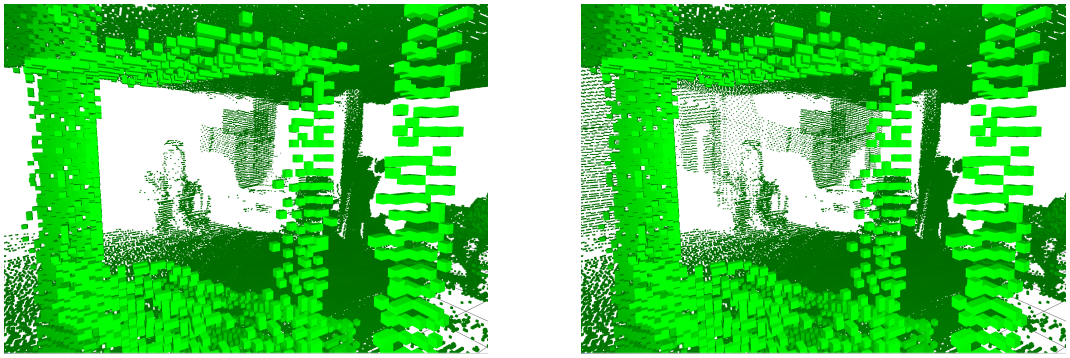
Im Gegensatz zu einer Voxelliste oder Voxelkarte stellt ein Octree eine Datenstruktur mit fragmentiertem Speichermanagement dar. Somit ist es bei einem Octree nicht möglich, dem Kernel zur Geometriegenerierung lediglich die Speicheradresse des Wurzelknotens zu übergeben, da die Visualisierung den Baum dann selbstständig traversieren müsste, um die zu zeichnenden Blattknoten zu finden. Diese Arbeit geschieht daher auf der Seite des Providers, wo eine zusammenhängende Liste aus Würfeln (definiert über kartesische Koordinaten und Seitenlängen) erstellt wird, die dann von der Visualisierung genau wie eine Voxelkarte abgearbeitet wird. Die Datenextraktion ist in Abschnitt 5.5.2 beschrieben.

Reduktion der darzustellenden Daten

Um auch bei umfangreichen Szenen eine hohe Bildrate zu erreichen, wurden zwei Ansätze zur Datenminimierung verfolgt: Zum einen wurde eine Abwandlung des in OpenGL verwendeten *Viewport-Cullings* (Filterung der zu zeichnenden Geometrie durch das Sichtfeld der Kamera) durchgeführt und zum anderen lassen sich Voxelkarten bei Bedarf in einer gröberen Auflösung visualisieren.

Beim **sichtfeldabhängigen Kopieren** wird der Blickwinkel der OpenGL-Kamera berücksichtigt, um nicht die komplette Voxel-Datenstruktur zu bearbeiten, sondern nur den gerade sichtbaren Teil. Aus Praktikabilitätsgründen wird hierfür im Gegensatz zum *Viewport-Culling* kein Kegelstumpf zur Repräsentation des sichtbaren Volumens genutzt, sondern eine Überabschätzung in Form eines Quaders. Zusätzlich kann der darzustellende Ausschnitt künstlich verkleinert werden, um beispielsweise bei der Darstellung geschlossener Räume die Decke zu entfernen, oder bei Distanzkarten einen Schnitt durch die Szene zu ermöglichen. Abb. 5.22 zeigt ein Beispiel, bei dem die Szene in der Tiefe beschnitten wurde.

Bei der **Supervoxel-Methode** werden Voxel zusammengefasst und als größere Würfel gezeichnet (vgl. Abb. 5.20), was die Anzahl der darzustellenden Dreiecke stark reduziert. Dies ist bei der Visualisierung eines Octrees nativ möglich, indem nicht die Blattknoten, sondern die (teilweise) belegten inneren Knoten gezeichnet werden. Bei Voxelkarten



(a) Einschränkung des Sichtbereiches (b) Keine Einschränkung des Sichtbereiches

Abb. 5.22.: Testszene mit und ohne Einschränkung des Sichtbereiches (sichtbar im Hintergrund der linken Bildhälfte).

oder Voxellisten muss die Zusammenfassung beim Iterieren über die Datenstruktur geschehen. Dafür läuft ein Kernel die Daten in einer Schrittweite ab, die der Supervoxelgröße entspricht und prüft dabei die in den Supervoxel fallenden Voxel. Mit dem ersten Fund wird der zugehörige Supervoxel in den VBO kopiert und die Iteration abgebrochen. Auch hierbei stellen sich die anfangs genannten Herausforderungen des a priori unbekannten Speicherverbrauchs und es kann wieder mit oder ohne exakte Vorberechnung gearbeitet werden. Ab einer gewissen Supervoxelgröße bietet es sich an, das Verfahren in zwei Schritte aufzuteilen und die gefundenen Supervoxel nicht direkt in den VBO, sondern in einen Zwischenspeicher zu schreiben. Dieser Puffer kann in seiner maximalen Größe angelegt werden, da er um das $8^{\text{Supervoxelgröße}}$-fache kleiner ist, als die Ausgangskarte. Somit müssen auch keine Schreibzeiger synchronisiert werden. In einem zweiten Schritt durchläuft der Geometrie-Kernel den Zwischenspeicher und überträgt belegte Supervoxel wie gehabt in den VBO.

5.7.2. Umsetzung

Die folgenden Abschnitte beschreiben relevante Details und Besonderheiten der Implementierung der Visualisierung aus GPU-Voxels.

Provider-Visualizer Kommunikation

Die lose Kopplung zwischen der Visualisierung und dem Provider, die es erlaubt, die Darstellung jederzeit starten und stoppen zu können, wurde über eine Interprozesskommunikation mittels eines geteilten Speichersegments im Host-System umgesetzt (*Host Shared Memory*). Darin speichert das Provider-Programm Metadaten über die verfügbaren Voxelkarten, unter anderem die Adressen der Voxeldaten im GPU-Speicher und die Kartengrößen. Die Visualisierung wertet diese Informationen aus und kann gleichzeitig ihre gewünschte Supervoxelauflösungen anfordern oder Semaphoren auf Karten setzen, um eine doppelt gepufferte Synchronisation sicherzustellen. Die Übertragungszeitpunkte der Metadaten unterscheiden sich je nach Datenstruktur: Während eine Voxelkarte statische Ausmaße besitzt und somit immer gleich viele Voxel von der Visualisierung zu

durchlaufen sind, weisen Octree und Voxelliste dynamische Größen auf. Daher müssen ihre Metadaten periodisch aktualisiert werden.

Interaktion

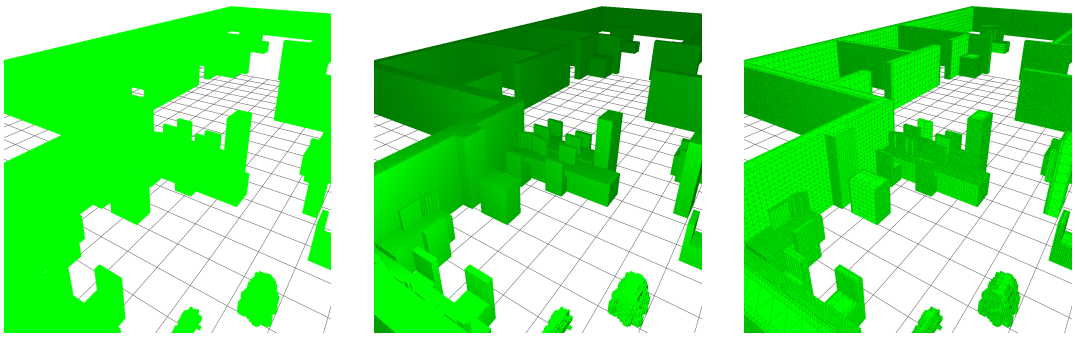
Durch anklicken eines Voxels kann sich der Nutzer detaillierte Information über diesen anzeigen lassen. Hierfür muss zunächst ermittelt werden, welcher Voxel unterhalb der 2D Position des Mauszeigers liegt. Um dabei den Aufwand eines 3D-Raycastings in der Szenengeometrie zu vermeiden, wird auf ein Verfahren zurückgegriffen, bei dem die Szene in unsichtbaren Fehlfarben dargestellt wird, in denen jeder Voxel individuell eingefärbt ist. Ein besonderer OpenGL-Shader bildet dazu nacheinander die X-, Y- und Z-Komponenten aller Voxelkoordinaten auf die 256^3 Farbschattierungen des RGB-Raumes ab und zeichnet die Szene in einen nicht dargestellte Framebuffer. Da die Dreiecke der Szene bereits im VBO vorliegen, ist der zusätzliche Aufwand hierfür minimal. Durch Auslesen der Farbwerte aus drei aufeinander folgenden Bildern (benötigt ca. 100 ms) an der Mausposition können somit direkt die Koordinaten des angeklickten Voxels bestimmt und seine Metainformationen abgerufen werden. Da die Berechnungen nur auf Anforderung ausgeführt werden, beeinträchtigt das Verfahren nicht die allgemeine Performance der Visualisierung.

Farbgebung

Für die Darstellung der Szenen wurde auf ein Lamerbertsches Beleuchtungsmodell mit ambientem Lichtanteil und einer einzelnen punktförmigen Beleuchtungsquelle, die sich mit der Kamera bewegt, zurückgegriffen [187]. Dieses einfache Modell erzeugt bereits einen guten räumlichen Eindruck, der dem Benutzer die Orientierung in einer 3D-Szene erleichtert. Weiterhin lassen sich die Kanten der Voxel einblenden, was bei homogenen Oberflächen eine zusätzliche Struktur und somit eine bessere Interpretierbarkeit ergibt. Um den Eindruck von Höhe oder Tiefe weiter zu verbessern, können innerhalb einer Datenstruktur Farbverläufe entlang einer Achse des globalen Koordinatensystems genutzt werden. Beispiele finden sich in Abb. 5.23.

5.7.3. Zusammenfassung

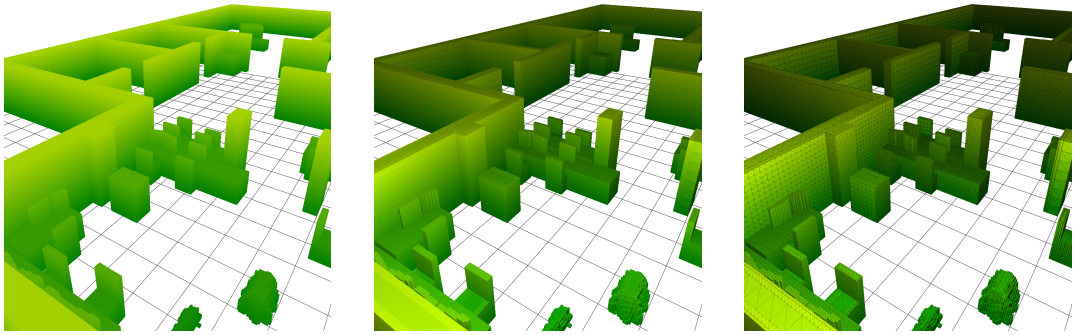
Es wurde eine leichtgewichtige Visualisierung mit umfangreichen Konfigurationsmöglichkeiten entwickelt, die sich an bereits laufende Programme anhängen kann. Über das Prinzip eines geteilten Speichers werden die darzustellenden Daten ausgelesen, als OpenGL Dreiecksnetze aufbereitet und schließlich gerendert. Die umgesetzte Technik vermeidet damit einen Datentransfer zwischen Device und Host nahezu vollständig und kann bis zu 3 Mio. Voxel flüssig und vor allem nahezu latenzfrei anzeigen. Detaillierte Tests zur Bewertung der Leistungsfähigkeit finden sich in Abschnitt 8.5.



(a) Einfarbiges Modell, ambientes Licht

(b) Einfarbiges Modell, ambientes Licht, Punktlicht

(c) Einfarbiges Modell, ambientes Licht, Kanten



(d) Farbverlauf entlang Z-Achse, ambientes Licht

(e) Farbverlauf, ambientes Licht, Punktlicht

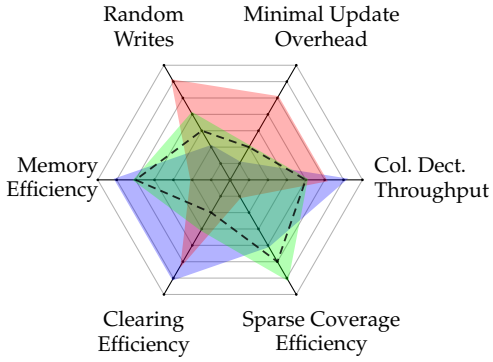
(f) Farbverlauf, ambientes Licht, Punktlicht, Kanten

Abb. 5.23.: Gebäudekarte in unterschiedlichen Darstellungsmodi

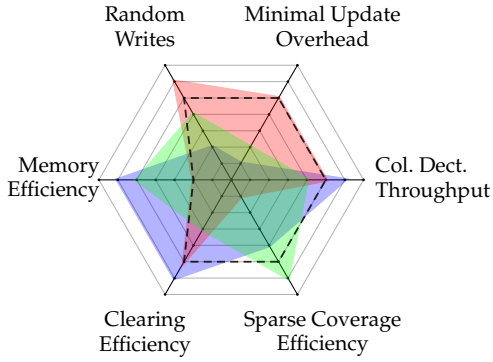
5.8. Fazit

Dieses Kapitel untersuchte sehr unterschiedliche Datenstrukturen, die auf einen parallelisierten Zugriff via GPU ausgelegt sind. Vergleicht man die individuellen Anforderungen einzelner Modelle, die in den Diagrammen aus Abb. 5.2 definiert wurden, mit den Diagrammen in Abb. 5.24, so bestätigt sich, dass Lösungen für eine Kollisionsprüfung in unterschiedlichen Robotikszenarien zur Verfügung stehen. Die Diagramme zeigen, dass Umgebungsinformationen mit hohem Raumvolumen, aber spärlicher Belegtheit sehr gut durch Octrees repräsentierbar sind. Bewegungsprimitive, die eine hohe Datendichte bei gleichzeitiger Lokalität und geringer Änderungsfrequenz aufweisen, entsprechen den Eigenschaften von Voxellisten. Bei der Speicherung von Roboter- und Hindernismodellen bzw. deren Swept-Volumen entscheidet das abzubildende Volumen ob Voxelkarten oder Octrees einzusetzen sind. Somit ist Forschungsfrage 3 zur Eignung von Voxeldatenstrukturen zunächst positiv beantwortet. Details folgen in Kapitel 6 zur Kollisionsdetektion sowie in der praktischen Evaluation in Kapitel 8.

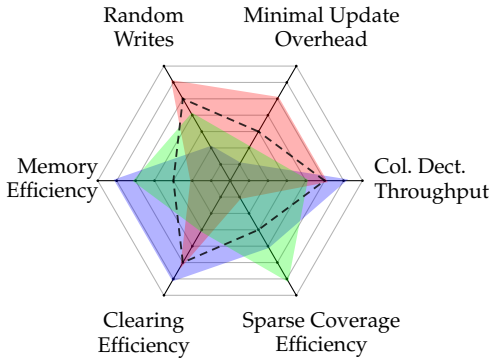
Neben den reinen Datenstrukturen wurden weiterhin onlinefähige Verfahren zur Berechnung von Distanzfeldern vorgestellt und für die GPU optimiert. Außerdem konnte eine leistungsfähige, latenzfreie 3D-Visualisierung entwickelt werden, die es ohne Umwege über den Host erlaubt, alle relevanten Daten der Kollisionserkennung intuitiv und interaktiv darzustellen.



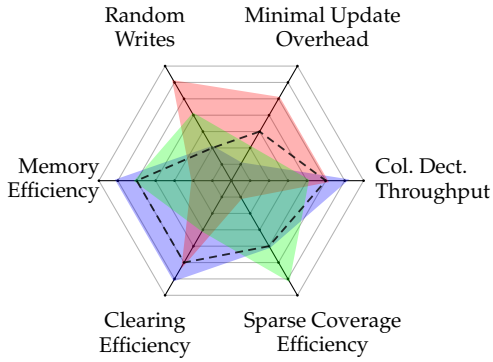
(a) Umgebungskarte



(b) Roboter oder dynamisches Hindernis



(c) Swept-Volumen



(d) Bewegungsprimitive

Abb. 5.24.: Vergleich der Anforderungen zur Verarbeitung von vier unterschiedlichen Datenquellen bei der Planung (schwarz gepunktete Linien) mit den Eigenschaften der implementierten Datenstrukturen: Voxelkarte (rot), Octree (grün) und Voxelliste (blau).

	Octree	**Voxelkarte**	**Voxelliste**
Unterstützt Swept-Volumen	nein	ja	ja
Speicherverbrauch	$\mathcal{O}(n \log n)$	$\mathcal{O}(\mathtt{dim}_x \cdot \mathtt{dim}_y \cdot \mathtt{dim}_z)$	$\mathcal{O}(n)$
Abbildbare geometr. Größe	unbegrenzt	begrenzt	unbegrenzt
Belegte Voxel iterieren	$\mathcal{O}(n \log n)$	$\mathcal{O}(\mathtt{dim}_x \cdot \mathtt{dim}_y \cdot \mathtt{dim}_z)$	$\mathcal{O}(n)$
Wahlfreier Zugriff	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Einfügen neuer Daten	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$

Tab. 5.3.: Vergleich der implementierten Datenstrukturen in GPU-Voxels.

6. Kollisionsdetektion

Dieses Kapitel stellt die zentralen Algorithmen der GPU-Voxels Bibliothek zu Kollisionsdetektion vor. Wie in Definition 1 beschrieben, ermittelt diese, ob sich zwei Entitäten zur selben Zeit am selben Ort befinden und somit eine Kollision vorliegt. Dafür werden zunächst die wichtigsten Kategorien von Verfahren zur Kollisionsdetektion vorgestellt und verglichen, bevor dann die implementierten voxelbasierten Ansätze beschrieben werden.

Da die Möglichkeiten der Kollisionsdetektion aber über die rein binäre Frage hinausgehen, muss bei der Auswahl eines Verfahrens zunächst entschieden werden, welche Informationen als Resultate erwartet werden und welche Eingabedaten vorliegen: Sind lediglich Paare von Entitäten gegeneinander zu prüfen (*Narrow Phase*) oder liegen Mengen von Entitäten (*Broad Phase*) vor? Ist die generelle binäre Information *Kollision / keine Kollision* ausreichend oder muss ermittelt werden, welche Entität mit welcher anderen Entität in Kollision liegt? Sollen Kontakte, Durchdringung oder vollständige Umschließung identifizierbar sein? Handelt es sich um dynamische oder statische Szenen? Kann von einem vollständigen Umweltwissen ausgegangen werden? Ist bei Kollisionsfreiheit die minimale Distanz zwischen Entitäten von Bedeutung?

Diese Fragen bestimmen zunächst die verwendbare Repräsentation von Entitäten und somit das Ego- und Umweltmodell. Aufbauend auf den bereits vorgestellten, diskretisierenden Datenstrukturen sollen in diesem Kapitel unterschiedliche Algorithmen zur Kollisionsdetektion und -vermeidung entworfen werden. Dabei liegt der Fokus darauf, die individuellen Eigenschaften der Strukturen gewinnbringend zu nutzen und die Parallelisierbarkeit bzw. den Datendurchsatz auf der GP-GPU zu maximieren.

Im Folgenden wird vom dreidimensionalen Fall ausgegangen, da mobile Manipulationsroboter, wie sie in Abb. 6.1 zu sehen sind, aufgrund ihrer Arme bzw. zusätzlicher Körperachsen eine variabel ausladende, geometrische Struktur aufweisen. Zahlreiche bestehende, performante Ansätze projizieren hingegen Hindernisse und Roboter auf eine oder mehrere Ebenen, um die Kollisionsprüfung auf ein zweidimensionales / 2,5D Problem zu reduzieren [102]. Solche Verfahren sollen hier genau so wenig betrachtet werden, wie Ansätze zur Auswertung von nicht-rigiden Objekten.

6.1. Taxonomie Kollisionserkennungsverfahren

Um unterschiedliche Technologien besser beurteilen zu können, stellt die folgende Taxonomie vier etablierte Verfahren zur Kollisionsdetektion vor. Dabei lehnt sie sich an die Taxonomie unterschiedlicher Umweltmodelle aus Abschnitt 4.2 an und beantwortet die eingangs gestellten Fragen.

(a) Roboter IMMP und HoLLiE (b) Valeri Roboter

Abb. 6.1.: Roboter mit großem Arbeitsraum und variabler Geometrie durch ausladende Kinematik.

Schnitt von Oberflächennetzen Verfahren dieser Kategorie weisen die weiteste Verbreitung auf, da sie auf Dreiecksnetzen arbeiten, die auch in der Computergrafik genutzt werden. Die Dreiecke zweier Netze können mit einfachen Gleichungssystemen auf Überschneidungen geprüft werden [156], was sich sehr gut auf GP-GPUs parallelisieren und beschleunigen lässt. Die Repräsentation erlaubt es, auch mehrere Entitäten zu identifizieren, auf Kollisionen zu überwachen, oder Distanzen zwischen ihnen zu bestimmen. Allerdings skaliert die Laufzeit dieser Verfahren direkt mit der Anzahl der Dreiecke und ist somit von der Objektanzahl, der Approximationsgüte und der Komplexität der Objektgeometrien abhängig. Da keine Volumen, sondern Oberflächen betrachtet werden, können komplette Durchdringungen nicht ohne zusätzlichen Aufwand bestimmt werden.

Da die Modellierung der Oberflächen mit beliebiger Genauigkeit umsetzbar ist, lassen sich bei der Kollisionsprüfung auch Kontaktflächen bestimmenn, weshalb diese Verfahren in der Physiksimulation bevorzugt sind. Hier kommen bspw. parametrisierte CAD-Modelle aus NURBS (Non-Uniform Rational B-Splines) in Frage, die jedoch nur sehr aufwendig berechnet und auf Kollisionen geprüft werden können.

Hierarchien aus Hüllkörpern / Bounding Volume Hierachies (BVH) Um nicht in jedem Kollisionsberechnungsschritt alle Entitäten (bspw. Dreiecke eines Netzes) gegeneinander überprüfen zu müssen, aber auch um bei komplexen Modellierungen nicht alle Details in Betracht ziehen zu müssen, können die Entitäten in BVHs eingeschlossen werden. Wie bereits in Unterabschnitt 4.2.3 beschrieben, werden Modelle und ihre Bestandteile hierbei rekursiv in einfach zu berechnende Hüllkörper (Geometrische Primitive oder einfache konvexe Formen) eingeschlossen. So bildet sich ein Baum, an dessen Wurzel ein Hüllkörper liegt, der alle Entitäten beinhaltet. In Richtung seiner Blätter schließen die Hüllkörper dann immer kleinere Subvolumen ein. Arbeitet der Kollisionsalgorithmus entlang dieser Baumstruktur, können bei der Kollisionsprüfung sehr effizient Entitäten oder Teile von ihnen von einer genaueren Betrachtung ausgeschlossen werden, falls ihre Hüllkörper nicht in Kol-

lision liegen [178]. Bei guter Unterteilbarkeit lässt sich so der durchschnittliche Aufwand der Kollisionsprüfung eines Objektes gegen n andere Objekten von $\mathcal{O}(n)$ auf $\mathcal{O}(\log n)$ Prüfungen reduzieren [76]. Neben der Kollision lässt sich auch die Distanz analytisch effizient berechnen und auf der GP-GPUs parallelisieren, wie in *gProximity* gezeigt [130].

Auch bei BVHs besteht die Herausforderung nicht in der Kollisionsprüfung, sondern in der automatischen Aufteilung der Modelle in geometrische Primitive oder Hüllkörper. Diese stellt einen initialen Aufwand dar, weshalb diese Verfahren bevorzugt mit vorausberechneten Modellen und nicht auf Livedaten eingesetzt werden.

Diskretisierende Modellierung Bereits frühe Ansätze zur Kollisionsdetektion nutzten eine diskretisierende Datenstruktur, die sowohl die Umwelt, als auch das Egomodell des Roboters enthielt [87]. War eine Zelle (Voxel) der Datenstruktur gleichzeitig von zwei Entitäten belegt, lag eine Kollision zwischen diesen vor. Ein hoch optimiertes Beispiel dieser Verfahrensklasse ist der Voxmap Pointshell Algorithmus aus [177], der ursprünglich für haptisches Rendering entwickelt wurde [142]. Auch in der Robotik wird die sequentielle CPU-Implementierung des ROS Collider Paketes [95] häufig genutzt, die in Kapitel 8 für Vergleiche herangezogen wird. Ausschlaggebend für den Berechnungsdurchsatz und den Speicherverbrauch diskretisierender Verfahren sind die genutzten Datenstrukturen, weshalb diese im vorhergehenden Kapitel vorgestellt wurden. Einige der umgesetzten Verfahren besitzen eine konstante Laufzeit, die unabhängig vom Belegtheitsgrad der Eingabedaten sind. Ausschlaggebend für den Speicherbedarf und die Berechnungsdauer der meisten Algorithmen ist dagegen die Anzahl der verwendeten Zellen. Da sie vom Raumvolumen und der Zellgröße bestimmt wird, sollte beides adäquat zum Problem gewählt werden. Je größer jedoch das Volumen einer Zelle ist, desto größer fällt die Überabschätzung von Hindernissen aus, da auch ein kleines Hindernis mindestens eine komplette Zelle als belegt markieren. Bei der Bestimmung von Traversierbarkeit muss somit die Kantenlänge kleiner als der halbe Durchmesser der kleinsten noch zu passierenden Öffnung sein. Anderenfalls könnten Durchgänge durch Diskretisierungsfehler als verschlossen erscheinen.

Da die Zellen diskretisierender Modelle auch eine Belegtheitswahrscheinlichkeit speichern können, lassen sich je nach Anwendungsszenario nicht nur binäre, sondern probabilistische Entscheidungen treffen, wenn teilweise belegte Zellen betrachtet werden. Somit sind konservative / pessimistische bzw. opportunistische / optimistische Entscheidungen bei der Kollisionsprüfung und Planung umsetzbar.

Ein Nachteil der Diskretisierung sind fehlende Objektoberflächen und damit auch fehlende Normalen. Somit können keine exakten Kontaktflächen und deren physikalische Interaktion berechnet werden.

Punktwolken Klassifikation / Probabilistische Verfahren Letztendlich existieren auch Verfahren, die direkt auf Punktwolken arbeiten. Hierfür kann die Punktwolke in einem kd-Baum repräsentiert werden, womit die Kollisionsprüfung zu einem Suchproblem wird [181]. Eine weitere Alternative bieten probabilistische Verfahren oder Ansätze aus dem Bereich des maschinellen Lernens, bei der die Verschränkung zweier Punktemengen als Klassifikationsproblem betrachtet wird [157]. In beiden

Fällen ist eine semantische Annotation zur Identifikation von einzelnen Entitäten nur schwer möglich.

Für eine detailliertere Übersicht über weitere Verfahren (vor allem aus dem Gebiet der interaktiven Computergrafik) sei auf das umfangreiche Nachschlagewerk *Real-time collision detection* von Ericson verwiesen [76].

Auswahl des geeignetsten Verfahrens

Im Hinblick auf das Ziel dieser Arbeit, eine hochparallele Kollisionsprüfung von detaillierten Egomodellen mit Punktwolken einer nur teilweise bekannten Umwelt zu erlauben, können die eingangs gestellten Fragen wie folgt beantwortet werden: Da die Umweltdaten nur in Ausnahmen segmentiert vorliegen, müssen alle bekannten Entitäten bei der Kollisionsprüfung berücksichtigt werden (*Broad Phase*), wobei für eine zielgerichtete Planung dennoch erwünscht ist, festzustellen, welche Entitäten des Egomodells kollidieren. Irrelevant ist dagegen, ob es sich um Kontakte, Durchdringung oder vollständige Umschließung handelt. Für viele Planungsalgorithmen sind jedoch Informationen über die schwere der Kollision (also die Überschneidungstiefe) bzw. die minimale Distanz zwischen nicht kollidierenden Entitäten eine hilfreiche Information, die die Kollisionserkennung zur Verfügung stellen sollte. Wie im späteren Kapitel zur Bewegungsplanung ersichtlich wird, werden meist nicht nur einzelne Posen eines Roboters auf Kollisionen hin überprüft, sondern ganze Bewegungsabläufe. Hierfür bieten sich die in Abschnitt 4.5 eingeführten Swept-Volu-men an, die folglich von der Kollisionsdetektion effizient verarbeitet werden müssen. Diese Antworten führen zu folgender Aussage:

Feststellung 9. Die Anforderungen einer Kollisionsprüfung von Punktwolkenrepräsentationen werden von Verfahren, die auf einer diskretisierenden Modellierung arbeiten, am besten erfüllt. Zusätzlich bieten sie optimale Voraussetzungen für eine parallelisierte Implementierung, wie die folgenden Abschnitte beschreiben.

Die aktuell relevantesten Arbeiten zur Kollisionsdetektion zwischen Punktwolken, mit denen sich GPU-Voxels vergleichen kann, sind ein CPU-basierter *k*d-Tree-Ansatz von Schauer et at. [181] und ein GPU-basierter Voxelansatz von Bedkowski et al. [47]. Ein Benchmark von Schauer et al. [180] liefert einen Verglich zwischen diesen. In dessen Bewertung werden die beiden Verfahren als ähnlich performant eingestuft, wobei spezifische Vorteile hauptsächlich von unterschiedlichen Punktedichten der Eingabedaten abhängen. Eine detaillierte Gegenüberstellung zu dieser Arbeit findet sich in Kapitel 8.

6.2. Voxelbasierte Kollisionsdetektion

Abb. 1.3 aus der Einleitung visualisiert die Schnittmengenbildung zwischen Umwelt- und Egomodell und bringt damit das relevanteste Ziel der voxelbasierten Kollisionsprüfung auf den Punkt. Es gilt, die Paare von Voxeln aus zwei Datenstrukturen zu finden, die am selben Ort liegen und eine gewisse Belegtheitseigenschaft aufweisen. Für diese

Schnittmengenbildung $\cap$ zwischen den Datenstrukturen M_a und M_b muss im Allgemeinen über eine der Strukturen iteriert werden, und für jeden belegten Voxel V_a der Voxel V_b an derselben Position in der zweiten Struktur M_b nachgeschlagen werden. Ist auch der Voxel der zweiten Karte belegt, wird er zur Menge der kollidierenden Voxel hinzugefügt:

$$V_a \,\&\, V_b := \begin{cases} 1 & : (\blacksquare(V_a) = 1) \wedge (\blacksquare(V_b) = 1) \\ 0 & : \text{sonst} \end{cases} \tag{6.1}$$

$$M_a \,\cap\, M_b := \bigvee V_a \;\mid\; \big(coord(V_a) = coord(V_b)\big) \wedge \big(V_a \& V_b = 1\big)\,,\;\; \forall V_a \in M_a\,, \forall V_b \in M_b \tag{6.2}$$

$$coll(M_a, M_b) := \begin{cases} 0 & : M_a \,\cap\, M_b = \emptyset \\ 1 & : \text{sonst} \end{cases} \tag{6.3}$$

Um diese Arbeit parallel ausführen zu können, geschieht die Iteration sowie die Interpretation der besuchten Voxel im Code eines CUDA-Kernels. Da als Datenstruktur für die beiden zu prüfenden Modelle alle drei Implementierungen (Voxelkarte, Voxelliste und Octree) aus dem vorherigen Kapitel in Frage kommen, müssen dementsprechend mehrere $\cap$-Operatoren in Form von Kerneln vorhanden sein. Diese Kernel bilden den Kern der GPU-Voxels Bibliothek, weshalb sie darauf ausgerichtet sind, die Eigenheiten der Datenstrukturen berücksichtigen, um den bestmöglichen Datendurchsatz zu erreichen. Da weiterhin verschiedene Voxeltypen Verwendung finden, muss der $\&$-Operator die Belegtheit mittels $\blacksquare(V)$ individuell interpretieren. Um in der Software nicht das Kreuzprodukt aller Karten- und Voxeltypen als Programmcode umsetzen zu müssen, wird jedem Kollisions-Kernel ein vorkonfiguriertes `Collider`-Objekt mit übergeben, welches den $\&$-Operator entsprechend der eingesetzten Typen implementiert. Neben den, als Templateparameter übergebenen, beteiligten Voxeltypen erhält der `Collider` weitere Parameter (bspw. Schwellwerte oder Bitvektoren), um der $\blacksquare(V)$-Operator voll zu definieren und um kollidierende Voxel in einer der beiden Datenstrukturen zu kennzeichnen. Der Kernel selbst muss somit zwar spezifisch für die Iteration über die zu prüfenden Datenstrukturen sein, bleibt jedoch Voxeltyp agnostisch, da er mit jedem potentiell kollidierenden Voxelpaar den `Collider` aufruft.

Die Besonderheiten der Kollisions-Kernel bei allen möglichen Kombinationen der Datenstrukturen werden im folgenden beschrieben. Voraussetzung für alle Algorithmen ist, dass beide Datenstrukturen achsparallel ausgerichtet sind und sich überlappen.

6.2.1. Semantik der Kollisionsprüfung

Durch die Überlagerung zweier Voxel-Datenstrukturen können je nach verwendetem Voxeltyp unterschiedliche semantische Informationen abgeleitet werden:

Kollisionsprüfung mit probabilistischen und deterministischen Voxeln

Bestehen beide Strukturen aus probabilistischen und / oder deterministischen Voxeln, so kann entweder lediglich die binäre Aussage *Kollision / keine Kollision* getroffen werden, oder aber durch das Zählen der in Kollision liegenden Voxel der Grad der Überschneidung und somit die Schwere der Kollision bestimmt werden. Im ersten Fall können die Berechnungen mit der Detektion der ersten Kollision abgebrochen werden (*Lazy Evaluation*), was zu großen Zeitersparnissen führen kann. Im zweiten Fall müssen alle Voxel erschöpfend bearbeitet werden und zusätzlicher Aufwand zur Zusammenfassung der parallel ermittelten Kollisionen erbracht werden. Da diese Zusammenfassung von der Implementierung der verwendeten Datenstruktur abhängig ist, wird in den folgenden Abschnitten detaillierter auf sie eingegangen.

Kollisionsprüfung mit Bitvector-Voxeln

Ist eine Datenstruktur aus Bitvektor-Voxeln an der Kollisionsprüfung beteiligt, müssen hierfür, je nach Semantik der Karteninformationen, unterschiedliche Verfahren eingesetzt werden.

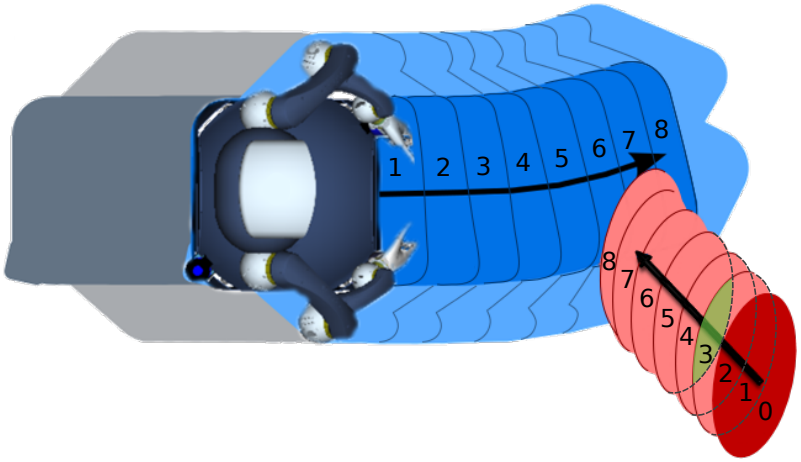


Abb. 6.2.: Dynamsche Beispielszene zur Verdeutlichung von SSV-IDs: Die Subvolumen des geplanten Roboterpfades sind in blau dargestellt, der grau markierte Bereich wurde bereits abgefahren. In rot ist die Bewegung eines Hindernisses quer zur Fahrtrichtung des Roboters gezeichnet. Darin ist ein Teilvolumen in grün hervorgehoben, welches zu drei Zeitschritten (3, 4, 5) belegt ist. Es bildet die Grundlage für das Beispiel aus Abb. 6.3. Roboter und Hindernis befinden sich anfangs in SSV-IDs 0.

Stellen die SSV-IDs unterschiedliche Entitäten dar, bspw. die einzelnen beweglichen Segmente eines Roboters, so ist es für die Detektion einer Kollision irrelevant, welches Segment in Kollision liegt. Dennoch kann es hilfreich sein, Informationen über die betroffenen Segmente zu erhalten, um gezielter zu reagieren. In diesem Fall ist es nicht ausreichend, den $\blacksquare(V)$ Operator zu nutzen. Vielmehr ist das *Belegt*-Bit der Voxel auszuwerten,

und bei einer Kollision alle anderen SSV-ID-Bits in einem Ergebnis-Bitvektor zu verodern:

$$\begin{aligned} V_1 \,\&\, V_2 &:= \Psi_{V_1,1} \wedge \Psi_{V_2,1} \\ \text{Ergebnis-Bitvektor} &:= \Psi_{V_1,i} \vee \Psi_{V_2,i}\,,\ 4 \leq i \leq 254 \end{aligned} \tag{6.4}$$

Repräsentieren die SSV-IDs jedoch Zeitstempel der Belegtheit, so müssen dieselben Bits in beiden Voxeln gesetzt sein, um die Voraussetzungen für eine Kollision zu erfüllen (selber Ort und selber Zeitpunkt).

$$\begin{aligned} V_1 \,\&\, V_2 &:= \bigvee_{n=4}^{254} \Psi_{V_1,n} \wedge \Psi_{V_2,n} \\ \text{Ergebnis-Bitvektor} &:= \Psi_{V_1} \wedge \Psi_{V_2} \end{aligned} \tag{6.5}$$

Ein Beispiel ist in Abb. 6.4 gezeigt. Der Ausschnitt aus den Bitvektoren von Roboter und Hindernis zeigt eine Überschneidung in den Bits 18, 19 und 20. Das Ergebnis einer Kollisionsprüfung mittels &-Operator ist im Ergebnis-Bitvektor UND eingetragen.

Wie im Falle der Kollisionsprädiktion aus Abschnitt 4.6 gezeigt, kann es nötig sein, das Kollisionskriterium aufzuweichen, um ein Zeitfenster aus mehreren angrenzenden Bits zu überprüfen, damit so Unsicherheiten ausgeglichen werden können. Das Ergebnis einer &-Operation mit einer Fensterbreite von $k = 5$ Bits ist in Abb. 6.4 in Zeile UND_5 gezeigt. Hierbei werden auch die Zeitpunkte O_{n-k} bis O_{n+k} kurz vor bzw. nach dem eigentlich untersuchten Moment R_n berücksichtigt und somit eine größere Toleranz erreicht. Nach einer weiteren &-Operation mit dem Bitvektor des Roboter-Swept-Volumens steht dann das Ergebnis bereit. Um die Bitvektoren zweiter Voxel A und B mit diesem zusätzlichen Sicherheitsfenster auf gemeinsame gesetzte SSV-IDs zu prüfen, muss einer der beiden Vektoren in 1-Bit Schritten positiv wie negativ um die Fensterbreite verschoben werden. Dabei stellt sich das Problem, dass in CUDA zwar ein Bitshifting-Operator zur Verfügung steht, der größte verfügbare Datentyp jedoch nur 64 Bit aufweist. Somit ist es nicht möglich, einen kompletten Bitvektor aus 256 Bits auf einmal zu shiften. Die Kernel-Implementierung bedient sich daher der Konzepte des `std::bitset`[1] aus der C++-Standardbibliothek und arbeitet intern mit einem 64 Bit breiten Puffer, der Wortweise zusammengesetzt wird. Folglich ist in Abschnitten von je einem Byte aus Vektor B vorzugehen, die innerhalb eines 8 Byte Puffers verschoben werden. Hieraus ergibt sich eine maximale Fensterbreite von 21 Bit. Weiterhin ist darauf zu achten, die niederwertigsten vier Bits und das hochwertigste Bit der Eingabedaten auszublenden, da diese keine SSV-IDs darstellen. Das Ergebnis der Bitweisen &-Operation jedes Abschnitts muss letztendlich an der korrekten Position mit den Bits des Ausgabevektors verodert werden. Der Pseudocode des implementierten Kernel ist in Algorithmus 8 gelistet. Die Parallelisierung erfolgt auf Voxelebene.

Eine weitere Funktion des `Colliders` ist es, durch die Definition einer Bitmaske gezielt Kollisionen zwischen definierten SSV-IDs zu ignorieren. Diese werden dann weder gezählt, noch führen sie zu einer Kollision im Gesamtresultat. Dies ist bspw. bei der Ausblendung von Eigenkollisionen nützlich (vgl. Unterabschnitt 4.4.3).

[1] `https://gcc.gnu.org/` bzw. `www.cplusplus.com/reference/bitset/bitset/`

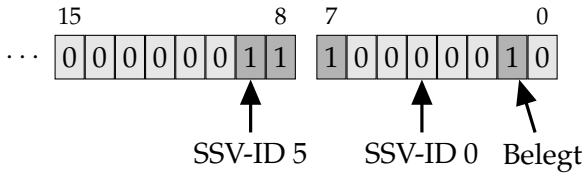


Abb. 6.3.: Die ersten beiden Bytes aus dem Bitvektors eines Voxels, der zu drei Zeitschritten belegt ist (vgl. grünes Volumen aus Abb. 6.2). Alle nicht dargestellten Bits des Vektors sind nicht gesetzt.

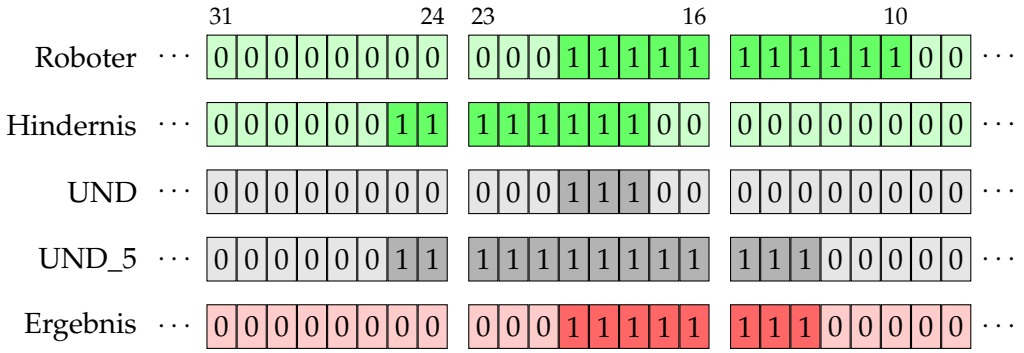


Abb. 6.4.: Beispiel der Kollisionserkennung zwischen dem Swept-Volumen Bitvektor eines Roboters und eines Hindernisses aus [27]. Die Bitvektoren von Roboter und Hindernis sind in grün dargestellt, das Ergebnis der Kollisionsprüfung in rot.

Letztendlich können alle Kernel die Kollisionsinformationen in eine der beiden Datenstrukturen zurückschreiben. Zur Kennzeichnung kollidierender Voxel steht im Bitvektor explizit das Bit 2 zur Verfügung, welches wiederum von der Visualisierung ausgewertet wird. Da hierfür jedoch ein schreibender Speicherzugriff nötig ist, sollte diese Möglichkeit zugunsten der Laufzeit nur bei Bedarf aktiviert werden.

6.2.2. Kollisionsprüfung Voxelkarte ∩ Voxelkarte

Nachdem die Kollisionsprüfung auf Voxelebene definiert wurde, soll sie nun auf der Ebene der Datenstrukturen erläutert werden. Der geradlinigste Fall ist dabei die Iteration über zwei Voxelkarten mit denselben Dimensionen. Hierbei kann in einer einzigen Grid-Stride-Loop und bei maximaler Parallelisierung mit q verfügbaren Threads über die Anzahl s der Voxel pro Karte iteriert werden. Der Laufindex wird zu den Basis-Speicheradressen der beiden Karten addiert, um jeweils auf Voxel an denselben Koordinaten zuzugreifen, und diese dem `Collider` zuzuführen. Da der Zugriffsoperator eine konstante Laufzeit von $\mathcal{O}(1)$ aufweist, gilt für den Aufwand $A_{\text{Voxelkarte}}$ dieser Kollisionsprüfung:

$$A_{\text{Voxelkarte}}(s) = 2 \cdot (s/p) \tag{6.6}$$

Der Aufwand ist also insbesondere nicht vom Belegtheitsgrad der Karten abhängig, womit harte Laufzeitgarantien gegeben werden können. Da die Daten beider Karten linear im globalen Speicher abgelegt sind, werden pro Warp zwar durchschnittlich zwei Speicherzugriffe benötigt, diese profitieren jedoch maximal von Memory Coalescing.

Nicht nur bei der Verwendung von Karten unterschiedlicher Größe kann es jedoch nötig sein, Voxelkarten vor der Kollisionsprüfung gegeneinander zu verschieben (keine Verdrehung). Dies ist durch die Addition eines Versatzes auf eine der Basisadressen, wie in Unterabschnitt 5.3.1 beschrieben, einfach möglich. Somit können bspw. unterschiedliche Platzierungen eines Roboters in der Umwelt sehr effizient realisiert werden.

Für das Zählen der Kollisionen während einer Iteration verfügt jeder Thread eines Blocks über eine eigene Zählvariable, die er mit jedem Treffer inkrementiert. Somit sind Ressourcenkonflikte ausgeschlossen. Da diese Zähler in einem Feld im geteilten Speicher liegen, können sie am Ende der Laufzeit per Reduktion zu einer blockweiten Summe aufaddiert werden. Weiterhin liegt im globalen Speicher ein Feld aus Zählvariablen pro Block vor, in die ein einzelner Thread des Blocks das Ergebnis der Reduktion schreibt. Dieses Feld wird nach der Ausführung des Kernels auf den Host kopiert. Dort lassen sich in einer kurzen Schleife die Zähler aller Blöcke aufaddieren, womit das Gesamtergebnis feststeht.

Dasselbe Reduktionsschema wird auch verwendet, um im Falle von Bitvektor-Voxeln kollidierende SSV-IDs zu bestimmen. Statt der Addition kommt dabei der $\|$ -Operator für die Reduktion zum Einsatz und es werden Bitfelder anstatt Zählvariablen genutzt.

6.2.3. Kollisionsprüfung Voxelliste ∩ Voxelliste

Voxellisten richten sich in ihrem Adressierungsschema nach einer virtuellen Voxelkarte bzw. einem Octree. Daher ist die Voraussetzung für die Kollisionsprüfung zwischen zwei Listen, dass ihre entsprechenden virtuellen Karten dieselben Dimensionen bzw. die selbe Maximaltiefe aufweisen. Ist dies sichergestellt, kann eine einfache Kollisionsprüfung, bei der lediglich die Voxel zu zählen sind, die in beiden Listen vorhanden sind, durch eine Suche in den Zeigerlisten realisiert werden.

Soll dagegen jedoch ein `Collider` genutzt werden, ist zunächst die Menge der kollidierenden Elemente zu bestimmen. Unter der Voraussetzung, dass die Listen sortiert vorliegen und keine doppelten Voxel aufweisen, können dafür folgende Thurst-Operationen eingesetzt werden: Die eigentliche Kollisionsprüfung besteht aus einer Suche der Einträge der kürzeren Eingabeliste innerhalb der längeren Liste. Jeder Fund einer Kollision wird in temporären binären Masken eingetragen, mit deren Hilfe die betroffenen Einträge beider Eingabelisten in Ergebnislisten kopiert werden. Über diese kann dann im letzten Schritt sehr effizient iteriert werden, um entweder die Menge der Kollisionen zu zählen, oder den `Collider` anzuwenden um ein gewünschtes Resultat zu bestimmen. Da alle Teilalgorithmen aus linear ablaufenden Primitiven der Parallelverarbeitung aufgebaut sind, ist Thrust in der Lage, auf Basis der zu verarbeitenden Datengrößen eine optimale GPU-Auslastung zu erreichen.

Für die Umsetzung der Greifplanung aus Unterabschnitt 7.2.7 wurde für die Listen noch eine besondere Auswertung implementiert, die die Anzahl an Kollisionen getrennt pro SSV-ID ermittelt. Da für die GPU-seitige Umsetzung dieser Funktion eine umfangreiche Datenreduktion der 250 geteilten Zählvariablen benötigt würde, ist es in diesem Fall performanter, die Ergebnislisten auf den Host zu kopieren und dort sequentiell auszuwerten. Dies ist ein gutes Beispiel für die heterogene Parallelverarbeitung mittels GPU und CPU.

6.2.4. Kollisionsprüfung Voxelliste ∩ Voxelkarte

Wie bei der Prüfung zwischen zwei Listen, ist auch hier die Voraussetzung, dass die virtuelle Voxelkarte der Liste dieselben Dimensionen wie die zu prüfende Datenstruktur aufweist. Somit kann in einer Grid-Stride-Loop über die in der Liste gespeicherten Voxeladressen iteriert werden. Durch die Addition der Karten-Basisadresse können die entsprechenden Voxel direkt in der Karte abgerufen werden, um mit dem `Collider` evaluiert zu werden. Weiterhin kann auch hier ein Versatz eingebracht werden, um eine virtuelle Translation der Liste zu erzeugen. Diese Kombination aus Datenstrukturen ist sehr effizient, da das durchlaufen einer Voxelliste den linearem Aufwand $\mathcal{O}(n)$ erzeugt, während der Zugriff auf einzelne Voxel in einer Voxelkarte mit $\mathcal{O}(1)$ möglich ist. Somit bestimmt sich der Gesamtaufwand rein über die Länge der Liste, die in vielen Anwendungen gering gehalten werden kann.

Eine Kommutation der Datenstrukturen (also das Iterieren über die Voxelkarte und das Prüfen der entsprechenden Voxel in der Liste) ist in keinem Falle sinnvoll, da Voxellisten nicht über einen Operator zum wahlfreien Zugriff verfügen und daher bei jedem Lesen eine Suche nötig wäre.

6.2.5. Kollisionsprüfung Octree ∩ Octree

Voraussetzung für den Schnitt von zwei Bäumen ist, dass diese dasselbe Raumvolumen beschreiben und eine gleiche maximale Tiefe aufweisen. Ist dies gegeben, können die Octrees simultan traversiert werden, um diejenigen Knoten zu finden, die in beiden belegt sind. Hierfür bietet sich die Nutzung des bereits vorgestellten Lastausgleiches an, wobei die Arbeitselemente nun nicht mehr nur Knoten aus einem Baum, sondern aus beiden Bäumen enthalten. Zusätzlich verfügen sie über zwei Boolsche Variablen, die angeben, ob die Tiefensuche die Blattknoten eines Baumes erreicht hat. Wie im ursprünglichen Algorithmus wird wieder die Ebene der zu bearbeitenden Knoten benötigt und alle Arbeitsstapel müssen die Invariante aus Gleichung 5.21 erfüllen. Der um die Kollisionsprüfung erweiterte Algorithmus 6 findet sich in Anhang A. Er bearbeitet die Elemente eines Stapels pro Thread in einer Schleife und verteilt diese neu, wenn eine Lastungleichheit vorliegt. Dafür werden zunächst Knotenpaare für jeden Thread aus dem globalen Speicher in den schnellen, geteilten Speicher kopiert und auf Kollision überprüft. Hierbei kann eine maximale Abstiegstiefe berücksichtigt werden, die die Granularität der Kollisionsbestimmung erhöht und ihre Berechnungszeit verringert. Wurde auf der vorgegebenen Maximaltiefe ein kollidierender Knoten gefunden, steigt der Algorithmus noch eine Ebene weiter ab, um die Zuverlässigkeit der Kollisionsaussage zu verbessern. Bestätigt sich die Kollision, werden bei der Aufsummierung der sich schneidenden Voxel alle Blattknoten als kollidierend angenommen, also die Anzahl der kleinsten Kindvoxel gezählt. Ihre Menge muss jedoch nicht berechnet werden, sondern kann in einer Tabelle im sehr schnellen konstanten Speicher nachgeschlagen werden. Ein Beispiel dazu ist in Abb. 6.5 zu sehen.

Eine konservative Abschätzung des Rechenaufwands des vorgestellten Verfahrens zeigt, dass dieser im schlechtmöglichsten Fall lediglich ca. 14% höher ausfällt, als der Aufwand einer linearen Kollisionsprüfung zwischen zwei Voxelkarten: Wie oben in Gleichung 6.6 gezeigt, benötigt die Voxelkarten-Kollision einen Aufwand von $A_{\text{Voxelkarte}}(s) = 2 \cdot (s/q)$

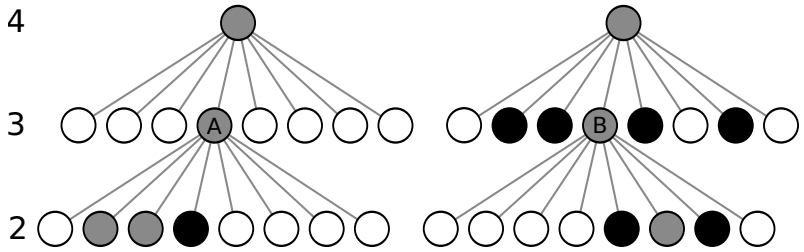


Abb. 6.5.: Erkannte Kollision bei Abbruch der Prüfung auf Ebene 3. Durch den Abstieg auf Ebene 2 kann jedoch die Kollisionsfreiheit erkannt werden. Voxel: belegt (schwarz), frei (weiß), teils belegt (grau).

Der Aufwand zur Traversierung zweier Octrees hängt hingegen von vielen Faktoren ab, unter anderem dem Belegtheitsgrad und -muster, da diese die Parallelisierbarkeit der Tiefensuche bestimmen. Daher soll hier der Fall mit dem höchstmöglichen, aber eindeutig zu berechnenden Aufwand untersucht werden, in dem jeder innere Knoten des Baumes über Kindknoten verfügt und die Tiefensuche somit immer bis zur Blattebene auszuführen ist:

$$A_{Octree}(s) = 2 \cdot \left(k + \sum_{i=1}^{\log_8 \frac{s}{q}} 8^i \right) = 2 \cdot \left(k + 8^1 + 8^2 + \ldots + \frac{s}{q} \right) \tag{6.7}$$
$$\text{für} \quad s > q,\ s = 8^{\#\text{Ebenen}},\ q = 8^k,\ k, \#\text{Ebenen} \in \mathbb{N}$$

Die ausgeschriebene Summe in der Gleichung setzt sich aus den Aufwänden der einzelnen Ebenen zusammen. Für die oberen k Ebenen sind genau k Rechenschritte erforderlich, da diese Ebenen weniger Knoten aufweisen, als Threads q verfügbar sind. Somit kann jede Ebene voll parallelisiert in je einem Schritt bearbeitet werden. Unterhalb der Ebene k verachtfacht sich der Aufwand pro Ebene für jeden Thread ($8^{\#\text{Ebene}}$), bis auf der Blattebene s Voxel durch q Threads zu verarbeiten sind. In der Summenschreibweise wird die Anzahl der Ebenen zwischen k und der Blattebene mittels $\log_8 \frac{s}{q} - 1$ bestimmt. Diese Summe kann umgeformt und mittels der geometrischen Reihenentwicklung zu Gleichung 6.8 abgeschätzt werden (siehe Gleichung A.6 in Abschnitt A.7).

$$2 \left(k - 1 + \frac{s}{q} \cdot \frac{8}{7} \right) \tag{6.8}$$

Vernachlässigt man hier $k = \log_8 q$ zeigt sich im Vergleich mit dem Voxelkartenaufwand aus Gleichung 6.6 ein Mehraufwand von $1/7 \approx 14\%$. Dies beschreibt jedoch den schlechtmöglichsten Fall, bei voller Belegung. Bei der Verwendung mit realistischen Umweltdaten, bei denen weit weniger als die Hälfte des Raumvolumens belegt ist, ist der Octree für den lesenden Zugriff immer performanter als eine Voxelkarte, da der Aufwand im Schnitt logarithmisch mit der Menge an Daten abfällt.

6.2.6. Kollisionsprüfung Octree ∩ Voxelliste

Da eine Voxelliste lediglich belegte Voxel speichert, ist es am effizientesten, die Einträge der Liste parallel abzuarbeiten, wobei jeder Thread eine Tiefensuche im Octree durchführt, um den Voxel aus der Liste zu suchen. Ein inverses Vorgehen wäre auch aufgrund der gewählten Implementierung der Voxelliste nicht zielführend, da diese keinen wahlfreien Zugriff auf bestimmte Voxel unterstützt. Ist die Suche erfolgreich, führt der `Collider` die nutzdatenspezifische Kollisionsprüfung aus. Da aufeinanderfolgende Voxel der Liste meist eine örtliche Lokalität aufweisen, verfolgen viele der parallelen Tiefensuchen einen zumindest teilweise übereinstimmenden Pfad, wie in Abb. 6.6 zu sehen ist. Der Grad der Überlappung, d.h. die kleinste gemeinsame Ebene im Baum, kann mittels weniger Bit-Operationen aus den Morton-Codes der gesuchten Voxel bestimmt werden (siehe Abschnitt A.3). Durch einen Vorverarbeitungsschritt, der die Ähnlichkeit mehrerer Voxel vor der Tiefensuche überprüft und Abstiege zusammenfasst, kann viel Rechenzeit gespart werden, da individuelle Threads erst ab der divergierenden Ebene starten müssen. Daher verbindet auch diese Kombination effektiv die Vorteile beider Datenstrukturen.

6.2.7. Kollisionsprüfung Octree ∩ Voxelkarte

Wie aus den Beschreibungen der unterschiedlichen Datenstrukturen im vorigen Kapitel und den Grafiken aus Abb. 5.24 hervorgeht, weisen Octree und Voxelkarte unterschiedliche Vorteile auf, die bei einer Kollisionsprüfung gewinnbringend kombiniert werden können. Dies ist gut nachvollziehbar, wenn die Kommutation der Datenstrukturen betrachtet wird, die in Abb. 6.7 dargestellt ist und die im Folgenden erläutert wird.

Im ersten Fall wird die Voxelkarte maximal parallelisiert abgelaufen. Jeder dabei gefundene belegte Voxel b muss dann durch eine Tiefensuche im Octree nachgeschlagen werden, was abhängig von dessen Belegtheitsgrad unterschiedlichen Aufwand verursacht. Da konsekutive Threads örtlich nah aneinanderliegende Voxel bearbeiten und sich diese Lokalität auch auf den Octree überträgt, verfolgen viele Threads bei ihrem Abstieg im Baum zunächst den gleichen Pfad. Wie auch bei der Voxelliste kann dies erkannt und zur Effizienzsteigerung ausgenutzt werden. Der durchschnittliche, theoretische Aufwand ohne Parallelisierung liegt somit bei $\mathcal{O}(m+(b \log n))$ (wenn b der insgesamt m Voxel in der Voxelkarte belegt sind und der Baum n Voxel beinhaltet).

Dennoch ist diese Methode dem kommutierten Fall meist unterlegen: Wird als primäre Struktur der Octree traversiert, können größere freie Regionen effizient ausgelassen und nur belegte Voxel gezielt in der Voxelkarte in $\mathcal{O}(1)$ nachgeschlagen werden. Dazu kommt wiederum die parallele Tiefensuche mit Lastausgleich zum Einsatz, wobei deren Arbeitselemente nun, neben der bearbeiteten Ebene und den Zeigern auf Kindknoten, auch deren 3D-Koordinaten beinhalten. Diese werden für die Adressierung der Voxelkarte benötigt und lassen sich beim Abstieg im Baum einfach generieren. Die Prüfung auf Belegtheit in der Voxelkarte ist dann voll parallel möglich. Eine effiziente Implementierung dieses Algorithmus muss jedoch einige Besonderheiten aufweisen, um Lastungleichheit zu vermeiden und ist daher wesentlich komplexer als die Implementierung des ersten Falls. So eignet sich die Bearbeitung ohne Arbeitsstapel für die beiden untersten Baumebenen besser, als das differenzierte Bearbeiten von belegten Teilbäumen. Weiterhin werden ab einer

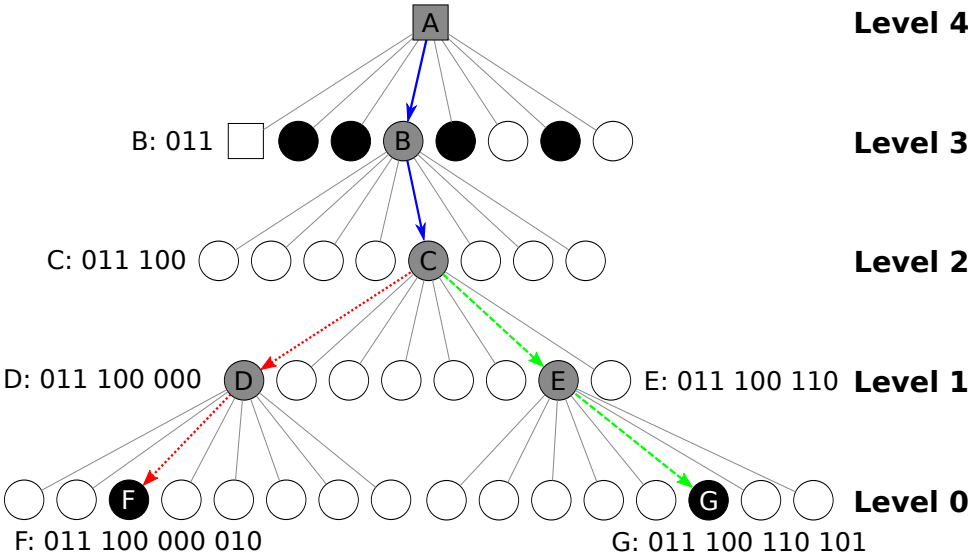
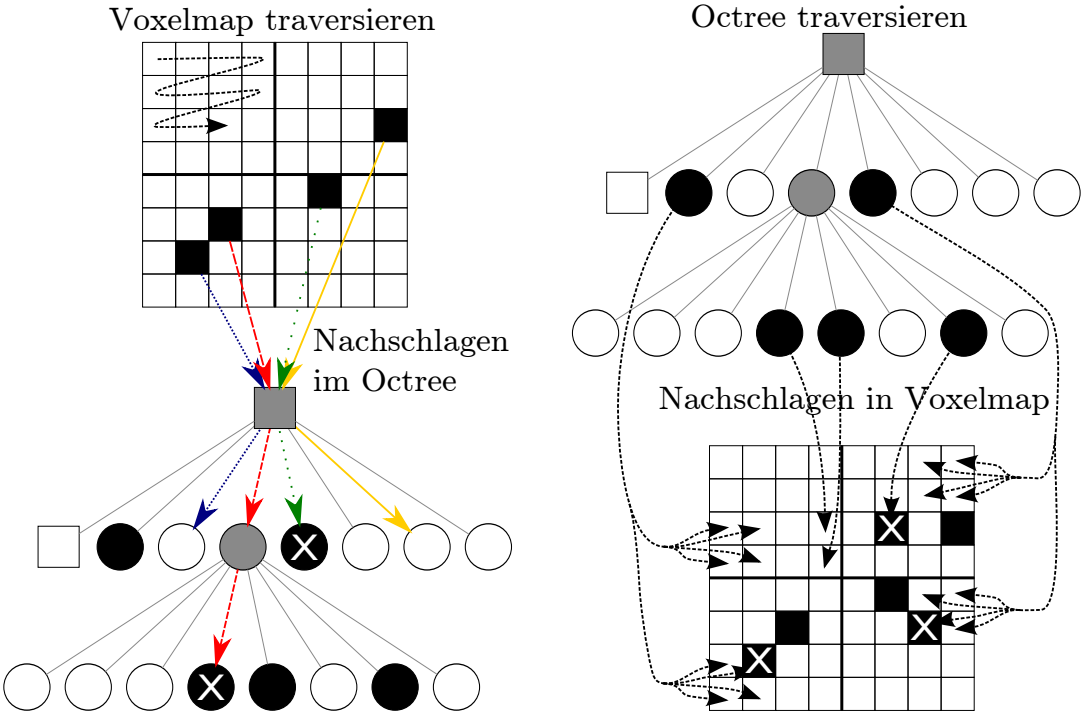


Abb. 6.6.: Gemeinsamer Pfad der Tiefensuche.



(a) Travesieren der kompletten Voxelkarte und mehrfaches Travesieren des Octrees für jeden belegten Voxel.

(b) Einmaliges Traversieren des Octree und gezieltes Nachschlagen der belegten Blattknoten in Voxelkarte.

Abb. 6.7.: Effizienzsteigerung durch Kommutation der Datenstrukturen bei einer Kollisionsprüfung (X markiert eine erkannte Kollision).

gewissen Ebene auch vollständig belegte Knoten zu Gunsten der gleichmäßigen Lastverteilung weiter traversiert (wenn auch mit einem statischen Zeiger auf das Elternelement), obwohl das Verarbeitungsergebnis bereits feststeht. Die dadurch vermiedene Divergenz im Programmfluss verbessert dennoch die Gesamtlaufzeit (vgl. Unterabschnitt 3.2.1).

Die Abwägung, welche der beiden Methoden eingesetzt wird, hängt von der Problemgröße und der Belegtheit der Datenstrukturen ab. Ist der Octree sehr ungleichmäßig belegt, macht der Mehraufwand des Lastausgleiches im zweiten Verfahren die Vorteile zunichte.

Da sich das repräsentierte Volumen eines Octrees und einer Voxelkarte unterscheiden, ist zunächst ihre Überlappung zu berechnen. Dafür werden die minimalen und maximalen Koordinaten der Octree-Teilbäume herangezogen, die sehr effizient ermittelbar sind (siehe Abschnitt A.3). Mit diesen Koordinaten kann die Überlappung und die vollständige Umfassung eines Teilbaumes von einer Voxelkarte direkt bestimmt werden.

6.2.8. Kollisionsprüfung Distanzkarte ∩ Voxelliste

Distanzkarten verhalten sich hinsichtlich des Zugriffs auf einzelne Voxel wie Voxelkarten, weshalb sie analog eingesetzt werden können. Allerdings wäre die feingranulare Abtastung der Distanzkarte ein suboptimales Vorgehen, da insbesondere im Freiraum eine einzelne Abfrage bereits implizit Informationen über ein kugelförmiges Volumen im Umfeld liefert und somit alle anderen Abfragen innerhalb der Kugel redundant wären. Daher ist es eine effizientere Vorgehensweise, Geometrien durch Kugeln mit konstantem Radius zu approximieren und lediglich deren Mittelpunkte bei der Kollisionsprüfung in der Distanzkarte nachzuschlagen (ähnlich der Arbeit von Greenspan [90]). Da es sich bei dieser Darstellung um sehr spärliche Daten handelt, bietet sich die Verwendung einer Voxelliste an, um die Mittelpunkte zu speichern. Über einen `Collider`, der als zusätzlichen Parameter den verwendeten Kugelradius erhält, können diese mit der Distanzkarte auf Kollision geprüft werden.

Für die Versuche in Abschnitt 8.9 wurde jedoch letztendlich gar keine eigene Datenstruktur zur Modellierung des Roboters verwendet, da die komplette Flugdrohne durch eine einzelne Kugel repräsentiert werden konnte.

Da der Aufbau und die Aktualisierung einer Distanzkarte aus Sensordaten, verglichen mit den anderen Datenstrukturen, verhältnismäßig lange dauert und Lese- / Schreibzugriffe nicht parallel stattfinden können, bremst dies die Reaktivität der Datenstruktur empfindlich aus. Daher bietet sich die Verwendung zweier Distanzkarten zur Umsetzung eines doppelten Puffers an, um eine Parallelisierung über den abwechselnden zeitversetzten Zugriff zu erreichen.

6.3. Fazit

Bedingt durch die Vielzahl an Datenstrukturen mussten bei der Kollisionsdetektion spezifische Vorgehen implementiert werden, um die Zugriffseigenschaften der beteiligten

Strukturen je nach Kombination zum Vorteil zu nutzen. Bei der Verwendung der Algorithmen ist die folgende Aussage zu beachten:

Feststellung 10. Datenstrukturen bei der Kollisionsprüfung verhalten sich nicht kommutativ. Eine verwendete Ausgangsdatenstruktur sollte möglichst effizient iterierbar sein und wenige Voxel enthalten, während die zweite Struktur einen effizienten, wahlfreien Zugriff erlauben sollte, da in ihr nachgeschlagen wird.

In der praktischen Anwendung finden jedoch meist konträre Datenstrukturen Verwendung, die den Anforderungen der zu repräsentierenden Daten gerecht werden und die sich gleichzeitig bei der Kollisionsprüfung optimal ergänzen. Daher sind die Laufzeiten im Vergleich zur Verwendung gleichartiger Strukturen fast immer kürzer, wie auch die Evaluation in Kapitel 8 belegt. Somit ist auch der zweite Teil der Forschungsfrage 3 zu den Eigenschaften der Datenstrukturen positiv beantwortet.

Durch das umgesetzte Konzept des `Colliders`, welches von den verfügbaren Voxeltypen abstrahiert, potenziert sich die Anzahl der zu implementierende Fälle dennoch nicht. Bei der praktischen Verwendung von GPU-Voxels hilft eine eingängige, templatebasierte API, die die verfügbaren Kollisionsfunktionen pro Datenstrukturpaar übersichtlich gestaltet.

7. Bewegungsplanung

In diesem Kapitel sollen nun, aufbauend auf den Datenstrukturen und der Kollisionsdetektion, Verfahren vorgestellt werden, die es einem Roboter ermöglichen, seine Bewegungen zu planen. Wie in Definition 3 beschrieben, ist die Prämisse dabei, kollisionsfrei von einem Ausgangszustand zu einem gegebene Zielzustand zu gelangen. Dafür müssen als weitere Eingabedaten ein Umweltmodell und ein Egomodell zur Verfügung stehen. Auf Basis dieser Daten kann dann ein Plan in Form einer Trajektorie aus Zwischenzuständen generiert werden, entlang derer sich der Roboter dann bewegt. Dabei kann es nötig sein, Zwischenzustände einzunehmen, die den Roboter zunächst noch weiter von seinem Ziel entfernen, aber global gesehen zur Lösung des Problems dienen. Die Verfahren der Planung können daher sehr komplex ausfallen (Klasse der NP-harten [167] und auch NP-kompletten Probleme [53]). Weiterhin entsteht insbesondere in dynamischen Umgebungen die Problematik, dass erstellte Pläne bereits veraltet und nicht mehr kollisionsfrei sind, noch bevor sie zur Ausführung kommen.

Im Gegensatz zur zeitintensiven Planung stehen reaktive Verfahren, die den großen Rechenaufwand vermeiden, indem sie zielorientiert, aber lediglich lokal arbeiten, um einen Roboter von Hindernissen fernhalten. Sie können schneller auf Änderungen in der Umwelt eingehen, sind jedoch anfällig dafür, an lokalen Minima zu scheitern, weshalb sie nur für kürzere zeitliche und örtliche Horizonte ausgelegt sind. Oftmals werden daher beide Verfahrensklassen zu hybriden Systemen kombiniert. Die vorliegende Arbeit hat hingegen das Ziel, die Planungzeit so weit zu verkürzen, dass auf lokale, reaktive Komponenten verzichtet werden kann. Möglich wird dies, indem die zeitintensivste Komponente der Planung, nämlich die Kollisionsprüfung, durch die Parallelisierung auf der GPU massiv beschleunigt wird.

Dieses Kapitel untersucht daher zunächst unterschiedliche Planungsverfahren, um die Kandidaten zu ermitteln, die am meisten von einer GPU-beschleunigten Kollisionsprüfung profitieren. Ihr Einsatz wird dann anhand konkreter Probleme detaillierter beschrieben. Eine symbolischen Planung, die in der Steuerungshierarchie oberhalb der Bewegungsplanung steht, wird in dieser Arbeit nicht näher behandelt.

7.1. Grundlagen

Zu Beginn sollen einige Grundlagen definiert werden, um den Einsatz schneller Verfahren der Kollisionsdetektion in der Planung zu motivieren.

Fast alle Planungsalgorithmen bestehen aus den folgenden vier Schritten, die teilweise wiederholt ausgeführt werden: Zunächst ist eine Transformation aus dem Arbeitsraum in den Konfigurationsraum des Roboters zu definieren. Daraufhin wird eine Diskretisierung des Konfigurationsraumes durchgeführt (explorativ oder analytisch) und

zwischen den diskretisierten Zuständen ein Graph aufgespannt. Liegen Start- und Zielzustand im Graphen, kann eine Verbindung zwischen ihnen gesucht werden. Während dieses Prozesses muss die Validität einzelner Zustände bzw. der Zustandsübergänge entlang der Graphenkanten evaluiert werden. Hierunter fallen geometrische Vorgaben wie Kollisionsfreiheit und Gelenkwinkelbeschränkungen, aber auch dynamische Einschränkungen.

7.1.1. Arbeitsraum, Konfigurationsraum und Planungsraum

In der Bewegungsplanung unterscheidet man zwischen zwei Räumen unterschiedlicher Dimensionalität, in denen die Bewegung eines Roboters beschrieben werden können: Dem *Arbeitsraum* $\mathcal{S}$ und dem *Konfigurationsraum* $\mathcal{C}$. Der Arbeitsraum definiert sich über die physische Umgebung und wird über das Umweltmodell (siehe Kapitel 4) repräsentiert. Dagegen beschreibt der Konfigurationsraum die Posen des Roboters, weshalb er durch die beweglichen Freiheitsgrade (Degree of freedom (DOF)) aufgespannt wird. Im Allgemeinen ist der Konfigurationsraum surjektiv, aber nicht injektiv zum Arbeitsraum. Somit ist die Abbildung einer Roboterkonfiguration, die durch ein Tupel aus Gelenkwinkeln beschrieben ist, in den Arbeitsraum durch die so genannte *Direkt Kinematik* eindeutig und trivial möglich. Das Ergebnis ist die Pose des Endeffektors im Raum. Der inverse Fall hingegen, also die Berechnung der Gelenkwinkel aus einer gegebenen Endeffektorpose mittels der *Inversen Kinematik*, ist nur in Sonderfällen eindeutig und direkt berechenbar. Da in den meisten Fällen der *Planungsraum* der *Konfigurationsraum* ist, beeinflusst diese Mehrdeutigkeit die Planbarkeit bzw. die Planungszeit von Bewegungsproblemen. Hierzu seien drei Beispiele gegeben:

- Bei einer rotationssymmetrischen mobilen Roboterplattform, welche ihre Position in einer Ebene steuern kann, entspricht der Konfigurationsraum dem Arbeitsraum, da in beiden Räumen die Freiheitsgrade durch die Position der Plattform in der Ebene ($SE(2)$: x, y, θ) bestimmt werden.

- Bei einem stationären Roboterarm mit sechs steuerbaren Gelenken $j_i, i \in [1..6]$, der sein Werkzeug frei im Raum bewegen kann ($SE(3)$: $x, y, z, \alpha, \beta, \gamma$), ist zwar die Anzahl, nicht aber die Bedeutung der Dimensionen beider Räume gleich. Je nach Aufbau seiner kinematischen Kette ist solch ein Roboter in der Lage, denselben Punkt im Raum bei vorgegebener Orientierung mit acht unterschiedlichen Gelenkkonfigurationen zu erreichen. In Sonderfällen, in denen zwei Freiheitsgrade achsparallel liegen (sog. Singularitäten), sogar mit unendlich vielen verschiedenen Stellungen.

- Auch der mobile Serviceroboter HoLLiE (siehe Abb. 6.1a) bewegt sich im selben sechsdimensionalen Arbeitsraum, verfügt aber (unter Vernachlässigung der Freiheitsgrade der Hände) über wesentlich mehr Freiheitsgrade: 3 DOF der mobilen Plattform + 2 x 6 DOF der Arme + 2 DOF des Körpers = 17 DOF. Hierbei ist eine Mehrdeutigkeit offensichtlich, da der Roboter mit seinem TCP eine Pose auf sehr unterschiedliche Arten erreichen kann.

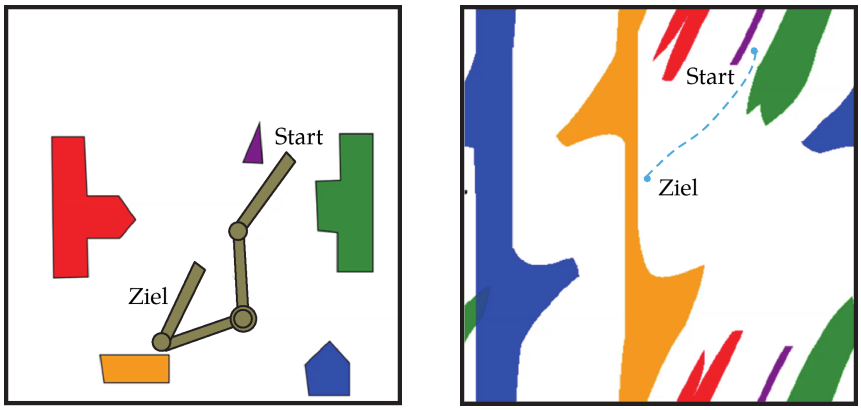
Die Vorgabe aller Freiheitsgrade positioniert einen Roboter also eindeutig in seinem Arbeitsraum und bestimmt damit, ob eine Kollision mit der Umgebung vorliegt. Entsprechend wird der *Freiraum* als die Teilmenge des Konfigurationsraums definiert, in der

keine Kollisionen auftreten:

$$\mathcal{C}_{\text{frei}} \subseteq \mathcal{C} \quad \text{wobei} \quad \mathcal{C}_{\text{frei}} = \mathcal{C} \setminus \mathcal{C}_{\text{Hindernis}} \tag{7.1}$$

Da jedoch, wie in den Beispielen gezeigt, im Allgemeinen keine eindeutige Abbildung von Arbeitsraum in Konfigurationsraum existiert, kann auch der Freiraum nicht direkt in den Konfigurationsraum projiziert werden, selbst wenn die Umgebung komplett bekannt ist. Ein Ansatz zur Erstellung einer Freiraumkarte im Konfigurationsraum wäre es, diesen schrittweise abzutasten und jeden Abtastschritt im Arbeitsraum auf Kollisionen zu prüfen. Dies ist jedoch nicht tragbar: Wenn beispielsweise der Konfigurationsraum des 6-DOF-Roboterarmes ($-180\,\text{deg} \leq j_i \leq 180\,\text{deg}$) in 1 Grad Schritten diskretisiert würde, ergäbe dies $360^6 \approx 2,177 * 10^{15}$ zu prüfende Konfigurationen, die bei einem dynamischen Arbeitsraum für jede Posenänderung erneut geprüft werden müssten.

Dies stellt ein Problem für die Bewegungsplanung dar, da vieles darauf hindeutet, dass die Berechnungskomplexität exponentiell mit der Dimension des Konfigurationsraumes wächst [168]. Da die Art des Roboters und die Anzahl seiner Freiheitsgrade in dieser Arbeit nicht weiter eingeschränkt werden soll, werden unterschiedliche Herangehensweisen an diese Problematik in späteren Abschnitten beschrieben.



(a) Arbeitsraum (b) Konfigurationsraum

Abb. 7.1.: Vergleich der Hindernisformation im Arbeits- und Konfigurationsraum eines Roboters mit serieller 2 DOF Kinematik. Abbildung adaptiert aus [159].

7.1.2. Graphensuche

Ein grundlegender Schritt der Planung ist das Aufspannen eines abstrakten Graphen G innerhalb des Konfigurationsraums, dessen Knoten V meist diskrete Zustände des Roboters beschreiben, während die Kanten E unterschiedlich teure Zustandsübergänge abbilden. Auf diesem Graphen können dann unterschiedliche Suchfunktionen genutzt werden, um Wege von einem Start- zu einem Zielknoten zu finden. Die Auswahl der Suche bestimmt mit über die Leistungsfähigkeit des Planers und sollte daher passend zu den Eigenschaften des Graphen getroffen werden.

Der klassische Algorithmus, der auf einem statischen Graphen mit gewichteten Kanten den kürzesten Weg zwischen zwei Knoten finden kann, ist der Dijkstra Algorithmus, der

1959 publiziert wurde [71]. Um auf einem engmaschigen Graphen, wie z.B. einer diskretisierten 2D Karte mit Achternachbarschaft effizienter zu suchen, wurde das ursprüngliche Greedy-Verfahren 1968 von von Hart et al. [97] um eine Heuristik ergänzt. Der entstandene A*-Algorithmus schätzt die Distanz eines Knotens zum Ziel, um darüber die Suche in Richtung des Ziels zu lenken (informierte Suche). Inzwischen existieren einige parallelisierte Varianten (vgl. Übersicht in [209]), die den Suchraum aufteilen und im Falle von R* auch auf der GPU implementiert wurden [118]. Wie später noch gezeigt wird, ist der Einsatz einer parallelen Suche in dieser Arbeit jedoch nicht zielführend.

Die Nachteile von A* sind sein hoher Speicherverbrauch und die Tatsache, dass bei Änderungen im Graphen bereits generierte Teilpläne wertlos werden. Vor diesem Hintergrund wurde D* (Dynamic A*) entwickelt [191], der es ermöglicht, gefundene Teilpläne beizubehalten und lediglich ab dem blockierten Teil inkrementell neu zu planen. Populär ist jedoch die D*-Lite Variante, die König und Likhachev 2005 veröffentlichten [121], da diese wesentlich einfacher umzusetzen ist, und dennoch mindestens so effizient wie D* ist. D*-Lite plant von Ziel zu Start und nutzt einen zweidimensionalen Tupel k, um den n-ten Knoten zu bewerten:

$$k(n) = \begin{pmatrix} min(g(n), rhs(n)) + h(n_{start}, n) + k_m \\ min(g(n), rhs(n)) \end{pmatrix} \tag{7.2}$$

Neben dem $g(n)$-Wert, der wie bei A* die kürzeste Distanz ab dem Startknoten n_{start} beschreibt und der Heuristik $h(n, n_{start})$, die die erwartete Entfernung bis zum Startknoten abschätzt, spielt der $rhs(n)$-Wert (*Right Hand Side*) eine wichtige Rolle: Er speichert, ähnlich zu g, in jedem Knoten den vom Zielknoten aus zurückgelegten Wert, allerdings unabhängig von den g-Werten des Vorgängerknotens n'. Daher kann er genutzt werden, um bei Änderungen im Graphen Inkonsistenzen aufzudecken. Diese entstehen, wenn aufgrund von Hindernissen höhere Kosten $c(n', n)$ zwischen den Knoten n und n' auftreten und somit $g(n) = g(n') + c(n', n) \neq rhs(n)$ ist. In diesem Fall muss $rhs(n)$ rekursiv aus neu bestimmten g-Werten gesetzt werden. Diese gezielte Reparatur des Graphen ist effizienter als der komplette Neuaufbau und erlaubt den Einsatz von D*-Lite auf dynamischen Daten. Somit können Pfade durch teilweise unbekannte Regionen geplant werden, die erst während der Pfadausführung einsehbar sind. Da D*-Lite den kürzesten Pfad jedoch durch die Minimierung von rhs sucht und dieser während dem Abfahren des Pfades automatisch abnimmt, ist der Ausgleichsterm k_m notwendig, der die bereits abgelaufene Distanz kompensiert.

Bei der Komplexitätsbetrachtung eines samplingbasierten Planers, wie dem RRT oder RRT*, erscheint die Kollisionsdetektion nur als konstanter Zeitfaktor während Graphen-Operationen wie Nachbarschaftssuche und Einfügeoperationen mit $\mathcal{O}(\log n)$ den größeren Aufwand erzeugen. Dies trifft jedoch nur für den in der Praxis irrelevanten Fall der asymptotischen Betrachtung zu. In realistischen Szenarien, insbesondere jedoch bei der Betrachtung einer dreidimensionalen Umwelt, überwiegt die Laufzeit der Kollisionsprüfung der Planer-Samples bei weitem die Berechnungen im Graphen [46].

7.1.3. Taxonomie der Planungsverfahren

So vielfältig wie die Kollisionsprüfungsverfahren sind auch die Ansätze zur Bewegungsplanung, die auf ihnen aufbauen. Und auch hier unterscheiden sich die Verfahren dar-

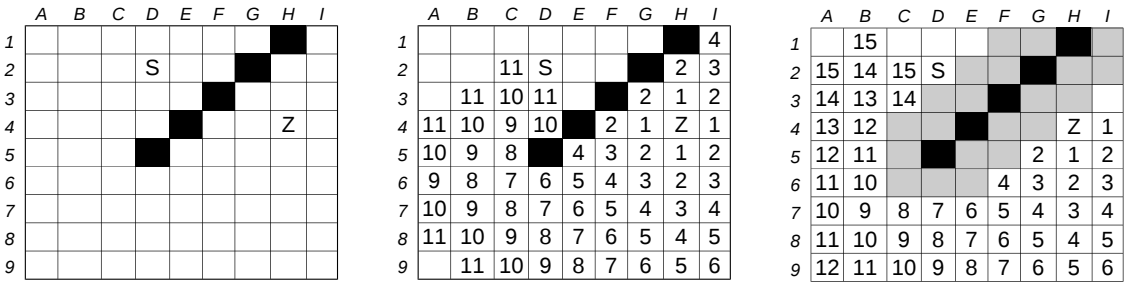
in, ob sie auf a priori Wissen in Form von abstrakten, exakten Modellen ausgelegt sind oder auf aktuellen, probabilistischen Sensordaten arbeiten können. Im ersten Fall wird meist von einem vollständigen Umweltwissen ausgegangen, um global korrekte und möglichst optimale Pläne zu generieren. Soll hingegen auf Sensordaten und somit auf unvollständigem Wissen geplant werden, müssen sich die Ergebnisse einfacher an unvorhergesehene Gegebenheiten anpassen lassen. Hierbei können Planer konservativ und pessimistisch vorgehen, oder aber optimistische und opportunistische Entscheidungen treffen (Vgl. Sensorhorizont: *Over the Horizon Planning* [73]).

Gleiches gilt für die Annahme einer statischen oder dynamischen Umgebung. Wie bereits bei der Berechnung von Distanzkarten, gibt es auch hier zwei Möglichkeiten: Sind die Algorithmen leichtgewichtig, lassen sie sich zyklisch oder bei jeder Änderung der Umwelt komplett neu berechnen. Alternativ müssen die Auswirkungen einer Veränderung analysiert werden, um dann gezielt lokale Anpassungen durchzuführen. In beiden Fällen findet die Planung auf Standbildern der Umgebung statt. Um dennoch die zeitliche Komponente berücksichtigen zu können, nutzen einige Verfahren dynamische Modelle der Hindernisse und des Roboters, um Bewegungen in Form räumlicher Ausdehnungen explizit zu modellieren (vgl. Abschnitt 4.6 und Abschnitt 6.2.1).

Eine Übersicht der einflussreichsten Ansätze zur Bewegungsplanung soll nun helfen, die Verfahren zu vergleichen und ihre Verwendbarkeit in Kombination mit voxelbasierter Kollisionsdetektion beurteilen zu können. Die Liste orientiert sich am Referenzwerk *Robot Motion Planning* von Latombe [126].

Kombinatorische Verfahren

Entspricht die Dimensionalität des Konfigurationsraumes der des Planungsraumes, und ist diese niedrig ($n \leq 3$), können geometrische Verfahren zur Zelldekomposition (bspw. Voronoi Zerlegung, Sichtbarkeitsgraph, Line-Sweep ...) den Arbeitsraum in freie und belegte Regionen unterteilen. In den entstehenden Strukturen spannt sich ein Graph auf, der alle kollisionsfreien Trajektorien repräsentiert [175]. Darauf lassen sich dann die oben genannten erschöpfenden oder heuristischen Suchverfahren einsetzen, um einen Weg von Start zu Ziel zu finden. Eingesetzt werden diese Verfahren oft auf 2D oder 2,5D Datenstrukturen, in denen ein Planungszustand die Grundfläche einer mobilen Plattform innerhalb einer Kostenkarte beschreibt [127]. Es wird entweder von einem punkt-, scheiben- oder kugelförmigen Roboter ausgegangen, so dass dessen Ausrichtung vernachlässigt werden kann. Im Sonderfall einer äquidistanten Aufteilung des Planungsraumes ist der einfachste Ansatz der Planung eine Breitensuche, bei der alle Zellen nach einer Metrik (z.B. Manhattan- oder Chebyshev-Distanz) mit Kosten beschrieben werden. Dies entspricht einer künstlichen Wellenfront (*Wave Front*) oder einer Flutung der Karte (*Flood Fill*), wie auch im Bsp. aus Abb. 7.2b zu sehen ist. Erreicht die Welle das Ziel, kann entlang des Kostengradienten direkt der kürzeste Pfad abgelesen werden. Diese Art der Planung findet immer einen Pfad zwischen Start und Ziel, falls dieser existiert. Allerdings sind die Kosten dafür meist höher als bei einer heuristischen Suche. Unter Verwendung von Kugel- oder Zylinderkoordinaten können die Verfahren auch für einfache serielle Kinematiken verwendet werden.



(a) Ausgangsdaten mit Start und Ziel

(b) Welle ausgebreitet, Pfad gefunden

(c) Berücksichtigung eines Mindesabstandes

Abb. 7.2.: Ausbreitung einer Wellenfront. Wurden Abstandswerte zu den Hindernissen bspw. mittels einer EDT berechnet, können Felder, die nah an Hindernissen liegen, wie in (c) effizient ausgelassen werden.

Dienen Voxel als Basis der Zelldekomposition, eignen sich kombinatorische Verfahren sehr gut für die Verwendung in dieser Arbeit, insbesondere in der Kombination mit rotationsinvarianten Egomodellen. Als Beispielanwendung wird in Unterabschnitt 7.2.2 die Planung einer mobilen Plattform mittels Rotations-Swept-Volumen detailliert beschrieben und in Unterabschnitt 8.7.2 evaluiert.

Potentialfelder

Anstatt den Suchraum kombinatorisch aufzuteilen, kann er auch als Potentialfeld interpretiert werden, in welchem die Hindernisse in Form von abstoßenden Potentialen dargestellt werden. Kombiniert man deren Felder mit einem zweiten Feld, in dem das Ziel eine anziehende Kraft ausübt, entsteht ein Gradient, dem ein rotationsinvarianter Roboter folgen kann, um ohne weiteren Suchaufwand von jedem beliebigen Punkt im Arbeitsraum zum Ziel zu gelangen [116]. Allerdings müssen vielfältige Randbedingungen beachtet werden, deren Verletzung sonst zu lokalen Minima und somit zu Blockaden oder Sackgassen führt, in denen sich die Potentiale aufheben. Durch diese Problematik ist der Ansatz nur sehr bedingt für globale Planungsprobleme einsetzbar und wird häufig für reaktive Verfahren verwendet.

Entstehen können diese durch unpassende Kostenfunktionen oder ungeschickte Gewichtung der beiden Potentialfelder bei ihrer Linearkombination. Ein Beispiel ist in Abb. 7.3a zu sehen. Hier ist die Abstoßung durch das Hindernis so groß, dass ein Agent vor dem Durchlass gefangen wäre. Um lokale Minima weitestgehend zu vermeiden, sollten die Felder durch harmonische Funktionen [201, 138] wie beispielsweise

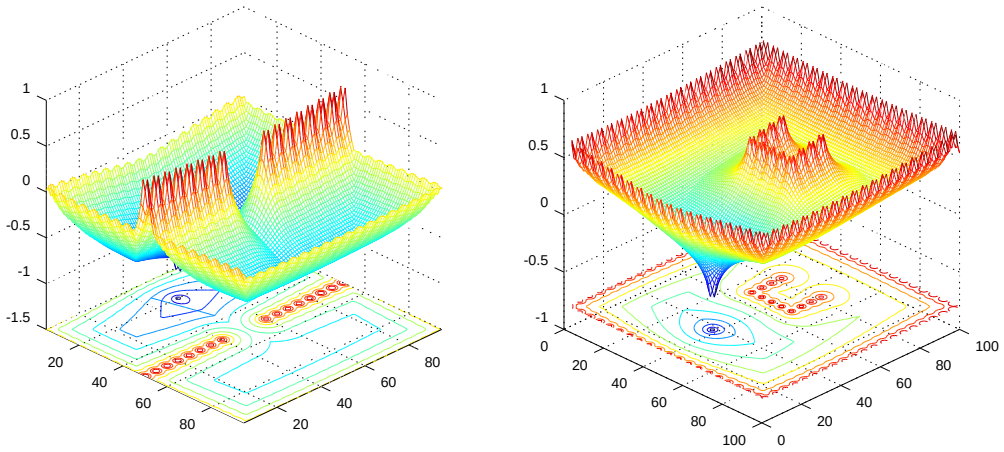
$$\phi = \lambda \ln d, \qquad d = \|Position - Ziel\| \quad \text{bzw.} \quad d = \|Position - Hindernis\|$$

beschrieben werden. λ ist hierbei ein Skalierungsfaktor, dessen Vorzeichen über Anziehung oder Abstoßung entscheidet. Doch auch dies schützt nicht vor Fallen in Hindernissen, die konvexe Formen aufweisen [78]. Ein Beispiel zeigt Abb. 7.3b, wobei hier die starke Gewichtung des anziehenden Feldes das Problem umgeht. Strategien zum Umgang mit lokalen Minima, wie *Random-Walks* oder *Wavefront Expansion* (*Best First Search*), beschreiben unter anderem Barraquand et al. [41].

Enthält eine Karte in jedem Punkt Informationen über das nächstgelegene Hindernis, können Potentialfeldplaner sehr effizient arbeiten, da sie nur einen kleinen Bereich um die aktuelle Roboterposition auswerten müssen, um den Gradienten zu finden, dem zu folgen ist. Im Gegensatz dazu skaliert die Rechenzeit von Planern, die künstliche Wellenfronten nutzen, mit der Größe der betrachteten Karten und auch mit den Längen der erzeugten Pläne. Die Komplexität von Navigationsfunktionen [123] ist wiederum von der Anzahl der Hindernisse in der Umgebung abhängig und eignet sich daher nur bedingt für dichte und hoch auflösende 3D-Karten.

Weiterhin existieren Arbeiten, in denen auch Bewegungen für artikulierte Roboterkinematiken mittels Potentialfeldern geplant werden [137]. Da hierbei nicht mehr von einer rotationsinvarianten Geometrie ausgegangen wird, müssen die Potentialfelder nach dem Vorbild eines elektrostatischen Feldes auch Drehmomente erzeugen, die auf die einzelnen Roboterglieder wirken. Einen aktuellen, GPU beschleunigten Ansatz mit einem 2,5D Potentialfeld stellen Kaldestad et al. in [109] vor. Allerdings handelt es sich hierbei eher um ein Regelungverfahren und nicht um einen Planer, da hier die Impedanzregelung eines Leichtbauroboters durch virtuelle Kräfte des dynamischen Kraftfeldes von Hindernissen ferngehalten wird. Ein ähnliches CPU-basiertes Verfahren wurde bereits 2012 von Flacco et al.[80] publiziert. Ein echtes Planungsverfahren hingegen konnte Kitamura bereits 1995 in [120] vorstellen, bei dem Voxel-Potentialfelder für die Navigation in dreidimensionalen Umgebungen genutzt werden.

Vergleichbar dazu wird auch in der vorliegenden Arbeit der Arbeitsraum diskretisiert, um jeder Zelle ein Potential zuweisen zu können. Dafür kommen Distanz-Voxel und die parallelisierter Distanzberechnung aus Abschnitt 5.6 zum Einsatz.



(a) (b)

Abb. 7.3.: Kostenfunktion aus kombinierten Potentialfeldern: Das Navigationsziel liegt im Minimum der Funktion (dunkelblau), während Hindernisse hohe Kosten (rot) aufweisen. Bilder aus [24].

Samplingbasierte Verfahren

Hochdimensionale Planungsräume werden fast ausschließlich mit randomisierten, samplingbasierten Planern erschlossen. Die Herausforderung liegt dabei in der geschickten Verteilung der Samples, die sich bestenfalls an lokale Gegebenheiten anpasst. Anders als deterministische Ansätze, die durch ein *Resolution Complete Sampling* Aussagen zur Existenz einer Lösung treffen können, geben diese Algorithmen zwar keine Garantie, eine vorhandene Lösung zu finden, sind aber dennoch probabilistisch vollständig (die Wahrscheinlichkeit, eine existente Lösung zu finden konvergiert mit steigender Planungszeit gegen 1). Zu den populärsten Verfahren zählen die *Rapidly Exploring Random Trees* (RRT) [131].

Wie der Name andeutet, bauen diese Algorithmen zunächst einen Graphen $G = (V, E)$ aus randomisierten Samples auf, bevor sie auf diesem einen Pfad suchen. Zustände im Planungsraum werden durch Knoten V im Roadmap-Graphen repräsentiert. Trajektorien zwischen ihnen durch Kanten E. Ausgehend vom Startzustand wird der Graph so expandiert, dass möglichst schnell eine Abdeckung des gesamten Raumes erreicht wird. Dafür werden zunächst randomisiert Punkte im Planungsraum gewählt, deren nächster Nachbar im Graphen gesucht wird. Ausgehend von diesem Nachbarn wird in Richtung des Zufallspunktes, in einer festen Distanz, ein weiterer Punkt generiert. Ist dieser kollisionsfrei, wird er zu V hinzugefügt, eine Kante zu seinem Nachbarn generiert, und G damit erweitert. Je nach Explorationsdistanz muss dafür die neue Verbindung nochmals interpoliert und auch ihre Zwischenpunkte auf Kollisionsfreiheit überprüft werden. Werden zwei Bäume genutzt, die gleichzeitig von Start- und Zielzustand ausgehend wachsen, um sich zu verbinden, spricht man von *Bidirectional Rapidly-Exploring Random Trees* (BiRRT) [11]. Hierbei wird die Verbindung zwischen den Bäumen provoziert, indem diese abwechselnd in Richtung des zuletzt hinzugefügten Knotens des anderen Baumes wachsen, bis es zu einer Kollision kommt. Das Überprüfen der Graphenkanten auf Kollisionsfreiheit ist ein rechenintensiver Prozess, der für alle Kanten ausgeführt wird, auch wenn diese am Ende nicht am Lösungspfad beteiligt sind. Daher bietet sich eine so genannte *Lazy Evaluation* an: Dabei werden lediglich die Knoten auf Kollision geprüft und alle Kanten zunächst als valide angenommen. Erst wenn ein Pfad im Graph gefunden wurde, werden seine Kanten auf Kollisionsfreiheit geprüft. Da in vielen realen Planungsszenarien mindestens von einem 70/30 Verhältnis aus freiem / belegtem Raum ausgegangen werden kann, erspart die Lazy Evaluation viel Rechenzeit. Sollte eine Kante nicht ausführbar sein, muss ein anderer Pfad im Graphen gesucht werden. In Unterabschnitt 8.6.2 sind Versuche mit einem *Lazy Bi-directional KPIECE with one level of discretization* (LBKPIECE1) Planer beschrieben. Dieser kombiniert die nachgelagerte Kollisionsprüfung mit einer bidirektionalen Suche, die über eine Projektion in einen zweidimensionalen Raum geleitet wird [194].

Die Nachteile samplingbasierter Verfahren liegen in ihren nachweisbar suboptimalen Lösungen und der häufigen Notwendigkeit einer nachgelagerten Glättung, da die entstehenden Lösungswege sehr unstetig sind. Auch hierbei sind Kollisionsprüfungen unerlässlich, was ihren Aufwand steigert. Abhilfe schafft hier der RRT* Algorithmus, der asymptotische Optimalität erreicht (mit steigender Planungszeit konvergiert die Lösung zur optimalen Lösung), indem er den entstehenden Graphen reorganisiert und dabei auch glattere Trajektorien erzeugt. Wegen der praktisch beschränkten Samplingdichte

stellen enge oder verwinkelte Korridore jedoch immer große Herausforderungen an alle Suchverfahren.

Parallelisierte Varianten des RRT bzw. RRT* wurde von Devaurs et al. [69] und Bialkowski et al. [46] vorgestellt, wobei Letzteres lediglich eine Art Stapelverarbeitung von GPU Kollisionsprüfungen nutzen und nicht den Planer parallelisieren.

Bei Nutzung der parallelisierten Voxel-Kollisionsprüfung mit einem samplingbasierten Verfahren ergibt sich bei komplexeren Kinematiken die Problematik, dass die Konfiguration jedes Samples einzeln in Voxel umgewandelt werden muss. Eine Vorberechnung ist nicht möglich, und auch die effiziente Translation mittels Basisversatz aus Unterabschnitt 5.3.1 kommt nicht in Frage. Weiterhin erfordern viele Planer eine sequentielle Prüfung ihrer Samples, da diese inkrementell die Expansionsrichtung beeinflusst. Aus diesen Gründen kann mit ihnen nicht das volle Potential der GPU-Parallelisierung genutzt werden.

Roadmap Verfahren

Sollen in einer quasistatischen Umgebung mehrfach Bewegungen geplant werden, bietet es sich an, die Suche von der Erstellung der bereits beschrieben Graphen (Raodmaps) loszulösen. Somit kann einerseits auch bei noch unbekanntem Start- / Zielpunkt bereits ein Graph aufgebaut werden, und dieser anderseits über mehrere Suchanfragen hinweg beibehalten oder erweitert werden, um den Freiraum $\mathcal{C}_{\text{Frei}}$ möglichst gut abzudecken.. Im Falle einer Anfrage müssen lediglich zwei kurze Pfade zu den nächstgelegenen Start- und Zielknoten bestimmt werden, während der eigentliche Pfad im existierenden Graphen schnell gefunden werden kann. Ein bekanntes Beispiel ist der *Probabilistic Roadmap Planner* (PRM) von Kavraki et al. [113], der inkrementell arbeitet und einen lokalen Planer zum Hinzufügen von Kanten involviert. Dafür wird zunächst eine Menge randomisierter Punkte im Planungsraum erzeugt und diese im Arbeitsraum auf Kollisionsfreiheit überprüft. Im nächsten Schritt wird versucht, Verbindungen zwischen kollisionsfreien Punkten und ihren k-nächsten Nachbarn zu erstellen. Wie bei RRT müssen die Verbindungen dafür eventuell auch feingranular auf Kollisionsfreiheit überprüft werden oder es kommt sogar ein lokaler Planer zum Einsatz. Kann mindestens eine Verbindung zum Graphen hergestellt werden, wird der Punkt zur Menge V und die Verbindung(en) zu E hinzugefügt. Diese Schritte werden so lange wiederholt, bis ein möglichst dichter und zusammenhängender Graph entstanden ist. Bei einer Suchanfrage müssen der Start- und Zielpunkt auf dieselbe Art mit dem Graphen verbunden werden. Danach kann mittels A*, Dijkstra o.Ä. ein Pfad vom Start- zum Zielpunkt gesucht werden. Eine Eigenschaft der PRM Methode ist die problembezogene Optimierung der Samplingstrategie. So gibt es Verfahren, die besonders für enge Passagen im Arbeitsraum geeignet sind und große zusammenhängende Freiräume möglichst spärlich abtasten (Bridge Sampling, Obstacle based Sampling, Gaussian Pair Sampling).

Eine PRM Umsetzung auf der GPU stammt von Pan et al. [158]. Der vorgestellte *g-Planner* arbeitet auf globalem Wissen und ermöglicht eine Echtzeitplanung, indem alle relevanten Schritte paralellisiert wurden und für die Kollisionsprüfung das bereits genannte BVH Verfahren *gProximity* [130] eingesetzt wird.

Sollen Roadmap Verfahren in veränderlichen Umgebungen eingesetzt werden, ist die Invalidierung von Kanten im Graphen sehr wichtig, wenn Änderungen in der Umwelt Teilpläne blockieren. Dies erfordert jedoch eine bijektive Abbildung zwischen Ausführungs- und Planungsraum, welche, wie eingangs beschrieben, nur in niedrigdimensionalen Fällen gegeben ist. Anderenfalls muss über eine zusätzliche Datenstruktur für jedes Raumvolumen die Menge an Plänen vorgehalten werden, die dieses Volumen kreuzen. Ist dies gegeben, kann eine Invalidierung inklusive einer Suche innerhalb weniger Millisekunden durchgeführt werden. Dies zeigt Schumann-Olsen in [184] bei der Planung eines 5-Achs-Roboters. Die Besonderheit dabei ist die effiziente bidirektionale Abbildung zwischen Arbeitsraum $\mathcal{S}$ und Planungsraum $\mathcal{C}$ mit Hilfe einer komprimierten Lookup-Tabelle. Allerdings ist das Verfahren nicht allgemeingültig und lässt sich schlecht auf komplexere Kinematiken erweitern.

Auch in dieser Arbeit wurde erwägt, jeden Voxel des Umweltmodells mit der Menge an Plänen zu annotieren, die durch ihn hindurchführen. Somit wären im Falle einer Kollision des Voxels direkt ablesbar, welche Teilpläne zu invalidieren sind. Da dies jedoch einen dynamischen, pro Voxel unterschiedlichen Speicheraufwand erfordert, wurde die Idee verworfen.

Optimierungsverfahren

Um global konsistente Pläne bei kleineren Änderungen in der Umwelt nicht komplett verwerfen zu müssen, entstanden Arbeiten zur lokalen, dynamischen Modifikation von Plänen. In den klassischen Methoden *Elastic Bands* [165] und *Elastic Strips* [52] ist es möglich, zur Laufzeit die Graphenkanten in einem gewissen Maß zu modifizieren, und so lokalen Hindernissen auszuweichen, während globale Vorgaben eingehalten werden. Einen aktuellen, optimierenden GPU-Planer stellen Park et al. in [161] vor. Dieser ließe sich durch die Verwendung der vorgestellten Distanzkarten generalisieren, was in weiterführenden Arbeiten geplant ist. Da diese Arbeit jedoch lokale Optimierungen generell vermeiden möchte, werden diese Verfahren zunächst nicht weiter beleuchtet.

Planung mit Bewegungsprimitiven

In vielen Fällen kann es zielführend sein, einen Plan nicht aus diskreten Posen, sondern mit Hilfe einer Bibliothek aus Primitiven fundamentaler Bewegungen zu synthetisieren (*Motion Primitive Planning*). Ähnlich zur Diskretisierung des Arbeitsraumes, zur Einschränkung der Berechnungskomplexität, wird hierbei das Ziel verfolgt, den Konfigurationsraum zu diskretisieren. Anstatt jeden Freiheitsgrad als unabhängig zu betrachten werden also zusammenhängende Bewegungen eines oder mehrerer Freiheitsgrade zu Primitiven zusammengefasst, womit eine beliebig starke Diskretisierung möglich ist. Dabei gilt es, einen Kompromiss zwischen dem Reduktionsfaktor (also der Anzahl an Primitiven) und Abbildungsgenauigkeit zu erreichen (Erreichbarkeit aller Zielpunkte). Die Anzahl und Art der Primitive entscheidet somit, ob ein Planungsproblem lösbar ist. Das Erzeugen einer Menge geeigneter Primitive geschieht vor der eigentlichen Planungsphase und kann entweder manuell, durch den Einsatz von Vorwissen geschehen, oder durch maschinelle Lernverfahren [135].

Im Falle von nichtholonomen Robotern kann durch die Planung mit Primitiven bspw. die kinematische Durchführbarkeit des resultierenden Planes sichergestellt werden, ohne die physikalischen Beschränkungen des Roboters in der Planung explizit zu betrachten [183]. Somit wird verhindert, dass der Planer intrinsisch unmögliche Zustandsübergänge evaluieren und eventuell verwerfen muss. Dadurch reduziert sich das Planungsproblem auf die Suche einer geeigneten Konkatenation von Primitiven, welche den Roboter von seinem Start- in einen gewünschten Zielzustand überführen. Weiterhin kann eine Zustandsabhängigkeit in jedem Bewegungsprimitiv modelliert werden, um nicht ausführbare Übergänge zu unterbinden. Aber auch bei holonomen Robotern kann damit eine Vorzugsrichtung und somit eine für den Menschen intuitivere Bewegungsbahn realisiert werden.

Wie bei den Roadmap Verfahren entspricht die Planung letztendlich dem Aufbau eines gerichteten Graphen, dessen Knoten die diskretisierten Zustände darstellen, während die Kanten die durch die Primitive ausführbaren Zustandsübergänge repräsentieren. Mit fortschreitender Suchtiefe bei der Exploration verzweigt sich der Baum an jedem Knoten um den Faktor der verfügbaren Bewegungsprimitive. Daher ist es wichtig, dass im Baum entstehende Schleifen erkannt und geschlossen werden, um ein exponentielles Anwachsen des Baumes zu verhindern. Auch die Herausforderungen der Roadmaps finden sich hier wieder: Ein inhärentes Problem bei der Arbeit mit Primitiven besteht, wenn sich nach der Planung eines (Teil-)Pfades Änderungen in der Umwelt ergeben. Wird dadurch ein Knoten im Planungsgraphen als unpassierbar markiert, werden nur diejenigen Knoten aktualisiert, von denen aus der geänderte Knoten über ein Primitiv erreichbar ist. Liegt die Änderung jedoch auch auf der Strecke eines anderen Primitivs oder zwischen solchen Endknoten, wird sie bei der Aktualisierung übersehen. Dies führt zu einer Inkonsistenz im Planungsgraphen, da die Kosten der Transitionen nicht mehr der realen Umwelt entsprechen. Da keine bijektive Abbildung zwischen Planungsraum und Ausführungsraum existiert, ist somit nicht klar, ob und welche anderen Kanten von dem Hindernis betroffen sind. In diesem Fall muss der Planungsgraph invalidiert werden. Ein Beispiel dazu ist in Abb. 7.4 zu sehen. Weiterhin ergibt sich das Problem, von einer beliebigen Pose zunächst auf das Planungsgitter zu gelangen, bzw. von einer Pose auf dem Gitter zu einer nichtdiskretisierten Zielpose. Dieses *Lattice Problem* wird meist durch ein zusätzliches lokales Planungsverfahren gelöst.

Einige ausgewählte Arbeiten aus dem Stand der Technik sind die folgenden: Hornung et al. evaluieren Bewegungsprimitive in mehreren zweidiemensionalen Schnitten durch die Umwelt, die sie auf unterschiedlichen Höhen aus 3D-Sensordaten gewinnen [102]. Somit vermeiden sie den Aufwand einer echten 3D-Kollisionsprüfung. Die eigentliche Planung geschieht dann mit einem *Anytime Repairing A** Algorithmus. Paranjape et al. entwickelten einen Planer, der enge Wendemanöver für Flugzeuge planen kann, und damit auch in hindernisreichen Umgebungen, wie einem Wald, erfolgreich reaktiv geplantes Fliegen ermöglicht [160]. Ebenfalls auf schnelle Reaktionen und die Einhaltung von Echtzeitanforderungen sind die Planer von Likhachev et al. optimiert, die Pfade für autonome Autos generieren [136]. Sie reduzieren die Komplexität und somit die Laufzeit, indem auf unterschiedlichen Auflösungen geplant wird, zwischen denen nahtlos gewechselt werden kann. Ebenfalls für den Automobilbereich ausgelegt sind die Tentakelplaner von Wang Ke-ke et al., bei denen virtuelle Tentakel den Raum vor dem Fahrzeug explorieren um schnell den am besten geeignetsten Pfad durch unbekanntes Gelände zu finden [114].

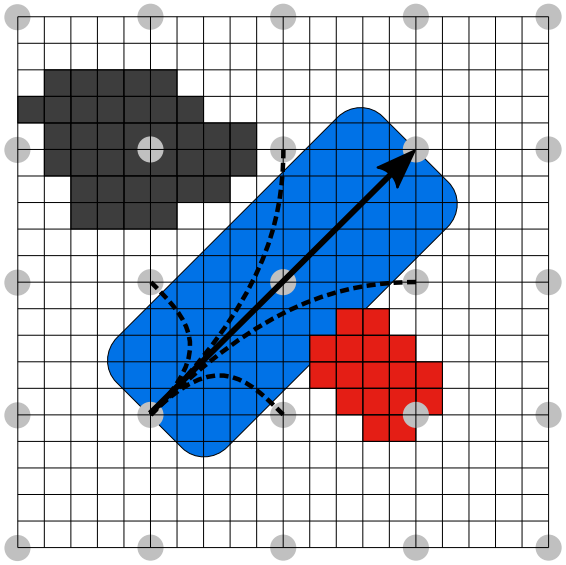


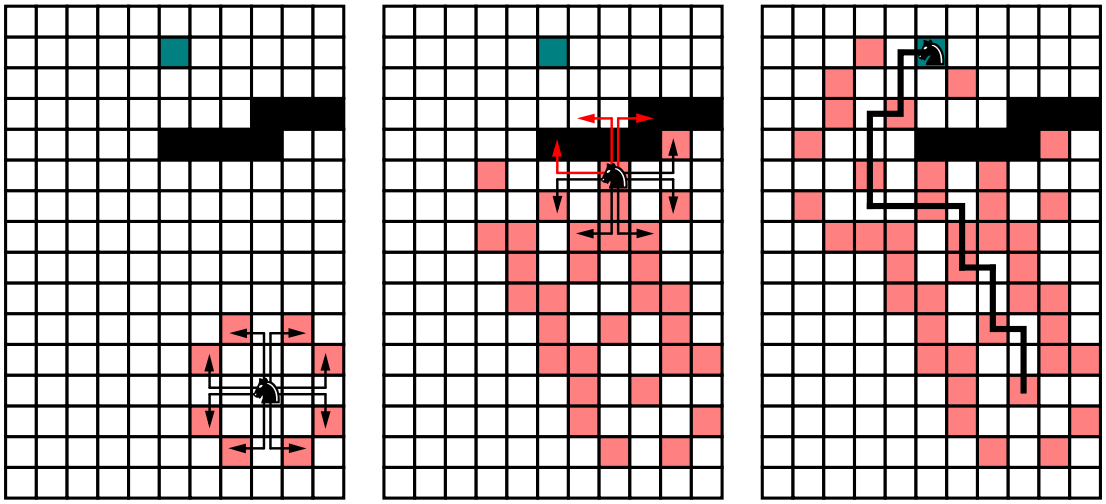
Abb. 7.4.: Ein neues Hindernis, das nicht am Ende eines Primitives liegt, wird übersehen.

Auch in der vorliegenden Arbeit werden Bewegungsprimitive eingesetzt. Da sie offline vorberechnet werden können, lassen sie sich sehr gut durch Swept-Volu-men repräsentieren. Ein Verfahren, das mit rotierenden Bewegungsprimitven Bewegungen für eine holonome mobile Plattform auf einem engmaschigen Gitter generiert, wird in Unterabschnitt 7.2.2 vorgestellt. Die Planung von Bewegungen für nichtholonome Fahrzeuge anhand von längeren Primitiven folgt dann in Unterabschnitt 7.2.3. Beide Ansätze werden weiter unten in diesem Kapitel noch detaillierter ausgeführt und in Unterabschnitt 8.7.3 bzw. Unterabschnitt 8.7.2 für die Planung mobiler Plattformen evaluiert.

Es liegt nahe, die Vorteile der Planung mit Bewegungsprimitiven auch auf serielle Kinematiken übertragen zu wollen. Hierzu existieren Arbeiten von Cohen et al., die adaptiven Primitive verwenden [64], sowie von Barry, deren DARRT System [42] mit parametrisierten Bewegungen arbeitet. Einer Verknüpfung dieser Planer mit der parallelisierten Voxel-Kollisionsprüfung scheitert jedoch an zwei Punkten: Da die verwendeten Primitive veränderlich sind, können ihre Swept-Volu-men nicht vorberechnet werden. Und selbst wenn dies der Fall wäre, ließen sie sich nicht effizient im Voxelraum konkatenieren, da sie hierfür frei im Raum positiniert werden müssten. Dies verursacht neben hohem Berechnungsaufwand auch Abtastfehler und macht die Vorteile der Vorberechnung zunichte.

Kinodynamische Planung

Muss ein Planer auch dynamische Einschränkungen (ausgedrückt durch Differentialgleichungen) berücksichtigen, ergeben sich komplexe Probleme in sehr hochdimensionalen Planungsräumen. Diese werden in der Regel entweder durch samplingbasierte Planer gelöst, oder auch durch die Planung mittels diskretisierter Bewegungsprimitive [83]. Diese Klasse der Planungsprobleme liegt jedoch außerhalb des Rahmens dieser Arbeit.



(a) Start (*Pferd*) und Ziel (türkis) der Planung. Pfeile zeigen 8 der 16 Bewegeungsprimitive des *Pferdes*.

(b) Zwischenstand der Planung: Aktuell sind drei Primitive nicht ausführbar. Hellrote Felder wurden besucht.

(c) Ergebnis der erfolgreichen Planung: Ziel ist erreicht, kürzester Pfad wurde extrahiert.

Abb. 7.5.: Bewegung der Schachfigur *Pferd* als Beispiel für die Planung mit konkatenierten Bewegungsprimitiven. Adaptiert nach [23].

7.1.4. Zusammenfassung

Der hauptsächliche Berechnungsaufwand aller Planer stammt aus der unverzichtbaren Kollisionsprüfung und kann somit durch den parallelisierten Ansatz dieser Arbeit verkürzt werden. Aus obigem Vergleich ist jedoch ersichtlich, dass sich nicht alle Planer effizient mit der GPU Kollisionsprüfung verbinden lassen. Müssen hochdimensionale Probleme mit samplingbasierten Verfahren gelöst werden, verschiebt sich der Aufwand von der eigentlichen Kollisionsprüfung in Richtung der Voxelumwandlung des Egomodells. Somit nutzen diese Planer das Potential der vorgestellten Kollisionserkennung nicht optimal aus. Daher sind Verfahren, die eine Vorberechnung der Voxelumwandlung erlauben, und die Ergebnisse in Swept-Volumen speichern, zu bevorzugen. Prädestiniert ist somit die Planung mit Bewegungsprimitiven, in der Kombination mit den sehr effizient implementierbaren translativen Bewegungen.

Eine weitere Methode zur Beschleunigung der Planung ist die Aufteilung der Probleme in Teilprobleme, die entkoppelt einfacher zu lösen sind: So ist es bei mobilen Robotern noch immer üblich, die Plattformbewegung von den Armbewegungen zu trennen und beides, auf Kosten der Flexibilität, separat zu planen. Um diese Einbußen zu umgehen, wird in späteren Abschnitten zur Planung mobiler Plattformen ein anderer Ansatz verfolgt: Durch die zeitliche Verflechtung von Planungs- und Ausführungsvorgänge (*interleaved planning and execution*) lässt sich die Reaktivität eines Planungssystems ebenfalls steigern.

Implementierungen der meisten beschriebenen Verfahren für das Forschungsfeld der Ro-

botik finden sich in der *Open Motion Planning Library (OMPL)*[1]. Daneben ist auch die *Search-based Planning Library (SBPL)*[2] sehr stark verbreitet, die maßgeblich von Maxim Likhachev entwickelt wird. Beide Bibliotheken bieten jedoch keine integrierten Lösungen für die Arbeit mit volumetrischen Bewegungsprimitiven, weshalb diese im folgenden Abschnitt entwickelt werden.

7.2. Umgesetzte Planungsverfahren

Ausgehend von den zuvor genannten Erkenntnissen wurden mehrere Ansätze praktisch umgesetzt, um Bewegungen sowohl in Szenarien mit mobilen Plattformen als auch mit Manipulatoren planen zu können. Die wichtigste gemeinsame Eigenschaft der entwickelten Verfahren liegt in der Verwendung von dichten Swept-Volu-men. Diese werden einerseits bei der Planung als Alternative zu interpolierenden Kollisionsprüfungsschritten entlang eines Zustandsüberganges eingesetzt, andererseits können sie nach der Planung direkt zur Ausführungsüberwachung weiterverwendet werden.

7.2.1. Überwachung der Planausführung

Wurde durch einen Planer eine Trajektorie gefunden, werden alle darin genutzten Teilpfade in einem großen Swept-Volumen konkateniert, wobei sie eindeutige SSV-IDs erhalten. Somit entsteht ein Korridor, in dem sich der Roboter sicher bewegen kann, und der während der Ausführung des Planes auf eindringende dynamische Hindernisse hin überwacht werden kann. Wird eine Kollision mit dem Korridor erkannt, lässt sich aus der betroffenen ID die Distanz zum Hindernis abschätzen, um den Roboter entsprechend anzuhalten oder zu verlangsamen, während der Planer nach einer alternativen Route sucht. Diese Technik der verschränkten Planung und Ausführung lässt sich unabhängig vom Robotertyp einsetzen und ist schematisch in Abb. 7.6 zusammengefasst. Eine Herausforderung dabei ist es, den aktuellen Plan während der Ausführung nahtlos in den neuen Plan zu überführen. Hierfür ist eine komplexe Logik auf der Ausführungsseite des Roboters nötig, auf die hier jedoch nicht genauer eingegangen werden soll.

7.2.2. Planung mit Rotations-Swept-Volumen

Dieser Abschnitt beschäftigt sich mit der Bewegungsplanung für eine nicht rotationssymmetrische, mobile Plattform mit holonomem Antrieb. Der $SE(2)$ Planungsraum weist die drei Dimensionen $\mathcal{C}(x, y, \theta)$ auf, die mit einer Kombination aus Swept-Volu-men und einem kombinatorischen Verfahren geplant werden. Die Effizienz des entwickelten Verfahrens ergibt sich aus der getrennten Betrachtung der translatorischen und des rotatorischen Freitheitsgrades. Somit können vorberechnete Rotationsbewegungen durch Translation mittels Basisversatz sehr schnell an unterschiedlichen Positionen in der Umwelt auf Kollision geprüft werden (siehe Unterabschnitt 5.3.1). Die Umwelt wird daher nicht

[1] Open Motion Planning Library `http://ompl.kavrakilab.org/`
[2] Search-based Planning Library `http://sbpl.org`

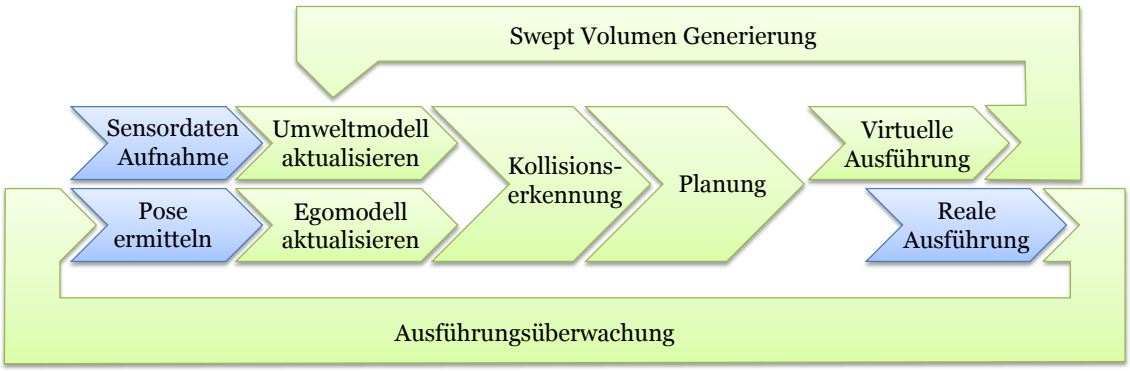


Abb. 7.6.: Virtueller Roboter fährt voraus und erzeugt dabei einen Swept-Volumen-Korridor, den der reale Roboter auf dynamische Hindernisse hin überwachen kann. Blaue Komponenten werden nicht auf der GPU ausgeführt.

durch einen Octree, sondern durch eine Voxelkarte repräsentiert, die in unterschiedlichen Auflösungen vorgehalten wird. Somit können, wie in Unterabschnitt 5.3.2 beschrieben, Kollisionsprüfungen in zwei Auflösungen durchgeführt werden. Die beschriebenen Techniken wurden in der Diplomarbeit von Jörg Bauer [20] erfolgreich umgesetzt und evaluiert.

Unabhängig, aber beinahe zeitgleich mit der hier vorgestellten Methode wurde auch von Dakulovic et al. in [67] ein Verfahren vorgestellt, das mit ähnlichen Techniken arbeitet. Auch Lau et al. arbeiten in [129] mit rotationsabhängigen Modellen, jedoch nur in einer 2,5D Umwelt. Sie generieren Höhenkarten ihres Roboters, die über einen speziellen Faltungsoperator mit Umweltkarten zur Durchfahrtshöhe zu einem befahrbaren Bereich umgerechnet werden.

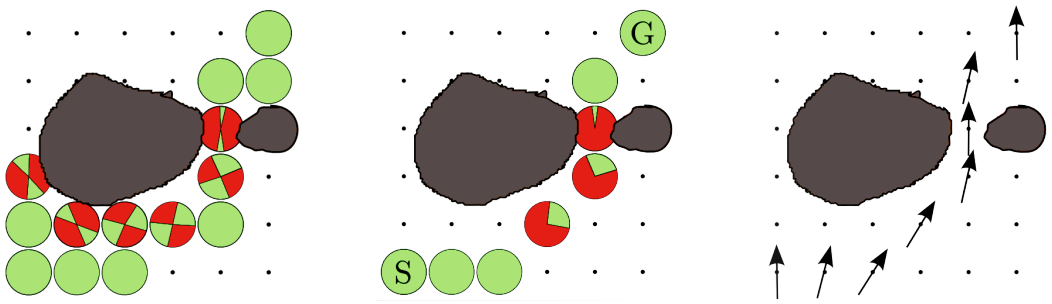
Arbeits- und Planungsraum

Viele Arbeiten, die Plattformbewegungen in einem zweidimensionalen Umweltmodell planen, vernachlässigen den rotierenden Freiheitsgrad θ, indem sie den Roboter als punktförmig annehmen und dafür die Hindernisse um die maximale Ausdehnung des Roboters erweitern. Durch diese konservative Abschätzung können Kollisionen für alle Orientierungen ausgeschlossen werden. Die Ausrichtung des Roboters wird bei diesen Verfahren in einem Nachbearbeitungs- oder Glättungsschritt frei gewählt und meist in Fahrtrichtung ausgerichtet. Problematisch ist diese Abschätzung jedoch in engen Passagen, die der Roboter zwar praktisch durchfahren könnte, die jedoch in der Planung aufgrund der Erweiterung der Hindernisse als nicht passierbar erscheinen.

Um dieses Problem zu vermeiden, werden im hier vorgestellten Verfahren mögliche Plattformorientierungen bei der Planung explizit berücksichtigt. Hierfür wird der Arbeitsraum durch ein zweidimensionales Gitter in der Fahrtebene diskretisiert und auf diesem die Kollisionsfreiheit der Plattform in unterschiedlichen Orientierungen getestet, wie in Abb. 7.7 gezeigt. Zur Reduzierung des Planungsaufwands lässt sich der Gitterabstand wesentlich gröber wählen, als die Auflösung des Umweltmodells. Bleibt er unter-

halb der Breite des Roboters, kann bei einem Übergang zwischen zwei Gitterzellen die Kollisionsfreiheit garantiert werden.

Für die Ermittlung der kollisionsfreien Orientierungen wird ein Rotations-Swept-Volumen eingesetzt, das zunächst in einem Offline Schritt zu erstellen ist. Das Robotermodell wird dafür schrittweise um $360°$ um seine zentrale Rotationsachse gedreht und seine Voxeldarstellung in eine Voxelliste eingetragen (siehe Abb. 7.8b). Da hierbei Bitvektor-Voxel verwendet werden, kann gemäß Unterabschnitt 5.1.4 eine Rotation durch 250 individuell identifizierbare Abschnitte repräsentiert werden. Bei einer Kollisionsprüfung mit der Umwelt lassen sich somit valide Winkelbereiche auf $\sim 1{,}5°$ genau ermitteln.



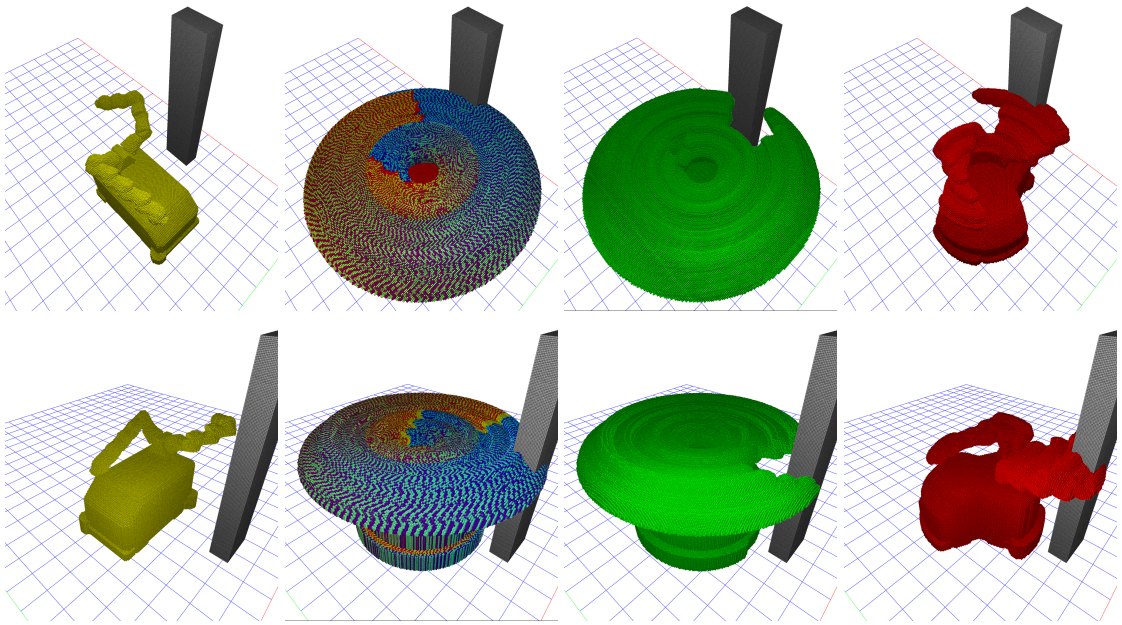
(a) Erweiterung der Nodes (b) Extraktion des Pfades (c) Optimierung des finalen Pfades

Abb. 7.7.: Planung einer Trajektorie für einen mobilen Roboter anhand eines Swept-Volumens seiner Rotation. Veröffentlicht in [3].

Planung mittels D*-Lite

Die eigentliche Planung findet auf einem eng vermaschten Graphen statt, dessen Knoten Zustände aus $\mathcal{C}_{\text{frei}}$ darstellen, die auf dem Planungsgitter liegen. Kanten im Graphen repräsentieren den Übergang zwischen einer Gitterzelle und einer ihrer acht Nachbarzellen. Da jedoch innerhalb einer Gitterzelle mehrere valide Winkelbereiche auftreten können, bedarf es in diesen Fällen mehrerer Graphenknoten um eine Zelle zu repräsentieren. Kanten im Graphen werden nur erstellt, wenn zwei Zellen einen überlappenden, kollisionsfreien Winkelbereich aufweisen. Ein Beispiel dazu ist in Abb. 7.9 gezeigt. Die Kanten im Graphen werden mit den Kosten für einen Übergang zwischen zwei Zellen annotiert. Details zur verwendeten Kostenfunktion folgen später.

Die Suche im Graphen erfolgt mittels des D*-Light-Algorithmus, wobei der Aufbau des Graphen und die eigentliche Suche zeitlich verschränkt ablaufen. Wird eine Zelle expandiert, müssen ihre Nachfolger ermittelt bzw. aktualisiert, und die Kollisionsprüfung des Rotations-Swept-Volumens an den Koordinaten der Zelle durchgeführt werden. Danach können die Pfadkosten jedes Nachfolgers bestimmt werden, wobei die Kosten antiproportional zum übereinstimmenden Winkelbereich sind. Die Überprüfung, ob Nachbarknoten im Graphen überlappende Rotationswinkel aufweisen und welche Winkel dies sind, ist nach der Kollisionsprüfung beider Zellen effizient möglich: Da die Ergebnisse der Prüfungen durch Bitvektoren dargestellt werden, können diese über eine einfache



(a) Mobile Plattform bei 0° Ausrichtung (b) Swept-Volumen einer 360° Drehung mit SSV-IDs (c) Subvolumen des kollisionsfreien Winkelbereichs (d) Subvolumen des kollidierenden Winkelbereichs

Abb. 7.8.: Rotatives Swept-Volumen des IMMP Roboters, das mit einer einzelnen Kollisionsprüfung ausgewertet wird.

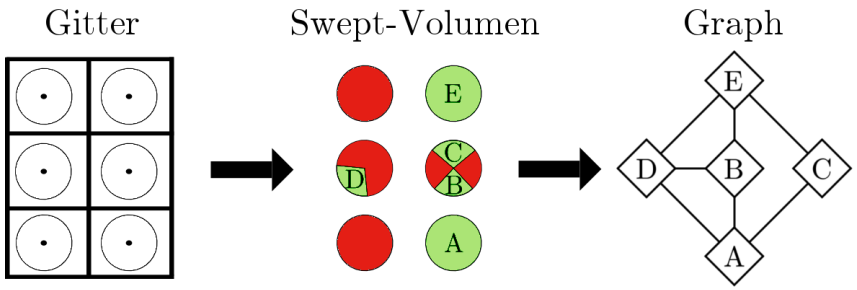


Abb. 7.9.: Überführen der möglichen Rotationen in einen Planungsgraphen. Veröffentlicht in [3].

||-Operation zusammengefasst werden. Nicht gesetzte Bits kennzeichnen dann kollisionsfreie Rotationswinkel. Kann der Roboter somit ohne Änderung seiner Ausrichtung zwischen zwei Zellen wechseln, entstehen keine Übergangskosten und es fließen lediglich die euklidischen Distanzen in die Kostenfunktion ein. Sind die Kosten gültig (< unendlich), kann der Schlüsselwert k (vgl. Gleichung 7.2) des Knotens berechnet werden und damit der Knoten in die Liste der zu expandierenden Knoten eingetragen werden. Da bei der Suche Knoten auch mehrfach auf Kollisionen untersucht werden können, verfügen diese über eine zusätzliche Boolsche Variable, die kennzeichnet, ob ihre Prüfung bereits durchgeführt wurde und somit übersprungen werden kann. Als Resultat entstehen einer oder mehrere neue Knoten und entsprechende Kanten im Surchgraphen. Zusätzlich wird geprüft, ob das Stopkriterium des D*-Lite Algorithmus erfüllt ist. Ist dies der Fall, kann der Pfad von Ziel zu Start extrahiert werden, indem rekursiv der Vorgänger mit den niedrigsten Kosten ausgewählt wird. Der resultierende Pfad besteht aus den zu befahrenden Zellen, wobei sichergestellt ist, dass diese einen zumindest teilweise überlappenden, kollisionsfreien Winkelbereich aufweisen.

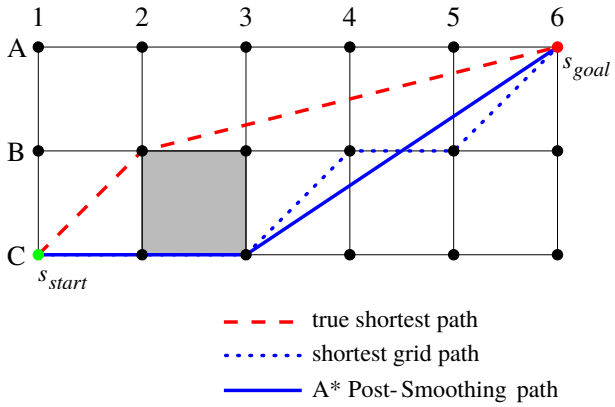
Eine Besonderheit des D*-Light-Algorithmus ist die Wiederverwendung von Teilpfaden, wenn es aufgrund von Änderungen in die Umwelt entlang des geplanten Pfades zu einer Kollision kommt. In diesem Fall werden alle Knoten entlang des Pfades erneut auf Kollisionen geprüft. Repräsentieren mehrere Graphenknoten unterschiedliche Rotationsbereiche innerhalb derselben Gitterzelle, so werden diese nur einmalig auf Kollision geprüft, um Rechenzeit zu sparen. Nicht länger freie Knoten werden aus dem Pfad entfernt, und alle angrenzenden Nachbarn werden wieder zur Liste der zu untersuchenden Knoten hinzugefügt. Durch die folgenden Suchschritte werden die rhs-Werte aller betroffenen Knoten aktualisiert, und die neuen Kosten bis zum Ziel propagiert. Dabei werden im Allgemeinen auch weitere Knoten expandiert und Zellen zum Graphen hinzugefügt. Ist das Stop-Kriterium erfüllt, wurden alle relevanten g-Werte aktualisiert, und ein Pfad, der an die neuen Gegebenheiten angepasst ist, kann extrahiert werden. Ist das Kriterium auch nach der Bearbeitung aller Knoten nicht erfüllt, existiert kein Pfad zwischen Ziel und Start.

Für die Graphendatenstruktur des D*-Lite Algorithmus wurde ein *Geometric Near-neighbor Access Tree (GNAT)* genutzt, weshalb Knoten keine Zeiger auf ihre Vorgänger oder Nachfolger halten müssen, da diese immer auf der Datenstruktur gesucht werden. Somit liegt der benötigte Speicher unter dem normalerweise hohen Speicheraufwand eines A*-Algorithmus. Ob ein Nachbarknoten auf dem Pfad vor oder nach einem anderen Knoten liegt, wird alleine über die Suchheuristik und den rhs-Wert der Knoten entschieden. Die Verwendung eines GNAT vereinfacht weiterhin die Ermittlung der nächstgelegenen Knoten zu gegebenen Start- und Zielkonfigurationen, da diese nicht auf dem diskretisierten Raster liegen müssen. Die Liste der zu expandierenden Knoten wurde als Prioritätsschlange umgesetzt, so dass Einträge sortiert nach ihren Kosten bearbeitet werden können.

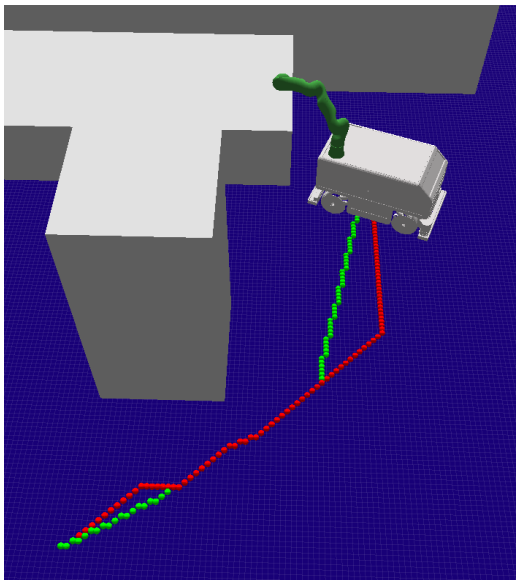
Pfadoptimierung

Planer, die auf einer diskretisierten Umweltrepräsentation arbeiten, erzeugen meist suboptimale Pläne, was die zurückgelegten Distanzen betrifft. Abb. 7.10a verdeutlicht, dass die Manhattan-Distanz auf dem Planungsgitter länger ist, als die direkte Verbindung im

kontinuierlichen Raum. Daher schließt sich an die Planung ein einfacher Nachbearbeitungsschritt an, der versucht, den Pfad mit einem Verfahren aus der Doktorarbeit von Nash [149] zu kürzen. Der umgesetzte Algorithmus iteriert dafür über den Pfad $[s_0..s_n]$, wobei geprüft wird, ob eine Sichtverbindung zwischen s_0 und s_2 besteht. Ist dies der Fall, kann s_1 aus dem Pfad entfernt werden, und es wird zwischen s_0 und s_3 auf Sicht geprüft. So wird weiter verfahren, bis die Sichtprüfung bei s_x fehlschlägt, woraufhin die Iteration bei s_x neu startet und auf Sichtkontakt mit s_{x+2} prüft. Ist das Verfahren bei s_n angelangt, wurden alle unnötigen Zwischenpunkte entfernt und der direkteste Weg ist das Resultat. Sichtverbindung heißt in diesem Fall, dass alle Zellen auf der Geraden zwischen zwei Gitterfeldern unter allen Rotationswinkeln kollisionsfrei sind, womit sichergestellt ist, dass in der Nähe von Hindernissen keine Optimierung durchgeführt wird. Das Verfahren wurde als Erweiterung der Pfadextraktion der D*-Lite Suche implementiert. Ein beispielhaftes Ergebnis findet sich in Abb. 7.10b.



(a) Vergleich unterschiedlicher Pfade (Grafik aus [149]).



(b) Pfadoptimierung durch Nachbearbeitung.

Abb. 7.10.: Optimierung der suboptimalen Pfade eines A*-Planers, die bedingt durch die Diskretisierung entstehen.

Rotationsoptimierung

Neben seiner Länge zeichnet sich ein guter Pfad für eine mobile Plattform auch durch seine Glattheit und die gewahrte Minimaldistanz zu Hindernissen aus. Da die beschriebene Graphensuche bisher lediglich einen Pfad aus zusammenhängenden Gitterzellen liefert, muss durch einen nachgelagerten Schritt eine optimale Plattformorientierung entlang der X/Y-Trajektorie festgelegt werden. Auch wenn die verwendete Plattform einen holonomen Antrieb aufweist, ist es von Vorteil, bei der Fahrt eine Vorzugsrichtung zu beachten, da dies intuitiver für den Menschen ist und bessere Fahreigenschaften erreicht werden. Seitliches oder sogar rückwärts gerichtetes Fahren soll durch hohe Kosten weitestgehend vermieden werden. Weiterhin sind gerade Strecken vor häufig alternierenden

Richtungswechseln zu bevorzugen, so dass höhere Geschwindigkeiten erreichbar sind. Um Aussagen über die Ausrichtung der Plattform verwalten zu können, speichert jede Gitterzelle den Plattformwinkel.

Nachdem ein Pfad gefunden und gekürzt wurde, muss ausgehend von einem Startwinkel die finale Ausrichtung θ der Plattform bestimmt werden. Da für jede Gitterzelle s_n entlang des Pfades der minimale und maximale Winkel $\theta_{s_n \min|\max}$ bekannt ist, kann über eine einfach Mittelung zwischen zwei benachbarten Zellen eine valide Ausrichtung bestimmt werden. Dadurch ist einerseits der Abstand zu allen Hindernissen maximiert, andererseits ist es jedoch wahrscheinlich, dass sich der Winkel häufig ändert, wie in Abb. 7.12a zu sehen ist. Um über längere Streckenabschnitte eine konstante Plattformorientierung zu halten, ist eine geometrische Betrachtung der Winkel wie in Abb. 7.12b hilfreich, bei denen der Winkel so gewählt wird, dass er über eine maximale Anzahl von Zellen nicht geändert werden muss. Die glattesten Fahrbewegungen entstehen jedoch, wenn nicht der Winkel, sondern seine Änderung möglichst konstant bleiben, wie in Abb. 7.12c gezeigt. Hierfür kann mit den minimalen und maximalen Winkeln aufeinander folgenden Zellen die Steigung der Winkelgeschwindigkeit eingegrenzt werden, bis diese eindeutig bestimmt ist (angedeutet durch die gestrichelten Geraden). Da sich minimale und maximale Steigung nur monoton ändern dürfen, entstehen die rot markierten Bereiche, in denen keine Anpassung der Steigung stattfindet. Bei allen Berechnungen werden die Ausrichtungen der Plattform als Polarkoordinaten auf dem Einheitskreis betrachtet, um zwischen den Winkeln mitteln zu können.

Beginnend mit der Orientierung im Startzustand wird diese Ausrichtung entlang des geplanten Pfades weitergegeben und nur geändert, wenn der Winkel in einer Kollision resultiert. In diesem Fall wird der ähnlichste, kollisionsfreie Winkel gewählt. Die Winkeldifferenz α (siehe Abb. 7.11a) geht als $c_R(\alpha)$ in die Kostenfunktion ein (teilweise auch in Form ihrer Ableitung).

Neben diesen Rotationskosten müssen auch die zurückgelegten Distanzen optimiert werden. Die dafür zuständige Kostenfunktion $c_T(\beta)$ sorgt weiterhin dafür, dass seitliche oder rückwärts gerichtete Bewegungen zwar planbar sind, aber aufgrund ihrer hohen Kosten im Allgemeinen vermieden werden. Der verwendete Winkel β liegt zwischen der verwendeten kollisionsfreien Plattformausrichtung und der eigentlichen Bewegungsrichtung beim Zellenübergang (siehe Abb. 7.11b).

Zusammen mit den Kosten für die Länge der zurückzulegenden Strecke c_{Euclid} ergibt sich für die Pfadkosten folgende Summe:

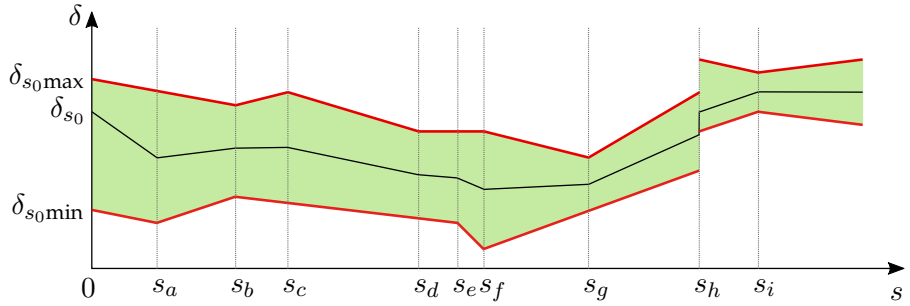
$$c_\Sigma = c_{\text{Euclid}} + c_T(\alpha) + c_R(\beta) \tag{7.3}$$

Planung von Manipulationsposen

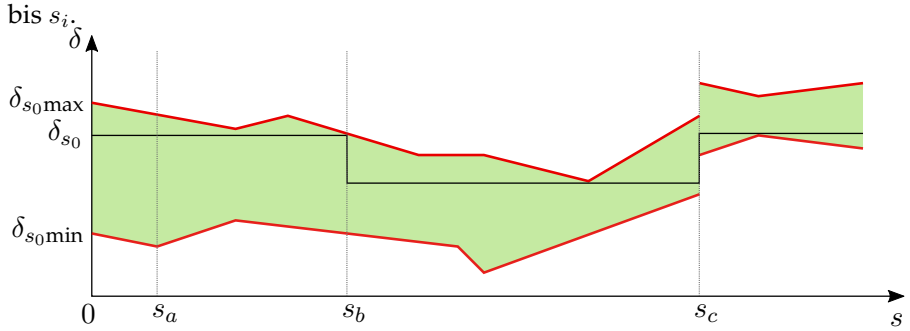
Eine Fragestellung, die bei der Planung von Manipulationsaufgaben mit mobilen Robotern eine große Rolle spielt, ist die Auswahl von geeigneten Plattformposen für die auszuführenden Aufgaben. Dabei muss für die Ausführung sichergestellt sein, dass der Manipulatorarm alle relevanten Objekte erreichen kann, und die Armbewegung zu keinem Zeitpunkt mit der Umgebung in Kollision liegt. Nach aktuellem Stand der Technik

(a) Rotationskosten (b) Kosten der Ausrichtung

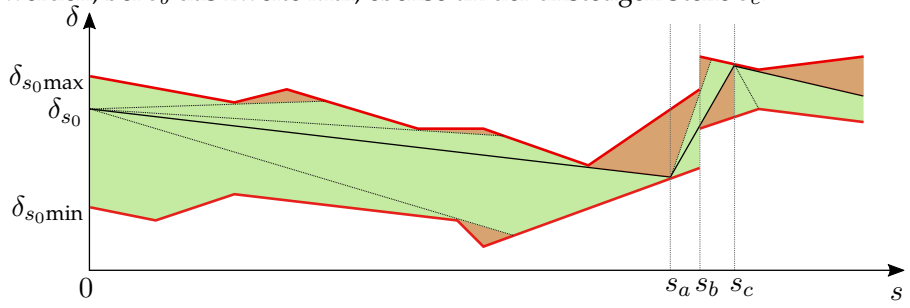
Abb. 7.11.: Bestandteile der Kostenfunktion.



(a) Optimierung auf großem Hindernisabstand. Dies führt zu häufigen Änderungen des Winkels bei s_a bis s_i.



(b) Optimierung auf einen konstanten Winkel. Bei s_a muss die Orientierung das erste Mal angepasst werden, bei s_b das zweite Mal, ebenso an der unstetigen Stelle s_c.

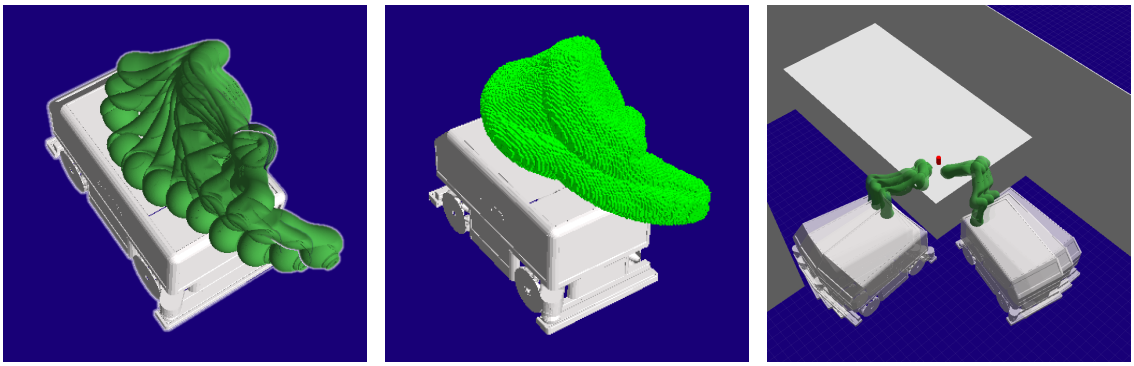


(c) Optimierung auf eine konstante Drehrate. Bei s_a muss die Winkelgeschwindigkeit das erste Mal angepasst werden, bei s_c das zweite Mal. An der unstetigen Stelle s_b dreht sich die Plattform vorbei.

Abb. 7.12.: Unterschiedliche Möglichkeiten zur Wahl der Plattformorientierung θ entlang des Pfades s.

wird dafür eine vielversprechende Plattformpose ausgewählt, und an dieser die Armbewegung simuliert, um auftretende Kollisionen und die Erreichbarkeit evaluieren zu können. Treten Probleme auf, wird die Plattform so lange iterativ verschoben und erneut die Manipulationsaufgabe simuliert, bis diese erfolgreich ist. Dieser zeitaufwendige Prozess kann durch die Verwendung von Swept-Volumen stark verkürzt werden. Dafür muss im Vorfeld durch zahlreiche Simulationen leicht unterschiedlicher Manipulationsausführungen einer Aufgabenklasse (bspw. *Objekt von Tisch aufnehmen* oder *Schublade öffnen*) zunächst das Swept-Volumen des Manipulator-Arbeitsraumes berechnet werden. Beispiele sind in Abb. 7.13 zu sehen. Dieses Volumen rotiert man dann zusammen mit der Plattform, jedoch nicht um das Plattformzentrum, sondern um den Tool Center Point (TCP) des Manipulators. Während der Arm bei der Erzeugung von Rotationsvolumen für die Plattformplanung meist auf einer eingezogenen Parkposition steht, um kompakte Volumen zu erhalten, entstehen nun ausladende Rotationskörper. Prüft man diese mittels Bitvektor-Kollisionstests gegenüber der Umwelt, ist sofort ersichtlich, an welchen Posen im Raum eine potentielle Manipulationsaufgabe ausführbar ist. Diese Posen bilden dann passende Zielkandidaten für den beschriebenen D*-Lite Planer.

Die vorgeschlagene Methode abstrahiert durch eine statistische Vorberechnung von der Vielzahl an Freiheitsgraden, die ein mobiler Manipulator im Allgmeinen aufweist, und nutzt zur Laufzeit lediglich eine sehr schnelle Kollisionsprüfung. Probabilistischen Verfahren, wie *Inverse Capability Maps* [193], ist diese einfache Form der Erreichbarkeitsanalyse in so fern Überlegen, dass sie keine unausführbaren Hypothesen liefert.



(a) Armbewegung einer typischen Manipulationsaufgabe

(b) Swept-Volumen mehrerer, leicht unterschiedlicher Armbewegungen

(c) Potentielle Manipulationsposen: Erreichbar und Kollisionsfrei

Abb. 7.13.: Rotierte Swept-Volumen zur effizienten Evaluierung von Plattformposen bei Manipulationsaufgaben. Veröffentlicht in [3].

Zusammenfassung

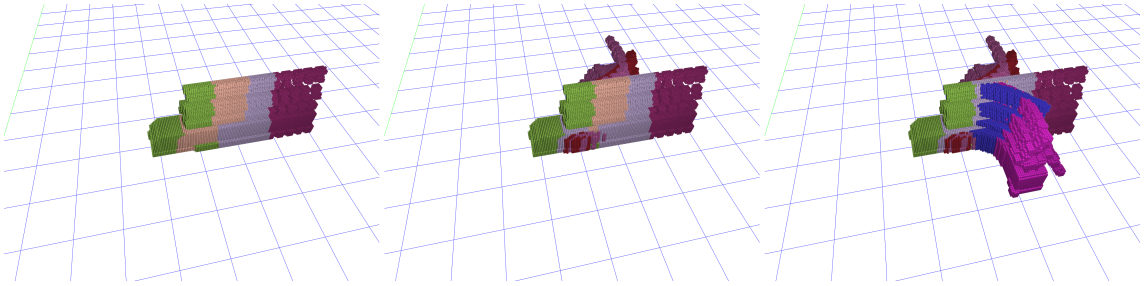
Durch die beschriebene D*-Lite-Planung auf Basis von Rotations-Swept-Volu-men ist es möglich, auch ohne eine zusätzliche Nachbearbeitung glatte Pfade für eine mobile Plattform zu generieren. Da dabei die Orientierung des Roboters in der Kostenfunktion explizit berücksichtigt wird, können über unterschiedliche Kriterien gewisse Bewegungsmuster bevorzugt werden. Durch die Verwendung eines diskretisierten Planungsgitters

bleibt der Konfigurationsraum und damit auch der Planungsgraph übersichtlich. Weiterhin ist der Gitterabstand so gewählt, dass keine interpolierenden Kollisionsprüfungen entlang der Zustandsübergänge erforderlich sind, wodurch kurze Berechnungszeiten erreicht werden. Zusätzlich kann dadurch die Wiederverwendbarkeit von Teilplänen bei Veränderungen in der Umwelt durch den D*-Lite Planer risikofrei ausgeschöpft werden, da bei Zustandsübergängen keine unvorhergesehenen Hindernisse auftreten können. Das Verfahren ist vollständig, so dass existierende Lösungen immer gefunden werden. Es eignet sich somit besonders für die Planung in stark zerklüfteten Innenräumen mit beschränkter Ausdehnung.

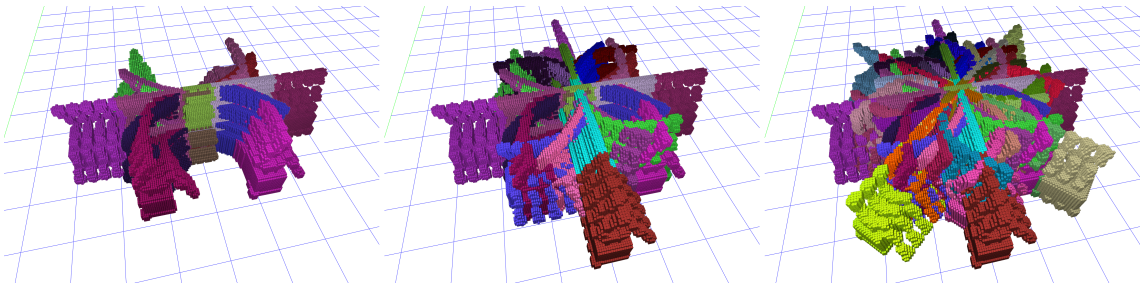
7.2.3. Plattformplanung mit generischen Bewegungsprimitiven

Die Verwendung von Rotations-Swept-Volu-men ist nicht der optimale Ansatz, wenn es größere Distanzen zu überbrücken gilt. Hier bietet sich eine Planung mit Bewegungsprimitiven an, deren grundlegenden Eigenschaften bereits in der Übersicht der Planungsverfahren vorgestellt wurden. Zielplattform ist wieder ein holonomes Fahrzeug, weshalb die verwendeten Bewegungsprimitive lediglich aus 2D Trajektorien in der Fahrtebene bestehen. Ähnlich wie bei der Planung mit Rotationsvolumen sollen auch hier konkatenierbare und universell verwendbare Trajektorienstücke genutzt werden, um daraus längere Pläne zusammenzusetzen. Ziel ist dabei, die rotierenden Freiheitsgrade durch Vorberechnungen abzudecken, so dass zur Planungszeit mit reinen Translationen gearbeitet werden kann, um wieder die Effizienzvorteile des Basisversatzes aus Unterabschnitt 5.3.1 nutzen zu können. Die hier vorgestellten Verfahren wurden von Klaus Fischnaller in seiner Masterarbeit [23] implementiert.

Die für die Fahrtplanung gewählten Primitve decken unterschiedlich lange, geradlinige Pfade sowie Kurvenfahrten mit unterschiedlichen Längen und Radien ab. Sie beginnen alle mit derselben Startorientierung und enden fächerförmig (vgl. Abb. 7.14) mit fest definierten Orientierungen (in dieser Arbeit 0, ± 45 und ± 90 Grad). Die Definition der Primitive erfolgt komfortabel in einem eigens implementierten grafischen Designer, in dem einzelne Pfade entweder als Polynom definiert oder als Spline über Stützpunkte editiert werden können. Dabei muss die Geometrie der Pfade so gewählt werden, dass die Endpunkte der Primitive nicht nur die diskretisierten Orientierungen aufweisen, sondern sich auch auf virtuellen Gitterpunkten befinden. Nur so ist sichergestellt, dass nach einigen Konkatenationen Zirkelschlüsse entstehen können, wie sie in Abb. 7.15 zu erkennen sind. Anderenfalls wächst das Netz aus Primitiven sowie der Planungsgraph bei der Suche ins Unendliche. Sind die 2D-Pfade festgelegt, werden sie mit dem Robotermodell virtuell abgefahren und dabei die Swept-Volu-men der Bewegungen generiert. Alle Volumen erhalten eine individuelle SSV-IDs und werden zusammen in einer Voxel-Datenstrukur gespeichert. Somit lassen sich alle Primitive gleichzeitig mit nur einer Kollisionsprüfung gegenüber dem Umweltmodell auf Ausführbarkeit prüfen. Durch das Einbringen eines Versatzes können die vorberechneten Pfade effizient an beliebigen Positionen in der Umwelt evaluiert werden.



(a) Aufbau der acht fächerförmig angeordnete Bewegungsprimitive.



(b) Kombination von acht Bewegungsfächern mit unterschiedlicher Startorientierung.

Abb. 7.14.: Bewegungsprimitive des IMMP Roboters, die bei der Planung auf Kollisionsfreiheit geprüft werden.

Planer

Die erste Umsetzung eines Planers stützte sich auf die Algorithmen der SBPL (ARA*, Anytime D*, R*), welche die GPU-basierte Kollisionsdetektion nutzen. Da SPBL-Planer die Primitive jedoch serialisiert evaluieren, wurden sie so abgeändert, dass die Anfragen teilweise parallel ablaufen konnten, was die Effizienz drastisch steigerte. Dennoch ergaben Vergleiche mit einem reinen 2D Szenario und fünf Bewegungsprimitiven keine zufriedenstellenden Ergebnisse. Die Originalimplementierung führte dabei $64\,000$ 2D-Kollisionsprüfungen in $4{,}7\,\mathrm{s}$ durch, während die GPU für $12\,800$ 3D-Kollisionsprüfungen (Parallelisierung mit Faktor fünf) $25{,}37\,\mathrm{s}$ benötigte. Auch wenn sich dieses Verhältnis mit einer steigenden Anzahl bzw. längeren Primitiven verbessert, wurde der SBPL-Ansatz wieder verworfen. Als Alternative wurde ein eigener D*-Lite Planer entwickelt, der bessere Einsichten und mehr Freiheiten bei der Entwicklung erlaubte.

Probleme bei dynamischer Umwelt

Wie in der Taxonomie der Planungsverfahren beschrieben wurde, haben Planer, die mit Pfadabschnitten arbeiten, das Problem, bei neu auftretenden Hindernissen zu beurteilen, welche Teile des Planungsgraphen zu invalidieren sind. Um dieses Problem zu vermeiden, wurde die Expansion des Planungsgraphen von der eigentlichen Suche entkoppelt. Wird nun eine Kollision erkannt, können die betroffenen Abschnitte aus dem Graphen entfernt werden, womit eine spätere Suche nur auf konsistenten Daten stattfindet. Ähnlich dem A*-Algorithmus wird der Graph durch Expansionsschritte so lange erweitert,

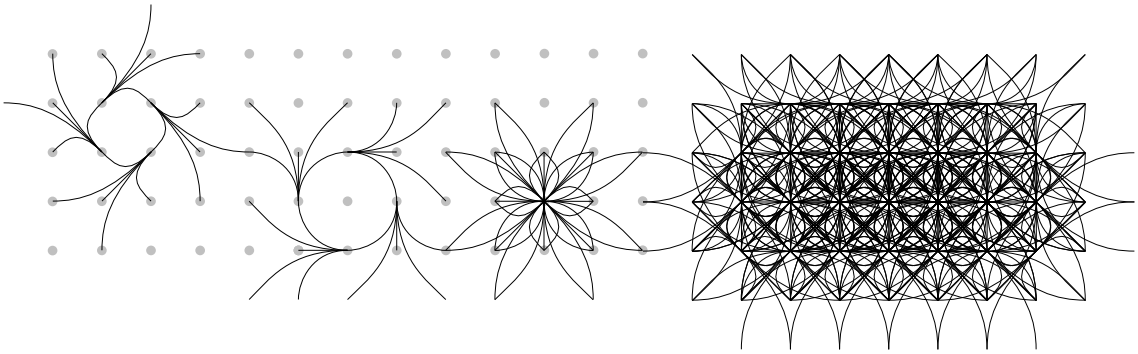


Abb. 7.15.: Bewegungspfade der Primitive für 45 bzw. 90 Grad Startorientierungen und ihren möglichen Konkatenationen.

bis das Ziel erreichbar ist. Allerdings wird danach nicht direkt der Pfad extrahiert, sondern über eine separate Dijkstra-Suche ermittelt. Somit ist sichergestellt, dass auch bei einer Änderung im Graphen durch neue Umweltinformationen noch ein gültiger Pfad gefunden werden kann. Der entstandene Algorithmus entspricht nun ansatzweise einem dynamischen Roadmap-Planer, in dem eine A*-Expansion für die Wegesuche eingesetzt wird. Umgesetzt wurde dies mit Hilfe der LEMON Graphenbibliothek [70]. Dieses Vorgehen löst nicht das eigentliche Problem, sorgt aber dafür, dass der Planungsgraph durch zusätzliche Berechnungen konsistente Lösungen generieren kann.

Somit besteht weiteres Optimierungspotential bei der Suche nach alternativen Plänen, nachdem neue Hindernisse auf dem Weg zum Ziel bekannt wurden. Diese blockieren in der diskretisierten Umgebung meist mehr als nur eine Kante entlang des Lösungspfades, da sie eine Ausdehnung über mehrere Zellen aufweisen. Da versperrte Knoten jedoch erst bei der Expansion des Graphen erkannt werden, tastet sich die Suche dennoch nur in direkter Nachbarschaft zum ursprünglichen Plan voran. Dies führt zu zahlreichen Fehlversuchen aufgrund desselben Hindernisses, bis ein valides Primitiv neben dem Hindernis gefunden wurde. Da das Verhalten keinen Fehler, sondern nur unnötige Rechenzeit bedeutet, wurde es in der Implementierung nicht umgangen. Eine mögliche Lösung wäre beispielsweise eine gezielte Abtastung der Region um das Hindernis, bevor die Berechnung alternativer Pläne immer wieder vom Ziel aus startet.

7.2.4. Manipulatorarm Planung mit Bewegungsprimitiven

Die vorgestellten Bewegungsprimitive (Rotationsvolumen bzw. kurze Plattformpfade) eignen sich gut für die GPU-beschleunigte Planung, da ihre vorberechneten Swept-Volumen durch eine translative Bewegung konkateniert werden können. Diese Translation kann im Voxelgitter sehr effizient durch einen einfachen Basisversatz umgesetzt werden.

Anders sieht es hingegen bei den Bewegungen eines Roboterarmes aus, die für eine Konkatenation nicht nur verschoben sondern auch rotiert werden müssten. Um die Rotationskonstellationen in den Primitiven zu codieren (wie bei der Plattformplanung), müsste eine zu grobe Diskretisierung der Winkel vorgenommen werden, um die Daten identifizierbar zu speichern. Diese Diskretisierung würde insbesondere in den Basisgelenken

einen zu hohen Diskretisierungsfehler im TCP des Roboters bedeuten, um allgemeingültige Bewegungen planen zu können. Auch das Arbeiten mit Polarkoordinaten wäre (abgesehen von den schwierigen und unregelmäßigen Voxelformen) keine Lösung, da eine Translation über einen Offset nicht möglich ist.

Aus diesen Gründen konnte in dieser Arbeit keine Lösung für die feingranulare Planung von Manipulatortrajektorien mittels Primitiven umgesetzt werden.

7.2.5. Manipulatorarm Planung mit samplingbasierten Verfahren

Wie bereits in der Taxonomie beschrieben, verschiebt sich der Berechnungsaufwand bei der Planung mit samplingbasierten Verfahren von der Kollisionsprüfung hin zur Voxelumwandlung der gesampelten Posen bzw. den abgetasteten Bewegungen zwischen den Samples. Diese Problematik kann durch die Verwendung von so genannten *Lazy evaluating* Planungsverfahren minimiert werden. Dabei wird während einer ersten Planungsphase auf die Kollisionsprüfung der Bewegungen verzichtet und während dem Sampling lediglich Start- und Endposen evaluiert. Erst wenn ein potentieller Pfad gefunden wurde, werden in einer zweiten Phase nur die benötigten Bewegungen zwischen den verwendeten Posen abgetastet und auf Kollisionsfreiheit geprüft. Werden dabei unausführbare Abschnitte detektiert, kann der Planer lokal nach Alternativen suchen. Dies reduziert den Aufwand der Voxelumwandlung eklatant und es können konkurrenzfähige Planungszeiten erreicht werden. Die Prüfung der Bewegungen erfolgt mit Hilfe von Swept-Volu-men, die die abgetastete Bewegung repräsentieren und die dann mit einer einzelnen Kollisionsprüfung bearbeitet werden können. Auf Grund der nicht existenten Abbildung von Ausführung zu Planungsraum können keine Roadmap Planer eingesetzt werden. In der Evaluation wurden die Planer LBKPIECE1 und SBL aus der OMPL erfolgreich getestet.

Ähnlich wie bei der Plattformplanung entsteht durch die Kollisionsprüfung der Bewegungen zwischen den Samples ein virtueller Korridor, in dem sich der Roboter während der Ausführung bewegt. Dieser kann mittels der SSV-ID in bis zu 250 Abschnitte unterteilt werden, um bei einer detektierten Kollision die verbleibende sichere Strecke bestimmen zu können. Durch eine Interaktion zwischen der Ausführung und dem Planer lässt sich der Startpunkt einer Neuplanung auf die aktuelle Pose des Roboters festlegen, um unterbrechungsfrei an die alte Trajektorie anknüpfen zu können.

7.2.6. Ganzkörperplanung

Sollen Pfade für mobile Manipulatoren (also die Kombination einer mobilen Plattform mit einem oder mehreren Armen) gefunden werden, lassen sich auch die im vorigen Abschnitt beschriebene samplingbasierte Planer einsetzen. Auf Grund der Vielzahl an Freiheitsgraden solcher Systeme muss das Sampling allerdings sehr effizient gestaltet werden. Hier bietet es sich an, die geplanten Freiheitsgrade durch eine Heuristik zu beschränken. RRT-Goalbias und RRT-Goalzoom sind dafür zwei Modifikationen des RRT Algorithmus, die Vahrenkamp et al. in [198] vorgestellt haben. Die erste Erweiterung verbessert die Konvergenz der Suche in Richtung des Zieles, während die Zweite die aktiven Freiheitsgrade bei der Planung mit der Annäherung zum Zielobjekt dynamisch

inkrementiert. Die generierten Pfade sind allerdings nicht optimal und weisen Unstetigkeiten auf, die in einem Nachbearbeitungsschritt geglättet werden müssen.

Ein Ansatz von Yang et al. aus [207] nutzt einen PRM Planer, dessen Planungsgraph möglichst klein gehalten wird, indem nur Knoten zum Graphen hinzugefügt werden, die den Arbeitsraum des Manipulators signifikant erhöhen. Der genutzte Manipulator wies jedoch lediglich fünf Freiheitsgrade auf und die Skalierbarkeit des Ansatzes ist fraglich.

Auch der Ansatz von Gochev et al. alterniert die Anzahl an geplanten Freiheitsgraden [89]. Zunächst wird versucht, das Ziel über eine Planung in einem niedrigdimensionalen Raum zu erreichen. Ist dies nicht möglich, schaltet der Planer in Regionen mit Kollisionen in einen hochdimensionalen Planungsraum.

Die genannten Verfahren sollen in Folgearbeiten praktisch in der Kombination mit der GPU Kollisionsprüfung evaluiert werden.

7.2.7. Greifplanung

Greifplanung gehört zu den essentiellen Fähigkeiten eines Serviceroboters, da diese unterschiedlichste Objekte manipulieren müssen. Gleichzeitig stellt die Greifplanung eine große Herausforderung für autonome Systeme dar, da vielfältige unterschiedliche Vorgaben erfüllt sein müssen, um einen stabilen Griff erfolgreich auszuführen. Menschen benötigen in der frühen Kindheit mehrere Jahre, um diese Fähigkeit zu erlangen. Allerdings nutzen sie umfangreiches Kontext-, Objekt- und Hintergrundwissen um über einen Griff zu entscheiden. Entsprechende Herangehensweisen existieren auch in der Robotik [199], sollen hier aber nicht vertieft werden. Im Gegensatz zu ihnen nutzt das hier vorgestellte Verfahren ausschließlich geometrische Berechnungen und fokussiert sich darauf, die Kontaktberechnung zwischen Hand und Objekt so weit zu beschleunigen, dass die Greifplanung online ablaufen kann.

Während Industrieroboter individuell auf die zu handhabenden Objekte abgestimmte Greifer verwenden, sind Serviceroboter meist mit Mehrzweckhänden ausgestattet. Ihre Flexibilität erhalten sie durch zahlreiche bewegliche Freiheitsgrade, welche alle koordiniert angesteuert werden müssen, um einen erfolgreichen Griff auszuführen. Um passende Gelenkwinkel zu finden, muss der Kontakt zwischen Hand und Objekt in komplexen Simulationen bestimmt und optimiert werden. Da Objekte weiterhin auf zahlreiche unterschiedliche Weisen gegriffen werden können, sind diese Simulationen zur Beurteilung der Alternativen zeitaufwendig.

Klassische Greifplaner [205] stützen sich daher auf vorberechnete Datenbanken, in denen passende Griffe für bekannte Objekte abgelegt sind. Die Erstellung solcher Datenbanken benötigt vollständige geometrische Modelle, was die Einsetzbarkeit auf eine begrenzte Menge an Objekten einschränkt.

Alternative Ansätze, die Greifhypothesen auch für unbekannte Objekte herleiten können, nutzen dafür Griffe, die für geometrische Grundobjekte oder Superquadriken definiert sind [74]. Diese Primitive werden dann an den, in Sensoraufnahmen sichtbaren, Ausschnitt des Zielobjektes angepasst [63]. Darüber hinaus ist es auch möglich, eine Menge von maximal großen Kugeln in das detektierte Objekt einzupassen [164] und diese als bekannte Greifkörper zu nutzen. Eine weitere Möglichkeit, Annahmen über die der

Kamera abgelegenen Objektseite zu treffen, ist die Spiegelung der Sensordaten an der Schattenkante der Aufnahme [48].

Die Qualität der beschriebenen Prozesse hängt klar von der Aufnahmeperspektive und somit vom sichtbaren Abschnitt des Objektes ab. Um die Daten zu vervollständigen ist ein Perspektivwechsel notwendig, der jedoch je nach Situation und Roboter mit großem Aufwand verbunden sein kann, z.B. wenn der Roboter erst um einen Tisch herumfahren muss, um das Objekt von der gegenüberliegenden Seite betrachten zu können.

Eine weitere Schwierigkeit, die aus dem kinematischen Aufbau klassischer Serviceroboter hervorgeht, ist die Hand-Augen-Kalibrierung: Zahlreiche Methoden des Greifens basieren auf einem einfachen Sense-Plan-Act-Zyklus, in dem ein Objekt einmalig detektiert und ein passender Griff geplant wird, der dann ohne weitere Anpassung ausgeführt wird. Somit ist eine exakte, extrinsische Kalibrierung der kinematischen Kette zwischen der Sensorik und dem Endeffektor nötig, welche je nach Roboterkonfiguration schwierig aufrechtzuerhalten ist.

Eine robustere Herangehensweise ist das biologisch motivierte Visual Servoing [92], bei dem das Zielobjekt mittels einer Kamera lokalisiert und der Endeffektor relativ dazu positioniert wird. In einem geschlossenen Regelungskreis wird dann der Abstand zwischen Objekt und Greifer minimiert, bis ein Griff möglich ist. Eigens für diesen Zweck sind einige Roboter mit Kameras in den Unterarmen ausgestattet. Dennoch verbleibt auch hier die Schwierigkeit der Greifplanung. Daher wird Visual Servoing oft mit einer haptischen Sensorik kombiniert [208], durch die sich die Finger schließen lassen, bis ein vordefinierter Druck erreicht wird. Dabei entsteht jedoch das Problem, dass leichtere Objekte durch den ersten Finger, der das Objekt berührt, verschoben werden. Abhängig von der Objektgeometrie kann bereits diese Verschiebung ein erfolgreiches Greifen verhindern, falls das Objekt beispielsweise kippt oder wegrollt.

Um dem entgegenzuwirken, wurde eine Methode entwickelt, die visuelle Exploration mittels einer *In-Hand-Kamera* nutzt, um Objektmodelle zu erzeugen. Diese ersetzen a priori Wissen über die Objektgeometrie, wodurch Griffe für zunächst unbekannte Objekte gefunden werden können. Hochparallele Algorithmen ermöglichen es, diese Greifhypothesen zu evaluieren, während sich die Hand noch um das Objekt bewegt. Die pro Finger individuell geplanten Bewegungen stellen sicher, dass alle Fingerglieder das Objekt gleichzeitig berühren, ohne es zu verschieben oder die Umwelt sonst zu verändern. Diese Technik ist weiterhin mit taktiler Sensorik oder kraftbasierter Regelung kombinierbar, um ausgeführte Griffe zusätzlich zu stabilisieren.

Technische Fortschritte in der 3D-Sensorik (siehe Abb. 4.1) erlauben es, die benötigten hochauflösenden Kameras direkt in den Roboterarm [88] oder auch in den Greifer zu integrieren. Durch GPU-Voxels ist es möglich, unterschiedliche Griffe sehr schnell zu evaluieren und zu optimieren und diesen Prozess schritthaltend auszuführen, während sich der Greifer bereits dem Objekt nähert.

Im Gegensatz zu Ansätzen wie [43, 139], die Oberflächenmodelle als Dreiecksnetze aus Sensordaten generieren, ist dieser Schritt hier nicht nötig, da wie in allen anderen Experimenten volumetrische Voxelmodelle zum Einsatz kommen.

Optimierungsproblem

Bei der Greifplanung müssen unterschiedliche und teilweise gegensätzliche Kriterien erfüllt werden, die nicht in linearem Zusammenhang stehen. Daher wird sie hier als mehrdimensionales Optimierungsproblem der Bewertungsfunktion f betrachtet, welche die folgenden Eingabedaten bewertet:

- $\vec{\Delta}_{xyz}$ und $\vec{\Delta}_{\alpha\beta\gamma}$: Geometrische Transformation zwischen Objekt und Hand (6 DOF). $\vec{\Delta} \in [\vec{\Delta}_{\text{min}}, \vec{\Delta}_{\text{max}}]$
- $\vec{\varphi}$: Gelenkwinkel aller N Finger. Bilden Vektor $\vec{\varphi} \in [\vec{\varphi}_{\text{min}}, \vec{\varphi}_{\text{max}}]$
- O: Objekt Geometrie: Wird als unveränderlich angenommen
- H: Hand Geometrie: Kein Freiheitsgrad, da die Bewegung durch $\vec{\varphi}$ geschieht

Folglich sucht man nach

$$\arg\max_{\vec{\Delta},\vec{\varphi}} \left(f\left(O, H, \vec{\Delta}, \vec{\varphi}\right)\right) \tag{7.4}$$

Um dieses Maximum analytisch zu berechnen, wäre eine Formel nötig, welche die komplexe physikalische Interaktion einer Mehrkörperkinematik der Hand H mit dem Objekt O abbildet. Die Berechnung eines solchen Systems, in einer für die Greifplanung ausreichenden Qualität, ist nicht tragbar. Auch kann der Konfigurationsraum nicht systematisch abgesucht werden, da seine Dimensionalität $>= 8$ zu hoch ist.

Daher wird in dieser Arbeit eine Partikelschwarmoptimierung (PSO) eingesetzt, die in Abschnitt 8.10.2 näher beschrieben ist. Diese optimiert die Untermenge $(\vec{\Delta}_{xyz}, \vec{\Delta}_{\alpha\beta\gamma})$ des Konfigurationsraumes, welche die Pose des Objektes beschreibt, während kontinuierlich die Gelenkwinkel $\vec{\varphi}$ der Finger mittels GPU Kollisionsprüfung bestimmt werden.

Die Bewertungsfunktion

$$f\left(O, H, \vec{\Delta}, \vec{\varphi}\right) = \left(V_{\text{col Handfläche}} + V_{\text{col Finger}}\right) \cdot \sum_{n\in[2,N]} (\varphi_n) \tag{7.5}$$

die von der PSO genutzt wird nutzt die folgenden Variablen:

- $V_{\text{col Handfläche}}$: Kontaktvolumen zwischen Handfläche und Objekt
- $V_{\text{col Finger}}$: Kontaktvolumen zwischen Fingern und Objekt
- $\vec{\varphi}$: Gelenkwinkel der Finger zum Kontaktzeitpunkt. Größere Winkel bedeuten weitere Schließung.

Die Funktion bewertet Griffe anhand des Kontaktvolumens zwischen Hand und Objekt und bevorzugt Griffe, bei denen die Finger weiter geschlossen sind.

Die Eingabedaten werden auf der GPU durch die Simulation mehrerer tausend Griffe g erzeugt. Das Ergebnis jedes simulierten Griffs besteht aus folgendem Tupel:

$$g\left(O, H, \vec{\Delta}\right) = \left(V_{\text{col Handfläche}}, V_{\text{col Finger}}, \vec{\varphi}\right) \tag{7.6}$$

Die Parameter, unter denen $f(g)$ sein Maximum erreicht, definieren den stabilsten Griff mit den Fingergelenkwinkel $\vec{\varphi}$ und der Objektpose $\vec{\Delta}$ relativ zur Handwurzel. Jede PSO Optimierungsiteration liefert bereits die Gelenkwinkel eines Griffes, bei dem alle Finger in Kontakt mit dem Objekt liegen und der somit theoretisch ausführbar wäre. Dies beschreibt die Charakteristik eines *Anytime Algorithmus*, die es erlaubt, den Optimierungsprozess jederzeit abzubrechen und dennoch valide Ergebnisse zu erhalten. Praktisch benötigt es jedoch einige Iterationen, um einen stabilen Griff zu erreichen.

Zusammenfassung

Mit Hilfe der voxelbasierten Kollisionsdtetektion kann die Planung von Griffen direkt auf Punktwolkendaten durchgeführt werden. Somit entfällt die Notwendigkeit für abstrakte Modelle und es lassen sich auch Griffe für unbekannte Objekte erzeugen und bewerten. Weitere Details der praktischen Umsetzung des Verfahrens finden sich in Abschnitt 8.10. Ebenso eine Evaluation, die zeigt, dass die Algorithmen schnell genug ablaufen, um während einer Anrückbewegung zu einem Objekt bereits gute Griffe zu finden. Die vorgestellte Greifplanung wurde in [8] veröffentlicht.

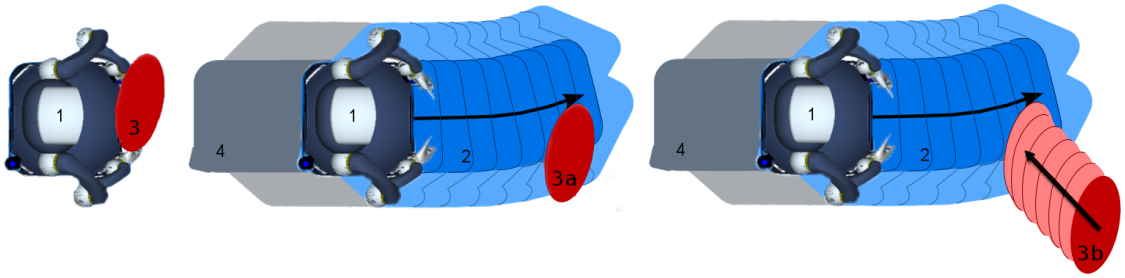
Eine Funktion, die zugunsten der Voxelumwandlung aufgegeben werden muss, ist die physikalische Modellierung der Greifkontakte, die bei Oberfächenmodellen durch ihre Normalen ermöglicht wird. Auch wenn dies in einigen Szenarien einen Nachteil darstellt, ergeben sich daraus jedoch keine generelle Probleme. Ebenso verringert sich die Genauigkeit mit der Kantenlänge der Voxel. Da Serviceroboter jedoch häufig über eine Impedanzregelung verfügen, lassen sich darüber die verlorenen Millimeter problemlos ausgleichen.

7.3. Fazit

Die in diesem Kapitel vorgestellten Verfahren ermöglichen es Robotern, in einer sensoriell erfassten, dynamischen Umwelt kollisionsfreie Bewegungen zu planen. Durch die Nutzung der parallelen Kollisionsprüfung erreichen die Planer dabei ein reaktives Verhalten, wie in der folgenden Evaluation gezeigt wird. Es konnten unterschiedliche Herangehensweisen aufgezeigt werden, mit denen sich der Aufwand zur Voxelumwandlung während der Planung minimieren lässt, um somit eine effiziente Implementierung zu erhalten. Insbesondere die Verwendung von vorberechneten Bewegungsprimitiven, die von rotierenden Bewegungskomponenten abstrahieren, erweisen sich als sehr gute Lösung für die Planung von mobilen Plattformen. Weiterhin wurden Lösungen für Roboterarme, sowie eine Greifplanung von Mehrfingergreifern vorgestellt, die die universelle Verwendbarkeit der Voxelverfahren unterstreichen. Zusammenfassend konnte dadurch die Forschungsfrage 4 positiv beantwortet werden, da sowohl etablierte Planungsansätze als auch neue Herangehensweisen erfolgreich mit parallelisierten Datenstrukturen kombinierbar sind.

8. Experimentelle Evaluation

In diesem Kapitel werden die im Laufe der Arbeit entwickelten Verfahren der Voxelmodellierung und der hochparallelen Kollisionsprüfung evaluiert und ihr praktischer Einsatz in sehr unterschiedlichen Anwendungen erprobt. Die Szenarien decken dabei drei Problemklassen ab, die in Abb. 8.1 dargestellt sind:



(a) Momentaufnahme (b) Ego-Swept-Volumen und Hindernis Momentaufnahme (c) Ego- und Hindernis Swept-Volumen

Abb. 8.1.: Die durchgeführten Experimente decken drei Problemklassen ab, die hier anhand von HoLLie (1) (Ansicht von Oben) dargestellt sind. Das Swept-Volumen der geplanten Trajektorie ist unterteilt in den bereits abgefahrenen Bereich (4, grau) und die ausstehende Bewegung (2, blau). Das Hindernis (3) ist in rot dargestellt.

1. Zunächst ist die Parallelisierbarkeit verschiedener Datenstrukturen und ihre Eignung zur Modellierung unterschiedlicher Entitäten der Bewegungsplanung zu prüfen. Hervorzuheben ist der GPU-optimierte Octree, der durch probabilistische Lastbalancierung alle bekannten Implementierungen aussticht. Aufbauend auf den Datenstrukturen wird eine **Kollisionsprüfung** von Momentaufnahmen umgesetzt. Der Erfolg wird anhand der Laufzeit der Verfahren gemessen und mit etablierten Algorithmen verglichen. Es soll mindestens eine Verarbeitungsrate erreicht werden, die der Bildrate aktueller 3D-Sensoren entspricht.

2. In einem zweiten Schritt wird die **Kollisionsvermeidung** beurteilt. Hierfür sind die geplanten Robotertrajektorien in Form von Swept-Volumen dargestellt, die während der Ausführung auf eindringende Hindernisse (Momentaufnahmen der Umwelt) hin überwacht werden. So entsteht ein überwachter Korridor, in dem sich der Roboter sicher bewegen kann. Herausforderungen liegen in der zeitlichen und örtlichen Identifizierbarkeit von Subvolumen bei der Kollisionsprüfung. Weiterhin wird die Planung mittels Bewegungsprimitiven untersucht, bei der Trajektorien für nichtholonome Plattformen aus Swept-Volu-men synthetisiert werden.

3. Im dritten Szenario werden schließlich nicht nur die Eigenbewegungen, sondern auch die Bewegungen dynamischer Hindernisse als Swept-Volu-men modelliert,

um eine **Kollisionsprädiktion** zu erlauben, damit kollisionsfreie Bahnen in dynamischen Umgebungen geplant werden können. Hierfür kommt eine voxelbasierte Bewegungssegmentierung und -prädiktion zum Einsatz. Zur Planung werden Distanzfelder (EDT) auf Voxelkarten definiert, über die ein variabler Sicherheitsabstand (3D-Potentialfeld) um Hindernisse herum gewahrt werden kann. Somit ist neben einer exakten Kollisionsberechnung auch die Distanzberechnung zwischen zwei Modellen umgesetzt, die für zahlreiche Planungsalgorithmen benötigt wird.

8.1. CUDA Laufzeitparametrierung

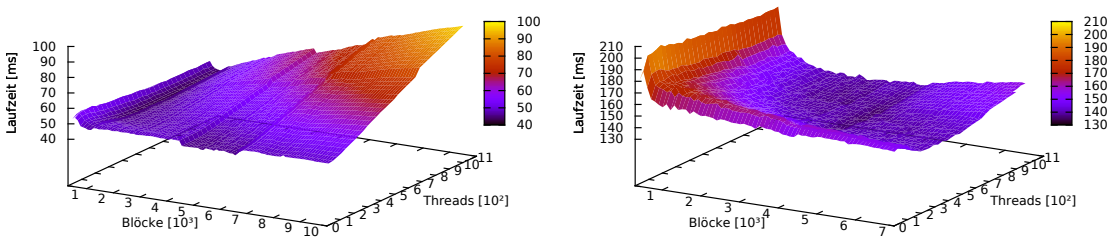
Bevor auf komplexe Anwendungen eingegangen wird, sollen zunächst noch Grundlagen der CUDA Laufzeitparametrierung evaluiert werden, die allgemein anwendbar sind. Bereits in Kapitel 3 wurde dargestellt, dass für die quantitative Beurteilung von CUDA Algorithmen zwischen rechen- und speicherintensiven Programmen unterschieden werden muss. Theoretisch lassen sich beide Klassen durch einen Vergleich mit der rechnerisch erreichbaren Rechenleistung der verwendeten GPU (GFLOPS / s) bzw. deren Speicherbandbreite (GiB / s) beurteilen. Für die Evaluation wurde in dieser Arbeit eine Titan-GPU verwendet (siehe Unterabschnitt A.7.4), deren Prozessor bis zu 4,709 GFLOPS erreicht, während der theoretische Speicherdurchsatz bei 288,4 GiB/s liegt. Da die erreichbare Beschleunigung durch die Parallelisierung bei komplexeren Algorithmen aber von einem Zusammenspiel vieler Faktoren abhängt, liegen die praktisch erreichbaren Werte weit unterhalb der Spezifikation. Daher werden in dieser Evaluation bevorzugt die Laufzeiten spezifischer Problemlösungen betrachtet.

Einer der einflussreichen Faktoren bei der Optimierung ist der Grad der Parallelisierung, der sich in CUDA über zwei Parameter steuern lässt: *Anzahl an Blöcken* und *Anzahl an Threads pro Block* (vgl. Unterabschnitt 3.2.2). Je nach Aufgabe bedeutet ein hoher Parallelisierungsgrad jedoch auch mehr Aufwand zur Arbeitsverteilung, Synchronisierung und Zusammenführung der Ergebnisse. Somit ergibt sich ein zweidimensionales Optimierungsproblem, bei dem die Anzahl an Blöcken zwischen 1 und $2^{31} - 1$ und die Anzahl der Threads pro Block zwischen 1 und 1024 zu wählen ist. Nvidia bietet mit dem *Occupancy Calculator*[1] ein Werkzeug an, um im Vorfeld eine passende Parametrierung zu berechnen und die Auslastung (*Occupancy*) der GPU zu verbessern. Allerdings sollten die berechneten Werte nicht generell als optimale Parameter angesehen werden, da diese von weitaus mehr Faktoren abhängen, welche zur Laufzeit bspw. mit dem *Visual Profiler*[2] tiefgehend analysierbar sind.

Aus diesen Gründen wurden beispielsweise für das Traversieren des Octrees umfangreiche empirische Versuchsreihen auf der Zielhardware ausgeführt, um das Laufzeitverhalten bei unterschiedlicher Parallelisierung zu bestimmen. Die Ergebnisse aus Abb. 8.2 verdeutlichen, wie sehr optimale Parameter von der Problemgröße abhängen (4096 Blöcke mit je 32 Threads bei 3 Mio. Punkten bzw. 4096 Blöcke mit je 512 Threads bei 13 Mio. Punkten). Da nicht für jede Problemgröße eine solche Studie durchgeführt werden konnte, wurde eine lineare Abhängigkeit der Threads pro Block mit der Punkteanzahl

[1] `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`

[2] `https://developer.nvidia.com/nvidia-visual-profiler`



(a) Octree aus $3 \cdot 10^6$ Punkten

(b) Octree aus $13 \cdot 10^6$ Punkten

Abb. 8.2.: Empirische Ermittlung der optimal Anzahl an Blöcken und Threads für den Octree-Aufbau anhand der benötigten Laufzeit in Millisekunden.

angenommen, deren Stützstellen die ermittelten Optima sind. Diese Wahl erweist sich als valide Annahme, wie das Diagramm in Abb. 8.4 belegt.

Die Laufzeitparameter weiterer Klassen von Algorithmen, die in dieser Arbeit eingesetzt werden, wurden ähnlich ermittelt.

8.2. Voxelkarte

Da die Erstellung und Verarbeitung von Voxelkarten keine komplexen Berechnungen erfordert, und somit auch keine Datenabhängigkeiten vorliegen, ist diese Datenstruktur hauptsächlich durch die Speicherbandbreite der Hardware begrenzt. Entsprechend knapp kann die Evaluation ausfallen.

Liegen die Ausgangsdaten bereits im Speicher der GPU vor, ergeben Messungen zum Eintragen von Sensorpunktwolken folgendes Bild: Pro Millisekunde können ca. 3 Mio. Messpunkte aus $\mathbb{R}^3$ in probabilistische Voxel eingetragen werden. Im Testszenario waren die Punkte dabei so verteilt, dass pro Punkt ein Voxel zu aktualisieren ist, also maximal viele Speicherzugriffe nötig waren. Die Laufzeit skaliert linear mit der Anzahl an Punkten, wobei diese maximal parallel verarbeitet werden, während die Dimensionen der Zielkarte dagegen keine Auswirkung auf die Laufzeit aufweisen.

Eine Kollisionsprüfung verarbeitet innerhalb einer Millisekunde zwei Voxelkarten aus je knapp 5 Mio. probabilistischen oder 7,5 Mio. binären Voxeln zu einer einfachen binären Kollisionsaussage. Hierbei kann der Speicherzugriff den Bus voll ausnutzen, da parallel ablaufende Threads sequentielle Daten verarbeiten.

Da eine Kinect-Kamera bei voller Bildrate von 30 FPS ca. 9,2 Mio. Messpunkte pro Sekunde erzeugt, wäre es bei ausschließlicher Betrachtung der Voxelumwandlung möglich, die Daten von über 300 Kameras simultan in eine Voxelkarte einzutragen. Auf Grund von hostseitigen Einschränkungen ist die Grenze praktisch jedoch bereits bei vier Kinects erreicht.

Sind bei gleicher Verarbeitungsrate Voxelkarten auf Kollision zu prüfen, dürfen diese unter Vernachlässigung weiterer Verarbeitungsschritte folglich $5 \cdot 10^6\,\mathrm{Voxel/ms} \cdot 33\,\mathrm{ms} =$

165 Mio.Voxel pro Karte aufweisen. Praktisch evaluiert wurden Szenarien mit komplexen Verarbeitungsketten auf 128 Mio. Voxeln. Dies entspricht bei 2 cm Voxelauflösung einem abgedeckten Volumen von bspw. 20 m × 20 m × 2,5 m, das in ausreichender Geschwindigkeit auf Kollisionen prüfbar ist. Der Speicherverbrauch für zwei probabilisitsche Karten liegt hierbei bei 1,26 GiB. Selbst große Innenräume stellen folglich kein Problem dar.

Laufzeitvergleiche gegenüber dem implementierten Octree und anderen CPU- und GPU-basierten Verfahren finden sich im nächsten Abschnitt.

Mehrstufiger Kollisionscheck

Um die praktischen Auswirkungen eines hierarchischen Kollisionschecks auf Voxelkarten unterschiedlicher Auflösungen (vgl. Unterabschnitt 5.3.2) besser beurteilen zu können, soll hier der erreichbare Laufzeitvorteil betrachtet werden. Dafür wurde in vier unterschiedlichen Versuchen eine Voxelliste mit einer Voxelkarte geschnitten. Das Szenario entstammt der Planung mit Rotationsvolumen, auf das weiter unten eingegangen wird. Die Ergebnisse finden sich in Tab. 8.1, wobei ausschließlich die reinen Laufzeiten der Kollisionsprüfung gemessen wurden. Ausgehend von vorberechneten Listen ergibt sich bei der Verwendung einer acht mal gröberen Auflösung nur ein Zehntel der Laufzeit, wenn keine Kollisionen auftreten.

Voxel-kantenlänge [cm]	Größe Umweltkarte [MVoxel]	Länge Roboter-Voxelliste [Voxel]	Kollisionscheck Laufzeit [ms]	Kollisionscheck Durchsatz [Voxel/ms]
1	200	1 889 192	0,696	2 714 356
2	25	250 818	0,263	953 681
4	3,125	33 789	0,089	379 652
8	0,391	4820	0,067	71 940

Tab. 8.1.: Durchschnittlicher Zeitaufwand für Kollisionschecks bei varierender Auflösung.

8.3. Octree

Wesentlich umfangreicher fällt die Evaluation des implementierten Octrees aus, da dieser komplexere Verarbeitungsketten beinhaltet, um die Problematik der Parallelisierung dynamischer Datenstrukturen anzugehen. Die folgenden Abschnitte geben daher einen detaillierten Einblick in die Laufzeiten der wichtigsten Funktionen und vergleichen diese mit einer CPU-Implementierung eines Octrees und eines kd-Trees.

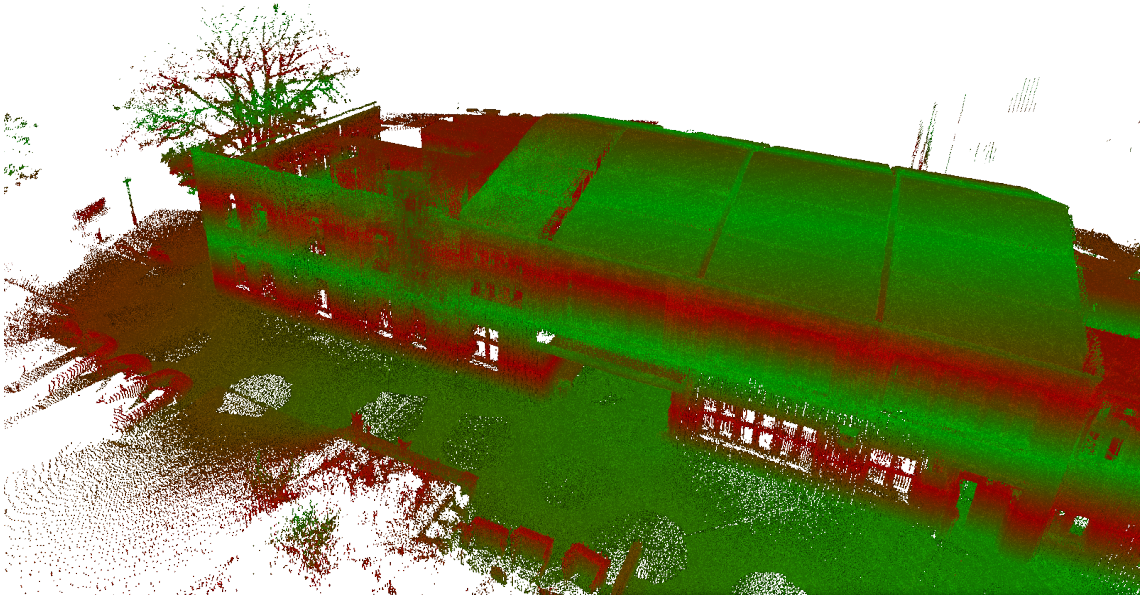


Abb. 8.3.: Ausschnitt einer Punktwolke mit $40,4 \cdot 10^6$ Punkten, die ein Volumen von $101{,}5\,\mathrm{m} \times 97{,}2\,\mathrm{m} \times 24{,}7\,\mathrm{m}$ abdecken. Um diese in $2\,\mathrm{cm}$-Auflösung in einem Octree zu speichern sind lediglich $300\,\mathrm{MiB}$ an GPU RAM nötig, während eine Voxelkarte hingegen mindestens $28\,\mathrm{GiB}$ benötigt. Die Punktwolke des FZI Gebäudes wurde von Jan Oberländer erstellt.

8.3.1. Aufbau eines Octrees

Zunächst soll der Aufbau eines neuen Baumes mit 15 Ebenen aus einer Punktwolke betrachtet werden. Um eine repräsentative Laufzeitmessung in Abhängigkeit der Punkteanzahl zu erhalten, muss gewährleistet sein, dass die Punktwolke pro Voxel nur einen Punkt enthält. Dies wurde durch eine Diskretisierung mittels eines Voxelfilters erreicht, dessen Voxelgröße direkt die Dichte der entstehenden Punktwolke bestimmt. Für die folgenden Experimente variiert die Voxelgröße zwischen $1{,}7\,\mathrm{cm}$ und $30\,\mathrm{cm}$.

Wie Abb. 8.4 zeigt, kann durch die in Abschnitt 5.5.2 beschriebenen Techniken des adaptiven Lastausgleichs ein nahezu lineares Laufzeitwachstum bei steigender Größe der Eingabedaten erreicht werden. Die Messungen zeigen den Mittelwert aus 100 Durchläufen mit einem probabilistischen Octree (die Zeiten eines deterministischen Octrees unterscheiden sich nur marginal und sind daher nicht angegeben). Den größten Aufwand verursacht das Sortieren der Eingabedaten, sowie der ebenenweise Aufbau der Octreestrukturen. Bei Kopieren + Diskretisieren entfallen ca. 87%der Laufzeit auf den reinen Datentransfer zur GPU, wogegen das Berechnen der Morton-Codes sehr schnell abläuft. Für das Sortieren kam die Thrust-Implementierung des Radix-Sort zum Einsatz, die einen Durchsatz von 310 Mio. Morton-Codes pro Sekunde erreicht. Das Aufbauen der Ebenen beinhaltet das Setzen der Kindzeiger sowie des Knotenzustandes. Hier entfällt erwartungsgemäß die meiste Arbeit auf die Blattknoten in Ebene 0. Das anschließende Herstellen der Baum-Invariante fasst das Ab- und Aufsteigen im Baum bei probabilistischem Lastausgleich zusammen, wobei der Ausgleich ab ca. 3 Mio. Punkten gut skaliert (bei kleineren Punktemengen sollte ein Octree mit weniger Ebenen genutzt werden).

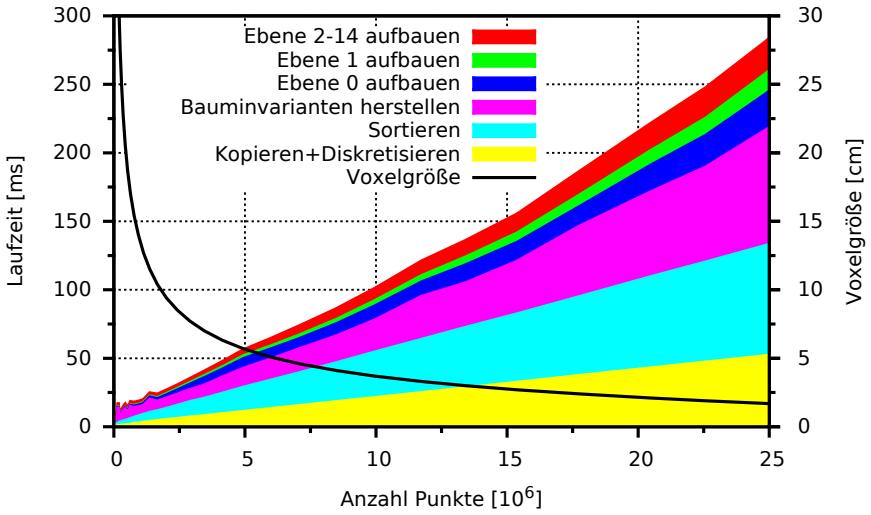


Abb. 8.4.: Nahezu lineares Laufzeitverhalten für den Aufbau eines Octrees aus unterschiedlich großen Punktwolken (bis zu 25 Mio. Punkte) mit Aufschlüsselung der Algorithmenbestandteile. Diagramm aus [22].

Der ermittelte Datendurchsatz reicht folglich einerseits aus, um einen Octree aus den Daten von bis zu drei Kinect-Kameras mit voller Bildrate komplett neu zu konstruieren, aber auch, um extrem detaillierte Punktwolken aus 25 Mio. Messpunkten mehrmals pro Sekunde zu verarbeiten.

8.3.2. Kollisionsprüfung

Zur Evaluierung der Leistungsfähigkeit der Kollisionsprüfung wurden praxisnahe Versuche der Bewegungsplanung am Modell des HoLLiE-Roboters durchgeführt. Dafür mussten zufallsgenerierte Swept-Volu-men von Bewegungen auf Überschneidung mit einem Ausschnitt der bereits gezeigten, statischen Punktwolke des FZI Gebäudes geprüft werden.

Octree ∩ Octree

Zunächst wurden beide Modelle in probabilistische Octrees eingefügt und diese mittels der lastbalancierten Tiefensuche (siehe Unterabschnitt 6.2.5) miteinander geschnitten. Die hierarchische Struktur des Baumes erlaubt eine zielgerichtete Abarbeitung der Octrees für eine effiziente Kollisionsprüfung, wobei sich die kompakte Datenspeicherung zusätzlich positiv auf die Laufzeit auswirkt. Über eine Einschränkung der Suchtiefe lässt sich dabei die Auflösung und somit die erforderliche Berechnungszeit beschränken. Im Laufzeitdiagramm aus Abb. 8.5 verursacht der Lastausgleich bis zu 50%des Berechnungsaufwandes, was durch die hohe Lokalität der Bewegungsdaten bedingt ist. Dennoch sind durch die Heuristik zur Gruppierung vergleichbarer Arbeitselemente nur maximal sieben Lastausgleichsschritte nötig. Die Berechnungszeit steigt mit zunehmender

Suchtiefe nahezu linear an, was die Eignung der implementierten Parallelisierungsstrategien und des Lastausgleiches unterstreicht.

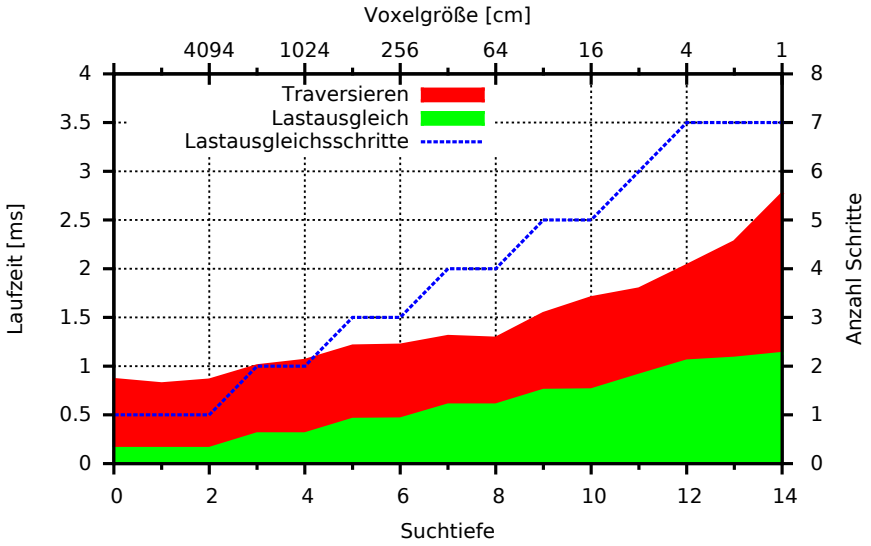


Abb. 8.5.: Laufzeit der Kollisionsprüfung zwischen zwei Octrees mit anteiligem Aufwand für den Lastausgleich. Abhängig von der maximalen Suchtiefe werden Laufzeiten zwischen 1 und 3 ms erreicht um bis zu 0,5 Mio. Voxel zu prüfen. Diagramm aus [22].

Octree ∩ Voxelkarte

In Szenarien, in denen ein schneller, wahlfreier Zugriff auf eine dichte Voxelmenge benötigt wird, kann ein Octree auch gegenüber einer Voxelkarte auf Kollision geprüft werden. Dabei werden belegte Einträge des Octrees gezielt in der Voxelkarte nachgeschlagen. Abb. 8.6 vergleicht die Laufzeiten unterschiedlicher Kombinationen von Datenstrukturen. Hier zeigt sich, dass der Einsatz einer Voxelkarte nur in Szenarien mit einer kleinen Menge an Voxeln (großer Voxelkantenlänge) gerechtfertigt ist, da die Kombination aus zwei Octrees bei umfangreicheren Datenmengen um bis zu einer Größenordnung schneller auszuwerten ist. Da sich die Voxelmenge mit jeder Halbierung der Voxelkantenlänge verachtfacht, ist die Nutzung mindestens eines Octree gegenüber zwei Voxelkarten fast immer überlegen. Voxelkantenlängen unter sieben Zentimetern konnten nicht evaluiert werden, da die resultierende Voxelmenge mit der 32 Bit Adressierung der Voxelkarte nicht darstellbar ist. Weitere Ergebnisse sind in Tab. 8.2 zu finden. Die bewerteten Messungen berücksichtigen jedoch nicht den Aufwand für das Aufbauen oder Modifizieren des Octrees. Wie im nächsten Abschnitt beschrieben, hat dies jedoch gerade in dynamischen Szenen einen großen Einfluss auf die Laufzeiten und muss daher bei der Auswahl einer passenden Datenstruktur beachten werden.

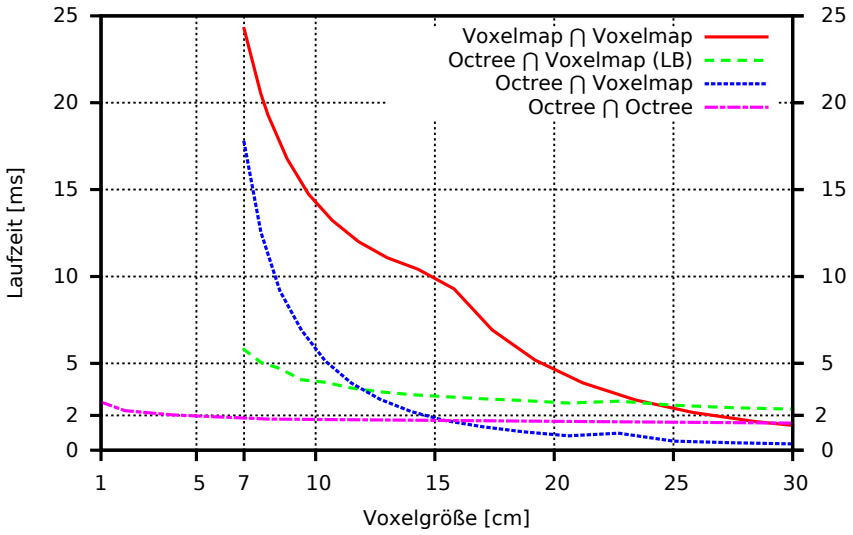


Abb. 8.6.: Laufzeiten der Kollisionsprüfung bei unterschiedlichen Kombinationen von Datentypen. Diagramm aus [22].

Aufbau aus Sensordaten

In der Praxis ist der Aufbau eines Octrees aus Sensordaten ein relevanter Benchmark. Daher wurden für diesen Test die Punktwolken einer bewegten Innenraumszene aufgezeichnet, wobei die Daten von einer Kinect-Kamera stammten, die mittels einer Schwenk-Neige-Einrichtung kontrolliert gedreht wurde. Einzelne Aufnahmen bestanden dabei aus $200\,000$ bis $300\,000$ Messpunkten, die kontinuierlich in einen Octree eingefügt wurden. Gleichzeitig wurde der Freiraum mittels Raycasting bestimmt (siehe Unterabschnitt 4.3.1). Da hierbei für die Freiraumvoxel dieselbe Auflösung gewählt wurde, überwiegen diese die belegten Voxel um ein 6 bis 107-faches. Dies zeigt sich auch im Laufzeitdiagramm aus Abb. 8.7, in dem das Sortieren der Freiraumvoxel den größten Anteil der Berechnungszeit einnimmt und zusammen mit dem Berechnen und Einfügen 80% der Laufzeit ausmacht. Die Messungen belegen, dass der Octree bei einer Voxelgröße von $3\,\mathrm{cm}$ in der Lage ist, die Punktwolken einer Kinect-Kamera mit $25\,\mathrm{Hz}$ zu verarbeiten. Wird die Auflösung auf $1\,\mathrm{cm}$ erhöht, liegt der erreichte Durchsatz noch immer bei $133\,\mathrm{MVoxel/s}$. Eine signifikante Laufzeitersparnis ergibt sich durch die Berechnung des Freiraumes mit einer geringeren Auflösung im Vergleich zu den Hindernisvoxeln: So ist bei Verdopplung der Freiraumvoxelgröße weniger als 50% der Gesamtlaufzeit nötig. Durch den kompletten Verzicht lässt sich der Berechnungsaufwand sogar um Faktor fünf reduzieren, wodurch eine simultane Auswertung von vier Kinect-Kameras möglich wird.

Vergleich mit anderen Arbeiten

Leider sind nur wenige andere Softwarebibliotheken verfügbar, die nicht mit Oberflächennetzen arbeiten und daher für einen Benchmark herangezogen werden können. Des Weiteren fällt der Vergleich aufgrund einer fehlenden, etablierten Metrik schwer, soll aber dennoch unternommen werden. Als Datengrundlage dient dafür der Vergleich zwischen

Hindernis Seiten-länge [Voxel]	⌀ Anzahl Kollisionen [Voxel]	Octree ∩ Octree [ms]	Octree ∩ Voxelliste [ms]	Voxelkarte ∩ Voxelkarte [ms]	Octree ∩ Voxelkarte [ms]
2	0,01	1,46	0,10	15,31	4,21
4	0,04	1,46	0,10	15,31	4,21
8	0,44	1,46	0,10	15,31	4,20
16	5,25	1,47	0,11	15,31	4,21
32	37,60	1,44	0,11	15,32	4,28
64	217,03	1,65	0,15	15,32	4,36
128	1862,98	1,88	0,41	15,32	4,71
187	6781,29	1,91	1,04	15,34	5,61

Tab. 8.2.: Median-Laufzeiten über 100 Kollisionsprüfungen zwischen der FZI Gebäudekarte (150 684 belegte Voxel bei einer Kartengröße von $884 \times 1004 \times 187$ Voxeln) und einem zufällig platzierten Würfel aus bis zu 6,5 Mio Voxeln. Publiziert in [4].

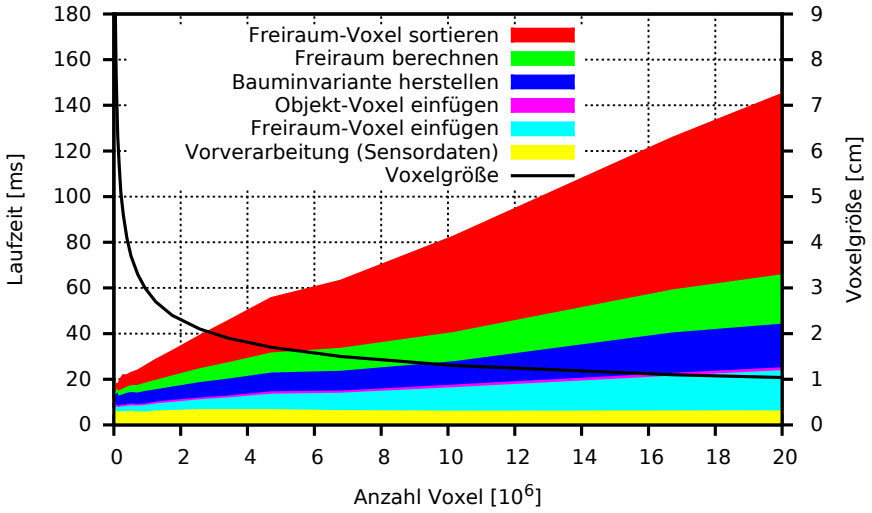


Abb. 8.7.: Laufzeit zum Aufbau eines Octrees aus Sensordaten inklusive Freiraumberechnung bei variierender Auflösung. Diagramm aus [22].

CPU- und GPU-basierten Kollisionsprüfungsverfahren von Schauer et al. aus dem Jahr 2016 [180]. Hier wird zum einen ein mittels OpenMP parallelisierter *k*d-Tree vorgestellt, der genutzt wird, um in einem definierten Suchradius um eine Eingabekoordinate einer Punktwolke nach Punkten einer zweiten Punktwolke zu suchen. Zum anderen ein GPU RGD-Ansatz, der die Punktwolke der Umgebung in einem Vorverarbeitungsschritt zunächst grob in Voxel diskretisiert. Im Unterschied zur vorliegenden Arbeit werden dabei jedoch die zugehörigen Punkte in jedem Voxel gespeichert. In einem zweiten Schritt können dann gezielt nur die Punkte derjenigen Voxel auf ihre Distanz zu Egomodellpunkten überprüft werden, die in kollidierenden Voxeln liegen.

	Voxelkarte	Octree Kollisionsprüfung		
Anzahl Einträge [Knoten]	**Kollisions-prüfung [ms]**	**Gesamt-aufwand [ms]**	**Balancierungs-aufwand [ms]**	**Balancierungs-schritte**
32 K	0,55	1,52	0,28	2
256 K	0,13	2,16	0,40	2
2 M	0,55	3,22	0,41	3
16 M	3,86	5,13	0,55	3
128 M	17,59	21,05	0,57	4

Tab. 8.3.: Median-Laufzeiten über 100 Kollisionsprüfungen zwischen zwei voll ausgefüllten Octrees bzw. zwei voll belegten Voxelkarten bei unterschiedlichen Datenmengen. Durch die gleichmäßige Verteilung der Daten sind nur sehr wenige Balancierungsschritte im Octree nötig.

Als Metrik für den Vergleich soll der Durchsatz an Voxeln bzw. Punkten pro Millisekunde herangezogen werden. Dieser ergibt sich aus der addierten Menge an Eingabedaten (Umwelt + Egomodell) die mit den durchgeführten Kollisionsprüfungen multipliziert werden. Verglichen werden Szenarien mit vergleichbaren Umgebungsbedingungen mit der letzten Zeile aus Tab. 8.2. Schauer et al. evaluieren die Algorithmen auf unterschiedlicher Hardware. Für den hier gezogenen Vergleich werden die Messungen mit einer GTX 980 GPU verwendet, die eine leicht höhere Leistung aufweist, als die in dieser Arbeit eingesetzte Titan GPU. Die CPU Messungen basieren auf einer Intel Xeon Workstation, die über acht Kerne (16 virtuelle hyperthrading Kerne) und 32 GiB RAM verfügt. Die Ergebnisse finden sich in Tab. 8.4. Alle Zeiten beziffern exklusiv die Kollisionsprüfung und beinhalten nicht den Aufbau der verwendeten Datenstrukturen.

Die Ergebnisse der Vergleichskandidaten unterscheiden sich erheblich innerhalb der untersuchten Szenarien, obwohl diese ähnliche Umgebungen abbilden (Fahrten durch tunnelartige Strukturen). Als Ursache geben die Autoren unterschiedliche Dichteverteilungen in den Datensätze an. Es zeigt sich, dass die verglichenen Verfahren auf zwei der Datensätze (El Teniente, Tunnel) einen höheren Gesamtdurchsatz von Punkten bzw. Voxeln als GPU-Voxels erreichen. Betrachtet man jedoch den Durchsatz der erkannten Kollisionen pro Millisekunde, liegt GPU-Voxels in realen Szenarien um mehr als eine Größenordnung über den Vergleichskandidaten. Gleichzeitig benötigen diese einen wesentlich zeitaufwendigeren Aufbau ihrer Datenstrukturen, was sie für eine schritthaltende Datenauswertung ungeeignet macht. Der Aufbau des *k*d-Trees aus 16 Mio. Punkten wurde bspw. in der vorausgegangenen Arbeit [75] mit 9 Sekunden beziffert und ist damit auch um beinahe zwei Größenordnungen aufwendiger, als die Konstruktion des vorgestellten GPU Octrees. Auf Datengrößen, wie sie bei der Verarbeitung von typischen 3D Kameras auftreten, liegt der erreichbare Datendurchsatz beider Verfahren ebenfalls weit unterhalb der Leistung der hier implementierten Datenstrukturen.

Datensatz	Umwelt [Einträge]	Egomodell [Einträge]	Verfahren	Durchsatz [Einträge/ms]	Durchsatz [Koll./ms]
Hannover	$55{,}87 \cdot 10^6$	$0{,}21 \cdot 10^6$	CPU *k*d-Tree	$0{,}77 \cdot 10^6$	2,27
			GPU RGD	$1{,}34 \cdot 10^6$	3,56
Wolfsburg	$350{,}11 \cdot 10^6$	$0{,}43 \cdot 10^6$	CPU *k*d-Tree	$0{,}61 \cdot 10^6$	111,02
			GPU RGD	$1{,}17 \cdot 10^6$	217,41
Tunnel	$18{,}92 \cdot 10^6$	$0{,}03 \cdot 10^6$	CPU *k*d-Tree	$22{,}27 \cdot 10^6$	162,72
			GPU RGD	$3{,}34 \cdot 10^6$	15,35
El Teniente	$806{,}18 \cdot 10^6$	$0{,}10 \cdot 10^6$	CPU *k*d-Tree	$75{,}51 \cdot 10^6$	195,70
			GPU RGD	$46{,}28 \cdot 10^6$	117,42
Planung	$6{,}54 \cdot 10^6$	$0{,}15 \cdot 10^6$	GPU 2x Octree	$3{,}50 \cdot 10^6$	3550,41
			GPU Oct.+Liste	$6{,}43 \cdot 10^6$	6520,47
Maximum	$128{,}00 \cdot 10^6$	$128{,}00 \cdot 10^6$	GPU 2x Octree	$14{,}55 \cdot 10^6$	$14,55 \cdot 10^6$

Tab. 8.4.: Vergleich des Berechnungsdurchsatzes der Kollisionsprüfung von GPU-Voxels (letzte zwei Datensätze) mit zwei anderen Verfahren aus [180].

Vergleich mit der *OctoMap*

Ein weiterer Vergleich soll mit der in der Robotik weit verbreiteten Octree-Implementierung *OctoMap* von Hornung et al. [104] gezogen werden. Diese CPU-basierte Software verfügt über einen ähnlichen Funktionsumfang, wie die hier vorgestellte GPU Variante, weshalb sie für einen Vergleich herangezogen werden soll. Hierfür wurde der im vorigen Abschnitt beschriebene Datenstrom einer Kinect-Kamera verwendet und mittels der Funktion `insertPointCloud()` in die OctoMap eingefügt. Zur Performanzsteigerung waren die Optionen `lazy_eval` und `discretize` aktiviert.

Der Laufzeitvergleich als doppelt-logarithmische Darstellung in Abb. 8.8 bescheinigt der GPU-Implementierung eine Beschleunigung um bis zu zwei Größenordnungen, verglichen mit der OctoMap. Bei gleichzeitiger Berechnung des Freiraums und unter Einbeziehung des Datentransfers auf die GPU (vgl. [91]) ist die probabilistische Version bis zu 80 mal, die deterministische bis zu 170 mal schneller. Weiterhin ist ersichtlich, dass der GPU-Octree besser mit der Menge der Voxel skaliert.

Vergleicht man den Speicherverbrauch der Datenstrukturen, so fällt dieser bei der GPU-Implementierung um den Faktor 3,4 niedriger aus, als bei der Octomap. Dieses sehr gute Verhältnis der benötigten Bytes pro Punkt wird durch den Verzicht auf acht separate Kindzeiger (OctoMap) zugunsten von nur einem Zeiger (GPU-Octree) auf acht gleichzeitig zu allozierende Kindknoten erreicht. Selbst bei einer Vervierfachung der Nutzdaten in den Blattknoten wäre der GPU-Octree noch immer 25% speichereffizienter als die OctoMap-Implementierung.

Zusammengefasst ist die GPU-Implentierung der OctoMap in den Punkten Berechnungsgeschwindigkeit und Speicherverbrauch weit überlegen. Es soll jedoch angemerkt werden, dass die OctoMap über eine Templatesierung beliebige Nutzdaten in ihren Knoten

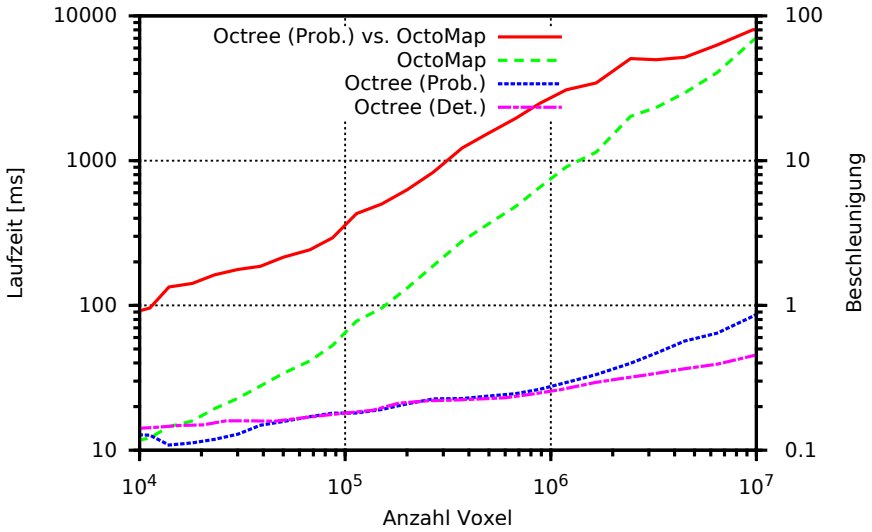


Abb. 8.8.: Vergleich der Laufzeiten zum Einfügen von Daten in den GPU-Octree und die CPU-Implementierung OctoMap (doppelt logarithmische Darstellung). Die GPU-Variante ist gegenüber OctoMap um bis zu zwei Größenordnungen schneller. Diagramm aus [22].

erlaubt, wogegen die GPU Variante auf die zwei verfügbaren Voxeltypen beschränkt ist. Eine Erweiterung um neue Datentypen ist hier aufgrund besonderer Optimierungen nur mit großem Aufwand möglich.

Fazit

Dynamische Datenstrukturen, deren Speicherverbrauch sich inkrementell ändert, eignen sich prinzipiell schlecht für eine Umsetzung auf der GPU. Auch die ungleichmäßigen und a priori unbekannten Traversierungsdistanzen innerhalb des Baumes lassen sich nicht kanonisch zur parallelen Ausführung aufteilen. Dennoch übertrifft der realisierte Octree in seiner Leistung bei der geometrischen Modellierung von Volumendaten den Stand der Technik. Wie die Laufzeitmessungen zeigen, konnte eine effiziente Implementierung umgesetzt werden, indem GPU-typische Programmierparadigmen eingehalten wurden: Inkrementelle Speicherallokationen konnten durch vorgeschaltete Zähloperationen zusammengefasst werden, während einer Speicherfragmentierung durch zyklischen Neuaufbau der Datenstruktur begegnet wird. Eine Strategie zur systematischen, leichten Überallokation von GPU-Speicher kaschiert die langsame Speicherverwaltung der GPU bei dennoch beinahe linearem Speicherzuwachs durch neue belegte Voxel. Gleichzeitig konnte der Speicherverbrauch minimiert werden, indem für je acht Kindknoten nur ein Zeiger verwendet wird. Weiterhin sorgt eine heuristische Lastbalancierung auch bei unterschiedlich aufwendigen Aufgabenfragmenten für eine gleichmäßig hohe Auslastung der GPU.

Durch die genannten Punkte wird eine Bandbreite erreicht, die es erlaubt, 25 mal pro Sekunde die Daten von vier hochauflösenden 3D-Sensoren (640 x 480 Datenpunkte) gleichzeitig in einen Octree (2cm Auflösung) einzutragen und diesen in weniger als zehn Mil-

lisekunden auf Kollision mit einer Robotertrajektorie zu prüfen. Gegenüber weit verbreiteten CPU-Implementierungen stellt dies eine 80-fache Beschleunigung dar.

Verglichen mit Voxelkarten ist es somit möglich, extrem große Volumen speichereffizient zu repräsentieren und in vielen Fällen sogar einen effizienteren Lesezugriff zu gewähren. Auch im Vergleich der Kollisionsprüfung von Voxelkarten konnte in realistischen Szenarien eine zehnfache Beschleunigung erzielt werden. Gleichzeitig fällt der Speicherverbrauch für die Modellierung typischer Innenraumszenen gegenüber Voxelkarten um ca. 70%, da in menschlichen Umgebungen meist nur ca. 30% des Raumvolumens Objekte enthalten.

8.4. Vergleich von Voxel- und Mesh-basierter Kollisionsdetektion

Um einen Eindruck der Möglichkeiten einer voxelbasierten Kollisionsdetektion zu erhalten, wurde diese in zwei aussagekräftige Beispielszenarien mit der weit verbreiteten Flexible Collision Library (FCL) Bibliothek [156] aus ROS vergleichen. Da es sich bei der FCL um ein BVH-Verfahren handelt, ist hier vor der eigentlichen Kollisionsprüfung ein Vorverarbeitungsschritt auszuführen, bei dem die Oberflächennetze aller rigiden Einzelmodelle einer Szene in Hüllvolumen aufgeteilt werden. Dieser Schritt ist bei jeder Änderung der Modelle erneut durchzuführen, bspw. wenn ein Umweltmodell aus Sensordaten tesseliert wird und sich Teile der Umwelt dynamisch verändern.

8.4.1. Voxel-Swept-Volumen

Zunächst sollen die Laufzeiten einer Swept-Volumen Kollisionsprüfung den akkumulierten Laufzeiten mehrerer Einzelprüfungen gegenübergestellt werden. In der verwendeten Testszene bewegt sich der mobile Roboter HoLLiE geradlinig vorwärts und durchdringt dabei das Modell eine Bar-Theke. Hierfür wurde die Fahrstrecke in 100 Zwischenpositionen unterteilt und diese zum einen einzeln mit der FLC evaluiert bzw. als Swept-Volumen gerendert und mittels GPU-Voxels ausgewertet. Durch die Verwendung von Bitvektor-Voxeln konnten dabei alle einzelnen Schritte getrennt bewertet werden.

In FCL bestehen die komplexen Dreiecksnetze des Roboters aus insgesamt $322\,254$ Dreiecken, während die Bar aus $18\,197$ Dreiecken zusammengesetzt ist. Das Voxelmodell hingegen ergibt sich aus $230\,751$ Punkten für den Roboter, der in einer Voxelkarte mit $1\,\mathrm{cm}$ Auflösung im Schnitt $10\,427\,712\,\mathrm{Voxel}$ belegt (bzw. $1\,600\,656\,\mathrm{Voxel}$ bei einer Auflösung von $2\,\mathrm{cm}$). Die Punktwolke der Bar enthält $13\,200\,504$ Punkte, die $830\,635\,\mathrm{Voxel}$ bzw. $109\,361\,\mathrm{Voxel}$ belegen.

Die Zeitmessungen aus zehn Kollisionsprüfungen sind in Tab. 8.5 gelistet. Als Initialisierung zählt bei FCL der Aufbau der BVHs, bei GPU-Voxels das Generieren des Swept-Volumens. Die Zahlen verdeutlichen, dass bei einem Voxelansatz die Initialisierung weit mehr Zeit kostet, als die eigentliche Prüfung. Daher ergibt sich für den Voxelansatz ein Laufzeitvorteil erst dann, wenn mehrere gleichartige Prüfungen auszuführen sind. Dies

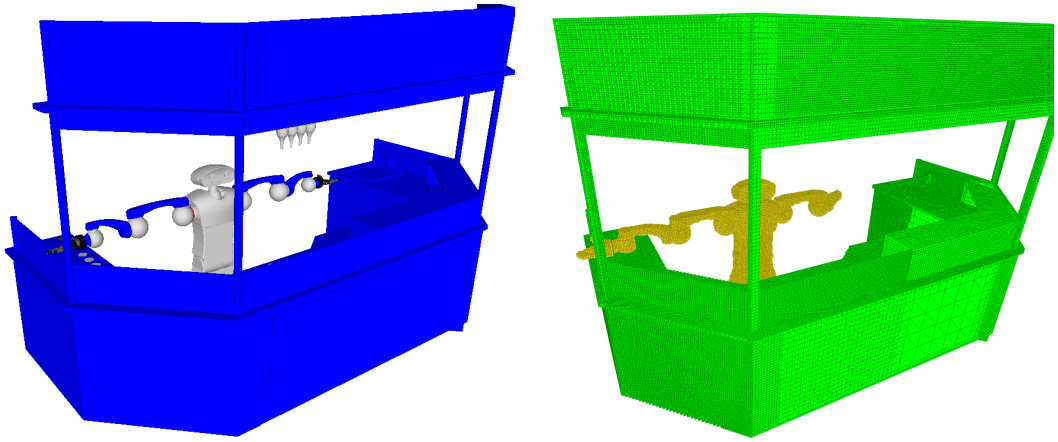
ist jedoch sehr häufig der Fall. Beispielsweise wenn bei Planungsaufgaben mit Bewegungsprimitiven gearbeitet wird (siehe Unterabschnitt 7.2.3), oder bei der Ausführungsüberwachung, wie sie weiter unten noch beschrieben wird. Je nach Auflösung ist dann bereits ab zwei bis drei Durchläufen der Voxelansatz effizienter, als die Prüfung einzelner Posen mittels Oberflächennetzen. Insbesondere kann nur durch die Verwendung von Swept-Volumen eine schritthaltende Kollisionsprüfung erreicht werden, die mit der Bildrate der Kinect-Kamera durchführbar ist. Im Gegensatz dazu ist es selbst unter Vernachlässigung des Tesselierungsaufwandes nicht möglich, mit Dreicksnetzen ein Bewegungsvolumen mit der nötigen Wiederholungsrate auf eindringende Hindernisse zu prüfen.

8.4.2. Prüfung einzelner Posen

Anwendungen, in denen keine vorberechnete Swept-Volumen genutzt werden können, sind samplingbasierte Planungsszenarien, die in Unterabschnitt 7.2.5 vorgestellt wurden. Hierbei ist die Verarbeitungszeit der Kollisionsprüfung einzelner, unabhängiger Roboterposen ausschlaggebend. Um diese realistisch bewerten zu können, wurde ein Szenario mit einem komplexen Umweltmodell und einem einfachen Robotermodell gewählt, in dem nur wenig Bewegungsfreiraum gegeben ist. In Abb. 8.10 sind Umwelt- und Robotermodell zu sehen, zwischen denen es bei der Planung häufig zu Kollisionen kommt.

Die Umwelt besteht in der Oberflächendarstellung aus $384\,624$ Dreiecken, der Roboter aus 5750 Dreiecken. In GPU-Voxels deckt das Volumen der gesamten Szene 8,2 Mio. Voxel bei einer Auflösung von $3\,\mathrm{mm}$ ab. Eingefügt werden Punktwolken der Umgebung mit $593\,784$ Punkten und der Roboter aus 1046 Punkten. Davon sind $591\,062$ Voxel durch das Umweltmodell und 995 Voxel durch den Roboter belegt. Die Laufzeiten in Tab. 8.6 wurden aus ca. 3000 Durchführungen der Kollisionsprüfungen ermittelt. Einige Konstellationen wiesen im Falle einer Kollision andere Laufzeiten auf, als im kollisionsfreien Fall. Diese sind separat gelistet.

Es zeigt sich, dass durch die Kombination einer probabilistischen Voxelkarte für die Umweltrepräsentation mit einer Liste aus Bitvektor-Voxeln konkurrenzfähige Laufzeiten für die eigentliche Kollisionsprüfung erreichbar sind. Allerdings dürfen diese nicht losgelöst betrachtet werden. In einer komplexen veränderlichen Umgebung müsste das Umweltmodell regelmäßig neu aufgebaut werden. Bei der Verwendung der FCL benötigt diese Generierung der BVH um drei Größenordnungen mehr Zeit, als die Kollisionsprüfung. Bei GPU-Voxels fällt der Unterschied um eine Größenordnung geringer aus. Bedingt durch den großen Komplexitätsunterschied zwischen Umwelt- und Robotermodell sind die Laufzeiten für die Vorverarbeitung des Roboters bei dieser Betrachtung vernachlässigbar. Somit bestehen die Vorteile des voxelbasierten Ansatzes wieder klar bei der Verarbeitung eines Umweltmodells, das aus Sensordaten generiert wird.

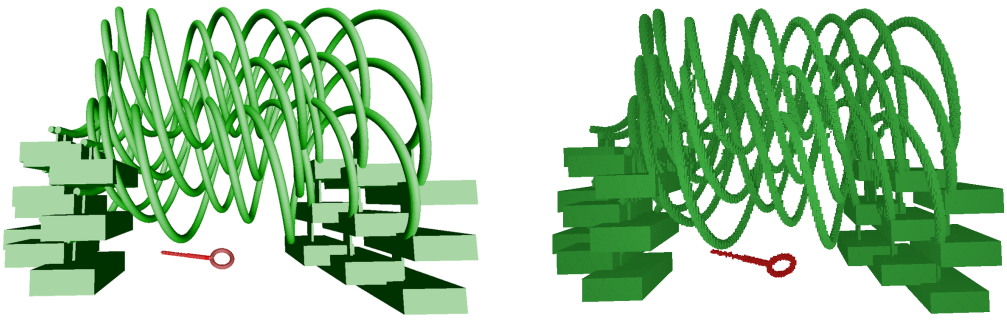


(a) Mesh-Darstellung (b) Voxel-Darstellung

Abb. 8.9.: Szenario zum Vergleich der Mesh-basierten Kollisionsdetektion mit GPU-Voxels. Der Roboter fährt vorwärts und durchdringt dabei den Tisch mit seinen Armen und dem Torso.

Szenario	Teilschritt	Dauer [ms]	σ [ms]
ROS FCL Mesh ∩ Mesh	Initialisierung	46,911	0,212
(100 Zwischenprüfungen)	Σ Kollisionschecks	428,980	37,385
Octree ∩ Voxelkarte (1 cm)	Initialisierung	1591,987	51,705
	Kollisionscheck	48,223	2,034
Octree ∩ Voxelliste (1 cm)	Initialisierung	1672,968	55,284
	Kollisionscheck	5,086	0,362
Octree ∩ Voxelkarte (2 cm)	Initialisierung	473,452	43,575
	Kollisionscheck	44,206	0,596
Octree ∩ Voxelliste (2 cm)	Initialisierung	537,206	47,095
	Kollisionscheck	2,437	0,197

Tab. 8.5.: Vergleich der Berechnungsdauer von Mesh-basierter Kollisionsdetektion mit GPU-Voxels (gemittelt über jeweils zehn Messungen, Standardabweichung σ).



(a) Mesh-Darstellung (b) Voxel-Darstellung

Abb. 8.10.: Szenario zum samplingbasierten Planen in Voxel- und Mesh-Darstellung. Das Robotermodell ist in rot dargestellt.

	Laufzeit [ms]			
	⌀	**Median**	**Min**	**Max**
ROS FCL Mesh ∩ Mesh				
Initialisierung BVHs	53,354	53,281	53,199	54,682
Kollisionschecks positiv	0,053	0,041	0,005	0,398
Kollisionschecks negativ	0,091	0,08	0,001	0,519
Prob. Voxelkarte ∩ Prob. Voxelkarte				
Modelle einfügen	0,292	0,283	0,266	4,008
Kollisionscheck	1,864	1,654	1,648	5,125
Prob. Voxelkarte ∩ Bitvektor Voxelliste				
Modelle einfügen	1,043	0,930	0,878	6,219
Kollisionscheck	0,098	0,064	0,061	1,964
Prob. Octree ∩ Bitvektor Voxelliste				
Initialisierung	16,172	15,988	13,742	25,587
Kollisionscheck	0,280	0,203	0,19	2,948
Prob. Octree ∩ Bitvektor Voxelkarte				
Initialisierung	16,205	15,926	13,088	31,904
Kollisionscheck	1,223	0,968	0,861	4,27
Prob. Octree ∩ Prob. Octree				
Initialisierung	24,484	23,834	17,912	35,664
Kollisionscheck positiv	6,159	6,073	5,257	9,493
Kollisionscheck negativ	5,782	5,613	5,04	8,894

Tab. 8.6.: Vergleich der Berechnungsdauer von Mesh-basierter Kollisionsdetektion mit GPU-Voxels. Daten aus ca. 3000 Durchläufen.

8.5. Visualisierung

Um einen Eindruck der Leistungsfähigkeit der implementierten Visualisierung zu gewinnen, wurde diese anhand mehrerer Beispielszenen evaluiert. Zunächst wird eine Innenraumszene in Form einer Voxelkarte aus ca. $62,5$ Mio. Voxeln betrachtet, in welcher knapp $60\,000$ Voxel belegt sind (siehe Abb. 5.20). Eine solche Szene kann in voller Auflösung mit über $40\,\mathrm{Hz}$ angezeigt werden, was mit einem Umweg über den Host nicht erreichbar wäre. Durch die Verwendung von Supervoxeln zur Auflösungsreduktion lässt sich die Bildrate auf über $100\,\mathrm{Hz}$ steigern. Das Diagramm aus Abb. 8.11 zeigt auch, unter welchen Randbedingungen die einzelnen Verfahren, die in Abschnitt 5.7 verglichen wurden, Vorteile haben. So eignet sich der Algorithmus mit Zwischenspeicher für umfangreiche Szenen, die nur mit geringer Auflösung zu zeichnen sind, da seine Laufzeit mit der Größe des Zwischenspeichers skaliert. Bei den Verfahren ohne Zwischenspeicher ist die Version ohne Vorberechnung erwartungsgemäß um bis zu Faktor zwei schneller, als die Version mit Vorberechnung, da die Datenstruktur nur einmal durchlaufen werden muss. Deutlich zu sehen sind auch die Effekte der sequenziellen Codeausführung innerhalb der Supervoxel-Kernel, welche beide Algorithmen bei größeren Supervoxelgrößen ausbremsen. Diese sind jedoch eher von theoretischem Belang, da Supervoxel mit Kantenlängen > 8 in der Praxis nicht benötigt werden. Tests mit synthetischen Szenen, die im Gegensatz zu realen Szenen eine Voxelkarte sehr dicht belegen, zeigen jedoch erwartungsgemäß höhere Bildraten bei großen Supervoxeln, da die Kernel stoppen, sobald ein erster belegter Voxel im Supervoxel gefunden wurde.

Supervoxel Größe	Anzahl Voxel
1	59808
2	19974
3	9821
4	5954
5	4074
6	2994
8	1801
10	1148
12	860
14	607
16	534
20	356
32	158

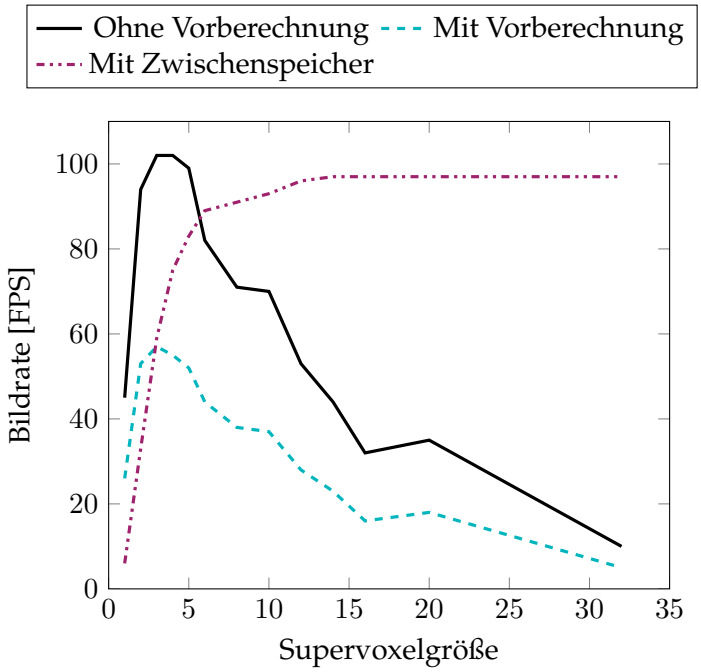


Abb. 8.11.: Vergleich der drei umgesetzten Verfahren: Gezeichnete Bilder pro Sekunde bei unterschiedlicher Supervoxelgröße. Daten aus [32].

Weiterhin wurde die Auswirkung einer Beschränkung des Sichtbereiches auf die Bildrate anhand einer Gebäudeszene evaluiert. Diese wurde zunächst komplett visualisiert, und

dann auf einen quaderförmigen Bereich vor der Kamera beschnitten. Da sich dabei die Anzahl der zu zeichnenden Voxel kaum verändert (siehe Diagramm aus Abb. 8.12), sind die entstehenden visuellen Unterschiede in Abb. 5.22 marginal. Dennoch zeigt sich je nach Supervoxelgröße eine um ein Vielfaches gesteigerte Bildrate im Diagramm. Grund hierfür ist das stark verkleinerte Iterationsintervall, das der Geometrie-Kernel auf der beschnittenen Karte abarbeiten muss.

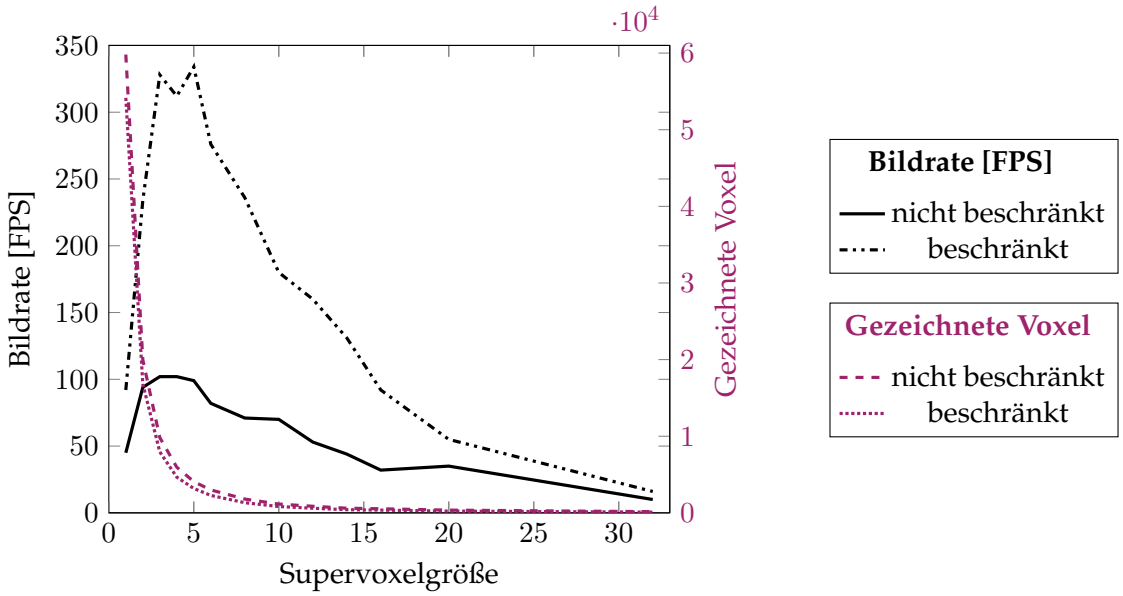


Abb. 8.12.: Anstieg der Bildrate bei Einschränkung des Sichtbereiches (Szene aus Abb. 5.22). Daten aus [32].

Ein Vergleich der unterschiedlichen Darstellungsvarianten zeigt die für OpenGL zu erwartenden Ergebnisse: Die Darstellung der Szene unter Nutzung der ambienten Beleuchtung ist erwartungsgemäß am schnellsten. Das zusätzliche Einzeichnen von Wireframes reduziert die Bildrate um etwa die Hälfte, da die Szene hierfür zweimal gerendert werden muss. Dagegen reduzieren die komplexeren Berechnungen der OpenGL-Shader bei der Verwendung einer zusätzlichen Punktlichtquelle die Bildrate nur um ca. 40%.

Wie bei allen CUDA Anwendungen muss auch bei den Geometrie-Kerneln ein möglichst optimales Verhältnis der Laufzeitparameter bestimmt werden (siehe Unterabschnitt 3.2.2). In umfangreichen empirischen Versuchen hat sich gezeigt, dass in durchschnittlichen Szenen CUDA Blöcke mit 4^3 oder 8^3 Threads die höchsten Bildraten erzielen, wobei pro Thread ca. 15 Supervoxel sequentiell abgearbeitet werden (bei einem Datenstrukturvolumen von 62 Mio. Voxeln). Unter Verwendung dieser Parameter können bei der umfangreichen Szene zur Octree Evaluierung aus Abb. 8.3 die Bildraten des Diagramms in Abb. 8.13 erreicht werden.

Ein Vergleich mit *rviz*, der auf OGRE basierenden Visualsierungslösung aus ROS, erbrachte bei einer ähnlichen Szene aus 3 Mio. gezeichneten Würfeln zwar noch annehmbare Bildraten zwischen 7 und 10 FPS. Jedoch verzögerte die Datenübermittlung zwischen Device und Host die Anzeige neuer Daten im Schnitt um 2 Sekunden. Dies macht die Auswertung von in Voxel umgewandelten Echtzeitdaten mit *rviz* unmöglich.

Seitenlänge [cm]	Dimension Voxelkarte	Anzahl Voxel
7,5	$1296 \times 1353 \times 330$	3 046 267
8,0	$1215 \times 1269 \times 309$	2 708 613
9,0	$1080 \times 1128 \times 275$	2 179 571
10,0	$972 \times 1015 \times 248$	1 789 652
12,5	$778 \times 812 \times 198$	1 167 927
15,0	$648 \times 677 \times 165$	814 039
20,0	$486 \times 508 \times 124$	458 672

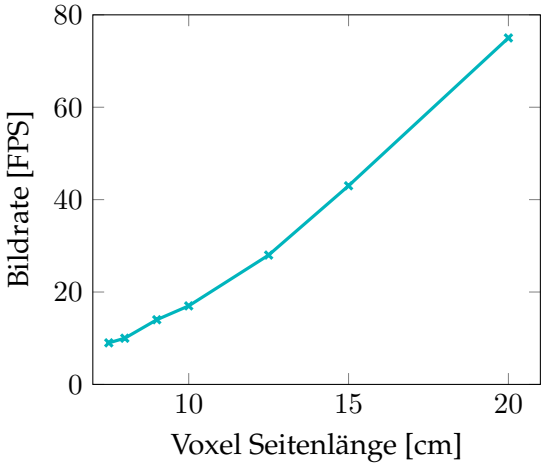


Abb. 8.13.: Erreichte Bildraten bei der Visualisierung umfangreicher Daten (Szene siehe Abb. 8.3) mit unterschiedlichen Voxelseitenlängen. Daten aus [32].

8.6. Experimente mit stationärem Roboter

Die folgenden Experimente beschäftigen sich mit der Planung und Überwachung von Bewegungen eines Roboterarmes mit serieller Kinematik. Der Planungsraum ist dabei der sechdimensionale Konfigurationsraum, während die Ausführung im dreidimensionalen Arbeitsraum $SE(3)$ stattfindet. Die gewählten Szenarien decken die beiden großen Problemklassen der samplingbasierten Planung sowie der feingranularen Arbeitsraumüberwachung ab und stehen somit für sehr allgemeine Fragestellungen.

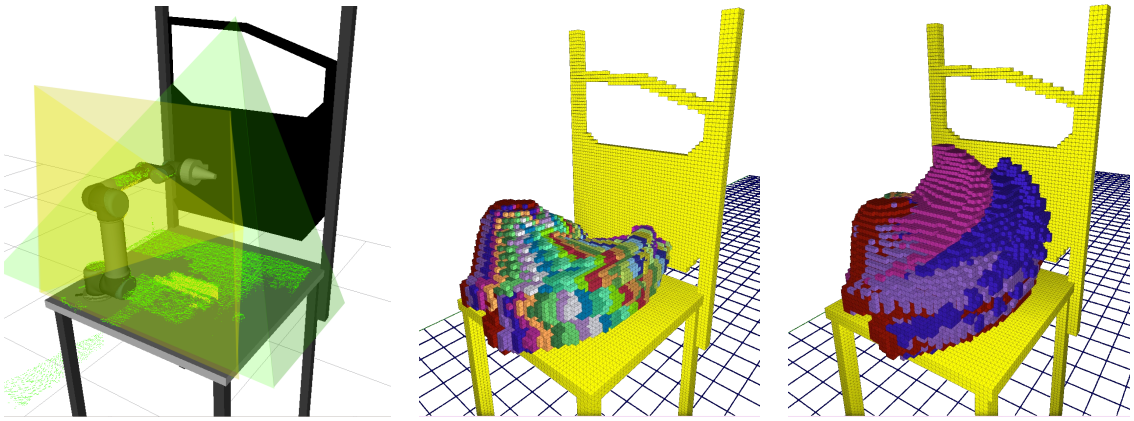
8.6.1. Geteilter Arbeitsraum

Ein in der Industrie immer relevanteres Szenario ist der geteilte Arbeitsraum, in dem Mensch und Roboter gemeinsam agieren und eine Vielzahl von unabhängigen Teilaufgaben abarbeiten. Dabei ist es unabdingbar, dass der Roboter seine Ausführungen überwacht und so plant, dass er einerseits die Sicherheit des Menschen wahrt, aber andererseits auch seine Aufgaben mit möglichst geringer Störung durch den Menschen durchführen kann. Aktuelle Lösungen teilen hierfür den Arbeitsraum in Bereiche ein, die exklusiv durch den Mensch oder durch den Roboter belegt sein dürfen. Die Sensorik, die die Anwesenheit des Menschen ermittelt, ist dabei der limitierende Faktor, der Form und Größe der Bereiche einschränkt. So können Laserscanner oder Lichtgitter pro Sensor lediglich eine zweidimensionale Ebene überwachen, wodurch nur eine sehr grobe Aufteilung des Raumes möglich ist. Betritt der Mensch einen Bereich, in dem der Roboter gerade aktiv ist, so muss dieser seine Ausführung unterbrechen und abwarten, bis der Mensch sich wieder entfernt hat. Diese binäre Entscheidung ist dabei unabhängig von den genauen Posen und Bewegungsrichtungen von Mensch und Roboter. Um dieses konservative und ineffiziente Verhalten mit potentiell unnötigen Stopps des Roboters zu vermeiden, muss eine feingranulare Arbeitsraumüberwachung ermöglicht werden, die im Folgenden beschrieben wird.

Als Beispielanwendung wurde ein Teilaspekt der industriellen Fertigung einer Autotür gewählt, bei der Mensch und Roboter gemeinsam Schrauben in der Türverkleidung setzen. Dabei ist es unerheblich, wer welche Schraubpositionen bearbeitet. Auf Grund von Gewährleistungsfragen wird in industriellen Anwendungen eine weitreichende Autonomie häufig eher kritisch betrachtet. Da es in einem solchen Szenario für den Roboter jedoch ausreicht, aus der Menge der Teilaufgaben diejenigen auszuwählen, die in Abhängigkeit vom Aufenthaltsort des Menschen gerade ausführbar sind, erübrigt sich eine dynamische Neuplanung der Bewegungen. So herrscht jederzeit eine deterministische Situation, da lediglich im voraus geplante und als sicher eingestufte Trajektorien ausgeführt werden. Dennoch kann GPU-Voxels auch hier einen wichtigen Beitrag leisten, um einen klassischen Roboter flexibler einzusetzen.

Dafür wird der Arbeitsraum von zwei Kinect-Kameras überwacht, deren Blickwinkel so gewählt wurden, dass sich Verschattungen minimieren. Wie in Abb. 8.14a zu sehen, waren die Sensoren oberhalb und unterhalb der Tür angebracht. Ihre Aufnahmen werden in eine probabilistische Voxelkarte M_{Env} mit 2 cm Voxelkantenlänge eingetragen: $\boxplus(M_{\text{Env}}, P_{\text{Kinect}})$. Anschließend muss das Volumen des Egomodells aus der Karte entfernt werden, um nicht fälschlicherweise zu Kollisionen zu führen. Hierfür wird das Robotermodell anhand der aktuellen Gelenkwinkelstellungen in eine weitere Voxelliste eingetragen und diese von der Umweltkarte subtrahiert: $M_{\text{Env}} = (M_{\text{Env}} - M_{\text{Rob}})$ mit $\boxplus(M_{\text{Rob}}, P_{\text{Rob}})$.

Für die Planung einer Aktion wurden im Vorfeld alle N möglichen Bewegungen des Roboters zu den verschiedenen Schraubpositionen vorberechnet und ihre Swept-Volumen in je eine Voxelliste $M_{\text{Ego},n}$ eingetragen. Wie in Abb. 8.14b zu sehen, ist dabei jede Bewegung in $K = 250$ äquidistante Schritte unterteilt, die einer individuellen SSV-ID s zugeordnet sind. Die Winkelinkremente innerhalb des Volumens berechnen sich aus $\vec{\varphi}_\Delta = (\vec{\varphi}_{\text{ziel}} - \vec{\varphi}_{\text{start}})/K$. Eine zusätzliche Liste $M_{\text{Ego},All}$ (siehe Abb. 8.14c) enthält zudem die Menge aller Trajektorien, wobei hier pro Bewegung nur eine SSV-ID genutzt wurde. Durch die Verwendung von Listen, die ausschließlich belegte Voxel speichern, liegt der kombinierte Speicherverbrauch unter 2 GB.



(a) Kalibrierung der zwei Kinect Sensoren in ROS

(b) Subvolumen einer einzelnen Trajektorie

(c) Swept-Volumen aller möglichen Trajektorien

Abb. 8.14.: Kalibrierung und Swept-Volumen der auszuführenden Trajektorien.

Vor Beginn einer Schraubaktion kann mittels einer einzigen Kollisionsprüfung $M_{\text{Env}} \cap M_{\text{Ego,All}}$ zwischen der Umweltkarte und der Liste mit allen Trajektorien bestimmt werden, welche SSV-IDs kollisionsfrei sind. Somit ist klar, welche Bewegungen aktuell ausführbar sind. Aus dieser Liste wird dann eine Bewegung n gestartet.

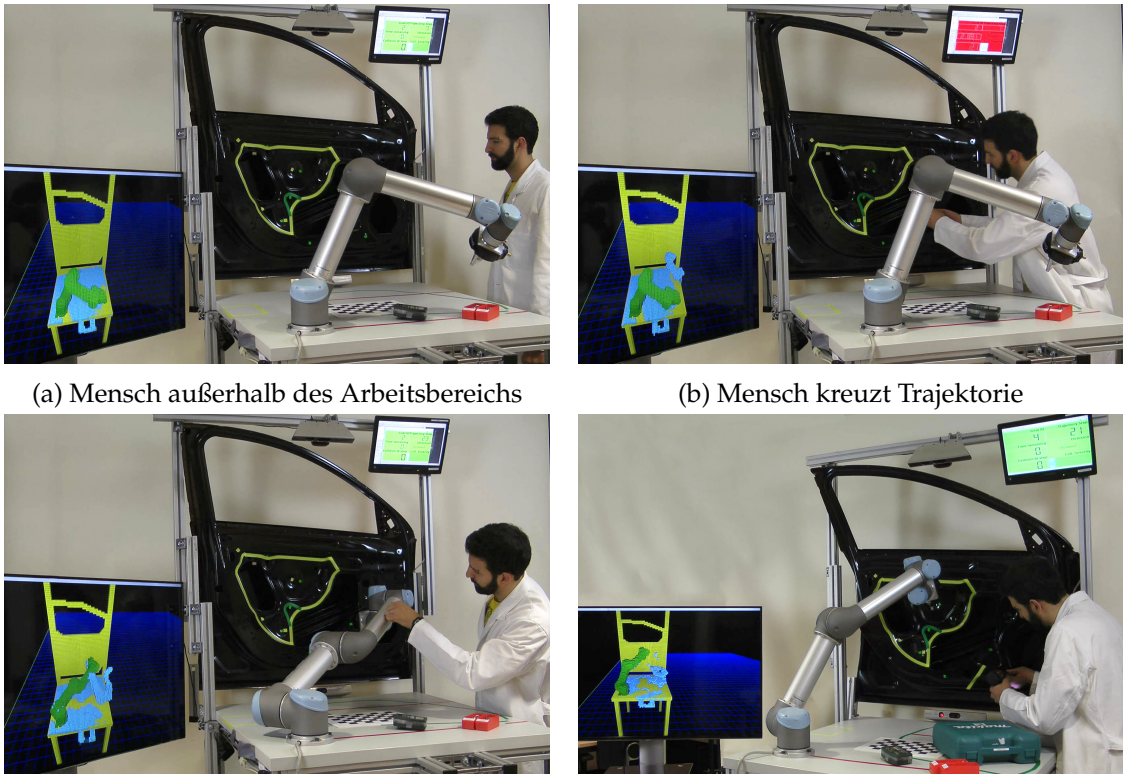
Ab diesem Moment läuft mit jeder Aktualisierung der Umweltkarte durch eine neue Punktwolke eine Kollisionsprüfung mit der n-ten Voxelliste $M_{\text{Ego},n}$ ab, die die zeitlich abgetastete Einzelbewegung enthält: $M_{\text{Env}} \cap M_{\text{Ego},n}$. Gleichzeitig kommuniziert die Bewegungsausführung den aktuellen Gelenkwinkelzustand $\vec{\varphi}_t$ des Roboters an die Überwachung. Aus dieser kann mit der step Funktion der Abschnitt s im Swept-Volumen berechnet werden, dem die Roboterpose $\vec{\varphi}_t$ am ähnlichsten ist. Wird nun eine Kollision erkannt, lässt sich über die kleinste, in Kollision liegende SSV-ID ermitteln, ob die Kollision in einem bereits abgefahrenen Teil der Trajektorie liegt, oder in einem noch zu durchquerenden Volumen. Somit entscheidet die Funktion stop aus Gleichung 8.1, ob der Roboter angehalten werden muss, oder ob die Kollisionen ignoriert werden, da sie zeitlich gesehen bereits hinter der aktuellen Roboterstellung liegen. In den Formeln ist nicht dargestellt, dass Kollisionen nur berücksichtigt wurden, wenn die Menge an kollidierenden Voxeln über einem festen Grenzwert lag. Über diesen lassen sich minimale Eigenkollisionen ignorieren, die von der detektierten Verkabelung am Roboter ausgelöst werden.

$$\text{stop}(P_{\text{Kinect}}, \vec{\varphi}_t) := \begin{cases} 1 & : \min\limits_{\text{SSV-ID}} (M_{\text{Env}} \cap M_{\text{Ego},n}) \leq \text{step}(M_{\text{Ego},n}\ ,\ \vec{\varphi}_t) \\ 0 & : \text{sonst} \end{cases} \tag{8.1}$$

$$\text{step}(M_{\text{Ego},n}\ ,\ \vec{\varphi}_t) := s \mid \min(\|\vec{\varphi}_t - (s \cdot \vec{\delta_\varphi})\|) \tag{8.2}$$

Muss der Roboter anhalten, wartet er eine definierte Zeit ab, ob sich das Hindernis wieder entfernt und die Bewegung fortgesetzt werden kann. Im anderen Fall wird die Bewegungsrichtung invertiert und der Roboter fährt zu seiner Ausgangspose zurück. Auch diese Bewegung wird überwacht und bei Bedarf auf unbegrenzte Zeit angehalten. Ist die Startpose erreicht, kann erneut mit der Auswahl einer alternativen Trajektorie begonnen werden. Dabei wird die Wahl so getroffen, dass die neue Bewegung möglichst viel Abstand zur zuletzt blockierten Ausführung liegt, um die Wahrscheinlichkeit einer wiederholten Kollision zu minimieren. Dies kann aus der bekannten örtlichen Anordnung der Zielpunkte abgeleitet werden.

Die Beispielanwendung wurde im Rahmen von Präsentationen mehrfach öffentlich gezeigt. Momentaufnahmen verschiedener Situationen sind in Abb. 8.15 abgebildet. Der Statusmonitor oberhalb der Tür visualisiert in rot und grün das vorliegen einer relevanten Kollision. Durch die erfolgreichen Experimente konnte gezeigt werden, dass eine feingranulare, schritthaltende Überwachung des Arbeitraumes mittels GPU-Voxels möglich ist, und dieser somit sehr effizient gemeinsam genutzt werden kann. Die Berechnungszeiten aller nötigen Schritte lagen durchgehend unter den verfügbaren 33 ms, die zwischen zwei Sensoraufnahmen vergehen. Eine große Herausforderung bei der praktischen Umsetzung bestand in der extrinsischen Kalibrierung der Kameras gegenüber dem Roboter. Liegt hier ein Fehler vor, schlägt die Subtraktion des Egomodells aus den Kameradaten fehl, und der Roboter detektiert sich selbst als Hindernis.



(a) Mensch außerhalb des Arbeitsbereichs

(b) Mensch kreuzt Trajektorie

(c) Mensch hinter der Ausführung

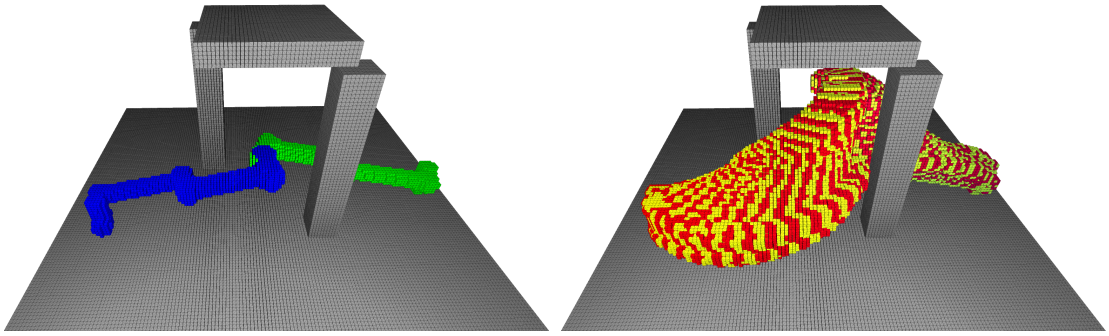
(d) Getrennte Arbeitsvolumen

Abb. 8.15.: Geteilter Arbeitsraum zur Autotürenmontage. In allen vier Szenen bewegt sich der Roboter auf die Tür zu und bremst, falls der Mensch seine auszuführende Trajektorie kreuzt. Bewegt sich der Mensch zeitlich gesehen hinter dem Roboter, wird hingegen nicht angehalten.

8.6.2. Samplingbasiertes Planen

Auch wenn die Kollisionsprüfung einzelner unabhängiger Posen theoretisch nicht das optimale Anwendungsszenario für GPU-Voxels darstellt, sollte dennoch geprüft werden, wie gut sich ein samplingbasiertes Planungsverfahren damit umsetzen lässt. Als Basis wurde hierfür die OMPL herangezogen, die eine Vielzahl an unterschiedlichen Planungsverfahren unter einer einheitlichen API zusammenfasst. Geplant werden die sechs Freiheitsgrade eines Universal Robot 10 Armes mittels LBKPIECE1 [194] (siehe auch Abschnitt 7.1.3). Die Evaluation erfolgte in zwei Szenarien: Im ersten Versuch (siehe Abb. 8.16) musste sich der Roboter durch eine statische Engstelle bewegen, im zweiten Versuch (siehe Abb. 8.17) in einer offenen Umgebung um ein dynamisches Hindernis herum. In beiden Fällen war auch der Boden der Szene als Hindernis modelliert. Der verwendete Arbeitsraum hatte eine Größe von $3\,m \times 3\,m \times 2\,m$ die durch $2\,250\,000$ Voxel mit 2 cm Kantenlänge dargestellt wurden.

Neben der generellen Zeitmessung sollte verglichen werden, ob eine Voxelkarte oder eine Voxelliste als Repräsentation für das Robotermodell effizienter ist. Dieses bestand aus zehn Segmenten, die insgesamt $36\,504$ Punkte enthielten, welche durchschnittlich knapp 4000 Voxel belegten.



(a) Szenario mit Start und Zielkonfiguration (b) Swept-Volumen der geplanten Trajektorie

Abb. 8.16.: Beispiel einer Trajektorienplanung durch einen engen Korridor. Die durchschnittliche Planungszeit liegt bei 2 Sekunden.

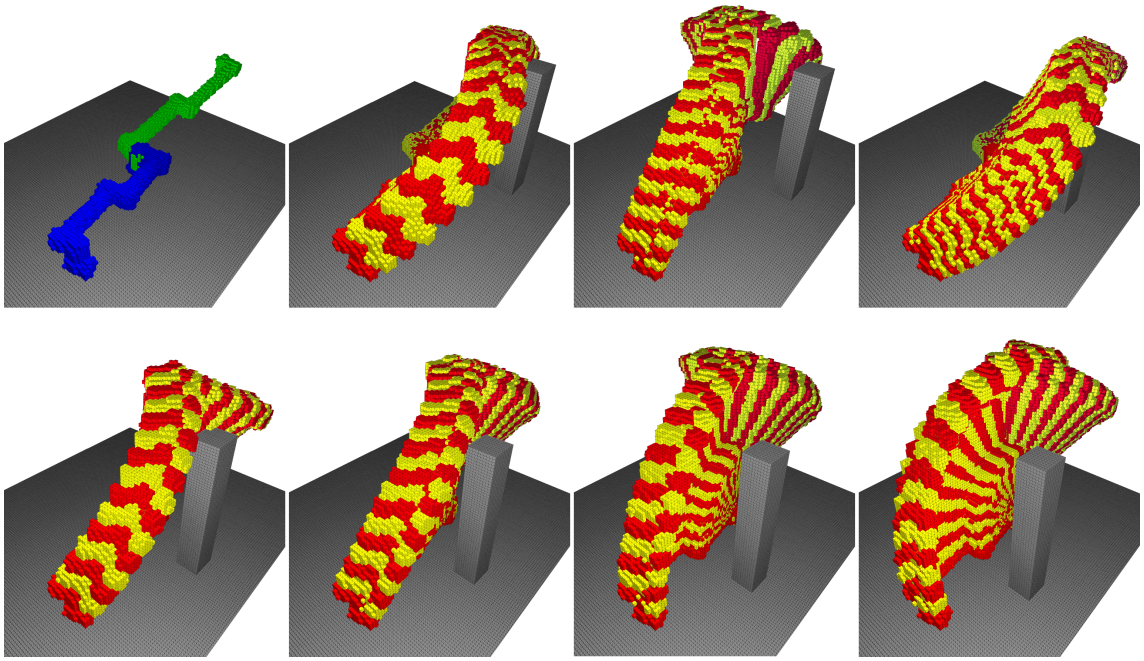


Abb. 8.17.: Planungsergebnisse in dynamischer Szene. Die Säule bewegt sich von hinten nach vorne am Roboter vorbei. Die durchschnittliche Planungszeit liegt bei 0,77 Sekunden.

Beide Szenarien wurden pro Repräsentationstyp 100 mal geplant. Die resultierenden Trajektorien enthielten zwischen 5 und 137 Zwischenzustände, während die Glättung diese auf eine Länge von 5 bis 45 reduzierte. Die Zeitmessungen der vier Tests, die sich in Tab. 8.7 finden, zeigen, dass trotz der geringen Voxelzahl des Robotermodells eine Voxelkarte in diesem Szenario deutlich performanter als eine Voxelliste ist. Zum genaueren Verständnis schlüsselt Tab. 8.8 die Zeiten der verwendeten Kollisionstests detaillierter auf. Hier ist ersichtlich, dass der Aufbau der Listen und nicht die eigentliche Kollisionsprüfung für den hohen Zeitaufwand verantwortlich ist. Bei den Voxelkarten hingegen ist Rechenzeit dagegen nahezu gleichmäßig aufgeteilt.

Der verwendete LBKPIECE1 Planer nutzt drei unterschiedliche Kollisionsprüfungen, die mittels GPU-Voxels implementiert wurden: Zunächst sind im *Lazy*-Teil diskrete Posen für den Aufbau des Planungsgraphen zu prüfen (erste Spalte). Ist eine Lösung gefunden, wird diese abschnittsweise durch Einzelposen abgetastet, die wiederum auf Kollisionsfreiheit geprüft werden (dritte Spalte). Bei der finalen Glättung kommt dann ein Swept-Volumen zum Einsatz (zweite Spalte), das optimierte Abschnitte repräsentiert. Hier bescheinigt die Zeitmessung die Vorteile des Swept-Volumens, da damit im Schnitt auf 10 Posen im Pfad nur eine Kollisionsprüfung kommt.

Die Ergebnisse zeigen, dass ein samplingbasierter Planungsansatz auf dynamischen Punktwolkendaten mit der entwickelten Voxel-Kollisionsprüfung in vertretbaren Berechnungszeiten umsetzbar ist. Kann jedoch eine statische Umgebung vorausgesetzt werden, lassen sich vergleichbare Planungsszenarien mit demselben Planer und einer BVH-Kollisionsprüfung bis zu 80 mal schneller lösen (evaluiert mittels ROS MoveIt[3]).

Szenario	**Laufzeit [ms]**					**Evaluierte**
	⌀	**Median**	**σ**	**Min**	**Max**	**Posen ⌀**
Mobiles Hindernis			**Voxelkarte ∩ Voxelkarte**			
Planung	625,6	581,1	287,4	119,4	1390,9	711
Glättung	214,9	197,7	85,1	73,3	602,9	496
Σ	840,5	778,8	372,5	192,7	1993,8	1207
Mobiles Hindernis			**Voxelliste ∩ Voxelkarte**			
Planung	2621,7	2806,8	919,8	1292,0	4444,1	564
Glättung	3487,7	3323,2	1031,4	2387,6	5841,4	675
Σ	6109,4	6130,0	1951,2	3679,6	10 285,5	1239
Enge Passage			**Voxelkarte ∩ Voxelkarte**			
Planung	2103,3	1761,5	1475,3	501,5	7843,2	2026
Glättung	293,1	289,0	133,3	138,8	837,8	772
Σ	2396,4	2050,5	1608,6	640,3	8681,0	2798
Enge Passage			**Voxelliste ∩ Voxelkarte**			
Planung	9851,6	9285,6	4956,6	1620,9	20 026,1	1854
Glättung	6405,1	5961,0	2743,8	3011,1	14 581,1	1073
Σ	16 256,7	15 246,6	7700,4	4632,0	34 607,2	2927

Tab. 8.7.: Berechnungsdauer der samplingbasierten Planung eines Roboterarmes (gemittelt über jeweils 100 Durchläufe, Standardabweichung σ).

8.6.3. Ablaufplanung von mehreren Robotern

In industriellen Roboterzellen teilen sich oft mehrere Roboter einen Arbeitsraum, in dem sie gemeinsam ein Bauteil bearbeiten. Jeder Roboter führt dabei mehrere, meist unabhängige Aktionen aus. Ab einer gewissen Anzahl von Bewegungen wird es für den Roboter-

[3] ROS MoveIt Planning Framework `http://moveit.ros.org/`

Datenstruktur des Roboters	Teil-Operation	∅ Laufzeit: Kollisionscheck / Pose [ms] Diskrete Posen	Swept-Volumen	Swept-Volumen aus Einzelposen
Voxelkarte	Einfügen	0,415	0,180	0,457
	Koll. Prüfung	0,489	0,123	0,511
	Σ	0,904	0,303	0,968
Voxelliste	Einfügen	5,016	3,626	6,372
	Koll. Prüfung	0,081	0,027	0,220
	Σ	5,097	3,653	6,592

Tab. 8.8.: Durchschnittliche Berechnungsdauer der Kollisionstests pro Pose bei der samplingbasierten Planung (Zeiten incl. Voxelumwandlung).

programmierer jedoch schwierig, eine kollisionsfreie Abfolge aller Aktionen zu bestimmen. Pausen innerhalb des Ablaufs verringern die Taktzeit der Produktion und sollten vermieden werden. Dieses Problem der Ablaufplanung wird aktuell in CAD Programmen durch die Trial-and-Error-Methode gelöst, bei denen die Roboterbewegungen mit klassischen Kollisionsprüfungsverfahren langwierig evaluiert werden. Treten Kollisionen auf, wird die Reihenfolge der Bewegungen durch menschliches Expertenwissen so lange verändert, bis die Ausführung sicher ist.

Um das Problem automatisch zu optimieren, muss das Kreuzprodukt aller Bewegungen auf Kollisionen überprüft werden. Dies kann durch den Einsatz von Swept-Volumen aus Bitvektor-Voxeln sehr effizient per GPU-Kollisionsprüfung erreicht werden. Ist in den IDs der Sweeps die Zeit codiert, lässt sich mit dem Bitvektor-Voxel Schnitt aus Abschnitt 6.2.1 feststellen, welche Aktionen in Kollision liegen. Voraussetzung ist, dass die Aufgaben unabhängig voneinander sind und somit in einer beliebigen Reihenfolge ausführbar sind. Weiterhin müssen alle Einzelbewegungen zu Beginn diskreter, gleich langer Zeitfenster starten, deren Dauer sich nach der längsten Bewegung richtet. Um die Rechenzeit zu verkürzen und die Auflösung zu maximieren, sollten alle n_r beteiligten Roboter etwa gleich viele Einzelbewegungen n_m aufweisen, die ähnlich lange dauern. Dann werden insgesamt $n_r \cdot n_m$ Voxellisten erstellt, die in einem rekursiven Backtracking-Verfahren (siehe Algorithmus 9 in Appendix Anhang A) gegeneinander auf Kollisionen geprüft werden. Da dabei häufig dieselben Bewegungen miteinander geschnitten werden, wurde ein Zwischenspeicher implementiert, der zu jedem Bewegungspaar das Ergebnis ihrer Kollisionsprüfung vorhält. So ist in vielen Fällen nur ein Nachschlagen notwendig.

Das Ergebnis sind n_r Listen mit je n_m Einträgen, die pro Roboter die validen Bewegungs-IDs in ihrer auszuführenden Reihenfolge enthalten. Ist keine kollisionsfreie Abfolge möglich, so schlägt das Backtracking fehl. In diesem Fall kann der Nutzer jedem Roboter eine Pausenbewegung hinzufügen und den Algorithmus erneut starten.

Für eine Laufzeitmessung wurde ein Beispiel mit unterschiedlichen Robotern herangezogen, deren Trajektorien sich regelmäßig kreuzen (siehe Abb. 8.18). Der Speicherverbrauch des Ablaufplaners steigt linear mit der Anzahl der Roboter und deren Anzahl an

Bewegungen ($\mathcal{O}(n_r \cdot n_m)$). Bei der Berechnungsdauer muss zwischen zwei Ergebnissen unterschieden werden: Die Dauer zur Ermittlung aller möglichen Lösungen wächst exponentiell mit beiden Parametern ($\mathcal{O}((n_m!)^{n_r})$). Beachtenswert ist hingegen, dass die Zeit für die Berechnung einer ersten validen Lösung im Schnitt lediglich linear mit der Anzahl der Roboter und deren Bewegungen steigt. Da im Allgemeinen eine Lösung ausreichend ist, ist dies die relevantere Aussage. Prinzipbedingt schwankt die genaue Laufzeit mit der Sortierung der Eingabedaten, da sie die Rechenzeit des Backtrackings beeinflusst. Dies spiegelt sich auch in den Ergebnissen aus Tab. 8.9 wieder, die einen Ausschnitt aus den ermittelten Messungen auflistet. Im realistischen Szenario mit vier Robotern und je vier Bewegungen konnte aus 331 776 Kombinationen eine kollisionsfreie Reihenfolge aller 16 Aktionen im Schnitt in nur 1,52 s gefunden werden.

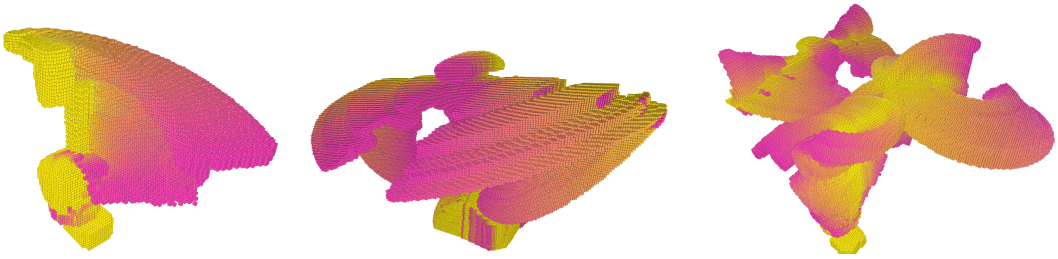


Abb. 8.18.: Beispieltrajektorien für das Scheduling von Roboterbewegungen (Links und Mitte). Kombination im geteilten Arbeitsraum (Rechts).

Konfiguration		⌀ Laufzeit [s]		
# Roboter	**# Bewegungen**	**Initialisierung**	**Erstes Ergebnis**	**Alle Ergebnisse**
6	4	2,170	2,481	80,661
5	4	1,893	2,250	10,524
4	4	1,672	1,520	5,211
3	4	1,458	0,763	2,880
2	4	1,310	0,324	1,243
1	4	1,186	0,001	0,001
3	2	0,877	0,398	0,715
3	3	1,202	0,572	1,547
3	4	1,450	0,765	2,864
3	5	1,779	0,974	6,058
3	6	2,192	1,156	180,629

Tab. 8.9.: Unterschiedliche Kombinationen aus Roboteranzahl bzw. Bewegungsanzahl und ihr Einfluss auf die Berechnungsdauer des Backtracking-Verfahrens. Ergebnisse gemittelt über 10 Durchläufe.

8.7. Experimente mit mobilen Robotern

Im Folgenden sollen Planungsaufgaben im $SE(2)$-Raum betrachtet werden, die kollisionsfreie Bewegungspfade für mobile Plattformen erzeugen. Die komplexe Geometrie,

der in den Beispielen verwendeten Roboter HoLLiE und IMMP, erfordert dabei eine dreidimensionale Kollisionserkennung.

8.7.1. Demonstrationssysteme

Da besonders in der Robotik die theoretische Evaluation von Algorithmen mittels Simulationen erfahrungsgemäß weit von den Ergebnissen mit physischen Systemen abweicht, wurden im Rahmen dieser Arbeit mehrere mobile Robotersysteme entwickelt. Sie dienen als Test- und Demonstrationsplattformen für viele praxisnahe Versuche, unter anderem mit GPU-Voxels. Ihre wichtigsten Eigenschaften sollen daher hier in aller Kürze beschrieben werden.

Die Gemeinsamkeit der konstruierten Systeme liegt in ihrem kinematischen Aufbau, der eine hohe Anzahl an beweglichen Freiheitsgraden aufweist und der den Robotern eine sehr wandelbare Geometrie verleiht. Aus diesem Grund lassen sich bei einer Bewegungsplanung keine validen Approximationen der Geometrien durch 2D Projektionen umsetzen, was einen der praktischen Beweggründe für diese Arbeit darstellt.

(a) Adero (b) HoLLiE (c) IMMP

Abb. 8.19.: Demonstratorsysteme, deren Entwicklung der Autor im Laufe der Dissertation leitete: Anthropomorphe mobile Manipulationsplattform Adero, House of Living Labs intelligent Escort (HoLLiE), Industrielle Mobile Manipulations Plattform (IMMP)

Adero - Advanced Dexterous Robot

Zur Untersuchung zweihändiger Manipulationsaufgaben im Rahmen der Projekte DESIRE [176] und Dexmart [15] standen am Forschungszentrum Informatik (FZI) zwei KUKA Leichtbauroboterarme zur Verfügung. Um diese nicht nur stationär, sondern auch für mobile Manipulationsaufgaben nutzen zu können, wurde eine erste mobile Plattform entwickelt, die diese Arme tragen konnte. Grundlage bildete die Antriebsmechanik eines Roboters des Instituts für Anthropomatik (IFA) von Professor Dillmann, die mit modernen Motorreglern, zwei Computern und vielfältiger Sensorik ausgestattet wurde. Der

Aufbau des Systems fand ab 2010 statt, genutzt wurde der Roboter bis 2012. Herausragendes Merkmal von Adero war eine passive Achse im Oberkörper, mit welcher der Arbeitsraum der Arme von Tischhöhe auf die Bodenebene gebracht werden konnte. Die Konstruktion des Roboters war so ausgelegt, dass der Schwerpunkt der Plattform eine statische Stabilität auch bei ausgestreckten Armen garantierte [10]. Trotz ausreichend dimensionierter Akkumulatoren zur Stromversorgung war jedoch kein kabelloser Betrieb des Roboters möglich, da die industriell ausgelegten Steuerungseinheiten der Leichtbauarme zu groß waren, um auf einem mobilen Roboter untergebracht werden zu können.

Softwareseitig konnte mit Adero das vorhandene Greifplanungssystem von Xue [204] mit einer mobilen Komponente erweitert werden, um unterschiedliche Umweltmodellierungen (teilweise basierend auf Graphendatenbanken) und Objekterkennungsverfahren zu evaluieren.

HoLLiE - House of Living Labs intelligent Escort

Um über ein wirklich kabelloses und somit mobiles System zu verfügen, wurde 2011 der Roboter HoLLiE ins Leben gerufen [9]. Seine Entwicklung verfolgte zwei maßgebliche Strategien: Da eine höhere Verlässlichkeit und eine beschleunigte Entwicklung hohe Priorität hatten, sollten beim Aufbau zum größtmöglichen Anteil industrielle Komponenten Verwendung finden. Des weiteren sollte HoLLiE in der Lage sein, bestmöglich in menschlichen Umgebungen arbeiten und für Menschen gemachte Gegenstände handhaben zu können. Entsprechend wurde die Größe des Roboters auf rollstuhlgerechte Umgebungen abgestimmt und seine Kinematik so gestaltet, dass er Gegenstände vom Boden aufnehmen kann. Das Erscheinungsbild des Roboters wurde in Zusammenarbeit mit einem Industriedesigner entwickelt und in großen Teilen im SLS 3D-Druck gefertigt. Es wirkt durch seine leicht humanoiden Züge zwar freundlich, bleibt dabei jedoch sehr weit von einem Uncanny Valley Effekt [147] entfernt. Eine flexible Stofffront erlaubt die Bewegung des Oberkörpers ohne sichtbare Scharniere.

Geplant und umgesetzt wurde eine innovative Parallelogrammechanik zum Last- / Momentenausgleich im Körper des Roboters [31]. Diese nimmt die Torsionsmomente, die im Körper durch ein Ausstrecken der Arme entstehen können, auf, so dass sie nicht auf den Antriebseinheiten lasten, sondern in die Basis des Roboters abgeleitet werden. Außerdem hält sie die Schulter- und Hals-Aktuatoren stets waagerecht. Eine verwindungssteife Verbindung zwischen Kopf und Armen stellt eine konstante Hand-Augen-Kalibrierung sicher. Die Basiskomponenten bilden eine omnidirektional verfahrbare Segway Plattform, zwei SCHUNK LWR 6-Achs Roboterarme, zwei SCHUNK Antriebsmodule zur Körperbewegung sowie ein SCHUNK Zwei-Achs-Antriebsmodul im Hals. Als Hände können sowohl DLR-HIT 4-Finger-Hände als auch SCHUNK SVH 5-Finger-Hände angebracht werden. Neben der selbst entwickelten Sicherheits- und Energiemanagementelektronik verfügt HoLLiE über ein RGB-LED-Band, das die mobile Basis umgibt und für die intuitive Visualisierung unterschiedlicher Informationen (bspw. beabsichtigter Bewegungsrichtung oder detektierte Hindernisse) genutzt wird. Die Basis enthält außerdem zwei Computer, Laserscanner und Akkus, während im Körper Netzwerktechnik, Lautsprecher und Verstärker untergebracht sind.

Zur zielgerichteten Bewegung des Roboters wurden in der Masterarbeit von Ruscheweyh [30] unterschiedliche Ansätze zur Berechnung einer inversen Ganzkörperkinematik unter Nutzung von Nullräumen und unterschiedlichen Gewichtungsfaktoren der kinematischen Teilketten implementiert.

HoLLiE absolvierte über die Jahre viele medienwirksame Auftritte, teilweise auch vor großem Messepublikum. Im Rahmen der GPU-Voxels Entwicklung diente der Roboter mehrfach als Testplattform. In diesen Fällen wurde HoLLiE jedoch kabelgebunden verwendet, da die Bordcomputer nicht über eine GPU verfügen, und alle Daten daher auf einem externen PC verarbeitet werden mussten.

IMMP - Industrial Mobile Manipulation Plattform

Zum Auftakt des Projektes ISABEL konnte eine weitere mobile Plattform angeschafft werden, um die Arme von Adero aufzunehmen: Ein holonomer KUKA omniRob. Diese Plattform verfügt über eine außerordentlich solide, mechatronische Basis und sollte daher für industrienahe Aufgaben eingesetzt werden. Zusätzliche Umbauten ermöglichten den mobilen Betrieb der leistungsstarken Nvidia Titan GPU an Bord des Roboters. Die GPU dient der Sensordatenverarbeitung von insgesamt acht Kinect-Kameras mittels GPU-Voxels, die dem Roboter eine 3D Rundumsicht ermöglichen.

Trotz mehrjähriger Anstrengungen und einer intensiven Kooperation mit dem Hersteller wurde leider keine passende Möglichkeit gefunden, die Plattform über eine externe, hochfrequente Regelung anzusteuern. Somit konnte der Roboter während der Dissertation nur zur Datenaufnahme genutzt werden (siehe Abb. 8.20), nicht aber für eine reaktive Bewegungsplanung, wie sie in Unterabschnitt 7.2.3 entwickelt wurde.

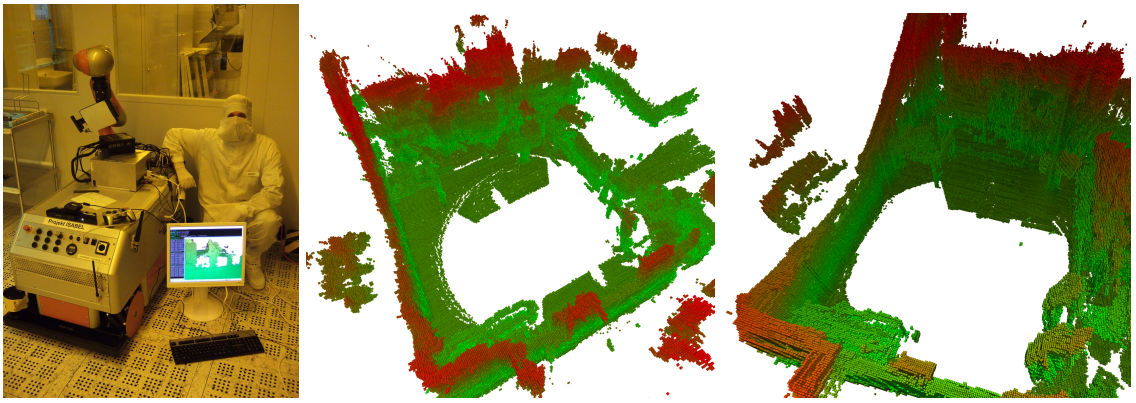
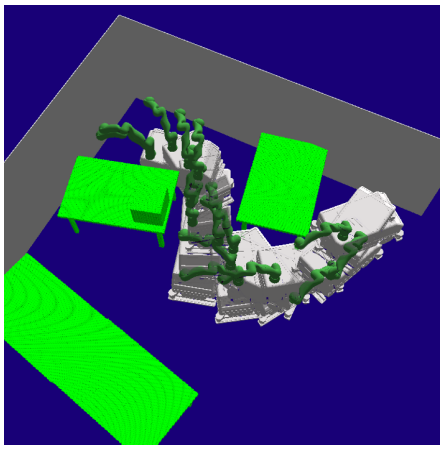


Abb. 8.20.: Tests im Rahmens des ISABEL-Projektes im Reinraum bei Infineon Regensburg zur Demonstration eines geteilten Arbeitsraumes mit einem mobilen Roboter. Rechts: Erstellte Voxelkarten zur Kollisionsvermeidung.

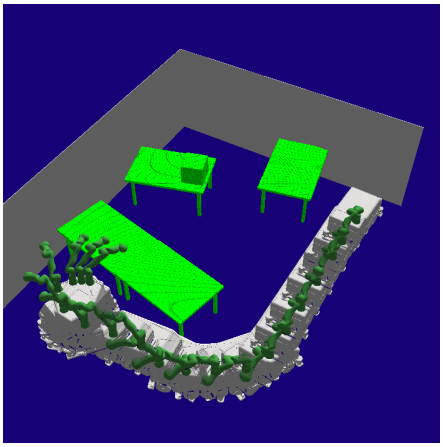
8.7.2. Planung mit Rotations-Swept-Volumen

Die Verfahren aus Unterabschnitt 7.2.2 zeigen, wie mit Rotations-Swept-Volu-men effizient kollisionsfreie Bewegungen einer mobilen Plattform auf einem Planungsgitter ge-

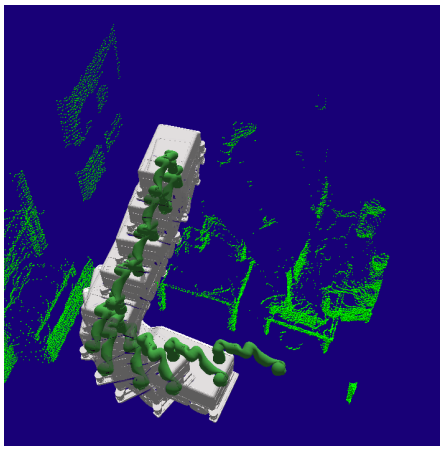
neriert werden können. Um die Praxistauglichkeit des Verfahrens bewerten zu können, wurden Versuche mit unterschiedlichen Robotern durchgeführt.



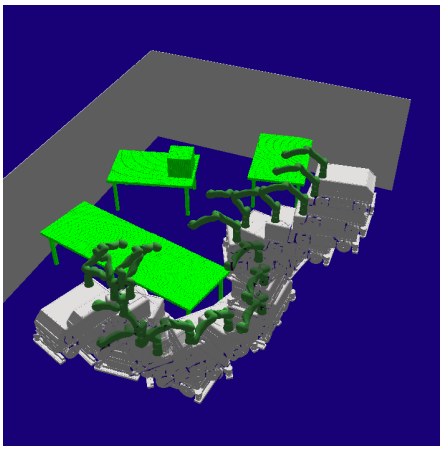
(a) Szanerio 1: Roboter muss durch enge Passage hinter die Tische. Rotationen sind durch Box auf Tisch eingeschränkt, weshalb eine Drehung vor der Passage eingeplant werden muss.



(b) Szenario 2: Planung einer längeren Strecke im freien Raum. Der Planer generiert eine sehr glatte Trajektorie.



(c) Szanerio 3: Planung auf Sensordaten einer engen Passage, die die Rotationsmöglichkeiten stark einschränkt.



(d) Ergebnis der RRT-Connect Planung auf Szenario 2 (oben) als Vergleich.

Abb. 8.21.: Ergebnisse des implementierten Plattformplaners auf drei Testszenarien und das Ergebnis einer RRT-Connect Planung zum Vergleich. Veröffentlicht in [3].

Zunächst soll der Einfluss der Voxel, bzw. Zellengröße auf die Planungsgeschwindigkeit ermittelt werden, wofür Voxel mit Kantenlängen von $1\,\mathrm{cm}$, $2\,\mathrm{cm}$, $4\,\mathrm{cm}$ und $8\,\mathrm{cm}$ und Zellen mit $4\,\mathrm{cm}$ und $8\,\mathrm{cm}$ betrachtet werden. Da bei der Planung eine hierarchische Kollisionsprüfung eingesetzt wird (siehe Unterabschnitt 5.3.2), ermöglichen diese Voxelgrößen das einfache Umrechnen zwischen den unterschiedlichen Auflösungen. Weiterhin kommt die Translation mittels Basisversatz aus Unterabschnitt 5.3.1 zum Einsatz, wobei die Voxelliste des Egomodells gegenüber der Umweltkarte verschoben wird. Die ersten Tests verdeutlichen den Zusammenhang zwischen Voxel-, bzw. Zellengröße und der Planungszeit.

Im ersten Testszenario muss der mobile Roboter seinen Weg durch zwei Tische finden, während die Armpose nicht verändert werden kann. Ein zusätzliches Hindernis auf einem der Tische schränkt die möglichen Rotationswinkel weiter ein. Die Resultate in Tab. 8.10 zeigen, wie sowohl die Planungszeit, als auch die Zeit der Kollisionsprüfung bei kleinerer Diskretisierung ansteigt, wobei die Pfadkosten annähernd gleich bleiben. Die Kosten setzen sich dabei aus der zurückzulegenden euklidischen Distanz, als auch den Rotationskosten zusammen.

Diskretisierung				**Laufzeit [s]**	
Zellgröße [m]	**Kantenlänge Voxel [m]**	**Kollisions-checks**	**Pfad-kosten**	**Kollisions-checks**	**Pfad-planung**
0,04	0,01	2651	2,472	5,73	3,621
0,04	0,02	2634	1,114	5,77	2,241
0,04	0,04	2670	0,546	5,95	1,692
0,08	0,01	745	0,608	6,17	0,787
0,08	0,02	720	0,292	5,86	0,471
0,08	0,04	757	0,139	5,99	0,322

Tab. 8.10.: Resultate der Planung mit Rotationsprimitiven im ersten Testszenario (vgl. Abb. 8.21a).

Im zweiten Szenario muss ein etwas längerer Pfad gefunden werden, wobei ein höherer Anteil an freiem Raum durchfahren wird. Hierbei zeigen sich die Vorteile der Kollisionsprüfung auf unterschiedlich aufgelösten Voxelkarten, wie an den Ergebnissen in Tab. 8.12 abzulesen ist: Trotz einer größeren Anzahl an Prüfungen ist die Laufzeit kürzer, da viele Checks im Freiraum stattfinden, und somit nur auf der niedrig aufgelösten Karte durchgeführt werden.

Diskretisierung				**Laufzeit [s]**	
Zellgröße [m]	**Kantenlänge Voxel [m]**	**Kollisions-checks**	**Pfad-kosten**	**Kollisions-checks**	**Pfad-planung**
0,04	0,01	3412	2,464	7,54	4,393
0,04	0,02	3417	1,060	7,55	3,007
0,04	0,04	3412	0,414	7,60	2,331
0,08	0,01	943	0,628	7,55	0,859
0,08	0,02	1009	0,299	7,56	0,560
0,08	0,04	1214	0,107	7,70	0,411

Tab. 8.11.: Resultate der Planung mit Rotationsprimitiven im zweiten Testszenario (vgl. Abb. 8.21b).

Das dritte Szenario nutzt eine Punktwolke, die mit einem rotierenden Laserscanner im Labor des FZI erstellt wurde. Da der Roboter auch hier eine enge Passage traversieren muss, konnte der Planer bei einer Auflösung von $0{,}08\,\mathrm{m}$ des Planungsgitters keinen gültigen Pfad finden. Erwartungsgemäß unterscheiden sich die Planungszeiten nicht von

den Szenarien mit synthetischen Daten, wie Tab. 8.12 belegt.

Diskretisierung				Laufzeit [s]	
Zellgröße [m]	**Kantenlänge Voxel [m]**	**Kollisions-checks**	**Pfad-kosten**	**Kollisions-checks**	**Pfad-planung**
0,04	0,01	1233	3,71	1,108	1,688
0,04	0,02	1208	3,72	0,414	1,011
0,04	0,04	1078	3,69	0,161	0,679

Tab. 8.12.: Resultate der Planung mit Rotationsprimitiven im dritten Testszenario auf realen Sensordaten (vgl. Abb. 8.21c).

Diese Evaluation zeigt, dass auch in verwinkelten Szenarien innerhalb weniger Sekunden valide und sogar glatte Trajektorien geplant werden können. Es ist hervorzuheben, dass die gezeigten Testfälle nicht mit einem 2D oder 2,5D Kollisionserkennungsverfahren lösbar sind, da sich die Arme des Roboters teilweise über den Hindernissen befinden müssen, oder die Plattform unter ihnen.

Mehrstufige Kollsionsprüfung in der Planung

Da die Listen, welche die Rotationsvolumen beinhalten, statisch sind, lassen sich diese in unterschiedlichen Auflösungen vorberechnen und für eine mehrstufige Kollisionsprüfung nutzen. Damit kann ein merklicher Laufzeitvorteil in der Planung erzielt werden. Folgende Formel beschreibt den Laufzeitvorteil (Speedup) gegenüber dem ausschließlich hochauflösenden Kollisionscheck:

$$Speedup = \frac{t_{\text{fein}} * n_{\text{Expand}}}{t_{\text{fein}} * m_{\text{fein}} + n_{\text{Expand}} * t_{\text{grob}}} \tag{8.3}$$

Dabei beziffert n_{Expand} die Anzahl der Expansionsschritte im Graphen während des Planungsprozesses, t_{fein} die Laufzeit der hochauflösenden Kollisionsprüfungen, die m_{fein} mal stattfinden und t_{grob} die Laufzeit einer niedrig auflösenden Kollisionsprüfung.

Nimmt man die Laufzeiten aus Abschnitt 8.2 als Ausgangspunkt, ergibt sich wiederum der Speedup aus Tab. 8.13 für die Bewegungsplanung.

Somit ist ersichtlich, dass eine Kombination mit möglichst unterschiedlichen Voxelgrößen den besten Laufzeitgewinn bewirkt. Gröbere Auflösungen als 4 cm wurden nicht betrachtet, da bei diesen zu viele Kollisionen im ersten Prüfungsschritt detektiert werden.

Zusammengefasst kann festgestellt werden, dass eine Voxelgröße von $4\,\text{cm}$ für die Planung von Plattformtrajektorien ausreichend ist. Durch den hierarchischen Ansatz ist dennoch sichergestellt, dass auch in engen Passagen keine Lösungen übersehen werden. Kleinere Auflösungen führen erwartungsgemäß zu einem starken Anstieg des Aufwands für die Kollisionsprüfung und somit zu längeren Planungszeiten. Weiterhin erwies sich

Feine Voxelgröße [m]	**Grobe Voxelgröße [m]**	**Expandierungs-schritte n_{Expand}**	**Anzahl feine Kollisionschecks m_{fein}**	**Speedup**
0,01	0,02	839	413	1,149
0,01	0,04	839	434	1,547
0,02	0,04	863	432	1,188

Tab. 8.13.: Laufzeitgewinn durch unterschiedliche Auflösungskombinationen der hierarchischen Kollisionsprüfung.

bei der Länge der mobilen Plattform von etwas über einem Meter ein Abstand im Planungsgitter von $8\,\text{cm}$ in Büroumgebungen als ausreichend. In verwinkelten Szenarien, wie z.B. einer Reinraum-Produktionsumgebung (vgl. Abb. 8.20) sollte diese auf $4\,\text{cm}$ reduziert werden.

(a) HoLLiE plant zunächst rechts am Hindernis vorbei, wird dann jedoch durch eine Person gezwungen, nach links auszuweichen.

(b) Hindernis auf Höhe des rechten Ellenbogens erfordert eine Rotation um die Engstelle quer zu passieren. Der linke Ellenbogen rotiert dabei über das schwarze Hindernis.

Abb. 8.22.: Planung mit rotierenden Bewegungsprimitiven auf dem Roboter HoLLiE. In beiden Beispielen konnte schnell genug umgeplant werden, so dass der Roboter seine Ausführung nicht stoppen musste.

Vergleich mit RRT-Connect Planer

Als Vergleich zum hier vorgestellten Fahrtplanungsverfahren soll die RRT-Connect-Implementierung der OMPL herangezogen werden. Für die Kollisionsprüfung kommt jedoch aus praktischen Gründen nicht das hierarchische Verfahren mit der Translation durch Basisversatz zum Einsatz, sondern ein weit weniger performantes Verfahren, bei dem die Punktwolke des Roboters bei jeder Bewegung neu in eine Voxelkarte eingetragen wird. Um dennoch einen direkten Vergleich ziehen zu können, wird hier lediglich die Anzahl der auszuführenden Kollisionsprüfungen berücksichtigt und nicht deren Laufzeit.

Ein Vergleich der Laufzeit mit einer Mesh-basierten Kollisionsprüfung wurde dennoch durchgeführt, indem die Anzahl der Prüfungen mit den durchschnittlichen Laufzeiten einer vergleichbar komplexen Kollisionsprüfung multipliziert werden. Dafür wurde mit einem Messwert von $0{,}24\,\mathrm{s}$ gerechnet, den die weit verbreiteten FCL Bibliothek laut [156] für 1000 Kollisionsprüfungen benötigt (dies deckt sich auch mit eigenen Test).

Die Kollisionsprüfung beider Planer fand auf $4\,\mathrm{cm}$ Voxeln statt. In Szenario 1 und 2 wurden $8\,\mathrm{cm}$ Gitterabstand genutzt, in Szenario 3 hingegen $4\,\mathrm{cm}$. Die Parametrierung des RRT-Connect wurde zuvor empirisch optimiert.

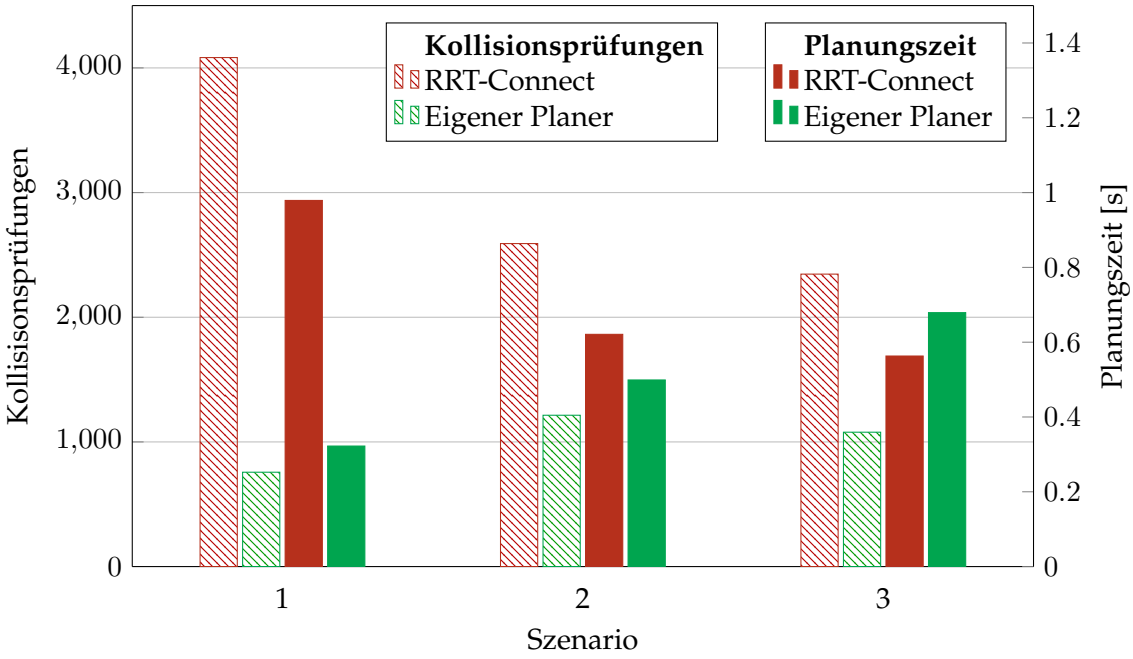


Abb. 8.23.: Vergleich des Plattform Planers dieser Arbeit mit RRT-Connect aus der OMPL Bibliothek.

Das Diagramm in Abb. 8.23 zeigt, dass bei dem selbst entwickelten Plattformplaner die Planungszeit weniger von der Komplexität des Szenarios abhägt, als bei RRT-Connect. So ist die eigene Implementierung im ersten, kurvigeren Szenario wesentlich schneller, wohingegen im dritten Szenario der RRT-Connect bei einer kürzeren euklidischen Distanz effizienter ist. Dieses Ergebnis ist exemplarisch und deckt sich mit weiteren durchgeführten Versuchen. Weiterhin wird die Kollisionsprüfung durch den eigenen Planer erheblich

seltener aufgerufen. Dabei ist jedoch zu berücksichtigen, dass ein einzelner Aufruf alle Rotationen simultan evaluiert, was bei den Samples der RRT-Connect nicht der Fall ist.

Bei der Beurteilung eines Planers sollte jedoch neben der Berechnungsdauer auch die Pfadlänge und die Glattheit der entstehenden Lösungen beurteilt werden. So fällt bei der Betrachtung von Abb. 8.21b und Abb. 8.21d auf, wie glatt die Pfade des implementierten Planers ohne weitere Nachbearbeitungsschritte ausfallen. Der samplingbasierte RRT-Connect generiert im freien Raum dagegen mehrere unnötige Knicke, die den Pfad zusätzlich verlängern.

Weiterverwendung von Teilplänen

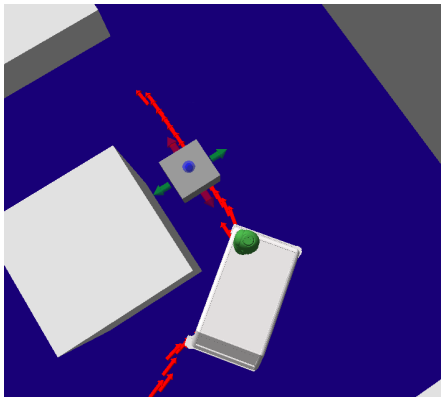
Durch die Verwendung eines D*-Lite Planers ist es möglich, Teilpläne bzw. Graphenknoten weiter zu verwenden, auch wenn geänderte Umweltdaten eine Änderung der Trajektorie verlangen. Um diesen Vorteil bestmöglich zu nutzen, expandiert der Algorithmus auch nach dem Finden eines Plans weiterhin systematisch Graphenknoten. Da der gefundene Plan währenddessen bereits ausgeführt werden kann, erzeugt dies keine Laufzeitnachteile. Im Falle einer Neuplanung stehen so jedoch bereits umfangreichere, nutzbare Informationen für die Graphensuche zur Verfügung.

Zwei exemplarische Versuche, in denen dynamische Hindernisse einen Plan unausführbar machen, sind in Abb. 8.24 gezeigt. Ein Vergleich der Adaptions- bzw. Neuplanungszeiten mit neuen Umweltinformationen ist in Tab. 8.14 zu sehen. Dabei zeigt sich, dass die erreichbaren Zeiteinsparungen erwartungsgemäß mit der Länge des neuen Pfades steigen, auch wenn viele Graphenknoten aktualisiert werden müssen (rote Punkte in den Grafiken).

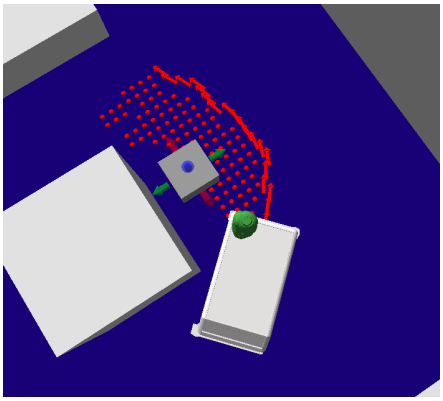
	Laufzeit [s]	
Szenario	**Kollisionsprüfung**	**Planung**
Szenario 4: Neuplanung	0,304	0,605
Szenario 4: Weiterverwendung	0,299	0,567
Szenario 5: Neuplanung	0,763	1,511
Szenario 5: Weiterverwendung	0,362	0,947

Tab. 8.14.: Laufzeitgewinn durch Weiterverwendung von Graphenknoten bei der Planung mit geänderten Umweltinformationen.

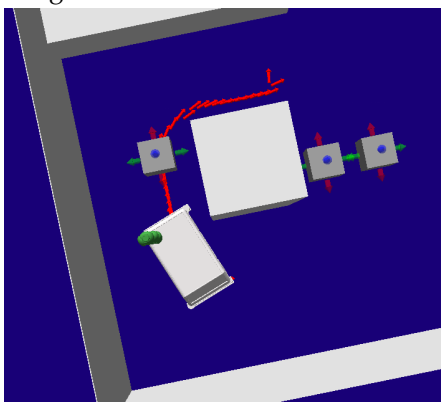
Nach diesen Versuchen in einer ROS-basierten Simulation wurde der Planer auch erfolgreich in einer realen Umgebung mit dem Roboter HoLLiE getestet. Ausschnitte aus zwei Versuchen sind in Abb. 8.22 zu sehen. Im ersten Szenario blockierte eine Person mehrfach den geplanten Pfad des mobilen Roboters, der um ein statisches Hindernis herum führte, so das dynamisch neue Lösungen generiert werden mussten. Da dies in weniger als einer Sekunde möglich war, konnte der Roboter übergangslos zwischen den Trajektorien umschalten, ohne dabei komplett zum Halten zu kommen. Dies funktionierte auch im zweiten, wesentlich engeren Szenario. Hier führte der ursprünglich geplante Pfad den Roboter vorwärts gerichtet durch einen engen Korridor. Während der Ausführung wurde der



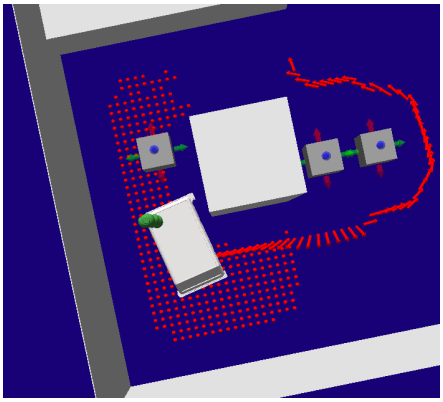
(a) Szenario 4: Ursprünglich geplante Trajektorie. Dynamisches Hindernis wird gerade wahrgenommen.



(b) Neuer Pfad mit Ausweichbewegung.



(c) Szenario 5: Ursprünglich geplante Trajektorie links am statischen Hindernis vorbei. Zwei dynamsiche Hindernisse rechts sind bereits bekannt, dynamisches Hindernis links wird neu wahrgenommen.



(d) Neuer Pfad rechts an allen Hindernissen vorbei. Das Beispiel zeigt auch die Fähigkeit des Planers, rückwärts gerichtete Bewegungen zu planen, wenn keine Alternativen bestehen.

Abb. 8.24.: Weiterverwendung von Teilplänen, nachdem neue Hindernisse erkannt wurden. Rote Punkte markieren inkonsistente Graphenknoten, deren kürzester Pfad zum Ziel ungültig wurde. Veröffentlicht in [3].

Korridor durch ein zusätzliches Hindernis auf Höhe des Ellenbogens versperrt. Der adaptierte Pfad beinhaltete daraufhin eine 90 Grad Drehung, mittels der der Roboter dann quer zur Fahrtrichtung die neue Hindernissituation passieren konnte. Das Umweltmodell stammte in beiden Versuchen ausschließlich aus Punktwolkendaten aus der im Kopf von HoLLiE montierten Kinect-Kamera. Da der Roter über keine GPU verfügt, wurden die Berechnungen auf einem externen Planungscomputer durchgeführt, der die Umwelt- und Lokalisierungsdaten über eine Netzwerkverbindung erhielt. Berechnete Pläne wurden dann wiederum an eine ROS-Komponente auf dem Roboter gesendet, welche zum passenden Zeitpunkt zwischen altem und neuem Plan wechselte.

Zusammenfassung

Die dargestellten Ergebnisse bestätigen die Laufzeiteinsparungen durch die Verwendung von Swept-Volumen aus Bitvektor-Voxeln. Das verwendete Planungsverfahren mit Rotationsprimitiven ist ein optimaler Anwendungsfall, nicht nur für kumulative Kollisionsprüfungen, sondern auch für die versatzbasierte Kollisionsprüfung zwischen Voxellisten und Voxelkarten. Diese konnte durch hierarchische Prüfung mit unterschiedlichen Auflösungen noch weiter beschleunigt werden.

Durch den D*-Lite Suchalgorithmus können Teilpläne weiterverwendet werden, womit sich das Verfahren auch gut für dynamische Umgebungen eignet. Da bei der Planung alle drei Dimensionen des Konfigurationsraums berücksichtigt werden, erzeugt das Verfahren glatte Trajektorien für holonome Plattformen, während dennoch intuitive Fahrbewegungen mit Vorzugsrichtung präferiert werden. Als Erweiterung wurde bereits in Abschnitt 7.2.2 gezeigt, wie sich mit Hilfe von Swept-Volumen des Manipulatorarmes auch erfolgversprechende Plattformposen für mobile Manipulationsaufgaben bestimmen lassen.

Die Ergebnisse dieses Abschnittes wurden in [3] veröffentlicht.

8.7.3. Planung mit generischen Bewegungsprimitiven

Bevor die Planung mittels Bewegungsprimitiven, wie sie in Unterabschnitt 7.2.3 entwickelt wurde, in einem realen Szenario evaluiert werden konnte, mussten umfangreiche Tests in der Simulation durchgeführt werden. Geprüft und optimiert wurden hierbei einerseits die eingesetzten Primitive, so dass am Ende auch lange Pfade in verschachtelten Umgebungen in wenigen Sekunden erzeugt werden konnten. Zum anderen aber auch der rechtzeitige Wechsel zwischen Pfaden, der ein mehrstufiges Protokoll erfordert. Hierfür wurde ein einarmiges Modell der IMMP Plattform verwendet, das mit vier virtuellen Tiefenkameras ausgestattet war. Deren Punktwolken wurden anhand der simulierten Kameraperspektive aus einem gegebenen 2,5D Modell der Umwelt berechnet (siehe Abschnitt 4.7).

Momentaufnahmen aus zwei Simulationsdurchläufen finden sich in Abb. 8.25. In beiden Szenarien musste der Roboter durch eine, vorerst unbekannte, Umwelt navigieren und dabei die geplanten Trajektorien mehrfach aufgrund neu detektierter Hindernisse anpassen. Planungszeiten des zweiten Szenarios sind in Tab. 8.15 angegeben. Dabei

spiegelt jede Zeile eine neue Trajektorie wider, wobei sich die Dauer der Planung aus mehreren Komponenten zusammensetzt: Angegeben sind die durchgeführten Expandierungen des Suchgraphen, für die jeweils die Voxelliste eines kompletten Fächers aus Bewegungsprimitiven durch eine Kollisionsprüfung mit der Umwelt zu evaluieren ist. Die verbleibende Zeit ist hauptsächlich dem Aufbau einer neuen Voxelliste zur Ausführungsüberwachung aus allen verwendeten Primitiven geschuldet. Deren kontinuierliche Kollisionsprüfung ist wiederum in Tab. 8.16 aufgeschlüsselt. Angegeben sind hier die ausgeführten Prüfungen pro Abschnitt, die multipliziert mit den Segmenten die Anzahl der überwachten Primitive ergeben. Hier zeigt sich, dass die Ausführung im Median mit 33 Hz auf neue Hindernisse hin überwacht werden kann, was sogar über der Kinect-Bildrate liegt.

Setzt man die gemessenen Zeiten mit der durchschnittlichen Bewegungsgeschwindigkeit von $0{,}7\,\mathrm{m/s}$ des Roboters in Relation wird klar, dass die Planung im Normalfall schnell genug abläuft, um statischen Hindernissen im Fahrtkorridor ausweichen zu können, ohne dabei anhalten zu müssen: Da die Sichtweite der Kinect mindestens 4 Meter beträgt, und die Latenz der Datenverarbeitung vernachlässigbar ist (ca. 60 ms), ergibt sich bis zu einer Kollision eine verbleibende Fahrtdauer von ca. 7,7 Sekunden. In realistischen Szenarien werden dagegen bereits in unter 3 Sekunden alternative Trajektorien berechnet, was ausreichend Zeit für einen Wechsel in der Ausführung lässt. Selbstverständlich gilt diese Annahme nicht, wenn bei einer Kurvenfahrt neue Hindernisse direkt hinter einer Ecke erkannt werden. In diesem Fall muss der Roboter gestoppt werden und kann erst nach der Planung weiterfahren.

(a) Start- und Zielpose sind in gelb dargestellt, die aktuelle Roboterpose in rot.

(b) Start- und Zielpose, sowie die aktuelle Roboterpose sind in grün dargestellt.

Abb. 8.25.: Dynamische Adaption einer Plattformtrajektorie des IMMP Roboters an neue Umweltinformationen bei der Planung mit Bewegungsprimitiven.

Abschnitt	Länge [Primitive]	Planung [s]	Expandierung Dauer [ms]	Expandierung Schritte	Dauer pro Prüfung [ms] ∅	Median	Min	Max
1	4	2,220	8,63	3	2,86	2,36	2,28	3,95
2	8	11,663	9727,53	101	96,30	9,50	2,22	732,58
3	3	2,119	27,09	3	8,99	7,59	2,20	17,18
4	7	2,160	42,47	7	6,06	2,86	2,20	25,51

Tab. 8.15.: Berechnungsdauer der Planung mittels Bewegungsprimitiven in unbekannter Umgebung aus Abb. 8.25b. Pro Schritt in der Expandierung wird ein vollständiger Bewegungsfächer auf Kollision geprüft.

Abschnitt	Kollisionsprüfungen	Evaluierte Primitive	Dauer pro Prüfung [ms] ∅	Median	Min	Max
1	88	352	182,43	9,53	2,21	704,76
2	150	1200	250,91	29,55	2,54	786,85
3	74	222	214,71	16,18	2,18	743,85
4	190	1330	254,78	25,79	2,38	779,71

Tab. 8.16.: Ausführungsüberwachung der geplanten Trajektorienabschnitte aus Abb. 8.25b.

Nach den erfolgreichen Tests in der Simulation wurde das Verfahren auch mit HoLLiE in realen Szenarien erfolgreich demonstriert (siehe Abb. 8.26). Der Roboter musste dabei sowohl um statische, als auch um dynamische Hindernisse navigieren, wobei seine einzigen Datenquellen die Kinect im Kopf, sowie die Radodometrie waren. Auch bei diesen Versuchen in der Laborumgebung des FZI betrug die Geschwindigkeit ca. $0{,}7\,\mathrm{m/s}$. Wie bei der Planung mit Rotations-Swept-Volu-men mussten die Berechnungen auf einem externen Computer durchgeführt werden.

Die Ergebnisse erfüllen alle Erwartungen und belegen die Vorteile der kontinuierlichen Trajektorienüberwachung mittels Swept-Volumen. Traten in der Umgebung neue Hindernisse in einem Abstand von mindestens einem Meter vor dem Roboter auf, war kein Anhalten der Plattform nötig, um auszuweichen, da alternative Pfade schnell genug geplant werden konnten. Andere Hindernisse, auch in direkter Nähe zum Roboter, lösten hingegen keine Neuplanung aus, so lange sie außerhalb des zu durchquerenden Volumens lagen.

Abb. 8.26.: Reaktive Planung um eine Person mittels Bewegungsprimitiven auf dem Roboter HoLLiE. Die dynamische Neuplanung erfolgt schnell genug, so dass der Roboter nicht stoppen muss.

8.8. Evaluierung der Bewegungsprädiktion

Der folgende Abschnitt untersucht die Praxistauglichkeit der Bewegungsprädiktion aus Abschnitt 4.6 anhand mehrerer Versuche. Um die angestrebte Verarbeitungsgeschwindigkeit von mehreren Berechnungen pro Sekunde zu erreichen, mussten starke Reduktionen der Eingabedaten in Kauf genommen werden. Daher sollen zunächst ihre Auswirkungen betrachtet werden:

8.8.1. Datenbasis

Da die genutzten Algorithmen zunächst pixelweise arbeiten, liegt es nahe, die benötigte Rechenzeit durch eine Verkleinerung der Eingabebilder zu reduzieren. Hieraus ergeben sich zwar per se keine qualitativen Einbußen bei den Ergebnissen, jedoch sinkt mit der Auflösung auch die minimale Größe der detektierbaren Objekte. Dass dies in den untersuchten Szenarien keine praktische Einschränkung darstellt, wurde empirisch anhand von Hindernissen unterschiedlicher Größe ermittelt.

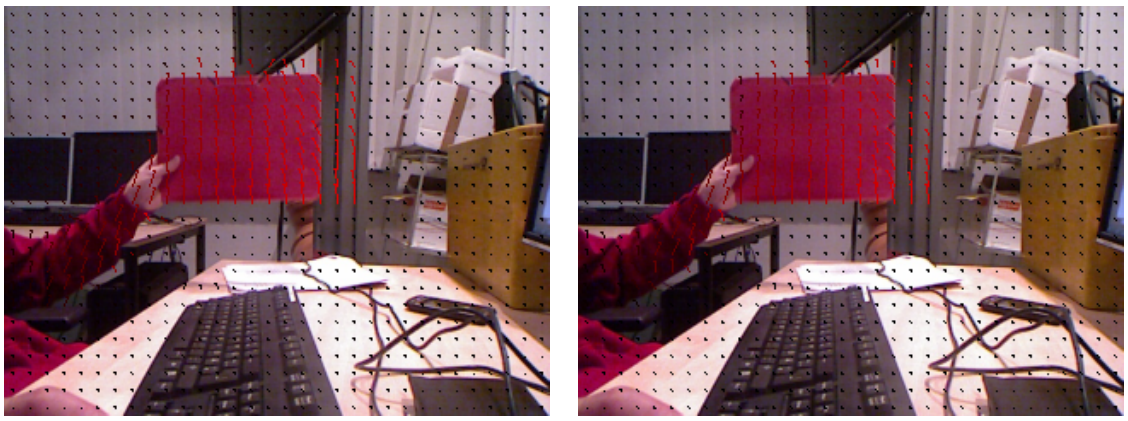
Wie in der Masterarbeit von Mauch [27] beschrieben, kann weiterhin durch die Reduktion der maximalen Durchgänge der Energieoptimierungsfunktion und ihrer inneren SOR-Optimierungszyklen bei gleichzeitigem Aufweichen der Abbruchkriterien eine Laufzeitreduktion um ca. Faktor 7 erreicht werden (siehe Tab. 8.17). Auch hier lassen sich die Qualitätseinbußen der Änderungen visuell gut beurteilen: Abb. 8.27 zeigt, dass die Abweichungen der berechneten Bewegungsvektoren auf denselben Eingabedaten marginal ausfallen.

Auflösung	**Parameter**	**Laufzeit**			
		⌀	**Median**	**Min**	**Max**
320×240	Original	3723	3046	1414	22 665
320×240	Modifiziert	441	429	390	582
160×120	Original	1226	1103	405	4111
160×120	Modifiziert	177	176	162	236

Tab. 8.17.: Vergleich der RGBD-Flow Laufzeiten auf unterschiedlichen Eingabedimensionen mit und ohne Modifikationen. Daten aus [27]. Ermittelt über 83 Bildübergänge.

8.8.2. Experimente

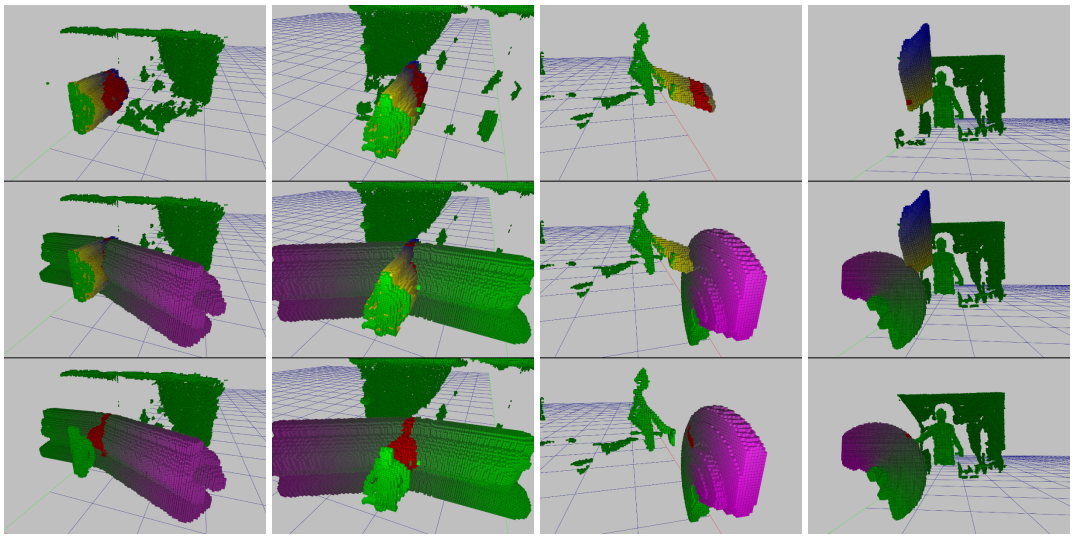
Um die berechneten Bewegungen quantitativ beurteilen zu können, wurden Versuche durchgeführt, bei denen ein Objekt einerseits anhand eines künstlichen Markers und andererseits mittels der in Abschnitt 4.6 beschriebenen Verarbeitungskette getrackt wurde. Das Objekt war dafür an einem durchsichtigen Stab befestigt, so dass es damit vor der Kamera bewegt werden konnte. Für die Ermittlung der Referenzdaten kam das ROS-Paket



(a) RGBD-Flow mit Originalparametern. Laufzeit: 5,649 Sekunden

(b) RGBD-Flow mit modifizierten Parametern. Laufzeit: 0,594 Sekunden

Abb. 8.27.: Qualitativer Vergleich des RGBD-Flow-Vektorfeldes mit unterschiedlichen Parametrisierungen. Die rote Färbung der Pfeile symbolisiert eine sich entfernende Bewegung. Bild aus [27].



(a) Prädizierte Kollision zwischen mobilem Roboter und Person

(b) Prädizierte Kollision zwischen mobilem Roboter und Person

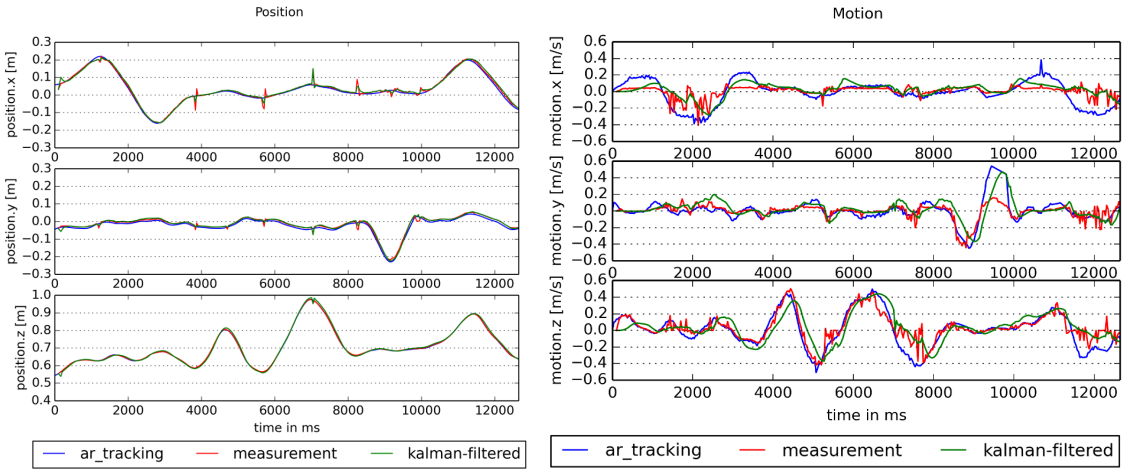
(c) Prädizierte Kollision zwischen Roboterarm und Person

(d) Prädizierte Kollision zwischen Roboterarm und Person

Abb. 8.28.: Momentaufnahmen aus vier Experimenten zur prädizierten Kollisionserkennung zwischen mobilem bzw. stationärem Roboter und einer Person. Die Bilder einer Spalte zeigen denselben Zeitpunkt. Oben: Prädiktion des Menschen, Mitte: Schnitt mit dem Bewegungsplan des Roboters. Unten: In Kollision liegendes Volumen (rot). Veröffentlicht in [7].

ar_track_alvar[4] zum Einsatz, dessen gemessene $SE(3)$-Posen über den Differenzenquotienten in Geschwindigkeiten umgerechnet wurden.

Die Ergebnisse des Vergleichs, welche in Abb. 8.29 zu sehen sind, wurden in [7] veröffentlicht. Sie zeigen in Abb. 8.29a für alle drei Dimensionen eine sehr geringe Abweichung der berechneten Position. Ausnahmen sind kleine Spitzen an den Hoch- / Tief- und Sattelpunkten des Graphen. Zu diesen Zeitpunkten sinkt die Objektgeschwindigkeit nahezu auf null, weswegen der RGBD-Flow einen Großteil der Objektmesspunkte aufgrund ihrer zu geringen Geschwindigkeit bereits vor der Segmentierung verwirft. Das Rauschen in den wenigen verbleibenden Punkten führt dann zu den fehlerhaften Ergebnissen. Diese sind für eine weiterführende Verarbeitung allerdings nicht von Belang und wurden daher auch nicht weiter untersucht.



(a) Ergebnis der Positionsbestimmung

(b) Ergebnis der Bewegungsbestimmung

Abb. 8.29.: Evaluation des Objekttrackings mittels 3D-Szenenfluss. Der blaue Graph visualisiert die Referenzdaten des Marker-Trackings. Rot ist das Messergebnis aus dem Szenenfluss, grün das Kalman-gefilterte Tracking. Veröffentlicht in [7].

Das Ergebnis der Bewegungsgeschwindigkeiten aus Abb. 8.29b ist hingegen wesentlich stärkeren Abweichungen unterworfen. Die Filterung mittels EKF glättet zwar die Schwankungen in den Messungen, führt jedoch prinzipbedingt auch zu einer zeitlichen Verzögerung des Signales. Der grobe Bewegungsverlauf wird allerdings ausreichend gut wiedergegeben, so dass Experimente mit Objekten einer gewissen Massenträgheit erfolgreich durchgeführt werden konnten. Eine statistische Auswertung der Abweichung von Positions- bzw. Bewegungsschätzung von den Referenzdaten in Tab. 8.18 und Tab. 8.19 zeigt dieselben Ergebnisse im Detail. $\bar{e}$ bezeichnet hier das arithmetische Mittel und *RMS* das quadratische Mittel des Fehlers.

In einem weiteren Testszenario mit einem bewegten Objekt, das ca. 20% des Kamerabildes ausfüllte, wurden Laufzeitmessungen der einzelnen Funktionsblöcke durchgeführt. Die Ergebnisse in Tab. 8.20 zeigen, dass die Szenenflussberechnung die höchste Berechnungsdauer aufweist. Da jeder Funktionsblock in einem eigenen Thread abläuft,

[4] Siehe `http://wiki.ros.org/ar_track_alvar`

bestimmt somit die Laufzeit der Flussberechnung t_{flow} ≈125 ms die maximal erreichbare Framerate f_{flow} des Systems, die bei $\frac{1}{t_{flow}}$ ≈8 Hz liegt. Die Reaktionszeit ergibt sich aus der Summe aller Verarbeitungsschritte und liegt somit bei ∼200 ms.

Die praktische Anwendbarkeit der Kollisionsprädiktion konnte in zwei Szenarien erprobt werden: Ein geteilter Mensch-Roboter-Arbeitsplatz aus Abb. 8.30 demonstriert die Erkennung und Prädiktion menschlicher Gliedmaßen, während ein zweites Szenario die Prädiktion menschlicher Bewegungen aus größerer Entfernung testete (Abb. 8.32). Als Roboter kam in beiden Fällen HoLLiE zum Einsatz, die ihren Arbeitsraum mittels der im Kopf eingebauten Kinect-Kamera überwachte. Die Bewegung des Roboters liegt nach ihrer Planung bereits als Swept-Volumen vor, welches mit den oben beschriebenen Verfahren auf eine Kollision mit dem Volumen der prädizierten Bewegung des Menschen überprüft wird. Beide Szenarien wurden mehrfach erfolgreich getestet: Der Roboter unterbrach seine Bewegungen bereits lange, bevor es zu einer Kollision kommen konnte.

Abb. 8.30.: Beispielszenario: Geteilter Mensch-Roboter-Arbeitsplatz. Bild aus [27].

8.8.3. Einschränkung und mögliche Erweiterungen

Die Laufzeit des verwendeten Szenenflussverfahrens ist der einschränkende Faktor bei der Auswertung dynamischer Bewegungen, was Raum für Optimierungen lässt. Weiterhin stellen die rein linearen Bewegungsmodelle des EKF eine große Einschränkung dar, insbesondere bei der Betrachtung menschlicher Armbewegungen. Hier wäre es hilfreich, auf skelettbasierte Bewegungsmodelle zu wechseln, was jedoch dem ursprünglichen Gedanken eines möglichst allgemeingültigen Ansatzes widersprechen würde.

Ohne Widersprüche ließen sich hingegen Strategien entwickeln, die den Roboter im Falle einer prädizierten Kollision proaktiv reagieren ließen. Hier wäre es je nach Szenario denkbar, die unterbrochene Bewegung rückwärts auszuführen, um Zusammenstöße zu verhindern, oder den Roboter in eine weiche Impedanzregelung zu versetzen, um ein Verletzungsrisiko zu minimieren.

Soll das entwickelte Verfahren auf einem mobilen Roboter zum Einsatz kommen, bei dem die Kamera nicht statisch verbleibt, müssen robuste und exakte Verfahren gefundene werden, um die Eigenbewegung aus dem Vektorfeld des Szenenflusses herauszu-

	$\bar{\mathbf{e}}_{\mathbf{pos}}$ [m]	$\mathbf{RMS}_{\mathbf{pos,x}}$ [m]	$\mathbf{RMS}_{\mathbf{pos,y}}$ [m]	$\mathbf{RMS}_{\mathbf{pos,z}}$ [m]	$\mathbf{e}_{\mathbf{pos,min}}$ [m]	$\mathbf{e}_{\mathbf{pos,max}}$ [m]
Nach Segmentierung	0,0101	0,0121	0,0099	0,0020	0,0091	0,1392
Nach EKF	0,0145	0,0116	0,0100	0,0062	0,0070	0,1773

Tab. 8.18.: Fehler der Positionsschätzung aus [27].

	$\bar{\mathbf{e}}_{\mathbf{mot}}$ [m/s]	$\mathbf{RMS}_{\mathbf{mot,x}}$ [m/s]	$\mathbf{RMS}_{\mathbf{mot,y}}$ [m/s]	$\mathbf{RMS}_{\mathbf{mot,z}}$ [m/s]	$\mathbf{e}_{\mathbf{mot,min}}$ [m/s]	$\mathbf{e}_{\mathbf{mot,max}}$ [m/s]
Nach Segmentierung	0,1392	0,1149	0,0683	0,1032	0,0192	0,5292

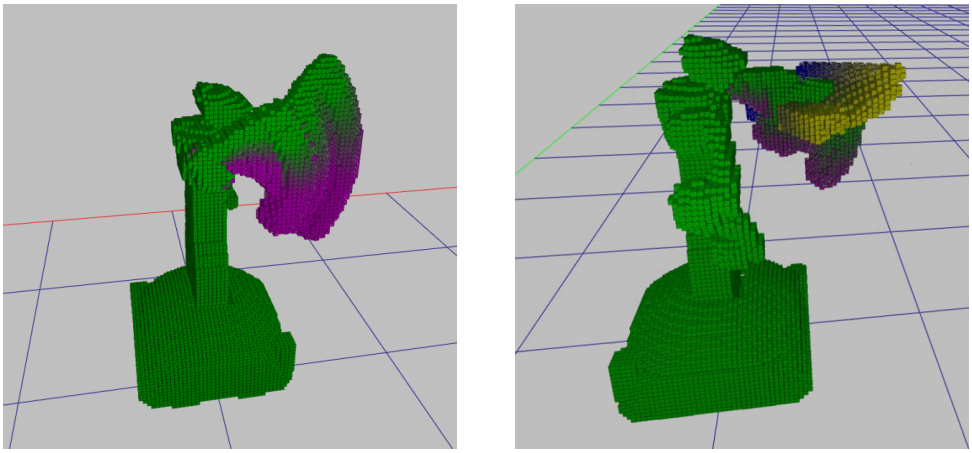
Tab. 8.19.: Fehler der Bewegungsschätzung aus [27].

Thread	**Laufzeit [ms]**				**Durchläufe**
	⌀	**Median**	**Min**	**Max**	
Vorverarbeitung	9,588	9,113	7,035	14,363	523
RGBD-Flow	125,955	125,623	91,312	184,952	467
SV-Rendering	8,519	8,284	5,518	19,513	441
Kollisionscheck	51,557	50,621	49,121	54,628	523
Gesamtlaufzeit	195,619	193,641	152,986	273,456	

Tab. 8.20.: Laufzeiten der einzelnen Threads. Aus der Laufzeit des RGBD-Flow-Threads ergibt sich eine Framerate von ca. 8 Hz, während die Reaktionszeit im Durchschnitt unter 200 ms liegt.

Szenario	**Laufzeit Median [ms]**	**Framerate [Hz]**
Kontinuierliche Bewegung mehrerer Objekte	158,149	6,32
25% des Eingangsbildes ohne Bewegung	130,812	7,64
Wakeln der Kamera, 100% Bewegung	166,882	5,99

Tab. 8.21.: Laufzeit der Bewegungsprädiktion, abhängig von der Menge an Bewegungen im Bild. Gemessen in drei Szenarien über je 400 Bildübergänge, ohne Kollisionsprüfung, aber mit Visualisierung.



(a) Geplante Roboterbewegung

(b) Kollision mit prädizierter menschlicher Bewegung

Abb. 8.31.: Geteilter Arbeitsraum: Die Trajektorie des linken Roboterarmes verläuft von oben nach unten, ihr Swept-Volumen von grün nach magenta. Die prädizierte menschliche Bewegung verläuft orthogonal dazu und ist von gelb nach blau eingefärbt. Da zwischen beiden Swept-Volumen eine Kollision herrscht, unterbricht der Roboter seine Ausführung, bevor es in der Realität wahrscheinlich zu einer Kollision gekommen wäre. Bild aus [27].

rechnen. Vielversprechend erscheint hier die Nutzung von visueller Odometrie, um Fehlerquellen durch Radodometrie und Sensorsynchronisation zu vermeiden. Hintergründe dazu wurden in Unterabschnitt 4.6.8 gegeben.

Letztendlich wäre ein probabilistischer Ansatz bei der Generierung des prädizierten Swept-Volumens denkbar, bei dem die Objektpunktewolke während ihrer Verschiebung entlang des prädizierten Bewegungsvektors vergrößert wird, um so der zunehmenden Unsicherheit gerecht zu werden. Hierfür wären die EDT-Algorithmen aus Abschnitt 5.6 denkbar.

8.8.4. Zusammenfassung

In diesem Abschnitt wurde eine vollständige Verarbeitungskette evaluiert, die prädizierte Bewegungen zur Kollisionsdetektion nutzt, und somit der Kollisionsprädiktion aus Definition 4 entspricht.

Die Umsetzung basiert auf einem 3D-Szenenfluss, dessen Ausgabevektorfelder in dynamische, nichtrigide Objekte segmentiert werden. Ein erweiterter Kalmanfilter trackt die Segmente, so dass ihre Bewegungen in die Zukunft extrapoliert und als Swept-Volumen gerendert werden können. Die Vorhersagen wurden in praktischen Versuchen erfolgreich genutzt, um die Sicherheit eines Roboters zu erhöhen, indem dieser seine geplanten Bewegungen gegenüber den Vorhersagen prüft, und abbremst, lange bevor ein Mensch seine Bahnen kreuzt. Das onlinefähige Verfahren läuft mit ca. 8 Hz und weist eine Reaktionszeit von ca. 200 ms auf. Es wurde in [7] veröffentlicht.

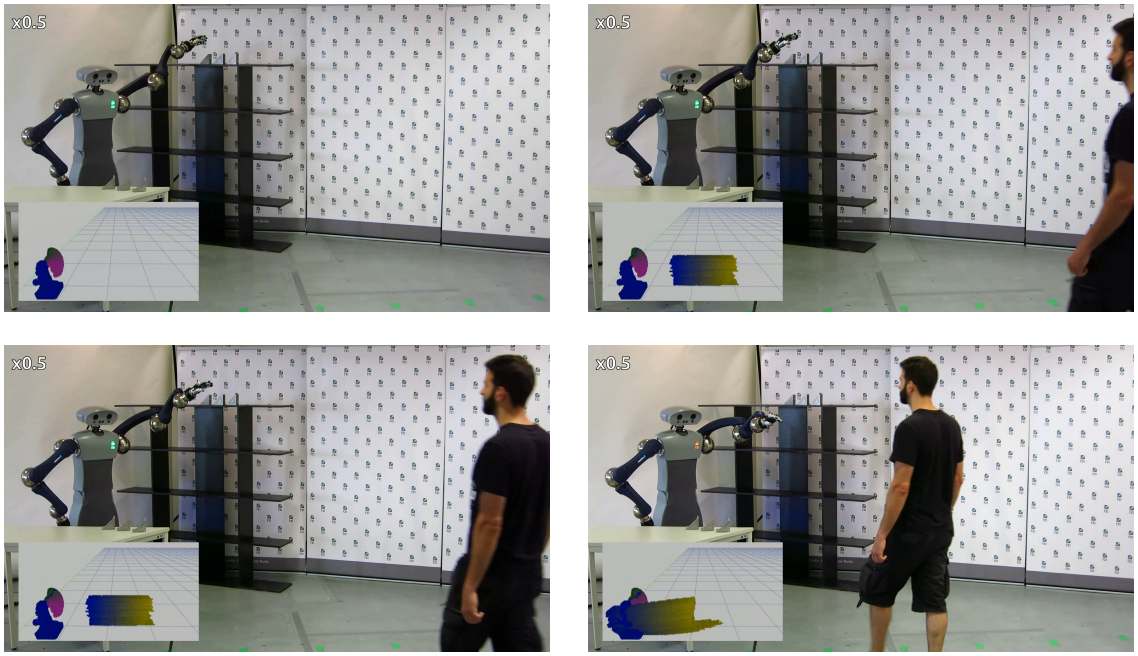


Abb. 8.32.: Der linke Roboterarm bewegt sich zwischen Regal und Tisch, sein Swept-Volumen verläuft von grün nach magenta. Die prädizierte menschliche Bewegung ist von gelb nach blau eingefärbt. Da zwischen beiden Swept-Volumen eine Kollision herrscht, unterbricht der Roboter seine Ausführung (erkennbar an der roten Brust-LED), bevor es in der Realität wahrscheinlich zu einer Kollision gekommen wäre.

8.9. Experimente zur Onlineberechnung von 3D-Distanzkarten

Bei der Untersuchung von Verfahren zur Erzeugung von Distanzfeldern müssen maßgeblich zwei Kriterien betrachtet werden: Die Berechnungsdauer und die Genauigkeit der Ergebnisse. Da es sich beim umgesetzten Verfahren nicht um einen approximativen Algorithmus handelt, konnten die Messergebnisse mit den Referenzdaten aus einer kanonischen Berechnung (siehe Abschnitt 5.6.3) abgeglichen und für korrekt erklärt werden.

Für eine Beurteilung der Berechnungsdauer standen die Implementierungen von PBA, JFA und der kanonischen Methode zur Verfügung, also explizit nur „statische“ Verfahren. Dies ist mit ihrer guten Parallelisierbarkeit und der einhergehenden Laufzeit begründet, welche „dynamische“ Verfahren aussticht. Alle Algorithmen arbeiten mit 32 Bit-Distanz-Voxeln (vgl. Unterabschnitt 5.1.3), die in jedem Voxel eine Referenz zu ihrem nächstgelegenen Hindernis speichern.

Neben dem reinen Benchmarking der Algorithmen wurden zur praktischen Evaluierung der Distanzfelder auch zwei reale Szenarien untersucht: Die Navigation einer Flugdrohne und die Berechnung der inversen Kinematik eines mobilen Manipulators unter Optimierung des Freiraumes. Die Umwelt wird hier als Punktwolke wahrgenommen, wobei als Datenquelle reale oder simulierte Kinect-Kameras eingesetzt wurden. Die 3D-Daten

Größe Voxelkarte	Laufzeit [ms]			Durchsatz [MVoxel/s]		
	JFA	PBA	PBA (orig)	JFA	PBA	PBA (orig)
256^3 Voxel	255,7	24,9	10,2	230	673	1315
512^3 Voxel	2325,8	101,3	64,1	57	1325	2093

Tab. 8.22.: Laufzeiten und Durchsatz von drei EDT Algorithmen auf dreidimensionalen Karten. Die kleine Karte (256^3) enthält 67 625 Hindernisvoxeln, die große (512^3) 89 295 Hindernisvoxel. PBA (orig) bezieht sich auf die Referenzimplementierung zu [54], während PBA und JFA die eigenen Implementierungen darstellen. Die Messwerte repräsentieren den Median über 20 Durchläufe.

werden auf der GP-GPU in Weltkoordinaten transformiert und als Hindernisse in die Distanzkarte eingefügt.

Ausgehend von einer Kartengröße von 5 m $\times$ 5 m $\times$ 5 m bei einer Auflösung von 2 cm sind die Distanzen von 256^3 (also über 16,7 Millionen) Voxeln zu berechnen. Um die Daten einer Kinect-Kamera schritthaltend verarbeiten zu können, beträgt die dafür angestrebte Latenz 33 ms. Somit sollte die EDT bei der gegeben Kartengröße einen Berechnungsdurchsatz von 500 Millionen Voxeln (MVoxel) pro Sekunde erbringen. Zum Vergleich: Das approximative Verfahren SKW erreicht auf Karten aus 512^3 Voxeln laut Cao et al. [54] auf einer Nvidia Tesla C1060 GPU einen Berechnungsdurchsatz von 134 MVoxel/s.

Laufzeitmessungen

Tab. 8.22 zeigt die Laufzeiten bzw. den Durchsatz unterschiedlicher EDT Implementierungen. JFA liegt hier mit steigender Kartengröße um Faktor 3 bis 23 hinter PBA, während die Referenzimplementierung nochmals nahezu doppelt so performant ist, wie die Umsetzung in GPU-Voxels. Der Laufzeitnachteil ist durch den Verzicht auf Texturspeicher (siehe Abschnitt 3.2.3) zu Gunsten der Codetransparenz begründet, was aber in späteren Programmversionen änderbar ist. Dennoch kann die angestrebte Latenz übertroffen werden.

Wie bereits in [54] beschrieben, haben die Parameter m_1 und m_2 den größten Einfluss auf die Berechnungszeiten des PBA, wenn die Dimension der Eingabedaten, verglichen mit den verfügbaren CUDA Threads, klein ausfällt. Ist dies zum Beispiel bei 2D Problemen der Fall, können die Bänder durch große m_1 und m_2 noch weiter aufgeteilt und somit besser parallelisiert werden. Da die Eingabedaten in dieser Arbeit aber dreidimensional sind, ist die Parallelisierbarkeit kein Problem, und m_1 bzw. m_2 haben keinen großen Einfluss (siehe Diagramm in Abb. 8.33). Der Parameter m_3 wirkt sich hingegen unterschiedlich aus, da er auch Speicherzugriffsmuster beeinflusst. Da sich die drei Parameter nicht gegenseitig beeinflussen, lassen sie sich unabhängig voneinander auf die Eingabedaten und die verwendete Hardware optimieren.

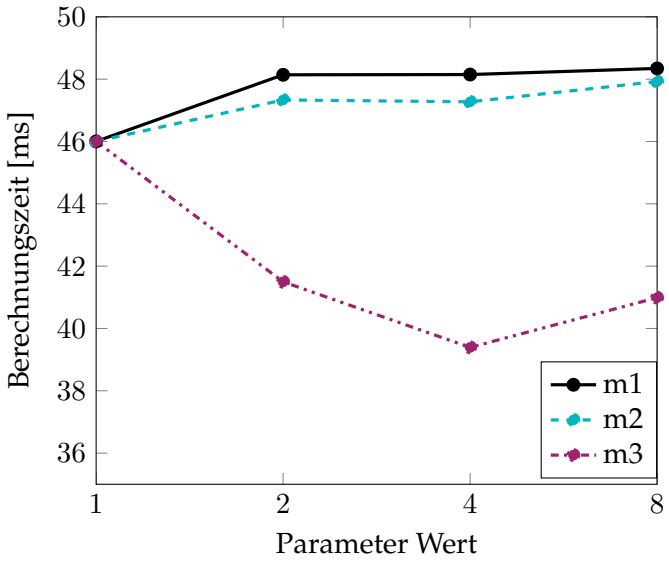


Abb. 8.33.: PBA Laufzeiten für 256^3 Voxel mit unterschiedlichen Parallelisierungsparametern. Geändert wird pro Graph nur ein Parameter, während die beiden anderen statisch auf 1 gesetzt sind. Diagramm aus [24].

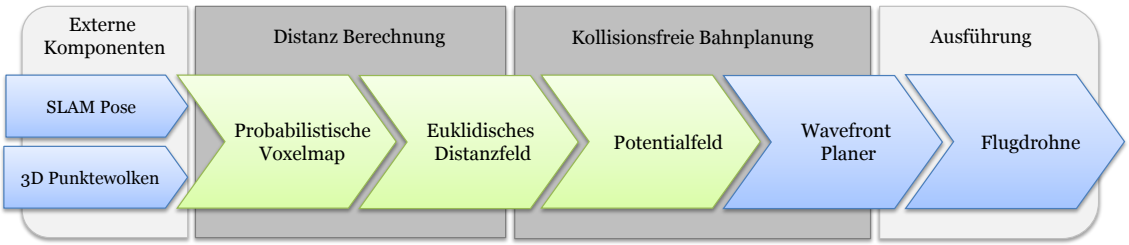


Abb. 8.34.: Systembestandteile der Potentialfeld-Navigation einer Flugdrohne (grün: GPU, blau CPU). Der Fokus dieser Arbeit liegt auf den dunkel hinterlegten Teilen.

Navigation einer Flugdrohne

Nachdem sichergestellt ist, dass der Berechnungsdurchsatz mehr als ausreichend ist, lassen sich nun praktische Anwendungen betrachten. Dieses erste Beispiel nutzt ein Distanzfeld, um die Flugroute einer Drohne durch ein zunächst unbekanntes Szenario zu bestimmen (siehe Abb. 8.35). Das Umweltmodell bestand in den Tests aus einer 21 Mio. Voxel großen probabilistischen Voxelkarte, deren Distanzfeld mit dem Parallel-Banding-Verfahren aus Abschnitt 5.6 im Schnitt in 22 ms berechnet wurde. Als Kollisionsmodell der Drohne wurde eine Kugel verwendet. Die komplette Verarbeitungskette ist in Abb. 8.34 dargestellt. Zwei alternative Planungsansätze wurden umgesetzt und in Tab. 8.23 bewertet: Eine Potentialfeld- und eine Wavefront-basierte Navigation. Beide Algorithmen liefen mit jeder neuen Sensorpunktwolke ab, wobei die Hinderniskarte ständig um alle neuen Messungen erweitert wurde, und somit auch die Distanzkarte jedes Mal neu aufzubauen war. Die eigentliche Planung lief auf dem Host ab, wofür es nötig war, die Distanzkarten von der GPU zu kopieren.

Im ersten Ansatz folgte die Flugroute dem Gradienten eines Potentialfeldes, welches wie in Abschnitt 7.1.3 beschrieben, aufgebaut war. Ein abstoßendes Feld wurde mit Hilfe einer harmonischen Funktion aus dem Distanzfeld abgeleitet, während ein weiteres anziehendes Feld die Drohne in Richtung ihres Zieles führte. Der Einfluss des anziehenden Potentials konnte durch den Planer inkrementell verstärkt werden, falls die Drohne sich nicht weiter Richtung Ziel bewegte. Auch wenn diese Strategie über lokale Minima an Engstellen hinweg führte, war das Ergebnis nicht zufriedenstellend, da das Labyrinth zu viele konkave Sackgassen aufwies, aus denen sich der Algorithmus nicht befreien konnte.

Den zweiten Ansatz bildete eine 3D-Wavefront-Suche, welche direkt auf dem Distanzfeld arbeitete und den global optimalen inversen Pfad vom Ziel zur aktuellen Position der Drohne berechnete. Durch einen Schwellwert der Distanzen konnte sichergestellt werden, dass die Flugroute dabei einen Mindestabstand zu allen Hindernissen aufwies. Naturgemäß stellten lokale Minima und Sackgassen kein Problem dar, jedoch lag die Berechnungsdauer wesentlich höher als bei der Potentialfeldmethode und variierte mit der Länge des Pfades.

Planungsverfahren	**PBA auf Host kopieren [ms]**				**Pfad konstruieren [ms]**				**Erreichbare Pfadlänge**
	Min	**Max**	**Median**	**∅**	**Min**	**Max**	**Median**	**∅**	
Gradientenabstieg	5,79	11,87	8,48	8,43	0,01	2,5	0,48	0,53	23 Felder
Wavefront	22,03	129,63	55,42	65,62	0,02	2,7	0,66	0,70	84 Felder

Tab. 8.23.: Laufzeiten von zwei 3D-Pfadplanungsverfahren für eine Flugdrohne mittels Distanzfeldern.

Die Verwendung einer schnellen Distantanzfeldberechnung ermöglicht ein reaktives Verhalten einer Flugdrohne in einer komplexen dynamischen Umwelt, die erst zur Ausführungszeit erkundet wird. Zwar war es nicht möglich, allein auf Basis eines Gradientenabstiegs zum Ziel zu gelangen, jedoch war auch eine Wavefront-Suche schnell genug, um global konsistente Pläne unter lokaler Hindernisvermeidung mehrmals pro Sekunde zu generieren. Durch eine Portierung des Planers auf die GPU könnte in Zukunft der Aufwand des Datentransfers zwischen GPU und CPU vermieden werden. Die kugelförmige Approximation der Robotergeometrie erlaubt es, die Kollisionsfreiheit mit nur einem Lesezugriff in der Distanzkarte sicherzustellen. Eine spannende Erweiterung des Planungsszenarios wäre es, die Distanzkarte mit dem Skelettierungsprozess aus [128] so weit auszudünnen, bis nur noch ausgewählte Voronoi-Achsen erhalten blieben, die direkt als Navigationsgraph verwendet werden könnten.

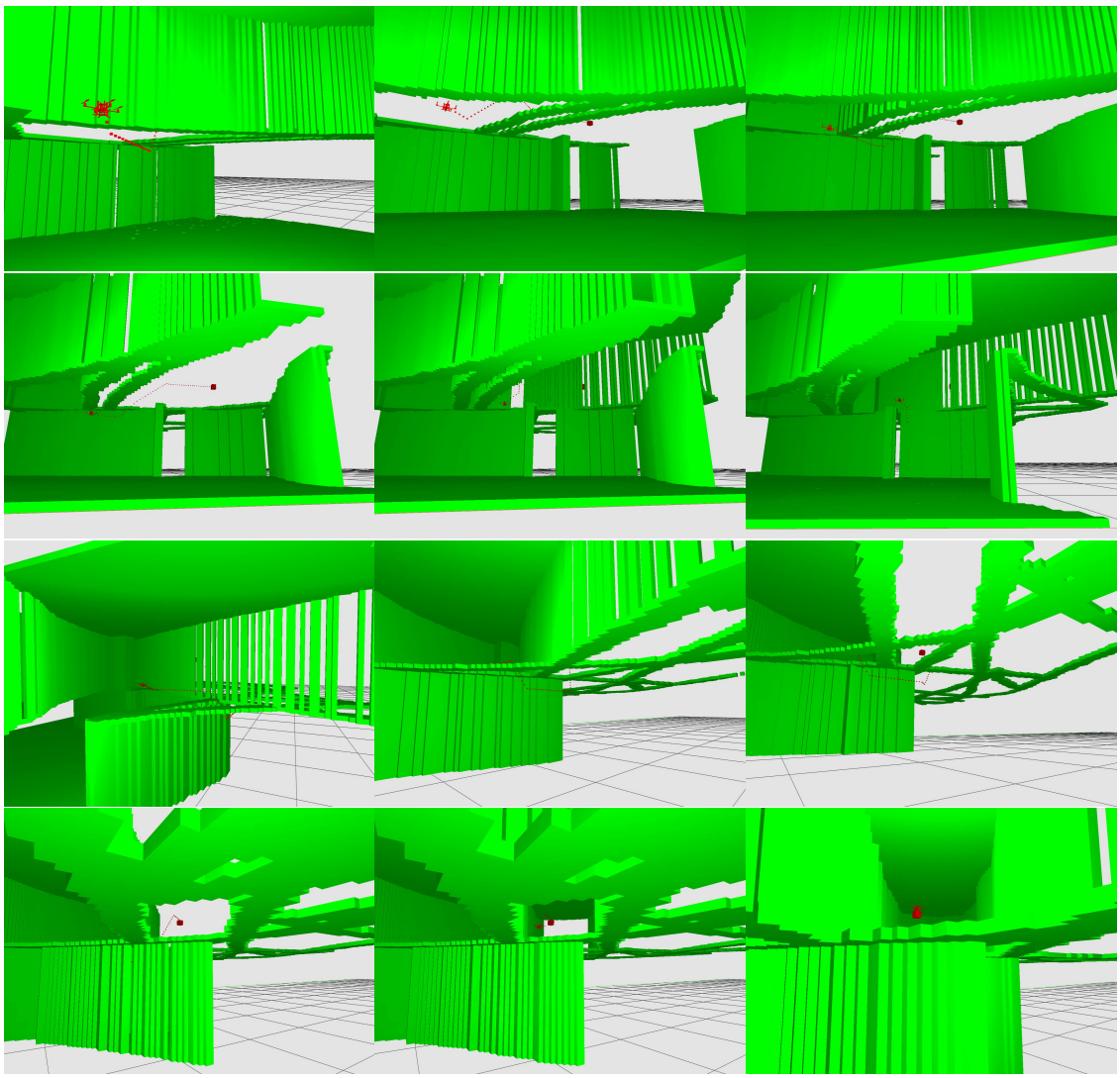


Abb. 8.35.: Momentaufnahmen der Navigation einer Flugdrohne in einem unbekannten, zerklüfteten Szenario mittels Distanzfeldern. Der Pfad wird kontinuierlich mit 15-20 Hz an neue Umweltdaten angepasst.

Distanz optimierende inverse Kinematik

Der zweite Anwendungsfall nutzt ein Distanzfeld, um den verfügbaren Freiraum um einen Roboter zu bestimmen und über eine Variation seiner Konfiguration zu maximieren. Ein Roboter könnte mit diesem Verfahren seinen Aufgaben nachkommen und dennoch einem Menschen aus dem Weg gehen oder die Distanz zu allen Hindernissen maximieren, um seinen Manipulationsspielraum zu vergrößern.

Für die Berechnung wird eine statische Punktwolke der Umgebungsgeometrie mit den Punkten einer Kinect-Kamera kombiniert und das gemeinsame Distanzfeld mittels PBA berechnet. In diesem Distanzfeld lassen sich dann Roboterkonfigurationen bewerten, indem an unterschiedlichen Punkten der kinematischen Kette der verfügbare Freiraum abgefragt und aufsummiert wird. Dies entspricht sinngemäß einer Menge von virtuellen Abstandssensoren, die an den Ecken der mobilen Plattform und an den Gelenken des Roboters angebracht sind. Der aufsummierte Abstandswert geht dann in eine Metrik ein, mit der unterschiedliche IK-Lösungen verglichen werden können. Somit werden kleine Bewegungsinkremente bevorzugt. Die Metrik bildet die Grundlage einer Partikel-Schwarm-Optimierung, deren Partikel randomisierte Startwerte einer iterativen inversen Kinematik darstellen. Als Ziel der inversen Kinematik wurde eine konstante TCP Pose vorgegeben.

Folgende Notation wird dabei verwendet:

- P: Punktwolke aller Hindernisse, zusammengesetzt aus einem statischen Anteil und der dynamischen Kamera-Punktwolke: $P = P_{\text{Statisch}} \cup P_{\text{Kinect}}$
- EDT_P: Distanzfeld der Punktwolke P
- $\vec{\varphi}$: Gelenkwinkel aller N Freiheitsgrade. $\vec{\varphi} \in [\vec{\varphi}_{\text{min}}, \vec{\varphi}_{\text{max}}]$
- $\vec{\varphi}_{t-1}$: Vorherige Roboterpose
- $\text{dk}(\vec{\varphi}, k)$: Vorwärtskinematik. Berechnet die Pose des k-ten Elements der Roboterkinematik anhand der Gelenkwinkel $\vec{\varphi}$
- $\Delta_{\varphi\,\text{max}}$: Maximal erlaubte Gelenkwinkelbewegung pro Gelenk
- $\Delta_{d\,\text{max}}$: Obergrenze für Distanz zu nächstem Hindernis

Folglich sucht man nach einer Roboterpose $\vec{\varphi}_t$, welche die Funktion f unter Berücksichtigung der vorherigen Roboterpose $\vec{\varphi}_{t-1}$ und dem aktuellen Distanzfeld EDT maximiert:

$$\arg\max_{\vec{\varphi}_t} \Big(f\big(\vec{\varphi}_{t-1}, \vec{\varphi}_t, \text{EDT}_P\big) \Big) \tag{8.4}$$

Die Funktion f setzt sich daher aus zwei gewichteten Teilen a und b zusammen, die die Gelenkwinkelbewegung und den Freiraum bewerten:

$$f\big(\vec{\varphi}_{t-1}, \vec{\varphi}_t, \text{EDT}_P\big) = \Big(\alpha \cdot a\big(\vec{\varphi}_{t-1}, \vec{\varphi}_t\big) + \beta \cdot b\big(\vec{\varphi}_t, \text{EDT}_P\big)\Big) \cdot \frac{1}{\alpha + \beta} \tag{8.5}$$

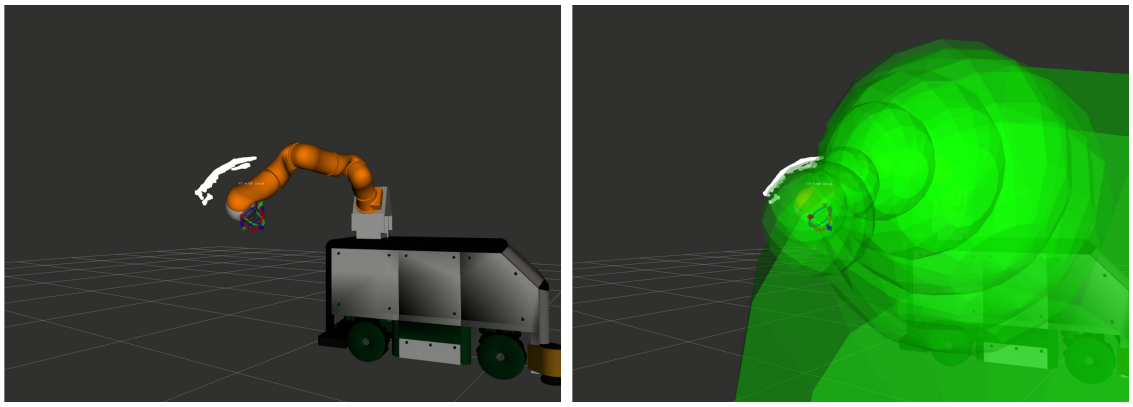
Da die Funktionen a und b auf den Wertebereich [0, 1] normiert sind, bewegt sich auch f in diesem Wertebereich. Die beiden Formeln setzen sich wie folgt zusammen:

$$a\big(\vec{\varphi}_{t-1}, \vec{\varphi}_t\big) = \frac{1}{N} \sum_{j=0}^{N} \Big(1 - \min\big(1, \frac{|\varphi_{j(t-1)} - \varphi_{j(t)}|}{\Delta_{\varphi\,\text{max}}}\big)\Big) \tag{8.6}$$

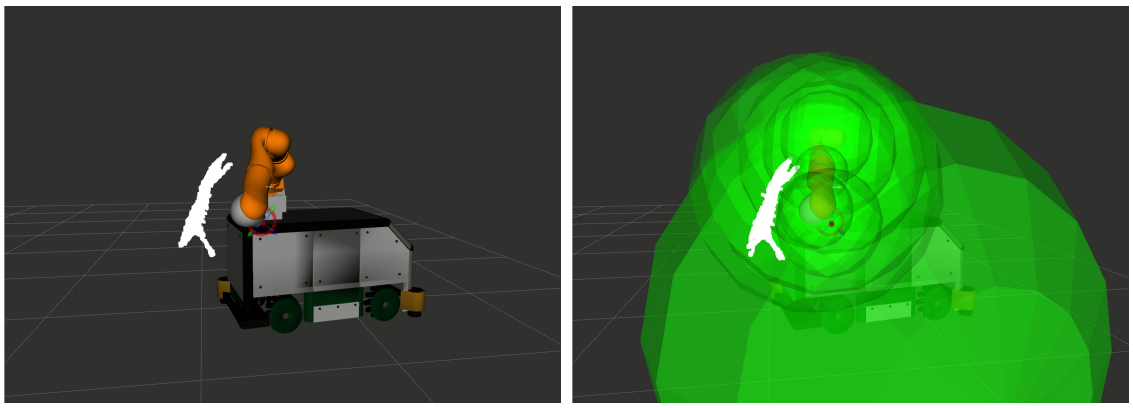
$$b\big(\vec{\varphi}_t, \mathrm{EDT}_P\big) = \frac{1}{N+1} \sum_{k=0}^{N+1} \Big(\min\big(1, \frac{\mathrm{EDT}_P\big(\mathrm{dk}(\vec{\varphi}_t, k)\big)}{\Delta_{d\,\max}}\big)\Big) \tag{8.7}$$

$\mathrm{EDT}_P\big(\mathrm{dk}(\vec{\varphi}_t, k)\big)$ entspricht hierbei der Gleichung 5.22 und beschreibt somit den kleinsten Abstand zwischen Roboterkinematik und Umweltmodell, gegeben einer Roboterstellung.

Das Verfahren wurde mit Hilfe einer Simulation des mobilen Roboters IMMP in Kombination mit Punktwolken einer realen Umgebung erfolgreich getestet. Abb. 8.36 zeigt zwei Momentaufnahmen, in denen der TCP des Armes an einer Pose verbleibt, während alle 10 Freiheitsgrade des Roboters genutzt werden, um den Freiraum zur Punktwolke zu maximieren. Für die Berechnung der inversen Kinematik wurde die besonders schnelle Trac-IK Bibliothek [44] eingesetzt. Im Test mit 10 Partikeln und 5 Iterationen des Schwarms konnte eine Wiederholrate von 15 Hz erreicht werden, was für flüssige Ausweichbewegungen des Roboters ausreichte. Durch die Beschränkung der Gelenkwinkeländerungen pro Schritt mittels $\Delta_{\varphi\,\max}$ werden dabei große Sprünge effektiv verhindert.



(a) Roboterarm weicht nach unten aus



(b) Mobile Plattform weicht nach rechts aus

Abb. 8.36.: Interaktive inverse Kinematik für mobilen Manipulator IMMP: Bei vorgegebener TCP Pose werden die Freiräume der einzelnen Roboterglieder maximiert (visualisiert durch grüne Kugeln). Basis ist ein online berechnetes Distanzfeld der Kinect Punktwolke.

Zusammenfassung

Die umgesetzte EDT stellt eine wertvolle Ergänzung zur voxelbasierten Kollisionsdetektion dar, da sie es einem Roboter erlaubt, Abstände gegenüber Hindernissen zu wahren, auch wenn diese aus Sensordaten gewonnen wurden. Durch die Parallelisierung auf der GPU lassen sich auch Distanzfelder mit einem großen Volumen mit der Bildrate eines 3D-Sensors komplett neu aufbauen und auswerten. Es müssen folglich keine approximierenden oder lokal agierenden Verfahren eingesetzt werden.

In zwei sehr unterschiedlichen Robotikanwendungen konnte die Praxistauglichkeit demonstriert werden: Die Navigation einer Flugdrohne in einer dynamisch explorierten, hoch aufgelösten Umgebungskarte und die reaktive inverse Kinematikberechnung eines mobilen Manipulators, die den Freiraum zu Hindernissen maximiert. Beide Versuche zeigen, dass 3D-Daten durch parallelisierte Algorithmen schritthaltend ausgewertet werden können, um aus ihnen Distanzfelder zu generieren. Darüber hinaus sind auch Anwendungsfälle von Planungsalgorithmen denkbar, die neben einer Kollisionsprüfung auch Distanzen zu Hindernissen verwerten, um ihre Exploration zu steuern.

8.10. Experimente zur Greifplanung

Abschließend soll ein untypischer Anwendungsbereich der voxelbasierten Kollisiosnprüfung betrachtet werden. Wie bereits in Unterabschnitt 7.2.7 gezeigt wurde, benötigt die Greifplanung in der Robotik eine möglichst performante Simulation von Greifhypothesen, um darüber den erfolgversprechendsten Griff zu bestimmen. Dies ist im Stand der Technik bisher nicht direkt auf Modellen möglich, die aus Punktwolken gewonnen wurden, sondern lediglich über abstrakte, a priori bekannte Modelle.

Fortschritte in der Hardwareentwicklung brachten unterschiedliche mechatronische Multi-Finger-Hände hervor, die das Verfahren motivieren. Beispiele sind die Shadow Hand, das DLR Hand Arm System und die in dieser Arbeit verwendete SCHUNK SVH Hand. Diese Hände verfügen über fünf bis hin zu zwanzig aktive und weitere gekoppelte passive Freiheitsgrade, die für einen erfolgreichen Griff koordiniert werden müssen. Die entwickelte Greifplanung lässt sich aber auch für andere, einfachere Endeffektors verwenden, wie z.B. die Dreifingergreifer der Firma Robotiq.

Generell stellt die Berechnung der Gelenkstellungen das Ziel der Greifplanung dar. Bei der verwendeten SCHUNK SVH sind dies 20 DOF, die über neun Motoren bewegt werden. Ihre kinematische Konfiguration erlaubt die Handhabung einer Vielzahl von unterschiedlichen Objekten, wie in [16] beschrieben wurde. Abb. 8.37 zeigt die Hand mit angebrachten Sensoren.

8.10.1. Datenakquise

Das Ziel der umgesetzten Greifplanung ist die Berechnung und Verbesserung von Fingerstellungen, während sich der Greifer des Roboters einem Objekt nähert. Um dabei

(a) Montierter LEAP-Motion Sensor.

(b) Montierter PMD Nano Sensor.

(c) Montierter Intel RealSense Sensor.

Abb. 8.37.: Unterschiedliche Tiefenkameras, die auf der anthropomorphen SCHUNK SVH Hand montiert wurden. Die Anbringung wurde so gewählt, dass der Arbeitsraum der Finger möglichst nicht eingeschränkt wird.

möglichst exakte Messungen zu erhalten, ist es von Vorteil, den Sensor direkt im Greifer zu integrieren. Zwar können mit allen in Abschnitt 4.1 beschriebenen Sensoren kontinuierliche 3D-Datenströme erzeugt werden, wie sie im Fall der Online-Greifplanung benötigt werden. Allerdings sind nur wenige Sensoren klein genug, um im Endeffektor verbaut zu werden. Von den in Abb. 8.37 gezeigten Alternativen wies jedoch lediglich die Intel RealSense Kamera ein ausreichendes Signal/Rausch-Verhältnis auf, um Punktwolken aus mehreren Aufnahmen zu kombinieren. Daher wurde die Datenaufnahme mit diesem Sensor durchgeführt.

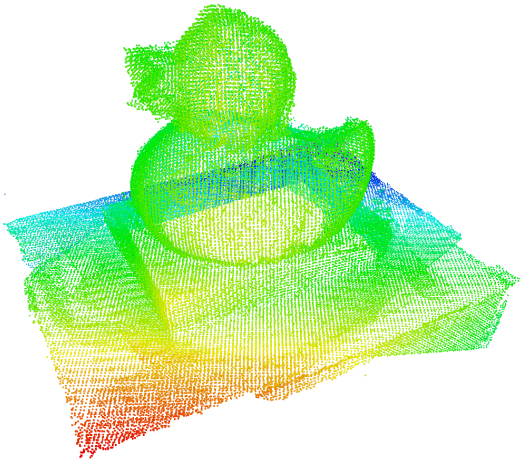


Abb. 8.38.: Zusammengesetzte Punktwolke aus acht Einzelaufnahmen.

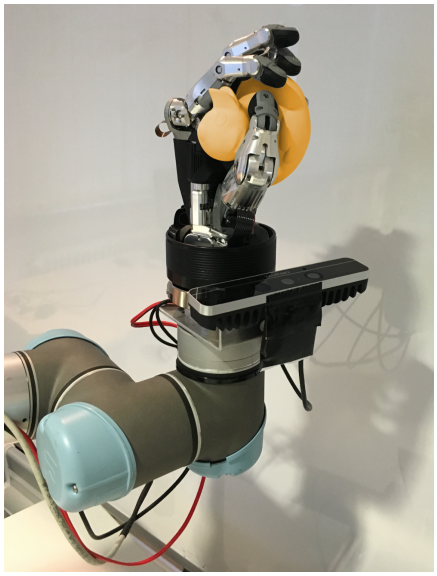


Abb. 8.39.: Ausgeführter Griff mit einer SCHUNK SVH.

8.10.2. Implementierung

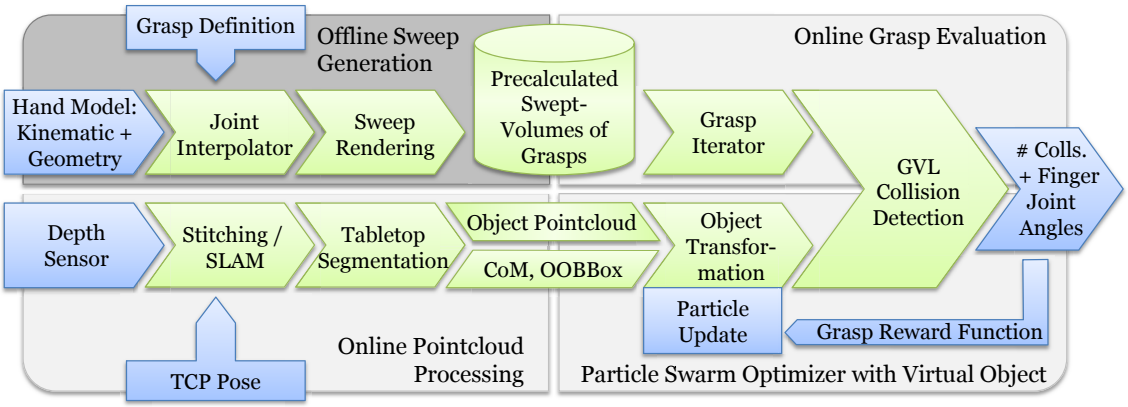


Abb. 8.40.: Datenfluss: Alle grünen Komponenten wurden auf der GPU parallelisiert. Alle blauen Komponenten sind Eingabe- oder Ausgabedaten, die über das Host-System laufen. Das obere linke Viertel muss lediglich zur einmaligen Offlineberechnung der Greif-Swept-Volumen ausgeführt werden.

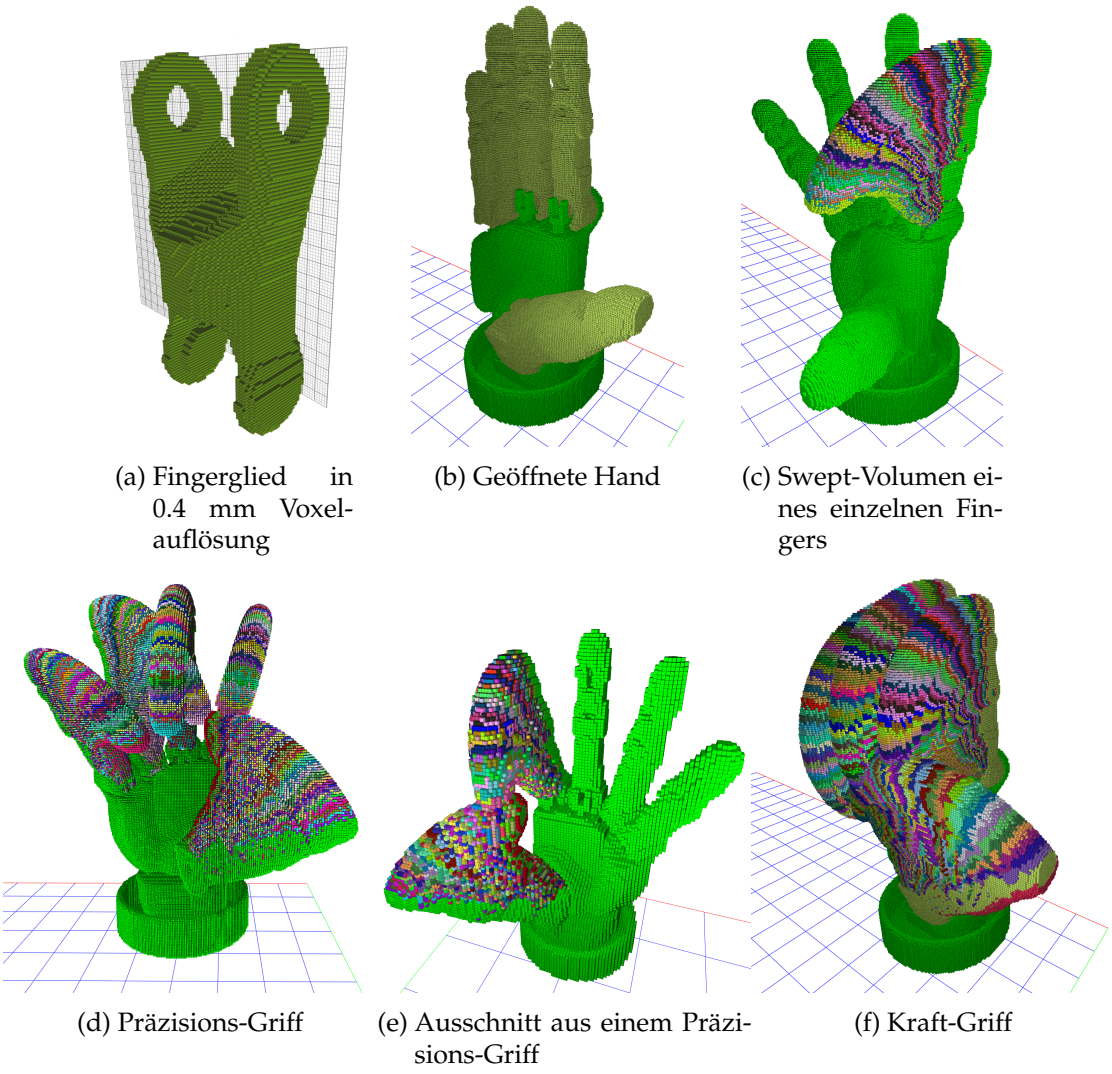
Wie in Abb. 8.40 zu sehen, gliedert sich die Software in einen offline Vorverarbeitungsschritt, in welchem die Swept-Volumen der Griffe generiert werden und zwei online ablaufenden Algorithmen zur Verarbeitung von Sensordaten und zum Evaluieren der Greifoptionen. Diese Komponenten werden im Folgenden näher beschrieben.

Offlineerzeugung von Griff-Swept-Volumen

In einem initialen Schritt müssen zunächst die Swept-Volumen aller ausführbaren Greifbewegungen (Kraft-, Präzisionsgriff, usw.) erzeugt werden. Die Griffe sind dabei definiert über die Start- und Endwinkel der Fingergelenke und über spezifische Kopplungsfaktoren zwischen einzelnen Gelenken. Zur Erzeugung der Volumen werden die Finger nacheinander von ihrem Start- zum Endwinkel bewegt und währenddessen das überstrichene Volumen in einer Voxelliste aus Bitvektor-Voxeln gespeichert. In definierten Winkelabständen werden dabei die SSV-IDs inkrementiert, um identifizierbare Abschnitte der Bewegung zu erhalten. Einzelne Subvolumen eines Griffes sind in Abb. 8.41 farblich abgegrenzt. Die verfügbaren SSV-IDs müssen auf die aktiven Freiheitsgrade der Hand aufgeteilt werden. Weiterhin wird vorausgesetzt, dass sich ein Finger während eines Griffes in einer kontinuierlichen Bewegung schließt, auch wenn er über mehrere aktive Freiheitsgrade verfügt (wie es bei Mittel- und Ringfinger der SCHUNK SVH der Fall ist). In diesem Fall werden die Antriebe per Software mit einem griffspezifischen Faktor aneinander gekoppelt. Von den neun DOF der SVH wird die Spreizung der Finger und das Anlegen des Daumens während einem Griff auf einem griffspezifischen Wert gehalten. Somit blieben $N = 5$ bewegte Freiheitsgrade, auf welche sich die verwendeten $K = 250$ SSV-IDs verteilen. Die Winkelintervalle aller Gelenke $\vec{\delta}_\varphi$ und somit die Auflösung der Sub-Volumen lassen sich über

$$\vec{\delta}_\varphi = \vec{\varphi}_{\text{max}} \cdot \frac{N}{K} \tag{8.8}$$

berechnen. Bei einem typischen Griff reicht der Bewegungsbereich eines Fingers von $0°$ bis zu $\sim 90°$, die in 50 Sub-Volumen δ zu je $\sim 1{,}8°$ pro SSV-ID aufgeteilt werden.



(a) Fingerglied in 0.4 mm Voxelauflösung

(b) Geöffnete Hand

(c) Swept-Volumen eines einzelnen Fingers

(d) Präzisions-Griff

(e) Ausschnitt aus einem Präzisions-Griff

(f) Kraft-Griff

Abb. 8.41.: Volumetrisches Handmodell und vorausberechnete Swept-Volumen unterschiedlicher Griffe. Jede Farbe repräsentiert ein Sub-Volumen, das bei einer Kollisionsprüfung individuell identifizierbar ist.

Sensordatenverarbeitung

Der verwendete Sensor liefert Wolken aus 640×480 3D Punkten mit Bildrate von 30 Hz. Diese Daten müssen zunächst in ein globales Koordinatensystem transformiert werden, wo sie sich dann per ICP, KinFu oder ähnlichen Fusionierungstechniken in ein konsistentes Modell akkumulieren lassen.

Für die hier beschriebenen Experimente wurden die Aufnahmen lediglich durch eine exakte extrinsische Kalibrierung direkt in einem probabilistischen GPU-Octree gesammelt. Auch eine rechenintensive Oberflächenrekonstruktion ist für die Verarbeitung der Voxel nicht nötig.

Für die weiterführende Verarbeitung wird die Punktwolke in das Objekt und die Tischfläche segmentiert. Hierfür kommt eine RANSAC Ebenenschätzung aus der Point Cloud Library (PCL) zum Einsatz. Das freigeschnittene Objekt und eine darum gebildete Object Oriented Bound Box (OOBB) sind der Ausgangspunkt für weitere Berechnungen, bei denen die Tischfläche zur Kollisionsvermeidung berücksichtigt wird.

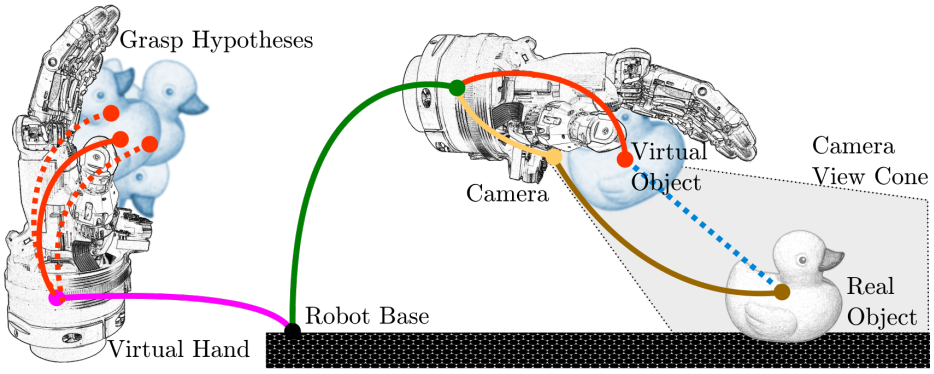
Vermeidung unbekannter Regionen

Eine Herausforderung bei der Greifplanung mit unvollständigem Umweltwissen ist es, zu verhindern, dass die Finger der Hand in nicht eingesehene Regionen bewegt werden, und dort mit unbekannten Hindernisse kollidieren. Dennoch sind auch bei unvollständigem Wissen erfolgreiche Griffe möglich, wenn zwischen *unbekanntem* und *freiem* Raum unterschieden werden kann. Dies ist bei Dreiecksnetzmodellierungen schwer möglich, da hier lediglich die Oberflächen der detektierten Objekte repräsentiert sind, wohingegen es bei einer Volumenmodellierung einfach umzusetzen ist: Wie in Unterabschnitt 4.3.1 beschrieben, können mittels Raycasting die Voxel des freien Raumes markiert und so von unbekanntem Raum unterschieden werden. Dadurch ist es möglich, bei Kollisionsprüfungen unbekannten Raum explizit zu berücksichtigen und Griffe, die darin eindringen, zu verwerfen. Somit können auch Objekte, die lediglich von zwei gegenüberliegenden Seiten aus gesehen wurden, sicher gegriffen werden, während die nicht bekannten Objektseiten vermieden werden.

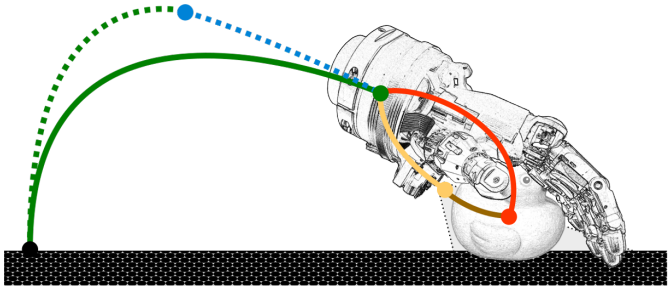
Virtueller Arbeitsraum

Bei der Griffevaluierung ist es effizienter, nicht die offline berechneten Swept-Volumen relativ zum Objekt zu transformieren, sondern das Objekt in einem virtuellen Arbeitsraum relativ zu den Griffvolumen zu bewegen. Daher ist eine Reihe von Koordinatensystemtransformationen nötig, die in Abb. 8.42 nachzuverfolgen sind und die aus folgenden Schritten bestehen:

1. Transformation der Sensorpunktwolken (in Kamerakoordinaten) in das globale Koordinatensystem mit Hilfe der direkten Kinematik des Roboters, um sie zu fusionieren, zu segmentieren und um den Freiraum zu bestimmen.
2. Transformation des Objektes O in den virtuellen Arbeitsraum der vorberechneten Griffe, um darin alle Partikel via Greifsimulation zu bewerten.



(a) Anrückbewegung



(b) Objekt gegriffen

Abb. 8.42.: Unterschiedliche Koordinatensysteme während dem Greifprozess: Grün: Roboterursprung zu Hand. Blau gestrichelt: Transformation von virtuellem zu realem Objekt bzw. Anrückbewegung. Pink: Ursprung des virtuellen Handmodells und seines Swept-Volumens. Rot: Zu optimierende Pose des virtuellen Objekts. Veröffentlicht in [8].

3. Invertierung der Transformation der besten Objektposition, gegeben im Bezugskoordinatensystem der virtuellen Hand.

4. Bestimmung der Transformation zwischen dem realen Objekt und der Greifpose der realen Hand, mittels der vorigen, invertierten besten Objektlage.

5. Planung / Berechnung der inversen Kinematik, um die Hand mittels Roboterarm zur Greifpose zu verfahren.

Hybride Partikelschwarmoptimierung

Wie bereits beschrieben, wird die Greifplanung in dieser Arbeit als komplexes Optimierungsproblem aufgefasst, welches mit einer hybriden PSO gelöst wird. Jeder Partikel $P = (\vec{\Delta}_{xyz}, \vec{\Delta}_{\alpha\beta\gamma})$ stellt dabei eine Translation und Rotation des zu greifenden Objektes relativ zur Hand dar. Während der Optimierung wird in jeder Iteration die Greiffunktion g aus Gleichung 7.6 für jeden Partikel ausgewertet und das Ergebnis mittels der Bewertungsfunktion f aus Gleichung 7.5 beurteilt. Somit lässt sich das Problem in Form einer

PSO darstellen als:

$$\begin{aligned} &\underset{P_{i,j}}{\arg\max} \Big(f\Big(g\big(\mathrm{PSO}_{i,j}, O, H\big)\Big)\Big), \\ &\text{mit: } i \in [1, I],\, j \in [1, J],\, f(g) < \sigma_f \end{aligned} \tag{8.9}$$

wobei I die Anzahl der verwendeten Partikel, J die maximale Zahl an Iterationen und σ_f ein vorgegebenes Abbruchkriterium über die erreichte Greifqualität darstellt.

Bevor die Optimierung gestartet werden kann, müssen die Geometriemodelle des Objektes O und der Hand H zusammen mit den Swept-Volumen des Griffes in den Speicher der GPU geladen werden. Anschließend sind I initiale Partikel zu erzeugen. Da alle Dimensionen des Suchraumes als unkorreliert betrachtet werden, kann eine Gleichverteilung aller Partikelparameter angenommen werden.

Die wichtigste Funktion g wird ausgewertet, indem ein Griff in GPU-Voxels simuliert wird, um damit die Gelenkwinkel φ_n zu bestimmen, unter denen die Finger gerade das Objekt O berühren. Gleichzeitig liefert g auch die Anzahl der in Kollision liegenden Voxel:

$$V_{\text{col}} = |V_O \cap V_{H\,\text{Finger}}| + |V_O \cap V_{H\,\text{Handfläche}}| \tag{8.10}$$

Diese Daten fließen in die Bewertungsfunktion f aus Gleichung 7.5 ein und bestimmen zusammen mit der Historie des Partikels dessen Neuparametrierung. Hierbei wird das Partikel gleichermaßen durch das beste Individuum im Schwarm angezogen aber auch durch das Optimum in seiner eigenen Historie (PSO-Gewichtungsfaktoren $\alpha = \beta = 1.0$, siehe Abschnitt A.6).

Zur Maximierung des Datendurchsatzes bei der Arbeit mit Swept-Volu-men wird auf die Technik der Translation mittels Basisversatz aus Unterabschnitt 5.3.1 zurückgegriffen, um die Voxelkarte des Objektes relativ zur Voxelliste des Griff-Swept-Volumens zu verschieben. Durch dieses Implementierungsdetail können Griffe mit variierender Translation wesentlich schneller geprüft werden, als Griffe mit unterschiedlichen Rotationen. Daher wurde mit Gleichung 8.11 ein hybrider Ansatz verfolgt, der die PSO mit einer erschöpfenden Suche verbindet, die die unterschiedlichen Translationen bei fester Orientierung bewertet.

Somit durchläuft jedes Partikel einen lokalen Optimierer, der die Position $\vec{\Delta}_{xyz}$ innerhalb eines engen Rahmens variiert, und X, Y und Z auf das lokale Optimum setzt, bevor die PSO die Partikel neu verteilt und dabei auch die Orientierung α, β und γ festlegt:

$$P_{i,j} = \underset{P_{i,j}}{\arg\max} \Big(f\Big(g\big(\underbrace{\underset{P_{i,j}}{\arg\max}\, f\left(g\left(P_{xyz}, O, H\right)\right)}_{\text{Lokale Optimierung}} \big), O, H\Big)\Big) \tag{8.11}$$

Die lokale Optimierung ist im nächsten Abschnitt erläutert.

Auswertung der Greiffunktion

Zunächst muss die Art des Griffes festgelegt werden, der zu prüfen ist. Aktuell geschieht dies manuell, könnte aber über eine Objektklasse oder die Größe der OOBB automatisiert geschehen. Alternativ werden unterschiedliche Griffe sequentiell evaluiert und die beste Bewertung gewählt.

Die eigentliche Evaluierung besteht aus zwei Schritten. In der ersten Phase sind einfache Kollisionsprüfungen zwischen der ausgestreckten Hand und dem Objekt nötig: Wie in Algorithmus 10 im Anhang skizziert, wird die Punktwolke des Objektes zunächst entsprechend der durch den Partikel vorgegebenen Pose in die Nähe des Handballens transformiert und dort in Voxel umgewandelt. Danach wird die Y-Koordinate so verschoben, dass das Objekt in der Hand liegt, also mit dieser kollidiert. Ausgehend von diesem Startzustand variieren zwei Schleifen dann die X und Z Koordinaten, während eine dritte innere Schleife die Y Koordinate so weit inkrementiert und das Objekt aus der Hand heraus bewegt, bis keine Kollision mehr vorliegt.

An dieser Stelle beginnt dann die zweite Phase, die aus einer besonderen Kollisionsprüfung zwischen dem vorberechneten Greif-Swept-Volumen und dem Objekt O besteht. Ein CUDA Kernel kann dabei alle Gelenkwinkel aller Finger gleichzeitig in einer einzigen Kollisionsprüfung evaluieren. Das Ergebnis besteht aus einer Liste aller SSV-IDs l, die in Kollision liegen, sobald die Finger das Objekt berühren. Als zusätzliches Ergebnis liegt die zugehörige Anzahl an Kollisionen pro Finger vor. Durch die monoton steigenden SSV-IDs können anhand der Gleichung 8.8 die Gelenkwinkel $\vec{\varphi}$ bestimmt werden, unter denen die Finger das Objekt berühren:

$$\begin{aligned} \varphi_{n\,\text{col}} = \min_{l} \left(\left(l \operatorname{modulo} \frac{N}{K} \right) \cdot \frac{\varphi_{n\,\max} \cdot N}{K} \right), \\ l \in \left[n \cdot \frac{K}{N}, (n+1) \cdot \frac{K}{N} \right) \end{aligned} \tag{8.12}$$

Das Ergebnistupel aus $(\vec{\varphi}, V_{\text{col Handfläche}}, V_{\text{col Finger}})$ wird für jedes Partikel gespeichert, so dass es für die Bewertungsfunktion f zur Verfügung steht.

Evaluierung

In den beschrieben Experimenten wurden 3D-Daten aus einer Simulation verwendet, da der Schwerpunkt dieser Evaluation auf der Leistung der Kollisionsprüfung liegt. Für den Test umkreiste eine virtuelle Kamera das Zielobjekt. Die Punktwolken wurden zunächst in einem Octree mit einer Auflösung von 1 mm in Voxel umgewandelt und segmentiert. Das freigestellte Objekt wurde danach in eine Voxelkarte mit 2 mm übertragen. Zwei Voxellisten mit je 2 mm Auflösung hielten ein Modell der ausgestreckten Hand und das Griff-Swept-Volumen vor.

Um die Leistungsfähigkeit der Greifplanung zu bestimmen, wurden verschiedene Griffe mit wechselnden PSO Parametrierungen durchgeführt. Die Ergebnisse lassen sich in Tab. 8.24 ablesen. Zusätzlich listet Tab. 8.25 die Laufzeiten der beiden Hauptfunktionen und Abb. 8.43 zeigt den Verlauf der Partikelbewertungen während der Optimierung des Griffes für die Gummiente aus Abb. 8.38.

Bei der Ausführung der Greifplanung waren durch die Anytime-Charakteristik des Ansatzes erste Griffhypothesen bereits nach 353 ms verfügbar, wenn mit 5 Partikeln gerechnet wird. Bei 50 Partikeln nach 3,3 s. Die Griffqualität verbessert sich mit der zur Verfügung stehenden Zeit. In verschiedenen Tests zeigte sich, dass bei 10 Partikeln brauchbare Griffe mit einer Bewertung um 200 meist bereits nach ca. 1 s. gefunden wurden.

Anzahl Partikel	Anzahl Iterationen	∅Laufzeit pro Iteration [ms]	Gesamt-laufzeit [sec]	Bewertungs Metrik
5	5		1,8	231
5	10	353,66	2,5	235
5	20		5,6	278
10	5		3,9	205
10	10	715,33	4,5	256
10	20		8,6	282
20	5		6,8	335
20	10	1441,66	9,8	399
20	20		23,1	397
50	5		15,6	387
50	10	3375,00	29,6	405
50	20		49,2	502

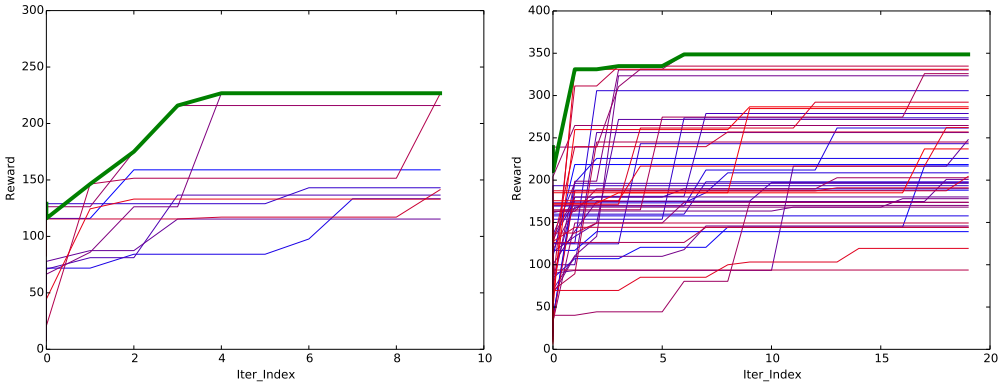
Tab. 8.24.: Laufzeiten mehrerer PSO-Iterationen während der Optimierung eines Griffes mit unterschiedlich vielen Partikeln. Das umgesetzte Verfahren liefert bereits nach der ersten Iteration einen ausführbaren Griff und fällt somit in die Klasse der Anytime-Algorithmen. Die gelisteten Belohnungsmetriken sind für abgeschlossene Durchläufe angegeben.

Funktion	Untersuchte Voxel	Laufzeit [ms]			
		∅	Median	Min	Max
Kollisionsprüfung	$8 * 10^6 \times 61499$	0,103	0,072	0,062	16,357
Griff-Bewertung	$8 * 10^6 \times 70622$	0,507	0,473	0,445	2,508

Tab. 8.25.: Laufzeiten der beiden wichtigsten Funktionen der Greifplanung, gemittelt über 50000 Durchläufe: Kollisionsberechnung zwischen einem Objekt und a) dem statischen Handmodell mit ausgestreckten Fingern. b) dem Swept-Volumen einer Greifbewegung.

8.10.3. Zusammenfassung

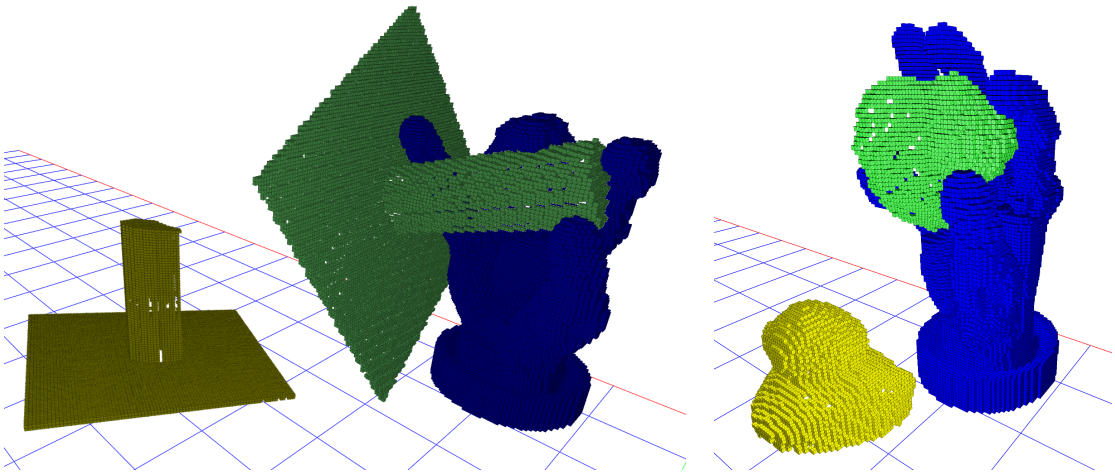
In diesem Abschnitt wurde ein vollständiger Ansatz zur Greifplanung bewertet, der sich die Vorteile der raumpartitionierenden Datenstrukturen in mehreren Hinsichten zu Nutzen macht: Die explizite Unterscheidung zwischen unbekanntem und freiem Volumen



(a) Beispiel der Bewertungsfunktion von 10 Partikeln über 10 Iterationen in einer Gesamtlaufzeit von 4,5 s.

(b) Beispiel der Bewertungsfunktion von 50 Partikeln über 20 Iterationen in einer Gesamtlaufzeit von 49,2 s.

Abb. 8.43.: Bewertungsfunktion aller Partikel aufgetragen über die Iterationen. Die grüne Linie gibt das globale Maximum an. Gute Griffe wurden bereits nach vier bis fünf Iterationen gefunden.



(a) Präzisionsgriff eines zylindrischen Objektes unter Berücksichtigung der Bodenebene

(b) Kraftgriff einer Gummiente

Abb. 8.44.: Resultierende Griffe für zwei unterschiedliche Objekte nach einer Optimierung von $\sim$ 2 Sekunden mit zehn Partikeln.

im Arbeitsraum erlaubt zum einen eine Greifplanung auch bei unvollständigem Umweltwissen. Gleichzeitig arbeiten die Algorithmen, im Gegensatz zu oberflächenbasierten Verfahren, direkt auf 3D-Sensordaten, was es erlaubt, Griffe für vorher unbekannte Objekte zu finden und zu optimieren. Durch die konsequente Parallelisierung ist die Verarbeitungskette onlinefähig und Griffe können während der Annäherung des Greifers an ein Objekt kontinuierlich optimiert werden. Das Verfahren lässt sich zusätzlich mittels taktiler Sensorik erweitern, um Griffe weiter zu stabilisieren oder verformbare Objekte sicher zu greifen.

8.11. Fazit

Die zahlreichen und sehr heterogenen Experimente dieses Kapitels legen die Praxistauglichkeit der GPU-Voxels Bibliothek dar. Sowohl in virtuellen aber auch in realen Testszenarien konnten überzeugende Ergebnisse erreicht werden, die die Fragen aus Forschungsfrage 5 positiv beantworten können: Die entwickelten hochparallelen Verfahren sind schnell genug, um eine Planung von Roboterbewegungen in dynamischen Umgebungen zu ermöglichen, die durch detaillierte Sensordaten repräsentiert werden. Dies gilt sowohl für die Planung mittels Bewegungsprimitiven als auch für Verwendung von samplingbasierten Verfahren. Weiterhin sind die entwickelten Algorithmen so generisch, dass sich umfangreiche Nutzungsmöglichkeiten ergeben, die über die Trajektorienplanung und -überwachung hinaus gehen, wie das Beispiel der schritthaltenden Greifplanung zeigt. Den Laufzeitvergleich mit oberflächenbasierten Kollisionsprüfungsverfahren gewinnt der entwickelte Voxelansatz durch den Einsatz von Swept-Volu-men. Aber auch im direkten Vergleich mit alternativen Punktwolkenverfahren erreicht GPU-Voxels den höchsten Durchsatz von erkannten Kollisionen pro Zeit.

9. Zusammenfassung und Ausblick

9.1. Zusammenfassung und Beitrag

Eines der großen Ziele der Robotik ist die Einsetzbarkeit von flexiblen Maschinen in unstrukturierten Umgebungen des alltäglichen Lebens. Auch der Ausgangspunkt dieser Arbeit liegt in der Verbesserung der Fähigkeit eines Roboters mit beschränktem Wissen über seine Umgebung und mit dynamischen Änderungen dieser Umgebung umgehen zu können. Dies soll einerseits zu sicheren und effizienteren geteilten Arbeitsräumen führen und zum anderen eine schnelle Bewegungsplanung ermöglichen. Der technologische Kern der vorliegenden Arbeit besteht in der Verknüpfung von **Volumenrepräsentationen**, die bereits seit mehreren Jahrzehnten untersucht werden, mit modernen Methoden der **heterogenen Parallelverarbeitung** auf CPUs und GPUs. Hierfür mussten zahlreiche Teilaspekte bearbeitet werden: Zunächst galt es, Datenstrukturen zu definieren, die einerseits Sensordaten der Umgebung, sowie die Repräsentation des Roboters und seiner Bewegungen aufnehmen können und die andererseits einen sehr effizienten, parallelen Zugriff erlauben. Aufbauend darauf mussten Algorithmen implementiert werden, die parallelisiert die Schnittmenge zweier Datenstrukturen ermitteln können, um eine Kollisionsdetektion zu ermöglichen. Mit Hilfe dieser Operationen konnten dann spezialisierte Planungsverfahren entwickelt werden, die die Vorteile der Datenstrukturen ausnutzen, und eine hoch performante Erzeugung von kollisionsfreien Trajektorien möglich machen. Dabei ist es jedoch ineffizient, einzelne Roboterposen zu betrachten. Um mit diesem Problem umzugehen, wurden an vielen Stellen Swept-Volumen verwendet, die komplette Bewegungsabläufe repräsentieren können, ohne dabei einen Zusatzaufwand für ihre Erzeugung zu erfordern, wie es bei etablierten Verfahren mit Oberfächenmodellen der Fall ist.

Die drei wichtigsten Beiträge, die diese Arbeit dabei für die Robotik erbringt, lassen sich wie folgt zusammenfassen:

1. Zunächst arbeiten alle vorgestellten Verfahren direkt auf Punktwolkendaten, wie sie von 3D-Sensoren erzeugt werden. Somit ist keine zeitaufwendige Konvertierung in Oberflächenmodelle oder sonstige abstraktere Repräsentationen nötig. Dies vereinfacht alle Verarbeitungsketten und macht die Nutzung unterschiedlicher Modelle für unterschiedliche Teilaufgaben (wie sie in der Robotik häufig anzutreffen sind) obsolet.

2. Annotierte, voxelbasierte Darstellungen ermöglichen die effiziente Erzeugung und Handhabung von Swept-Volumen, also der echten volumetrischen Modellierung von Bewegungsabläufen, ohne dabei Informationen über Zeit oder Zugehörigkeit aufgeben zu müssen. Im Gegensatz zu Oberflächenmodellen können damit die

selben generischen Daten sowohl für die Planung, als auch für die Ausführungsüberwachung herangezogen werden, was eine feingranulare Arbeitsraumanalyse ermöglicht.

3. Die entwickelten Datenstrukturen und Algorithmen übertreffen den Stand der Technik in Hinsicht auf ihren Datendurchsatz (insbesondere der Octree). Gleichzeitig ist sie so generisch gehalten, dass sich vielfältige Nutzungmöglichkeiten auch außerhalb der bearbeiteten Fragestellungen ergeben. Auch wenn die Implementierung in CUDA umgesetzt wurde, lassen sich die präsentierten Lösungsansätze auch auf andere Plattformen zu Parallelverarbeitung übertragen.

Technisch konnte der Einsatz von General Purpose **GPUs für die Robotik** vorangebracht werden, indem dynamische Datenstrukturen, die den Programmierparadigmen von Parallelprozessoren zunächst widersprechen, hoch performant für die GPU implementiert wurden. Ein sehr praktischer Beitrag ist dabei die Zusammenstellung und Veröffentlichung aller Softwarekomponenten in Form einer **Open-Source Bibliothek** (www.gpu-voxels.org), die in das sehr stark verbreitete *Robot Operation System* (ROS) integrierbar ist.

Durch die Arbeit ist es möglich, hoch auflösende Punktwolken aus mehreren 3D-Kameras mit 30 FPS in einen Octree einzufügen und den Freiraum darin mittels Raycasting zu berechnen. Gleichzeitig können mit derselben Framerate die Kollisionen gegen bis zu 256 Trajektorien oder Roboterposen parallel berechnet werden, was einem **Verarbeitungsdurchsatz von $\sim$ 50 GB/s an Voxeldaten** entspricht. Basierend auf diesen Ergebnissen konnten mehrere **reaktive Planungsverfahren** erfolgreich demonstriert werden. Weitere Werkzeuge, wie die Berechnung von Distanzfeldern oder die Prädiktion der Bewegung von nichtrigiden Körpern, erlauben eine vorausschauende Planung. Diese lässt sich verwenden, um geteilte Mensch-Roboter-Arbeitsräume effizienter auszunutzen.

Eine umfangreiche Evaluation anhand unterschiedlichster Problemstellungen, die verschiedenste geometrische Größenordnungen abdecken, konnte die Skalierbarkeit der Ansätze unter Beweis stellen.

Mit der Beantwortung der eingangs gestellten Forschungsfragen (siehe Abschnitt 1.3) zeigt diese Arbeit somit, dass Voxel eine echte Alternative zu etablierten Oberflächenmodellen darstellen. Sie sind diesen sogar weit überlegen, wenn es gilt, auf Punktwolken aus Sensordaten kollisionsfreie Trajektorien zu planen oder ihre Ausführung feingranular zu überwachen.

9.2. Diskussion und offene Probleme

Ein wichtiges Ziel der Arbeit war es, eine hohe Generalisierbarkeit durch möglichst modellfreie Ansätze sicherzustellen. Da dies erreicht wurde, kommen alle vorgestellten Algorithmen ohne abstrakte geometrische und semantische Informationen aus und es müssen somit keine Annahmen über die zu erwartenden Hindernisse oder Umgebungsbeschaffenheiten getroffen werden. Dies stellt jedoch auch einen der größten Diskussionspunkte dar. So kann in Frage gestellt werden, ob dieses Vorgehen immer zielführend ist. Erwiesenermaßen liefern etablierte, modellbehaftete Ansätze teilweise wesentlich performantere oder qualitativ hochwertigere Ergebnisse. Jedoch stets nur unter Annahme

spezifischer Randbedingungen. Entsprechend lässt sich diese prinzipielle Frage auch nur anwendungsspezifisch beantworten, weshalb hier keine finale Aussage getroffen werden soll.

Zu den offenen Punkten, die weitere spannende Fragestellungen aufwerfen, zählt sicherlich die Bewegungsplanung mittels Bewegungsprimitiven für serielle Kinematiken. Hier konnte keine befriedigende Lösung gefunden werden, konkatenierbare Swept-Volumen vorzuberechnen, um dann effizient mit ihnen zu planen. Eventuell kann hierfür auf eine Modellierung in Polar- oder Kugelkoordinaten oder einem nichteuklidischen Raum zurückgegriffen werden, was jedoch viele der vorgestellten Parallelisierungstechniken verteiteln würde. Die Erforschung wird daher zukünftigen Arbeiten überlassen.

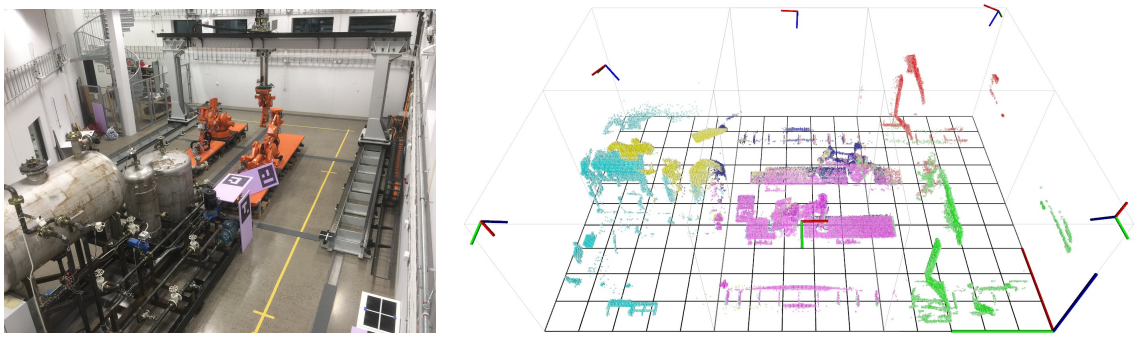
Zwei weitere höchst relevante Fragen ergeben sich, wenn die Kollisionserkennung als sicherheitskritisches Überwachungssystem eingesetzt werden soll: Wie sieht eine zertifiziert sichere Sensortechnik zur Erzeugung von 3D Punktwolken aus? Sensoren dieser Klasse wurden zwar bereits 2016 angekündigt, sind jedoch auch 2018 noch immer nicht am Markt verfügbar. Sind weiterhin die entwickelten GPU Algorithmen formal beweisbar korrekt und nachweislich echtzeitfähig? Die Beantwortung dieser Frage erscheint für die Kollisionsprüfung zwischen zwei Voxelkarten noch durchaus machbar, ist dagegen für den lastbalancierten Octree eine große Herausforderung. Eventuell ließe sich der Aufwand aber durch die Verwendung redundanter Kameras und GPUs vermeiden, so dass auch eine Zertifizierung in Zukunft möglich werden könnte.

Eine letzte, eher oberflächliche Problematik, die sich aus der Planung von Plattformtrajektorien mit Hilfe von Bewegungsprimitiven ableitet, ist die Schwierigkeit, die entwickelten Verfahren in weit verbreitete Planungsbibliotheken wie MoveIt zu integrieren. Da dort keine passenden APIs für die Arbeit mit Primitiven zur Verfügung stehen, liegt die Hürde für die Benutzung durch Dritte etwas höher, als bei samplingbasierten Verfahren. Die enorm überwiegenden Vorteile bei der Planung auf Sensordaten mit GPU-Voxels rechtfertigen jedoch den leicht höheren Aufwand der Integration, den die Bibliothek erfordert.

9.3. Ausblick

Die praktische Relevanz der vorliegenden Arbeit ist allein durch die vielfältigen Anwendungen in der Robotik gegeben. Das allgemeine Interesse an GPU-basierten Technologien steigt zusammen mit der Leistungsfähigkeit der Hardware kontinuierlich. Daher werden in Zukunft immer mehr Roboter auch über Parallelprozessoren verfügen, womit die Voraussetzungen für die Verwendung von GPU-Voxels auch ohne externe Datenverarbeitung erfüllt sind. Ein internationales Interesse mehrerer Robotikforscher und Firmen an der entwickelten Software lässt vermuten, dass die Bibliothek in naher Zukunft in zahlreichen Projekten produktiv eingesetzt wird. Es existieren bereits konkrete Pläne und Partnerschaften, die auch interessante Erweiterungen für neue Anwendungsfälle abdecken: Unter anderem interessiert sich das Unternehmen *Hivemapper* (Silicon Valley) für Voxel, um damit ihre 3D Kartierung für Flugdrohnen zu erweitern und eine Änderungshistorie der Umwelt erzeugen zu können. Weit vorangeschritten sind die Implementierungen des *SFI Centers für Offshore Mechatronik* an University of Agder (Norwegen). Hier

setzen Professor Geir Hovland und sein Team GPU-Voxels ein, um die Roboter auf zukünftigen autonomen Ölbohrplattformen abzusichern (siehe Abb. 9.1). Eine andere Richtung schlägt eine Gruppe am *Autonomous Robotic Manipulation (ARM) Lab* bei Professor Dmitry Berenson der University of Michigan ein. Dort helfen probabilistische Voxel bei der haptischen Exploration und Kartierung eines Roboter-Arbeitsraumes. Weitere Planungsthemen (Roadmap Motion Planning für Echtzeitanwendungen wie Drohnenflug) werden von Dr. Hofmann an den *Dynamic Object Language Labs* (Massachusetts) untersucht. Aber auch aus dem asiatischen Raum stammen zahlreiche Anfragen zu großen Software-Projekten.



(a) Offshore Anlage mit 3 Industrierobotern im Laboraufbau

(b) Kombinierte Voxel-Darstellung der Daten aus 6 Kinect2 Sensoren

Abb. 9.1.: Szenario am SFI in Norwegen zur Untersuchung von GPU-Voxels in Kombination mit unterschiedlicher 3D Sensorik für die Offshore-Anwendung.

Abgesehen von Neuigkeiten in der Software entwickeln große Hardwareanbieter mehr und mehr Embedded GPU Hardware, wodurch höhere 3D-Leistung bei niedrigem Energieverbrauch auch in kleinen Geräten zur Verfügung steht. Insbesondere der von CPU und GPU gemeinsam genutzte Speicher dieser Geräteklasse macht sie für neue, intelligente 3D-Kameras attraktiv. Da hierdurch kein Aufwand für den Datentransfer zwischen Host und Device anfällt, reduziert sich die Latenz bei der Echtzeitverarbeitung enorm. Denkbar wären dadurch smarte integrierte Sensoren, die durch die Berücksichtigung eines adaptiven Egomodells den Nutzer direkt vor Kollisionen warnen können. Anwendungen finden sich bspw. in Assistenzsysteme für Gabelstapler oder Baumaschinen, die eine flexible Geometrie aufweisen und eine 360°-Überwachung erfordern.

Letztendlich ergibt sich durch das sehr allgemeingültige Konzept der Voxel aber auch eine Anwendbarkeit außerhalb der Robotik. Beispielsweise werden in der Medizin bereits seit der Erfindung der Computertomographie Voxel als Datenformat genutzt. Diese Daten könnten für eine OP Überwachung oder Planung verwendet werden, indem Trajektorien von Instrumenten mit den Echtzeitdaten wichtiger Gefäße auf Kollision geprüft werden. Aber auch die Spieleindustrie setzt vermehrt auf Voxelumgebungen, da sich diese durch den Spieler realistischer interaktiv verändern oder zerstören lassen, als dies bei Dreiecksnetzmodellen der Fall ist.

Ebenso ließe sich die Kollisionserkennung gut mit dem großen Forschungsfeld *Deep Learning* verknüpfen: Erfolgt die Prädiktion von Hindernissen nicht wie beschrieben, durch simple, physikalische Modelle, sondern durch eingelernte Bewegungsmodelle, so ließen

sich die Ausgabeneuronen direkt an die probabilistischen Voxel einer GPU-Datenstruktur anschließen. In der Folge könnte eine Ausführungsüberwachung durch neuronale Prädiktion enorm verbessert werden, falls diese situationsspezifisch reagiert.

Anhang

A. Appendix

Abb. A.1.: Logo der GPU-Voxels Bibliothek, das die Würfelstruktur andeutet.

Hier werden allgemeingültige Grundlagen und Definitionen aufgelistet, die in der vorliegende Arbeit verwendet werden.

A.1. Log-Odd

Um Rundungsfehler bei der Multiplikation von kleinen Wahrscheinlichkeiten zu vermeiden, können diese in Form ihres Log-Odds ausgedrückt und somit Addiert werden:

$$l(x) = log\frac{p(x)}{1-p(x)} \tag{A.1}$$

Um aus der Log-Odd Darstellung wieder die Wahrscheinlichkeit zu berechnen, bedient man sich folglich:

$$p(x) = 1 - \frac{1}{1+10^{l(x)}} \tag{A.2}$$

A.2. CUDA Intrinsics

Die CUDA-API stellt dem Programmierer viele Befehle zur Verfügung, die komplexe Funktionen hardwarenah umsetzen und wesentlich effizienter ablaufen, als eine manuelle Implementierung. Hier sollen die für diese Arbeit wichtigsten API-Befehle gelistet und kurz erklärt werden. Dazu zählen in erster Linie Funktionen, die innerhalb eines Warps Ergebnisse zusammenführen, aber auch mathematische Funktionen oder Synchronisations-Barrieren für ganze Threadblöcke. Detaillierte Informationen finden sich im CUDA Programming Guide [154].

Warp Vote Funktionen Diese Gruppe von Funktionen trägt Ergebnisse aller Threads eines Warps zusammen und stellt diesen ein kombiniertes Ergebnis zur Verfügung.

unsigned int __all(int predicate)
Wird in jedem Thread zu einem Wert ungleich 0 ausgewertet, wenn die Eingabedaten aller Threads ungleich 0 waren.

unsigned int __any(int predicate)
Wird in jedem Thread zu einem Wert ungleich 0 ausgewertet, wenn die Eingabedaten mindestens eines Threads ungleich 0 waren.

unsigned int __ballot(int predicate)
Stellt jedem Thread ein Bitmuster aus 32 Bits zur Verfügung, in dem jedes Bit einen Thread repräsentiert, und auf `true` steht, wenn die Eingabedaten dieses Threads ungleich 0 waren.

Synchronisation Um alle Threads eines Blocks zu koordinieren, können diese über eine Barriere synchronisiert werden:

void __syncthreads()
Blockiert die Ausführung, bis alle Threads des Blocks diesen Befehlt erreicht haben. Gleichzeitig wird Speicherkonsistenz im globalen und geteilten Speicher sichergestellt.

Mathematik Viele Bit-basierten Zählfunktionen sind direkt als Spezialbefehle verfügbar. In Kombination mit den Warp Vote Funktionen können mit ihnen beispielsweise Prefixsummen effizient umgesetzt werden.

__device__ int __popc(unsigned int x)
Zählt die gesetzten Bits in einem Bitmuster aus 32 Bits.

__device__ int __clzll (long long int x)
Zählt die führenden, aufeinander folgenden, nicht gesetzten Bits in einem 64 Bit Datum, beginnend bei Bit 63.

A.3. Morton-Codes

Morton-Codes stellen eine Linearisierung einer n-Dimensionalen Adressierung dar und wurden bereits 1966 vorgestellt [148]. Da ihr Verlauf im zweidimensionalen Z-förmig ist, spricht man auch von Z-Kurve. Z-Kurven sind raumfüllende Kurven (FASS-Kurven), decken also rekursiv den kompletten Raum ab, den sie beschrieben. Ihre Berechnung erfolgt durch einfache Bit-Verschachtelung, wie im Beispiel aus Abb. A.2 gezeigt. Die Binärwerte der einzelnen Koordinaten werden mit je zwei 0-Bits gespreizt und die entstehenden Worte verodert. Um aus einem Morton-Code die ursprünglichen Koordinaten zu erhalten, müssen die Operationen invers angewendet werden. Alle Schritte können durch vorberechnete Nachschlagetabellen stark beschleunigt werden[1].

[1] `http://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/`

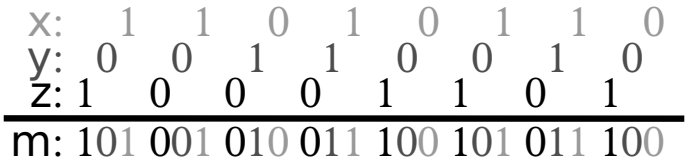


Abb. A.2.: Beispiel zur Erzeugung des Morton-Codes $m = 101001010011100101011100_b$ einer dreidimensionalen Koordinate $P = (x, y, z) = (214, 50, 141)_{10} = (11010110, 00110010, 10001101)_b$. Die Koordinate fällt also in die $m = 10828124_{10}$. Zelle entlang der Z-Kurve. Beispiel aus [22].

Der rekursive Verlauf der Kurve entspricht dem Ablaufen eines 2^n-Baumes mittels post-order Tiefensuche, weshalb jedes n-Tupel aus Bits beim Abstieg im Baum den zu verfolgenden Kindknoten beschreibt. Zwei Beispiele anhand des Binärbaumes sind in Abb. A.4 gegeben.

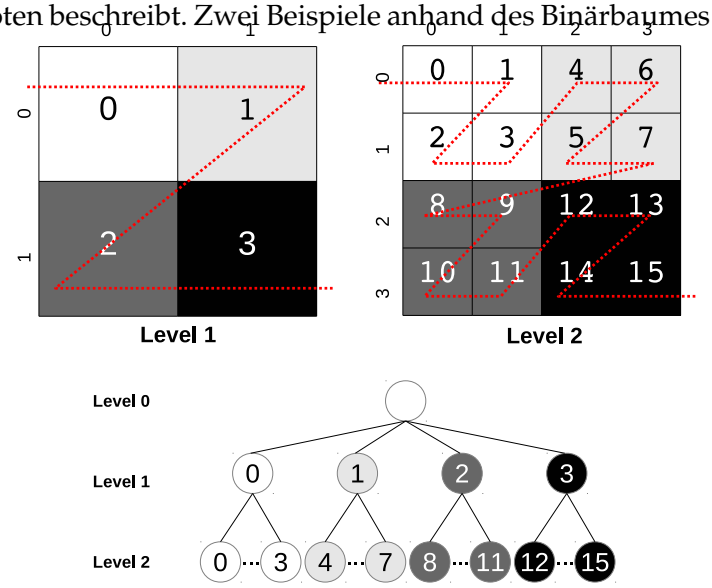


Abb. A.3.: Verwendung von Morton Codes in einem Binär-Baum. Die Grafik aus [40] verwendet eine zu dieser Arbeit inverse Numerierung der Baumebenen.

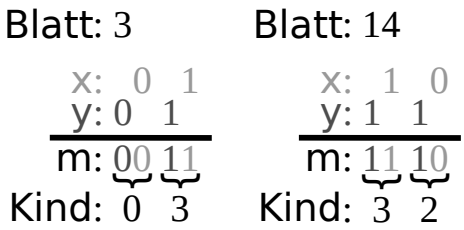


Abb. A.4.: Beispiele zum Abstieg im Binärbaum aus Abb. A.3 zu Blattknoten 3 und 14 anhand der Morton-Code-Tupel.

Gemeinsamer Elternknoten

Die Eigenschaft des Kurvenverlaufs erlaubt es auch, den kleinsten gemeinsamen Elternknoten p zweier Blätter m_a und m_b mit Hilfe einfacher Bit-Operationen zu bestimmen. Hierfür ist zunächst dessen Ebene l_p im Baum zu finden:

$$l_p = \lceil (64 - \texttt{\_\_clzll}(m_a \; XOR \; m_b))/3) \rceil \tag{A.3}$$

Die Funktion `__clzll()` zählt in CUDA die führenden Nullen in einer 64 Bit-Zahl.

Der Pfad ist dann der Präfix der Länge $3 \cdot (d-1-l_p)$ von m_a oder m_b, wobei d die Tiefe des Octrees (Anzahl Ebenen) angibt. Dies ist insbesondere Hilfreich, wenn mehrere Traversierungen in dieselben Baumregionen durchgeführt werden müssen. Dabei lassen sich dann gemeinsame Abschnitte schnell ermitteln und somit die Arbeit mehrerer Threads zusammenfassen.

Sukzessive Berechnung

Die zugrunde liegenden Eigenschaften des Morton-Codes erlauben die sukzessive Berechnung aus dem Code des Elternknotens ($morton_{l+1}$) und der Nummer des zu adressierenden Kindknotens ($child_{l+1}$) mit folgender effizient umsetzbarer Formel:

$$morton_l := (morton_{l+1} \ll 3) \mid child_{l+1} \tag{A.4}$$

Dabei gilt für den Wurzelknoten $morton_{l=0} := 0$ sowie für nicht vorhandene Kindknoten $morton_{l+1} := 0$.

Volumen eines Teilbaumes

Jeder Knoten eines Octrees, der kein Blattknoten ist, repräsentiert einen Teilbaum der Größe $k \times k \times k$ mit $k := 2^l$, wobei l die Ebene ist, auf der sich der Knoten befindet. Das Volumen eines solchen Würfels lässt sich durch die kartesischen Koordinaten seines minimalen (min) und maximalen (max) Eckpunktes beschreiben. Diese wiederum berechnen sich direkt aus der Kind-ID $i \mid 0 \leq i < 8$ und der minimalen Koordinate c des Elternknotens:

$$\text{calcCoords}\,(c, l, i) := \begin{pmatrix} min \\ max \end{pmatrix} = \begin{pmatrix} c + 2^l \cdot \text{morton}^{-1}\,(i) \\ c + 2^l \cdot \left(\text{morton}^{-1}\,(i) + \mathbb{1}\right) \end{pmatrix} \tag{A.5}$$

Die Funktion $\text{morton}^{-1}(i)$ erzeugt aus dem Morton Code i die Koordinaten des Kind-Würfels, also einen Vektor $(x, y, z)^T \mid x, y, z \leq 1$. Weiterhin bezeichnet $\mathbb{1}$ den Vektor $(1, 1, 1)^T$.

Das folgende Beispiel zur Berechnung der min und max Koordinaten bezieht sich auf einen Baum mit vier Ebenen, wie in an Abb. A.5 dargestellt. Gesucht werden der dritte Mortonvoxel auf Baumebene eins, dessen Elternvoxel auf Ebene zwei seinen minimalen Eckpunkt bei $(4, 0, 4)^T$ hat.

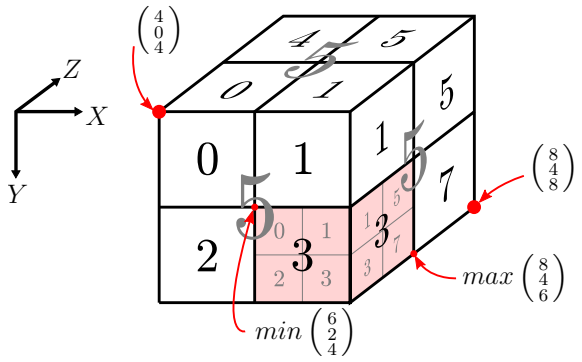


Abb. A.5.: Subvolumens des dritten Mortonvoxels (rot gefärbt) auf Ebene 1 und dessen minimale / maximale Koordinaten.

$$
\begin{aligned}
& \text{calcCoords}\left(\begin{pmatrix}4\\0\\4\end{pmatrix}, 1, 3\right) = \begin{pmatrix} min \\ max \end{pmatrix} \\
\Rightarrow \quad & min = \begin{pmatrix}4\\0\\4\end{pmatrix} + 2^1 \cdot \text{morton}^{-1}(3) = \begin{pmatrix}4\\0\\4\end{pmatrix} + \left(2 \cdot \begin{pmatrix}1\\1\\0\end{pmatrix}\right) = \begin{pmatrix}6\\2\\4\end{pmatrix} \\
\Rightarrow \quad & max = \begin{pmatrix}4\\0\\4\end{pmatrix} + 2^1 \cdot (\text{morton}^{-1}(3) + \mathbb{1}) = \begin{pmatrix}4\\0\\4\end{pmatrix} + \left(2 \cdot \left(\begin{pmatrix}1\\1\\0\end{pmatrix} + \mathbb{1}\right)\right) = \begin{pmatrix}8\\4\\6\end{pmatrix}
\end{aligned}
$$

A.4. Structure-of-Arrays und Arrays-of-Structures

Sollen mehrere Threads elementweise auf Daten im globalen Speicher der GPU zugreifen, so ist es für die effiziente Nutzung des Speicherbusses und Caches unabdingbar, dass die Elemente fortlaufend im Speicher angeordnet sind (Memory Coalescing). Die folgenden Code-Beispiele demonstrieren dies anhand der Nutzung von *Structures of Arrays* und einem *Array of Structures*.

Algorithmus 2 Non Coalesced

```
1  int N = 100;
2  struct {
3    float x, y, z;
4  } AoS;


7  __device__ pnt_cloud = AoS[N];

9  __device__
10 void scale(float factor){
11   int tid = blockIdx.x*blockDim.x +
         threadIdx.x;
12   if (tid < N){
13     pnt_cloud[tid].x *= factor;
14     pnt_cloud[tid].y *= factor;
15     pnt_cloud[tid].z *= factor;
16   }
17 }
```

Algorithmus 3 Coalesced

```
1  int N = 100;
2  struct {
3    float x[N];
4    float y[N];
5    float z[N];
6  } SoA;
7  __device__ SoA pnt_cloud;

9  __device__
10 void scale(float factor){
11   int tid = blockIdx.x*blockDim.x +
         threadIdx.x;
12   if (tid < N){
13     pnt_cloud.x[tid] *= factor;
14     pnt_cloud.y[tid] *= factor;
15     pnt_cloud.z[tid] *= factor;
16   }
17 }
```

In Algorithmus 2 können mit einer Speichertransaktion lediglich 10 bis 11 Threads eine Koordinate lesen, da diese mit 12 Bytes Abstand im Speicher liegen (3 `floats` zu 4 Bytes) und nicht an 128 Byte Grenzen ausgerichtet sind. Somit benötigt der Warp insgesamt 9 Transfers: 11+11+10 Threads lesen x, 11+10+11 lesen y, 10+11+11 lesen z. Dagegen können in Algorithmus 3 alle 32 Threads des Warps gleichzeitig eine Koordinate pro Speicherzugriff lesen (32 `floats` zu 4 Bytes = 128 Byte) und es werden insgesamt nur genau 3 Speichertransfers pro Warp (alle Threads lesen x, dann y, dann z) gebraucht.

A.5. Primitive der Parallelverarbeitung

Weit verbreitete Primitive der datenparallelen Verarbeitung sind: Transformation, Sortierung (Radix, Bitonic), Reduzierung / Verdichtung, Aufteilung. In dieser Arbeit wird häufig auf die Präfixsumme und die Radix-Sortierung zurückgegriffen, weshalb diese hier näher beschrieben werden:

A.5.1. Präfixsummen auf Threadebene

Die Problemstellung liegt in der Verdichtung (Compaction) von Daten anhand einer Entscheidungsfunktion (Prädikat $p(e) \in [1, 0]$), die auf jedes Datum angewendet wird. Nur wenn das Prädikat zu 1 ausgewertet wird, soll das Datum in die Ausgabedaten übernommen werden. Dabei dürfen keine Lücken entstehen und die Reihenfolge der Daten soll beibehalten werden. Bei einem seriellen oder schwach parallelen Ansatz wäre die Verwendung eines atomaren gemeinsamen Zählers, der den Zeiger in die Ausgabedaten inkrementiert, die naheliegendste Lösung. Bei einer massiven Parallelisierung auf der GPU wäre dies aus zwei Gründen nicht effizient: Zunächst weisen atomare Operationen eine hohe Latenz auf, zum anderen ist zusätzlicher Aufwand nötig, um die Reihenfolge der Daten aufrechtzuerhalten (sortieren nach Thread ID, da CUDA Threads keine deterministischer Reihenfolge besitzen). CUDA bietet jedoch hardwarenahe Funktionen an, um diese Aufgabe gegenüber der kanonischen Implementierung 32-fach zu beschleunigen. Grundlage ist der Befehl `__ballot(bool)`, der boolsche Variablen aller 32 Threads eines Warps zu einem Bitvektor der Länge 32 zusammenfasst, ohne dabei geteilten Speicher nutzen zu müssen. Die Anzahl der Einsen im Bitvektor kann mittels `__popc()` gezählt und über geteilten Speicher mit anderen Warps ausgetauscht werden. Algorithmus 4 zeigt die Nutzung dieser Funktionen zur Berechnung der Präfixsumme auf Thread-Ebene.

A.5.2. Parallelsierte Reduktion

Ziel einer Reduktion ist das Zusammenführen einer Menge von Eingabedaten zu einem einzelnen Wert durch die wiederholte Anwendung eines binären Operators. Ein einfaches Beispiel ist die Aufsummierung einer Menge aus Zahlen.

Wie Mark Harris in [96] schreibt, ist eine prinzipielle Umsetzung der Reduktion in CUDA zwar problemlos möglich, die performante Umsetzung jedoch mit zahlreichen Tücken versehen: Eine Baum-ähnliche Zusammenführung über Block-Grenzen hinweg birgt das

Problem einer Synchronisation zwischen den Blöcken, die zu Lasten der Laufzeit geht. Daher sollte eine Implementierung schrittweise über mehrere Kernel-Aufrufe (mit jeweils halbierter Anzahl an Blöcken) vorgehen und diese als Synchronisationspunkte nutzen. Lädt nun jeder Thread ein Element aus dem Globalen Speicher in den schnellen geteilten Speicher, darf nur jeder zweite Thread eine Zusammenführung berechnen. Diese Divergenz bedeutet eine niedrige Auslastung der Hardware. Daher sollten die zu fusionierenden Daten so über die Thread-ID ausgewählt werden, dass jeder Thread arbeiten kann, was bei einer falschen Strategie jedoch schnell zu Bank-Konflikte im geteilten Speicher führt. Harris beschreibt Lösungen dieser Punkte und präsentiert weitere Verbesserungen, die den Durchsatz und die Bandbreite der Reduktion nahe an die theoretischen Obergrenzen der Hardware bringen. In dieser Arbeit wird der Reduktionsalgorithmus aus Thrust genutzt, der die von Harris vorgeschlagenen Techniken umsetzt.

A.5.3. Parallelsierte Radix-Sortierung

Für die parallele Sortierung von Daten bietet sich eine Radix-Sortierung an, da diese keinen vergleichenden, sondern einen Zählenden Charakter aufweist. Radix besteht aus drei Phasen, die im Falle von Dezimalzahlen für die einzelne Stellen der zu sortierenden Werte ausgeführt werden, und die sich gut parallelisieren lassen:

1. **Zählen:** Hier wird das Vorkommen einzelner Werte einer Stelle gezählt, was sich effizient parallelisieren und mittels Reduktion zusammenfassen lässt.

2. **Bestimmung der Schreibposition**: Hier wird die Schreibposition jedes Eingabewertes anhand einer parallelen exklusiven Präfixsumme berechnet.

3. **Schreiben:** Nach dem die Positionen bekannt sind, werden auch diese sortiert. Somit kann das eigentliche Schreiben der neu geordneten Eingabedaten einen Memory Coalescing Effekt erzielen.

Da der Prozess stabil ist (also die Reihenfolge von Elemente mit gleichen Werten an der Sortierstelle nicht ändert), kann problemlos mehrfach sortiert werden, beginnend von der niederwertigsten Stelle.

Im Falle von Binär-Repräsentationen muss die Zahl der pro Durchlauf sortierten Bits (Stellen) an die Wortbreite der Hardware angepasst sein. Eine extrem performante Implementierung unter Ausnutzung von *Dynamic Parallelism* stellen Merrill et al. in [144] vor. Im dieser Arbeit wurde jedoch aus praktischen Gründen die Implementierung aus Thrust genutzt.

Algorithmus 4 Präfixsumme auf Thread-Ebene.

```
template<int NUM_WARPS, int WARP_SIZE>
__device__ int thread_prefix(uint32_t* shr_sum, uint32_t tid, bool pred) {
  uint32_t warp_votes = __ballot(pred); // warp vote
  if (tid % WARP_SIZE == tid / WARP_SIZE)
    shr_sum[tid / WARP_SIZE] = __popc(warp_votes); // population count
  __syncthreads();

  // exclusive sequential prefix sum
  if (tid == 0) {
    uint32_t sum = 0;
    #pragma unroll
    for (int i = 0; i < NUM_WARPS; ++i) {
      uint32_t tmp = shr_sum[i];
      shr_sum[i] = sum;
      sum += tmp;
    }
  }
  __syncthreads();

  int index = shr_sum[tid / WARP_SIZE];
  return index + __popc(warp_votes << (WARP_SIZE - (tid % WARP_SIZE)));
}
```

A.6. Partikelschwarmoptimierung

Die Partikelschwarmoptimierung ist ein biologisch motiviertes Optimierungsverfahren, das 1995 durch Kennedy et al. vorgestellt [115] wurde. Es bildet das Verhalten eines Schwarmes nach, dessen Individuen bzw. Partikel n einerseits eigenständig nach einem Optimum suchen, und andererseits durch das Verhalten des erfolgreichsten Schwarm-Individuums b beeinflusst werden. Dabei wandert jedes Partikel mit einer sich verändernden Geschwindigkeit $\vec{v}_n$ durch den Suchraum X des Optimierungsproblems und speichert dabei seine bisher beste erreichte Position $\vec{x}_{n\,\text{best}}$. Da der Schwarm ein randomisiert exploratives Verhalten aufweist, können lokale Minima bei passender Parametrierung vermieden werden. In dieser Arbeit wird die PSO dafür genutzt, nichtlineare Optimierungsprobleme anzugehen. Dafür gilt:

Maximiere die Bewertungsfunktion $f(\vec{x})$ unter der Nebenbedingung $\vec{x} \in X$ mit $f : W \to \mathbb{R}$ eine reellwertige Funktion und $X \subseteq W$. Die zulässige Menge X ist durch ihren konkreten Wertebereich beschrieben.

Die Implementierung iteriert nach einer Initialisierung durch drei Phasen, bis ein Maximum an Iterationen erreicht ist, oder ein Abbruchkriterium für das Ergebnis von $f(\vec{x})$ erreicht wurde.

Zunächst wird ein Schwarm mit einer festen Anzahl N an Partikeln erzeugt, indem jedem Individuum $n \in N$ ein randomisierter Startwert $\vec{x}_n$ zugewiesen wird. Danach startet die Optimierung:

1. Bewerte: Berechne $f(\vec{x}_n)$ für alle n und setze $\vec{x}_{n\,\text{best}}$, falls $f(\vec{x}_n) > \vec{x}_{n\,\text{best}}$
2. Ermittle Schwarm-Besten $\vec{x}_b = \arg\max_{n \in N}(f(\vec{x}_n))$ aus allen N Partikeln

3. Berechne neue Partikel-Geschwindigkeit:
   $\vec{v}_n = \vec{v}_n + \left(\alpha \cdot p_1 \cdot (\vec{x}_{n\,\text{best}} - \vec{x}_n)\right) + \left(\beta \cdot p_2 \cdot (\vec{x}_b - \vec{x}_n)\right)$
   mit Zufallsvariablen $p_1, p_2 \in [0, 1]$

4. Verschiebe Partikel: $\vec{x}_n = \vec{x}_n \cdot \vec{v}_n$

Dabei kann über die Faktoren α und β gesteuert werden, wie stark Partikel von ihrem eigenen Optimum $\vec{x}_{n\,\text{best}}$ oder dem Schwarm Optimum $\vec{b}$ angezogen werden. Durch die Zufallsvariablen p_1 und p_2 streut der Schwarm und konvergiert nicht direkt gegen ein einziges Maximum. Analog kann über entsprechende Umstellungen max $\rightarrow$ min auch ein Minimierungsproblem gelöst werden.

A.7. Octree

Hier werden einige Details und Datenstrukturen der Octree-Implementierung gelistet:

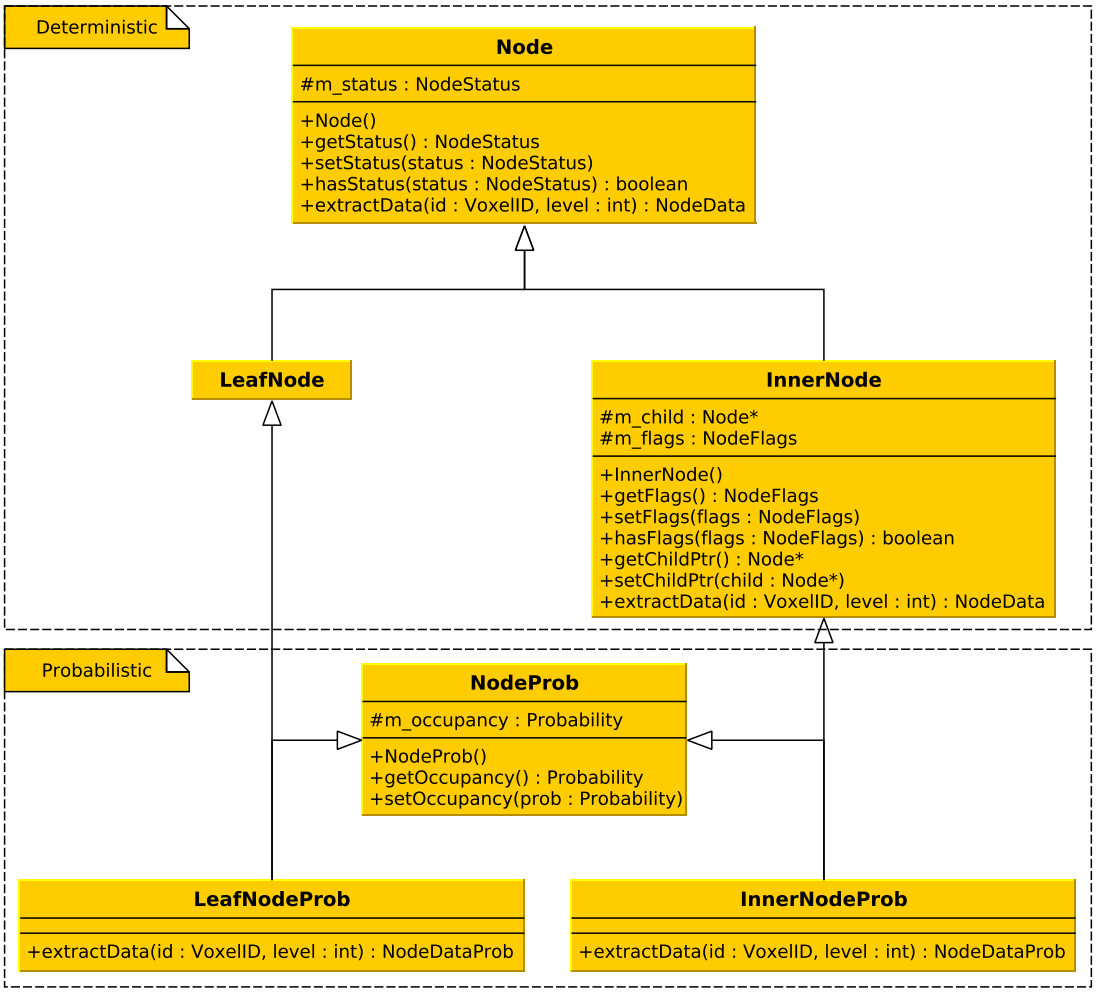


Abb. A.6.: UML Diagramme des Octree und von dessen Knoten.

A.7.1. Lastbalancierung (Balance Work)

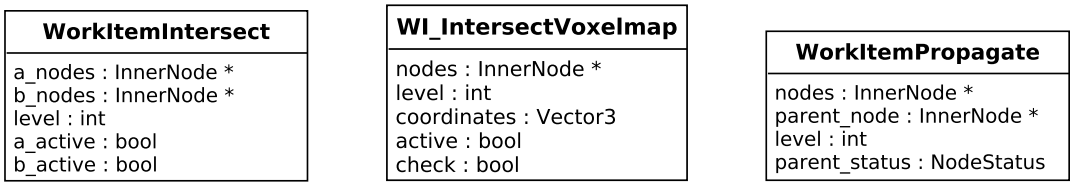
Algorithmus 5 ist der grundlegende Algorithmus zur Lastbalancierung bei der Verarbeitung unterschiedlich aufwendiger Berechnungselemente mit mehreren Threads. Der Algorithmus berechnet die Umverteilung der Elemente auf die Arbeitsstapel der Threads, unter Aufrechterhaltung der Stapel-Invariante.

Algorithmus 5 Balance Work

Require: $S_{in}[n]$: WorkItem, each Stack $S_{in}[i]$ fulfills invariant 5.21
Ensure: $S_{out}[n]$: WorkItem, each Stack $S_{out}[i]$ fulfills invariant 5.21

\\ count step
1: $C[0 : \#level - 1][0 : n - 1] \leftarrow \{0 \dots 0\}, \quad I[0 : n - 1][0 : \#level - 1] \leftarrow \{0 \dots 0\}$
2: **for** $i \leftarrow 0, n - 1$ **do** \\ in parallel
3: $S \leftarrow S_{in}[i], \quad S[0 : \#level - 1] \leftarrow \{0 \dots 0\}$
4: **for** $j \leftarrow tid, \text{size}(S) - 1, j \leftarrow j + \#threads$ **do** \\ in parallel
5: $S[\ \text{level}(S[j])\ ]\ +=\ 1$ \\ atomic increment
6: **end for**
7: **for** $j \leftarrow tid, \#level - 1, j \leftarrow j + \#threads$ **do** \\ in parallel
8: $C[\ \#level - l - 1\ ][i] \leftarrow S[j]$
9: **end for**
10: $I[i] \leftarrow suffixSum(S)$
11: **end for**
12: $C \leftarrow prefixSum(C)$ \\ in parallel
\\ move step
13: **for** $i \leftarrow 0, n - 1$ **do** \\ in parallel
14: $S \leftarrow S_{in}[i]$
15: **for** $j \leftarrow tid, \text{size}(S) - 1, j \leftarrow j + \#threads$ **do** \\ in parallel
16: $l \leftarrow level(S[j])$
17: $p \leftarrow C[\ \#level - l - 1\ ][i] + (j - I[i][l])$ \\ compute target position
18: $S_{out}[\ p \bmod n\ ][\ p/n\ ] = S[j]$
19: **end for**
20: **end for**

Die folgenden Algorithmen nutzen die beschriebene Lastbalancierung als Grundlage. Ihre Arbeitselemente, die zwischen den Arbeitsstapeln umverteilt werden, sind in Abb. A.7 zu sehen.



(a) Arbeitselement von *Intersect Octrees* (b) Arbeitselement von *Intersect Voxelmap* (c) Arbeitselement von *Load Balancing Propagate*

Abb. A.7.: Elemente der Arbeitsstapel aller Lastbalancierten Algorithmen.

A.7.2. Schneiden von zwei Octrees (Intersect Octrees)

Modifizierte Tiefensuche zur Kollisionsprüfung zwischen zwei Octrees. Nutzt den vorherigen Lastausgleich, wenn zu viele Threads untätig sind.

Algorithmus 6 Intersect Octrees

Require: $S[n]$: WorkItemIntersect, fulfills stack invariant 5.21
Ensure: $S[m]$: WorkItemIntersect, fulfills stack invariant 5.21
1: $\#collisions \leftarrow 0$
2: **while** $n > 0$ **and** $\#idle <$ MAX_IDLE **do**
3: $n \leftarrow n -$ min($\#threads/8$, n)
4: $w \leftarrow S[n + tid/8]$ \\ $w \leftarrow \perp$ if no item available
5: $(node_a, node_b) \leftarrow$ (a_nodes(w), b_nodes(w))
6: **if** a_active(w) **then** $node_a \leftarrow$ a_nodes(w)[$tid \bmod 8$] **end if**
7: **if** b_active(w) **then** $node_b \leftarrow$ b_nodes(w)[$tid \bmod 8$] **end if**
8: $(p_a, p_b, c) \leftarrow$ (isPart $(node_a)$, isPart $(node_b)$, areInConflict$(node_a, node_b)$)
9: **if** c **and** $((\neg p_a$ **and** $\neg p_b)$ **or** level$(w) > LEVEL_{stop})$ **then** \\ handle inner nodes
10: $\#collisions \leftarrow \#collisions + 8^{\text{level}(w)}$
11: **if** a_active(w) **then** setCollision$(node_a)$
12: **else** setCollision$(node_b)$ **end if**
13: **end if**
14: **if** level$(w) = 1$ **and** $(a_{active}$ **or** $b_{active})$ **and** $LEVEL_{stop} < 1$ **then** \\ handle leaf nodes
15: $\#collisions \leftarrow \#collisions +$ countCollisions$(node_a, node_b)$
16: **end if**
17: $insert_{tid} \leftarrow$ c **and** $(p_a$ **or** $p_b)$ **and** level$(w) >$ max$(1, LEVEL_{stop})$
18: $off \leftarrow \sum_{i=0}^{tid-1} insert_i$ \\ prefix sum of $insert_i \in \{0, 1\}$
19: **if** $insert_{tid}$ **then** \\ add new work items
20: $(a_a, a_b, c_a, c_b) \leftarrow$ (a_active(w) **and** p_a, b_active(w) **and** $p_b, node_a, node_b)$
21: **if** a_a **then** $c_a \leftarrow$ childPtr$(node_a)$ **end if**
22: **if** a_b **then** $c_b \leftarrow$ childPtr$(node_b)$ **end if**
23: $S[n + off] \leftarrow$ WorkItemIntersect$(c_a, c_b,$ level$(w) - 1, a_a, a_b)$
24: **end if**
25: **end while**
26: $(\#idle, m) \leftarrow (\#idle + 1, n)$

Umformung zur Aufwandsabschätzung des Abstiegs in einem voll ausgeprägten Octree. Durch die Verwendung der geometrischen Reihe lässt sich zeigen, dass der Aufwand lediglich $1/7$ höher als bei der Traversierung zweier Voxelkarten liegt.

$$A_{oct}(s) = 2\left(k + \sum_{i=1}^{\log_8 \frac{s}{p}} 8^i\right) \tag{A.6}$$

$$\overset{z=\log_8 \frac{s}{p}}{=} 2\left(k - 1 + \frac{s}{p}\sum_{i=0}^{z} 8^{-z}8^i\right) \tag{A.7}$$

$$= 2\left(k - 1 + \frac{s}{p}\sum_{i=0}^{z} \frac{1}{8^{z-i}}\right) \tag{A.8}$$

$$= 2\left(k - 1 + \frac{s}{p}\sum_{i=0}^{z} \left(\frac{1}{8}\right)^i\right) \tag{A.9}$$

$$\leq 2\left(k - 1 + \frac{s}{p}\sum_{i=0}^{\infty} \left(\frac{1}{8}\right)^i\right) \overset{\text{geom. Reihe}}{=} 2\left(k - 1 + \frac{s}{p} \cdot \frac{8}{7}\right) \tag{A.10}$$

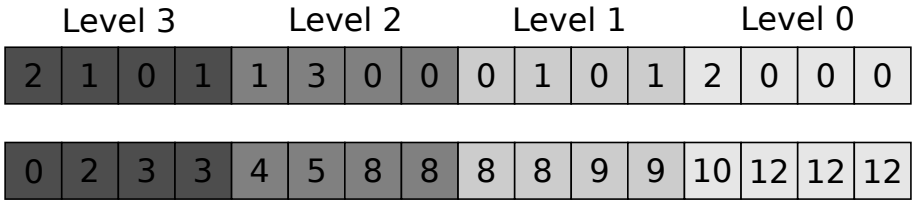


Abb. A.8.: Array C aus Algorithmus 5 vor und nach Berechnung der Präfixsumme über C. Beispiel nutzt Eingabedaten aus Abb. 5.12.

A.7.3. Eingeschränkte Zwei-Phasen-Tiefensuche mit Lastausgleich

In Algorithmus 7 (Load-Balancing Propagate) führt jeder CUDA Block mehrere Tiefensuchen für die Elemente (siehe Abb. A.7c) auf seinem Arbeitsstapel durch. Auch wenn die Anzahl der Arbeitsstapel-Elemente durch den Algorithmus verändert werden kann, so gilt vor und nach der Abarbeitung die Stapel-Invariante (siehe Gleichung 5.21), welche die Voraussetzung für den effizienten Lastausgleich und für eine Abschätzung des Aufwandes des gesamten Arbeitsstapels ist. Zu Beginn nehmen sich immer acht Threads dasselbe Arbeitselement w vom Arbeitsstapel (sind nicht ausreichend Elemente vorhanden, werden leere Elemente $w \leftarrow \perp$ und bearbeiten gemeinsam die acht Kindknoten. Zunächst wird geprüft, ob neue Arbeitselemente ($insert_{\downarrow}$ bzw. $insert_{\uparrow}$) entstehen. Um mehrfache Bearbeitungen zu verhindern, kann nur einer der acht Threads neue *Bottom-Up* Elemente generieren. An dieser Stelle wird auch abgebrochen, wenn der betrachtete Knoten nicht im relevanten Sektor des Baumes liegt (updateFlag), oder keine Kindknoten vorhanden sind (isPart). Zeile 9 und 10 initialisieren neu erstellte Knoten mit dem Status ihres Elternknotens. Die eigentlich Herstellung der Baum-Invariante beginnt in Zeile 16: Nachdem sichergestellt ist, dass alle benötigten Konten bereits aktualisiert sind (updateFlag), wird der Zustand des neuen Elternknotens als $\|$ -Verknüpfung der Kindknoten gesetzt und sein updateFlag entsprechend entfernt. Ist die Datenabhängigkeit nicht aufgelöst, wird das Arbeitselement durch den ersten Thread wieder auf den Arbeitsstapel

gelegt, und über das *makeProgress*-Flag gespeichert, dass kein Fortschritt erzielt wurde. Die ‖ -Verknüpfung ist effizient über die WARP-Funktion *__ballot()* umgesetzt. Der Algorithmus profitiert von Memory Coalescing, da auf Knotenelemente immer geordnet über die Thread-ID zugegriffen wird. Weiterhin wird die letzte Baumebene zur Optimierung ohne Arbeitsstapel abgearbeitet (ab Zeile 12), wobei analog zum beschriebenen Vorgehen verfahren wird.

Am Ende jeden Durchlaufs müssen die generierten Elemente auf den Arbeitsstapel gelegt werden, und dabei die Stapel-Invariante erfüllen. Dafür sind in der *computeOffset(*$insert_{\downarrow}$, $insert_{\uparrow}$*)* Funktion die folgenden Summen implementiert:

$$\begin{aligned} off_{\downarrow}^{tid} &:= \sum_{i=0}^{tid-1} insert_{\downarrow}^{i} + \sum_{i=0}^{\max_k(level^k = level^{tid})} insert_{\uparrow}^{i} \\ off_{\uparrow}^{tid} &:= \sum_{i=0}^{tid-1} insert_{\uparrow}^{i} + \sum_{i=0}^{\min_k(level^k = level^{tid})-1} insert_{\downarrow}^{i} \end{aligned} \tag{A.11}$$

Sie berechnen aus den Flags $insert_{\downarrow}$ und $insert_{\uparrow}$ der einzelnen Threads die relativen Positionen $off_{\downarrow}$ und $off_{\uparrow}$ der neuen Elemente im Arbeitsstapel, so dass diese parallel zurückgeschrieben werden können. Implementiert sind sie als effiziente Prefixsumme mittels *__ballot()* und *__popc()*, bzw. als zur Laufzeit erzeugte Lookup-Tabelle für die min/max Ausdrücke.

Algorithmus 7 Load Balancing Propagate

Require: $S[n]$: WorkItemPropagate, fulfills stack invariant 5.21
Ensure: $S[m]$: WorkItemPropagate, fulfills stack invariant 5.21

1: $makeProgress \leftarrow$ true
2: **while** $n > 0$ **do**
3: $n \leftarrow n$ - min$(\#threads/8, n)$
4: $w \leftarrow S[n + tid/8]$ \\ $w \leftarrow \perp$ if no item available
5: $node \leftarrow$ node(w)[$tid \bmod 8$]
6: $makeProgress_{tid} \leftarrow w \neq \perp$
7: $insert_{\uparrow} \leftarrow$ isTopDown(w) **and** level(w) $\neq 1$ **and** $tid \bmod 8 = 0$
8: $insert_{\downarrow} \leftarrow$ isTopDown(w) **and** level(w) $\neq 1$ **and** updateFlag($node$) **and** isPart($node$)
\\ process work items
9: **if** isTopDown(w) **and** hasInvalidState($node$) **then** \\ propagate$_{\downarrow}$
10: setState($node$, parentState($node$))
11: **end if**
12: **if** isBottomUp(w) **and** level(w) $= 1$ **and** isPart($node$) **then** \\ handle without stack
13: leaf_propagate$_{\downarrow}$()
14: leaf_propagate$_{\uparrow}$()
15: **end if**
16: **if** isBottomUp(w) **then** \\ propagate$_{\uparrow}$
17: $children \leftarrow$ node(w)
18: **if** $\forall 0 < i < 8 \mid$ updateFlag($children[i]$) = false **then** \\ assure children are up-to-date
19: $newState \leftarrow$ OR$_{0<i<8}$ state($children[i]$) \\ disjunction of children
20: setState(parentNode(w), $newState$)
21: setUpdateFlag(parentNode(w), false)
22: **else**
23: $insert_{\uparrow} \leftarrow tid \bmod 8 = 0$ \\ back on stack
24: $makeProgress_{tid} \leftarrow$ false
25: **end if**
26: **end if**
\\ push work items on stack
27: ($off_{\downarrow}$, $off_{\uparrow}$, $\#insert_{\downarrow}$, $\#insert_{\uparrow}$) $\leftarrow$ computeOffset($insert_{\downarrow}$, $insert_{\uparrow}$)
28: **if** $insert_{\downarrow}$ **then** \\ WorkItemPropagate(node, parent_node, level, state)
29: $S[n + off_{\downarrow}] \leftarrow$ WorkItemPropagate$_{\downarrow}$(childPtr($node$), $node$, level($node$) -1, state($node$))
30: **end if**
31: **if** $insert_{\uparrow}$ **then** $S[n + off_{\uparrow}] \leftarrow$ WorkItemPropagate$_{\uparrow}(w)$ **end if**
32: $n \leftarrow n + \#insert_{\downarrow} + \#insert_{\uparrow}$
33: $makeProgress \leftarrow$ OR$_{0<i<\#threads}$ $makeProgress_i =$ true
34: **if** $\neg makeProgress$ **or** $\#idle \geq$ MAX_IDLE **then break end if**
35: **end while**
36: **if** $makeProgress$ **then** $\#idle \leftarrow \#idle + 1$ **end if**
37: $m \leftarrow n$

A.7.4. Verwendete Hard- und Software

Um die Laufzeitangaben in Kapitel 8 in Relation setzen zu können, ist hier die verwendete Hardware aufgelistet:

Host-System Workstation PC

- Prozessor: Intel® Core™ i7-4770 4-Kern CPU
- Hauptspeicher: 8 GB DDR3 RAM

Grafikkarte NVIDIA® GeForce® TITAN

- Kepler Architektur (GK110)
- 2688 Kerne (4,709 GFLOPS)
- 14 Streaming Multiprozessoren zu je 192 Threads (= 6 Warps)
- 6144 MB GDDR5-Hauptspeicher (384 Bit Speicherinterface, 6 GHz: 288.4 GB/s Speicherdurchsatz)
- Host-Anbindung über eine 16× PCI-Express 3.0 Schnittstelle

Software Ubuntu 16.04 LTS

- CUDA 7.5
- GCC 4.9 (Optimierungsstufe *-O3*)

A.7.5. Unscharfe Prüfung von Bitvektor-Voxeln mittels Zeitfenster

Der gelistete Code iteriert in Byte Schritten über alle SSV-IDs von Vektor `v2` und prüft dabei in einer inneren Schleife alle im Fenster liegenden IDs aus Vektor `v1`. Dafür wird das entsprechende Byte aus `v2` in den Puffer geladen und dort bitweise um die Fensterbreite geshifted. In jedem Durchlauf der inneren Schleife erfolgt eine Bitweise &-Operation mit dem betrachteten Byte aus `v1` und dem verschobenen Byte aus `v2`.

Algorithmus 8 Bitshifting in der gefensterten Kollisionsprüfung.

```
1   uint64_t buffer = 0;
2   const size_t buffer_half = 4*8; // middle of uint64_t
3   if (m_type_range > buffer_half)
4   {
5   printf("ERROR: Window size for SV collision check must be smaller than %lu\n",
        buffer_half);
6   }

8   // Fill buffer with first 4 bytes. We start at byte 1 and not 0 because we're only
        interested in SV IDs
9   for (size_t byte_nr = 1; byte_nr < 5; ++byte_nr)
10  {
11  buffer += static_cast<uint64_t>(v2.bitVector().getByte(byte_nr * 8)) << (3*8 +
        byte_nr*8);
12  }

14  // We start at bit 8 and not 0 because we're only interested in SV IDs
15  for (uint32_t i = 8; i < eVT_SWEPT_VOLUME_END; i+=8)
16  {

18  uint8_t byte = 0;
19  uint64_t byte_1 = static_cast<uint64_t>(v1.bitVector().getByte(i)) << (buffer_half
        -m_type_range);

21  // Check range for collision
22  for (size_t bitshift_size=0; bitshift_size <= 2*m_type_range; ++bitshift_size)
23  {
24  byte |= (byte_1 & buffer) >> (buffer_half - m_type_range + bitshift_size);
25  //    if ((byte_1 & buffer) != 0)
26  //    {
27  //      printf("Byte_1 step %u is %lu, buffer is %lu, Overlapping: %u\n", i/8,
        byte_1, buffer, byte);
28  //    }
29  byte_1 = byte_1 << 1;
30  }

32  collisions->setByte(i, byte);

34  // Move buffer along bitvector
35  buffer = buffer >> 8;
36  if (i < length - buffer_half)
37  {
38  buffer += static_cast<uint64_t>(v2.bitVector().getByte(i + buffer_half)) << 56;
39  }
40  }
41  return !collisions->isZero();
```

A.7.6. Backtracking für Scheduling

Verwendeter Algorithmus zur Bestimmung einer kollisionsfreien Ausführungsfolge bei der Verwendung mehrerer Roboter:

Algorithmus 9 Backtracking für Scheduling.

```
1    fitMotions( Robot[] robots )
2    {
3    result = Result.emptyResult();
4    if( recursiveFit(result, 0, robots.first) )
5    return result;
6    else
7    return Result.emptyResult();
8    }

10   recursiveFit( Result result, int timeIndex, Robot currentRobot) )
11   {
12   if( timeIndex >= getNumSlots() )
13   {
14   if( currentRobot.hasNext() )
15   return recursiveFit( result, 0, currentRobot.next() );
16   else
17   return true;
18   }
19   foreach( movement in currentRobot.movements)
20   {
21   if( result.collides(movement, timeIndex) )
22   continue;
23   solution.get(currentRobot).push_back(movement);
24   if( recursiveFit( result, timeIndex + 1, currentRobot) )
25   return true;
26   solution.get(currentRobot).pop_back(movement);
27   }
28   return false;
29   }
```

A.8. Visualisierung

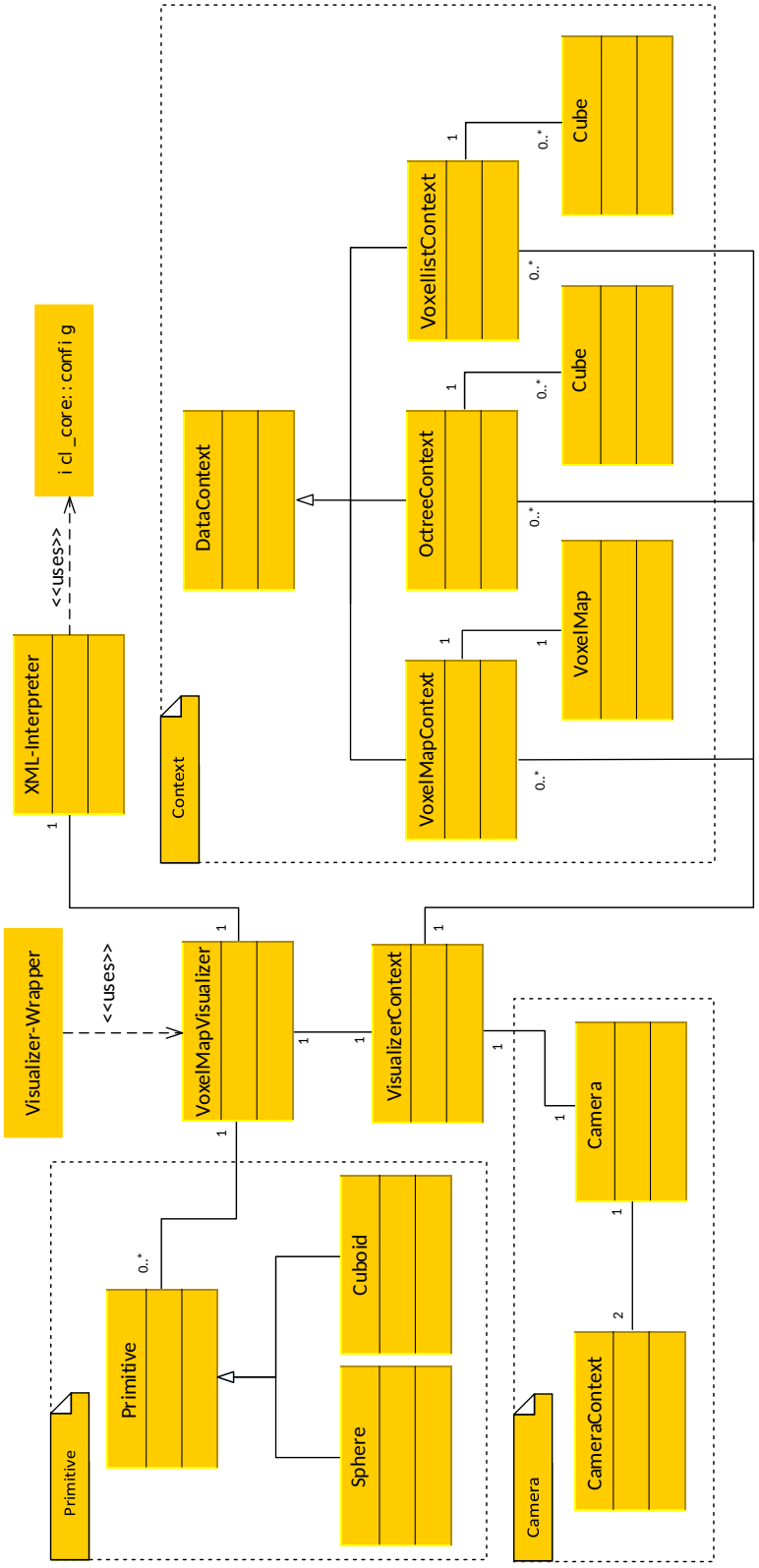


Abb. A.9.: Klassendiagramm der Visualisierung

A.9. Greifplanung

Algorithmus zur lokalen Optimierung der Objektpose innerhalb der Hand während der Greifplanung:

Algorithmus 10 Local Grasp Optimizer

Require:
O : Pointcloud of object surface
H : Swept-Volume of grasp
P : Particle defining initial (x, y), (α, β, γ) pose of the object in hand
Ensure:
$\vec{\varphi}_{\text{best}}$: Finger joint angles for best grasp
P: Contains object pose for best grasp

```
1: O ← transform(O, P)                                  \\ Place object at initial pose
2: O ← transform(O, y-start)                            \\ Stick object into hand
3: for xz-shift ← -max-shift to max-shift do
4:   for y-shift ← 0 mm to 200 mm do                    \\ Move object out of hand
5:     offset ← xz-shift + y-shift
6:     #colls ← collisionCheckOffset(O, H, offset)
7:     if #colls == 0 then
8:       φ⃗, #colls ← sweptCollisionCheck(O, H, offset)
9:       if #colls > #colls_best then
10:        #colls_best ← #colls                         \\ Local optimum found
11:        pose_best ← offset
12:        φ⃗_best ← φ⃗
13:      end if
14:    end if
15:  end for
16: end for
17: return P, φ⃗_best
```

Abbildungsverzeichnis

Tabellenverzeichnis

Eigene Veröffentlichungen

Dieses Verzeichnis listet alle Publikationen, bei denen der Autor dieser Dissertation entweder der Erstautor ist, oder als Co-Autor maßgeblich zu der Veröffentlichung beigetragen hat (in Form von Problemstellung, -lösung, Diskussion oder experimenteller Evaluation).

[1] Uwe Herbst, Steffen W. Rühl, Andreas Hermann, Zhixing Xue, and Klaus Bengler. Ergonomic 6d interaction technologies for a flexible and transportable robot system: A comparison. In *12th IFAC/IFIP/IFORS/IEA Symposium on Analysis, Design, and Evaluation of Human-Machine Systems, IFAC HMS 2013*, pages 58–63, 2013.

[2] Andreas Hermann. *Industrie 4.0 –Use Cases aus Forschung und Unternehmenspraxis: viEMA: Schwankenden Stückzahlen in Industrie 4.0 durch flexiblen Wechsel zwischen Hand- und Automatenmontage begegnen*, pages 133–246. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[3] Andreas Hermann, Jörg Bauer, Sebastian Klemm, and Ruediger Dillmann. Mobile manipulation planning optimized for GPGPU Voxel-Collision detection in high resolution live 3d-maps. In *Conference ISR ROBOTIK 2014*, Munich, Germany, June 2014.

[4] Andreas Hermann, Florian Drews, Joerg Bauer, Sebastian Klemm, Arne Roennau, and Ruediger Dillmann. Unified GPU voxel collision detection for mobile manipulation planning. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 4154–4160, Sept 2014.

[5] Andreas Hermann, Sebastian Klemm, Zhixing Xue, Arne Roennau, and Ruediger Dillmann. Ergonomic Rating of Interaction Technologies for A Mobile Robot System. In *Human-Computer Interaction (HCI 2013), 15th International Conference on*, jul. 2013.

[6] Andreas Hermann, Sebastian Klemm, Zhixing Xue, Arne Roennau, and Ruediger Dillmann. GPU-based Real-Time Collision Detection for Motion Execution in Mobile Manipulation Planning. In *16th International Conference on Advanced Robotics, ICAR 2013*, 2013.

[7] Andreas Hermann, Felix Mauch, Klaus Fischnaller, Sebastian Klemm, Arne Roennau, and Ruediger Dillmann. Anticipate your surroundings: Predictive collision detection between dynamic obstacles and planned robot trajectories on the GPU. In *Mobile Robots (ECMR), 2015 European Conference on*, pages 1–8, Sept 2015.

[8] Andreas Hermann, Felix Mauch, Sebastian Klemm, Arne Roennau, and Ruediger Dillmann. Eye in hand: Towards GPU accelerated online grasp planning based on pointclouds from in-hand sensor. In *Humanoid Robots, International Conference on*, Nov 2016.

[9] Andreas Hermann, Jian Sun, Xue Zhixing, Steffen W. Ruehl, Jan Oberlaender, Arne Roennau, J. Marius Zoellner, and Ruediger Dillmann. Hardware and Software Architecture of the Bimanual Mobile Manipulation Robot HoLLiE and its Actuated Upper Body. In *Advanced Intelligent Mechatronics (AIM), 2013 IEEE/ASME International Conference on*, pages 286 –292, jul. 2013.

[10] Andreas Hermann, Zhixing Xue, Steffen W. Ruehl, and Ruediger Dillmann. Hardware and software architecture of a bimanual mobile manipulator for industrial application. In *Robotics and Biomimetics (ROBIO), 2011 IEEE International Conference on*, pages 2282 –2288, dec. 2011.

[11] Sebastian Klemm, Jan Oberländer, Andreas Hermann, Arne Roennau, Thomas Schamm, J. Marius Zollner, and Ruediger Dillmann. Rrt star-connect: Faster, asymptotically optimal motion planning. In *2015 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 1670–1677, Dec 2015.

[12] Jan Oberländer, Tanja Harbaum, Gerhard Kurz, Nadia Ahmed, Tomislav Kos-Grabar, Andreas Hermann, Arne Rönnau, and Rüdiger Dillmann. A student-built ball-throwing robotic companion for hands-on robotics education. In *Proceedings of the 14th International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines (CLAWAR)*, pages 233–240, Paris, France, September 2011.

[13] Lars Pfotzer, Martin Staehler, Andreas Hermann, Arne Roennau, and Ruediger Dillmann. Kairo 3: Moving over stairs amp; unknown obstacles with reconfigurable snake-like robots. In *Mobile Robots (ECMR), 2015 European Conference on*, pages 1–6, Sept 2015.

[14] Steffen W. Ruehl, Andreas Hermann, Zhixing Xue, Thilo Kerscher, and Ruediger Dillmann. Generating a symbolic scene description for robot manipulation using physics simulation. In *Multibody Dynamics*, Brussels, Belgium, July 2011.

[15] Steffen W. Ruehl, Andreas Hermann, Zhixing Xue, Thilo Kerscher, and Ruediger Dillmann. Graspability: A description of work surfaces for planning of robot manipulation sequences. In *ICRA*, Shanghai, China, May 2011.

[16] Steffen W. Ruehl, Christopher Parlitz, Georg Heppner, Andreas Hermann, Arne Roennau, and Ruediger Dillmann. Experimental evaluation of the schunk 5-Finger gripping hand for grasping tasks. In *Robotics and Biomimetics (ROBIO), 2014 IEEE International Conference on*, pages 2465–2470, Dec 2014.

[17] Zhixing Xue, Steffen W. Ruehl, Andreas Hermann, Thilo Kerscher, and Ruediger Dillmann. Autonomous grasp and manipulation planning using a tof camera. In *IROS*, Shanghai, China, October 2010.

[18] Zhixing Xue, Steffen W. Ruehl, Andreas Hermann, Thilo Kerscher, and Ruediger Dillmann. An autonomous ice-cream serving robot. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3451–3452, May 2011.

[19] Zhixing Xue, Steffen W. Ruehl, Andreas Hermann, Thilo Kerscher, and Ruediger Dillmann. Autonomous grasp and manipulation planning using a tof camera. *Robotics and Autonomous Systems*, 60(3):387 – 395, 2012. Autonomous Grasping.

Studentische Arbeiten

Dieses Verzeichnis listet studentische Arbeiten, die durch den Autor dieser Dissertation im Rahmen seiner Forschung ausgeschrieben und betreut wurden. Dies beinhaltet die maßgebliche Vorgabe der Problemstellung, Diskussion der Arbeit, sowie Randvorgaben zur Lösung, Visualisierung und experimentellen Evaluation.

[20] Jörg Bauer and Andreas Hermann. Dynamic Mobile Manipulation Planning based on GPU Voxel-Swept-Volume Collision Detection. Diplomarbeit, KIT Karlsruher Institut für Technologie, Karlsruhe, Germany, 2013.

[21] Tobias Cichos and Andreas Hermann. Greifpunktsynthese für Stoffstücke anhand von 3D-Punktwolken. Studienarbeit, KIT Karlsruher Institut für Technologie, Karlsruhe, Germany, 2013.

[22] Florian Drews and Andreas Hermann. GPU-optimierter Octree mit Algorithmen zur effizienten geometrischen Modellierung in der Robotik. Masterarbeit, KIT Karlsruher Institut für Technologie, Karlsruhe, Germany, 2014.

[23] Klaus Fischnaller and Andreas Hermann. GPU-basierte Evaluierung von Swept-Volumes zur Trajektorienplanung mit Bewegungsprimitiven. Masterarbeit, KIT Karlsruher Institut für Technologie, Karlsruhe, Germany, 2015.

[24] Christian Jülg and Andreas Hermann. Fast online collision avoidance for mobile service robots through potential fields on 3D environment data processed on GPUs. Diplomarbeit, KIT Karlsruher Institut für Technologie, Karlsruhe, Germany, 2016.

[25] Sebastian Klemm and Andreas Hermann. Autonome Kamera- und Hand-Auge-Kalibrierung eines zweiarmigen Robotersystems. Studienarbeit, KIT Karlsruher Institut für Technologie, Karlsruhe, Germany, 2010.

[26] Sebastian Klemm and Andreas Hermann. GPU-basierte 3D-Kollisionserkennung zur Online-Evaluierung von Bewegungstrajektorien. Diplomarbeit, KIT Karlsruher Institut für Technologie, Karlsruhe, Germany, 2013.

[27] Felix Mauch and Andreas Hermann. GPU-basierte Bewegungsprädiktion von Objekten in 3D-Punktwolken zur Kollisionsvermeidung. Masterarbeit, KIT Karlsruher Institut für Technologie, Karlsruhe, Germany, 2015.

[28] Herbert Pietrzyk and Andreas Hermann. Onlinefähige GPU-basierte Segmentierung und Klassifikation von Objekten in Punktwolkendaten aus urbanen Umgebungen. Bachelorarbeit, KIT Karlsruher Institut für Technologie, Karlsruhe, Germany, 2016.

[29] Tim Pollert and Andreas Hermann. Fehlerreduktion in KinFu Large Scale Raummodellen mit Hilfe von SLAM-Algorithmen. Bachelorarbeit, KIT Karlsruher Institut für Technologie, Karlsruhe, Germany, 2014.

[30] David Ruscheweyh and Andreas Hermann. Evaluierung von Algorithmen zur reaktiven Berechnung der inversen Kinematik für überaktuierte Roboter. Masterarbeit, KIT Karlsruher Institut für Technologie, Karlsruhe, Germany, 2013.

[31] Jian Sun and Andreas Hermann. Konstruktion und Aufbau eines Serviceroboters mit passivem Gewichtsausgleich durch eine federgespannte Parallelogramm-Struktur. Diplomarbeit, KIT Karlsruher Institut für Technologie, Karlsruhe, Germany, 2013.

[32] Matthias Wagner and Andreas Hermann. Echtzeit OpenGL-Visualisierung von umfangreichen CUDA-Datenstrukturen mittels Shared Memory. Bachelorarbeit, KIT Karlsruher Institut für Technologie, Karlsruhe, Germany, 2014.

Literaturverzeichnis

[33] Steven Abrams and Peter K Allen. Computing swept volumes. *The Journal of Visualization and Computer Animation*, 11(2):69–82, 2000.

[34] M Acuña and Takayuki Aoki. Real-time tsunami simulation on multi-node gpu cluster. In *ACM/IEEE conference on supercomputing*, 2009.

[35] James S Albus, H McCain, and Ronald Lumia. Nasa/nbs standard reference model for telerobot control system architecture (nasrem). *Technical Note (NIST TN)-1235*, 1989.

[36] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[37] Michael Riis Andersen, Thomas Jensen, Pavel Lisouski, Anders Krogh Mortensen, Mikkel Kragh Hansen, Torben Gregersen, and Peter Ahrendt. Kinect depth sensor evaluation for computer vision applications. Technical report, Århus Universitet, 2012.

[38] Oliver Armbruster. Kollisionsbestimmung mittels Swept-Volumes zur effizienten Bewegungsplanung, March 2013.

[39] Asma Azim and Olivier Aycard. Detection, classification and tracking of moving objects in a 3D environment. In *Intelligent Vehicles Symposium*, pages 802–807. IEEE, 2012.

[40] Jeroen Baert, Ares Lagae, and Ph Dutré. Out-of-core construction of sparse voxel octrees. In *Computer Graphics Forum*, volume 33, pages 220–227. Wiley Online Library, 2014.

[41] Jérôme Barraquand, Lydia Kavraki, Jean-Claude Latombe, Tsai-Yen Li, Rajeev Motwani, and Prabhakar Raghavan. *A Random Sampling Scheme for Path Planning*, pages 249–264. Springer London, London, 1996.

[42] Jennifer Barry, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. A hierarchical approach to manipulation with diverse actions. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 1799–1806. IEEE, 2013.

[43] J. Baumgartl, T. Werner, P. Kaminsky, and D. Henrich. A fast, GPU-based geometrical placement planner for unknown sensor-modelled objects and placement areas. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1552–1559, May 2014.

[44] Patrick Beeson and Barrett Ames. TRAC-IK: An open-source library for improved solving of generic inverse kinematics. In *Proceedings of the IEEE RAS Humanoids Conference*, Seoul, Korea, November 2015.

[45] D Berenson, S S Srinivasa, and J J Kuffner. Addressing pose uncertainty in manipulation planning using Task Space Regions. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 1419–1425, 2009.

[46] J. Bialkowski, S. Karaman, and E. Frazzoli. Massively parallelizing the RRT and the RRT*. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3513–3518, Sept 2011.

[47] J. Będkowski, K. Majek, and A. Nüchter. General Purpose Computing on Graphics Processing Units for Robotic Applications. volume 4, pages 23–33, 2013.

[48] J. Bohg, M. Johnson-Roberson, B. León, J. Felip, X. Gratal, N. Bergström, D. Kragic, and A. Morales. Mind the gap - robotic grasping under incomplete observation. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 686–693, 5 2011.

[49] Dorit Borrmann, Jan Elseberg, Kai Lingemann, Andreas Nüchter, and Joachim Hertzberg. Globally consistent 3d mapping with scan matching. *Robotics and Autonomous Systems*, 56(2):130–142, 2008.

[50] Michael Boyer, David Tarjan, Scott T Acton, and Kevin Skadron. Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[51] Jack E Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal*, 4(1):25–30, 1965.

[52] Oliver Brock, Oussama Khatib, and S. Viji. Task-consistent obstacle avoidance and motion behavior for mobile manipulation. *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, (May):388–393, 2002.

[53] John Canny. *The complexity of robot motion planning*. MIT press, 1988.

[54] Thanh-Tung Cao, Ke Tang, Anis Mohamed, and Tiow-Seng Tan. Parallel Banding Algorithm to Compute Exact Distance Transform with the GPU. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 83–90, New York, NY, USA, 2010. ACM.

[55] Daniel Cederman and Philippas Tsigas. Dynamic Load Balancing Using Work-Stealing. *GPU Computing Gems Jade Edition*, pages 485–499, 2011.

[56] Vipin Chaudhary, K. Kamath, P. Arunachalam, and J. K. Aggarwal. Parallel manipulations of octrees and quadtrees. In Akira Nakamura, Maurice Nivat, Ahmed Saoudi, Patrick S. P. Wang, and Katsushi Inoue, editors, *Parallel Image Analysis*, pages 69–86, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[57] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R Gao. Dynamic load balancing on single-and multi-GPU systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[58] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA C Programming*. John Wiley & Sons, 2014.

[59] Jaebum Choi, Simon Ulbrich, Bernd Lichte, and Markus Maurer. Multi-Target Tracking using a 3D-Lidar sensor for autonomous vehicles. In *Intelligent Transportation Systems - (ITSC), 2013 16th International IEEE Conference on*, 2013.

[60] Benjamin Choo, Michael Landau, Michael DeVore, and Peter A Beling. Statistical analysis-based error models for the microsoft kinecttm depth sensor. *Sensors*, 14(9):17430–17450, 2014.

[61] P. Cignoni. DeWall: A fast divide and conquer Delaunay triangulation algorithm in Ed. *Computer-Aided Design*, 30(5):333–341, April 1998.

[62] James H Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[63] T. T. Cociaş, S. M. Grigorescu, and F. Moldoveanu. Multiple-superquadrics based object surface estimation for grasping in service robotics. In *Optimization of Electrical and Electronic Equipment (OPTIM), 2012 13th International Conference on*, pages 1471–1477, 5 2012.

[64] Benjamin J Cohen, Gokul Subramania, Sachin Chitta, and Maxim Likhachev. Planning for manipulation with adaptive motion primitives. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 5478–5485. IEEE, 2011.

[65] NVIDIA Corporation. CUDA Toolkit Documentation, 2015.

[66] Nicolas Cuntz and Andreas Kolb. Fast Hierarchical 3D Distance Transforms on the GPU. In *Eurographics 2007 (short paper)*, pages 93–96, September 2007.

[67] Marija Dakulovic, Christoph Sprunk, Luciano Spinello, Ivan Petrovic, and Wolfram Burgard. Efficient navigation for anyshape holonomic mobile robots in dynamic environments. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 2644–2649. IEEE, 2013.

[68] J. Denavit and R. S. Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. *Trans. ASME, Journal of Applied Mechanism*, 22(2):215 – 221, 1955.

[69] D. Devaurs, T. Simeon, and J. Cortes. Parallelizing rrt on distributed-memory architectures. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 2261 –2266, may 2011.

[70] Balázs Dezső, Alpár Jüttner, and Péter Kovács. Lemon–an open source c++ graph template library. *Electronic Notes in Theoretical Computer Science*, 264(5):23–45, 2011.

[71] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[72] N. Duffy, D. Allan, and J. T. Herd. Real-time collision avoidance system for multiple robots operating in shared work-space. *IEE Proceedings E - Computers and Digital Techniques*, 136(6):478–484, Nov 1989.

[73] Erick Dupuis, Ioannis Rekleitis, Jean-Luc Bedwani, Tom Lamarche, Pierre Allard, and Wen-Hong Zhu. Over-the-horizon autonomous rover navigation: experimental results. In *International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS), Los Angeles, CA*, 2008.

[74] S. El-Khoury, A. Sahbani, and V. Perdereau. Learning the natural grasping component of an unknown object. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2957–2962, Oct 2007.

[75] Jan Elseberg, Stéphane Magnenat, Roland Siegwart, and Andreas Nüchter. Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. *Journal of Software Engineering for Robotics*, 3(1):2–12, 2012.

[76] Christer Ericson. *Real-Time Collision Detection*. CRC Press, Inc., Boca Raton, FL, USA, 2004.

[77] Ricardo Fabbri, Luciano Da F. Costa, Julio C. Torelli, and Odemir M. Bruno. 2d euclidean distance transform algorithms: A comparative survey. *ACM Computing Surveys (CSUR)*, 40(1):2:1–2:44, February 2008.

[78] Farbod Fahimi. *Autonomous robots: modeling, path planning, and control*, volume 107. Springer Science & Business Media, 2008.

[79] B. Faverjon. Obstacle avoidance using an octree in the configuration space of a manipulator. In *Proceedings. 1984 IEEE International Conference on Robotics and Automation*, volume 1, pages 504–512, Mar 1984.

[80] Fabrizio Flacco, Torsten Kröger, Alessandro De Luca, and Oussama Khatib. A depth space approach to human-robot collision avoidance. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 338–345. IEEE, 2012.

[81] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972.

[82] Uwe Franke, Clemens Rabe, Hernán Badino, and Stefan K. Gehrig. 6D-Vision: Fusion of Stereo and Motion for Robust Environment Perception. In Walter G. Kropatsch, Robert Sablatnig, and Allan Hanbury, editors, *DAGM-Symposium*, volume 3663 of *Lecture Notes in Computer Science*, pages 216–223. Springer, September 2005.

[83] E. Frazzoli, M. A. Dahleh, and E. Feron. Real-time motion planning for agile autonomous vehicles. In *Proceedings of the 2001 American Control Conference. (Cat. No.01CH37148)*, volume 1, pages 43–49 vol.1, 2001.

[84] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. *SIGGRAPH Comput. Graph.*, 14(3):124–133, July 1980.

[85] Andre K. Gaschler. *Efficient Geometric Predicates for Integrated Task and Motion Planning*. Dissertation, Technische Universität München, Munich, Germany, 2016.

[86] J. J. Gibson. *The Perception of the Visual World*. Houghton Mifflin, Boston, 1950.

[87] Sarah F Frisken Gibson. Beyond volume rendering: visualization, haptic exploration, and physical modeling of voxel-based objects. In *Visualization in Scientific Computing'95*, pages 10–24. Springer, 1995.

[88] PROVISIO GmbH. egomo - the smart wireless visual sensor for robots, 2016.

[89] Kalin Gochev, Alla Safonova, and Maxim Likhachev. Planning with adaptive dimensionality for mobile manipulation. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 2944–2951. IEEE, 2012.

[90] M. Greenspan and N. Burtnyk. Obstacle count independent real-time collision avoidance. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 2, pages 1073–1080 vol.2, Apr 1996.

[91] C. Gregg and K. Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pages 134–144, April 2011.

[92] M. Gridseth, K. Hertkorn, and M. Jagersand. On visual servoing to improve performance of robotic grasping. In *Computer and Robot Vision (CRV), 2015 12th Conference on*, pages 245–252, June 2015.

[93] S. Hadfield and R. Bowden. Scene Particles: Unregularized Particle-Based Scene Flow Estimation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36(3):564–576, March 2014.

[94] Tianyi David Han and Tarek S Abdelrahman. Reducing branch divergence in GPU programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 3. ACM, 2011.

[95] Adam Harmat, Gil E. Jones, Kai M. Wurm, and Armin Hornung. ROS Collider Package.

[96] Mark Harris. Optimizing parallel reduction in cuda. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07. ACM/IEEE, 2007.

[97] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[98] Taosong He and Arie Kaufman. Collision detection for volumetric objects. In *Proceedings of the 8th conference on Visualization'97*, pages 27–ff. IEEE Computer Society Press, 1997.

[99] Evan Herbst, Xiaofeng Ren, and Dieter Fox. RGB-D flow: Dense 3-D motion estimation using color and depth. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 2276–2282. IEEE, 2013.

[100] M. Herman. Fast, three-dimensional, collision-free motion planning. In *Proceedings. 1986 IEEE International Conference on Robotics and Automation*, volume 3, pages 1056–1063, Apr 1986.

[101] M. Hornacek, A. Fitzgibbon, and C. Rother. SphereFlow: 6 DoF Scene Flow from RGB-D Pairs. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 3526–3533, June 2014.

[102] A. Hornung, M. Phillips, E. Gil Jones, M. Bennewitz, M. Likhachev, and S. Chitta. Navigation in three-dimensional cluttered environments for mobile manipulation. In *2012 IEEE International Conference on Robotics and Automation*, pages 423–429, May 2012.

[103] Armin Hornung. 3D Collision Avoidance for Navigation in Unstructured Environments, August 2011.

[104] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, April 2013.

[105] Chris L. Jackins and Steven L. Tanimoto. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing*, 14(3):249–270, 1980.

[106] Mark W. Jones, J. Andreas Baerentzen, and Milos Sramek. 3D Distance Fields: A Survey of Techniques and Applications. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):581–599, July 2006.

[107] D. Jung and K. K. Gupta. Octree-based hierarchical distance maps for collision detection. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 1, pages 454–459 vol.1, Apr 1996.

[108] Derek Jung and Kamal K Gupta. Octree-based hierarchical distance maps for collision detection. *Journal of Field Robotics*, 14(11):789–806, Nov 1997.

[109] Knut B Kaldestad, Sami Haddadin, Rico Belder, Geir Hovland, David Anisi, et al. Collision avoidance with potential fields based on parallel processing of 3D-point cloud data on the GPU. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 3250–3257. IEEE, 2014.

[110] Tero Karras. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 33–37. Eurographics Association, 2012.

[111] Arie Kaufman. Efficient algorithms for 3d scan-conversion of parametric curves, surfaces, and volumes. *ACM SIGGRAPH Computer Graphics*, 21(4):171–179, 1987.

[112] Arie Kaufman and Reuven Bakalash. Memory and processing architecture for 3d voxel-based imagery. *IEEE Computer Graphics and Applications*, 8(6):10–23, 1988.

[113] Lydia Kavraki, Petr Svestka, Jean claude Latombe, and Mark Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *IEEE International Conference on Robotics and Automation*, pages 566–580, 1996.

[114] W. Ke-ke, Z. Han-qing, L. Qiang, and Z. Wei. A motion planning algorithm for autonomous land vehicle based on virtual tentacles. In *2011 Chinese Control and Decision Conference (CCDC)*, pages 2431–2435, May 2011.

[115] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948 vol.4, Nov 1995.

[116] O. Khatib. Real-Time Obstacle Avoidance for Manipulators and Mobile Robots. *The International Journal of Robotics Research*, 5(1):90–98, 1986.

[117] Kourosh Khoshelham and Sander Oude Elberink. Accuracy and Resolution of Kinect Depth Data for Indoor Mapping Applications. *Sensors*, 12(2):1437–1454, 2012.

[118] J. T. Kider, M. Henderson, M. Likhachev, and A. Safonova. High-dimensional planning on the gpu. In *2010 IEEE International Conference on Robotics and Automation*, pages 2515–2522, May 2010.

[119] Young J. Kim, Gokul Varadhan, Ming C. Lin, and Dinesh Manocha. Fast swept volume approximation of complex polyhedral models. *Computer-Aided Design*, 36(11):1013 – 1027, 2004.

[120] Y. Kitamura, T. Tanaka, F. Kishino, and M. Yachida. 3-d path planning in a dynamic environment using an octree and an artificial potential field. In *Intelligent Robots and Systems 95. 'Human Robot Interaction and Cooperative Robots', Proceedings. 1995 IEEE/RSJ International Conference on*, volume 2, pages 474 –481 vol.2, August 1995.

[121] S. Koenig and M. Likhachev. Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics*, 21(3):354–363, June 2005.

[122] Petr Kotas, Roberto Croce, Valentina Poletti, Vit Vondrak, and Rolf Krause. *A Massive Parallel Fast Marching Method*, pages 311–318. Springer International Publishing, Cham, 2016.

[123] Wojciech Kowalczyk, Mateusz Przybyla, and Krzysztof Kozlowski. Set-point control of mobile robot with obstacle detection and avoidance using navigation function - experimental verification. *Journal of Intelligent & Robotic Systems*, pages 1–14, 2016.

[124] T. De Laet, S. Bellens, H. Bruyninckx, and J. De Schutter. Geometric relations between rigid bodies (part 2): From semantics to software. *IEEE Robotics Automation Magazine*, 20(2):91–102, June 2013.

[125] T. De Laet, S. Bellens, R. Smits, E. Aertbelien, H. Bruyninckx, and J. De Schutter. Geometric relations between rigid bodies (part 1): Semantics for standardization. *IEEE Robotics Automation Magazine*, 20(1):84–93, March 2013.

[126] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[127] B. Lau, C. Sprunk, and W. Burgard. Improved updating of euclidean distance maps and voronoi diagrams. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 281–286, October 2010.

[128] Boris Lau. *Techniques for Robot Navigation in Dynamic Real-world Environments*. PhD thesis, Albert-Ludwigs-Universität, Freiburg im Breisgau, Deutschland, December 2013.

[129] Boris Lau, Christoph Sprunk, and Wolfram Burgard. Incremental updates of configuration space representations for non-circular mobile robots with 2d 2.5 d or 3d obstacle models. In *European Conference on Mobile Robotics (ECMR)*, pages 49–54, 2011.

[130] Christian Lauterbach, Qi Mo, and Dinesh Manocha. gProximity: Hierarchical GPU-based operations for collision and distance queries. In *Computer Graphics Forum*, volume 29. Wiley Online Library, 2010.

[131] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998.

[132] Leng-Feng Lee. Decentralized Motion Planning Within An Artificial Potential Framework (APF) For Cooperative Payload Transport By Multi-Robot Collectives. Master's thesis, State University of New York at Buffalo, Department of Mechanical and Aerospace Engineering, Buffalo, New York, 2004.

[133] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ACM SIGARCH Computer Architecture News*, pages 451–460. ACM, 2010.

[134] F. Leymarie and M. D. Levine. Fast raster scan distance propagation on the discrete rectangular lattice. *CVGIP: Image Understanding*, 55(1):84–94, January 1992.

[135] Yi Li, Cornelia Fermüller, J. Aloimonos, and Hui Ji. Learning shift-invariant sparse representation of actions. In *2010 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2630 – 2637. IEEE, IEEE, 2010/06/13/18 2010.

[136] Maxim Likhachev and Dave Ferguson. Planning long dynamically feasible maneuvers for autonomous vehicles. *The International Journal of Robotics Research*, 28(8):933–945, 2009.

[137] Chien-Chou Lin and Jen-Hui Chuang. Potential-based path planning for robot manipulators in 3-d workspace. In *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, volume 3, pages 3353–3358 vol.3, Sept 2003.

[138] F. Lingelbach. Path planning using probabilistic cell decomposition. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 1, pages 467–472 Vol.1, April 2004.

[139] V. Lippiello, F. Ruggiero, B. Siciliano, and L. Villani. Visual grasp planning for unknown objects using a multifingered robotic hand. *IEEE/ASME Transactions on Mechatronics*, 18(3):1050–1059, June 2013.

[140] A. Mahendran and A. Vedaldi. Understanding deep image representations by inverting them. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5188–5196, June 2015.

[141] C. R. Maurer, Rensheng Qi, and V. Raghavan. A linear time algorithm for computing exact euclidean distance transforms of binary images in arbitrary dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(2):265–270, Feb 2003.

[142] W.A. McNeely, K.D. Puterbaugh, and J.J. Troy. Six Degree-of-Freedom Haptic Rendering Using Voxel Sampling. In *Proceedings of ACM SIGGRAPH 99*, pages 401–408, 1999.

[143] Donald JR Meagher. *Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer*. Electrical and Systems Engineering Department Rensseiaer Polytechnic Institute Image Processing Laboratory, 1980.

[144] Duane Merrill and Andrew Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(02):245–272, 2011.

[145] John Michalakes and Manish Vachharajani. Gpu acceleration of numerical weather prediction. *Parallel Processing Letters*, 18(04):531–548, 2008.

[146] Hans Moravec and A. E. Elfes. High resolution maps from wide angle sonar. In *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, pages 116 – 121, March 1985.

[147] M. Mori. The uncanny valley. *Energy*, 7(4):33–35, 1970.

[148] Guy M Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Company New York, 1966.

[149] Alex Nash. *Any-Angle Path Planning*. PhD thesis, UNIVERSITY OF SOUTHERN CALIFORNIA, 2012.

[150] Richard A Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*, pages 127–136. IEEE, 2011.

[151] Chuong V Nguyen, Shahram Izadi, and David Lovell. Modeling kinect sensor noise for improved 3d reconstruction and tracking. In *3D Imaging, Modeling, Processing, Visualization and Transmission (3DIMPVT), 2012 Second International Conference on*, pages 524–530. IEEE, 2012.

[152] Fakir S Nooruddin and Greg Turk. Simplification and repair of polygonal models using volumetric techniques. *Visualization and Computer Graphics, IEEE Transactions on*, 9(2):191–205, 2003.

[153] Illah Reza Nourbakhsh. *Interleaving planning and execution for autonomous robots*, volume 385. Springer Science & Business Media, 2012.

[154] NVIDIA Corporation. *CUDA C Programming Guide v8.0*, June 2017.

[155] Jon Olick. Current generation parallelism in games. In *SIGGRAPH*, volume 8, pages 1–120, 2008.

[156] Jia Pan, S. Chitta, and D. Manocha. FCL: A general purpose library for collision and proximity queries. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3859–3866, 2012.

[157] Jia Pan, Sachin Chitta, and Dinesh Manocha. Probabilistic collision detection between noisy point clouds using robust classification. In *International Symposium on Robotics Research*, Flagstaff, Arizona, August 2011.

[158] Jia Pan, Christian Lauterbach, and Dinesh Manocha. g-planner: Real-time motion planning and global navigation using GPUs. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2010.

[159] Jia Pan and Dinesh Manocha. Efficient configuration space construction and optimization for motion planning. *Engineering*, 1(1):046–057, 2015.

[160] A. A. Paranjape, K. C. Meier, X. Shi, S. J. Chung, and S. Hutchinson. Motion primitives and 3-d path planning for fast flight through a forest. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2940–2947, Nov 2013.

[161] Chonhyon Park, Jia Pan, and Dinesh Manocha. Real-time optimization-based planning in dynamic environments using GPUs. In Daniel Borrajo, Ariel Felner, Richard E. Korf, Maxim Likhachev, Carlos Linares López, Wheeler Ruml, and Nathan R. Sturtevant, editors, *SOCS*. AAAI Press, 2012.

[162] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[163] Reinhold Preiner, Stefan Jeschke, and Michael Wimmer. Auto splats: Dynamic point cloud visualization on the gpu. In H. Childs and T. Kuhlen, editors, *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, pages 139–148. Eurographics Association 2012, May 2012.

[164] M. Przybylski, T. Asfour, and R. Dillmann. Unions of balls for shape approximation in robot grasping. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 1592–1599, Oct 2010.

[165] S. Quinlan and O. Khatib. Elastic bands: connecting path planning and control. In *[1993] Proceedings IEEE International Conference on Robotics and Automation*, pages 802–807 vol.2, May 1993.

[166] Julian Quiroga, Thomas Brox, Frédéric Devernay, and James Crowley. Dense Semi-rigid Scene Flow Estimation from RGBD Images. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, volume 8695 of *Lecture Notes in Computer Science*, pages 567–582. Springer International Publishing, 2014.

[167] J. Reif and M. Sharir. Motion planning in the presence of moving obstacles. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 144–154, Oct 1985.

[168] John H Reif. Complexity of the mover's problem and generalizations. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pages 421–427. IEEE, 1979.

[169] Alexander Reinefeld and Volker Schnecke. Work-load balancing in highly parallel depth-first search. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 773–780. IEEE, 1994.

[170] Ulrich Rembold and Rüdiger Dillmann. The Control System of the Autonomous Mobile Robot KAMRO of the University of Karlsruhe. In *Intelligent Autonomous Systems 2, An International Conference*, pages 565–575, Amsterdam, The Netherlands, The Netherlands, 1989. IOS Press.

[171] Matthias Renz, Carsten Preusche, Marco Pötke, Hans-Peter Kriegel, and Gerd Hirzinger. Stable haptic interaction with virtual environments using an adapted voxmap-pointshell algorithm. In *In Proc. Eurohaptics*. Citeseer, 2001.

[172] A Roennau, G Liebel, T Schamm, T Kerscher, R Dillmann, and Interactive Diagnosis. Robust 3D Scan Segmentation for Teleoperation Tasks in Areas Contaminated by Radiation. pages 2419–2424, 2010.

[173] Guodong Rong and Tiow-Seng Tan. Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, pages 109–116, New York, NY, USA, 2006. ACM.

[174] Guodong Rong and Tiow-Seng Tan. Variants of jump flooding algorithm for computing discrete Voronoi diagrams. In *Voronoi Diagrams in Science and Engineering, 2007. ISVD'07. 4th International Symposium on*, pages 176–181. IEEE, 2007.

[175] Waldir L. Roque and Dionísio Doering. Trajectory planning for lab robots based on global vision and voronoi roadmaps. *Robotica*, 23(4):467–477, 2005.

[176] S. W. Rühl, Z. Xue, J. M. Zöllner, and R. Dillmann. Integration of a loop based and an event based framework for control of a bimanual dextrous service robot. In *2009 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 110–116, Dec 2009.

[177] M. Sagardia, T. Hulin, Preusche C., and G. Hirzinger. Improvements of the voxmap-pointshell algorithm – fast generation of haptic data structures. In *53rd Internationales Wissenschaftliches Kolloquium*, Ilmenau, Germany, 2007.

[178] M. Saha, G. Sanchez, and J.C. Latombe. Planning multi-goal tours for robot arms. In *International Conference on Robotics and Automation*, Taipei, Taiwan, 2003.

[179] Richard Satherley and Mark W. Jones. Vector-City Vector Distance Transform. *Computer Vision and Image Understanding*, 82(3):238 – 254, 2001.

[180] Johannes Schauer, Janusz Bedkowski, Karol Majek, and Andreas Nüchter. Performance comparison between state-of-the-art point-cloud based collision detection approaches on the CPU and GPU. *IFAC-PapersOnLine*, 49(30):54–59, 2016.

[181] Johannes Schauer and Andreas Nüchter. Collision detection between point clouds using an efficient k-d tree implementation. *Advanced Engineering Informatics*, 29(3):440–458, 2015.

[182] Jens Schneider, Martin Kraus, and Rüdiger Westermann. GPU-based real-time discrete euclidean distance transforms with precise error bounds. In *International Conference on Computer Vision Theory and Applications (VISAPP)*, pages 435–442, 2009.

[183] Jonathan Scholz, Sachin Chitta, Bhaskara Marthi, and Maxim Likhachev. Cart pushing with a mobile manipulation system: Towards navigation with moveable objects. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 6115–6120. IEEE, 2011.

[184] Henrik Schumann-Olsen, Marianne Bakken, $\mathcal{O}(y)$stein Hov Holhjem, and Petter Risholm. Parallel dynamic roadmaps for real-time motion planning in complex dynamic scenes. In *3rd Workshop on Robots in Clutter, IEEE*, 2014.

[185] Michael Schwarz and Hans-Peter Seidel. Fast parallel surface and solid voxelization on GPUs. *ACM Transactions on Graphics*, 29(6 (Proceedings of SIGGRAPH Asia 2010)):179:1–179:9, December 2010.

[186] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences of the United States of America*, 93(4):1591–1595, 1996.

[187] Peter Shirley and Steve Marschner. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 3rd edition, 2009.

[188] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[189] Jan Smisek, Michal Jancosek, and Tomas Pajdla. 3D with Kinect. In *Consumer Depth Cameras for Computer Vision*, pages 3–25. Springer, 2013.

[190] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. Whippletree: Task-based scheduling of dynamic workloads on the GPU. *ACM Trans. Graph.*, 33(6):228:1–228:11, November 2014.

[191] A. Stentz. Optimal and efficient path planning for partially-known environments. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pages 3310–3317 vol.4, May 1994.

[192] Jörg Stückler and Sven Behnke. Efficient Dense Rigid-Body Motion Segmentation and Estimation in RGB-D Video. *International Journal of Computer Vision*, pages 1–13, 2015.

[193] Freek Stulp, Andreas Fedrizzi, Lorenz Mösenlechner, and Michael Beetz. Learning and Reasoning with Action-Related Places for Robust Mobile Manipulation. *Journal of Artificial Intelligence Research (JAIR)*, 43:1–42, 2012.

[194] Ioan A Şucan and Lydia E Kavraki. Kinodynamic motion planning by interior-exterior cell exploration. In *Algorithmic Foundation of Robotics VIII*, pages 449–464. Springer, 2009.

[195] H Taubig, B Bauml, and Udo Frese. Real-time swept volume and distance computation for self collision detection. *Intelligent Robots and Systems*, pages 1585–1592, 2011.

[196] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[197] Carlo Tomasi and Takeo Kanade. *Detection and tracking of point features*. School of Computer Science, Carnegie Mellon Univ. Pittsburgh, 1991.

[198] N. Vahrenkamp, C. Scheurer, T. Asfour, J. Kuffner, and R. Dillmann. Adaptive motion planning for humanoid robots. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2127–2132, Sept 2008.

[199] K. M. Varadarajan and M. Vincze. Object part segmentation and classification in range images for grasping. In *Advanced Robotics (ICAR), 2011 15th International Conference on*, pages 21–27, June 2011.

[200] Sundar Vedula, Simon Baker, Peter Rander, Robert T. Collins, and Takeo Kanade. Three-Dimensional Scene Flow. In *ICCV*, pages 722–729, 1999.

[201] Yunfeng Wang and G. S. Chirikjian. A new potential field method for robot path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 2, pages 977–982 vol.2, 2000.

[202] Andreas Wedel, Clemens Rabe, Tobi Vaudrey, Thomas Brox, Uwe Franke, and Daniel Cremers. Efficient Dense Scene Flow from Sparse or Dense Stereo Data. In David A. Forsyth, Philip H. S. Torr, and Andrew Zisserman, editors, *ECCV (1)*, volume 5302 of *Lecture Notes in Computer Science*, pages 739–751. Springer, October 2008.

[203] Kai M. Wurm, Armin Hornung, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: A probabilistic, flexible, and compact 3d map representation for robotic systems. In *In Proc. of the ICRA 2010 workshop*, 2010.

[204] Zhe Xu, T Deyle, and C C Kemp. 1000 Trials: An empirically validated end effector that robustly grasps objects from the floor. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 2160–2167, 2009.

[205] Z. Xue, A. Kasper, J. M. Zoellner, and R. Dillmann. An automatic grasp planning system for service robots. In *Advanced Robotics, 2009. ICAR 2009. International Conference on*, pages 1–6, June 2009.

[206] Jianming Yang and Frederick Stern. A highly scalable massively parallel fast marching method for the eikonal equation. *Journal of Computational Physics*, 2016.

[207] Jing Yang, Patrick Dymond, and Michael Jenkin. Hierarchical probabilistic estimation of robot reachable workspace. In *Proceedings of the 6th International Conference on Informatics in Control, Automation and Robotics (ICINCO-RA)*, pages 60–66, 2009.

[208] W. Zhang, F. Sun, C. Liu, C. Gao, and W. Su. Torque control for grasping by learning experience and tactile feedback. In *2015 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 212–218, Dec 2015.

[209] Yichao Zhou and Jianyang Zeng. Massively parallel a* search on a gpu. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.