Lehrstuhl für Angewandte Mechanik
Fakultät für Maschinenwesen
Technische Universität München

Effiziente Abstandsberechnungen mit Swept-Sphere Volumen für Echtzeit Kollisionsvermeidung in der Robotik

High-Performance Distance-Evaluation with Swept-Sphere-Volumes for Realtime Collision Avoidance in Robotics

Wissenschaftliche Arbeit zur Erlangung des Grades

Bachelor of Science (B.Sc.)

an der Fakultät für Maschinenwesen der Technischen Universität München

Themenstellende/r	Prof. dr.ir. Daniel J. Rixen Lehrstuhl für Angewandte Mechanik
Betreuer/Betreuerin	Philipp Seiwald M.Sc., Moritz Sattler M.Sc. Lehrstuhl für Angewandte Mechanik
Eingereicht von	Reinhold Poscher Oberer Wegländer 6 6403 Flaurling +43 680 5030100
Eingereicht am	18. November 2020 in Garching

Abstract

This thesis deals with the development of a software module designed for realtime collision avoidance by an efficient distance-evaluation. Complex geometries are therefore represented by a combination of three simple *swept-sphere-volume* primitives. The goal is an efficient evaluation of the minimal distance between two objects exploiting SIMD and parallelization capabilities. Besides the implementation of the module, the thesis includes a performance evaluation as well as a detailed code documentation. The developed module provides a good starting point for future extensions and acceleration methods.

Zusammenfassung

Diese Arbeit befasst sich mit der Entwicklung eines Software-Moduls zur effizienten Abstandsberechnung für Echtzeit Kollisionsvermeidung. Die Darstellung komplexer Geometrien erfolgt dabei durch Verknüpfung von drei einfachen Primitiven aus *Swept-Sphere-Volumes*. Ziel ist eine effiziente Auswertung des minimalen Abstands zwischen zwei geometrischen Objekten durch Ausnützung von Vektorisierung und Parallelisierung im Berechnungsprozess. Die Arbeit umfasst neben der Implementierung des Moduls eine Evaluierung der Performance sowie eine ausführliche Codedokumentation. Das entwickelte Modul stellt eine solide Grundlage für Erweiterungen und Beschleunigungen dar.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

In der Robotik reichen die Einsatzbereiche von stationären Manipulatoren in Industrieanlagen bis zu mobilen und vielseitigen Robotern, welche sich auch in unbekannten Umgebungen bewegen können. Im Bereich der stationären Robotik können meist die Umgebung modelliert und die Bewegung vorab (offline) berechnet werden. In vielen anderen Bereichen ist dies nicht möglich. Um Zusammenstöße in dynamisch veränderlichen Umgebungen zu vermeiden sowie für die Online-Bahnplanung bei mobilen Robotern, ist eine echtzeitfähige Kollisions- bzw. Abstandsberechnung notwendig.

Dazu wurde für den humanoiden Roboter *Lola* [13] am *Lehrstuhl für Angewandte Mechanik* ein Modul zur Kollisionsvermeidung entwickelt [17]. Dieses Modul nutzt sogenannte *Swept-Sphere-Volumen* (SSV) (siehe Abb. 1.1), welche eine komplexe und rechenintensive Struktur durch drei einfache Primitive (Punkt, Linie und Dreieck) abstrahieren. Vor knapp zehn Jahren entwickelt, entspricht die Implementierung dieses Moduls nicht mehr den heutigen Anforderungen. Vorteile moderner Rechnerarchitektur werden nicht genutzt.

1.2 Problemstellung

Anwendungen in der Robotik, Bildsynthese, CAD/CAM, physikalische Simulationen in der Mehrkörperdynamik und vieles mehr, erfordern den minimalen Abstand zwischen zwei geometrischen Objekten. Bei einigen dieser Anwendungen ist diese Information maßgeblich für die volle Funktionalität. Die Abstände müssen daher zuverlässig, dennoch in sehr kurzer Zeit, berechnet werden.

Eine besondere Herausforderung stellt dabei der Bereich der Robotik, insbesondere ein humanoider Roboter wie *Lola*, dar. Hier sind Kollisionen mit der Umgebung, vor allem aber Selbstkollisionen mit verschiedenen Teilen des Roboters möglich. Diese Kollisionen können zu einer Destabilisierung und damit zu erheblichen Schäden führen.
Die Schwierigkeit besteht nun darin, ein für die Anwendung passendes Gleichgewicht zwischen Genauigkeit und Schnelligkeit zu finden. So würde eine Berechnung mit den CAD-Daten eines Roboter am besten die Realität abbilden, aber die Forderung nach Echtzeitfähigkeit nicht erfüllen können. Der hohe Detaillierungsgrad des Modells und folglich die hohe Anzahl an möglichen Kollisionspaaren würde für eine Auswertung zu lange benötigen.

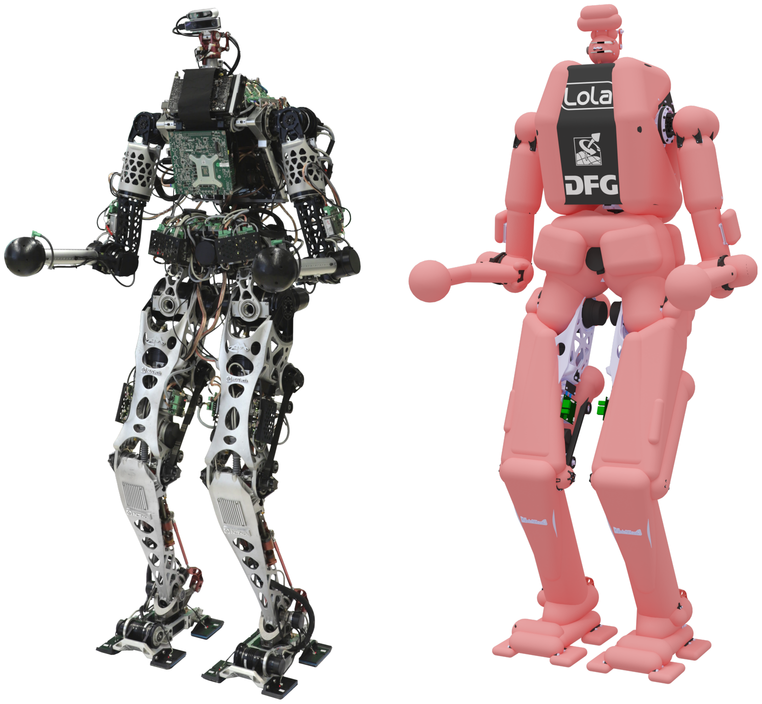


Abbildung 1.1: Links: 26 DoF humanoider Roboter *Lola* [13] (aktualisiertes Bild). Rechts: CAD und Kollisionsmodell von *Lola* basierend auf *Swept-Sphere-Volumes* [17] (aktualisiertes Modell).

Eine Vereinfachung der Geometrie bzw. eine Reduzierung der Kollisionspaare kann die Laufzeit der Berechnungen verbessern, aber zu Einschränkungen in der Bewegungsfreiheit führen.

1.3 Kollisionserkennung vs. Abstandsberechnung

Allgemein bedeutet Kollisionserkennung bzw. Kollisionsdetektion das Erkennen einer Berührung oder Überlappung von zwei (oder mehreren) geometrischen Körpern. Diese Erkennung kann im 2-, wie auch 3-dimensionalen Raum erfolgen, wobei im letzteren Fall ein wesentlich größerer Rechenaufwand notwendig ist.
Die Komplexität und Dauer einer Detektion hängen im wesentlichen von zwei Faktoren ab: der Anzahl der Objekte und der Komplexität der verwendeten Geometrien (siehe Abb. 1.2).
Gibt es in einer Berechnung mehr Objekte, so steigt auch die Anzahl der möglichen Kollisionspaare. Da theoretisch jedes Objekt mit einem anderen Objekt kollidieren kann, folgen für eine Auswertung von n Objekten $(n-1)+(n-2)+\cdots+1 = n(n-1)/2 = \mathcal{O}(n^2)$ paarweise Tests [8, Abschnitt 2.4.1]. Dies wiederum bedeutet eine längere Berechnungsdauer einer Abfrage und/oder die Einführung komplexer Berechnungshierarchien. Die Auswahl der Geometrien hingegen hat einen erheblichen Einfluss auf die Laufzeit und Komplexität der angewendeten Algorithmen.
Die Schwierigkeit besteht darin, eine geeignete Balance dieser vier Größen zu finden. Je nach Anwendung können verschiedene Aspekte mehr zum Tragen kommen. So steht in der Bildsynthese ein hoher Detaillierungsgrad der Geometrien im Vordergrund, während im Bereich der Robotik die Schnelligkeit der Berechnung Priorität hat.

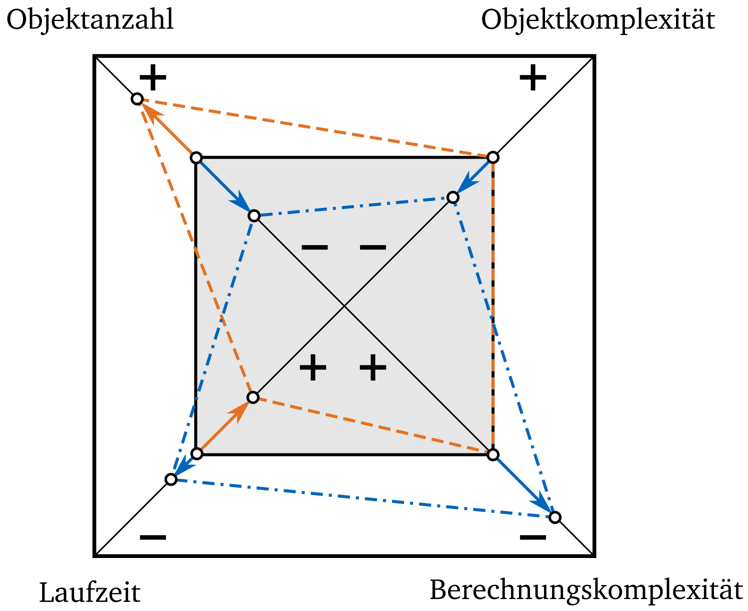


Abbildung 1.2: Eine Analogie zum *Teufelsquadrat* von Harry Sneed, bezogen auf die Kollisions- und Abstandsberechnung. Die Fläche des Vierecks bleibt immer konstant - zwei Beispiele sollen den Zusammenhang der vier Größen verdeutlichen.

Im Allgemeinen können Kollisionsberechnungen an statischen und dynamischen Objekten durchgeführt werden. Bei bewegten Objekten erfolgt eine weitere Unterscheidung der Berechnungsmethode in eine diskrete und kontinuierliche Bewegung.
Die Berechnung im diskreten Fall erfolgt anhand einer statischen Momentaufnahme des kompletten Systems, welche in diskreten Zeitschritten aktualisiert wird. Bei großen Relativgeschwindigkeiten der Objekte im Vergleich zur Zeitschrittweite besteht die Gefahr, dass auftretende Kollisionen unerkannt bleiben (siehe Abb. 1.3). Dieser Effekt einer unerkannten Kollision innerhalb eines Zeitschritts wird als *Tunneling* bezeichnet.

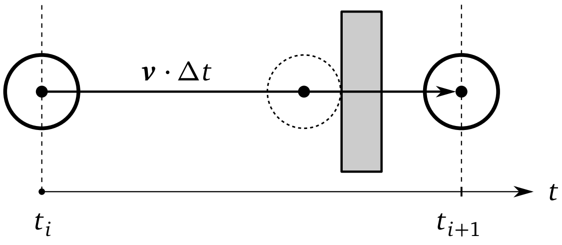


Abbildung 1.3: In Anlehnung an [8, Abb. 5.32]: Darstellung des *Tunneling*-Effekts. Die nicht detektierte Kollision innerhalb des Zeitschritts ist gestrichelt dargestellt.

Im kontinuierlichen Fall wird die Bewegung zwischen zwei diskreten Berechnungszeitpunkten zur Auswertung mit einbezogen. Es erfolgt die Erzeugung eines Volumens durch Abfahren des Objekts entlang der Bewegungstrajektorie. Dieses Volumen beschreibt die komplette Bewegung innerhalb dieses Zeitschritts, wodurch der Effekt des *Tunneling* nicht mehr auftreten kann [8, Abschnitt 5.5]. Dieser Vorteil überwiegt in den wenigsten Fällen den erheblich höheren Rechenaufwand. Die Geschwindigkeiten bzw. Zeitschritte sind meistens entsprechend klein und die Objekte groß genug, sodass eine diskrete Berechnung für den Großteil der dynamischen Anwendungen ausreichend ist.

Kollisionserkennung - Abstandsberechnung

Eine Kollisionserkennung gibt, abhängig davon ob eine Überschneidung von zwei Geometrien stattgefunden hat oder nicht, einen boolschen Wert zurück. Eine Ebene über der Detektion von Kollisionen steht die Abstands- bzw. Distanzberechnung. In der Regel erfolgt eine Berechnung der minimalen Distanz von zwei Objekten. Der größere Informationsgehalt dieser Berechnung bringt eine Vielzahl an Möglichkeiten mit sich: so liegt eine Kollision vor, wenn der minimale Abstand kleiner Null ist. Durch Betrachtung der Auswertungen von zwei oder mehreren aufeinanderfolgenden Zeitschritten, können sowohl Aussagen zur relativen Geschwindigkeit, als auch zur Bewegungsrichtung der Körper getroffen werden. Auch die genaue Lage der Punkte mit dem geringsten Abstand können abhängig vom Anwendungsfall von Interesse sein. Dieser Mehrwert an Information zum Abstand ist jedoch mit einem erhöhten Aufwand in der Berechnung verbunden.

1.4 Ziele der Arbeit

Die Hauptaufgabe besteht in der Entwicklung eines neuen Frameworks zur Distanzberechnung von Starrkörpern, welche durch eine Vereinigung von SSVs modelliert werden. Dieses Framework soll Teil der am Lehrstuhl entwickelten C++ Softwarebibliothek `broccoli` [18] werden und mit einer allgemeinen und flexiblen Struktur auch für andere Projekte in der Robotik verwendet werden können.
Dazu sollen die Hintergründe und mathematischen Grundlagen aus [17, Kap. 6] überprüft und gegebenenfalls durch zusätzliche Fälle erweitert werden. Der Fokus liegt auf einem Hochleistungsalgorithmus, unter Ausnützung von vektorisierten Rechenabläufen und Parallelisierung der Berechnungen auf mehrere Prozessorkerne für eine schnellstmögliche Abfrage. Für lineare Algebra soll als Backend die Open-Source Bibliothek *Eigen*[1] verwendet werden.
Performance Tests dienen zur Evaluierung der Effizienz des neuen Moduls. Es sollen Vergleiche verschiedener Berechnungsmethoden sowie eine Gegenüberstellung mit dem vorherigen Modul durchgeführt werden. Parallel zur Implementierung erfolgt eine vollständige und ausführliche Dokumentation des Codes mittels *Doxygen*[2]. Die Definition von verschiedensten Testfällen dient der Überprüfung auf korrekte Funktionsweise des Moduls. Die Hintergründe der Arbeit sowie Erklärungen und auftretende Probleme werden in einer abschließenden schriftlichen Arbeit zusammengefasst.

---

[1] http://eigen.tuxfamily.org

[2] https://www.doxygen.nl

Kapitel 2

Stand der Technik

Die `GAMMA`-*research-group*[1] von der *University of North Carolina* hat eine Vielzahl an Modulen zur Detektion von Kollisionen und Berechnung von Abständen entwickelt und implementiert. Eine Auflistung und Beschreibung dieser ist auf ihrer Homepage zu finden. Viele weitere Bibliotheken greifen auf Algorithmen, Strukturen oder Abläufe aus diesen Modulen zurück.

Im Folgenden werden einige Bibliotheken zur Kollisions- und Abstandsberechnung vorgestellt und auf ihre zugrundeliegenden Geometrien, Berechnungsmethoden, Vor- und Nachteile eingegangen.

2.1 Proximity Query Package PQP

Larsen et al. beschreiben in [11] einen Abstandsberechnungsalgorithmus, welcher als Teil des *Proximity Query Package* (PQP) implementiert ist. Dieser Bibliothek liegt das Konzept der Hierarchie von Begrenzungsvolumen (englisch: *Bounding Volume Hierarchies* (BVH)) zugrunde. Als Volumen werden *Rectangle-Swept-Spheres* (RSS) verwendet. Sie beschreiben einen Punktesatz, welcher durch Abfahren des Zentrums einer Kugel über ein dreidimensionales Viereck erzeugt wird, d.h. der Minkowksisumme einer ursprungs-zentrierten Kugel und einem willkürlich orientierten Viereck entspricht.

Ein Begrenzungsvolumen (BV) wird verwendet, um geometrische Primitive, wie Dreiecke, Polygone, NURBs[2], etc. zu umschließen. In einer BVH befinden sich die BVs an den internen Knoten der Baumstruktur (siehe Abb. 2.1). Die Wurzel enthält das gesamte Modell - die Kind-BVs jeweils separate Teile, welche in den Eltern-BVs zusammengefasst sind. Die Blätter des Baumes stellen die einzelnen Primitive dar.

Die Variable ϵ beschreibt den geringsten Abstand zweier Objekte. Zu Beginn einer Abfrage wird ϵ auf Unendlich oder einen Abstand von zwei willkürlich ausgewählten Primitiven gesetzt. Bei jedem Aufruf wird zwischen zwei BVs A und B bestimmt, ob der Abstand eines Primitivs aus A und eines aus B kleiner ϵ ist. In diesem Fall wird ϵ überschrieben. Wenn der Abstand der BVs A und B bereits größer ϵ ist, so wird dieser Fall nicht mehr genauer untersucht. Andernfalls wird der Algorithmus rekursiv auf die Kinder-BVs angewendet. An den Blatt-Knoten wird der exakte Abstand zwischen den Primitiven berechnet. Ist dieser kleiner ϵ, übernimmt ϵ diesen Wert.

---

[1] http://gamma.web.unc.edu/research/collision/packages/

[2] *Non-Uniform Rational B-Spline* - mathematisch definierte Kurven oder Flächen; dienen zur Modellierung beliebiger Formen.

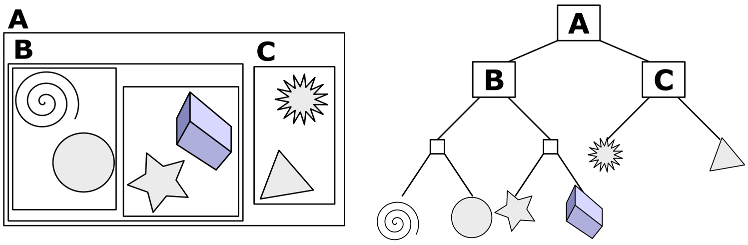


Abbildung 2.1: Hierarchische Baumstruktur von Begrenzungsvolumen [16].

Für die Berechnung der Abstände der Primitive, hier Dreiecke, werden alle neun möglichen Punktepaarungen ausgewertet. Wenn sich die beiden Dreiecke nicht schneiden, so entspricht das nächste Punktepaar dem geringsten Abstand.

Da es sich bei den RSSs um konvexe Geometrien handelt, können die Distanzen mit Algorithmen wie *GJK* [9], *Lin Canny* [12] o.ä. berechnet werden. Für bessere Effizienz und Genauigkeit wird hier ein eigens implementierter Algorithmus verwendet. Im Vergleich zum *GJK* weist dieser eine ca. viermal schnellere Berechnung auf.

Aufgrund mehrerer möglicher numerischer Fehlerquellen kann es bei sehr geringen Abständen von zwei BVs zu einem größeren Ergebnis wie erwartet kommen. Durch Verwendung von Toleranzen während der Berechnung können solche Fehler detektiert werden. Der fehlgeschlagene Test wird wiederholt.

2.2 Rapid and Accurate Polygon Interference Detection RAPID

Gottschalk et al. beschreiben in [10] eine Datenstruktur und einen Algorithmus für eine effiziente Erkennung von Überschneidungen. Die Software zur Kollisionserkennung wurde in C++ implementiert und ist als Paket *RAPID* erhältlich. Die zu berechnende Geometrie wird durch eine hierarchisch strukturierte Anordnung von orientierten Begrenzungsboxen (englisch: *Oriented Bounding Boxes* (OBB)) repräsentiert. Sie umschließen einen Teil der Gesamtgeometrie. Die Orientierung sowie Maße einer OBB werden so gewählt, dass sie der ursprünglichen Geometrie engstmöglich anliegt. Alternative Formen von Begrenzungsvolumen um Geometrien darzustellen sind Kugeln, achsen-ausgerichtete Begrenzungsboxen (*Axis-Aligned Bounding Boxes* (AABB)), k-Discretely Oriented Polytopes (*k-DOPs*), konvexe Hüllen, Ellipsen, etc. (siehe Abb. 2.2).

Der Vorteil der Kugeln und AABBs liegt in der einfachen geometrischen Beschreibung und der dadurch wesentlich simpleren Berechnung von Kollisionen zwischen zwei Volumen. Ein Problem stellen aber lange und dünne Polygone dar. Diese können von ihnen nur sehr schlecht umhüllt werden, bzw. wird eine große Anzahl an Volumen für eine passende Abbildung benötigt. Viele Volumen wiederum bedeuten einen erheblichen Mehraufwand in der Detektion von Kollisionen aufgrund der großen Anzahl an möglichen Kollisionspaaren.
Bei k-DOPs, konvexen Hüllen und Ellipsen verhält es sich umgekehrt. Eine Geometrie kann sehr passgenau und mit wenigen Volumen beschrieben werden. Dafür sind die Berechnungen um einiges komplexer und zeitaufwendiger. OBBs stellen in Bezug auf die Anzahl der BVs zur Abbildung und dem Umfang der Berechnungen einen guten Kompromiss dar.

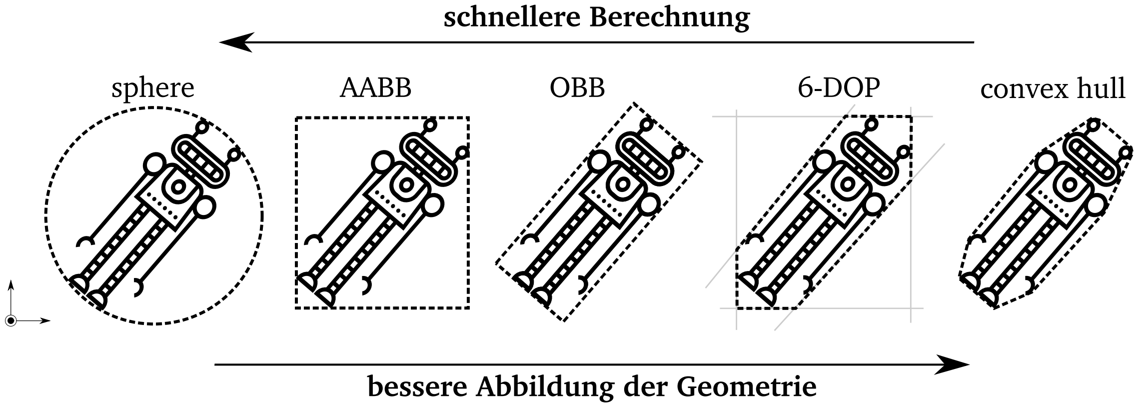


Abbildung 2.2: In Anlehnung an [8, Abb. 4.2]: Unterschiede der BVs *sphere*, *axis-aligned bounding box*, *oriented bounding box*, *k-DOPs* und *convex hulls*.

Ähnlich wie im *PQP* (Abschnitt 2.1) wird auch hier die Geometrie anhand einer hierarchischen Anordnung, hier dem sogenannten *OBBTree*, beschrieben. Ein Knoten dieses Baums besteht aus einer OBB. Diese beinhaltet die Daten wie Größe und räumlichen Lage der Bounding-Box, aber auch Verweise auf angrenzende Knoten.
Andere weit verbreitete hierarchische Strukturen wie *Octrees*, *Sphere Trees*, etc. [15] eignen sich sehr gut, wenn zwei weit entfernte Objekte getestet werden. Für den Fall dass diese sehr nah sind und eventuell mehrere Überschneidungen vorliegen, verwenden die zuvor genannten Strukturen spezielle Unterteilungstechniken oder überprüfen eine große Anzahl an Begrenzungsvolumen. Die Leistung wird dadurch erheblich verschlechtert.

Der hierfür neu entwickelte Algorithmus *Fast Overlap Test* projiziert die Boxen auf eine Achse (nicht zwingend eine Koordinatenachse) im Raum. Die Projektionen entsprechen einem Intervall auf der Projektionsachse. Wenn sich diese Intervalle überlappen, so wird mit einer andere Achse der Test fortgeführt. Falls zwei Intervalle disjunkt sind, so wird der Algorithmus verlassen. Für den Fall dass sich zwei BVs schneiden, müssen insgesamt 15 verschiedene Achsen getestet und bis zu 200 Rechenoperationen durchgeführt werden.

2.3 Software Library for Interference Detection SOLID

Bergen beschreibt in [2] und [3] ein Schema für eine Kollisionserkennung von komplexen Modellen, welche sowohl Starrkörperbewegungen ausführen als auch deformiert werden können. Dieses Modul wurde in der C++ Bibliothek *SOLID* implementiert.

Wie *RAPID* (Abschnitt 2.2) mit dem OBBTree, verwendet auch *SOLID* eine hierarchische Modellrepräsentation [10]. Anstatt der *Oriented Bounding Boxes* OBBs werden hier *Axis Aligned Bounding Boxes* AABBs verwendet. Trotz der bereits diskutierten Nachteile der AABBs gegenüber den OBBs gibt es Gründe für eine Verwendung. Hauptmerkmal der AABBs ist die Ausrichtung entsprechend den lokalen Koordinaten des Modells. Während ein Objekt eines OBBs 15 Werte zu korrekten Beschreibung benötigt ($3x3$ Matrix für die Orientierung, $3x1$ Vektor für die Position und drei Werte für die Ausdehnung), sind zur vollständigen Beschreibung einer AABB nur sechs Werte notwendig (für Position und Ausdehnung).
Die Erstellung des AABBTrees und die Kollisionsberechnung sind wesentlich einfacher und benötigen nur 30% der Zeit als bei einem OBBTree. Jedoch ist durch die immer vorgegebene Ausrichtung der AABBs die Anpassung an das Modell wesentlich schwieriger (siehe Abb. 2.2). Dieser Umstand hat eine höhere Anzahl an Überlappungen zur Folge, was bei einer Kollisions-

abfrage wiederum einen erhöhten Rechenaufwand bedeutet. Um diesen dennoch so gering wie möglich zu halten, wird bei der Unterteilung der Boxen auf eine möglichst quadratische Form geachtet.

Zum Testen, ob sich zwei Modelle mit unterschiedlichen Koordinatensystemen überlappen, wird der *Fast Overlap Test* aus Abschnitt 2.2 angewendet (im Weiteren *Separate Axis Test* (SAT) genannt). Er überprüft anhand von 15 verschiedenen Achsen, ob anhand einer Achse diese beiden Modelle getrennt werden. Versuche zeigen, dass bereits mit nur sechs Achsen ein zu 94% korrektes Ergebnis erzielt werden kann. Da AABBs von Grund auf mehr Überlappungen erzeugen als eigentlich vorhanden sind und somit mehr Tests anfallen, verwendet *SOLID* den sogenannten *SATlite*-Test, welcher nur mit sechs Achsen auf Überlappungen testet. So kann die Berechnungszeit signifikant gesenkt werden.

Der größte Vorteil von AABB-Strukturen liegt in der einfachen Handhabung für deformierbare Modelle. In diesem Zusammenhang wird unter deformierbar verstanden, dass sich die Form sowie Lage innerhalb des lokalen Koordinatensystems des Modells verändern kann. Bei einer solchen Deformierung erfolgt die Aktualisierung des Baumes von den Blättern weg hin zur Wurzel. Dieser Vorgang ist bis zu 10x schneller im Vergleich zu einem Neuaufbau der BVH-Struktur. Nachteilig kann sich aber die mitunter höhere Überlappung der Boxen auswirken.

2.4 Optimized Collision Detection OPCODE

Bei *OPCODE* [20] handelt es sich um eine Bibliothek zur reinen Kollisionserkennung, welche viele Gemeinsamkeiten zu den bereits bekannten Bibliotheken *SOLID* (Abschnitt 2.3) und *RAPID* (Abschnitt 2.2) hat. Sie nutzt auch die BVH Struktur, um Kollisionen von nicht-konvexen Geometrien detektieren zu können.

Ein großer Nachteil solcher Hierarchien ist aber der große Speicherbedarf. Ein kompletter BV-Baum besteht aus $2N-1$ Knoten, wobei N die Anzahl der Primitive beschreibt. Die Größe eines einzelnen Knotens ist hauptsächlich vom verwendeten BV abhängig. So benötigt eine Kugel als BV weniger Speicherplatz als ein OBB. Des Weiteren beinhaltet jeder Knoten Zeiger zu den jeweiligen Eltern- oder Kindelementen. Durch Anordnung der Knoten in einem zusammenhängenden Speicher können Zeiger eingespart werden. Auch die Verwendung von Quaternionen anstatt Transformationsmatrizen reduziert den Speicherbedarf.

In *OPCODE* werden zwei weitere Speicheroptimierungen angewendet:

Weglassen der Blattknoten

In einem kompletten Baum beinhalten die Blattknoten die Primitive sowie ein BV zu diesen. Bei einer Kollisionsabfrage zwischen zwei Blattknoten werden zuerst die jeweiligen BV getestet, falls sich diese überlappen, folgen die Primitive. Es wird ein Test eingeführt, welcher Kollisionen zwischen BVs und Primitiven detektieren kann. Somit wird kein BV mehr in den Blattknoten benötigt und das Primitiv kann in die Elternknoten verschoben werden. Die Größe des Baumes reduziert sich somit auf $N-1$ Knoten. Des Weiteren ist ein BV-Primitiv Test genauer und stoppt somit die Anfrage früher. Es entfallen auch die Berechnungen zwischen zwei BVs von Blattknoten.

Komprimierung des Baumes

Durch Quantisierung von Gleitkommavariablen zu 16-bit Integervariablen wird die Größe jedes BV reduziert. Dieser Vorgang mit der späteren Rücktransformation kann zu falschen Werten führen. So kann es vorkommen, dass das Volumen des BVs kleiner als ursprünglich berechnet ist. Damit trotz dieser Vereinfachung ein sicheres Ergebnis gewährleistet werden kann, wurde bei der Implementierung darauf geachtet, dass bei der Rücktransformation im Zweifelsfall immer das größere Ergebnis übernommen wird.

Durch Kombination dieser beiden zusätzlichen Optimierungen kann die Größe eines Strukturbaumes signifikant verringert werden. So ist der Speicherbedarf bis zu 7x geringer als bei *RAPID*. Trotzdem werden Kollisionsabfragen in der selben Zeit, mitunter auch schneller, durchgeführt.

2.5 Speedy Walking via Improved Feature Testing SWIFT

Ehmann und Lin stellen in [6] einen beschleunigten Algorithmus zur Abstandsberechnung zwischen bewegten konvexen Polyedern vor. Dieser Algorithmus ist in der Lage Überschneidungen zu erkennen, Distanzabschätzungen oder die genaue Distanz zu berechnen und Kontaktpunkte zu bestimmen. Er arbeitet mit einer hierarchisch angelegten Struktur mit verschieden hohen Detaillierungsgraden (englisch: *Level-of-Detail* (LOD)). Es wird der Vorteil der sogenannten *multiresolution representation* ausgenutzt, welcher auch in Render-Programmen Verwendung findet. Das rechenintensive Originalmodell wird sukzessive vereinfacht, indem jede Stufe von der vorherigen abgeleitet wird.

Es wird bereits im Voraus zu einem gegebenen konvexen Polyeder eine Reihe an verknüpften LOD-Repräsentationen erstellt. Die Hierarchie selbst besteht aus einer Sequenz an konvexen Polyedern P_0, P_1, P_2, ..., P_k. Dabei stellt P_0 das Input-Polyeder mit der höchsten Detaillierung dar. Jedes weitere Polyeder P_i setzt sich aus Teilen der Eigenschaften des Vorgängers P_{i-1} zusammen und hat die Bedingung P_i zu umschließen.

Die Erzeugung eines vereinfachten Polyeders P_i aus P_{i-1} erfolgt dabei in sechs Schritten:

1. Erzeugung der Vereinfachung P_i durch Entfernen von Eckpunkten von P_{i-1}.
2. Mache P_i konvex durch Berechnung seiner konvexen Hülle.
3. Berechne den Massenschwerpunkt von P_i und verschiebe P_i so, dass der Schwerpunkt mit dem von P_{i-1} zusammenfällt.
4. Skaliere P_i ausgehend vom Schwerpunkt, sodass er P_{i-1} umschließt.
5. Berechne die maximale Abweichung von P_i zu P_0.
6. Verknüpfe die Eigenschaften (Ecken und Kanten) von P_i mit P_{i-1} und umgekehrt.

Vorab wird festgelegt, um welchen Faktor die Eigenschaften eines Polygons je Abstraktionsschritt reduziert werden. Die Konstruktion der Hierarchie wird gestoppt, wenn eine minimale Anzahl an Dreiecken, welche das Polyeder beschreiben, erreicht wird.

Die Basis zur Berechnung der Abstände oder zur Detektion ist der Algorithmus *Voronoi-Marching*, basierend auf *Lin Canny* [12]. Je nachdem, welche Information von einer Abfrage gewünscht ist, unterscheiden sich die Vorgehensweisen im Detail. Beginnend bei einem Paar aus vereinfachten Polyedern P_i wird geprüft ob eine Überschneidung vorliegt. Ist dies der Fall wird das selbe Paar anhand der nächst feineren Auflösung P_{i-1} untersucht. Wird ein Level erreicht, auf welchem sich die Objekte nicht mehr schneiden, so erfolgt die Rückgabe, dass

diese Objekte disjunkt sind. Im Fall einer Distanzabfrage wird auf der detailliertesten Ebene der beiden Objekte die Entfernung berechnet. Falls nur eine Abschätzung benötigt wird, kann auf dem bestehenden LOD der Abstand berechnet werden. Dieser kann aber den maximalen Fehler (vorab in Punkt 5 berechnet) beinhalten.
Die Schnelligkeit und Genauigkeit der Berechnungen ist stark abhängig von dem gewählten Faktor der Reduzierung und der minimal zugelassenen Größe der Polyeder.

2.6 Speedy Walking via Improved Feature Testing for Non-Convex Objects SWIFT++

Eine Weiterentwicklung stellt die ebenfalls von Ehmann und Lin entwickelte Bibliothek *SWIFT++* [7] dar. Es können Informationen wie auch in *SWIFT* abgefragt werden. Zusätzlich kann sie Auskunft geben, ob der Abstand zweier Polyeder kleiner einer vorgegebenen Toleranz ist und gibt einem Paare von Ecken oder Kanten zurück, welche innerhalb einer ebenfalls zuvor definierten Toleranz liegen.

Die Arbeitsweise der Bibliothek kann in drei Phasen eingeteilt werden:

1. Zerlegung eine Polyeders in konvexe Flächen.
2. Erzeugung einer Hierarchie mit den Flächen aus Punkt 1.
3. Ein Paar von konvexen Flächen wird mit einem modifizierten *Lin-Canny* Algorithmus [12] untersucht.

Die ersten zwei Punkte werden im Voraus berechnet, Punkt 3 wird bei einer Abfrage ausgewertet.

Im Gegensatz zu *SWIFT* (Abschnitt 2.5) ist diese Bibliothek in der Lage, Berechnungen auch mit allgemeinen, nicht-konvexen Geometrien durchzuführen. Statt der Vereinfachung durch geschlossene konvexe Polyeder, wird hier die Oberfläche eines allgemeinen Polyeders in eine Gruppe an kleinen konvexen Stücken zerlegt [4]. Bei den kleinsten Objekten handelt es sich hierbei nicht um Volumen, sondern um Flächen. Während der Zerlegung wird eine Begrenzungsvolumen-Hierarchie mit den konvexen Flächen in den Blattknoten erstellt. Die internen Knoten der BVH stellen konvexe geschlossene Polyeder dar (vergleiche Abb. 2.1).

Der Kern der Bibliothek *SWIFT* wird auch hier zur Erstellung der Begrenzungsvolumen verwendet. Im Gegensatz zu anderen Bibliotheken welche auch mit BVHs arbeiten, hat die *SWIFT++* Hierarchie eine geringe Tiefe. Trotzdem kann die Geometrie sehr genau modelliert werden. Aufgrund der Zerlegung allgemeiner Polyeder in konvexe Flächen sind Berechnungen ohne Einschränkungen der Geometrie möglich. Durch die Struktur der BVH sind Abfragen/Berechnungen nur von Paaren konvexer Polyeder/Flächen notwendig.

2.7 Kinematic Continuous Collision Detection Library KCCD

Die von Täubig et al. entwickelte *Kinematic Continuous Collision Detection Library* (KCCD) [19] ist eine C++ Bibliothek für kontinuierliche echtzeitfähige Kollisionserkennung für humanoide und industrielle Roboter. Diese verwendet zur Vereinfachung der Geometrie *Sphere-Swept-Convex-Hulls* (SSCH). Eine solche SSCH wird durch einen Punktesatz und einem Radius definiert. Anhand dieser lassen sich besonders statische Volumen, wie z. B. ein Körperteil eines Roboters, gut nachbilden.

Eine Konfiguration eines Roboters besteht aus zwei Modellen. Einem kinematischen, welche die Lage und Art der Gelenke beschreibt und einem geometrischen mit den einzelnen Starrkörpern. Ein Starrkörper wird dabei durch jeweils eine SSCH beschrieben. Zu jedem Berechnungszyklus wird anhand dieser Daten und den Sensordaten des Roboters die räumliche Lage der einzelnen Körper aktualisiert.
Die Bibliothek stellt folgende Berechnungsergebnisse zur Verfügung: Kollision findet statt oder nicht, die beiden Körper mit dem geringsten Abstand und die Abstände von allen Körperpaarungen. Des Weiteren unterstützt sie die Erstellung eines Kollisionsmodells durch geeignete SSCHs anhand eines *Open Inventor*-Modells.

Das Hauptmerkmal dieser Bibliothek besteht in der Berechnung mit einem Intervall je Gelenk (z.B. Winkelintervall) und nicht mit einer fixen Konfiguration. Bei jedem Berechnungszyklus werden für alle Gelenke anhand der derzeitigen Stellung und Geschwindigkeit ein Intervall berechnet. Innerhalb dieses Intervalls ist es möglich, dass der Roboter, im Fall eines Nothalts, zum Stillstand kommen kann. Es werden für alle Kombinationen innerhalb dieses Intervalls mögliche Kollisionen berechnet.

Ablauf einer Kollisionsabfrage:

1. Berechne für alle Gelenke die Intervalle, sodass für den Fall eines Nothalts der Roboter innerhalb dieses Intervalls zum Stehen kommt.
2. Berechne die SSCH von allen Körpern im jeweiligen Koordinatensystem.
3. Berechne die Distanz für jede Kollisionspaarung.
4. Stoppe den Roboter wenn eine der Distanzen Null ist.

Nicht konvexe Geometrien können durch mehrere Volumen repräsentiert werden; ein SSCH ist immer konvex. Zur Berechnung der Abstände verwendet die KCCD-Bibliothek einen auf *GJK* [9] basierenden Algorithmus.

Ein interessantes Feature dieser Bibliothek ist auch der veränderliche Radius der Volumen zur Laufzeit. Bei einer höheren Geschwindigkeit der Gelenke des Roboters werden die Radien der zugehörigen Starrkörper vergrößert, da auch die Dauer der Bewegung bis zum Stillstand mit steigender Geschwindigkeit zunimmt.

2.8 Flexible Collision Library FCL

Flexible Collision Library (FCL) ist eine C++ Bibliothek, entwickelt von Jia Pan et al. [14], für Kollisions- und Abstandsberechnungen. Basierend auf einer hierarchischen Struktur können unterschiedliche Modellrepräsentationen/-geometrien verwendet werden. Die Bibliothek bietet die Möglichkeit für diskrete und kontinuierliche Kollisionserkennung, Abstandberechnungen von nicht überschneidenden Objekten und Abschätzungen zur Eindringtiefe zweier Körper. Die Auswertungen können anhand starrer sowie verformbarer Körper, Gelenksmodellen und auch Punktewolken durchgeführt werden.

FCL zeichnet sich neben seiner einfachen Bedienung besonders durch die Möglichkeit aus, neue Geometrien hinzuzufügen und neue Berechnungsalgorithmen in die bestehende Bibliothek einzubauen. Eine weitere Besonderheit ist die Kollisionsberechnung mit Dreiecksnetzen oder Punktewolken, welche anhand von Kameras oder LIDAR[3]-Sensoren aufgenommen werden. Bei diesen Berechnungen gibt es jedoch bei der Genauigkeit, als auch bei der Laufzeit Potenzial für Verbesserungen.

Zur Detektion von Kollisionen wird der *GJK*-Algorithmus [9] und für Berechnungen des Abstands der von *GJK* abgeleitete *EPA* (Expanding Polytope Algorithm) verwendet. Die Kollisionssowie Abstandsberechnungen wurden so implementiert, dass eine parallelisierte Berechnung auf Multicore-CPUs und Manycore[4]-GPUs durchgeführt werden kann.

Der Aufbau der Bibliothek besteht im wesentlichen aus drei Hauptkomponenten:

1. Objektrepräsentation: Die Objekte werden durch eine für die spezifische Abfrage passende, hierarchische Struktur dargestellt. Z. B. einfache geometrische Körper wie Kegel, Zylinder, Kugeln, usw. werden mit einer *Single-Level*-Hierarchie dargestellt, bei welcher das Bounding Volume einer Geometrie einen Knoten beschreibt. Bei komplexen geometrischen Objekten erfolgt die Darstellung anhand einer Bounding Volume Hierarchie.
2. „Traversierte“ Knoteninitialisierung: Ein sogenannter Traversierungsknoten ist eine Struktur zur Speicherung aller nötigen Informationen, die zur Ausführung einer gewünschten Abfrage benötigt werden. Dafür müssen, je nach Abfragewunsch, unterschiedliche Daten in den Knoten hinterlegt werden. Auch die dazugehörige Strategie zur Traversierung wird in den Knoten hinterlegt.
3. Hierarchie-Traversierung: Anhand der zuvor initialisierten Knoten erfolgt eine Traversierung der aufgebauten Hierarchie zur Bearbeitung einer Kollisions- bzw. Abstandsabfrage.

FCL kann als Standalone-Bibliothek verwendet werden. Software-Pakete von ROS[5] wie z. B. *MoveIt* greifen bei der Kollisionsberechnung auf *FCL* als Backend zurück.

2.9 Bullet Physics Library

Bullet ist eine Open-Source C++ Bibliothek von Erwin Coumans et al. [5], welche vorwiegend für Computerspiele, Visual-Effects und Robotersimulationen konzipiert wurde. Sie beinhaltet neben Anwendungen der Starrkörperdynamik sowie der Dynamik von verformbaren Körpern die Möglichkeit einer diskreten wie auch kontinuierlichen Kollisionserkennung. Der modulare Aufbau der Bibliothek ermöglicht die Verwendung einzelner Teilbibliotheken, wie z.B. die der Kollisionserkennung.

Die Bibliothek stellt Algorithmen und Beschleunigungsmethoden zur Detektion einer Kollision sowie zur Berechnung der Punkte wo der Kontakt auftritt, zur Verfügung. In der Vorphase einer Auswertung werden anhand einer Beschleunigungsstruktur Objektpaare zur weiteren Berechnung ausgewählt. Dabei werden jene Paare verwendet, deren AABBs sich überlappen. Der sogenannte *Collision Dispatcher* iteriert im Anschluss durch alle ausgewählten Paare. Abhängig von den Objekttypen je Paar wird mit dem passenden Algorithmus die Kollisionsberechnung durchgeführt.

Bullet enthält eine große Auswahlmöglichkeit an geometrischen Körpern. Durch Zusammensetzen von konvexen Primitiven wie Quader, Kugeln, Zylinder, Kegel, usw. können nichtkonvexe Geometrien anhand dieser einfachen Formen beschrieben werden. Auch Berechnungen mit konvexen Dreiecksnetzen können durchgeführt werden. Des Weiteren besteht

---

[3] *Light Detection and Ranging* - Methode zur optischen Abstand- und Geschwindigkeitsmessung.

[4] Spezielle Multicore-Prozessoren, welche für ein hohes Maß an Parallelverarbeitungen ausgelegt sind.

[5] *Robot Operating System* - Open-Source Framework zur Entwicklung von Roboter Software.

für den Benutzer die Möglichkeit weitere Formen zur Modellabstrahierung hinzuzufügen, aber auch Algorithmen zu ergänzen.

Zur Steigerung der Performance und Verlässlichkeit der Kollisionserkennung verwendet *Bullet* eine Toleranz. Diese beträgt standardmäßig 0.04 m und fließt, abhängig vom Objekttypus, auf verschiedene Weisen in die Berechnung ein. Mit *PyBullet* wird neben *Bullet* eine schnelles und einfaches Phyton Modul zur Verfügung gestellt, welches für Simulationen in der Robotik und Maschine Learning angewendet werden kann. Der Fokus liegt hierbei verstärkt auf dem Transfer von der Simulation in die Realität.

Kapitel 3

Bisherige Lösung

Derzeit wird beim humanoiden Roboter *Lola* für Distanzberechnungen ein Modul verwendet, welches fest in die restliche C++ Softwarebasis *am2b* (Angewandte Mechanik 2-Beiner) zur Steuerung des Roboters eingebunden ist. Das Projekt *am2b* wurde parallel mit dem Roboter *Lola* mitentwickelt. Es stellt die komplette softwareseitige Umgebung, die für den Betrieb des Roboters notwendig ist, zur Verfügung. Angefangen von Simulationen, über Schnittstellen zur Kommunikation mit Hardwarekomponenten wie Mikrocontrollern oder Servoantrieben bis zu Modulen zur Stabilisierung und Kraftregelung. Im Gegensatz zu `broccoli` ist *am2b* keine Header-Only Bibliothek, sondern besteht aus dezidierten Anwendungen und dynamisch gelinkten Modulen.

3.1 Aufbau des SSV-Moduls

Das Modul arbeitet mit *Swept-Sphere-Volumens*. Die mathematischen Grundlagen der Berechnungen sowie der Aufbau der verschiedenen Elemente werden in [17, Kapitel 6] erklärt. Die Neuimplementierung des Moduls in `broccoli` baut zum Großteil auf denselben Grundlagen auf. Es wird deshalb auf das nachfolgende Kapitel 4 verwiesen, in welchem die Inhalte zusammengefasst und im Detail beschrieben sind.
Die Implementierung der Geometrien erfolgt auf zwei Ebenen, begonnen mit den drei einzelnen *Elementen* (Punkt, Linie und Dreieck). Die Zusammenfassung mehrerer Elemente beschreibt die zweite Ebene der *Segmente*. Des Weiteren wurden separate Klassen zur Angabe von Kollisionspaaren, zur Distanzberechnung sowie für die Ergebnisse dieser Berechnung angelegt.
Zusätzlich zur Beschreibung des Roboters mittels SSVs besitzt dieses Modul *Oriented Bounding Boxes*, welche die einzelnen Segmente enganliegend umhüllen. Anhand dieser Boxen wird vor der eigentlichen Distanzberechnung überprüft, ob sich zwei dieser Boxen schneiden. Tritt dieser Fall ein, so erfolgt im weiteren Verlauf eine exakte Auswertung der Distanz zwischen den beiden Segmenten. Die Herangehensweise der Vorabreduzierung möglicher Kollisionspaare sollte den Gesamtaufwand eines Berechnungszyklus senken.

Für einen Großteil der Berechnungen wird die Bibliothek *matvec* verwendet. Sie verfügt über alle grundlegenden Funktionen der linearen Algebra. *matvec* ist ebenfalls Teil von *am2b* und findet ihren Ursprung Anfang der 90er-Jahre ebenfalls am *Lehrstuhl für Angewandte Mechanik* durch Rossmann et al. 2009 erfolgten die letzten großen Änderungen für bessere Performance.

Die Bibliothek wurde mit Befehlen zur Vektorisierung (siehe Abschnitt 5.2.1) des SSE3[1]-Standards implementiert. Trotz des mittlerweile wesentlich aktuelleren AVX2[2]-Standards wäre trotzdem eine nahezu gleich effiziente Berechnung möglich. Aufgrund der Abwärtskompatibilität von SIMD-Befehlssätzen des Compilers besteht nämlich die Möglichkeit einer Optimierung des alten Standards.

3.2 Evaluierung der bisherigen Lösung

Das Modul zur Distanzberechnung wurde im Zuge der Dissertation von Schwienbacher [17] im Juli 2010 zu *am2b* hinzugefügt. Mit der Weiterentwicklung des Roboters wurde fortwährend eine Vielzahl an Adaptierungen und Veränderungen am Code vorgenommen. Dies hat zur Folge, dass im Quellcode teils mehrere verschiedene Varianten einer Funktion existieren, Code teilweise auskommentiert wurde oder über Präprozessor-Flags[3] zwischen verschiedenen Implementierungen gewechselt werden kann. Viele dieser Änderungen wurden nicht kommentiert - Erklärungen oder Hintergründe sind nicht angegeben.
Die Benennung von Variablen oder Funktionen entspricht nicht mehr den heute üblichen Standards für *Clean Code*[4]. Es sind beschreibende Funktions- und Variablennamen aus ganzen Wörtern vorzuziehen. Sie erleichtern die Lesbarkeit des Codes und machen Kommentare zur Beschreibung des Objekts oft überflüssig. Eine Dokumentation wurde erst in den letzten Jahren bei neuen Modulen in *am2b* erstellt, das Distanzberechnungsmodul besitzt keine.

Aus den genannten Gründen ist die Erweiterbarkeit und Nachvollziehbarkeit innerhalb des SSV-Moduls von *am2b* leider nicht mehr gegeben. Das neue Modul sollte zudem universell für Anwendungen in der Robotik einsetzbar sein. Aus diesen Gründen war ein Refactoring und eine Überarbeitung des Moduls notwendig.

---

[1] Streaming SIMD Extensions

[2] Advanced Vector Extensions

[3] Merker für eine bedingte Kompilierung des Codes.

[4] https://webkit.org/code-style-guidelines

Kapitel 4

Aufbau & Funktionsweise

Im folgenden Kapitel erfolgt die Beschreibung der Grundlagen des neuen Moduls. Es wird zu Anfang die Wahl und der Aufbau der Geometrien erklärt. Im Anschluss erfolgt eine Zusammenfassung der mathematischen Hintergründe zur Berechnung der Distanzen zwischen den verschiedenen Elementen. Am Ende folgen die den Elementen übergeordneten Segmente und Szenen (siehe Abb. 4.1).

Literaturverweis

Die Abschnitte 4.2 und 4.3 sind stark an [17, Kap. 6] angelehnt. Tabellen, Grafiken und Herleitungen sind teilweise übernommen oder adaptiert worden. Sie werden hier angeführt, da sie für ein grundlegendes Verständnis der implementierten Algorithmen notwendig sind.
Aufbauend auf diesen Grundlagen gibt es aufgrund neuer Erkenntnisse Ergänzungen sowie Abänderungen.

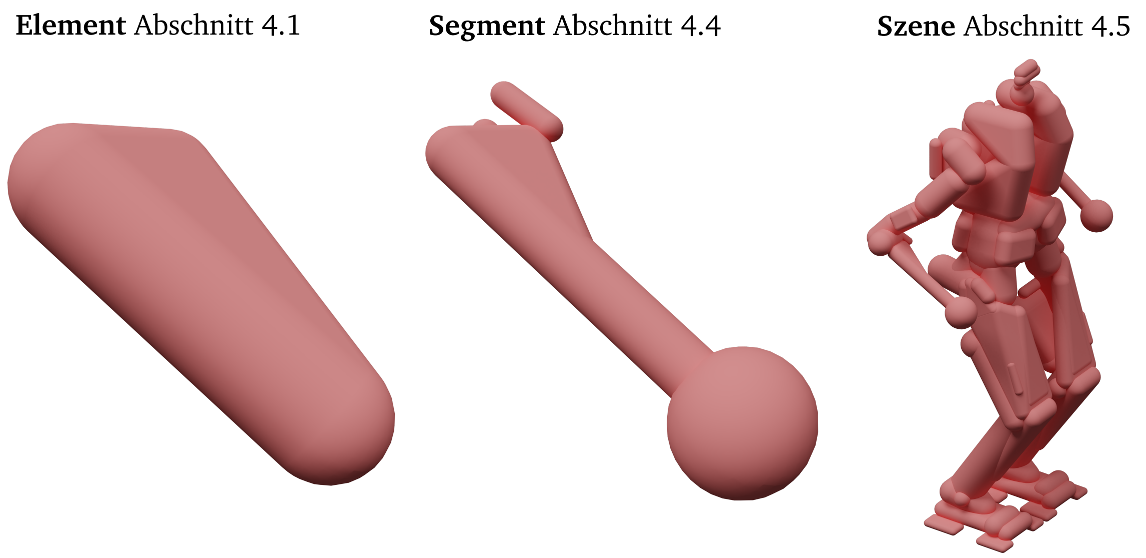
4.1 Swept-Sphere Geometrie

Wie in Kapitel 2 gezeigt, gibt es eine Vielzahl an Möglichkeiten zur Darstellung bzw. Abstrahierung von Geometrien. Dabei gibt es immer einen Zielkonflikt: Eine exaktere Beschreibung der realen Geometrien führt zu einem genaueren Ergebnis der Distanzberechnung. Dies ist jedoch mit einem höheren Rechenaufwand verbunden, welcher im direkten Zusammenhang mit der Berechnungsdauer steht. Die Schwierigkeit besteht darin, einen für die Anwendung zulässigen Kompromiss zwischen Genauigkeit und Schnelligkeit zu finden.
Eine mögliche Darstellung, welche diesen Ansprüchen gerecht wird, ist die Beschreibung durch sogenannte *Swept-Sphere-Volumens* (SSV). Trotz der sehr einfachen Beschreibung eines solchen Volumens, sind sie in der Lage auch sehr komplexe Geometrien ausreichend genau beschreiben zu können, ohne dabei das Kriterium der Schnelligkeit zu verletzen.

Allgemein ist ein *Swept-Sphere-Volume* durch zwei Bestandteile definiert: einer Kugel und einem 2-dimensionalen geometrischen Grundobjekt, auf welchem das Zentrum der Kugel abgefahren wird. Aus der Kombination dieser beiden Teile entsteht ein *Swept-Sphere-Volume*[1].

---

[1] Mathematisch ist ein solches Volumen durch die *Minkowski-Summe* beschrieben - https://de.wikipedia.org/wiki/Minkowski-Summe



Element Abschnitt 4.1 **Segment** Abschnitt 4.4 **Szene** Abschnitt 4.5

Abbildung 4.1: Übersicht der drei Ebenen zur Beschreibung einer Geometrie. Links: einzelnes SSV Element (Dreieck). Mitte: SSV Segment (Ellbogen Flexion von *Lola*) bestehend aus mehreren SSV Elementen. Rechts: SSV Szene bestehend aus mehreren SSV Segmenten.

Eine andere Entstehungsweise eines SSVs ist, dass das Grundobjekt gleichmäßig in alle drei Dimensionen ausgedehnt wird. Dadurch weist jeder Punkt der Außenhülle denselben Abstand zum Grundobjekt auf.

4.2 Swept-Sphere Element

Für die Anwendung der Distanzberechnung werden drei verschiedene Grundobjekte herangezogen: Punkt, Linie und Dreieck. Trotz der einfachen Beschreibung dieser 1- und 2-dimensionalen Geometrien lassen sich auch komplexe Objekte durch geschicktes Kombinieren darstellen (siehe Abschnitt 4.4). Die Berechnung des Abstands solcher Objekte erfolgt jedoch immer anhand der Auswertung von zwei einfachen Elementen.
Ein Vorteil dieser Geometriedarstellung ist, dass die Berechnung der Abstände nur anhand der beiden zu vergleichenden 2-dimensionalen Grundobjekte durchgeführt werden muss. Dies ist bedingt durch folgende zwei Eigenschaften:

- Die kürzeste Distanz zu einer Linie steht immer senkrecht auf diese; bei einem Dreieck immer senkrecht auf die aufgespannte Ebene oder eine der Kanten.
- Durch die Definition der SSVs weisen alle Punkte auf der Oberfläche des Volumens denselben Abstand zum Grundobjekt auf.

Nach Berechnung des minimalen Abstands der Grundobjekte folgt durch Subtraktion der beiden Radien die tatsächlich geringste Distanz zwischen den Elementen.

4.2.1 Punktelement

Das Punktelement, oder auch *Point-Swept-Sphere* (PSS), ist definiert durch:

- Radius r

- Ortsvektor zur Beschreibung des Kugelzentrums $\boldsymbol{p}$

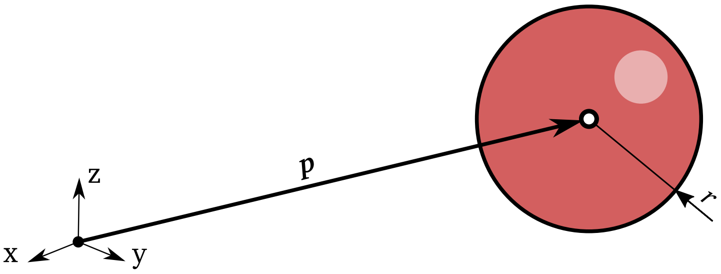


Abbildung 4.2: Vollständige Beschreibung eines Punktelements (PSS) mit dem Ortsvektor $\boldsymbol{p}$ und dem Radius r.

4.2.2 Linienelement

Das Linienelement, oder auch *Line-Swept-Sphere* (LSS), ist definiert durch:

- Radius r
- zwei Punkte $\boldsymbol{v}_0$ und $\boldsymbol{v}_1$, welche den Start- und Endpunkt des Linienelements beschreiben

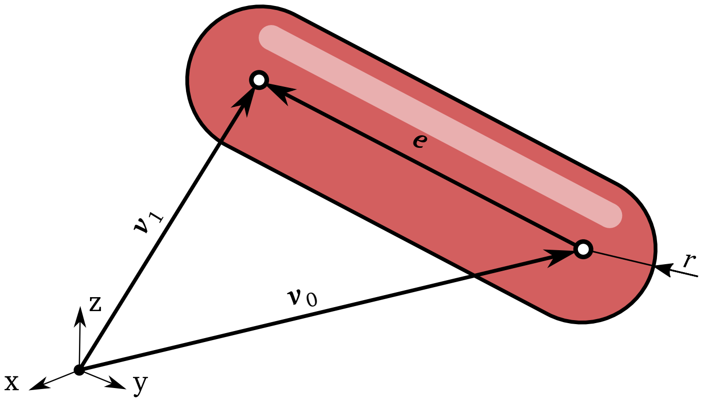


Abbildung 4.3: Vollständige Beschreibung eines Linienelements (LSS) mit den Vektoren der Eckpunkte $\boldsymbol{v}_i$, der Kante $\boldsymbol{e}$ und dem Radius r.

Anhand dieser beiden Eckpunkte ist auch die Kante des Elements definiert durch $\boldsymbol{e} := \boldsymbol{v}_1 - \boldsymbol{v}_0$. Jeder Punkt auf der Linie ist somit beschrieben durch

$$\mathcal{L}(s) = \boldsymbol{p}(s) = \boldsymbol{v}_0 + s\ \boldsymbol{e}, \tag{4.1}$$

mit dem Parameter $s \in [0,1]$.

4.2.3 Dreieckselement

Das Dreieckselement, oder auch *Triangle-Swept-Sphere* (TSS), ist definiert durch:

- Radius r
- drei Eckpunkte $\boldsymbol{v}_i$ für $i \in \{0,1,2\}$, welche das Dreieck eindeutig beschreiben.

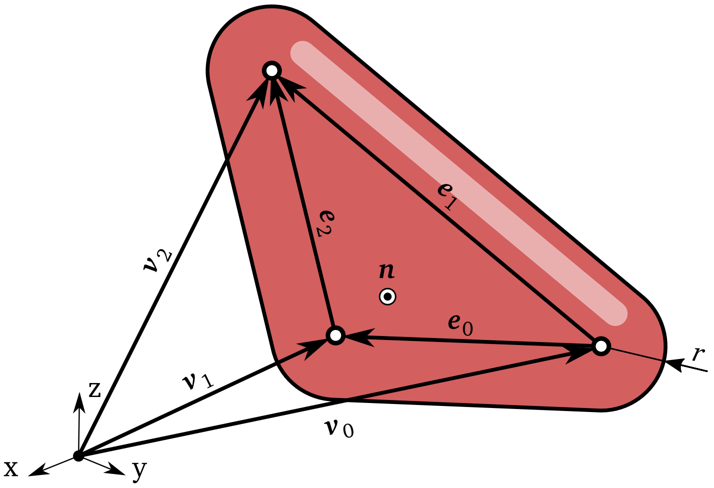


Abbildung 4.4: Vollständige Beschreibung eines Dreieckselements (TSS) mit den Vektoren der Eckpunkte $\boldsymbol{v}_i$, der Kanten $\boldsymbol{e}_i$, des Normalenvektors $\boldsymbol{n}$ und dem Radius r.

Anhand der Eckpunkte sind auch die drei Kanten des Elements definiert durch

$$\boldsymbol{e}_0 := \boldsymbol{v}_1 - \boldsymbol{v}_0\,, \qquad \boldsymbol{e}_1 := \boldsymbol{v}_2 - \boldsymbol{v}_0 \quad \text{und} \qquad \boldsymbol{e}_2 := \boldsymbol{v}_2 - \boldsymbol{v}_1. \tag{4.2}$$

Mit diesen kann der Normalenvektor der vom Dreieck aufgespannten Ebene berechnet werden

$$\boldsymbol{n} := \boldsymbol{e}_0 \times \boldsymbol{e}_1. \tag{4.3}$$

Jeder Punkt auf dem Dreieck ist somit beschrieben durch

$$\mathcal{T}(\boldsymbol{s}) = \boldsymbol{p}(s_0,\, s_1) = \boldsymbol{v_0} + s_0\, \boldsymbol{e}_0 + s_1\, \boldsymbol{e_1} \tag{4.4}$$

mit den Parametern s_i, für welche folgende Bedingungen gelten

$$0 \leq s_0 \leq 1 \quad \wedge \quad 0 \leq s_1 \leq 1 \quad \wedge \quad s_0 + s_1 \leq 1. \tag{4.5}$$

4.3 Distanzberechnungen zwischen den verschiedenen Elementen

Die genauen mathematischen Hintergründe und Herleitungen können in [17, Abschnitt 6.4] nachgelesen werden. Es werden die Ergebnisse und daraus resultierenden Fallunterscheidungen mit Erklärungen zum besseren Verständnis der Algorithmen angeführt.

Da jeder Punkt auf einem Linien- sowie Dreieckselement durch Parameter beschrieben wird, folgt aus Glg. (4.1) und (4.4) allgemein $\mathcal{S} = \mathcal{S}(\boldsymbol{x})$. $\mathcal{S}$ beschreibt somit einen Punkt auf dem Grundelement und $\boldsymbol{x}$ den Parametervektor des jeweiligen SSV Elements (z.B. der Paramter s beim Linienelement). Folgendes Minimierungsproblem kann nun aufgestellt werden

$$\phi = \frac{1}{2}\boldsymbol{c}^{\mathrm{T}}\boldsymbol{c} \qquad\qquad \text{mit} \quad \boldsymbol{c} = \mathcal{S}_1(\boldsymbol{x}_1) - \mathcal{S}_0(\boldsymbol{x}_0) \tag{4.6}$$

$$\boldsymbol{x}^* = \begin{pmatrix} \boldsymbol{x}_0^* & \boldsymbol{x}_1^* \end{pmatrix}^{\mathrm{T}} = \arg\min_{\boldsymbol{x}} \phi(\boldsymbol{x}), \tag{4.7}$$

mit ϕ als Kostenfunktion des Optimierungsproblems, $\boldsymbol{c}$ als allgemeinen Verbindungsvektor von zwei Elementen und dem Parametersatz $\boldsymbol{x}^*$, welcher die Kostenfunktion ϕ minimiert.

Somit ist der kürzeste Verbindungsvektor $\boldsymbol{c}^*$ bestimmt, anhand dessen Länge unter Berücksichtigung der Radien der geringste Abstand d der beiden Elemente gegeben ist

$$\boldsymbol{c}^* = \mathcal{S}_1(\boldsymbol{x}_1^*) - \mathcal{S}_0(\boldsymbol{x}_0^*) \quad \Longrightarrow \quad d = \|\boldsymbol{c}^*\| - r_0 - r_1. \tag{4.8}$$

Wie in Tab. 4.1 zu sehen, ergeben sich bei der gegebenen Anzahl von $n = 3$ Elementtypen folglich $n \cdot n = 9$ mögliche Paarungen von Elementen zur Distanzberechnung. Die in Tab. 4.1 verwendeten Abkürzungen sind wie folgt definiert: P $\mathrel{\hat{=}}$ point-, L $\mathrel{\hat{=}}$ line- und T $\mathrel{\hat{=}}$ triangle-Element. Z. B. PT $\mathrel{\hat{=}}$ Punkt- zu Dreiecksberechnung aus Abschnitt 4.3.3. Diese Abkürzungen werden im weiteren Verlauf der Arbeit beibehalten.

Tabelle 4.1: In Anlehnung an [17, Tab. 6.1]: Auflistung aller neun möglichen Paarungen zur Distanzberechnung mit Verweis auf den jeweiligen Abschnitt sowie Algorithmus.

nach von	PSS $\boldsymbol{p}_1$	LSS $\mathcal{L}_1(t)$	TSS $\mathcal{T}_1(\boldsymbol{t})$
PSS $\boldsymbol{p}_0$	$PP(\boldsymbol{p}_0,\ \boldsymbol{p}_1)$ Abschnitt 4.3.1 Algorithmus A.1	$PL(\boldsymbol{p}_0,\ \mathcal{L}_1(t))$ Abschnitt 4.3.2 Algorithmus A.2	$PT(\boldsymbol{p}_0,\ \mathcal{T}_1(\boldsymbol{t}))$ Abschnitt 4.3.3 Algorithmus A.3
LSS $\mathcal{L}_0(s)$	$LP(\mathcal{L}_0(s),\ \boldsymbol{p}_1)$	$LL(\mathcal{L}_0(s),\ \mathcal{L}_1(t))$ Abschnitt 4.3.4 Algorithmus A.4	$LT(\mathcal{L}_0(s),\ \mathcal{T}_1(\boldsymbol{t}))$ Abschnitt 4.3.5 Algorithmus A.5
TSS $\mathcal{T}_0(\boldsymbol{s})$	$TP(\mathcal{T}_0(\boldsymbol{s}),\ \boldsymbol{p}_1)$	$TL(\mathcal{T}_0(\boldsymbol{s}),\ \mathcal{L}_1(t))$	$TT(\mathcal{T}_0(\boldsymbol{s}),\ \mathcal{T}_1(\boldsymbol{t}))$ Abschnitt 4.3.6

Die Distanzberechnung zwischen zwei SSVs ist kommutativ, also unabhängig von der Reihenfolge der zu vergleichenden Elemente. Bei zwei gegebenen Elementen TSS und LSS entspricht die Distanz der Berechnung *dist*(TSS, LSS) demselben Ergebnis der Berechnung *dist*(LSS, TSS).
Dadurch weist Tab. 4.1 eine schiefsymmetrische Gestalt auf. Die Berechnungen des unteren Dreiecks erfolgen somit mit denselben Algorithmen wie jene des oberen. Dadurch ergeben sich für $n = 3$ Elementtypen insgesamt $n(n+1)/2 = 6$ verschiedene Elementpaarungen.

4.3.1 Punkt zu Punkt Distanzberechnung

Da es beim Punktelement keine Parametrisierung gibt, stellt die Verbindung der beiden Punkte bereits den geringsten Abstand dar

$$\boldsymbol{c}^* = \boldsymbol{p}_1 - \boldsymbol{p}_0. \tag{4.9}$$

4.3.2 Punkt zu Linie Distanzberechnung

Wie in Abb. 4.5 zu sehen, ist der Vektor $\boldsymbol{c}$ zur Verbindung von Punkt- zu Linienelement definiert durch

$$\boldsymbol{c} = (\boldsymbol{v}_0 + s\ \boldsymbol{e}) - \boldsymbol{p}, \tag{4.10}$$

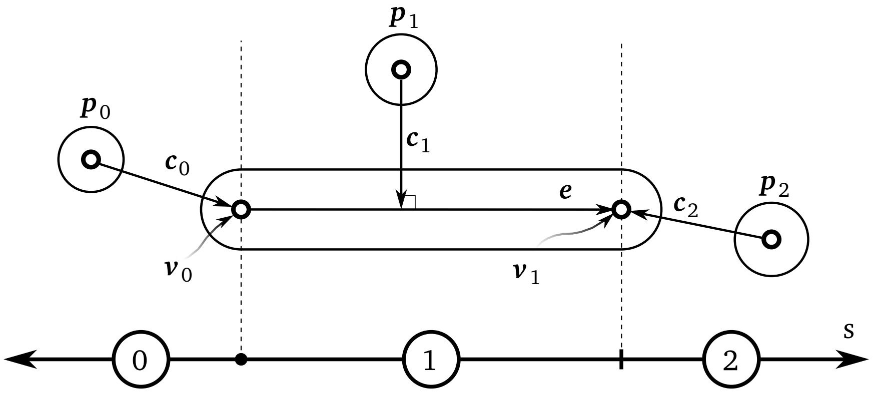


Abbildung 4.5: In Anlehnung an [17, Abb. 6.2]: Fallunterscheidung bei der Berechnung von einem Punkt- zu einem Linienelement.

mit den Vektoren $\boldsymbol{p}$ des Punktelements, $\boldsymbol{v}_0$ des ersten Eckpunkts, $\boldsymbol{e}$ der Kante und dem Parameter $s \in [0,1]$ des Linienelements. Nach dem Lösen von Glg. (4.6) nach Parameter s und Einbeziehen der drei verschiedenen Bereiche, in welchen das Punktelement in Bezug zum Linienelement liegen kann, ergibt sich folgende Berechnung für s^*:

$$s^* = \begin{cases} 0 & \text{wenn } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{e} \leq 0 \quad \Rightarrow \text{Bereich 0} \\ 1 & \text{wenn } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{e} \geq \boldsymbol{e}^{\mathrm{T}}\boldsymbol{e} \Rightarrow \text{Bereich 2} \\ (\boldsymbol{w}^{\mathrm{T}}\boldsymbol{e})/(\boldsymbol{e}^{\mathrm{T}}\boldsymbol{e}) & \text{sonst} \qquad\qquad \Rightarrow \text{Bereich 1} \end{cases} \qquad \text{mit} \quad \boldsymbol{w} = \boldsymbol{p} - \boldsymbol{v}_0 \tag{4.11}$$

Die beiden am nächsten gelegenen Punkte der beiden Elemente und deren Verbindungsvektor ergeben sich zu

$$\boldsymbol{p}^*_{\text{line}} = \boldsymbol{v}_0 + s^*\boldsymbol{e} \qquad \text{und} \qquad \boldsymbol{p}^*_{\text{point}} = \boldsymbol{p} \qquad \Rightarrow \qquad \boldsymbol{c}^* = \boldsymbol{p}^*_{\text{line}} - \boldsymbol{p}^*_{\text{point}}. \tag{4.12}$$

4.3.3 Punkt zu Dreieck Distanzberechnung

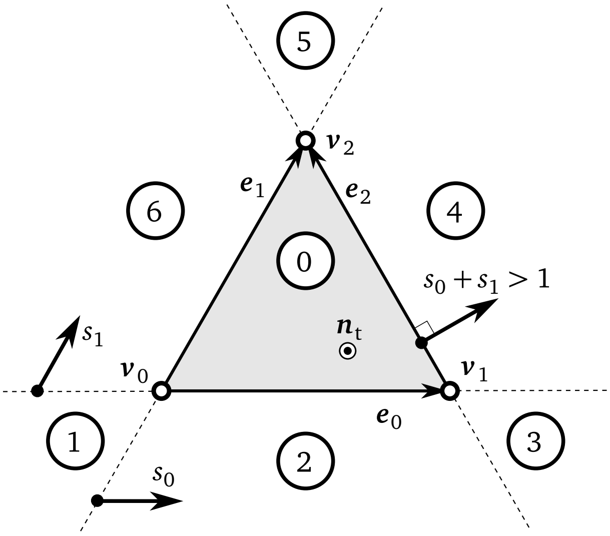


Abbildung 4.6: In Anlehnung an [17, Abb. 6.3]: Draufsicht eines Dreieckselements mit Definition der sieben möglichen Bereiche der Fallunterscheidung sowie der Linienparameter s_0 und s_1.

Die Berechnung wird in drei Schritten durchgeführt: Als erstes anhand einer Ebene, welche durch das Dreieck aufgespannt wird, im zweiten Schritt erfolgt das Einschränken der Lösung auf die möglichen Bereiche des Dreiecks (siehe Abb. 4.6) und im dritten besteht die Notwendigkeit, abhängig vom zuvor errechneten Bereich, eine weitere Fallunterscheidung an den Eckpunkten durchzuführen.

Allgemein ist der Vektor, welcher ein Punktelement mit einem Punkt auf dem Dreieckselement verbindet, wie folgt definiert:

$$\begin{aligned} \boldsymbol{c} &= \boldsymbol{v}_0 + s_0\,\boldsymbol{e}_0 + s_1\,\boldsymbol{e}_1 - \boldsymbol{p} \\ &= \boldsymbol{v}_0 + \begin{pmatrix}\boldsymbol{e}_0 & \boldsymbol{e}_1\end{pmatrix}\begin{pmatrix}s_0 & s_1\end{pmatrix}^{\mathrm{T}} - \boldsymbol{p} \\ &= \boldsymbol{v}_0 + \boldsymbol{D}\boldsymbol{s} - \boldsymbol{p} \qquad\qquad \text{mit} \quad \boldsymbol{D} = \begin{pmatrix}\boldsymbol{e}_0 & \boldsymbol{e}_1\end{pmatrix}. \end{aligned} \tag{4.13}$$

Punkt zu Ebene

Nach Lösen des Optimierungsproblems ergibt sich für den Vektor der Dreiecksparameter

$$\boldsymbol{s} = (\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D})^{-1}\boldsymbol{D}^{\mathrm{T}}(\boldsymbol{p} - \boldsymbol{v}_0) = \boldsymbol{S}\,(\boldsymbol{p} - \boldsymbol{v}_0) \qquad \text{mit} \quad \boldsymbol{S} = (\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D})^{-1}\boldsymbol{D}^{\mathrm{T}}. \tag{4.14}$$

Die Matrix $\boldsymbol{S} \in \mathbb{R}^{2\times 3}$ ist nur von den Kantenvektoren $\boldsymbol{e}_0$ und $\boldsymbol{e}_1$ abhängig und für eine Konfiguration eines Dreieckselements konstant. Sie ist damit eine fixe Eigenschaft und kann deshalb bei der Erstellung des Elements mitberechnet werden.

Punkt zu Dreieck

Anhand der Werte von s_0 und s_1 kann der Bereich des Dreiecks, in welchem der Punkt mit geringster Distanz vorliegt, ermittelt werden (siehe Tab. 4.2).

Tabelle 4.2: In Anlehnung an [17, Tab. 6.2]: Fallunterscheidung in die Bereiche des Dreieckselements anhand der Parameter s_0 und s_1 und Angabe der zugehörigen Berechnungsmethode.

Bereich	Lage auf Dreieck	Bedingung					Berechnung
0	innerhalb	$s_0 \geq 0$	$\wedge$	$s_1 \geq 0$	$\wedge$	$s_0 + s_1 < 1$	$PP(\boldsymbol{p}, \boldsymbol{v}_0 + \boldsymbol{D}\boldsymbol{s})$
1	auf Eckpunkt	$s_0 < 0$	$\wedge$	$s_1 < 0$			siehe Tab. 4.3
2	auf Kante	$s_0 \geq 0$	$\wedge$	$s_1 < 0$	$\wedge$	$s_0 + s_1 < 1$	$PL(\boldsymbol{p}, \boldsymbol{v}_0, \boldsymbol{e}_0)$
3	auf Eckpunkt			$s_1 < 0$	$\wedge$	$s_0 + s_1 \geq 1$	siehe Tab. 4.3
4	auf Kante	$s_0 \geq 0$	$\wedge$	$s_1 \geq 0$	$\wedge$	$s_0 + s_1 \geq 1$	$PL(\boldsymbol{p}, \boldsymbol{v}_1, \boldsymbol{e}_2)$
5	auf Eckpunkt	$s_0 < 0$		$\wedge$		$s_0 + s_1 \geq 1$	siehe Tab. 4.3
6	auf Kante	$s_0 < 0$	$\wedge$	$s_1 \geq 0$	$\wedge$	$s_0 + s_1 < 1$	$PL(\boldsymbol{p}, \boldsymbol{v}_0, \boldsymbol{e}_1)$

Punkt zu Eckpunkt

Zusätzlich zu den ersten beiden Schritten erfolgt nun eine weitere Fallunterscheidung. Liegt die minimale Distanz laut Tab. 4.2 im Bereich eines Eckpunkts (1, 3 oder 5), so gibt es im Fall eines stumpfwinkligen Dreiecks die Möglichkeit, dass die Verbindung vom jeweiligen Eckpunkt zum Punktelement nicht die kürzeste Distanz darstellt. Dieser Fall wurde weder in [17, Abschnitt 6.4.3] noch im SSV-Modul von *am2b* berücksichtigt, wodurch es bisher bei Dreiecken mit stumpfen Winkel zu falschen Ergebnissen kommen konnte.
Wie in Abb. 4.7 ersichtlich, gibt es für den Fall Punkt zu Eckpunkt drei weitere Bereiche. Zur

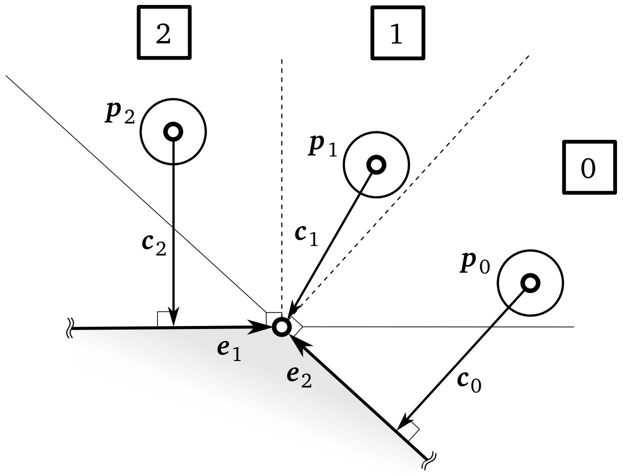


Abbildung 4.7: Draufsicht auf eine Ecke (Bereich 5) eines stumpfwinkligen Dreieckselements mit der Definition der drei Sektionen.

besseren Unterscheidung werden diese Bereiche im weiteren Verlauf als *Sektionen* bezeichnet. Abhängig von der genauen Lage des Punktelements kann bei einem stumpfen Eckwinkel die kürzeste Verbindung auch zu einer der beiden anliegenden Kanten bestehen.

Im Folgenden wird die Berechnung zur Unterscheidung der verschiedenen Sektionen angeführt. Abhängig vom Bereich und dem zugehörigen Eckpunkt ist auf die genaue Definition der Kantenvektoren (siehe Glg. (4.2)) und das korrekte Vorzeichen zur Berechnung zu achten.

Zuerst wird überprüft, ob es sich bei dem vorliegenden Dreieckselement um ein stumpfwinkliges handelt:

$$\left.\begin{array}{lr} \text{Bereich 1:} & \boldsymbol{e}_0 \cdot \boldsymbol{e}_1 < 0 \\ \text{Bereich 3:} & -\,\boldsymbol{e}_0 \cdot \boldsymbol{e}_2 < 0 \\ \text{Bereich 5:} & (-)\,\boldsymbol{e}_1 \cdot (-)\,\boldsymbol{e}_2 < 0 \end{array}\right\} \quad \Rightarrow \quad \text{stumpfwinklig} \tag{4.15}$$

Liegt ein stumpfer Winkel vor, so erfolgt die Unterscheidung der Sektionen durch Berechnung des Skalarprodukts $(\boldsymbol{p} - \boldsymbol{v}_\mathrm{i}) \cdot \boldsymbol{e}_\mathrm{i}$ mit $i \in \{0, 1, 2\}$. Auch hier ist auf das korrekte Vorzeichen bei den Kantenvektoren zu achten. In Tab. 4.3 ist die Fallunterscheidung der Sektionen zusammengefasst. Liegt ein spitzer Winkel vor, so kann ohne weitere Unterscheidung die Berechnungsmethode aus Sektion 1 verwendet werden (d.h. einfach der Eckpunkt).

Tabelle 4.3: Liegt nach Auswertung von Tab. 4.2 Bereich 1, 3 oder 5 vor und es handelt sich um ein stumpfwinkliges Eck, so erfolgt die Fallunterscheidung in eine der drei Sektionen.

Bereich	Sektion	Element mit min. Abstand	Bedingung			Berechnung
1	0	Kante	$(\boldsymbol{p}-\boldsymbol{v}_0)^\mathrm{T}\boldsymbol{e}_1$	$>$	0	$PL(\boldsymbol{p},\boldsymbol{v}_0,\boldsymbol{e}_1)$
	2	Kante	$(\boldsymbol{p}-\boldsymbol{v}_0)^\mathrm{T}\boldsymbol{e}_0$	$>$	0	$PL(\boldsymbol{p},\boldsymbol{v}_0,\boldsymbol{e}_0)$
	1	Eckpunkt	sonst			$PP(\boldsymbol{p},\boldsymbol{v}_0)$
3	0	Kante	$-(\boldsymbol{p}-\boldsymbol{v}_1)^\mathrm{T}\boldsymbol{e}_0$	$>$	0	$PL(\boldsymbol{p},\boldsymbol{v}_0,\boldsymbol{e}_0)$
	2	Kante	$(\boldsymbol{p}-\boldsymbol{v}_1)^\mathrm{T}\boldsymbol{e}_2$	$>$	0	$PL(\boldsymbol{p},\boldsymbol{v}_1,\boldsymbol{e}_2)$
	1	Eckpunkt	sonst			$PP(\boldsymbol{p},\boldsymbol{v}_1)$
5	0	Kante	$-(\boldsymbol{p}-\boldsymbol{v}_2)^\mathrm{T}\boldsymbol{e}_2$	$>$	0	$PL(\boldsymbol{p},\boldsymbol{v}_1,\boldsymbol{e}_2)$
	2	Kante	$-(\boldsymbol{p}-\boldsymbol{v}_2)^\mathrm{T}\boldsymbol{e}_1$	$>$	0	$PL(\boldsymbol{p},\boldsymbol{v}_0,\boldsymbol{e}_1)$
	1	Eckpunkt	sonst			$PP(\boldsymbol{p},\boldsymbol{v}_2)$

Punkt- zu Eckpunkt - pragmatische Herangehensweise

Liegt die kürzeste Distanz nach Auswertung von Tab. 4.2 im Bereich eines Eckpunkts 1, 3 oder 5, so gibt es eine weitere Möglichkeit den korrekten Abstand bei stumpfwinkligen Dreieckselementen zu berechnen. Anhand der am Eckpunkt anliegenden Kanten werden jeweils eine Punkt- zu Linienberechnung (siehe Abschnitt 4.3.2) durchgeführt und davon die kürzere Distanz übernommen. Es wurde die Performance der beiden Herangehensweisen zur Punkt- zu Eckpunktberechnung getestet und ausgewertet - mehr dazu in Abschnitt 6.2.2.

4.3.4 Linie zu Linie Distanzberechnung

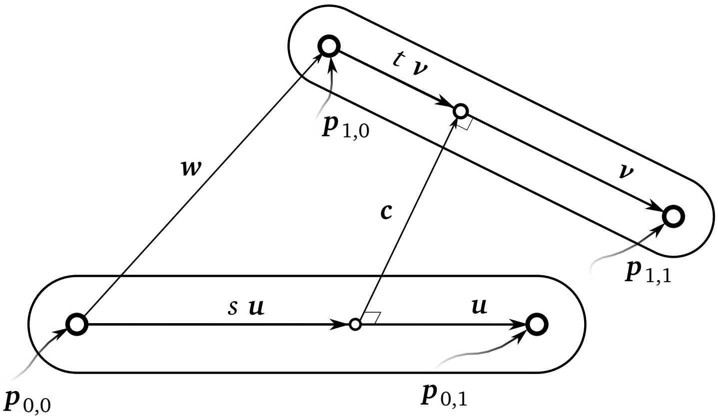


Abbildung 4.8: In Anlehnung an [17, Abb. 6.4]: Distanzberechnung von zwei Linienelementen mit Definition der Vektoren.

Wie in Abb. 4.8 zu sehen, sind die beiden Linienelemente wie folgt beschrieben

$$\mathcal{L}_0(s) = \boldsymbol{p}_{0,0} + s\,\boldsymbol{u} \qquad \text{bzw.} \qquad \mathcal{L}_1(s) = \boldsymbol{p}_{1,0} + t\,\boldsymbol{v} \tag{4.16}$$

mit $s \in [0,1]$ und $t \in [0,1]$, als jeweilige Linienparameter und

$$\boldsymbol{u} = \boldsymbol{p}_{0,1} - \boldsymbol{p}_{0,0} \qquad \text{und} \qquad \boldsymbol{v} = \boldsymbol{p}_{1,1} - \boldsymbol{p}_{1,0}. \tag{4.17}$$

Distanzberechnung zwischen zwei unendlichen Linien

Allgemein ist ein Verbindungsvektor zwischen zwei Linienelementen definiert durch

$$\boldsymbol{c} = \boldsymbol{w} - s\boldsymbol{u} + t\boldsymbol{v} = \boldsymbol{w} + \boldsymbol{D}\boldsymbol{x} \tag{4.18}$$

mit $\boldsymbol{w} = \boldsymbol{p}_{1,0} - \boldsymbol{p}_{0,0}$ als Verbindungsvektor der beiden Startpunkte, $\boldsymbol{D} = \begin{pmatrix} -\boldsymbol{u} & \boldsymbol{v} \end{pmatrix} \in \mathbb{R}^{3\times 2}$ und $\boldsymbol{x} = \begin{pmatrix} s & t \end{pmatrix}^{\mathrm{T}}$ als Vektor der Linienparameter.

Aufgrund der Eigenschaft $\boldsymbol{a}^{\mathrm{T}}\boldsymbol{b} = 0 \Leftrightarrow \boldsymbol{a} \perp \boldsymbol{b}$ und der Tatsache, dass der geringste Abstand von zwei Linien immer rechtwinklig auf diese steht, folgt aus Glg. (4.18) durch Erweiterung mit $\boldsymbol{u}^{\mathrm{T}}$ bzw. $\boldsymbol{v}^{\mathrm{T}}$

$$\boldsymbol{c} \perp \boldsymbol{u} \quad \Leftrightarrow \quad \boldsymbol{c}^{\mathrm{T}}\boldsymbol{u} = 0 = \boldsymbol{u}^{\mathrm{T}}\boldsymbol{c} \quad \Rightarrow \quad \boldsymbol{u}^{\mathrm{T}}\boldsymbol{w} - s\boldsymbol{u}^2 + t\boldsymbol{u}^{\mathrm{T}}\boldsymbol{v} = 0\,, \tag{4.19}$$

$$\boldsymbol{c} \perp \boldsymbol{v} \quad \Leftrightarrow \quad \boldsymbol{c}^{\mathrm{T}}\boldsymbol{v} = 0 = \boldsymbol{v}^{\mathrm{T}}\boldsymbol{c} \quad \Rightarrow \quad \boldsymbol{v}^{\mathrm{T}}\boldsymbol{w} - s\boldsymbol{v}^{\mathrm{T}}\boldsymbol{u} + t\boldsymbol{v}^2 = 0\,. \tag{4.20}$$

Die Lösung des Optimierungsproblems liefert

$$\begin{aligned}\boldsymbol{x}^* &= -(\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D})^{-1}\boldsymbol{D}^{\mathrm{T}}\boldsymbol{w}\\ &= \begin{pmatrix} s^* \\ t^* \end{pmatrix} := \frac{1}{\boldsymbol{u}^2\,\boldsymbol{v}^2 - (\boldsymbol{u}^{\mathrm{T}}\boldsymbol{v})^2}\begin{pmatrix} \boldsymbol{v}^2\,\boldsymbol{u}^{\mathrm{T}}\boldsymbol{w} - \boldsymbol{u}^{\mathrm{T}}\boldsymbol{v}\,\boldsymbol{v}^{\mathrm{T}}\boldsymbol{w} \\ \boldsymbol{u}^{\mathrm{T}}\boldsymbol{v}\,\boldsymbol{u}^{\mathrm{T}}\boldsymbol{w} - \boldsymbol{u}^2\,\boldsymbol{v}^{\mathrm{T}}\boldsymbol{w} \end{pmatrix} = \frac{1}{N}\begin{pmatrix} s_{\mathrm{Z}} \\ t_{\mathrm{Z}} \end{pmatrix}. \end{aligned} \tag{4.21}$$

Der Wert des Nenners N aus Glg. (4.21) gibt an, ob die Linien parallel zueinander stehen. Ist $N = 0$, so sind die Linien parallel. In Abschnitt 4.3.4 wird der numerische Aspekt dieser Beurteilung genauer beleuchtet.

Distanzberechnung zwischen nicht-parallelen Linien

Mit Glg. (4.21) können im Fall nicht-paralleler Elemente die Parameter s^* und t^* berechnet werden. Sie beschreiben den Schnittpunkt von unendlich langen Linien. Es sind somit Parameter $\notin [0,1]$ möglich. Zur Eingrenzung der Lösung erfolgt die Berechnung in drei Schritten. Hierfür werden für die Parameter die Indizes 0, 1 und 2 verwendet, so folgt z.B. $t_0 \to t_1 \to t_2$.

Im ersten Schritt wird anhand der zweiten Zeile von Glg. (4.21) $t_0 = t^*$ berechnet. Anschließend erfolgt in Tab. 4.4, abhängig von t_0, eine Fallunterscheidung. Im letzten Schritt wird anhand von s_1 in Tab. 4.5 eine letzte Fallunterscheidung durchgeführt.

In [17, Tab. 6.3] für den Fall $t_0 > 1$ und in [17, Tab. 6.4, Solution 2] für den Fall $s_1 > 1$ sind folgende Formeln zur Berechnung der Parameter gegeben

$$s_1 = \frac{\boldsymbol{u}^{\mathrm{T}}(\boldsymbol{u}-\boldsymbol{w})}{\boldsymbol{u}^2} \quad \text{und} \quad s_1 = \frac{\boldsymbol{v}^{\mathrm{T}}(\boldsymbol{u}-\boldsymbol{w})}{\boldsymbol{u}^{\mathrm{T}}\boldsymbol{v}} \quad \text{bzw.} \tag{4.22}$$

$$t_2 = \frac{\boldsymbol{u}^{\mathrm{T}}(\boldsymbol{v}-\boldsymbol{W})}{\boldsymbol{v}^2}\,. \tag{4.23}$$

Durch Einsetzen von $t_1 = 1$ in Glg. (4.19) und (4.20) bzw. $s_2 = 1$ in Glg. (4.20) folgen jedoch folgende Formeln

$$s_1 = \frac{\boldsymbol{u}^{\mathrm{T}}(\boldsymbol{v}+\boldsymbol{w})}{\boldsymbol{u}^2} \quad \text{und} \quad s_1 = \frac{\boldsymbol{v}^{\mathrm{T}}(\boldsymbol{v}+\boldsymbol{w})}{\boldsymbol{u}^{\mathrm{T}}\boldsymbol{v}} \quad \text{bzw.} \tag{4.24}$$

$$t_2 = \frac{\boldsymbol{v}^{\mathrm{T}}(\boldsymbol{u}-\boldsymbol{W})}{\boldsymbol{v}^2}\,. \tag{4.25}$$

Im weiteren Verlauf werden die Formeln aus Glg. (4.24) und (4.25) zur Berechnung der Parameter verwendet.

Tabelle 4.4: In Anlehnung an [17, Tab. 6.3]: Im Original liegt ein Tippfehler vor, deshalb werden zur Berechnung von t_1 und s_1 im Fall $t_0 > 1$ die Formeln aus Glg. (4.24) angegeben. Der Vollständigkeit halber wird auch *Lösung 2* angegeben, welche jedoch nicht für eine numerisch robuste Berechnung geeignet ist.

Fall	Grenzwert	Lösung 1	Lösung 2
$t_0 < 0$	$t_1 = 0$	$s_1 = \frac{\boldsymbol{u}^{\mathrm{T}}\boldsymbol{w}}{\boldsymbol{u}^2}$	$s_1 = \frac{\boldsymbol{v}^{\mathrm{T}}\boldsymbol{w}}{\boldsymbol{u}^{\mathrm{T}}\boldsymbol{v}}$
$t_0 > 1$	$t_1 = 1$	$s_1 = \frac{\boldsymbol{u}^{\mathrm{T}}(\boldsymbol{v}+\boldsymbol{w})}{\boldsymbol{u}^2}$	$s_1 = \frac{\boldsymbol{v}^{\mathrm{T}}(\boldsymbol{v}+\boldsymbol{w})}{\boldsymbol{u}^{\mathrm{T}}\boldsymbol{v}}$
sonst	$t_1 = t_0$	$s_1 = s_0$	$s_1 = s_0$

Wie in Tab. 4.4 und 4.5 angeführt, gibt es zwei mögliche Lösungen für s_1 bzw. t_2. Für eine numerisch robuste Berechnung der Parameter sollte bei Tab. 4.4 - *Lösung 1* und bei Tab. 4.5

Tabelle 4.5: In Anlehnung an [17, Tab. 6.3]: Im Original liegt ein Tippfehler vor, deshalb wird zur Berechnung von t_2 im Fall $s_1 > 1$ die Formel aus Glg. (4.25) angegeben. Der Vollständigkeit halber wird auch *Lösung 1* angegeben, welche jedoch nicht für eine numerisch robuste Berechnung geeignet ist.

Fall	Grenzwert	Lösung 1	Lösung 2
$s_1 < 0$	$s_2 = 0$	$t_2 = \begin{cases} 0 & -u^{\mathrm{T}}w < 0 \\ 1 & -u^{\mathrm{T}}w > u^{\mathrm{T}}v \\ \frac{-u^{\mathrm{T}}w}{u^{\mathrm{T}}v} & \text{sonst} \end{cases}$	$t_2 = \begin{cases} 0 & -\boldsymbol{v}^{\mathrm{T}}\boldsymbol{w} < 0 \\ 1 & -\boldsymbol{v}^{\mathrm{T}}\boldsymbol{w} > \boldsymbol{v}^2 \\ \frac{-\boldsymbol{v}^{\mathrm{T}}\boldsymbol{w}}{\boldsymbol{v}^2} & \text{sonst} \end{cases}$
$s_1 > 1$	$s_2 = 1$	$t_2 = \begin{cases} 0 & u^{\mathrm{T}}(u-w) < 0 \\ 1 & u^{\mathrm{T}}(u-w) > u^{\mathrm{T}}v \\ \frac{u^{\mathrm{T}}(u-w)}{u^{\mathrm{T}}v} & \text{sonst} \end{cases}$	$t_2 = \begin{cases} 0 & \boldsymbol{v}^{\mathrm{T}}(\boldsymbol{u}-\boldsymbol{w}) < 0 \\ 1 & \boldsymbol{v}^{\mathrm{T}}(\boldsymbol{u}-\boldsymbol{w}) > \boldsymbol{v}^2 \\ \frac{\boldsymbol{v}^{\mathrm{T}}(\boldsymbol{u}-\boldsymbol{w})}{\boldsymbol{v}^2} & \text{sonst} \end{cases}$
sonst	$s_2 = s_1$	$t_2 = t_1$	$t_2 = t_1$

- *Lösung 2* verwendet werden. Grund hierfür ist, dass das Skalarprodukt der beiden Kanten $\boldsymbol{u}^{\mathrm{T}}\boldsymbol{v}$ kann bei rechtwinkliger Anordnung der Elemente zu einer Division durch Null führen.

$$\boldsymbol{u} \perp \boldsymbol{v} \qquad \Rightarrow \qquad \boldsymbol{u}^{\mathrm{T}}\boldsymbol{v} = 0 \qquad \Rightarrow \qquad \frac{\cdots}{\boldsymbol{u}^{\mathrm{T}}\boldsymbol{v}} \to \infty \tag{4.26}$$

Distanzberechnung zwischen parallelen Linien

Durch Projektion der Vektoren $\boldsymbol{v}$ und $\boldsymbol{w}$ auf $\boldsymbol{u}$ wird die Berechnung der Parameter s und t auf ein eindimensionales Problem reduziert. Hierbei wird auf [17, Tab. 6.5] verwiesen, wo alle elf möglichen Fälle beschrieben sind.

Numerische Beurteilung des parallelen Falls

Zur Beurteilung ob ein paralleler Fall vorliegt, wird der Nenner N aus Glg. (4.21) herangezogen. Durch numerische Ungenauigkeiten in den Berechnungen ist es sehr unwahrscheinlich, dass N den exakten Wert 0 annimmt. Deshalb wird N mit der Maschinengenauigkeit ϵ[2] verglichen.
Bezugnehmend auf [17, Abschnitt 6.4.4] wird eine numerisch robustere Variante gezeigt.

Die Vektoren $\boldsymbol{u}$ und $\boldsymbol{v}$ der beiden Elemente sind genau dann parallel, wenn

$$\frac{\boldsymbol{u}}{\|\boldsymbol{u}\|} \cdot \frac{\boldsymbol{v}}{\|\boldsymbol{v}\|} = 1.$$

Quadrieren und Umformen ergibt

$$\frac{\boldsymbol{u}^2\boldsymbol{v}^2 - (\boldsymbol{u}^{\mathrm{T}}\boldsymbol{v})^2}{\boldsymbol{u}^2\boldsymbol{v}^2} = \frac{N}{\boldsymbol{u}^2\boldsymbol{v}^2} = 0. \tag{4.27}$$

Für eine verlässliche und numerisch robuste Entscheidung, ob ein paralleler Fall vorliegt, ist Glg. (4.27) der Bedingung $N = 0$ vorzuziehen. Es folgt somit folgendes Kriterium:

$$\boxed{\frac{N}{\boldsymbol{u}^2\boldsymbol{v}^2} < \epsilon} \tag{4.28}$$

---

[2] Nach *IEEE 754* liegt ϵ bei $2^{-24} \approx 6{,}0 \cdot 10^{-8}$ (*single precision*) bzw. $2^{-53} \approx 1{,}1 \cdot 10^{-16}$ (*double precision*).

Es wird darauf hingewiesen, dass im Realfall bei einer Berechnung der parallele Fall bei Linie- zu Linien-, sowie der folgenden Linie- zu Dreiecksberechnungen nur selten auftreten wird: einerseits durch numerische Fehler in den Berechnungen, andererseits wird bei einer bewegten Szene der parallele Fall nur kurz auftreten und im nächsten Berechnungszyklus mit hoher Wahrscheinlichkeit von einem nicht-parallelen Fall abgelöst werden.

4.3.5 Linie zu Dreieck Distanzberechnung

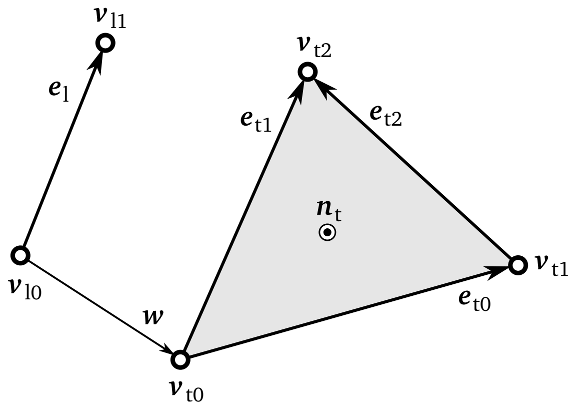


Abbildung 4.9: In Anlehnung an [17, Abb. 6.7]: Definition aller Größen welche zur Linien- zu Dreiecksberechnung benötigt werden.

Jeder beliebige Punkt auf einem Linien- bzw. Dreieckselement ist beschrieben durch

$$\mathcal{L}(t) = \boldsymbol{v}_{\mathrm{l0}} + t\,\boldsymbol{e}_{\mathrm{l}} \qquad \text{und} \qquad \mathcal{T}(\boldsymbol{s}) = \boldsymbol{v}_{\mathrm{t0}} + s_0\,\boldsymbol{e}_{\mathrm{t0}} + s_1\,\boldsymbol{e}_{\mathrm{t1}}\,, \tag{4.29}$$

mit den Parametern $s_0,\ s_1 \in [0,1]$ und $t \in [0,1]$ und den jeweiligen Kantenvektoren

$$\boldsymbol{e}_{\mathrm{l}} := \boldsymbol{v}_{\mathrm{l1}} - \boldsymbol{v}_{\mathrm{l0}}\,, \qquad \boldsymbol{e}_{\mathrm{t0}} := \boldsymbol{v}_{\mathrm{t1}} - \boldsymbol{v}_{\mathrm{t0}} \qquad \text{und} \qquad \boldsymbol{e}_{\mathrm{t1}} := \boldsymbol{v}_{\mathrm{t2}} - \boldsymbol{v}_{\mathrm{t0}}\,. \tag{4.30}$$

Schnittpunkt zwischen unendlicher Linie und Ebene

Eine Linie und eine Ebene schneiden sich im $\mathbb{R}^3$ immer, außer beide Elemente liegen parallel zueinander. Um den Schnittpunkt zu finden, definieren wir den Verbindungsvektor der beiden Elemente wie folgt

$$\begin{aligned} \boldsymbol{c} &= \mathcal{T}(\boldsymbol{s}) - \mathcal{L}(t) && (4.31)\\ &= \boldsymbol{w} + \boldsymbol{D}\boldsymbol{x} \qquad \text{mit} \quad \boldsymbol{D} = \begin{pmatrix} \boldsymbol{e}_{\mathrm{t0}} & \boldsymbol{e}_{\mathrm{t1}} & -\boldsymbol{e}_{\mathrm{l}} \end{pmatrix} \in \mathbb{R}^{3\times 3}. && (4.32) \end{aligned}$$

Vektor $\boldsymbol{w} = \boldsymbol{v}_{\mathrm{t0}} - \boldsymbol{v}_{\mathrm{l0}}$ verbindet die beiden Startpunkte der Elemente, $\boldsymbol{x} = \begin{pmatrix} \boldsymbol{s}^{\mathrm{T}} & t \end{pmatrix}^{\mathrm{T}} = \begin{pmatrix} s_0 & s_1 & t \end{pmatrix}^{\mathrm{T}}$ ist der Vektor mit den Parametern der Elemente. In [17, Glg. (6.38)] zur Definition von $\boldsymbol{D}$ liegt ein Tippfehler vor: statt $\boldsymbol{e}_{\mathrm{t1}}$ wurde $\boldsymbol{e}_{\mathrm{t0}}$ geschrieben.

Aus Glg. (4.6) folgt für $\boldsymbol{x}^*$

$$\begin{aligned} \boldsymbol{x}^* &= -\boldsymbol{D}^{-1}\boldsymbol{w} \\ &= \frac{1}{\boldsymbol{n}_{\mathrm{t}}{}^{\mathrm{T}}\boldsymbol{e}_{\mathrm{l}}} \begin{pmatrix} \boldsymbol{e}_{\mathrm{l}} \times \boldsymbol{e}_{\mathrm{t1}} & \boldsymbol{e}_{\mathrm{t0}} \times \boldsymbol{e}_{\mathrm{l}} & \boldsymbol{n}_{\mathrm{t}} \end{pmatrix}^{\mathrm{T}} \boldsymbol{w} \qquad \text{mit} \quad \boldsymbol{n}_{\mathrm{t}} := \boldsymbol{e}_{\mathrm{t0}} \times \boldsymbol{e}_{\mathrm{t1}}. && (4.33) \end{aligned}$$

Dabei stellt $\boldsymbol{n}_\mathrm{t}$ den Normalenvektor des Dreiecks dar.

Glg. (4.33) wird singulär, wenn

$$\boldsymbol{n}_\mathrm{t}{}^\mathrm{T}\boldsymbol{e}_\mathrm{l} = 0 \qquad \Leftrightarrow \qquad \boldsymbol{n}_\mathrm{t} \perp \boldsymbol{e}_\mathrm{l} \qquad \Leftrightarrow \qquad \mathcal{L}(t) \parallel \mathcal{T}(\boldsymbol{s}). \tag{4.34}$$

Somit gibt Glg. (4.34) an, ob Linie und Dreieck parallel zueinander liegen. Wie auch in Abschnitt 4.3.4, wird zur Vermeidung einer Division durch Null der parallele Fall vorab abgefangen.

Aus der Bedingung für Parallelität von einer Linie zu einer Ebene ergibt sich zur numerisch robusten Erkennung des parallelen Falls folgendes Kriterium:

$$\frac{\boldsymbol{n}_\mathrm{t}}{\|\boldsymbol{n}_\mathrm{t}\|} \cdot \frac{\boldsymbol{e}_\mathrm{l}}{\|\boldsymbol{e}_\mathrm{l}\|} = 0 \qquad \Rightarrow \qquad \boxed{\left| \frac{\boldsymbol{n}_\mathrm{t}}{\|\boldsymbol{n}_\mathrm{t}\|} \cdot \frac{\boldsymbol{e}_\mathrm{l}}{\|\boldsymbol{e}_\mathrm{l}\|} \right| < \epsilon} \tag{4.35}$$

Distanzberechnung zwischen nicht-paralleler Linie und Dreieck

Ähnlich zu Abschnitt 4.3.4 erfolgt auch hier die Berechnung in mehreren Schritten mit Fallunterscheidungen. Mit Glg. (4.33) kann der Linienparameter t_0 berechnet werden

$$t_0 = \frac{\boldsymbol{n}_\mathrm{t}{}^\mathrm{T}\boldsymbol{w}}{\boldsymbol{n}_\mathrm{t}{}^\mathrm{T}\boldsymbol{e}_\mathrm{l}}. \tag{4.36}$$

Liegt t_0 außerhalb des zulässigen Bereichs $t_0 \notin [0,1]$, so reduziert sich die Linien- zu Dreiecksberechnung zu einer Punkt- zu Dreiecksberechnung. Die weitere Berechnung erfolgt nach Abschnitt 4.3.3.

$$t_1 = \begin{cases} 0 & \text{für } t_0 < 0 \\ 1 & \text{für } t_0 > 1 \end{cases} \qquad \Rightarrow \boldsymbol{s} \text{ aus Glg. (4.14) mit } \boldsymbol{p} = \boldsymbol{v}_{\mathrm{l}0} + t_1\, \boldsymbol{e}_\mathrm{l} \tag{4.37}$$

Liegt t_0 innerhalb des zulässigen Bereichs $t_0 \in [0,1]$, folgt $t_1 = t_0$. Im nächsten Schritt werden die Parameter des Dreiecks $\boldsymbol{s}$ aus Glg. (4.33) berechnet

$$\boldsymbol{s} = \begin{pmatrix} s_0 \\ s_1 \end{pmatrix} = \frac{1}{\boldsymbol{n}_\mathrm{t}{}^\mathrm{T}\boldsymbol{e}_\mathrm{l}} \begin{pmatrix} \boldsymbol{e}_\mathrm{l} \times \boldsymbol{e}_{\mathrm{t}1} & \boldsymbol{e}_{\mathrm{t}0} \times \boldsymbol{e}_\mathrm{l} \end{pmatrix}^\mathrm{T} \boldsymbol{w}. \tag{4.38}$$

Beschreiben die Parameter einen Punkt innerhalb des Dreiecks (siehe Glg. (4.5)), so wird das Dreieck von der Linie durchstoßen und es folgt $\|\boldsymbol{c}^*\| = 0$.

Allgemein ergeben sich wie in Tab. 4.6 beschrieben, sieben mögliche Fälle. Sie fasst die Vorgehensweise zur Berechnung der minimalen Distanz von Linien- zu Dreieckselement zusammen.

Die zusätzliche Fallunterscheidung aus der Punkt- zu Dreiecksberechnung für stumpfwinklige Dreiecke (siehe Abschnitt 4.3.3) ist in diesem Fall nicht notwendig. Liegt der Schnittpunkt in einer der Eckpunktregionen (1, 3 oder 5), so werden die Distanzen zu den anliegenden Kanten berechnet und der kleinere Wert übernommen. Dadurch erfolgt eine korrekte Berechnung der minimalen Distanz auch bei einem stumpfwinkligen Dreieck.

Tabelle 4.6: In Anlehnung an [17, Tab. 6.7]: Fallunterscheidung Linie- zu Dreieckselement.

Bereich	Bedingung					Berechnung
0	$s_0 \geq 0$	$\wedge$	$s_1 \geq 0$	$\wedge$	$s_0 + s_1 < 1$	$PP(\boldsymbol{p}_{\mathcal{L}}, \boldsymbol{p}_{\mathcal{T}})$ mit $\begin{cases} \boldsymbol{p}_{\mathcal{T}} = \boldsymbol{v}_{t0} + \begin{pmatrix} \boldsymbol{e}_{t0} & \boldsymbol{e}_{t1} \end{pmatrix} \boldsymbol{s} \\ \boldsymbol{p}_{\mathcal{L}} = \boldsymbol{v}_{l0} + t\, \boldsymbol{e}_{l} \end{cases}$
1	$s_0 < 0$	$\wedge$	$s_1 < 0$			$\min_d \begin{pmatrix} LL(\boldsymbol{v}_{t0}, \boldsymbol{e}_{t0}, \boldsymbol{v}_{l0}, \boldsymbol{e}_{l}) \\ LL(\boldsymbol{v}_{t0}, \boldsymbol{e}_{t1}, \boldsymbol{v}_{l0}, \boldsymbol{e}_{l}) \end{pmatrix}$
2	$s_0 \geq 0$	$\wedge$	$s_1 < 0$	$\wedge$	$s_0 + s_1 < 1$	$LL(\boldsymbol{v}_{t0}, \boldsymbol{e}_{t0}, \boldsymbol{v}_{l0}, \boldsymbol{e}_{l})$
3			$s_1 < 0$	$\wedge$	$s_0 + s_1 \geq 1$	$\min_d \begin{pmatrix} LL(\boldsymbol{v}_{t0}, \boldsymbol{e}_{t0}, \boldsymbol{v}_{l0}, \boldsymbol{e}_{l}) \\ LL(\boldsymbol{v}_{t1}, \boldsymbol{e}_{t2}, \boldsymbol{v}_{l0}, \boldsymbol{e}_{l}) \end{pmatrix}$
4	$s_0 \geq 0$	$\wedge$	$s_1 \geq 0$	$\wedge$	$s_0 + s_1 \geq 1$	$LL(\boldsymbol{v}_{t1}, \boldsymbol{e}_{t2}, \boldsymbol{v}_{l0}, \boldsymbol{e}_{l})$
5	$s_0 < 0$		$\wedge$		$s_0 + s_1 \geq 1$	$\min_d \begin{pmatrix} LL(\boldsymbol{v}_{t0}, \boldsymbol{e}_{t1}, \boldsymbol{v}_{l0}, \boldsymbol{e}_{l}) \\ LL(\boldsymbol{v}_{t1}, \boldsymbol{e}_{t2}, \boldsymbol{v}_{l0}, \boldsymbol{e}_{l}) \end{pmatrix}$
6	$s_0 < 0$	$\wedge$	$s_1 \geq 0$	$\wedge$	$s_0 + s_1 < 1$	$LL(\boldsymbol{v}_{t0}, \boldsymbol{e}_{t1}, \boldsymbol{v}_{l0}, \boldsymbol{e}_{l})$

Distanzberechnung von paralleler Linie und Dreieck

Zur Berechnung der minimalen Distanz bei paralleler Anordnung werden in [17, Abschnitt 6.4.5] zwei Methoden diskutiert:

1. Der Anfangs- und Endpunkt des Linienelements werden in zwei einzelne Punktelemente zerlegt. Es erfolgt dann je Punkt eine Punkt- zu Dreiecksberechnung (siehe Abschnitt 4.3.3).
2. Das Dreieckselement wird in drei Linienelemente zerlegt. Jedes dieser Elemente beschreibt dabei eine Kante des Dreiecks. Es erfolgen insgesamt drei Linie- zu Linienberechnungen (siehe Abschnitt 4.3.4) anhand des Linienelements und der Kanten.

Bei beiden angeführten Methoden wird die kürzeste der berechneten Distanzen übernommen. In Abb. 4.10 ist zu sehen, dass es Elementkonfigurationen gibt, bei welchen es anhand dieser Methoden nicht möglich ist, die tatsächlich kürzeste Distanz zu berechnen.

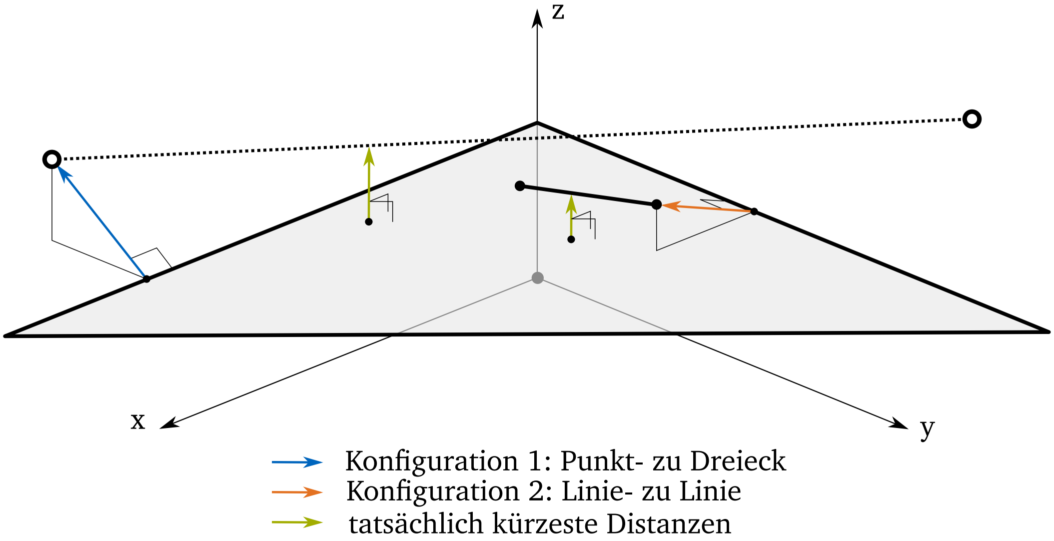


Abbildung 4.10: In *Konfiguration 1* wird die mögliche Anordnung für Methode 1 gezeigt, welche zu einer falschen Berechnung der kürzesten Distanz führt. *Konfiguration 2* zeigt den Fall für Methode 2.

Um auch im parallelen Fall ein korrektes Ergebnis gewährleisten zu können, wird bei der Implementierung in `broccoli` eine Kombination aus beiden Methoden verwendet. Es werden drei Linie- zu Linienberechnungen anhand der Dreieckskanten, sowie eine Punkt- zu Dreiecksberechnung anhand des Start- oder Endpunkts der Linie durchgeführt. Somit ist garantiert, dass zu jeder möglichen Konfiguration die tatsächlich minimale Distanz berechnet wird.
Es wird an dieser Stelle nochmals darauf hingewiesen, dass die Anzahl der parallelen Fälle zweier Elemente aufgrund der numerischen Berechnung sehr gering ist. Die hier verwendete, kostenintensivere Kombination in der Implementierung hat somit nur einen sehr geringen Einfluss auf die Gesamtberechnungsdauer eines Evaluierungszyklus.

Einheitliche Linien zu Dreiecks Distanzberechnung

Es gibt auch eine Möglichkeit, unabhängig von parallelem oder nicht-parallelem Fall, die minimale Distanz zu berechnen. Dazu werden beide Methoden aus Abschnitt 4.3.5 verwendet: zwei Punkt- zu Dreiecks-, sowie drei Linie- zu Linienberechnungen. Es wurde die Performance der beiden Varianten der Linie- zu Dreiecksberechnung in `broccoli` verglichen, Details folgen in Abschnitt 6.2.3. Dieser *einheitliche* Ansatz wird bei der bisherigen Lösung in *am2b* verwendet. Dabei ist keine korrekte Distanzberechnung gegeben, falls die Dreiecksfläche von der Linie durchstoßen wird.

4.3.6 Dreieck zu Dreieck Distanzberechnung

Zur Berechnung der Distanz zweier Dreiecke erfolgt eine Zerlegung der Kanten eines Dreieckselements in drei Linienelemente. Es werden drei Linien- zu Dreiecksberechnungen durchgeführt und daraus die minimale Distanz übernommen. Trotz der korrekten Linie- zu Dreiecksberechnung gibt es mögliche Anordnungen von zwei Dreieckselementen, welche zu einem falschen Ergebnis führen können. Es folgt eine Auflistung dieser Fälle inklusive einer Abschätzung über Auftreten und Einfluss des Fehlers.

Spezialfälle Dreieck- zu Dreiecksberechnung

Beim Funktionsaufruf zur Evaluierung von zwei Dreiecken wird immer das **erste** übergebene Dreieckselement anhand seiner Kanten zerlegt. Diese Abstrahierung hat zur Folge, dass die Information der zweiten Dimension dieses Elements verloren geht. Es wird während den drei Linien- zu Dreiecksberechnungen, die von den Kanten des ersten Elements eingeschlossene Fläche nicht mehr weiter berücksichtigt. Dadurch ergeben sich Spezialfälle, bei welchen die Reihenfolge der Elemente beim Funktionsaufruf Einfluss auf eine korrekte Auswertung haben.

In Abb. 4.11 ist in a) und b) dieselbe Konfigurationen dargestellt, jedoch wurde bei der Berechnung die Reihenfolge der Elemente im Funktionsaufruf vertauscht. Im Fall a) erfolgt die Zerlegung am größeren Dreieck. Da die Fläche nicht mehr berücksichtigt wird, erfolgt eine falsche Berechnung des Abstands. Wird im parallelen Fall hingegen das kleinere Dreieck zerlegt, wie im Fall b), so erfolgt immer eine korrekte Berechnung der minimalen Distanz.

Da die verwendete Linie- zu Dreiecksberechnung aus Abschnitt 4.3.5 im Gegensatz zur *einheitlichen* Berechnung erkennt ob eine Linie ein Dreieck durchstößt, werden Fälle wie in Abb. 4.12 a) zuverlässig erkannt. Dabei hat es keinen Einfluss, welches der Dreiecke zuerst

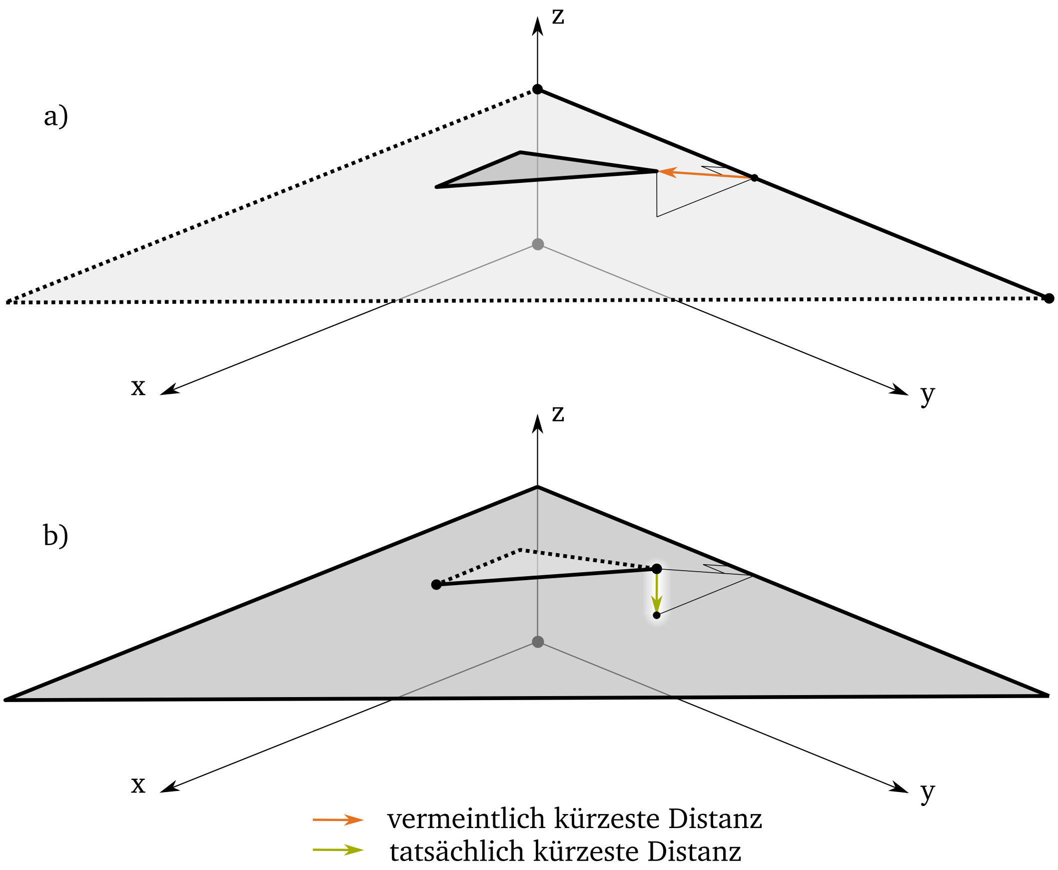


Abbildung 4.11: a) Mögliche falsche Berechnung bei paralleler Anordnung von zwei Dreieckselementen. b) Bei vertauschter Reihenfolge der Elemente beim Funktionsaufruf erfolgt eine korrekte Berechnung der minimalen Distanz.

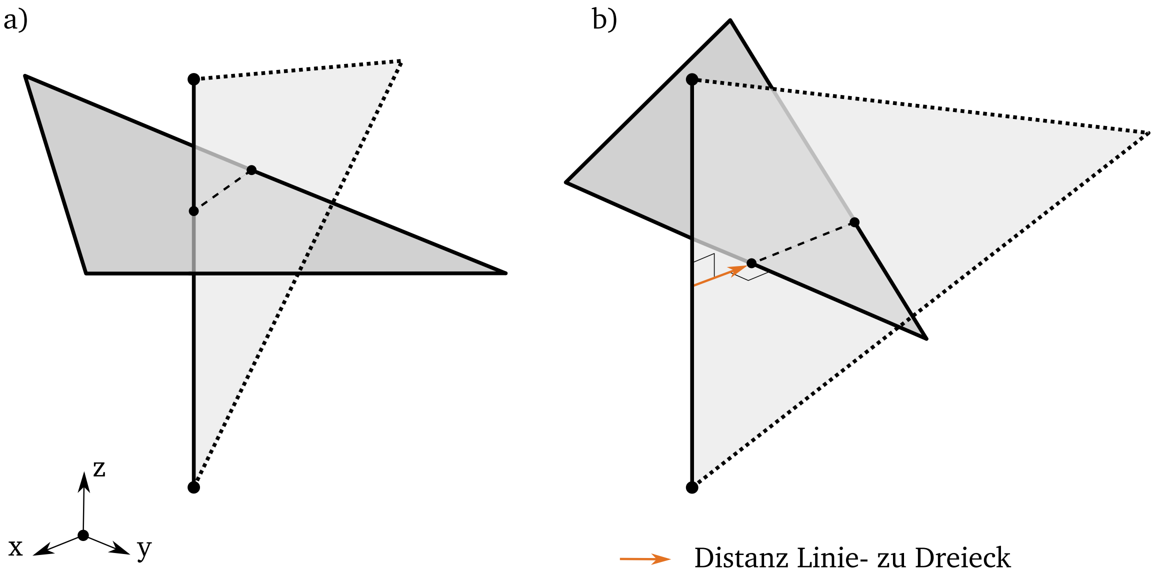


Abbildung 4.12: a) Mögliche Überschneidung von Dreieckselementen, welche mit der Linie- zu Dreiecksberechnung aus Abschnitt 4.3.5 richtig berechnet wird. b) Bei vorliegender Anordnung ist die korrekte Berechnung von der richtigen Reihenfolge der Elemente beim Funktionsaufruf abhängig.

der Funktion übergeben wird, da beide Elemente von der Kante des anderen durchstoßen werden. Mit dem Modul aus *am2b* würde dieser Fall zu einem falschen Ergebnis führen. Es wird aber darauf hingewiesen, dass es sich hierbei um einen sehr unwahrscheinlichen Fall handelt, da es bereits zu einem früheren Zeitpunkt zu einem Kontakt der beiden Elemente gekommen wäre.
Fall b) aus Abb. 4.12 zeigt eine mögliche Anordnung, welche zu einem falschen Ergebnis führen kann. Im dargestellten Beispiel würde die Verbindung der senkrechten Kante zum waagrechten Dreieck als geringste Distanz detektiert werden. Wären die Dreieckselemente beim Funktionsaufruf vertauscht übergeben worden, so wird anhand von einer der beiden durchstoßenden Kanten des waagrechten Dreiecks eine richtige Berechnung erfolgen.

Bei Eintritt eines Spezialfalls ist die Schwere des möglichen Fehlers abhängig von zwei Faktoren. Einerseits spielt das Größenverhältnis der zu vergleichenden Elemente eine wesentliche Rolle, wie in Abb. 4.11 a) zu erkennen ist. Je kleiner das „innenliegende“ Dreieck im Vergleich zum „äußeren“ ist, desto größer wird der Fehler in der Auswertung. Andererseits hat auch das Verhältnis von Radius des SSVs zur Größe des Grundelements einen Einfluss. Je größer dieser Wert ist, desto geringer wirkt sich die Länge des falschen Verbindungsvektors auf das Endergebnis aus.
Diese hier angeführten Spezialfälle, insbesondere jene in Abb. 4.11 und Abb. 4.12 a), spielen in der Berechnung einer Szene eine sehr untergeordnete Rolle. Um, unabhängig von der Anordnung der Elemente, die richtige Distanz bzw. einen minimalen Fehler in der Berechnung gewährleisten zu können, werden in Abschnitt 7.3 mögliche Ansätze zur Lösung dieses Problems dargelegt.

4.4 Swept-Sphere Segmente

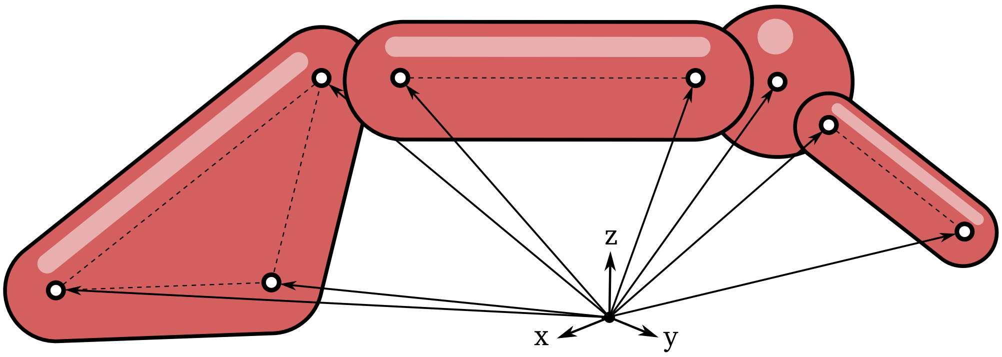


Abbildung 4.13: Beispielhaftes SSV Segment bestehend aus vier einzelnen Elementen. Die Position der Punkte zur Definition eines Elements sind alle auf das lokale Koordinatensystem des Segments bezogen.

Ein Zusammenschluss aus mehreren Elementen wird als Segment bezeichnet. Sie definieren somit die nächst höhere Ebene in der Distanzberechnung. Ein Segment besitzt ein lokales Koordinatensystem, in welchem die Position und Orientierung der einzelnen Elemente klar definiert ist. Segmente dienen zur Beschreibung einzelner Starrkörper, die Lage der Elemente innerhalb des Segments bleiben deshalb für eine Konfiguration konstant.

Ein Segment kann somit als eine reine Auflistung einzelner Elemente bezogen auf ein gemeinsames Koordinatensystem betrachtet werden.

Distanzberechnung von SSV Segmenten

Die minimale Distanz zwischen zwei Segmenten erfolgt durch Berechnung der Distanzen aller möglichen Elementpaarungen. Besteht ein Segment aus m und das andere aus n Elementen, so folgt der Rechenaufwand der Auswertung der zwei Segmente nach $\mathcal{O}(m \cdot n)$. Diese Berechnung liefert den geringsten Abstand zwischen den beiden Segmenten. Die Information zwischen welchen Elementen diese Distanz vorliegt ist auf dieser Ebene nicht von Interesse.

Eine Einschränkung der zu vergleichenden Elementpaare ist innerhalb der Segmente nicht notwendig, welches folgende Anschauung zeigen soll. Zur Beschreibung eines humanoiden Arms werden zwei Segmente verwendet: eines stellt den Ober- und das andere den Unterarm dar. Im Bereich des Ellbogens kann es je nach Modellierung der Segmente zu Überschneidungen der SSVs kommen. Um dennoch eine aussagekräftige Auswertung der Distanzen zu gewährleisten, müssen diese und vielleicht auch weitere Elemente von der Berechnung ausgenommen werden. Schlussendlich würde nur mehr die Distanz zwischen dem vordersten Teil des Unterarms mit dem Schulteransatz des Oberarms verglichen werden. Je nach Größe des Modells würde diese Problematik zu einer Vielzahl an Ausnahmen innerhalb der Berechnung führen.

Deshalb wird erst in der nächst höheren Ebene, den Szenen, festgelegt, welche Segmente miteinander verglichen werden. Zwei benachbarte Segmente wie im Beispiel des Arms werden nicht ausgewertet. Um eine Kollision in solchen Fällen zu vermeiden, ist die Bewegungsfreiheit des Ellbogengelenks durch ein Winkelintervall zwischen den beiden Segmenten beschränkt.

4.5 Swept-Sphere Szenen

Die Szene stellt die höchste Ebene in der Abstandsberechnung dar. Eine Szene beschreibt ein komplettes Set an Segmenten. In Bezug auf *Lola* besteht eine solche Szene aus dem Roboter selbst sowie aus Objekten in der Umgebung. Analog zu einem Segment mit den Elementen, ist eine Szene eine Auflistung von Segmenten. Durch Translation sowie Rotation lassen sich die Segmente innerhalb der Szene positionieren und orientieren.

Distanzberechnung einer SSV Szene

Wie in Abschnitt 4.4 erwähnt, erfolgt auf der Ebene der Szenen eine Auswahl an Segmentpaare, die zur Distanzberechnung herangezogen werden. Somit wird vermieden, dass zwei Segmente evaluiert werden, welche entweder starr in der Szene vorliegen und somit eine konstante Distanz aufweisen oder aufgrund kinematischer Randbedingungen nicht miteinander kollidieren können.

Zur Berechnung einer Szene werden alle angegebenen Segmentpaare evaluiert. Zu jeder Paarung wird die dazugehörige Distanz mit Angabe der verglichenen Segmente gespeichert.

Kapitel 5

Implementierung des neuen SSV-Moduls

Das neue SSV-Modul ist Teil der C++ Bibliothek `broccoli` (Beautiful Robot C++ Code Library) [18]. Die Bibliothek wird am *Lehrstuhl für Angewandte Mechanik* entwickelt und verfügt über allgemeine und oft angewendete Algorithmen und Funktionen in der Robotik. Im Gegensatz zu *am2b* ist sie nicht für den Betrieb eines Roboters konzipiert, sondern stellt verschiedenste unabhängige Module für z. B. Steuerung, Trajektorienberechnungen, Datenaustausch, Geometrie, u.v.m. zur Verfügung. Es wurde besonderes Hauptaugenmerk auf Echtzeit-Fähigkeit, Schnelligkeit und Wiederverwendbarkeit für Echtzeit-Betriebssysteme gelegt.

`broccoli` ist als Header-Only Bilbiothek aufgebaut. Das bedeutet, sowohl die Deklaration als auch die Definition von Methoden erfolgt ausschließlich in der Header-Datei. Nachteile dieser Implementierung sind, dass bei Änderungen innerhalb der Bibliothek alle Anwendungen, welche auf diese zugreifen, neu kompiliert werden müssen. Die Kompilierzeiten sind im Vergleich zu Header/Source Bibliotheken länger, da nicht nur ein Interface, sondern der komplette Code eingebunden wird. Es besteht auch eine höhere Gefahr von *code-bloat*[1] durch z. B. falsche Anwendung des Attributs *inline*.
Diesen Aspekten gegenüber stehen mehrere Vorteile. So verlangt die Verwendung effizienter Template Klassen (siehe Abschnitt 5.3.1) eine Definition in der Header-Datei. Des weiteren stehen dem Compiler bei einer Header-Only Bibliothek auch wesentlich mehr Möglichkeiten zur Optimierung des Codes zur Verfügung. Gerade in Bezug auf Echtzeitfähigkeit steht eine hohe Effizienz an vorderster Stelle. Ein weiterer Grund für die Verwendung dieses Aufbaus ist die einfache Einbindung der Bibliothek in ein Projekt.

5.1 Anwendung des Moduls

Die Bibliothek `broccoli` ist in *am2b* eingebunden und übernimmt zunehmend die Funktion der Module aus *utils*. Wie auch viele andere Module von `broccoli`, wird die Distanzberechnung ihre Anwendung im Betrieb des Roboters finden. In Algorithmus 5.1 wird in Form eines Pseudocodes im Groben der Programmablauf von *Lola* beschrieben. Er dient zur Einordnung, inwiefern die Information der Distanzen im Betrieb Verwendung findet.

---

[1] Aufgeblasener Code - zu langer oder langsamer Quellcode.

Bei der Berechnung der Gelenkwinkel durch Inverse Kinematik in Zeile 13; Algorithmus 5.1 kann es zum Auftreten eines sogenannten Nullraumes kommen. Liegen mehr Freiheitsgrade als Vorgaben vor, so gibt es theoretisch unendlich viele Gelenksanordnungen, welche zur selben Pose des Endeffektors führen. Aus diesen unendlich vielen Konfigurationen kann nun, durch eine zeitlich lokale Optimierung, unter Zuhilfenahme der berechneten Distanzen eine Variante gefunden werden, welche Kollisionen explizit vermeidet.

Algorithmus 5.1: Vereinfachter Programmablauf von *Lola*.

```
1  starte System
2  lade Topologie (kinematisches Modell des Roboters)
3  nimm Standardpose ein (siehe Abb. 1.1)
4  lade SSV-Geometrie
5  erzeuge Szene
6  Szene.setSegmentList()
7  Szene.setPairList()
8  for jeden Zeitschritt do
9      plane weiteren Bewegungsablauf
10     bestimme aktuelle Konfiguration des Roboters (Pose)
11     Szene.tranform() (Translation und Rotation von SSV Segmenten)
12     Szene.evaluate() (siehe Abb. 5.4)
13     berechne Inverse Kinematik unter Zuhilfenahme der Distanzen
14     führe Bewegung aus
15 end
```

5.2 *Eigen* - Lineare Algebra Bibliothek

Das Vorgänger-Modul zur Kollisionsberechnung aus *am2b* verwendete als Backend für lineare Algebra die Bibliothek *matvec*. Da diese den heutigen Standards an Effizienz, Flexibilität, Erweiterbarkeit und Modularität nicht mehr entspricht, wird für das neue Modul die Open-Source-Bibliothek *Eigen*[2] verwendet. *Eigen* überzeugt durch Metaprogrammierung mittels C++ Templates für hohe Effizienz, optimierte Matrix- und Vektoroperationen, geometrische Transformationen u.v.m. Des Weiteren verfügt *Eigen* eine sehr detaillierte Dokumentation und aufgrund der weiten Verbreitung über eine große Community, welche das Projekt aktiv weiterentwickelt. Um effizient Berechnungen durchführen zu können, ist *Eigen* in der Lage SIMD-Befehlssätze auszuführen. Es werden alle SSE[3]-, sowie AVX[4]-Standards bis zur letzten Erweiterung AVX512 von 2013 unterstützt.

5.2.1 SIMD - Single Instruction Multiple Data

Bei SIMD handelt es sich um eine Rechnerarchitektur für Vektorprozessoren. Diese arbeitet mit Daten in Form von Arrays oder Vektoren. Es ist somit möglich, einen Befehl (≙ Single Instruction) auf mehrere gleichzeitig eintreffende oder zur Verfügung stehende Eingangsströme

---

[2] http://eigen.tuxfamily.org

[3] Streaming SIMD Extensions

[4] Advanced Vector Extensions

($\hat{=}$ Multiple Data) auszuführen. Gerade in der Verarbeitung von Bild-, Ton und Videodaten findet diese Architektur Verwendung. Wird z. B. bei einem Bild der Kontrast verändert, oder bei einer Tonspur die Lautstärke, so wird auf eine große Datenmenge immer derselbe Befehl angewendet.

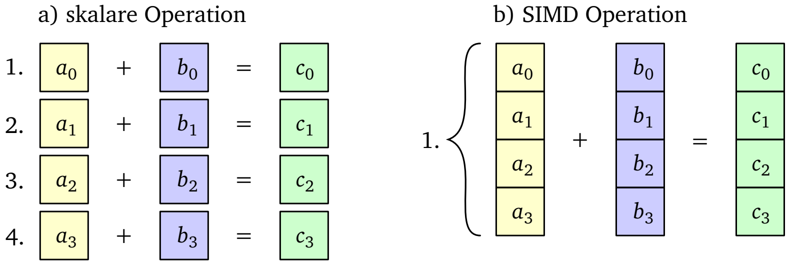


Abbildung 5.1: Anhand des Minimalbeispiels einer Addition von zwei Vektoren $\boldsymbol{a}^{\mathrm{T}}$ und $\boldsymbol{b}^{\mathrm{T}} \in \mathbb{R}^{1\times 4}$ wird der Unterschied zwischen einer skalaren und einer SIMD Operation gezeigt. Die Berechnung erfolgt mit *single precision* Gleitkommazahlen; dabei entspricht ein Kästchen einer Zahl mit 32 Bit, bzw. 4 Byte.

In Abb. 5.1 wird der Unterschied zwischen einer skalaren und einer SIMD Operation anhand der Vektorrechnung $\boldsymbol{a} + \boldsymbol{b} = \boldsymbol{c}$ dargestellt. Im skalaren Fall Abb. 5.1 a) müssen zur Berechnung von $\boldsymbol{c}$ vier Additionsbefehle nacheinander ausgeführt werden, während im Fall Abb. 5.1 b) nur ein Additionsbefehl für dasselbe Ergebnis notwendig ist. Da für eine gegebene Datenmenge weniger Befehle ausgeführt werden müssen, arbeiten SIMD Operationen somit wesentlicher effizienter als skalare.

5.2.2 Alignment Issues

Die Vektorisierung von *Eigen* kann nur angewendet werden, wenn die verwendeten Daten in durchgängigen 128-Bit, bzw. 16-Byte Paketen angelegt sind. Ein Paket entspricht dabei zwei *double-precision* Gleitkommazahlen. Deshalb wird bereits bei der Erstellung von Matrizen bzw. Vektoren auf ein *16-byte alignment* geachtet, sodass im späteren Verlauf keine Überprüfung auf durchgängigen Speicher erfolgen muss. Dies hat eine erhöhte Effizienz der Berechnungen zur Folge.
Bei Klassen mit *Eigen*-Objekten als Member muss der *new* Operator überladen werden. Dazu muss lediglich folgendes Macro hinzugefügt werden: `EIGEN_MAKE_ALIGNED_OPERATOR_NEW`.
Bei der Verwendung von STL Containern mit *Eigen*-Objekten, muss als template-Parameter *allocator type* die Übergabe eines `Eigen::aligned_allocator` erfolgen.

5.3 Modulstruktur

Der Programmaufbau erfolgt weitestgehend analog zum Aufbau der Geometrien aus Kapitel 4. Abbildung 5.2 zeigt die komplette Struktur des Moduls anhand eines UML[5] Klassendiagramms. Zur besseren Übersicht wird auf Details der jeweiligen Klassen verzichtet.
Auf unterster Ebene sind die Klassen der drei Grundelemente *SSVPointElement*, *SSVLineElement* und *SSVTriangleElement*, welche von der Basisklasse *SSVElement* abgeleitet sind. Die Verbindung einer Menge von Elementen stellt die nächst höhere Ebene der *SSVSegmente* dar. Distanzberechnungen erfolgen durch statische Methoden der Klasse *SSVDistanceEvaluator*.

---

[5] Unified Modeling Language https://www.uml-diagrams.org

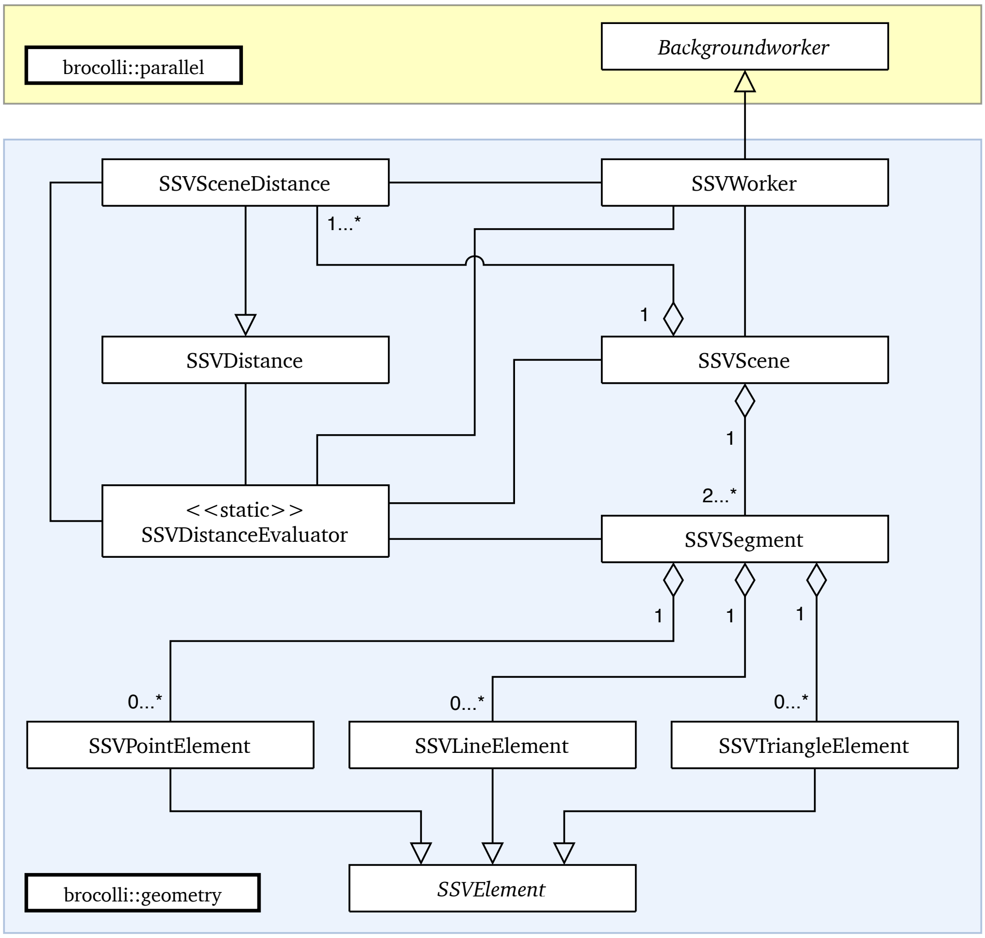


Abbildung 5.2: UML Klassendiagramm des neuen SSV-Moduls in `broccoli`. Das blaue Feld stellt das neue implementierte Modul dar. Dieses greift auf die abstrakte Klasse der *BackgroundWorker* aus dem Modul *parallel* (gelbes Feld) zurück.

Im Anwendungsfall erfolgt eine Berechnung auf Ebene der Segmente. Ergebnisse einer Auswertung werden in Objekten von *SSVDistance* abgespeichert.
Die höchste Ebene stellt eine Zusammenfassung mehrerer Segmente zu einer *SSVScene* dar. In ihr wird die komplette Szene beschrieben. Innerhalb dieser werden Paarungen, welche zur Distanzberechnung herangezogen werden, festgelegt. Für den Fall von Berechnungen auf mehreren Prozessorkernen wird die Auswertung von *SSVWorkern* durchgeführt. Diese sind von *BackgroundWorkern* aus dem `broccoli`-Modul *parallel* abgeleitet. Ergebnisse aller Paarungen einer Szenen werden in Objekten von *SSVSceneDistance* abgespeichert, welche von *SSVDistance* erbt. Es wird auch auf grundlegende Funktionen aus dem `broccoli`-Modul *core* zurückgegriffen.

Im folgenden werden die Grundzüge und Hintergründe der Klassen erklärt und auftretende Probleme in der Implementierung diskutiert. Für genaue Details zu einzelnen Attributen oder Methoden wird auf die Dokumentation des Moduls verwiesen (siehe Abschnitt 5.5).

5.3.1 SSVElement

Begonnen wurde die Implementierung mit der Klasse *SSVElement* auf unterster Ebene. Als Basisklasse enthält sie gemeinsame Attribute und Methoden der drei abgeleiteten Element-klassen. In dieser ersten Phase wurde sehr viel Wert auf die vollständige Beschreibung der Elemente und die grundlegende Struktur der Klasse gelegt. Im weiteren Verlauf des Modulaufbaus ergaben sich immer wieder neue Problemstellungen oder fehlende Funktionalitäten. In mehreren Iterationsschritten konnte schlussendlich eine zufriedenstellende und stabile Struktur auf unterster Ebene aufgebaut werden.

Ursprünglich war zur Speicherung von Elementeigenschaften eine Matrix vorgesehen, welche spaltenweise die Vektoren der Punkte, Kanten und im Fall des Dreieckelements die Matrix $\boldsymbol{S}$ und den Normalenvektor enthält. Durch Verwendung eines Klassen-Templates[6] ist je nachdem ob Punkt-, Linien- oder Dreieckselement die Anzahl der Spalten $m \in \{1, 3, 9\}$ fest vorgegeben. Der Grund für die Anlegung einer statischen anstatt einer dynamischen Matrix wird in Abschnitt 6.2.1 erläutert.

$$\texttt{m\_data} = \begin{pmatrix} \underbrace{\boldsymbol{p}_0 \dots \boldsymbol{p}_\mathrm{i}}_{\text{points}} & \underbrace{\boldsymbol{e}_0 \dots \boldsymbol{e}_\mathrm{j}}_{\text{edges}} & \boldsymbol{S} & \boldsymbol{n} \end{pmatrix} \in \mathbb{R}^{3 \times m} \tag{5.1}$$

Der Vorteil dieser Implementierung ist, dass es mittels Templates innerhalb der Basisklasse möglich ist Abfrage- bzw. Getter-Methoden für die Elementeigenschaften zu implementieren. Die Auswahl der Eigenschaften aus `m_data` erfolgt durch Befehle von *Eigen*. Diese erzeugen Objekte, welche Spalten, Zeilen oder Blöcke dieser Matrix beinhalten. Da der Aufruf dieser Befehle innerhalb der Getter erfolgt, handelt es sich hierbei um temporäre Objekte, welche nicht referenziert werden können. Es wäre somit nur möglich, Kopien der Vektoren oder Matrizen aus `m_data` zu übergeben, was sich negativ auf die Effizienz des Moduls auswirken würde.

Deshalb wurde in späterer Folge die Klasse so umgebaut, dass die Vektoren der Punkte und Kanten in Arrays aus dem STL Container[7] gespeichert werden. Die Größe der Arrays wird bei der Instanziierung durch Übergabe der Template-Parameter festgelegt. Die spezifischen Eigenschaften des Dreieckselements wurden in die abgeleitete Klasse *SSVTriangleElement* verschoben.

Die Verwendung von Arrays ermöglicht nun die Rückgabe von Referenzen. Durch diese notwendige Änderung ist es jedoch nicht mehr möglich einheitliche Getter in der Basisklasse zu implementieren. Stattdessen wurden rein abstrakte Methoden deklariert - die Definition erfolgt in den abgeleiteten Subklassen.

Die Verwendung des Moduls sieht vor, dass die relativen Positionen der Punkte eines Elements bereits bekannt sind. Mit Ausnahme des Radius, können die einzelnen Punkte nach Instanziierung nicht mehr verändert werden. Durch Skalierung, Rotation und Translation ist es aber möglich, das Element als Ganzes zu ändern.

Die geometrisch eindeutige Beschreibung der Elemente ist allein durch Angabe der Punkte und eines Radius gegeben (siehe Abschnitt 4.2). Trotzdem werden davon abgeleitete Größen, wie Kantenvektoren oder im Fall des Dreieckselements die Matrix $\boldsymbol{S}$ sowie der Normalenvektor, bereits bei der Erstellung eines Objektes berechnet und gespeichert. Somit kann bei der Berechnung der Distanz, ohne zusätzlichen Rechenaufwand, direkt auf diese Größen zugegriffen werden. Bei einer Skalierung oder Rotation des Elements werden auch die Hilfsgrößen berücksichtigt. Eine Translation hat nur eine Veränderung der Punkte zur Folge.

---

[6] Schablone oder Vorlage: ermöglichen eine generische Programmierung.

[7] *Standard Template Library* - https://en.cppreference.com/w/cpp/container

5.3.2 SSVPointElement, SSVLineElement und SSVTriangleElement

Die drei von *SSVElement* abgeleiteten Klassen beschreiben die einzelnen Primitive aus den Abschnitten 4.2.1 bis 4.2.3. Bei der Erstellung eines Elements erfolgt die Übergabe eines Templateparameters, welcher die Punkt- und Kantenanzahl angibt. Innerhalb der einzelnen Klassen erfolgt die Definition der rein virtuellen Methoden aus der Basisklasse.

Das Dreieckselement enthält zusätzlich die Attribute der Matrix $\boldsymbol{S}$ und einen Normalenvektor $\boldsymbol{n}$. Zur Berechnung dieser beiden Größen gibt es in *SSVTriangleElement* zusätzliche Funktionen.

5.3.3 SSVSegment

Durch eine individuelle Identifikationsnummer (kurz ID) ist ein Segment eindeutig bestimmt. Für jeden Elementtyp gibt es jeweils zwei Listen in Form von Vektoren des STL Containers - somit enthält ein Segment insgesamt sechs Listen: Drei dieser Listen mit den Präfix *initial* speichern die initiale Konfiguration des Segments. Dieser Zustand bezieht sich auf die Anordnung der Elemente im Ursprungszustand bei der Erstellung, bezogen auf das lokale Koordinatensystem des Segments. Anhand drei separater Funktionen zur Skalierung, Rotation und Translation ist es möglich, diesen Ursprungszustand zu verändern.
Im Gegensatz dazu ist auch eine Transformation des gesamten Segments möglich. Dabei werden die Elemente aus den initialen Listen transformiert und in jeweils drei Listen mit dem Präfix *transformed* gespeichert. Diese Struktur ist notwendig, da während der Anwendung des Moduls die Position und Orientierung der einzelnen Segmente in einem absoluten Koordinatensystem gegeben sind. Eine Konfiguration wird immer anhand der initialen Listen und des derzeitigen Zustandes des Roboters berechnet.
Berechnungen der Distanzen erfolgen immer mit den drei transformierten Listen eines Segments. Da jede Liste exakt einen Elementtyp beinhaltet, müssen bei einer Evaluierung von zwei Segmenten $3^2 = 9$ mögliche Listenkombinationen miteinander verglichen werden. Ergebnis einer Auswertung ist die kürzeste Distanz zwischen den beiden Segmenten.

Da Klassen Zugriff von außerhalb auf die Elementlisten eines Segments benötigen, muss sichergestellt sein, dass die Listen zum Zeitpunkt der Evaluation immer dem aktuellsten Stand entsprechen. Das bedeutet es darf keine Transformation übergeben worden sein, welche noch nicht ausgeführt wurde. Wird eine Transformationsmatrix dem Segment übergeben, so wird ein Update-Flag[8] gesetzt. Erst nach Ausführung der Transformation wird das Flag wieder gelöscht. Bei Aufruf eines Getters für eine Liste erfolgt im ersten Schritt die Überprüfung des Flags. Hat noch keine Transformation stattgefunden, so ist dieses noch gesetzt und es wird ein Runtime-Error[9] geworfen.

5.3.4 SSVDistanceEvaluator

Diese Klasse stellt alle zur Distanzberechnung benötigten Funktionen zur Verfügung. Durch Aufruf der mehrfach überladenen Funktion `evaluate()` erfolgt eine Distanzberechnung. Als Parameter werden dieser Funktion zwei auszuwertende Segmente bzw. Elemente und eine

---

[8] Indikator zur Markierung eines Zustandes. Kann gesetzt, gelöscht und geprüft werden.

[9] Fehler, welcher nur während der Laufzeit eines Programms auftreten und zu einem unvorhersehbaren Verhalten führen kann.

Instanz der Klasse *SSVDistance* übergeben. Abhängig von den übergebenen Elementtypen erfolgt innerhalb der Funktion die Zerlegung der Elemente in ihre Bestandteile wie Punkte, Kanten, u.s.w. und damit der Aufruf einer sogenannten *Core-Function*. Diese Funktionen enthalten die eigentlichen Algorithmen, welche die kürzeste Distanz ermitteln (siehe Anhang A). Die mathematischen Hintergründe, welche den Algorithmen zugrunde liegen, sind in Abschnitt 4.3 angeführt.

Bei *SSVDistanceEvaluator* handelt es sich um eine statische Klasse, d.h. die Methoden können ohne Instanz aufgerufen werden. Das Ergebnis einer Berechnung des *SSVDistanceEvaluator* ist nicht der Wert der kürzesten Distanz, sondern der Start- und Endpunkt des Verbindungsvektors zwischen den Elementen, welcher die geringste Länge aufweist.

5.3.5 SSVDistance und SSVSceneDistance

Das von der `evaluate()`-Funktion des *SSVDistanceEvaluator* berechnete Ergebnis wird in einer Instanz der Klasse *SSVDistance* gespeichert. Mit den beiden übergebenen Punkten und den Radien der Elemente erfolgt die Berechnung des Verbindungsvektors und dessen Länge ($\widehat{=}$ minimalen Distanz) anhand Glg. (4.8).
Die Klasse *SSVSceneDistance* erbt von *SSVDistance*. Bei einer Berechnung innerhalb einer Szene wird zusätzlich zur Distanz zwischen zwei Segmenten auch die zugehörige Segmentpaarung mitgespeichert. Wie in Abschnitt 5.1 erwähnt, werden die Ergebnisse einer Evaluierung für weitere Schritte im Betrieb des Roboters benötigt.

Richtung des Verbindungsvektors

Wie in Abschnitt 4.3 erwähnt, sind die Distanzberechnungen kommutativ, weshalb alle neun möglichen Paarungen anhand von fünf Algorithmen berechnet werden können (siehe Tab. 4.1). Wird z.B. `evaluate(LSS, PSS, Distance)` der Klasse *SSVDistanceEvaluator* aufgerufen, so erfolgt innerhalb dieser Funktion der Aufruf von `evaluate(PSS, LSS, Distance)`. Dabei wurden die Übergabeparameter der Elemente im Funktionsaufruf vertauscht. Erst in dieser Funktion erfolgt die oben genannte Zerlegung und Weitergabe an eine *Core-Function*. Aufgrund dieser Vorgehensweise entspricht die Zuordnung von Start- und Endpunkt nicht mehr der ursprünglich aufgerufenen Reihenfolge der Elemente. Da die korrekte Richtung des Verbindungsvektors für die Berechnung der inversen Kinematik zwingend erforderlich ist, werden im Fall einer solchen „vertauschten" Evaluierung sämtliche in *SSVDistance* gespeicherten Ergebnisse durch den Aufruf von `swap()` richtiggestellt.

5.3.6 SSVScene

Eine Szene besteht im Wesentlichen aus drei wichtigen Listen:

- **Segmentliste:** beinhaltet alle in dieser Szene enthaltenen Segmente
- **Paarliste:** enthält Paare der Identifikationsnummern jener Segmente, welche innerhalb der Szene miteinander verglichen werden
- **Distanzliste:** speichert zu jedem Eintrag aus der Paarliste die dazugehörigen Ergebnisse einer Auswertung

Wie bereits in Abschnitt 4.5 erwähnt, ist eine Evaluierung aller möglichen Segmentpaarungen in den wenigsten Fällen sinnvoll. Durch Anlegen der Paarliste ist es dem Nutzer möglich, jene Paare anzugeben, deren Distanz für Berechnungen im weiteren Verlauf von Interesse sind.
Der Aufruf eines Segments erfolgt immer anhand seiner ID (siehe Abb. 5.3). Um zeitaufwendige Suchdurchläufe innerhalb der Segmentliste zu vermeiden, wird parallel zur Erstellung der Segmentliste eine Map angelegt. Diese sogenannte *unordered_map* ist ein assoziativer Container, welcher jeder ID eines Segments den Index innerhalb der Segmentliste zuordnet. Diese Art der Map zeichnet sich dadurch aus, dass Suchen eines Eintrags nahezu immer mit $\mathcal{O}(1)$ Komplexität erfolgen.[10]

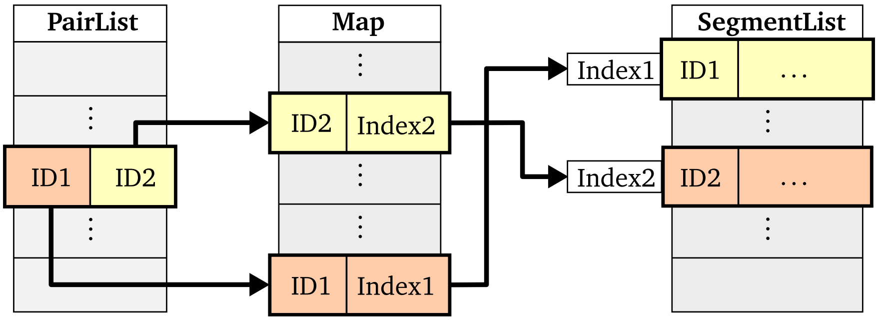


Abbildung 5.3: Zugriff auf die Segmentliste durch Verknüpfung der IDs der Paarliste mittels Map.

Der Zugriff auf ein Segment mittels ID bietet den Vorteil, dass Paarliste und Segmentliste „entkoppelt“ sind. Werden Einträge aus der Segmentliste entfernt, so ändert sich die Reihung und damit der Index der Segmente. Durch ein Update der Map ist die korrekte Verknüpfung von IDs und Indizes wieder gewährleistet - die Paarliste bleibt dabei unberührt.
Da Segment- und Paarliste unabhängig voneinander verändert werden können, besteht bei Unachtsamkeiten in der Bedienung die Möglichkeit von unzulässigen Paarungen. Aus diesem Grund wird im Debug-Modus vor der Evaluierung einer Szene ein Abgleich der Paarliste mit der Map durchgeführt. Sind IDs der Paarliste nicht in der Map enthalten, so erfolgt der Abbruch des Programms.
Des Weiteres muss auch darauf geachtet werden, dass beim Entfernen eines Segments aus der Szene alle Paare, welche dieses Segment beinhaltet, ebenfalls gelöscht werden. Werden neue Paare in der Liste hinzugefügt, so wird diese ebenfalls im Debug-Mode auf unzulässige Eingaben überprüft. Paare mit gleicher ID oder Duplikate werden erkannt und das Programm wird an dieser Stelle abgebrochen.

Um die Berechnungszeiten zu minimieren, besteht auf Ebene der Szene die Möglichkeit Berechnungen zu parallelisieren (siehe Abschnitt 5.4). Beim Aufruf der `evaluate()`-Funktion erfolgt die Übergabe der Anzahl der sogenannten Worker ($\hat{=}$ Anzahl der Threads). Die Berechnungen werden auf diese Worker aufgeteilt und parallel ausgeführt.
Für den Fall, dass keine Worker angefordert werden, erfolgt die Berechnung sequentiell direkt in der `evaluate()`-Funktion von *SSVScene*. Bei nur einem Worker übernimmt ein Thread die Auswertung, was nahezu derselben Berechnung wie in Fall von keinem Worker entspricht (weitere Details siehe Abschnitt 6.2.5).

---

[10] https://en.cppreference.com/w/cpp/container/unordered_map/operator_at

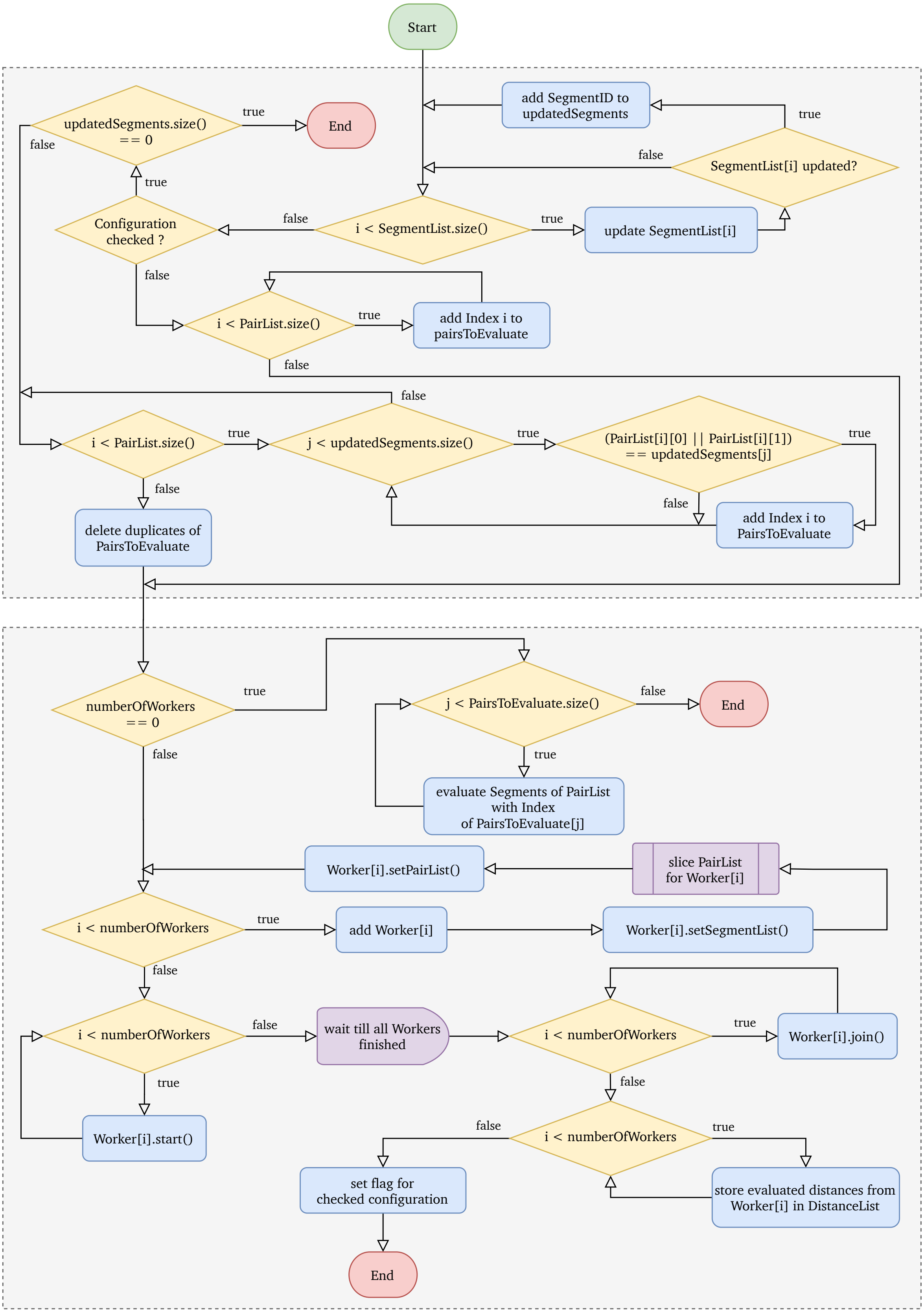


Abbildung 5.4: Flowchart des Ablaufs der `evaluate()`-Funktion der *SSVScene*. Im ersten Teil erfolgt die Auswahl der zu evaluierenden Paare, im zweiten die Evaluierung der Distanzen abhängig von der vorgegebenen Anzahl an Workern. Die Variablen i und j sind Laufvariablen. Für eine bessere Übersicht wurden auf die Operationen des Initialisierung und Inkrementierung der Variablen verzichtet.

Die Evaluierung einer Szene beinhaltet die Berechnung und Speicherung der geringsten Distanz von allen angegebenen Paaren. Dabei wurden zwei Herangehensweisen ausgearbeitet:

Einheitliche Evaluierung

Diese sieht bei jeder Ausführung die Auswertung aller Segmentpaarungen der Paarliste vor. Es werden Position und Orientierung aller Segmente der Szene aktualisiert. Im Anschluss erfolgt, abhängig der gewünschten Anzahl an Workern, eine gleichmäßige Aufteilung der zu berechnenden Segmentpaare und deren Auswertung. Die Möglichkeit, dass seit der letzten Evaluierung nur wenige oder gar keine Segmente transformiert wurden, wird bei dieser Herangehensweise nicht berücksichtigt.

Selektive Evaluierung

Es ist möglich, dass in einer Szene innerhalb eines Berechnungszyklus nur ein Teil aller Segmente transformiert wird. Somit würden sich viele der bereits berechneten Distanzen nicht verändern. Zur Vermeidung redundanter Auswertungen wurde ein Schema entwickelt (siehe Abb. 5.4), welches nur jene Paare zur weiteren Berechnung zulässt, von denen mindestens ein Segment aktualisiert wurde. Die Einsparung an Rechenzeit durch Reduktion der zu evaluierenden Paare überwiegt deutlich den Mehraufwand der Detektion und Selektion der aktualisierten Segmente. Aus diesem Grund wurde für die Implementierung in `broccoli` diese Herangehensweise verwendet.

5.3.7 SSVWorker

Diese Klasse ist von *BackgroundWorker* aus dem Modul *parallel* abgeleitet, welche die Berechnung auf mehreren Threads ermöglicht. Einem Worker wird die komplette Segmentliste sowie die aufgeteilte Liste der zu berechnenden Segmentpaare übergeben (siehe Abb. 5.5). Innerhalb der Worker erfolgt in der `execute()`-Funktion die eigentliche Distanzberechnung der zugeteilten Paare.

Bei der Verwendung von Workern gibt es Besonderheiten in der Implementierung. So muss bei Gettern und Settern sichergestellt sein, dass kein gleichzeitiger Lese-/Schreib- bzw. Schreib-/Schreibzugriff auf Daten erfolgen kann. Auch innerhalb der *SSVScene* muss eine korrekte Anwendung der Worker garantiert werden. Nähere Details sind in der Dokumentation von `broccoli` zu finden [18].

5.4 Single- vs. Multicore Processing

Numerische Berechnungen werden von einem Mikroprozessor durchgeführt. Um einen Prozessor schneller zu machen, wurde lange Zeit rein auf die Erhöhung der Taktfrequenz gesetzt, wodurch mehr Berechnungsschritte in kürzerer Zeit durchgeführt werden können. Aufgrund der zur Taktfrequenz proportionalen, höheren Leistungsaufnahme (mit verbundener hoher Wärmeentwicklung), stößt die Technik diesbezüglich an die Grenzen des Machbaren. Anstatt höherer Taktraten bei einem Kern, setzt man deshalb auf mehrere Rechenkerne mit jeweils

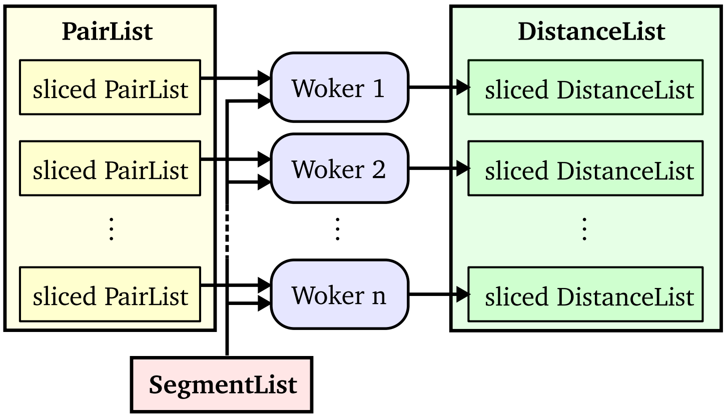


Abbildung 5.5: Schematische Darstellung der Funktion der *SSVWorker*.

niedrigerer Taktung. So gelingt eine Steigerung der Rechenleistung bei gleichzeitig geringerer Leistungsaufnahme.
Um mehrere Prozessorkerne effektiv nutzen zu können, müssen die Programme, Aufgabenstellungen bzw. Probleme welche berechnet bzw. bearbeitet werden, eine parallele Ausführung ermöglichen. Bei sehr einfachen Anwendungen ist keine Berechnung auf mehreren Kernen notwendig, bzw. gibt es Anwendungen, welche keine Möglichkeit bieten, die Aufgabe in kleinere Teilaufgaben zu zerlegen.
Theoretisch wäre bei n Kernen eine ver-n-fachung der Rechenleistung möglich, in der Praxis ist dies aber stark vom Parallelisierungsgrad der ausgeführten Programme und des verwendeten Betriebssystems abhängig. Besonders bei Prozessoren mit mehr wie vier Kernen ist eine volle Ausschöpfung der Kapazitäten nur bei wenigen Anwendungen gegeben.

Die Berechnung einer Szene eignet sich sehr gut zur Ausführung auf mehreren Kernen. Die geforderten Abstände der Segmentpaare lassen sich unabhängig voneinander berechnen und können entsprechend der geforderten Anzahl an Workern gleichmäßig aufgeteilt werden (siehe Abb. 5.5).

Multithreading

Es gibt die Unterscheidung von hardware- und softwareseitigem Multithreading:

- Softwareseitiges Multithreading beschreibt die Aufteilung eines Programms in mehrere gleichzeitig abzuarbeitende Stränge. Stehen mehrere Kerne zur Verfügung, so erfolgt die Abarbeitung eines Stranges bzw. Threads im Idealfall durch einen Kern.
- Hardwareseitiges Multithreading hingegen beschreibt die Fähigkeit eines Prozessorkerns, zwischen mehreren Threads in der Ausführung wechseln zu können. Wartet z. B. gerade ein Thread auf einen Speicherzugriff, so werden die freien Ressourcen des Prozessors zur Ausführung eines anderen Threads genutzt. Sind die beiden Abläufe voneinander unabhängig, so können durch diese Methode große Performanceverbesserungen erzielt werden. Im Fall schlecht parallelisierbarer Algorithmen kann es durch Overhead[11] beim Kontextwechel im schlechtesten Fall zur Reduktion der Gesamtsystemleistung kommen.

---

[11] Daten, welche nicht primär als Nutzdaten zur Verwendung stehen. Zusatzinformationen bei einer Übermittlung oder Speicherung.

5.5 Dokumentation

Ein wesentlicher Punkt bei der Neuimplementierung war die Dokumentation des Codes mittels *Doxygen*[12]. Diese Programmierschnittstelle wird seit Beginn der Entwicklung von `broccoli` angewendet. Die Dokumentation erfolgt in Form von Kommentaren innerhalb des Quellcodes. Beim Build-Prozess der Software erfolgt dann parallel die Generierung der Dokumentation als HTML-Datei. Durch diese enge Vernetzung kann eine zum Code konsistente Dokumentation gewährleistet werden.
Doxygen beinhaltet verschiedenste Features wie die Verwendung des LaTeX-Befehlsatzes, die Einbindung von Codeausschnitten oder Abbildungen, Visualisierungen von Beziehungen zwischen Klassen u.v.m. Ein bekanntes Beispiel für die Verwendung von *Doxygen* stellt die Bibliothek *Eigen* dar.

Im neuen Modul erfolgt je Klasse eine kurze Beschreibung des Inhalts und wie sie angewendet wird. Es werden sowohl die Attribute, als auch die Methoden erklärt. Bei letzteren erfolgt eine Auflistung der Übergabeparameter sowie Rückgabewerte mit kurzer Beschreibung.
Ein eigener Befehlssatz ermöglicht eine saubere Strukturierung und Darstellung. So erfolgt die Dokumentation von z. B. möglichen Fehlerquellen oder wichtigen Hinweisen durch vorangestellte Befehle wie `\warning` oder `\note`. Diese werden dann in der Dokumentation durch entsprechende Farbwahl und Formatierung hervorgehoben.

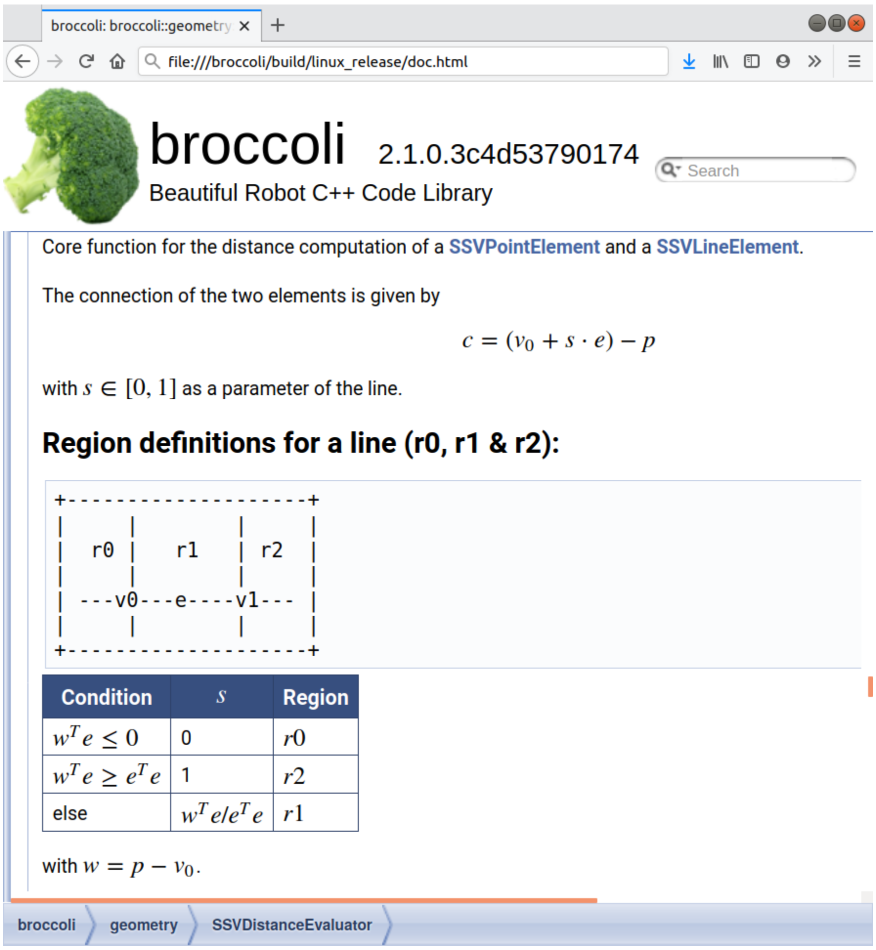
Des Weiteren wurde auch eine Kommentierung des Codes vorgenommen, welche nicht in der Generierung der Dokumentation mit einbezogen wird. Diese Kommentare dienen zum besseren Verständnis von speziellen Implementierungen und der Übersichtlichkeit des Quellcodes.

Codeausschnitt 5.1: Dokumentation der Funktion `evaluatePointToLine()` mittels *Doxygen*.

```
//! Core function for the distance computation of a SSVPointElement
//! and a SSVLineElement
/*!
* The connection of the two elements is given by
* \f[ c = (v_{0} + s \cdot e) - p\f],
* with \f$ s \in [0, 1]\f$  as a parameter of the line.
*
* Region definitions for a line (r0, r1 & r2):
* ------------------------------------------
* \verbatim
+--------------------+
|     |        |     |
|  r0 |   r1   | r2  |
|     |        |     |
| ---v0---e----v1--- |
|     |        |     |
+--------------------+
\endverbatim
*
* Condition                    | \f$ s \f$                  | Region
* ---------                    | ---------                  | ------
* \f$ w^{T} e \leq 0\f$        | 0                          | \f$ r0 \f$
* \f$ w^{T} e \geq e^{T} e\f$  | 1                          | \f$ r2 \f$
* else                         | \f$ w^{T} e / e^{T} e\f$   | \f$ r1 \f$
*/
```

---

[12] https://www.doxygen.nl



broccoli 2.1.0.3c4d53790174

Beautiful Robot C++ Code Library

Core function for the distance computation of a **SSVPointElement** and a **SSVLineElement**.

The connection of the two elements is given by

$$c = (v_0 + s \cdot e) - p$$

with $s \in [0, 1]$ as a parameter of the line.

Region definitions for a line (r0, r1 & r2):

```
+--------------------+
|     |        |     |
|  r0 |   r1   | r2  |
|     |        |     |
| ---v0---e----v1--- |
|     |        |     |
+--------------------+
```

Condition	s	Region
$w^T e \leq 0$	0	$r0$
$w^T e \geq e^T e$	1	$r2$
else	$w^T e / e^T e$	$r1$

with $w = p - v_0$.

Abbildung 5.6: Generierte Dokumentation aus dem Quelltext von Codeausschnitt 5.1.

Kapitel 6

Laufzeit- und Unit-Tests

Da das Modul für Echtzeit-Berechnungen auf dem on-board Rechner von *Lola* verwendet wird, muss die Implementierung so effizient wie möglich sein. Es wurden deshalb im Verlauf des Modulaufbaus mehrere Performance Tests verschiedener Funktionen und Algorithmen durchgeführt, um eine schnellstmögliche Implementierung zu erlangen. Alle der nachfolgenden Laufzeitmessungen wurden auf einem 64-bit Ubuntu 18.04 System, Kernel Version 5.4.0-52-generic, auf einer Intel® Core™ i7-2630QM CPU mit 2.00GHz durchgeführt. Die CPU verfügt über vier physikalische Kerne und acht Threads. Es wird der Compiler *Clang* Version 6.0.0 verwendet. Ausgeführt wird das Modul auf *Lola* auf einem 64-bit QNX® Neutrino® 7.0 RTOS[1] unter Verwendung einer Intel® Core™ i7-8700K CPU mit 3.60GHz auf sechs Kernen.

6.1 Allgemeiner Testaufbau

Codeausschnitt 6.1: Strukur eines Performance-Tests.

```
Eigen::Matrix<double, iterOuterLoop, 1> time;

for (int i = 0; i < iterOuterLoop; i++) {
   auto startTime = core::Time::currentTime();

   for (int j = 0; j < iterInnerLoop; j++) {
      // ----------------------------
      // Aufruf des zu testenden Codes
      // ----------------------------
   }

   auto stopTime = core::Time::currentTime();
   auto duration = stopTime - startTime;

   time(i) = duration.toDouble();
}

double result = time.minCoeff();
```

Es handelt sich bei Ubuntu 18.04 nicht um ein echtzeitfähiges System, deshalb wurden für eine repräsentative Aussage der Ergebnisse bestimmte Vorkehrungen getroffen. Um die Rechenleistung nicht zu beeinträchtigen, wurde beim Ausführen der Tests darauf geachtet, dass

---

[1] Real-Time Operating System

die Anzahl der im Hintergrund laufenden Anwendungen auf ein Minimum beschränkt ist.
In Codeausschnitt 6.1 wird allgemein die Implementierung eines Performance Tests gezeigt. Er besteht aus zwei ineinander verschachtelten for-Schleifen. Zur Laufzeitmessung wird die Dauer eines Durchlaufs der inneren for-Schleife aufgezeichnet und in einem Vektor gespeichert. Dieser Vorgang wird mehrere Male wiederholt. Das entscheidende Ergebnis ist die minimal benötigte Zeit für den Durchlauf der inneren for-Schleife. Sie gibt die maximal mögliche Geschwindigkeit der Funktion an. Größen wie der Mittelwert oder die Standardabweichung sind aufgrund nicht kontrollierbarer Hintergrundberechnungen anderer Anwendungen für eine repräsentative Aussage nicht geeignet. Durch diesen Aufbau ist die Wahrscheinlichkeit sehr groß, dass innerhalb einer Zeitnehmung die Berechnung ohne Einfluss anderer Anwendungen ausgeführt werden kann.
Mitunter ist es notwendig, die Ergebnisse der getesteten Funktion innerhalb der for-Schleife in einem sogenannten Dummy[2] aufzusummieren, um eine nicht erwünschte Optimierung durch den Compiler zu unterdrücken.

Allgemein ist darauf hinzuweisen, dass anhand der nachfolgenden Ergebnisse nur beschränkt Aussagen zum Laufzeitverhalten auf einem echtzeitfähigen System getätigt werden können. Die absoluten Zeiten dienen zum Vergleich unterschiedlicher Implementierungen und Ansätze. Zur besseren Vergleichbarkeit werden deshalb relative Unterschiede der Tests mit angeführt.

6.2 Evaluierung der Testergebnisse

6.2.1 Dynamische vs. statische Speicherallokation

Tabelle 6.1: Zeitdauer für 100000 Speicherallokationen für statische und dynamische Matrizen durch *Eigen*.

Allokation	abs. Laufzeit (t_{abs}) [μs]	rel. Laufzeit (t_{rel}) [$-$]
dynamisch	1609.48	1.000
statisch	1.414	0.000878

Da zu Beginn der genaue Aufbau der Implementierung nicht festgelegt war, wurden verschiedenste Varianten in Betracht gezogen. Wie in Abschnitt 4.2 erwähnt, besitzen die drei Elemente verschieden viele Eigenschaften und folglich wird je nach Element eine unterschiedlich große Matrix zur Speicherung benötigt (siehe Glg. (5.1)). Es standen anfangs zwei Varianten zur Auswahl: Einerseits statisches Anlegen einer Matrix mit Übergabe von Template-Parametern oder dynamisches Ändern der Matrix an das jeweilige Element.
Die relative Laufzeit wurde mit der absoluten Zeit der dynamischen Allokation als Referenz errechnet, $t_{\text{rel, i}} = \frac{t_{\text{abs, i}}}{t_{\text{abs, dyn.}}}$.
Der Test zeigt eine um drei Größenordnungen schnellere Speicherallokation im statischen Fall, verglichen zum dynamischen. Deshalb wurde auch in der ersten Implementierung eine statische Erzeugung der Matrix angewendet.

---

[2] Variable, welche keinen direkten Nutzen innerhalb der Anwendung hat.

6.2.2 Distanzberechnung Punkt zu Eckpunkt

Tabelle 6.2: Punkt- zu Dreiecksberechnungen an einem stumpfwinkligen Dreieck. *Variante 1* ist jene aus Abschnitt 4.3.3, *Variante 2* die pragmatische Herangehensweise aus Abschnitt 4.3.3.

Sektion	Testfälle je Sektion	*Variante 1* abs. Laufzeit ($t_{\text{abs, Var1}}$) [ns]	*Variante 2* abs. Laufzeit ($t_{\text{abs, Var2}}$) [ns]	rel. Laufzeit (t_{rel}) [−]
0	4			
1	6	20.1512	29.3587	0.686
2	2			
0	6			
1	-	34.9296	41.7174	0.830
2	6			
0	-			
1	12	19.9064	29.1320	0.683
2	-			

Wie in Abschnitt 4.3.3 beschrieben, gibt es bei der Punkt- zu Dreiecksberechnung zwei Möglichkeiten, um auch im Fall eines stumpfwinkligen Dreiecks die korrekte minimale Distanz zu berechnen. Der Test wurde für zwölf Distanzberechnungen durchgeführt, wobei die Anzahl der Fälle je Sektion variiert wurde.

Die relative Laufzeit ist definiert durch $t_{\text{rel}} = \frac{t_{\text{abs, Var1}}}{t_{\text{abs, Var2}}}$. Es ist zu sehen, dass in allen drei Tests *Variante 1* mit Unterscheidung der Sektionen um mindestens 17% schneller ist. Deshalb wird diese Variante in der Implementierung verwendet.

6.2.3 Distanzberechnung Linien- zu Dreieckselement

Tabelle 6.3: Laufzeitunterschiede der Linie- zu Dreiecksberechnung zwischen beiden möglichen Varianten. *Variante 1* entspricht dem einheitlichen Ansatz aus Abschnitt 4.3.5, *Variante 2* der Fallunterscheidung bei nichtparallelen Elementen aus Abschnitt 4.3.5.

Implementierung	abs. Laufzeit (t_{abs}) [ns]	rel. Laufzeit (t_{rel}) [−]
Variante 1	119.02	1.000
Variante 2	82.49	0.693

In Abschnitt 4.3.5 wurden die beiden Varianten zur Berechnung der Distanz zwischen den beiden Elementen gezeigt. In diesem Test wurde die Performance der beiden Möglichkeiten verglichen.

Die relative Laufzeit wurde mit der absoluten Zeit der einheitlichen Variante als Referenz errechnet, $t_{\text{rel}} = \frac{t_{\text{abs, i}}}{t_{\text{abs, Var1}}}$. Die komplexe Implementierung des Linie- zu Dreiecksalgorithmus weist dabei eine um über 30% schnellere Berechnung auf, weshalb sie auch im neuen Modul verwendet wird. Die Erstellung und Berechnung von Hilfsgrößen in *Variante 2* nehmen am Anfang der Evaluierung mehr Zeit in Anspruch, dafür erfolgt durch die anschließende Fallunterscheidung eine schnelle Eingrenzung auf nur wenige weiterführende Evaluierungen anhand von Elementabstraktionen. *Variante 1* hingegen benötigt keine Hilfsgrößen, jedoch müssen durch Verwendung von mehreren Punkt- zu Dreiecks- und Linie- zu Linienberechnungen wesentlich mehr Evaluierungen durchgeführt werden, was zu einer schlechteren Performance dieser Variante führt.

6.2.4 am2b- vs. broccoli-Modul

Tabelle 6.4: Laufzeittests aller neun möglichen Paarungen (siehe Tab. 4.1) mit den Distanzberechnungsmodulen von *am2b* und `broccoli`.

Berechnung von	*am2b* abs. Laufzeit ($t_{\text{abs, am2b}}$) [ns]	*am2b* rel. Laufzeit ($t_{\text{rel,am2b}}$) [–]	`broccoli` abs. Laufzeit ($t_{\text{abs, broccoli}}$) [ns]	`broccoli` rel. Laufzeit ($t_{\text{rel,broccoli}}$) [–]	rel. Laufzeit (t_{rel}) [–]
PP	8.52430	1.000	7.25800	1.000	0.851
LP	15.3277	1.798	9.49350	1.308	0.619
PL	15.7330	1.846	9.51970	1.312	0.605
LL	21.2802	2.496	16.9432	2.334	0.796
TP	18.6788	2.191	12.8067	1.765	0.686
PT	17.6437	2.070	12.8055	1.764	0.726
TL	86.5742	10.156	74.2182	10.226	0.857
LT	86.2693	10.120	77.0399	10.615	0.893
TT	224.4670	26.333	212.3200	29.253	0.946

Mit einer der Gründe für die Erneuerung des Moduls ist eine Laufzeitverbesserung in der Distanzberechnung. Dazu wurde ein bereits vorhandener Performancetest aus *am2b* als Vorlage genutzt und adaptiert. Zur Nutzung des *am2b*-Moduls waren kleine Änderungen in den Klassen nötig, welche aber keinen Einfluss auf die Evaluierung haben. Dieselbe Teststruktur mit den exakt gleichen Testfällen wurde auch für das Modul aus `broccoli` verwendet. Die Tests beider Module wurden sequentiell auf einem Prozessorkern durchgeführt.

Die relativen Laufzeiten $t_{\text{rel, am2b, i}}$ und $t_{\text{rel, broccoli}}$ beziehen sich immer auf die absoluten Laufzeiten von einer Punkt- zu Punktberechnung des jeweiligen Moduls; so z. B. $t_{\text{rel, am2b}} = \frac{t_{\text{abs, am2b, i}}}{t_{\text{abs, am2b, PP}}}$. Die relative Laufzeit in der letzten Spalte vergleicht die absoluten Laufzeiten der beiden Module je Berechnungsfall miteinander: $t_{\text{rel, i}} = \frac{t_{\text{abs, broccoli, i}}}{t_{\text{abs, am2b, i}}}$.

Der Test zeigt, dass die Implementierung in `broccoli` unter Verwendung von *Eigen* in jedem Testfall eine bessere Performance aufweist. Auch die relativen Laufzeiten weisen bis zum Fall PT bessere Werte im Vergleich zum *am2b*-Modul auf.

Interessant ist, dass sich ab den Linien- zu Dreiecksberechnungen die relativen Laufzeiten der beiden Module angleichen. In diesem Zusammenhang ist es wichtig zu erwähnen, dass im Gegensatz zu `broccoli` in *am2b* die weniger effiziente Berechnung aus Abschnitt 4.3.5 angewendet wird. Es wurden zum Vergleich Testläufe in `broccoli` mit der *einheitlichen* Methode durchgeführt, welche zu wesentlich schlechteren relativen Laufzeiten von $t_{\text{rel, broccoli, LT}} \approx 15$ und $t_{\text{rel, broccoli, TT}} \approx 49$ geführt haben. Die absolute Berechnungszeit von `broccoli` weist bei einer Dreieck- zu Dreiecksberechnung eine um nur 5% bessere Performance auf. Es konnte leider nicht festgestellt werden, warum trotz eines wesentlich effizienteren Algorithmus bei `broccoli` keine größere Laufzeitverbesserung möglich ist. Die genauen Hintergründe dafür wären ein Thema für weitere Untersuchungen.

6.2.5 Single- vs. Multicore-Evaluierung

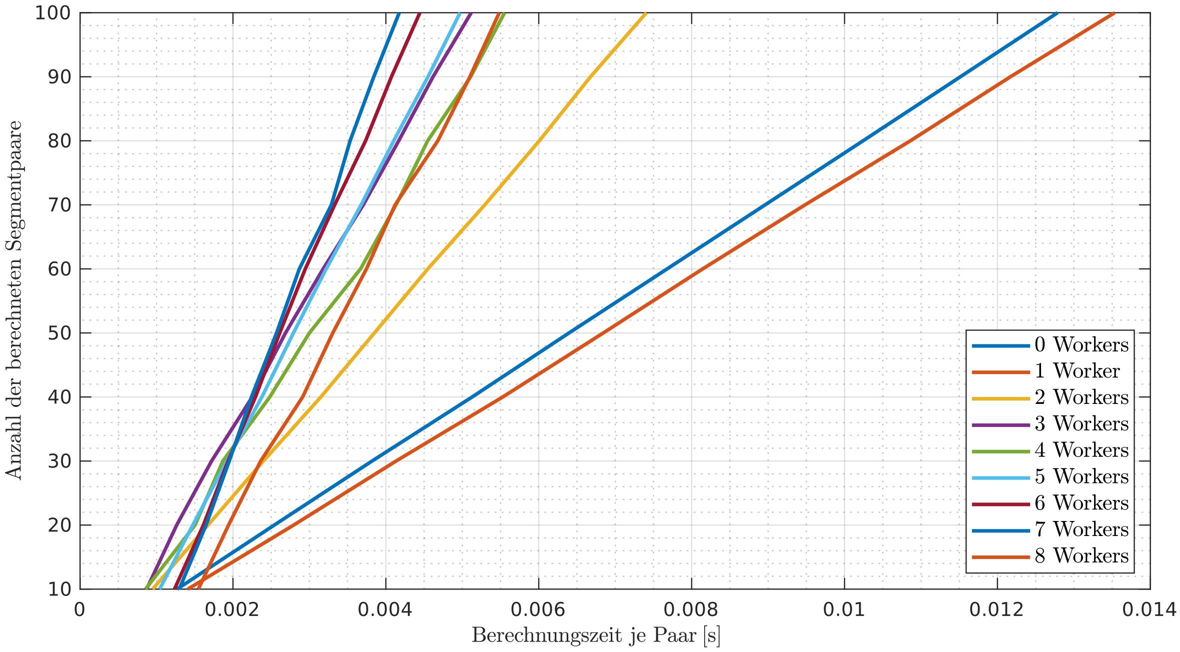


Abbildung 6.1: Berechnungszeiten von Szenenauswertungen abhängig von der Anzahl der Segmentpaare sowie der Anzahl der zur Berechnung verwendeten Worker. Die Evaluierung mit 0 Workers bedeutet eine sequentielle Berechnung im Haupt-Thread. Eine Auswertung durch 1 Worker erfolgt ebenfalls sequentiell, jedoch in einem Hintergrund-Thread. Die schlechtere Laufzeit im Vergleich zu 0 Workers ist dem Datenaustausch-Overhead geschuldet.

In Abb. 6.1 sind die Ergebnisse eines umfangreichen Tests zur Berechnung einer Szene aufgezeichnet. Es wurden für jede mögliche Anzahl an Workern Berechnungen von 10 bis 100 Segmentenpaarungen durchgeführt. Die Zeitmessung erfolgt hierbei ab der Erstellung der aufgeteilten Paarlisten je Worker bis zum Abschluss der Distanzberechnungen der Szene. Die Transformation und Erstellung der Worker fließt nicht in die Zeitnehmung ein.

Wie in Abschnitt 5.3.6 bereits erwähnt, laufen Berechnungen bei keinem oder einem Worker auf nur einem Thread und sollten somit gleich lang zur Evaluierung benötigen. Gründe für die etwas langsamere Berechnung durch einen Worker sind einerseits die Aufteilung der Paarliste und die Übergabe an den Worker und andererseits die Rückgabe der berechneten Distanzen vom Worker an die Szene. Dieser Ablauf bietet Potenzial für zukünftige Optimierungen am Modul (siehe Abschnitt 7.1).
Des Weiteren fällt auf, dass ab drei Workern kein direkter Zusammenhang zwischen der Anzahl der Worker und einer schnelleren Berechnungszeit zu erkennen ist. Die fehlende Korrelation von Laufzeit und der Anzahl der Worker ist auf den Einfluss des am verwendeten Rechner aktivierten Hyperthreading[3] zurückzuführen. Für aussagekräftigere Ergebnisse könnten die Tests mit deaktivierten Hyperthreading wiederholt werden.
Auf dem on-board Rechner von *Lola* wird hardwareseitiges Multithreading deaktiviert, da es ansonsten die Echtzeitfähigkeit verletzen kann. In der Regel weist die Evaluierung mit einer Anzahl der Worker entsprechend der Anzahl der Prozessorkerne die schnellste Berechnungszeit auf.

Klar zu erkennen ist, dass die Verwendung von mehr als zwei Threads die Dauer einer Szenen-

---

[3] Es handelt sich dabei ursprünglich um eine Implementierung von Intel®, welche hardwareseitiges Multithreading ermöglicht.

berechnung bereits ab 30 Paarungen um über 50% reduziert. Auch bei einer sehr geringen Menge von nur zehn Paarungen sind die Berechnungen mit zwei bis fünf Workern bereits schneller wie im Fall einer Singlecore-Auswertung. Die Verwendung von mehr als fünf Workern führt aufgrund der Aufteilung/Übergabe und Rückgabe von Listen bei sehr geringen Paarlisten zu einer schlechteren Performance. Verbesserungen der Performance können vor allem durch Optimierungen der zeitintensiven Über- und Rückgaben der Listen erzielt werden (siehe Abschnitt 7.1).

6.3 Sonstige Anmerkungen

Die Bibliothek `broccoli` stellt mit ihrem umfangreichen Angebot an Modulen und Funktionen auch eine numerisch robuste Alternative für Vergleichsoperatoren zur Verfügung. Im Modul *core* sind unter *floats.hpp* Funktionen implementiert, welche mittels der Maschinengenauigkeit und der ULP[4] zwei Werte miteinander vergleichen.
Beispiel: `a <= b` entspricht dem Funktionsaufruf `core::isLessOrEqual(a, b)`.

Für eine konsistente Programmierung wurden im späteren Verlauf in allen Fallunterscheidungen die Vergleiche mit den Funktionen aus *floats.hpp* durchgeführt. Anschließende Tests zeigten jedoch einen signifikanten Einfluss auf die Laufzeit der Evaluierungen. So führte diese Änderung bei Test aus Abschnitt 6.2.4 zu einem Wert von $t_{\text{rel, broccoli, LP}} = 1,669$, was einem Plus von 27.5% entspricht. Deshalb erfolgte in allen zeitkritischen Funktionen wieder die Verwendung der Standard-Vergleichsoperatoren zur Fallunterscheidung.

6.4 Unit-Tests

Um das Modul auf eine korrekte Funktionsweise überprüfen zu können wurde ein Vielzahl von verschiedensten Testfällen angelegt. Als Framework des Modul- bzw. Unit-Tests dient das für C++ ausgelegte *Google Test*[5]-Framework. Es besticht durch einen einfachen Aufbau und einer Vielzahl an Assertions ($\hat{=}$ Behauptungen, welche in den Tests überprüft werden).
So wurden parallel zur Implementierung des *SSVDistanceEvaluator* zu jeder Elementpaarung einfache Testfälle erstellt, welche so viele Fallunterscheidungen wie möglich abdecken. Dadurch konnten Bugs im Quellcode oder Fehler in den mathematischen Grundlagen, wie in den Glg. (4.22) und (4.23) bereits frühzeitig erkannt und behoben werden. Auch diverse Funktionen der Klassen wie z. B. zur Transformation werden durch Tests auf deren korrekte Funktionsweise überprüft.

---

[4] *Unit in the last place* oder *unit of least precision* - kleinst mögliche Differenz von zwei Gleitkommazahlen.
[5] https://github.com/google/googletest

Codeausschnitt 6.2: Beispiel für einen einfachen Unit-Test mit dem *Google Test*-Framework anhand von zwei Punktelementen.

```
//! Test computation of the distance between two SSVPointElement%s
TEST(SSVDistanceEvaluator, PSS_PSS)
{
   SSVPointElement firstPoint(Eigen::Vector3d(10, 0, 0), 1);
   SSVPointElement secondPoint(Eigen::Vector3d(0, 0, 0), 1);

   SSVDistance result;

   SSVDistanceEvaluator::evaluate(firstPoint, secondPoint, result);

   ASSERT_EQ(result.getDistance(), 10 - 2);
}
```

Codeausschnitt 6.3: Ausgabe nach Ausführung des Unit-Tests.

```
[----------] 1 tests from SSVDistanceEvaluator
[ RUN      ] SSVDistanceEvaluator.PSS_PSS
[       OK ] SSVDistanceEvaluator.PSS_PSS (0 ms)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suites ran. (0 ms total)
[  PASSED  ] 1 test.
```

Kapitel 7

Ausblick

7.1 Szenenberechnung

Das Ergebnis dieser Arbeit stellt eine gute Ausgangsposition für weiterführende Optimierungen und Beschleunigungen dar. Großes Potential zur Verbesserung der Laufzeit liegt in der Implementierung der Szene in Verbindung mit den Workern. Dabei bieten sich folgende Möglichkeiten an:

1. Derzeit erfolgt die Transformation der Segmente einer Szene innerhalb der `evaluate()`-Funktion der *SSVScene*. Bei der Evaluierung durch mehrere Worker wäre es somit möglich, dass neben der Paarliste auch eine Aufteilung der zu transformierenden Segmente auf die Worker erfolgt. Dazu würde in der `execute()`-Funktion der *SSVWorker* vor der eigentlichen Distanzberechnung zusätzlich die Aktualisierung aller Segmente erfolgen.

2. Abhängig von der übergebenen Anzahl an Workern, werden diese bei der Evaluierung erstellt und nach Abschluss wieder entfernt. Deshalb besteht die Möglichkeit, dass auch die Worker bei einer Szene gleich zu Beginn mit erstellt werden. Die Klasse der *BackgroundWorker* aus `broccoli` bietet aus diesem Grund die Möglichkeit, die `execute()`-Funktion zu pausieren und fortzusetzen. Somit würde bei der Multicore-Auswertung einer Szene die Übergabe der benötigten Listen und deren Berechnung an bereits vorhandenen Workern erfolgen.

3. Zur Berechnung der Szene werden jedem Worker mehrere Listen übergeben. Nach erfolgter Berechnung durch diese, erfolgt die Rückgabe der Distanzen an die Szene (siehe Abb. 5.5). Diese Übergaben von Kopien stellen somit einen Overhead in einer Szenenevaluierung dar. Deshalb wäre ein vielversprechender Ansatz, allen Workern direkten Zugriff auf die benötigten Listen der Szene zu geben. Die Threadsicherheit[1] wäre gegeben, da z. B. bei der Evaluierung durch Worker nur lesend auf die Segmentliste sowie Paarliste zugegriffen wird. Bei Verwendung der Aktualisierung aus Punkt 1 wird bei korrekter Aufteilung nur von genau einem Worker lesend und schreibend auf ein Segment zugegriffen. Bei der Rückgabe der Ergebnisse benötigt ebenfalls nur ein Worker schreibend Zugriff auf einen ihm zugewiesenen Eintrag in der Distanzliste.

---

[1] Sicherheit bei mehrfachem Zugriff auf Komponenten durch verschiedene Programmabläufe, sich nicht gegenseitig zu behindern.

7.2 Beschleunigungsmethoden

Im Gegensatz zum Modul aus *am2b* verwendet das neue Framework derzeit keine *Bounding Boxes* bzw. *Bounding Volumes*. Bei Szenen, welche aus einer Vielzahl an Segmenten und zu evaluierenden Paaren bestehen, würde zu Beginn einer Distanzberechnung durch Auswertung der *Bounding Boxes* anhand einer zugehörigen Hierarchie eine Vorfilterung an Segmentpaaren erfolgen, für welche eine genaue Berechnung erforderlich ist.
Eine weitere Möglichkeit zur Vorabreduzierung der Paare bei einer Szenenevaluierung stellt eine sogenannte *Raumpartitionierung* dar. Sie zerlegt einen dreidimensionalen Raum in disjunkte Teilregionen. Segmente innerhalb der Szene werden einer oder mehrerer Teilregionen zugeordnet. Im Gegensatz zu den *Bounding-Boxes* passt sich hierbei nicht die Struktur dem Segment an, sondern ordnet dieses in eine bestehende Partitionierung ein. Durch Traversierung eines zuvor erstellten Baums erfolgt zu Beginn einer Distanzabfrage eine Reduzierung der zu evaluierenden Paare der Szene. [1, Abschnitt 4.4]
Bei sehr großen Szenen, mit einer Vielzahl an Segmenten, kann anhand dieser Beschleunigungsmethoden die Laufzeit einer Evaluierung erheblich verbessert werden. Die Reduzierung der zu evaluierenden Segmentpaare durch eine Vorfilterung überwiegt dabei dem Aufwand der Erstellung und Auswertung einer solchen Struktur.

7.3 Dreieck- zu Dreiecksberechnung

Des Weiteren wurden in Abschnitt 4.3.6 mögliche Spezialfälle bei der Dreieck-zu Dreiecksberechnung angeführt, welche zu einem falschen Ergebnis führen können. Zwei mögliche Ansätze zur Minimierung bzw. Lösung dieses Problem sind:

1. Wie bereits bei der Auflistung der Spezialfälle beschrieben , spielt das Größenverhältnis der beiden Elemente eine entscheidende Rolle für die Größe des Fehlers. Würden im ersten Schritt einer Auswertung die Flächen der beiden Elemente verglichen werden, so könnte immer das Element mit dem kleineren Flächeninhalt für die Zerlegung in Linienelementen festgelegt werden (siehe Abb. 4.11 b)). Somit wird der parallele Spezialfall immer richtig berechnet und der Fehler im Fall b) aus Abb. 4.12 könnte minimiert werden. Dieser Ansatz würde nur einen zusätzlichen Vergleich zu Beginn einer Evaluierung mit sich bringen. Der Flächeninhalt eines Elements kann innerhalb eines *SSVTriangleElement*-Objekts gespeichert werden. Die Flächenberechnung kann dabei anhand des Betrags des Kreuzprodukts von zwei Kanten ($\hat{=}$ Normalenvektor $\boldsymbol{n}$) erfolgen. Da in der derzeitigen Implementierung der Normalenvektor normiert vorliegt, muss bereits der Betrag des Kreuzproduktes berechnet werden. Somit wird der Flächeninhalt als Hilfsgröße bei der Erstellung eines Dreieckselements bereits berechnet und würde keinen zusätzlichen Rechenbedarf in Anspruch nehmen.

2. Um eine korrekte Abstandsberechnung zu jeder möglichen Anordnung gewährleisten zu können, wird die Dreieck- zu Dreiecksberechnung aus Abschnitt 4.3.6 zweimal mit jeweils vertauschter Element-Reihenfolge durchgeführt. Die kürzere der beiden Distanzen entspricht dem tatsächlichen Abstand und wird hinterlegt. Diese Maßnahme führt immer zum richtigen Ergebnis, ist jedoch mit dem doppelten Rechenaufwand verbunden.

Kapitel 8

Fazit

Der Vergleich mit dem bisherigen SSV-Modul aus *am2b* zeigt, dass für alle sechs möglichen Elementpaarungen eine Verbesserung der Berechnungszeit erzielt werden konnte. Durch eine Überarbeitung der fünf Algorithmen konnten sowohl bisher nicht berücksichtigte Fälle bei der Punkt- zu Dreiecksberechnung in die Evaluierung hinzugefügt, als auch effizientere Ansätze bei der Linie- zu Dreiecksberechnung implementiert werden. Somit ist für fünf Elementpaarungen eine korrekte Distanzberechnung gewährleistet - unabhängig von der Konfiguration erfolgt eine zuverlässige Berechnung des geringsten Abstands von zwei Objekten. Bei der Dreieck- zu Dreiecksberechnung konnte durch Einführung des Algorithmus aus Abschnitt 4.3.5 die Anzahl möglicher Konfigurationen für eine falsche Auswertung reduziert werden. Bestehende Spezialfälle, welche zu falschen Ergebnissen in der Dreieck- zu Dreiecksberechnung führen können, wurden aufgezeigt und der Einfluss auf die Auswertung einer Szene und mögliche Lösungen diskutiert. Somit erfolgt mit dem neuen SSV-Modul in `broccoli` eine schnellere und zuverlässigere Abstandsberechnung im Vergleich zu *am2b*.
Ein Vergleich des entwickelten Moduls mit anderen Bibliotheken zur Distanzberechnung aus Kapitel 2 würde den Rahmen dieser Arbeit sprengen. Das Ergebnis eines solchen Vergleichs ist aufgrund verschiedenster zugrundeliegender Paradigmen der Bibliotheken mitunter schwer einzuordnen und bietet Raum für aufbauende Arbeiten.

Die Verwendung der Bibliothek *Eigen* für Berechnungen der linearen Algebra ermöglicht die Nutzung des aktuellsten SIMD-Standards. Durch Aufteilung der Evaluierung einer Szene auf mehrere Worker und einer parallelen Berechnung auf mehreren Prozessorkernen konnten bereits in der derzeitigen Implementierung eine bis zu dreimal schnellere Auswertung im Vergleich zu einer sequentiellen Berechnung erreicht werden.
Durch die stetige Überprüfung des Laufzeitverhaltens von Allokationen, Funktionen und Algorithmen während der Entwicklung, konnte eine bestmögliche Performance der Implementierungen erreicht werden. Die Entwicklung der *selektiven Evaluierung* für die Auswertung einer Szene ermöglicht eine signifikante Verbesserung der Laufzeit und bietet Möglichkeiten für weiterführende Optimierungen.

Anhand der überarbeiteten und aktualisierten Zusammenfassung der mathematischen Grundlagen der Distanzberechnung aus [17, Kapitel 6] in Kombination mit den Pseudocodes im Anhang A, können die Hintergründe und Vorgänge der implementierten Algorithmen genau nachvollzogen werden. Die Verwendung eindeutiger Klassen, Funktions- und Variablennamen fördert die Lesbarkeit des Codes und trägt damit maßgeblich zu einem besseren Verständnis der Funktionsweise bei. Anhand der Dokumentation werden Details zu Funktionen, Variablen und zum Aufbau der verschiedenen Klassen erklärt. Dies, in Verbindung mit einer einheitlichen und flexiblen Struktur, erleichtert die Verwendung und Einbindung des Moduls in weiteren Projekten in der Robotik.

Anhang A

Pseudocode der Algorithmen

Algorithmus A.1: Berechnung Punkt- zu Punktelement → `evaluatePointToPoint()`.

```
Input  : $\boldsymbol{p}_0$, $r_0$, $\boldsymbol{p}_1$, $r_1$
Output: $d$, $\boldsymbol{c}$, $\boldsymbol{p}_0^*$, $\boldsymbol{p}_1^*$
$\boldsymbol{p}_0^* \leftarrow \boldsymbol{p}_0$    // closest points
$\boldsymbol{p}_1^* \leftarrow \boldsymbol{p}_1$
$\boldsymbol{c} \leftarrow \boldsymbol{p}_1^* - \boldsymbol{p}_0^*$
$d \leftarrow \|\boldsymbol{c}\| - r_0 - r_1$
```

Algorithmus A.2: Berechnung Punkt- zu Linienelement → `evaluatePointToLine()`.

```
Input  : $\boldsymbol{p}$, $r_{\text{point}}$, $\boldsymbol{v}_0$, $\boldsymbol{e}$, $r_{\text{line}}$
Output: $d$, $\boldsymbol{c}$, $\boldsymbol{p}_{\text{point}}^*$, $\boldsymbol{p}_{\text{line}}^*$
$\boldsymbol{w} \leftarrow \boldsymbol{p} - \boldsymbol{v}_0$    // precalculation
if $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{e} < 0$ then
    $s \leftarrow 0$
else
    if $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{e} > \boldsymbol{e}^{\mathrm{T}}\boldsymbol{e}$ then
        $s \leftarrow 1$
    else
        $s \leftarrow \frac{\boldsymbol{w}^{\mathrm{T}}\boldsymbol{e}}{\boldsymbol{e}^{\mathrm{T}}\boldsymbol{e}}$
    end
end
$\boldsymbol{p}_{\text{line}}^* \leftarrow \boldsymbol{v}_0 + s\,\boldsymbol{e}$    // closest points
$\boldsymbol{p}_{\text{point}}^* \leftarrow \boldsymbol{p}$
$\boldsymbol{c} \leftarrow \boldsymbol{p}_{\text{line}}^* - \boldsymbol{p}_{\text{point}}^*$
$d \leftarrow \|\boldsymbol{c}\| - r_{\text{point}} - r_{\text{line}}$
```

Algorithmus A.3: Berechnung Punkt- zu Dreieckselement → `evaluatePointToTriangle()`.

Input : $\boldsymbol{p}$, r_{point}, $\boldsymbol{v}_0$, $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, $\boldsymbol{e}_0$, $\boldsymbol{e}_1$, $\boldsymbol{e}_2$, $\boldsymbol{S}$, r_{triangle}

Output: d, $\boldsymbol{c}$, $\boldsymbol{p}^*_{\text{point}}$, $\boldsymbol{p}^*_{\text{triangle}}$

```
$w \leftarrow p - v_0$    // precalculations
$s \leftarrow Sw$
if $s_0 \geq 0$ then    // region 0, 2, 3, 4
    if $s_1 \geq 0$ then    // region 0, 4
        if $s_0 + s_1 < 1$ then
            region 0
        else
            region 4
        end
    else    // region 2, 3
        if $s_0 + s_1 < 1$ then
            region 2
        else
            region 3
            $w \leftarrow p - v_1$    // redefining $w$
            if $-e_0{}^{T} e_2 > 0$ then    // acute triangle
                section 1
            else    // section distinction
                if $-w^{T} e_0 > 0$ then
                    section 0
                else    // section 1, 2
                    if $w^{T} e_2 > 0$ then
                        section 2
                    else
                        section 1
                    end
                end
            end
        end
    end
···
```

```
else  // region 1, 5, 6
    if $s_1 \geq 0$ then  // region 5, 6
        if $s_0 + s_1 < 0$ then
            region 6
        else
            region 5
            $\boldsymbol{w} \leftarrow \boldsymbol{p} - \boldsymbol{v}_2$  // redefining $\boldsymbol{w}$
            if ${\boldsymbol{e}_1}^{\mathrm{T}}\boldsymbol{e}_2 > 0$ then  // acute triangle
                section 1
            else  // section distinction
                if $-\boldsymbol{w}^{\mathrm{T}}\boldsymbol{e}_2 > 0$ then
                    section 0
                else  // section 1, 2
                    if $-\boldsymbol{w}^{\mathrm{T}}\boldsymbol{e}_1 > 0$ then
                        section 2
                    else
                        section 1
                    end
                end
            end
        end
    else
        region 1
        if ${\boldsymbol{e}_0}^{\mathrm{T}}\boldsymbol{e}_1 > 0$ then  // acute triangle
            section 1
        else  // section distinction
            if $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{e}_1 > 0$ then
                section 0
            else  // section 1, 2
                if $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{e}_0 > 0$ then
                    section 2
                else
                    section 1
                end
            end
        end
    end
end
further evaluation depends on region and section → table 4.2 and 4.3
```

Algorithmus A.4: Berechnung Linie- zu Linienelement → `evaluateLineToLine()`.

Input : $\boldsymbol{p}_{0,0}, \boldsymbol{u}, r_0, \boldsymbol{p}_{1,0}, \boldsymbol{v}, r_1$
Output: $d, \boldsymbol{c}, \boldsymbol{p}_0^*, \boldsymbol{p}_1^*$

```
$w \leftarrow p_{1,0} - p_{0,0}$   // precalculations
$N = u^2 v^2 - (u^T v)^2$
if eq. (4.28) is false then   // NOT parallel case
    $sZ \leftarrow v^2 u^T w - u^T v v^T w$   // precalculations
    $tZ \leftarrow u^T v u^T w - u^2 v^T w$
    $sN \leftarrow N$
    $tN \leftarrow N$
    if $tZ < 0$ then
        $sZ \leftarrow u^T w$
        $tZ \leftarrow 0$
        $sN \leftarrow u^2$
    else if $tZ > tN$ then
        $sZ \leftarrow u^T v + u^T w$
        $tZ \leftarrow tN$
        $sN \leftarrow u^2$
    end
    if $sZ < 0$ then
        $sZ \leftarrow 0$
        if $v^T w > 0$ then
            $tZ \leftarrow 0$
        else if $-v^T w > v^2$ then
            $tZ \leftarrow tN$
        else
            $tZ \leftarrow -v^T w$
            $tN \leftarrow v^2$
        end
    else if $sZ > sN$ then
        $sZ \leftarrow sN$
        if $u^T v - v^T w < 0$ then
            $tZ \leftarrow 0$
        else if $u^T v - v^T w > v^2$ then
            $tZ \leftarrow tN$
        else
            $tZ \leftarrow u^T v - v^T w$
            $tN \leftarrow v^2$
        end
    end
    $s \leftarrow sZ/sN$   // calculation of both line parameters
    $t \leftarrow tZ/tN$
...
```

```
else    // parallel case
    if $u^T w < 0$ then
        if $u^T v + u^T w < 0$ then
            $s \leftarrow 0$
            if $u^T v < 0$ then
                $t \leftarrow 0$
            else
                $t \leftarrow 1$
            end
        else if $u^T v + u^T w > u^2$ then
            $s \leftarrow 0.5$
            $t \leftarrow (u^2 - 2u^T w)/2u^T v$
        else
            $s \leftarrow (v^2 + v^T w)/2u^T v$
            $t \leftarrow (u^T v - u^T w)/2u^T v$
        end
    else if $u^T w > u^2$ then
        if $u^T v + u^T w < 0$ then
            $s \leftarrow 0.5$
            $t \leftarrow (u^2 - u^T w)/2u^T v$
        else if $u^T v + u^T w > u^2$ then
            $s \leftarrow 1$
            if $u^T v < 0$ then
                $t \leftarrow 1$
            else
                $t \leftarrow 0$
            end
        else
            $s \leftarrow (u^T v + v^2 + v^T w)/2u^T v$
            $t \leftarrow (u^T v + u^2 - u^T w)/2u^T v$
        end
    else
        if $u^T v + u^T w < 0$ then
            $s \leftarrow v^T w/2u^T v$
            $t \leftarrow -u^T w/2u^T v$
        else if $u^T v + u^T w > u^2$ then
            $s \leftarrow (u^T v + v^T w)/2u^T v$
            $t \leftarrow (u^2 - u^T w)/2u^T v$
        else
            $s \leftarrow (v^2 + 2v^T w)/2u^T v$
            $t \leftarrow 0.5$
        end
    end
end
$p_0^* \leftarrow p_{0,0} + s\, u$    // closest points
$p_1^* \leftarrow p_{1,0} + t\, v$
$c \leftarrow p_1^* - p_0^*$
$d \leftarrow \|c\| - r_0 - r_1$
```

Algorithmus A.5: Berechnung Linie- zu Dreieckselement → `evaluateLineToTriangle()`.

Input : $\boldsymbol{v}_{\mathrm{l}0}, \boldsymbol{v}_{\mathrm{l}1}, \boldsymbol{e}_{\mathrm{l}}, r_{\mathrm{line}}, \boldsymbol{v}_{\mathrm{t}0}, \boldsymbol{v}_{\mathrm{t}1}, \boldsymbol{v}_{\mathrm{t}2}, \boldsymbol{e}_{\mathrm{t}0}, \boldsymbol{e}_{\mathrm{t}1}, \boldsymbol{e}_{\mathrm{t}2}, \boldsymbol{n}_{\mathrm{t}}, \boldsymbol{S}, r_{\mathrm{triangle}}$
Output: $d, \boldsymbol{c}, \boldsymbol{p}^*_{\mathrm{line}}, \boldsymbol{p}^*_{\mathrm{triangle}}$

```
$\boldsymbol{w} \leftarrow \boldsymbol{v}_{\mathrm{t}0} - \boldsymbol{v}_{\mathrm{l}0}$    // precalculations
if eq. (4.35) is false then    // NOT parallel case
    $t \leftarrow {\boldsymbol{n}_{\mathrm{t}}}^{\mathrm{T}} \boldsymbol{w} / {\boldsymbol{n}_{\mathrm{t}}}^{\mathrm{T}} \boldsymbol{e}_{\mathrm{l}}$
    // evaluation based on $t$
    if $t < 0$ then
        $t \leftarrow 0$
        $\boldsymbol{s} \leftarrow \boldsymbol{S}\,(\boldsymbol{v}_{\mathrm{l}0} - \boldsymbol{v}_{\mathrm{t}0})$
    else if $t > 1$ then
        $t \leftarrow 1$
        $\boldsymbol{s} \leftarrow \boldsymbol{S}\,(\boldsymbol{v}_{\mathrm{l}1} - \boldsymbol{v}_{\mathrm{t}0})$
    else
        $\boldsymbol{s} \leftarrow \begin{pmatrix} \boldsymbol{e}_{\mathrm{l}} \times \boldsymbol{e}_{\mathrm{t}1} & \boldsymbol{e}_{\mathrm{t}0} \times \boldsymbol{e}_{\mathrm{l}} \end{pmatrix}^{\mathrm{T}} \boldsymbol{w} / {\boldsymbol{n}_{\mathrm{t}}}^{\mathrm{T}} \boldsymbol{e}_{\mathrm{l}}$
    end
    // evaluation based on $\boldsymbol{s}$
    if $s_0 \geq 0$ then    // region 0, 2, 3, 4
        if $s_1 \geq 0$ then    // region 0, 4
            if $s_0 + s_1 < 1$ then
                region 0
            else
                region 4
            end
        else    // region 2, 3
            if $s_0 + s_1 < 1$ then
                region 2
            else
                region 3
            end
        end
    else    // region 1, 5, 6
        if $s_1 < 0$ then
            region 1
        else    // region 5, 6
            if $s_0 + s_1 \geq 1$ then
                region 5
            else
                region 6
            end
        end
    end
    further evaluation depends on region → tab. 4.6
...
```

```
else    // parallel case
    foreach edge of triangle do
        evaluateLineToLine()    // see algorithm A.4
    end
    evaluatePointToTriangle()    // see algorithm A.3
    return smallest distance of the four previous evaluations
end
```

Literatur

[1] Aichele, F. „Kollisionserkennung für echtzeitfähige Starrkörpersimulationen in der Industrie- und Servicerobotik“. Dissertation. Stuttgart: Institut für Parallele und Verteilte Systeme der Universität Stuttgart, 2015. DOI: 10.18419/opus-8757.

[2] Bergen, G. v. d. „Efficient collision detection of complex deformable models using AABB trees“. In: *Journal of graphics tools* 2.4 (1997), S. 1–13. DOI: 10.1080/10867651.1997.10487480.

[3] Bergen, G. v. d. „A fast and robust GJK implementation for collision detection of convex objects“. In: *Journal of graphics tools* 4.2 (1999), S. 7–25. DOI: 10.1080/10867651.1999.10487502.

[4] Chazelle, B., Dobkin, D. P., Shouraboura, N. und Tal, A. „Strategies for polyhedral surface decomposition: an experimental study“. In: *Computational Geometry* 7.5-6 (1997), S. 327–342. DOI: 10.1016/S0925-7721(96)00024-7.

[5] Coumans, E. u. a. *Bullet 2.83 Physics SDK Manual*. 2015. URL: https://pybullet.org.

[6] Ehmann, S. A. und Lin, M. C. „Accelerated proximity queries between convex polyhedra by multi-level Voronoi marching“. In: *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000) (Cat. No.00CH37113)*. Bd. 3. 2000, 2101–2106 vol.3. DOI: 10.1109/IROS.2000.895281.

[7] Ehmann, S. A. und Lin, M. C. „Accurate and fast proximity queries between polyhedra using convex surface decomposition“. In: *Computer Graphics Forum*. Bd. 20. 3. Wiley Online Library. 2001, S. 500–511. DOI: 10.1111/1467-8659.00543.

[8] Ericson, C. *Real-Time Collision Detection*. CRC Press, 2004.

[9] Gilbert, E. G., Johnson, D. W. und Keerthi, S. S. „A Fast procedure for computing the distance between complex objects in three-dimensional space“. In: *IEEE Journal on Robotics and Automation* 4.2 (1988), S. 193–203. DOI: 10.1109/56.2083.

[10] Gottschalk, S., Lin, M. C. und Manocha, D. „OBBTree: A hierarchical structure for rapid interference detection“. In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 1996, S. 171–180. DOI: 10.1145/237170.237244.

[11] Larsen, E., Gottschalk, S., Lin, M. C. und Manocha, D. „Fast distance queries with rectangular swept sphere volumes“. In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*. Bd. 4. 2000, 3719–3726 vol.4. DOI: 10.1109/ROBOT.2000.845311.

[12] Lin, M. C. und Canny, J. F. „A fast algorithm for incremental distance calculation“. In: *Proceedings. 1991 IEEE International Conference on Robotics and Automation*. 1991, 1008–1014 vol.2. DOI: 10.1109/ROBOT.1991.131723.

[13] Lohmeier, S. „Design and Realization of a Humanoid Robot for Fast and Autonomous Bipedal Locomotion“. Dissertation. München: Technische Universität München, 2010. URL: https://mediatum.ub.tum.de/980754.

[14] Pan, J., Chitta, S. und Manocha, D. „FCL: A general purpose library for collision and proximity queries“. In: *2012 IEEE International Conference on Robotics and Automation*. 2012, S. 3859–3866. DOI: 10.1109/ICRA.2012.6225337.

[15] Samet, H. und Webber, R. E. „Hierarchical data structures and algorithms for computer graphics. I. Fundamentals“. In: *IEEE Computer Graphics and Applications* 8.3 (1988), S. 48–68. DOI: 10.1109/38.513.

[16] Schreiberx. *Example of bounding volume hierarchy (BVH) in two dimensions, where bounding volumes are AABB*. 2011. URL: https://upload.wikimedia.org/wikipedia/commons/2/2a/Example_of_bounding_volume_hierarchy.svg.

[17] Schwienbacher, M. „Efficient Algorithms for Biped Robots - Simulation, Collision Avoidance and Angular Momentum Tracking“. Dissertation. München: Technische Universität München, 2014. URL: https://mediatum.ub.tum.de/1175522.

[18] Seiwald, P. und Sygulla, F. *broccoli: Beautiful Robot C++ Code Library*. 2020. URL: https://gitlab.lrz.de/AM/broccoli.

[19] Täubig, H. und Frese, U. „A new library for real-time continuous collision detection“. In: *ROBOTIK 2012; 7th German Conference on Robotics*. VDE. 2012, S. 1–5. ISBN: 978-3-8007-3418-4.

[20] Terdiman, P. *OPCODE Optimized Collision Detection*. Techn. Ber. URL: http://www.codercorner.com/Opcode.htm.

Erklärung

Ich versichere hiermit, dass ich die von mir eingereichte Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Garching, 18. November 2020

(Unterschrift)