Auto packing für beliebige 3D Objekte und Container

Fachbereich 03: Mathematik/Informatik
Studiengang Informatik

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science

vorgelegt von

Hermann Meißenhelter
Matrikelnummer: 2693454

am 23. April 2019

Erstprüfer:	Prof. Dr. Gabriel Zachmann
Zweitprüfer:	Dr. René Weller

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Abschlussarbeit selbstständig und nur unter Verwendung der von mir angegebenen Quellen und Hilfsmittel verfasst zu haben. Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht. Die Arbeit hat in dieser oder vergleichbarer Form noch keinem anderen Prüfungsamt vorgelegen.

Datum: _____ Unterschrift: _____

Inhaltsverzeichnis

1. Einleitung

In diesem Kapitel wird eine Einleitung in das Thema dieser Arbeit gegeben. Erst wird der Hintergrund und die Bedeutsamkeit des Themas erörtert. Anschließend wird das Problem und Ziel dieser Arbeit formuliert.

1.1. Hintergrund und Motivation

Den Anstoß zu dieser Arbeit entsprang ursprünglich der Idee eines Künstlers. Der Künstler Peter Coffin[1] hatte die Vision einer Statue, welche mit verschiedenen Früchten gefüllt sein sollte. Dabei entspricht das innere der Statue einem Container. Die Früchte sollen dabei so dicht wie möglich aneinander sein, ohne zu überlappen. Eine weitere Bedingung ist, dass die gleichen Fruchtarten möglichst weit auseinander liegen (Inhomogenität) bzw. keine Fruchtart sich an einer Stelle “zu stark“ anhäuft. Es stand auch in Aussicht, das finale Modell mittels CNC aus Schaum auszuschneiden und daraus einen Bronzeguss als öffentliche Kunst aufzustellen.

Abbildung 1.1.: Entwurf zur Statue aus Früchten

[1] `http://petercoffinstudio.com`

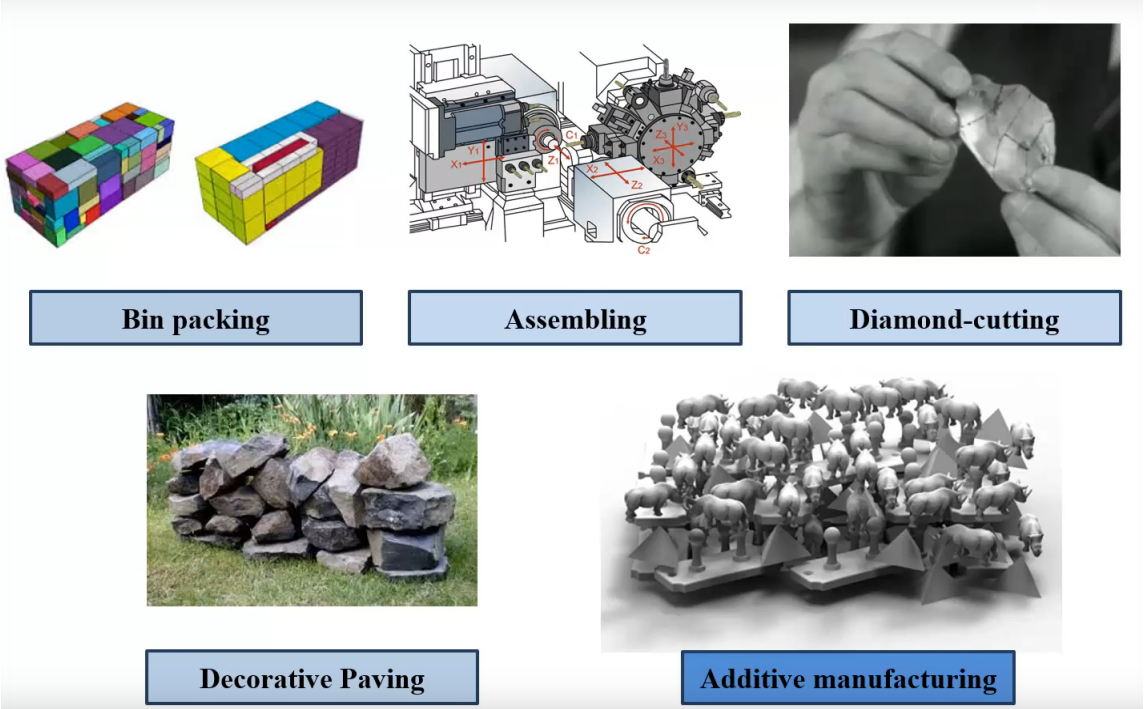


Abbildung 1.2.: Verschiedene Arten von Packproblemen[2]

Das Packen von Früchten in ein Modell lässt sich der Klasse von Packproblemen zuordnen. Packprobleme haben eine weit reichende Geschichte. Im Jahre 1611 hat Kepler seine Vermutung zu dichtesten Kugelpackungen von gleich großen Kugeln aufgestellt. Anlässlich der Keplerschen Vermutung debattierten 1694 Isaac Newton und David Gregory über die Kusszahl (Kontaktanzahl) von Einheitskugeln im dreidimensionalen Raum. Im Jahr 1900 wurden von Hilbert das Problem für dichteste Kugelpackungen und reguläre Tetraeder in seine bekannte Liste für mathematische Probleme aufgenommen (Hilbertsche Probleme) [Li et al., 2010]. Somit hatten Packprobleme schon damals das Interesse auf sich gezogen und besitzen dies bis heute.

Packprobleme lassen sich tiefer unterteilen, wie z.B. 2D/3D und Anwendungsszenario. In der Abbildung 1.2 sind einige Packprobleme zu sehen. Bin packing ist das klassische Packproblem, für das auch gezeigt wurde, dass es NP-Hart ist. Im Assembling sollen die Komponenten so angeordnet werden, dass die vorteilhafteste Wechselwirkung erzielt wird. Beim Diamond-cutting möchte möglichst wenig Reste erzeugen, welche schlecht weiterverarbeitet werden können. Beim decorative

[2]Yagudin Rustem R. (2013): Solving the non-convex polyhedrons dense packing problem, veröffentlicht am 18.06.2013 unter `https://www.youtube.com/watch?v=MHMB6siEz9g`, 27.10.2018

paving möchte man ein Muster aus (unterschiedlich großen) Steinen legen. Für bessere Effizienz besteht beim Additive manufacturing das Interesse, so viele Objekte wie möglich schichtweise aufzubauen. Also müssen hier die Objekte möglichst Dicht beieinander sein, weil die Arbeitsfläche begrenzt ist.

1.2. Problemstellung und Zielsetzung

Das Problem dieser Arbeit beschränkt sich nicht auf Früchte als Objekte oder einer Hand als Container. Als ein Entscheidungsproblem lässt sich das Problem wie folgt definieren: Gegeben sei eine Menge von Polyeder-Objekten S und ein Polyeder Container C. Gibt es eine Platzierung von Objekten innerhalb des Container, ohne Überlappung der Objekte?

Dieses Problem ist bereits NP-Vollständig, wenn alle Polyeder Würfel sind. Angenommen $v(P)$ definiert das Volumen eines Polyeder-Objektes P und $v(S) = \sum_{P \in S} v(P)$, dann liefert $d = v(S)/v(C) \cdot 100$ einen prozentualen Wert, wie gut der Raum des Containers gefüllt ist. Vorausgesetzt die Objekte in S überlappen sich nicht. Das Ziel ist es, diesen Wert zu maximieren. In [Egeblad et al., 2009] wird ein ähnliches Optimierungsproblem wie folgt definiert: Gegeben sei eine Menge von Polyeder-Objekten S in einem rechteckigen Parallelpiped C mit fester Länge l und Breite b, finde die minimale Höhe h bei der die Antwort für das Entscheidungsproblem positiv ist. Da das Ziel ist, beliebige Container mit fester Größe zu füllen, wird eine abgeänderte Definition benötigt. Die passende Problemdefinition ist wie folgt:

Definition 1 *Gegeben sei ein Container C, finde eine Menge von Polyeder-Objekten S bei der d maximal ist und die Antwort für das Entscheidungsproblem positiv ist.*

Die Anzahl der Objekte $|S|$ ist nicht fest in Definition 1. In der finalen Packung haben die Objekte und der Container ihre originale Größe.

Ein anderes Kriterium strebt eine gegebene Verteilung von Objekttypen in einer Packung an. Als Beispiel: Eine finale Packung hat 10 Objekte mit verschiedenen Typen (Äpfel, Birnen, Orangen). In dieser gegebenen Verteilung sind 20% Äpfel, 60% Birnen und 20% Orangen vorgesehen. Dann sollten sich in der finalen Packung 2 Äpfel, 6 Birnen und 2 Orangen befinden.

Ein weiteres Kriterium ist die räumliche Verteilung. Hier sollen die Objekte gleichen Typs gleichmäßig innerhalb des Containers Verteilt werden.

Die genannten Kriterien (Dichte, prozentuale Verteilung, räumliche Verteilung) können auch in Konflikt miteinander stehen. Es kann ein Kriterium verbessert werden, aber dafür wird ein anderes verschlechtert. In dieser Arbeit soll es mehr um Packprobleme gehen, statt um mehrkriterielle Optimierung. Daher wird eine Vereinfachung getroffen und die Dichte als ein dominierendes Kriterium angesehen.

1.3. Aufbau der Arbeit

Die Arbeit ist unterteilt in 6 Kapitel. Beginnend mit der Einleitung, in dem Motivation und Hintergrund für Thema erläutert wird, aber auch das Ziel dieser Arbeit erläutert wird.

In dem nächsten Kapitel 2 wird in die Literatur zu Packungs-Problemen eingetaucht. Begonnen wird dabei mit einer Überblick und es geht weiter mit einigen Paradigmen. Anschließend wird werden einige verwandte Arbeiten betrachtet.

Das Kapitel 3 beschreibt den wesentlichen Kern dieser Arbeit. Hier werden die entwickelten Algorithmen vorgestellt mit Pseudocode und Skizzen.

Im nächsten Kapitel 4 wird es technischer. Hier wird Architektur der Software dargestellt. Auch wird auf die technische Erweiterungen von CollDet eingegangen und eine grundlegende Benutzung der entwickelten Software beschrieben.

Eine Bewertung der entwickelten Algorithmen befindet sich vorletzten Kapitel 5. Hier werden die entwickelten Algorithmen im Detail anhand von verschiedenen Szenarien und Kriterien bewertet.

Zum Abschluss wird im letzten Kapitel 6 eine Zusammenfassung dieser Arbeit und ein Ausblick wiedergegeben.

2. Verwandte Arbeiten

Dieses Kapitel dient dazu, um verwandte wissenschaftliche Arbeiten zu erläutern. Begonnen wird mit einem Überblick zu den verschiedenen Varianten von Packproblem anhand einer Typologie. Danach werden einige typische Ansätze zum Lösen von Packproblem vorgestellt. Im nächsten Unterkapitel werden dann bestimmte Arbeiten tiefer gehend betrachtet, welche ein ähnliches Ziel verfolgt haben.

2.1. Typologie

Bei der Literaturrecherche zu *packing problems* findet man die verschiedensten Ausprägungen von Packproblemen. Um einen Überblick über die wichtigsten Unterschiede zwischen Packproblemen zu erlangen, eignet es sich eine Typologie zu betrachten. In [Dyckhoff, 1990] hat man sich damit bereits beschäftigt, aber im Laufe der Jahre fand die Typologie wenig Akzeptanz aufgrund von Defiziten. Daher hat [Wäscher et al., 2007] eine neue Typologie entwickelt die teils auf der Typologie von [Dyckhoff, 1990] basiert. Die Abbildung 2.1 gibt einen Überblick über die einfachen Typen von Packproblemen.

Es findet zuerst eine einfache Unterscheidung statt zwischen Output-Maximierung und Input-Minimierung. In beiden Fällen muss eine kleine Menge Items einer Menge von großen Objekten (Container) zugewiesen. Im ersten Fall können nicht alle kleinen Items den großen Containern zugewiesen werden, d.h es muss eine Auswahl von kleinen Items vorgenommen werden für die der Output maximal ist. Anders bei Input-Minimierung, es können alle kleinen Items den großen Containern zugewiesen werden. Es muss eine Auswahl für Containern gesucht werden, für die der Input minimal ist. Die Input und Output Werte können je nach Anwendungsszenario anders bestimmt sein.

Übertragen auf die Problemstellung in dieser Arbeit bedeutet das, dass man bei der Output-Maximierung das Gesamtvolumen der kleinen Items versucht zu maximieren. Das Ergebnis ist eine maximale bzw. hohe Packungsdichte. Die nächste Cha-

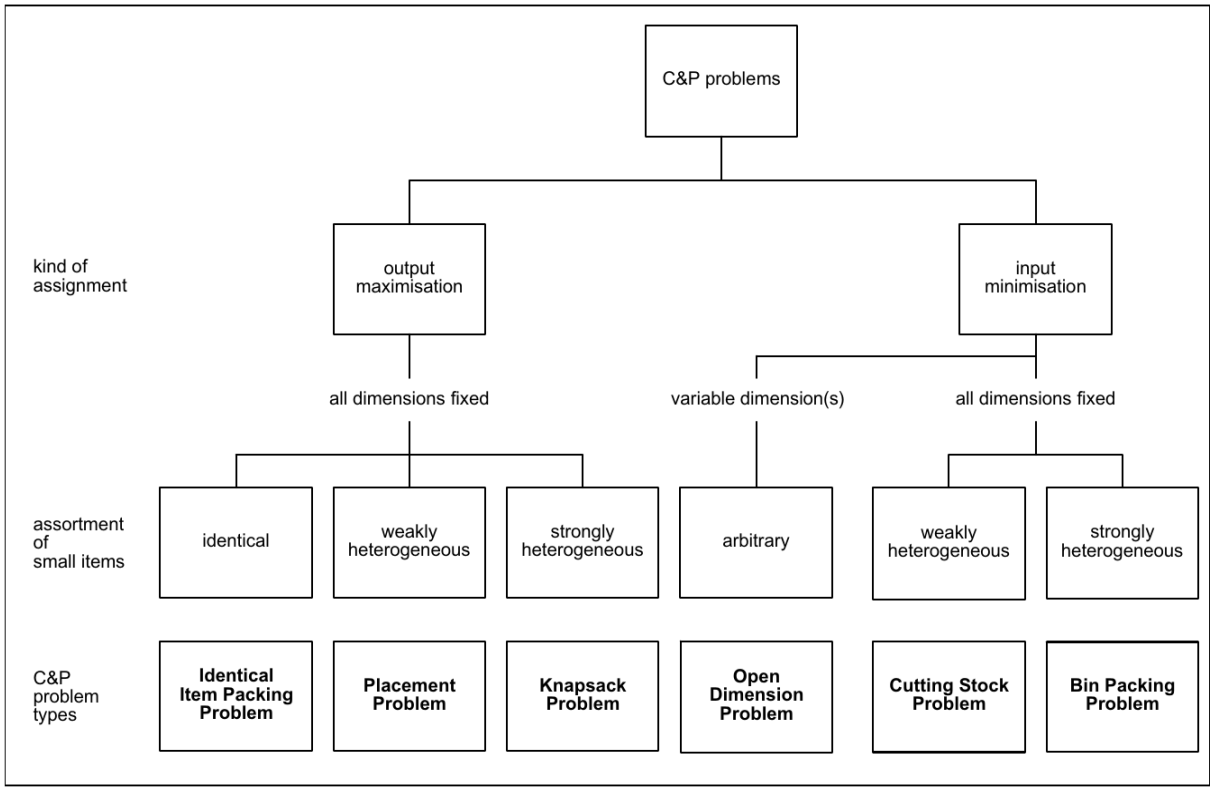


Abbildung 2.1.: Überblick der einfachen Packproblem-Typen nach [Wäscher et al., 2007]

rakteristik unterscheidet die kleinen Items. Schwach heterogene Items zeichnen sich dadurch aus, das viele Items zu wenigen Klassen zusammengefasst werden können. Sie haben bspw. die selbe Form und Größe, was auf die Früchte (siehe Kapitel 1.2) zutrifft. Es handelt sich somit um ein *Placement Problem* nach der Typologie von [Wäscher et al., 2007]. Nach Einordnung in die einfachen Packproblem-Typen, folgt eine fortgeschrittenere Unterteilung. Es folgen bswp. die Kriterien "Charakteristik großer Container", "Dimension" und "Form der kleinen Items". Nach Anwendung der Kriterien ergibt sich für diese Arbeit der Problem-Typ: *3-dimensional irregular non-orthogonal SLOPP*[1].

In der Literatur taucht der Placement Problem Typ unter vielen verschiedenen Namen auf [Wäscher et al., 2007]. Daher gestaltet sich eine Suche nach dem Problem Typ in dieser Arbeit als schwierig und es werden auch Arbeiten betrachtet, die nicht exakt dem Problem Typ dieser Arbeit entsprechen.

[1]Single Large Object Placement Problem

2.2. Grundlegende Ansätze

In diesem Kapitel werden einige Paradigmen[2] zum Lösen von Packungsproblemen vorgestellt. Die entwickelten Algorithmen in Kapitel 3 greifen auf einige Methoden zurück. Man kann bei der Platzierung unter folgenden Kategorien unterscheiden [Egeblad, 2008]:

- Legal Placement Method:
  - Objekt wird immer an einer leere Stelle im Container platziert
  - Begrenzter Suchraum
  - Lösungen können relativ schnell gefunden werden
- Relaxed Placement Method:
  - Überlappungen sind erlaubt und werden minimiert
  - Größerer Suchraum

Einen leeren Bereich im Container zu bestimmen ist nicht trivial. Es ist aber auch möglich ein Objekt zufällig im Container zu Platzieren und bei aufgetretener Überlappung zu verwerfen. Alternativ lässt man Überlappungen zu und versucht diese zu minimieren bzw. vollständig aufzulösen. Den Distance Algorithmus (Kapitel 3.2.2) kann man ganz klar zu *legal placement* zuordnen.

Eine typische Heuristik ist *bottom-left*. Die Objekte werden in einem 2D-Box Container in der oberen-rechten Ecke platziert. Anschließend wird das Objekt kontinuierlich nach unten und links bewegt, solange es möglich ist ohne Überlappung.

Eine modifizierte Version von *bottom-left* ist *bottom-left-fill* (BLF). In der Abbildung 2.2 ist ein Vergleich der Methoden. Zusätzlich wird eine Liste mit Punkten in BLF geführt, an denen möglicherweise ein Objekt platziert werden könnte. Damit lassen sich dann Lücken verkleinern oder schließen [Whitwell, 2004]. Das Ergebnis wird auch beeinflusst von der Reihenfolge der Objekte beim Platzieren. Dazu wurden in [Gomes and Oliveira, 2002] verschiedene Kriterien betrachtet nach der das nächste Objekt gewählt wird. Neben Kriterien wie Fläche und Seitenlängen, wird auch die Form der Objekte bewertet, durch bspw. eine Differenz zwischen einem Objekt und seiner konvexen Hülle.

Um Kollisionen zu erkennen werden in der Literatur zu Packungsproblemen auch sog. no-fit polygons verwendet. Die Idee ist es die Kollisionserkennung zu reduzieren

[2] In der Literatur zu Packungsproblemen auch Methoden oder Heuristiken genannt

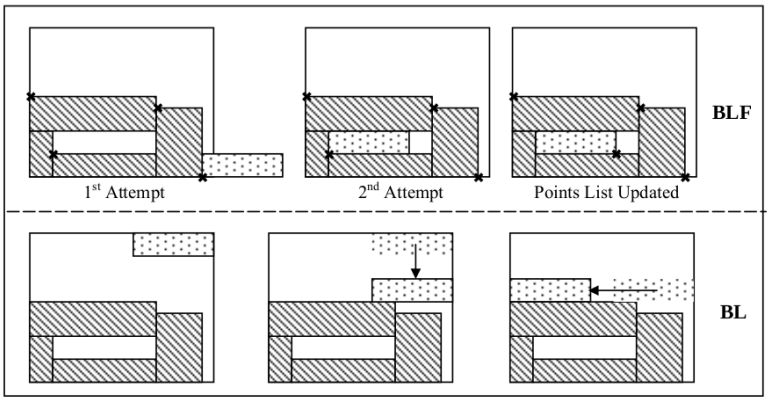


Abbildung 2.2.: Bottom-Left (BL) im Vergleich zu Bottom-Left-Fill (BLF) in [Whitwell, 2004]

auf eine Kollisionstest zwischen einem Punkt und einem Polygon. Dazu wird ein Polygon B mit einem frei gewählten Eckpunkt P_B um das anderes Polygon A herum bewegt und dabei ein neues Polygon NFP_{AB} konstruiert (siehe Abb. 2.3). Ein Kollisionstest zwischen P_B und NFP_{AB} bestimmt dann eine Überlappung zwischen den Polygonen A und B. Es gibt verschiedene Algorithmen zur Konstruktion. Für Polyeder gibt es sehr wenige Umsetzungen (2D nach 3D) und scheinbar nur einfache Polygone.

In der Praxis (Fertigungsindustrie) werden solche Modelle aufgrund der komplexen Implementierung und Robustheit der Algorithmen nicht verwendet. Die Standard Kollisionserkennung[3] wird bei “Packing Software‘"verwendet, wo es wichtig ist, das alle möglichen Polygone gehandhabt werden können [Burke et al., 2007].

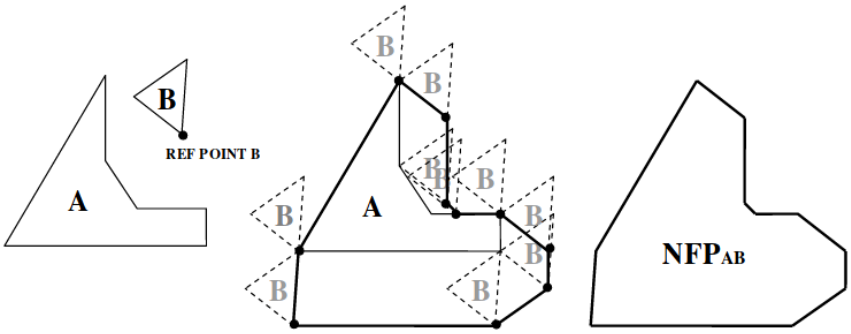


Abbildung 2.3.: No-fit polygon NFP_{AB} zwischen Polygon A und B in [Burke et al., 2007]

[3]Bspw. Tests zwischen: Kante-Kante oder Kante-Punkt

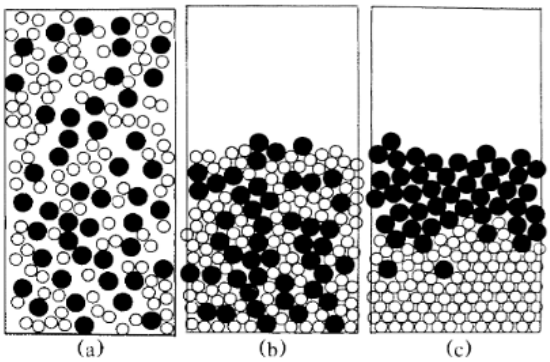


Abbildung 2.4.: Segregation einer 50/50 Mixture von Partikeln nach [Rosato et al., 1987]

2.3. Arbeiten zu Packungsproblemen

Als Nesting Probleme bezeichnet man sehr oft 2-dimensionale Probleme, bei denen eine Optimierung des Materialverbrauchs angestrebt wird. Da in der Additiven Manufaktur komplexere Objekte verwendet werden und ein optimiertes Layout anstrebt wird, wurde dafür in [Lutters et al., 2012] ein Algorithmus vorgestellt. Der Algorithmus setzt die Objekte in eine präferierte Rotation. Dabei handelt es sich um ein Qualitätskriterium in der Layered Manufaktur, um den *stair stepping* Effekt zu reduzieren. Anschließend wird eine Vibration verwendet um die Dichte zu erhöhen. Beim Schütteln kann es zum sog. "*Brazil Nutt Effect*" kommen. Dieser Effekt ist in der Abbildung 2.4 anschaulich dargestellt. Initial sind die Partikel zufällig platziert (a), danach werden sie fallen gelassen (b) und abschließend wird 300 mal geschüttelt (c). Dieser Effekt entsteht wenn zu eine längere Zeit geschüttelt wird. Die großen Objekte bewegen sich aufwärts, weil diese beim Schütteln Hohlräume hinterlassen, die von kleineren Objekten ausgefüllt werden. Die Idee in [Lutters et al., 2012] ist es, nicht zu viel zu schütteln, um möglichst den Zustand wie in der Abbildung 2.4 (b) zu erreichen.

Für diese Arbeit ist der beschriebene Zustand auch interessant, da verhindert wird das z.B. alle Äpfel am Boden dicht beieinander sind. Wie man den Zustand in (b) sichergestellt oder man es überhaupt sicherstellen kann, wurde nicht explizit erwähnt. Wesentliche Parameter sind die maximale bin Höhe als Performanz Indikator und die zu Beginn festgelegte Laufzeit. Die initiale Füllung ist nach einem 'rain model' realisiert. Die Objekte bewegen sich von oben in zufällige Richtungen. Nach jeder Vibration wird versucht ein Objekt von oben in einen Hohlraum einzufügen. Es ist

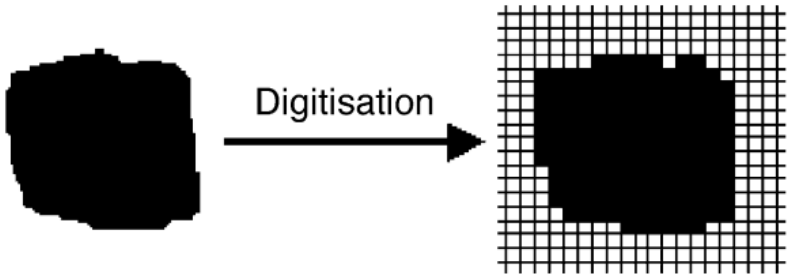


Abbildung 2.5.: Digitisation eines Objektes nach [Jia and Williams, 2001].

nicht genau beschrieben wie ein Hohlraum erkannt wird. Wenn die festgelegte Laufzeit abgelaufen ist, wird zum Abschluss die Packung in eine STL Datei exportiert [Lutters et al., 2012].

Die meisten Packproblem Algorithmen beziehen sich auf wenige reguläre Formen z.B. Kreise. Andere bzw. willkürliche Objekte miteinzubeziehen scheitert daran, dass sich diese Objekte schwierig mathematisch modellieren lassen. Oder die benötigte Zeit für Kollisions-Algorithmen ist sehr hoch. Aus den Gründen wurden in der Arbeit [Jia and Williams, 2001], die Objekte in Pixel in 2D und Voxel in 3D transformiert. Die Arbeit bezeichnet den Vorgang als '*Digitisation*' (siehe Abbildung 2.5).

Das Gitter bzw. Auflösung bestimmt den Kompromiss zwischen Genauigkeit und Laufzeit. Zu jeden Zeitpunkt darf sich ein Objekt nur um eine Gitterzelle bewegen. Diagonale Bewegungen werden als zwei orthogonale Bewegungen behandelt. Die Objekte bewegen sich zu jedem Zeitschritt zufällig in eine Richtung. Eine Art Gravitation wird dadurch erzeugt, dass die Aufwärtsbewegung an eine Wahrscheinlichkeit geknüpft ist (*rebounding probability*). Diese ist so gesetzt, dass die Objekte sich diffusiv und eher abwärts gerichtet bewegen. Die Diffusion soll helfen jede Lücke zu schließen und eine nach unten gerichtet Bewegung, soll die Objekte zu einer festen Struktur führen. Um die Objekte initial zu positionieren, stehen zwei Möglichkeiten zur Auswahl. Die erste Art ist eine Punktquelle, an der ein Objekt zu einem bestimmten Zeitpunkt eingeführt wird. Die zweite Art ist das sog. '*rain model*' bei dem bis zu 10 Objekte gleichzeitig und uniform verteilt eingeführt werden. In der Abbildung 2.6 a) wird das 'rain model' verwendet. Dabei ist keine Rotation erlaubt und kein rebounding bzw. Schwerkraft. In Abbildung 2.6 b) kommt eine Punktquelle zum Einsatz. Rebounding ist hier aktiviert und die Objekte rotieren zufällig. Man kann erkennen, dass die Art der initialen Platzierung, die Möglichkeit von Rotation und die rebounding Wahrscheinlichkeit einen Einfluss auf die Struktur der Packung

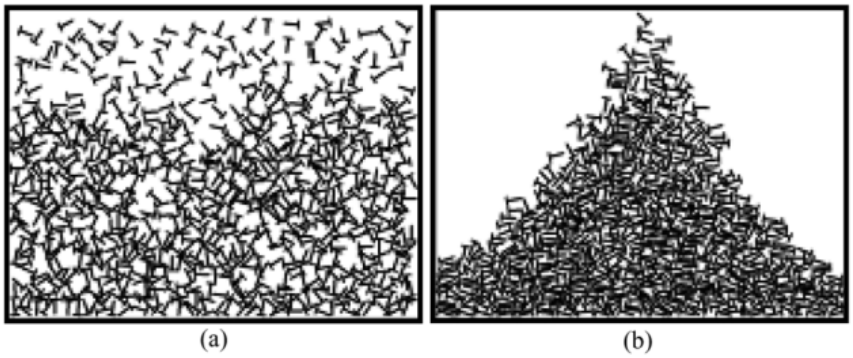
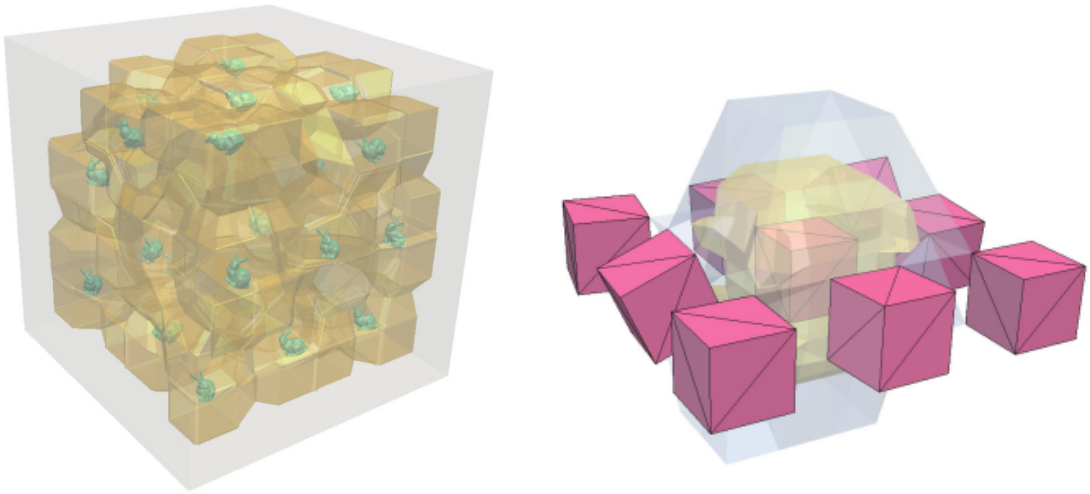


Abbildung 2.6.: Container werden mit Nagel-förmigen Objekten gefüllt [Jia and Williams, 2001].

haben [Jia and Williams, 2001]. Ein Nachteil dieses Ansatzes ist der hohe Speicherverbrauch. In einer folgenden Arbeit wurde dieser Ansatz auf 3D Objekte übertragen [Gan et al., 2004].

In der Arbeit [Ma et al., 2018] wurden eine kontinuierliche Optimierung mit einer kombinatorischen Optimierung kombiniert. Bei der initialen Platzierung wird die Position und Orientierung der Objekte optimiert. Vorher wird das Volumen des Containers in CAT[4]/ CDT[5] Zellen aufgeteilt (siehe Abbildung 2.7a). Jedes Objekt wird zentral in der CAT Zelle platziert und es findet eine Optimierung der Skalierung, Rotation und Position statt. Die CAT Zellen sind kleiner als die CDT Zellen (siehe 2.7b). Innerhalb der CAT Zellen finden bis zu $N = 8$ Vertauschungen untereinander statt, bei der die vorherige Optimierung wiederholt wird. Die CDT Zellen können als maximales verfügbares Volumen betrachtet werden. Entsprechend wird im nächsten Schritt für die Objekte ein Score berechnet, der den maximal erreichten Skalierungsfaktor in einer CDT Zelle angibt. Falls der Skalierungsfaktor unter 1 liegt, werden Ersetzungen mit neuen Objekten vorgenommen. Es wird also eine Teilmenge von verfügbaren Objekten platziert, ohne eine Verteilung zu berücksichtigen. Im letzten Schritt wird versucht die bestehende Hohlräume mit Objekten zu füllen. Einen Ausschnitt der Ergebnisse zu verschiedenen Objekten und Containern sind in der Abbildung 2.8 zu sehen.

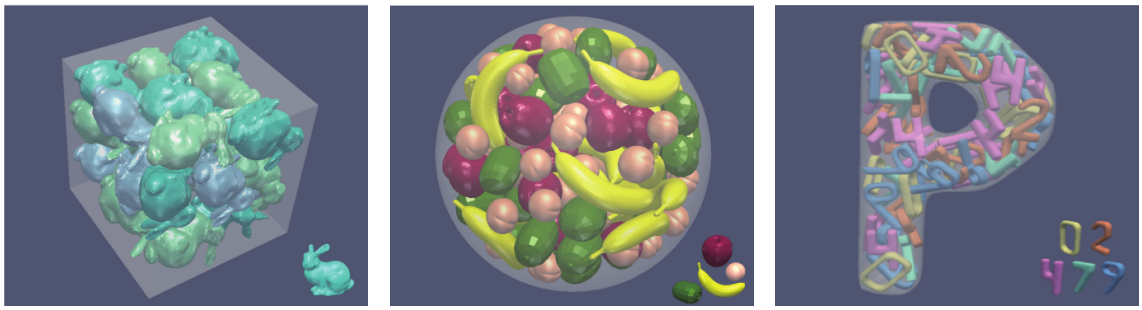
[4]Chordal Axis Transform [Prasad, 1997]
[5]Constrained Delaunay Tetrahedralization [Si and Gärtner, 2005][Si, 2015]



(a) CAT Zellen im Container.

(b) Pink ist das Objekt, Gelb die CAT Zelle und Blau die CDT Zelle.

Abbildung 2.7.: CAT und CDT Zellen in [Ma et al., 2018]



(a) 25 Hasen in einer Box, Packungsdichte 45%

(b) 108 Früchte in einer Kugel, Packungsdichte 35, 7%

(c) 66 Zahlen in einem P-Förmigen Container, Packungsdichte 23, 5%

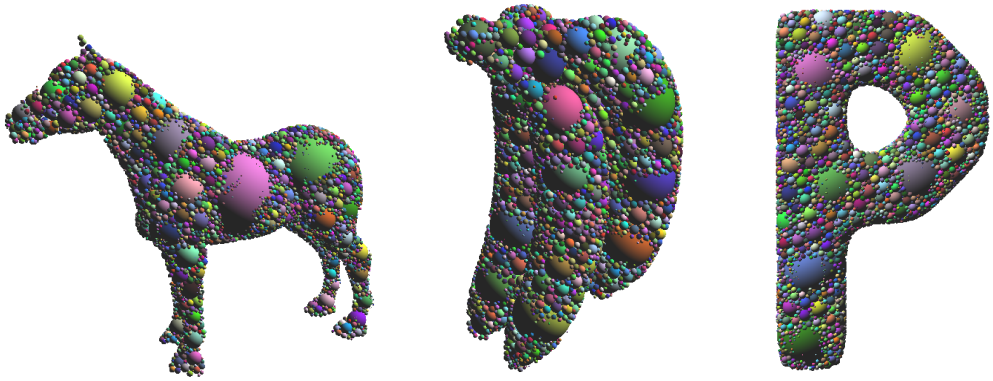
Abbildung 2.8.: Ausschnitt der Ergebnisse zu verschiedenen Objekten und Containern in [Ma et al., 2018]

2.4. Kugelpackungen zur Kollisionserkennung

In dieser Arbeit wird zur Kollisionserkennung die Bibliothek CollDet[6] verwendet. Darauf Aufbauend wurden die Algorithmen zum Packen von Objekten in Kapitel 3 entwickelt.

In [Weller, 2013] wurde eine neue geometrische Datenstruktur zur Kollisionserkennung entwickelt, welche in CollDet zur Verfügung steht und für diese Arbeit verwendet wird. Dabei wurde auch ein Algorithmus mit dem Namen *Protosphere* entwickelt. Dieser Greedy-Algorithmus füllt iterativ ein beliebiges Objekt mit n Kugeln. Das Ergebnis ist das sog. „Apollonian sphere packing“, was Raum-füllend ist. Einige Ergebnisse sind in der Abbildung 2.9 zu sehen.

In einem zweiten Schritt wird eine *Inner* Bounding Volume Hierarchy zu der Menge an Kugeln erzeugt, der sog. *Inner Sphere Tree*. Die Blätter liegen alle innerhalb eines Knotens, jedoch dürfen sich die Kinder-Knoten außerhalb der Eltern-Knoten befinden. Alle Inner Spheres[7] befinden sich innerhalb des Wurzel-Knotens. Zur Partitionierung der Inner Spheres wird eine eigene angepasste Version des batch neural gas Cluster Algorithmus verwendet. Die Datenstruktur ermöglicht eine relativ schnelle Kollisionserkennung, sowie die Berechnung von einem überlappten Volumen, als auch die Kontaktpunkte bei einer Kollision.



(a) Pferd (b) Bananen Gruppe (Peter Coffin Model) (c) Buchstabe „P“ aus Datenset in [Ma et al., 2018]

Abbildung 2.9.: Kugelpackungen erzeugt von Protosphere mit $\approx 1 \cdot 10^4$ Kugeln

[6] `http://cgvr.informatik.uni-bremen.de/research/colldet/index.shtml` (Abruf 20.04.2019)

[7] Bezeichnet in der Arbeit alle Kugeln innerhalb des Objektes siehe Abb. 2.9

3. Algorithmen

In diesem Kapitel werden die entwickelten Algorithmen zum lösen von Packungs Problemen vorgestellt. Die Algorithmen lassen sich in drei aufeinanderfolgende Schritte zerlegen (siehe 3.1). Im ersten Schritt werden Punkte innerhalb des Containers berechnet. Danach wird in der Konstruktionsheuristik eine initiale Lösung berechnet. Zum Abschluss soll die initiale Lösung verbessert werden mit den Verbesserungsheuristiken.

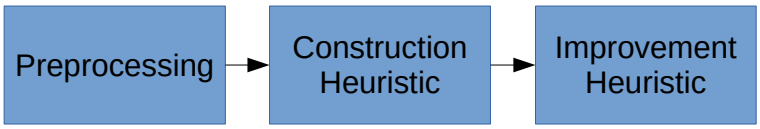


Abbildung 3.1.: Algorithmen Pipeline.

3.1. Preprocessing

Die Algorithmen in den folgenden Kapiteln 3.2 und 3.3 benötigen eine Liste mit Punkten die innerhalb des Containers liegen. Für beispielsweise einen Würfel oder Kugel lassen sich solche Punkte relativ leicht mathematisch bestimmen. Bei beliebigen Objekten ist das nicht mehr Fall.

Im Preprocessing-Schritt werden diese Punkte für beliebige Container ermittelt. Ein Punkt ist im Container, wenn der Collisioncheck zwischen einer Kugel und dem Container negativ ist. Die Kugel befindet sich dann vollständig innerhalb des Containers. Statt einer Kugel könnte man auch alternativ ein Objekt aus der Menge wählen. Nimmt man ein Objekt, so spielt bei der Wahl die Form, Größe und Verteilung des Objekttyps eine Rolle. Eine Kugel führt mit größerer Wahrscheinlichkeit zu einer gleichmäßigeren Verteilung der Sampling-Punkte, als eine beliebige Objektform aus der Menge. Noch entscheidender ist wohl, dass die Zeit für einen Collisionscheck mit nur einer Kugel sehr verkürzt wird.

Das Volumen V der Sampling-Kugel wird über einen gewichteten Durchschnitt aller Objektvolumen berechnet (siehe Gleichung 3.1). Für die normalisierten Gewichte gilt: $\sum_{i=0}^{n} w_i' = 1$. Jeder Objekttyp hat eine angestrebte prozentuale Verteilung, was hierbei als Gewichtung verwendet wird.

$$\overline{V} = \sum_{i=0}^{n} w_i' \cdot V_i \tag{3.1}$$

Eine ideale Anzahl Punkten ist schwer zu bestimmen. Bei zu vielen Punkten erhöht sich die Packungsdichte nicht mehr signifikant, aber dafür erhöht sich die Laufzeit relativ stark. Zu wenige Punkte verringern die Laufzeit, aber erreichen nicht eine hohe Packungsdichte. Die Schrittweite ist dafür entscheidend.

Die Kugel wird um die Schrittweite *stepSize* innerhalb des Containers bewegt. Um nachträglich noch die Sampling-Auflösung zu regulieren, gibt es einen Skalierungsfaktor $scaleFacor$:

$$stepSize = 2 \cdot radius \cdot scaleFactor \tag{3.2}$$

Wenn es von der Verteilung her viele kleine Objekte gibt, so wird das Kugelvolumen kleiner und damit das Sampling feiner. Bei vielen großen Objekten wird das Kugelvolumen größer und das Sampling gröber.

Das Sampling wird parallel ausgeführt mittels n Threads. Jeder Thread wird mit einer Reihe von Attributen initialisiert, wie z.B. Objekt, Container, Collision Pipeline. Anschließend wird das Sampling unter den Threads aufgeteilt:

Algorithm 1 Splitting work for $nThreads$

Input: Container bounding box (min, max), $stepSize, nThreads$
Output: $listOfPoints$

```
1: for x = min[0] to x < max[0] step x += stepSize do
2:     if nThreads are started then
3:         Wait for nThreads to finish
4:     end if
5:     Start a thread with current x and a 2-dimensional array for results
6: end for
```

Die Arbeit eines Threads ist in Algorithmus 2 festgehalten. Für einen festen x-Wert wird die ganze yz-Fläche betrachtet. Wenn es keine Collision gibt bzw. der Punkt innerhalb des Containers liegt, wird er in die Liste aufgenommen. Das Er-

gebnis ist eine Liste von Punkten, die innerhalb des Containers liegen.

Algorithm 2 Container sampling for fixed x value and all y, z values

Input: *samplingData*
Output: *yzArray*

```
1: for y = min[0] to y < max[0] step y += stepSize do
2:    Empty array zArray
3:    for z = min[0] to z < max[0] step z += stepSize do
4:       Set object to the position p = (samplingData.x, y, z)
5:       Check for collision
6:       if no collision then
7:          zArray.add(p)
8:       end if
9:    end for
10:   samplingData.yzArray.add(zArray);
11: end for
```

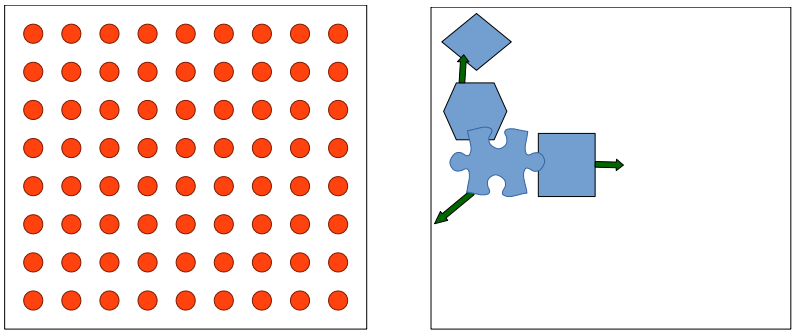
3.2. Konstruktionsheuristiken

Im folgenden werden vier Konstruktionsheuristiken vorgestellt. Die Konstruktionsheuristiken sind Algorithmen, die mit einer leeren Lösung bzw. Packung beginnen und aufbauend eine Lösung berechnen. Das Ergebnis ist eine initiale Packung.

3.2.1. Placement Algorithmus

Beim Placement Algorithmus ist die grundlegende Idee, eine Packung mittels Sampling-Punkten und Collision response zu erstellen. An den Sampling-Punkten werden die Objekte nach dem Prinzip *deepest-bottom-left* platziert. Das Objekt wird zufällig gewählt unter Berücksichtigung der prozentualen Typ-Verteilung. Falls das Objekt an dem Punkt nicht ohne Überlappung mit dem Container platziert werden kann, werden stattdessen kleinere Objekte (kleineres Volumen) genommen.

Wenn kein Objekt ohne Container-Überlappung platziert werden konnte, wird mit dem nächsten Punkt wiederholt. Alle Objekte, die nicht platziert werden konnten, werden in einem Puffer gespeichert. Die Objekte im Puffer haben Vorrang bei der Wahl, d.h. erst wenn der Puffer leer ist, wird ein neues Objekt zufällig gewählt. Die Idee hinter Puffer ist, eine bessere Annäherung für die prozentuale Typ Verteilung zu erreichen. Die Objektwahl berücksichtigt nicht die aktuell platzierte Typ Verteilung.



(a) Container Sampling

(b) Kollisionsauflösung bei der Platzierung

Abbildung 3.2.: Skizze zu dem Placement Algorithmus.

Mit dem Puffer haben Objekte-Typen, die nicht häufig vorkommen oder selten einen Platz finden, eine erhöhte Wahrscheinlichkeit zur Platzierung.

Zur Kollisionsauflösung wird eine Impulse basierte Simulation verwendet. Es werden also Geschwindigkeitsvektoren und Rotations-Geschwindigkeitsvektoren berechnet. Eine Gravitationskraft oder allgemein Kräfte werden nicht berücksichtigt. Ein Objekte bewegt sich oder rotiert nur solange es eine Überlappung mit einem anderen Objekt oder dem Container hat.

Zusammengefasst kann der Placement Algorithmus wie folgt beschrieben werden:

3.2.2. Distance Algorithmus

Die Kernidee beim Distance Algorithmus ist, eine Packung zu erstellen mit Hilfe der Sampling-Punkte und dem Anheften von Objekten. Sei ein Objekt A mit der Modelmatrix M_1 und ein Objekt B mit der Modelmatrix M_2 gegeben.A ist an B angeheftet, genau dann wenn A um einen Vektor x verschoben wird, sodass der Abstand zwischen A und B minimal und die Objekte überschneidungsfrei sind. In der CollDet-Bibliothek kann zu einem Paar von Collision-Objekten, ein Kugelpaar mit der kürzesten Distanz zueinander bestimmt werden. Jede Kugel ist verknüpft mit einem Dreieck aus dem Objekt-Mesh, bei der die Distanz zwischen Kugel und Dreieck minimal ist. Auf den Dreiecken lassen sich zwei Punkte P_{Tri1}, P_{Tri2} mit der kürzesten Distanz zueinander bestimmen. Daraus wird ein Translation Vektor berechnet, um ein Objekt zum jeweiligen anderen Objekt zu bewegen (siehe Gleichung 3.5). Die Distanz zwischen den Objekten wird minimal und ohne das sich die Objekte überlappen. In der CollDet wird intern ein Basiswechsel von M_1 nach M_2 durch-

Algorithm 3 Placement Algorithmus

Input: $objectTypeInfoArray, Container, placedObjectArray$
Output: $placedObjectArray$

```
1: objectsBufferArray {Objects which did not fit}
2: for each point in pointsInsideContainer do
3:    if objectsBufferArray not empty then
4:       Select an object from objectsBufferArray
5:    else
6:       Create a new random object
7:    end if
8:    Rotate object to avoid collision with container or try smaller object
9:    if no object is fitting then
10:      Add no fitting objects to objectsBufferArray
11:      Continue with new point
12:   end if
13:   Try to resolve overlap within time t
14:   if overlap not resolved within time t then
15:      Place objectToBePlaced far away and remove from placedObjects
16:      Add object to objectsBufferArray
17:   end if
18: end for
```

geführt (siehe Gleichung 3.3). Die Punkte befinden sich in der Basis von Objekt B und werden dann transformiert zur Basis A. In Weltkoordinaten wird dann Objekt A durch die Translation x verschoben.

$$T_{M_2}^{M_1} = M_2^{-1} \cdot M_1 \tag{3.3}$$

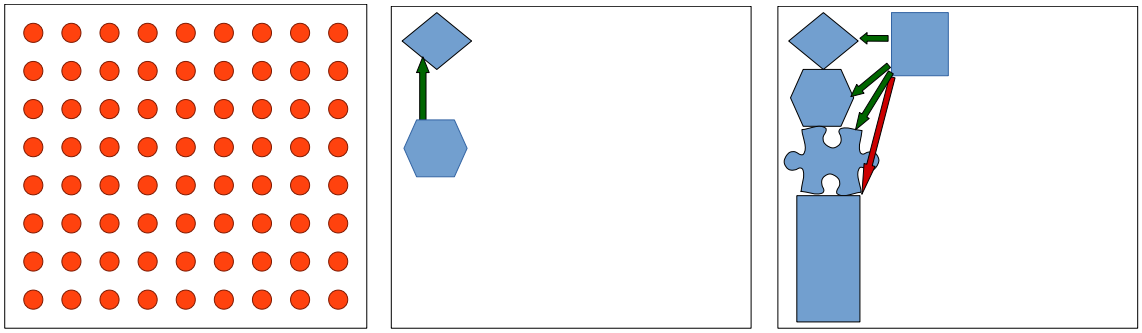
$$(T_{M_2}^{M_1})^{-1} = T_{M_1}^{M_2} \tag{3.4}$$

$$x_{translation} = M_1 \cdot (T_{M_2}^{M_1})^{-1} \cdot (p_{Tri2} - p_{Tri1}) \tag{3.5}$$

In einem kontinuierlichen Raum gibt es unendlich viele Richtungen aus denen man ein Objekt an ein anderes hängen könnte. Um diesen Suchraum einzugrenzen ist die Idee, die Sampling-Punkte als eine Richtung zu nutzen. Ergänzt bzw. kombiniert mit dem *deepest-bottom-left* Aufbau erhöht sich die Wahrscheinlichkeit, dass das Anheften innerhalb des Containers liegt und eine Dichte Packung entsteht. Mit dem Schichtenweisen Aufbau müssen nicht alle Paare mit dem neuen Objekt und den bereits platzierten Objekten betrachtet werden. Es werden nur die Paare (A, B_i) mit $i \ldots n$ betrachtet, wobei $n = 20$ beim Testen ein guter Wert schien

(Performanz/Dichte).

Der zusammengefasste Algorithmus ist in Algorithmus 4.



(a) Container Sampling

(b) Objekt wird ein anderes Objekt angehängt

(c) Greedy Anhängen bis Überlappungsfreier Zustand (Grün) erreicht ist.

Abbildung 3.3.: Skizze zu dem Distance Algorithmus.

3.2.3. Hybrid Algorithmus

Aus den Ansätzen bzw. Algorithmen Placement und Distance, entstand die Idee beides zu kombinieren im Hybrid Algorithmus. Hinzu kommt eine Rigid-Body Simulation mit einer Gravitationskraft. Mit der Gravitationskraft bzw. dem fallen lassen von Objekten soll eine dichte Packung erzielt werden.

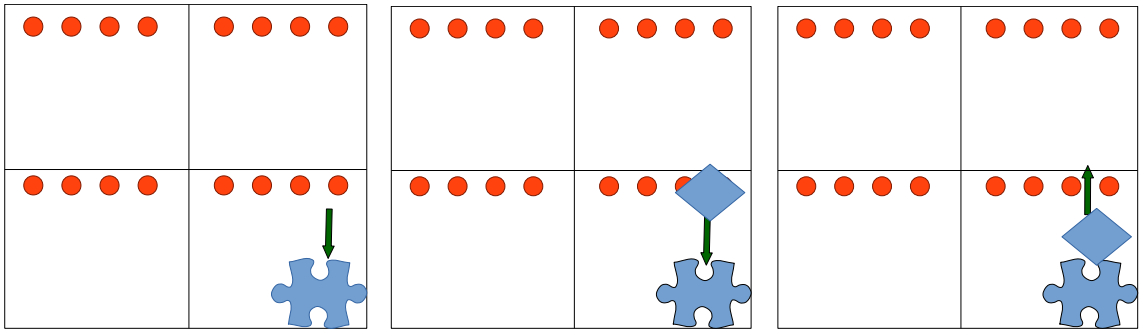
In der Abbildung 3.4 ist der Ablauf vom Hybrid Algorithmus skizziert. Der Container wird durch ein Grid aufgeteilt mit einer frei wählbaren Auflösung (x, y) bzw. Anzahl von Grid-Zellen. In der Skizze ist die Auflösung bei $x = 2, y = 2$, womit wir insgesamt $x * y = 4$ Zellen haben. Es werden in jeder Zelle nur die Sampling-Punkte betrachtet, die den größten y-Wert besitzen. Von diesen Sampling-Punkten wird das erste Objekt fallen gelassen (siehe Abbildung 3.4a).

Die Geschwindigkeit des Objektes wird verringert mit jeder Kollision. Wenn die Geschwindigkeit des Objektes unter einen Schwellwert geriet, dann wird die Geschwindigkeit auf null gesetzt bzw. das Objekt bewegt sich nicht. Beim Stillstand des fallen gelassenen Objektes bleibt eine Überlappung mit den anderen Objekten oder dem Container. Es wird ein kleines Δt verwendet, was dazu führt, dass die Überlappungen mit geringer Geschwindigkeit sehr klein sind. Diese Überlappungen werden wie im Placement Algorithmus aufgelöst.

Algorithm 4 Distance Algorithmus

Input: $placedObjectArray, pointsInContainerArray$
Output: $placedObjectArray$

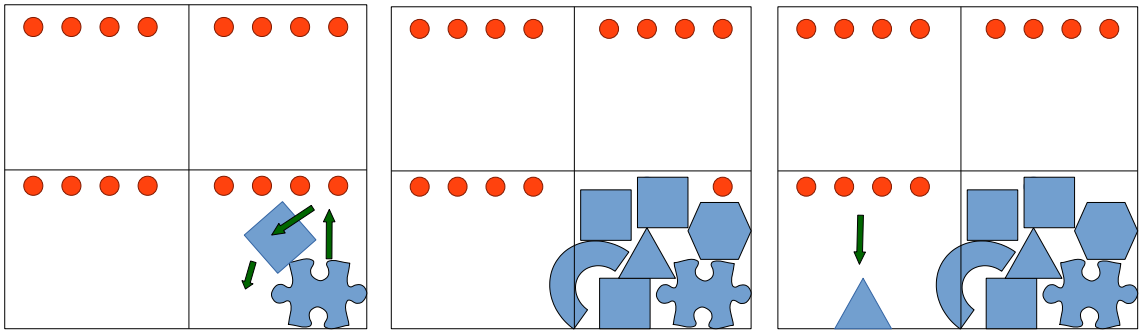
```
1: for each point p in pointsInContainerArray do
2:   if generateObject = true then
3:     Generate random object A
4:     generateObject = false
5:   end if
6:   Set A to position p
7:   Try to rotate A for no collision
8:   if no collion then
9:     if placedObjectArray is not empty then
10:      for the last lastMax = 20 objects B in placedObjectArray do
11:        distance = Attach A to B
12:        if distance > tooFar = 5 then
13:          Set A to position p
14:          Add A to placedObjectArray
15:          generateObject = true
16:          BREAK
17:        end if
18:        Check for collision
19:        if no collision then
20:          Add A to placedObjectArray
21:          generateObject = true
22:          BREAK
23:        end if
24:      end for
25:    else
26:      Add A to placedObjectArray
27:      generateObject = true
28:    end if
29:  end if
30: end for
```



(a) Erstes Objekt wird fallen gelassen vom Sampling-Punkt (rot).

(b) Folgende Objekte werden angehängt.

(c) Kraft wirkt entgegengesetzt der Gravitation, für einen kurzen Sprung.



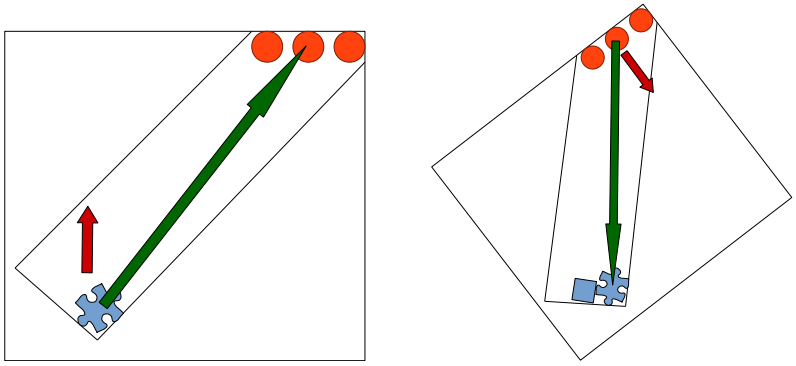
(d) Rigid-Body Simulation des hüpfenden Objektes.

(e) An den Sampling-Punkten können keine Objekte überlappungsfrei platziert werden.

(f) Nächste freie und tiefste Zelle wird gewählt.

Abbildung 3.4.: Skizze zu dem Hybrid Algorithmus.

Bevor die Objekte hüpfen, werden sie an ein mögliches vorheriges Objekt angehängt (siehe Abbildung 3.4b). Damit soll eine längere Gravitationssimulation, wie sie im Schritt 3.4a stattfindet verkürzt werden. Um dennoch die Wirkung der Gravitation mitzunehmen und die Fallzeit zu reduzieren, werden sie leicht zum Hüpfen gebracht mit Hilfe einer Hüpf-Kraft (siehe Abbildung 3.4c).



(a) Vektor zum Hüpfen (grün), statt in vertikale Richtung (rot)

(b) Neuer Gravitationsvektor (grün) ist entgegengesetzt dem Hüpf-Vektor und ersetzt vorherigen Gravitationsvektor (rot)

Abbildung 3.5.: Anpassung an den Container mit Hüpf- und Gravitationsvektor

Nach dem ersten platzierten Objekt in einer Zelle, wird der Vektor für zum Hüpfen bestimmt. Der Vektor zeigt vom ersten platzierten Objekt zum verwendeten Sampling-Punkt (siehe Abbildung 3.5a). Es ist eine Anpassung an den Container, um eine dichtere Packung zu erzeugen. Auch wird ein neuer Gravitationsvektor bestimmt, der nach dem hüpfen wirkt. Dieser ist entgegengesetzt der Hüpf-Kraft. Eine Vorstellung ist, das man den Container rotieren würde (siehe Abbildung 3.5b). Nachdem das erste Objekt von einem Punkt fallen gelassen wurde, beobachtet man an welcher stelle es gelandet ist und versucht mit diesen Informationen eine optimale Rotation des Containers zu erreichen.

Zum hoch Hüpfen des Objektes, findet eine Rigid-Body Simulation statt. Auf dem Objekt wirkt kurzzeitig eine große Hüpf-Kraft und fällt auch wieder schnell. Beim Fallen stößt es gegen andere Objekte oder den Container und findet so eine passende Rotation und Stelle (siehe Abbildung 3.4d).

Ob eine Zelle voll ist, wird daran festgemacht, ob an den Sampling-Punkten ein Objekt ohne Überlappung platziert werden kann (siehe Abbildung 3.4e). Wenn die Zelle voll ist, wird die nächste freie und tiefste Zelle gewählt. Mit der Tiefe ist

hierbei die vertikale Richtung bzw. Gravitationsrichtung gemeint. Dadurch wird der Container gleichmäßig und lokal optimal gefüllt.

Der Hybrid Algorithmus lässt sich wie folgt zusammenfassen:

Algorithm 5 Hybrid Algorithmus

Input: $objectTypeInfoArray, Container, pointsInContainerArray, grid(x, y, z)$
Output: *placedObjectArray*

```
1: for each lowest and free cell in grid do
2:    Search for top points in cell
3:    object = Generate a random object
4:    isFittingCell = Search for overlap free placement of object at top
5:    if isFittingCell = false then
6:       CONTINUE
7:    end if
8:    while isFittingCell do
9:       if object slowed down from RB-Simulation then
10:         object = Generate a random object
11:         isFittingCell = Search for overlap free placement of object at top
12:         if isFittingCell = false then
13:            BREAK
14:         end if
15:         Attach object to some object in cellObjectArray
16:         Set for object a jump force
17:      end if
18:      Do a rigid-body simulation with object
19:      Resolve the left overlap from simulation
20:      Place object in cellObjectArray
21:      if first object in cell then
22:         Calculate new gravity and jump force
23:      end if
24:   end while
25: end for
```

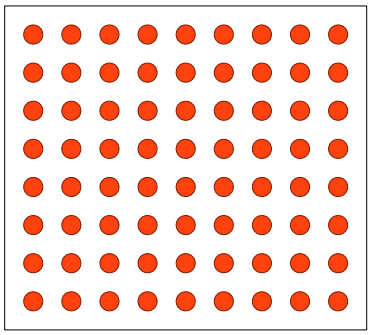
3.2.4. Growing Seeds Algorithmus

In der initialen Phase werden an jeden Punkt innerhalb des Containers zufällige Objekte positioniert (siehe Algorithmus 6). Die Verteilung der Objekte wird dabei berücksichtigt. Zu Beginn werden die Objekte uniform klein skaliert. Dabei überlappen sich die Objekte nicht. Der verwendete initiale Skalierungsfaktor in der Arbeit liegt bei $scaleBegin = 0,25$.

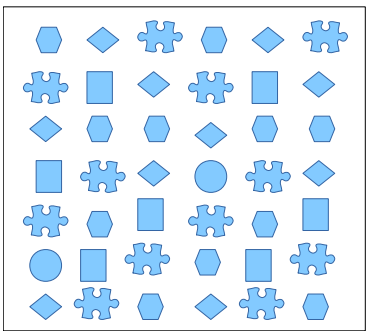
Algorithm 6 Initial step - Placing objects/seeds at each point

Input: *pointsInContainerArray*
Output: *placedObjectArray*
1: **for** each *point* in pointsInContainerArray **do**
2: *object* = Choose a random object regarding the distribution for objects
3: Set *object* to *position*
4: Scale *object* down
5: Choose a random rotation for *object*
6: Place *object* in *placedObjectArray*
7: **end for**

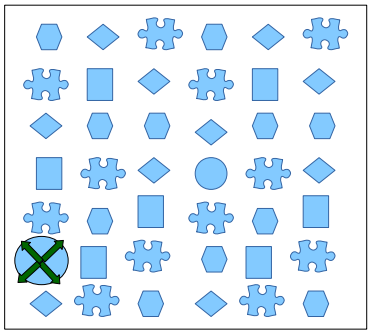
Im nächsten Schritt wird versucht die Samen bis zu ihrer originalen Größe wachsen zu lassen (siehe Algorithmus 7). Die originale Größe ist erreicht, wenn der Skalierungsfaktor des Objekts 1 beträgt. Es wächst genau ein Samen bzw. Objekt zu einem Zeitpunkt. Erst wenn das Objekt seine vollständige Größe erreicht bzw. nicht erreicht, wird das nächste Objekt gewählt.



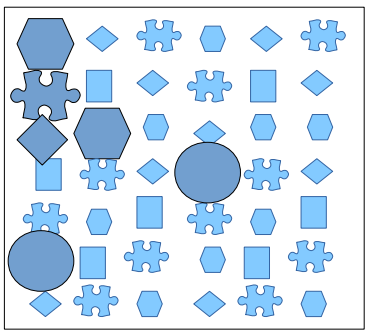
(a) Container Sampling



(b) Klein skalierte Objekte werden platziert



(c) Objekt bis zur originalen Größe skalieren



(d) Greedy Objektwahl in Abhängigkeit von Größe und angestrebter Verteilung

Abbildung 3.6.: Skizze zu dem Growing Seeds Algorithmus.

Die Wahl des Objektes erfolgt *greedy* unter Beachtung des Volumen und der Verteilung. Es wird der Objekttyp mit dem größten Volumen gewählt, für den die momentane Verteilung kleiner, als die angestrebte Verteilung ist. Zu der momentanen Verteilung zählen Objekte die vollständig ausgewachsen sind. Hier werden größere Objekte bevorzugt zuerst wachsen gelassen mit dem Gedanken, dass es für große Objekte schwieriger ist einen Platz zu finden (*large first heuristic*). Danach wird das erste Objekt was diesem Objekttyp entspricht wachsen gelassen. Das Wachstum nimmt um den Faktor $scaleStep \cdot \Delta t$ zu. Der Wert für $scaleStep = 0,25$ wurde durch experimentieren ermittelt. Nach jedem Wachstumsschritt werden die Kollisionen mit Hilfe einer vereinfachten Rigid-Body-Simulation aufgelöst. Beim der eulerschen Integration wird das gleiche Δt verwendet wie zur Skalierung der Objekte. Wenn ein Überlappungsfreier Zustand innerhalb der Zeit $t = 30s$ nicht erreicht wurde, dann wird das aktuelle Objekt verworfen. Der zusammengefasste Algorithmus zum wachsen lassen der Objekte ist in Algorithmus 7.

Algorithm 7 Growing objects to their original size

Input: *placedObjectArray*
Output: *placedObjectArray*

```
1:  for i = 0 to i < placedObjectArray step i++ do
2:    Choose greedy the object type considering volume and distribution
3:    Select first not grown object/seed with that type
4:    for s = scaleBegin to scaleEnd step s += scaleStep · Δt do
5:      Scale object to size s
6:      Try to resolve overlap within time t
7:      if overlap not resolved in time t then
8:        Remove object from placedObjectArray
9:      end if
10:   end for
11: end for
```

3.3. Verbesserungsheuristiken

In diesem Abschnitt werden zwei Verbesserungsheuristiken vorgestellt. Die Verbesserungsheuristiken beginnen mit einer bereits vorhandenen Packung und versuchen diese lokale oder global zu optimieren. Das Ergebnis ist eine verbesserte Packung.

3.3.1. Cavity Filling Algorithmus

Oft haben die generierten Packungen der Algorithmen in 3.2 kleine Hohlräume. Der Idee beim Cavity Filling Algorithmus ist, diese mit neuen Objekten zu füllen. Dabei existieren folgende Probleme:

- Die Hohlräume können sich an beliebigen Stellen innerhalb der Packung befinden
- Die Größe und Form der Hohlräume ist variable

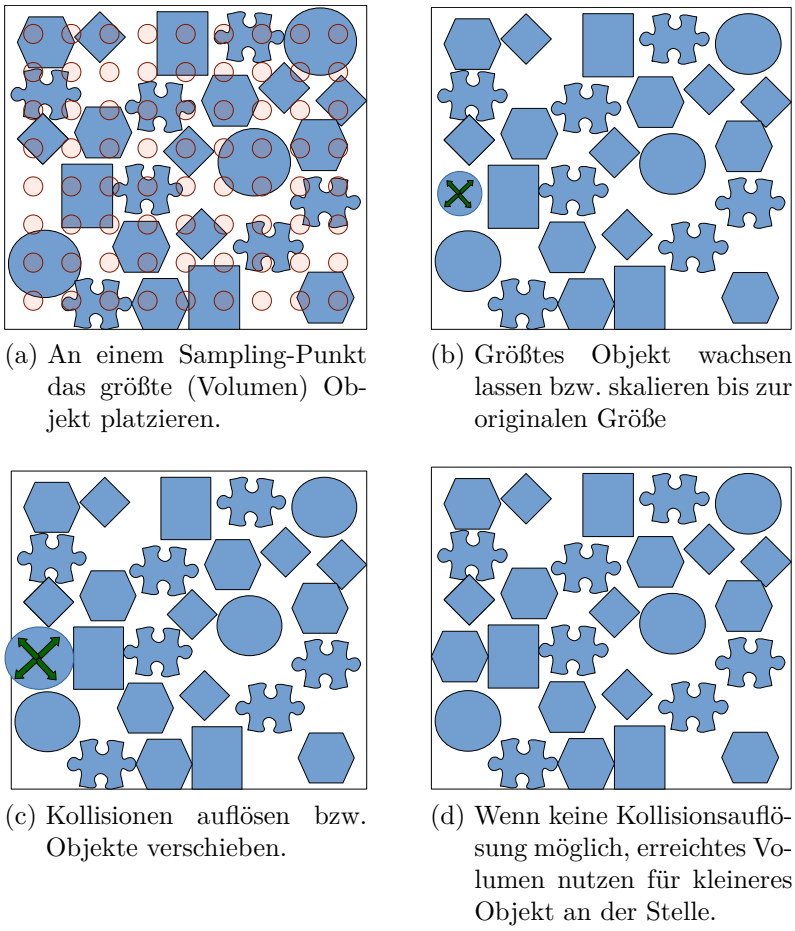
Zur Lösung des ersten Problems werden die Sampling-Punkte verwendet (siehe Abbildung 3.7a). Genau die Hohlräume, in denen sich auch die Sampling-Punkte befinden, werden geschlossen. An jedem Sampling-Punkt wird das Objekt mit dem größten Volumen platziert. Die initiale Skalierungsfaktor des Objektes ist klein (Skalierung $0,25$). Danach wird versucht das Objekt bis zur seiner originalen Größe (Skalierung 1) zu skalieren. Beim skalieren werden umliegende Objekte verschoben. Falls in innerhalb einer gewissen Zeit t die Kollisionen aufgelöst werden konnten, dann wird es nicht verworfen. Ansonsten ist das Objekt vermutlich zu groß, um den Hohlraum zu füllen. Mit Hilfe des erreichten Volumens, wird eine Entscheidung für das nächst kleinere Objekt getroffen.

Angenommen das Objekt wird verworfen und hat den Skalierungsfaktor s erreicht. Das originale Objektvolumen sei auch bekannt V. Dann lässt sich skalierte Volumen V_s, aufgrund der uniformen Skalierung, mit der Gleichung 3.6 berechnen. Da das Objekt verworfen wird, werden auch mögliche Überlappungen des Objektes (V_o) miteinbezogen. Das tatsächliche freie Volumen V_t wird in der Gleichung 3.7 bestimmt.

$$V_s = s^3 \cdot V \tag{3.6}$$

$$V_t = V_s - V_o \tag{3.7}$$

Damit werden kleinere Objekte übersprungen, deren originales Volumen größer ist als V_t. Es ist unwahrscheinlicher, dass das Objekt in dem Hohlraum platziert werden könnte. Das setzt voraus, dass die Form gleich oder ähnlich ist. Wegen dem zeitlichen Rahmen, wurde es bei der Volumeninformation belassen.



(a) An einem Sampling-Punkt das größte (Volumen) Objekt platzieren.

(b) Größtes Objekt wachsen lassen bzw. skalieren bis zur originalen Größe

(c) Kollisionen auflösen bzw. Objekte verschieben.

(d) Wenn keine Kollisionsauflösung möglich, erreichtes Volumen nutzen für kleineres Objekt an der Stelle.

Abbildung 3.7.: Skizze zu dem Cavity Filling Algorithmus.

3.3.2. Shuffle Algorithmus

Das Ziel vom Shuffle Algorithmus ist es eine bessere Lösung mittels globaler Optimierung zu finden. Im Kern ist die Idee, eine vorhandene Packungs-Struktur aufzubrechen, neue Objekte hinzuzufügen und wieder zusammenzusetzen.

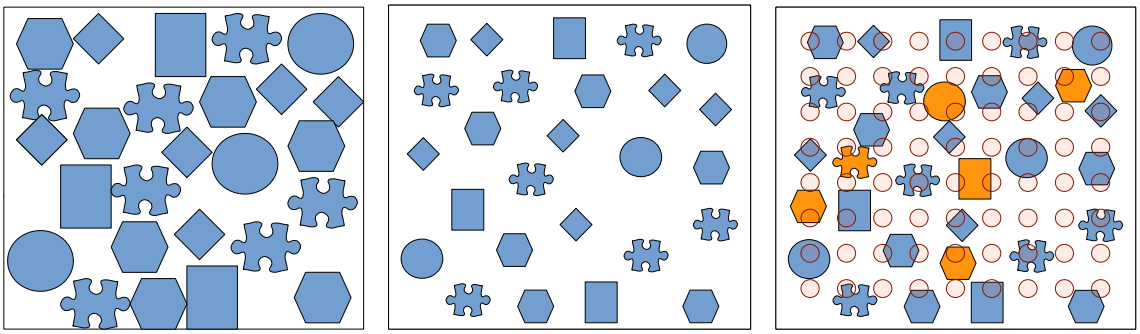
Im ersten Schritt werden alle Objekte einer Packung kleiner skaliert. Danach werden die Sampling Punkte verwendet, um an diesen Stellen neue Objekte einzufügen. Die neuen Objekte haben den gleichen kleinen Skalierungsfaktor (siehe Abbildung 3.8c). Da eine neue Struktur für die gesamte Packung gesucht bzw. erstellt wird, ist es nicht sinnvoll im ersten Schritt (oder immer) nur ein neues Objekt hinzuzufügen. Grundsätzlich erfordert die Bewegung aller Objekte viel Rechenzeit. Entsprechend wäre die Relation zwischen Verbesserung und Rechenzeit nicht unbedingt immer gut.

Ein guter Fall ist genau dann, wenn n Objekte einfügt werden und anschließend eine Lösung gefunden wird. Neben der Anzahl ist auch der Objekttyp entscheidend eine Lösung zu finden. Eine angemessene Menge von Objekten wird wie folgt berechnet: Man füge so lange zufällige Objekte in eine Menge M, solange bis $\sum_{x \in M} V(x) < p$ nicht mehr gilt. Die Variable p entspricht einem Anteil des Volumens vom Containers. Beim Experimentieren schien 10% ein guter initialer Wert zu sein.

Im nächsten Schritt wird versucht mit den neuen Objekten eine neue Packung zu erzeugen. Dazu werden die Objekte zufällig bewegt und gleichzeitig stoßen sich Objekte vom gleichen Typ ab. Beim letzteren ist das Ziel die räumliche Verteilung zu verbessern.

Anschließend werden die Objekte größer skaliert und es wird versucht die entstehenden Überlappungen aufzulösen. Falls nach einer bestimmten Zeit, die Überlappungen nicht aufgelöst werden können, dann werden die Objekte wieder etwas kleiner skaliert und zufällig bewegt. Es werden 3 Fehlversuche gewährt, bevor entschieden wird, dass die Überlappungen mit n neuen Objekten nicht aufgelöst werden können.

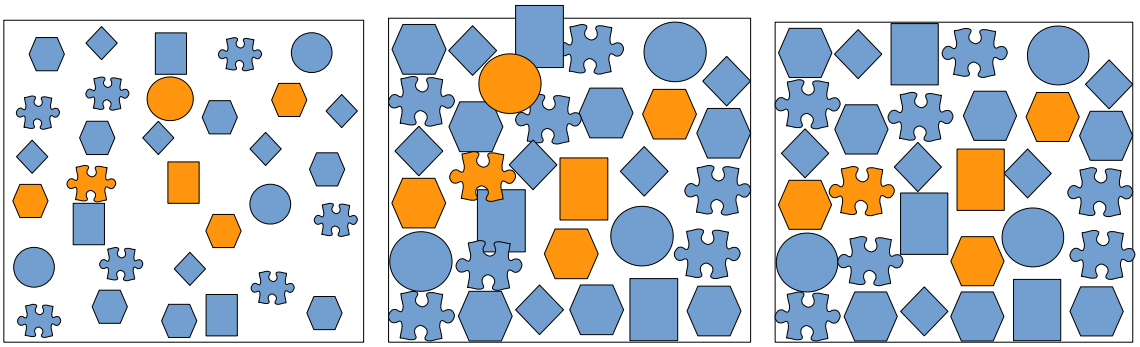
Beim Scheitern mit n Objekten wird das prozentuale Volumen des Containers p halbiert. Dann wird eine neue Menge M von zufälligen Objekten erzeugt. Diese besitzt ungefähr das halbe Volumen der vorherigen Menge. So wiederholt sich der Ablauf, solange das Volumen nicht kleiner, als vom kleinsten Objekt ist.



(a) Zu verbessernde Packung.

(b) Alle Objekte kleiner skalieren.

(c) An den Sampling-Punkten neue Objekte (orange) platzieren.



(d) Gleiche Objekt-Typen abstoßend und zufällig bewegen.

(e) Objekte größer skalieren und verschieben.

(f) Falls Kollisionen nicht aufgelöst werden können, dann mit weniger Objekten wiederholen.

Abbildung 3.8.: Skizze zu dem Shuffle Algorithmus.

4. Software-Architektur

In diesem Kapitel wird die gesamte Implementierung abstrakt betrachtet. In dem Kapitel 4.1 wird die Architektur der Software beschrieben. Damit soll zunächst ein Überblick geschaffen werden. Darauf folgend wird im nächsten Kapitel 4.2 das typische Anwendungsszenario kurz beschrieben.

4.1. Architektur der Software

Einen Überblick zu der gesamten Software und Abhängigkeiten erhält man in der Abbildung 4.1. Die entwickelte "Auto Packing" Software lässt sich in fünf Komponenten zerlegen. Im Kern befindet sich Komponente AutoPacking. Diese übernimmt das Laden von einer Konfigurationsdatei und den Modelldaten mit Hilfe von PackingUtils. Damit man das Ergebnis oder den Aufbau einer Packung beobachten kann, wurde ein einfaches Rendering implementiert. Basierend auf der neuen OpenGL Pipeline (*programmable pipeline*), lassen sich damit auch hoch aufgelöste Modelldaten schnell rendern. Um räumlichen Eindruck zu bekommen wurde das Phong lighting model implementiert. Ein wichtiger Bestandteil ist die Kollisionserkennung. Hierfür

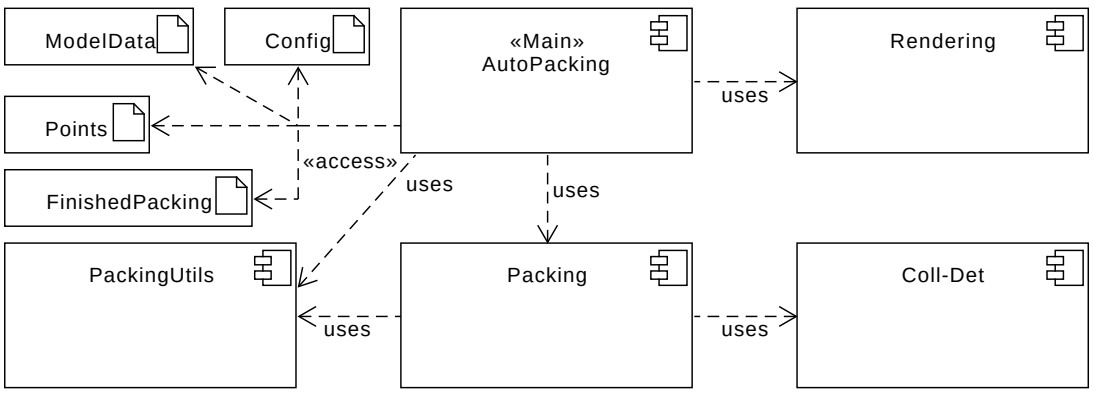


Abbildung 4.1.: Komponenten Diagramm der entwickelten Auto Packing Software

wird die Coll-Det Bibliothek verwendet. Die Packungs-Algorithmen befinden sich in der Komponente Packing. Als Hilfskomponente dient PackingUtils. Einige Aufgaben und Inhalte der Komponente: Laden von Modelldaten/Konfiguration, Sampling von Punkten innerhalb des Containers, Laden/Schreiben einer Packung, Wrapper und Datenstrukturen.

Um eine klare Trennung zwischen Konstruktions- und Verbesserungsheuristiken zu haben, wurden einige abstrakte Klassen definiert. In der Abbildung 4.2 ist das dazugehörige Klassendiagramm. Das Klassendiagramm ist auf das Nennenswerte gekürzt, es sind also nicht alle Attribute oder Methoden darstellt. In der abstrakten Klasse IPackingAlgorithm sind grundlegende Funktionalitäten implementiert, welche jedem Algorithmus zur Verfügung stehen und auch von außen genutzt werden können. Protected ist z.B. die Generierung des PackingObjekt.

Die Struktur PackingObject repräsentiert alle verwendeten Objekte und Container. Daher enthält sie Daten zu Objekttyp, Geometrie, Position, Farbe und weitere Attribute, die für eine Rigid-Body-Simulation notwendig sind. Es ist sinnvoll sich den den letzten überlappungsfreien Zustand zu merken bzw. dahin zurückzukehren, deswegen lässt sich der Zustand der Objekte speichern und laden.

Intern wird ein Gitter bzw. Gittergröße von einem Algorithmus genutzt, als auch von der Coll-Det Bibliothek. In Coll-Det kann damit die Anzahl der zu überprüfenden Kollisionsobjekt-Paare reduziert werden. Zur Evaluierung steht eine Funktion bereit, welche statisch relevante Daten ermittelt und an die Standardausgabe ausgibt. In Kapitel 5 werden die Daten zu weiteren Auswertung herangezogen.

Jede Konstruktionsheuristik erbt von der Klasse IConstructionAlgorithm und muss den Algorithmus in der Methode ConstructionHeuristic implementieren. Analog verhält es sich bei der Verbesserungsheuristik. Die Run Methode ruft die entsprechend implementierte Heuristik auf. Falls in der Konstruktionsheuristik eine Verbesserungsheuristik gesetzt wurde, so wurde diese im Anschluss ausgeführt.

Das Rendering und die Berechnung einer Packung sind von einander entkoppelt und laufen parallel in einem eigenen Thread. Dazu wird die Run-Methode als Thread gestartet.

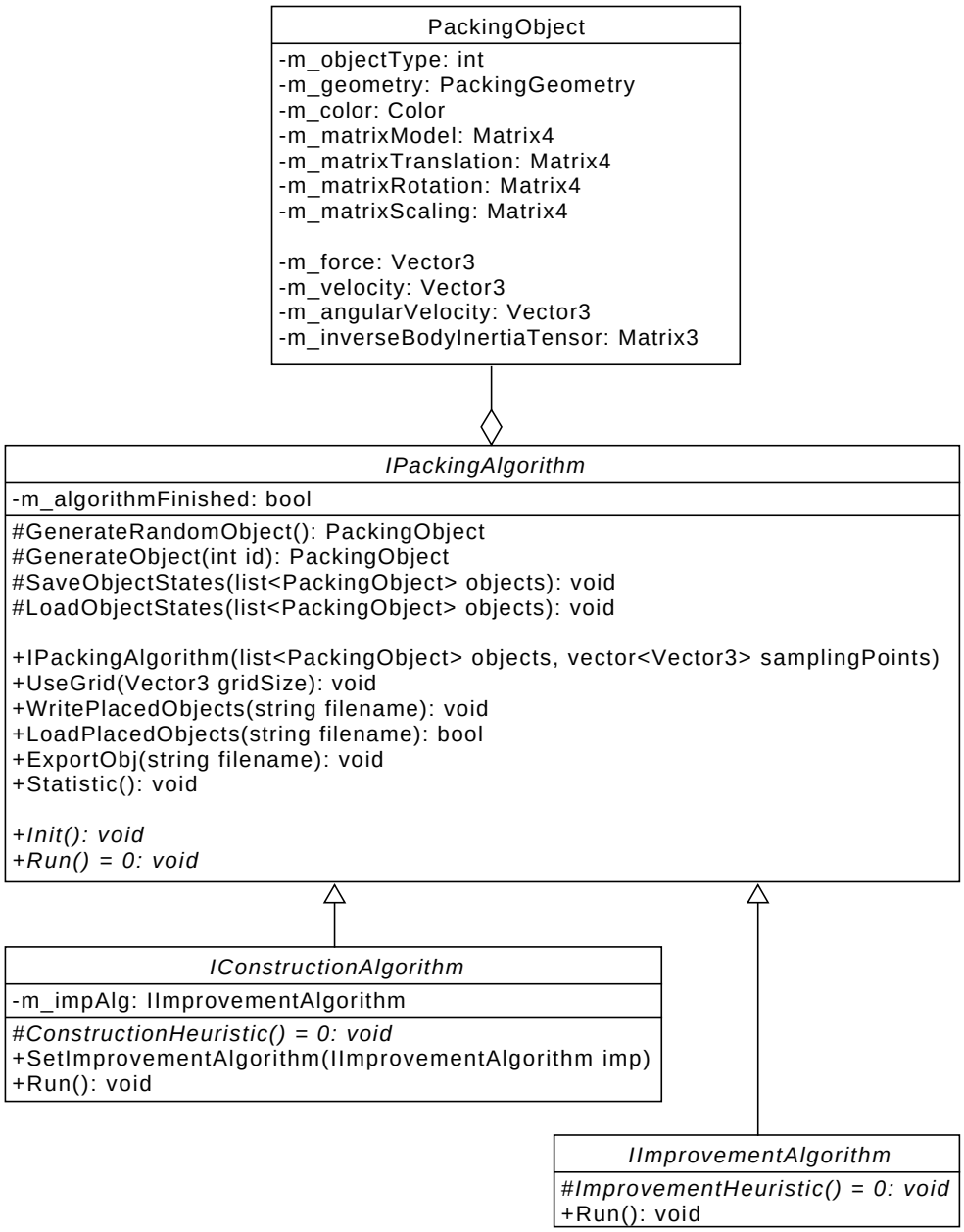


Abbildung 4.2.: Klassendiagramm zu den Algorithmen

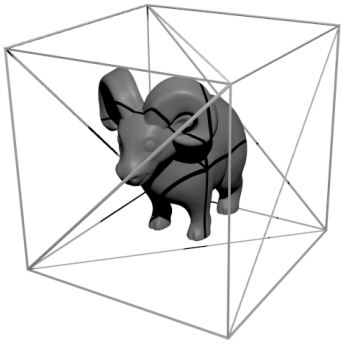
4.2. Erweiterungen für CollDet

Damit Kollisionen erkannt werden, wenn sich ein Objekt außerhalb oder am Rand des Containers befindet, wurde die verwendete Kollisionsbibliothek CollDet angepasst. Statt innerhalb des Objektes, wird außerhalb des Objektes eine Kugelpackung erzeugt. Die grundlegende Funktionalität war bereits bereits implementiert. Es musste nur in den gesamten Programmablauf entsprechend integriert werden. Um den Container herum wird eine größere Box platziert (siehe Abbildung 4.3a).

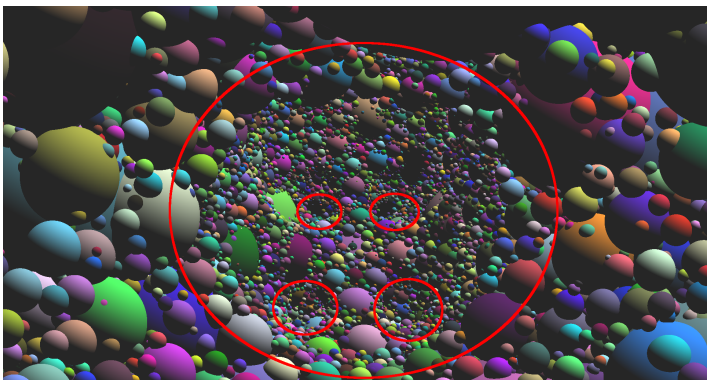
Sie dient als Grenze der äußeren Kugelpackung und sollte nicht zu klein gewählt werden. Es wird damit eine Art Wand dargestellt an der Objekte zurück ins innere gedrückt werden. Wenn sie zu klein bzw. dünn an gewissen Stellen ist, können Objekte durchdringen und landen nicht mehr im Container.

Protosphere füllt die Box und den Container mit Kugeln. Danach werden Kanten zwischen den Kugelpaaren berechnet. Falls ein Kugelpaar durch ein Dreieck getrennt wird, dann wird keine Kante gesetzt. Anschließend werden die verbundenen Komponenten bzw. Subgraphen berechnet.

Alle berechneten Graphen werden in Dateien geschrieben. Wenn alle Modelle geschlossen sind, würde man annehmen, dass es genau Zwei Komponenten geben müsste. Tatsächlich kommt es häufig vor, dass es mehr als zwei Komponenten gibt. Überwiegend sind es zwei größere Komponenten und die restliche Komponenten bestehen oft aus einer Kugel. Dies wurde nicht tiefer untersucht, weil beim Testen keine Auswirkungen festgestellt wurden, die darauf zurückzuführen sind. Die Kugeln gehören scheinbar auch immer zur inneren Container-Packung.



(a) Containermodell (Schafbock) in einer Box.



(b) Blick zum Unterkörper aus dem inneren des Kopfes. Rot markiert sind die Beine und der Hals. 92.516 Kugeln

Abbildung 4.3.: Kugelpackung zwischen dem Containermodell und der Box.

Einige Algorithmen verwenden uniforme Skalierungen auf die Objekte an. In CollDet wird eine Skalierung nicht direkt unterstützt. In homogenen Koordinaten ergibt sich die Model Matrix M aus einer Translations-Matrix T, Rotations-Matrix R und einer uniformen Skalierungs-Matrix S: $M = T \cdot R \cdot S$. Die Mittelpunkte der Kugeln in einer Kugelpackung werden mit dieser Matrix multipliziert. Der Radius bleibt jedoch unverändert. Für ein Objekt A mit der uniformen Skalierung $s_A = 1$ und ein Objekt B der uniformen Skalierung $s_B = 0,5$ ergeben sich zwei Fälle. Je nach dem von welcher Basis in die andere gewechselt wird, ergibt sich eine unterschiedliche Kollisionserkennung. Die Basiswechselmatrix aus der Gleichung 3.3 in Kapitel 3.2.2 lässt sich bei einer bloßen uniformen Skalierung einfacher ausdrücken[1]:

$$s_{s_2}^{s_1} = (s_2)^{-1} \cdot s_1 = \frac{s_1}{s_2} \tag{4.1}$$

Die Skalare s_1 und s_2 entsprechen den Skalierungsfaktoren eines Kollisionspaares. Bei dem Kollisionspaar (A, B) wird A doppelt so groß in der Basis von B, was zu einer porösen Kugelpackung führt. Anders sieht es beim Kollisionspaar (B, A) aus. Hier ist B halb so groß wie A in der Basis von A, was zu einer Kugelpackung führt die über das Modell hinaus geht. Einfach ausgedrückt: Es werden Kollisionen erkannt, wo es tatsächlich keine Kollisionen gibt, oder es werden keine Kollisionen erkannt, wo es tatsächlich Kollisionen gibt. Da die Objekte wachsen bzw. in der finalen Packung alle Objekte die originale Größe besitzen, ist garantiert, dass diese Packung frei von Kollisionen ist. Je näher die Objektskalierung an die originale Größe strebt, desto genauer wird die Kollisionserkennung.

Um eine exakte Kollisionserkennung zwischen einem Kollisionspaar mit beliebiger uniformer Skalierung zu berechnen, wurde eine Skalierung der Kugelradien in CollDet implementiert.

[1] Eine uniforme Skalierung lässt sich auf eine Multiplikation mit einem Skalar reduzieren.

4.3. Standardablauf

In diesem Kapitel wird das typische Anwendungsszenario beschrieben. Dazu werden die Ablaufschritte nummeriert aufgelistet. Das Ziel ist eine Packung für einen Container und einer Menge von Objekten zu erzeugen. Die Ablaufschritte sind wie folgt:

1. Modeldaten der Objekte liegen im „.obj“ Dateiformat vor mit dazugehöriger „.sphere“ Packung vor
2. Für den Container wird die äußere Packung „.sphere“ verwendet
3. In der Konfigurationsdatei wird festgelegt:
   - Der zu verwendende Container
   - Die zu verwendenden Objekte mit der prozentualen Verteilung
   - Die zu verwendende Konstruktionsheuristik und Verbesserungsheuristik
   - Die zu verwendende Sampling-Auflösung
   - Der Name der Ausgabedatei wird festgelegt
4. Das Auto Packing Programm wird ausgeführt

Nach Durchführung dieser Schritte liegt dem Akteur eine Ausgabedatei bzw. Packung im „.obj“ Format vor. Im Anhang sind die Parameter einer Beispiel-Konfiguration beschrieben.

5. Evaluation

Die beschriebene Algorithmen in dieser Arbeit werden in diesem Kapitel bewertet. Zu Beginn werden die getesteten Szenarien vorgestellt. Danach wird Kollision mit dem Container genauer betrachtet. Anschließend folgen Kapitel zu den einzelnen Phasen der Algorithmen (Preprocessing-Konstruktionsheuristik-Verbesserungsheuristik). Getestet wurde auf der Plattform: Linux (Kernel 5.0.7), CPU: AMD Ryzen 7 2700X Eight-Core Processor, Arbeitsspeicher: 32 GB, Grafikkarte: NVIDIA GeForce RTX 2070.

5.1. Problemszenarien

Zur Evaluierung der Algorithmen wurden 6 verschiedene Problemszenarien betrachtet. In der Abbildung 5.1 sind die dazugehörigen Container und Objekte. Gleichzeitig sind es die erzeugten Packungen aus Kapitel 5.5.1. Im Anhang sind höher aufgelöste Bilder zu den Beispielen mit vielen Objekten (siehe A.3).

Die ersten drei Szenarien bzw. Modelldaten sind aus der Arbeit in [Ma et al., 2018]: Abb. 5.1a, 5.1b, 5.1c. Das Beispiel Box/Polyhedra wird in [Ma et al., 2018] und [Romanova et al., 2018] betrachtet. In [Romanova et al., 2018] sind die Modell-Daten als Punkte aufgelistet. Mit einem Script wurde die dazugehörige konvexe Hülle berechnet und in ein OBJ-Format exportiert.

Ein echter Vergleich kann nicht zu allen Beispielen gezogen werden, da z.B. nicht der gleiche Problemtyp verfolgt worden ist (siehe Tabelle 5.1). In [Romanova et al., 2018] wurde ein minimaler Container für eine feste Menge gesucht. Der Unterschied von [Ma et al., 2018] zu dieser Arbeit ist, dass es dort keine prozentuale Zielverteilung gibt. Dort ist die Verteilung scheinbar zufällig, da die Objekte initial zufällig gewählt werden. Eine Ausnahme ist das Beispiel Torus/Dog, hier stimmen die Modellmaße und die Verteilung spielt keine Rolle, da es sich um ein einziges Objekt handelt. Zu dem Beispiel Vase/Herzen konnte eine höhere Packungsdichte mit weniger Objekten erzielt werden. In dem Beispiel ist nur die Skalierung von Objekten oder Containern

Attr. (this work / other)	Vase/Hearts (≈)	Box/Chess (≈)	Torus/Dog	Box/Polyhedra
Time [min]	13.8 / 29	23.8 / 40	36.2 / 58	30.4 / 45 / 1131*
Packing Density [%]	42.2 / 40.3	24.4 / 32.5	15.5 / 19.6	37 / 51.3 / 53.9*
# Objects	107 / 125	47 / 60	53 / 67	71 / 77 / 80*

Tabelle 5.1.: Ungefährer Vergleich mit Ergebnissen aus anderen Arbeiten [Ma et al., 2018], [Romanova et al., 2018]. Die ersten zwei Beispiele haben nicht die gleichen Modellmaße (≈). * aus [Romanova et al., 2018] mit Ziel: Container-Minimierung.

nicht identisch.

Zu allen Modellen wurden manuell mit Protosphere Kugelpackungen erzeugt und anschließend visuell überprüft. Für die Objekte wurden Packungen mit $\approx 1 \cdot 10^4$ Kugeln und für die äußere Container-Packung $\approx 1 \cdot 10^5$ Kugeln erzeugt.

Für die Coffin-Modelle (siehe 5.1d) wurde von jedem Frucht-Typ eine Frucht gewählt. Es wurden 31 verschiedene Früchte verwendet (insg. 32 vorhanden). Verworfen wurde die Brombeere, da ihr Mesh sehr brüchig bzw. nicht wasserfest ist. In der Abb. 5.2 sind alle zur Verfügung gestellten Daten von Peter Coffin.

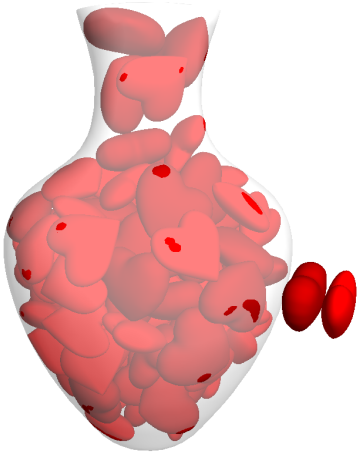
Das Beispiel mit dem Ziegenbock in der Abb. 5.1e wurde eigens zusammengestellt. Die dort verwendeten Modelle sind alle Lizenzfrei. Die goldene Münze in der Objekt-Menge wurde selbst modelliert.

Zu jedem sehr hoch aufgelösten 3D-Modell (fast alle Coffin-Modelle) wurde ein Modell mit reduziertes Polygon-Anzahl für das Rendering erzeugt. Jedoch wurden die Kugelpackungen mit den originalen Modellen erzeugt.

Die prozentuale Verteilung wurde gleichmäßig unter den Objekten aufgeteilt. Nur die die Beispiele mit der Hand und dem Ziegenbock sind Ausnahmen. Hier wurde den größeren Objekten ein höherer prozentualer Anteil gegeben.

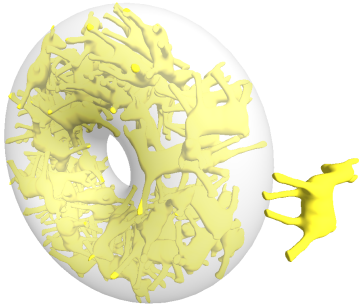
5.2. Kollision mit dem Container

Während dem Implementieren und Testen wurde erkenntlich, dass die Zeit zur Kollisionserkennung relativ hoch war, obwohl sich nicht viele Objekte in der Collision-Pipeline befanden. Beispielsweise läuft eine Rigid-Body-Simulation mit mehreren Objekten flüssiger, wenn die Kollisionserkennung für den Container aus ist. Die Kollisionserkennungen mit dem Container erhöhen sehr deutlich die Zeitkosten. Um das näher zu untersuchen wurde die Zeit gemessen für alle folgenden Kollisionspaare: (Objekt$_i$, Objekt$_j$) und (Container, Objekt$_i$) mit $i \neq j$. Die Zeit wurde direkt

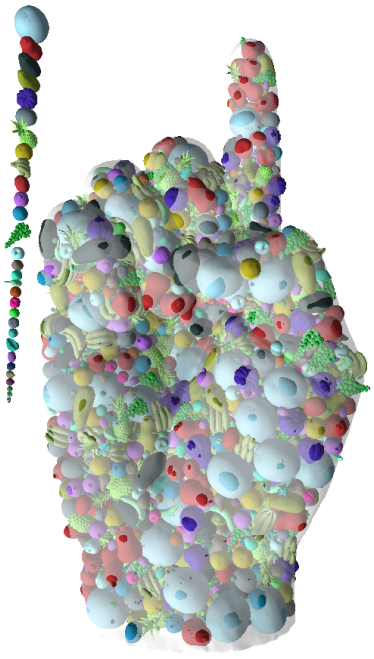


(a) Vase und Herzen. Packungsdichte $42,2\%$, # 107, Zeit $13,8$ min

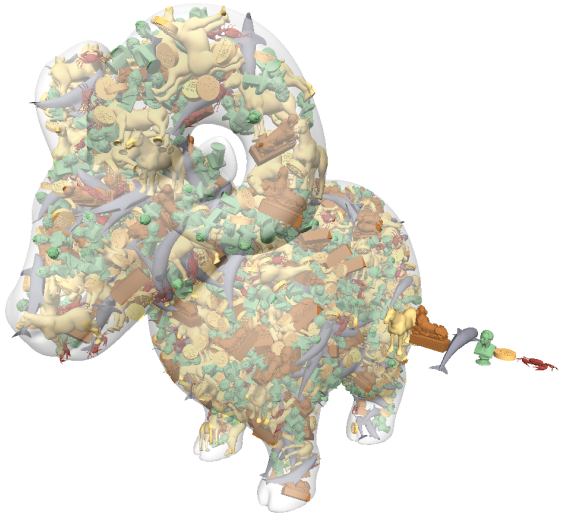
(b) Box und Schachfiguren. Packungsdichte $24,4\%$, # 47, Zeit $23,8$ min



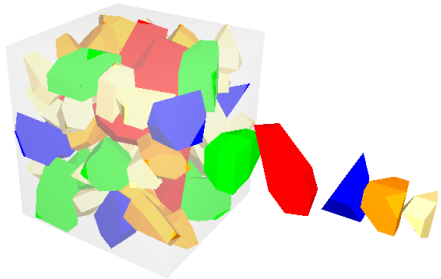
(c) Torus und Hunde. Packungsdichte 15.5%, # 53, Zeit $36,2$ min



(d) Hand und 31 Früchte. Packungsdichte $47,4\%$, # 3188, Zeit $354,5$ min



(e) Ziegenbock und 6 komplexe Objekte. Packungsdichte $25,4\%$, # 1990, Zeit $339,2$ min



(f) Box und 5 Polyeder. Packungsdichte 37%, # 71, Zeit $30,4$ min

Abbildung 5.1.: Ergebnisse mit relativ hoher Packungsdichte nach Anwendung von GrowingSeedsAlgorithm und lokaler Optimierung. Verwendete Objekte neben dem Container.

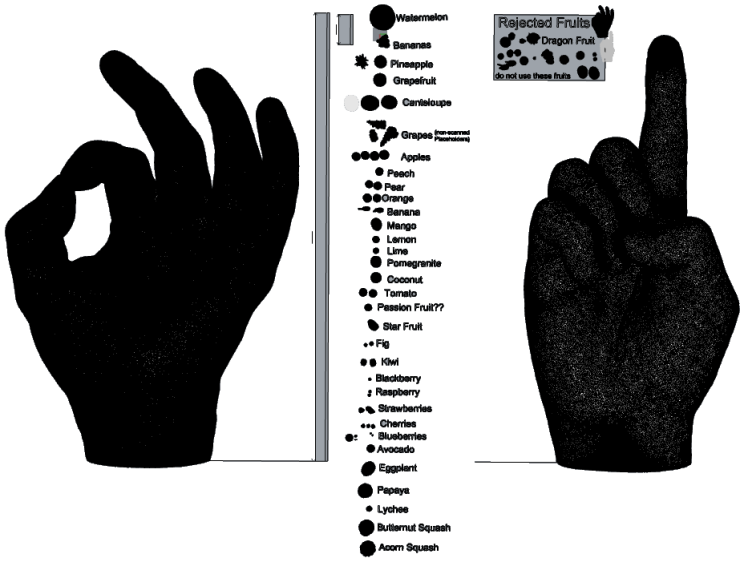


Abbildung 5.2.: Eine Menge von 3D-gescannten Modellen. Vom Künstler Peter Coffin zur Verfügung gestellt. Betrachtet in der Software Rhino3D.

in CollDet gemessen und ist somit nur die Zeit für eine Kollisionserkennung eines Paars. In der Abbildung 5.3 ist die Laufzeitverteilung für die Konstruktionsheuristiken ermittelt worden.

Die Diagramme 5.3 zeigen, dass die Kollisionserkennung mit dem Container einen sehr großen Anteil an der gesamten Laufzeit hat. Den zweitgrößten Anteil an der Laufzeit hat die Kollisionserkennung zwischen den Objekten. Die restliche Logik der Algorithmen, wie bspw. Initialisierung oder Auflösung der Kollision, machen den kleinsten Teil der Laufzeit aus. Der Distance Algorithmus hat eine relativ hohe Laufzeit zur Kollisionserkennung von Objekten (19.6%). Beim anheften eines Objektes an ein bereits platziertes Objekt, können Überlappungen mit anderen Objekten entstehen. Die Überlappung beim Anheften kann groß sein, was zu einer längeren Laufzeit führt. Eine Optimierung, wäre mit der Methode CheckDistance möglich. Die Methode CheckDistance bricht während der Traversierung sofort ab, wenn eine es Überlappung zwischen einem Kugelpaar existiert. Bei den anderen Algorithmen wird versucht vorhandene Überlappungen aufzulösen. Tendenziell sollte es daher nicht viele und große Überlappungen zwischen den Objekten geben. Durch Beobachtung lässt sich Abb. 5.3a und 5.3c erklären. Beim Hybrid Alg. wird ein Objekt fallen gelassen und verliert an Geschwindigkeit, es kommt dabei kaum zu großen Überlappungen. Dagegen wird bei Placement Alg. ein Objekt nach dem anderen in der originalen Größe Platziert. Je nach Größe/Form der Objekte und der Sampling-

	Collision Check				
		Object,Object (ms)		Container,Object (ms)	
Algorithm	# Checks	Avg.	Std.	Avg.	Std.
Placement	711.584	3.14	2.26	29.23	25.22
Distance	86.553	1.46	1.38	4.93	2.35
Hybrid	743.952	0.15	0.50	4.75	4.14
Growing	1.551.625	0.94	1.15	14.68	14.54

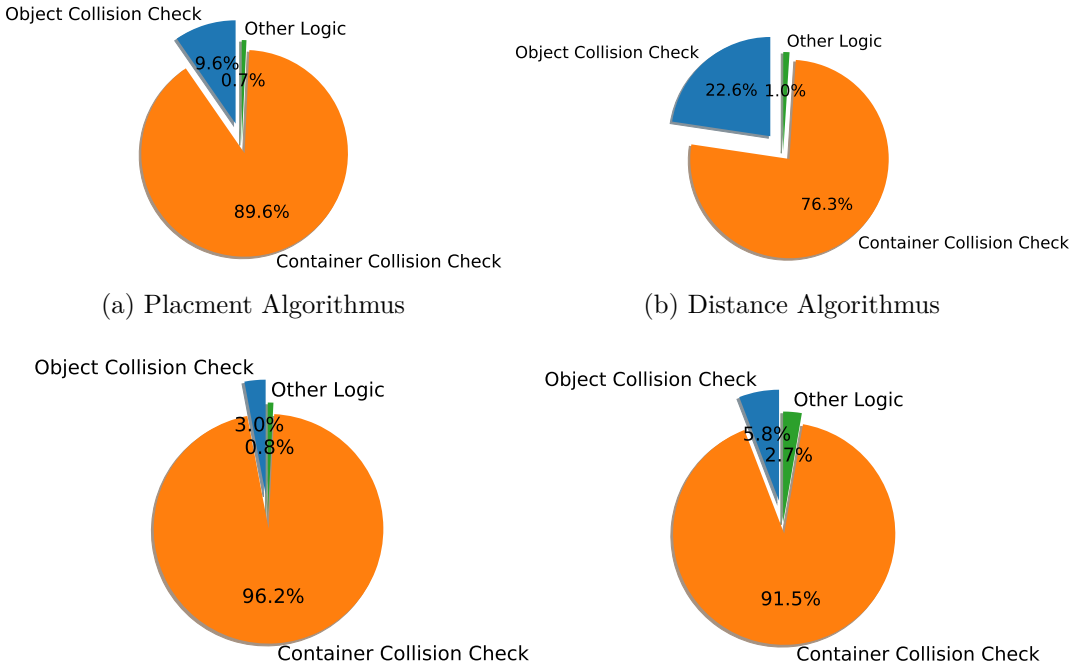
Tabelle 5.2.: Anzahl von Kollisionserkennungen und die durchschnittlich benötigte Zeit. Gemessen in 4 Szenarien mit gleicher Sampling-Auflösung pro Szenario.

Auflösung unterscheidet sich die Größe der Überlappung beim platzieren. Beim Growing Seeds Alg. ist es ähnlich. Hier wächst ein Objekt nach dem anderen. Durch Verschiebungen können noch nicht ausgewachsene Objekte verdeckt werden von bereits Ausgewachsenen Objekten. Die Kollisionerkennung wird für ein kleines Objekt aktiviert und es versucht in einem ausgewachsenen Objekt zu wachsen. Manchmal gelingt es den Objekten ausgestoßen zu werden. Der andere Fall ist, dass das Objekt nicht initial überlappt wird. Dann wird es in einem Tempo skaliert, was nicht zu großen Überlappungen und damit auch Suchzeiten führt.

In einem Versuch wurde ein BSP-Tree für den Container implementiert. Die wichtigste Methode die dabei entwickelt worden ist, ist die Kollisionsüberprüfung zwischen einem BSP- und Sphere-Tree. Es wurde getestet in einer Rigid-Body-Simulation, bei der ein Objekt fallen gelassen wurde. Dabei konnte man erkennen, dass die Kollisionserkennung mit dem Container nicht vollständig korrekt ist. Auch die Zeitkosten für die Kollisionserkennung wirkten hoch. Zumindest wenn alle Kugeln ermittelt werden, die eine Überschneidung mit einem Dreieck besitzen. Nach einem Fehler wurde gesucht, aber aus Zeitgründen wurde dieser Ansatz nicht mehr weiter verfolgt.

5.3. Preprocessing

Über die Hälfte der Zeit für diese Arbeit, war das Sampling ursprünglich eine teure Berechnung. Zuvor wurde statt einer Kugel, das kleinste Objekt aus der Objektmenge gewählt und damit das Sampling ausgeführt. Hierbei wurden zwei Sphere-Tree traversiert. Wenn das Objekt außerhalb des Containers liegt, so muss jedes mal der gesamte SphereTree von dem Objekt traversiert werden. Zeiten von bis



(a) Placment Algorithmus (b) Distance Algorithmus

(c) Hybrid Algorithmus (d) Growing Seeds Algorithmus

Abbildung 5.3.: Verteilung der durchschnittlichen Laufzeit für Konstruktionsheuristiken in verschiedenen Beispielen.

zu 20 Minuten wurden in vorherigen Experimenten gemessen. Daher kam die Idee, die berechneten Punkte in eine Datei zu schreiben. Bei einem erneuten Durchlauf, muss das Sampling nicht erneut ausgeführt werden, weil die Punkte aus einer Datei geladen werden[1].

Es folgten weitere Verbesserungen wie: Nutzung von der Methode checkDistance und Parallelisierung. Die Methode CheckDistance bricht bei der Traversierung sofort ab, wenn eine es Überlappung zwischen einem Kugelpaar existiert. Also eine Verkürzung der Suchzeit. Insbesondere, wenn das Objekt außerhalb des Containers liegt. In einem nachfolgenden Schritt wurde noch das Sampling parallelisiert.

Mit einer Kugel als Wurzel und Blatt, wird nur der SphereTree des Containers traversiert. Die Laufzeit des Sampling vom sich dadurch enorm verringert. Die benötigte Zeit liegt nun unter einer halben Sekunde. Die Grafik 5.4 zeigt die benötigte Zeit für das Preprocessing anhand von verschiedenen Beispielen. Die Anzahl der insgesamt durchgeführten Collision-Checks steht in der Legende. Dabei ist sichtbar, dass die benötigte Zeit kleiner wird, je mehr Threads verwendet werden. Jedoch ist

[1]So entstand der Name Preprocessing.

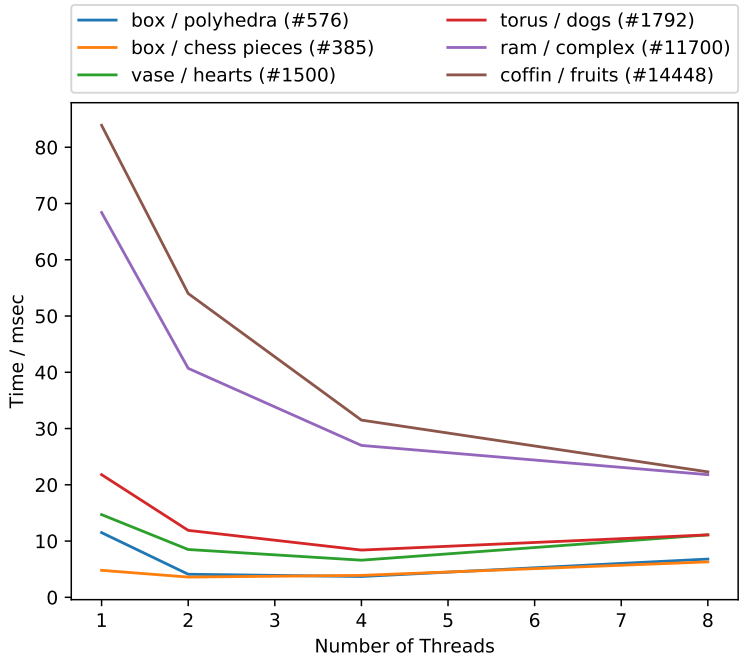


Abbildung 5.4.: Laufzeit des Samplings mit $n \in \{1, 2, 4, 8\}$ Threads für verschiedene Bespiele mit unterschiedlicher Anzahl von Collision-Checks.

das bei 4 Fällen mit 8 Threads nicht mehr der Fall. Grund dafür ist die niedrige Anzahl an Collision Checks und die benötigte Zeit zur Initialisierung der 8 Collision-Pipelines. Letzteres überwiegt und daher kann es bei wenig Collision-Checks sinnvoll sein, weniger zu parallelisieren oder ganz zu verzichten.

5.4. Konstruktionsheuristik

Um die Konstruktionsheuristiken zu evaluieren ,wurden verschiedene Packing Konfigurationen generiert und ausgeführt. Die variablen Parameter waren: Szenario, Konstruktionsheuristik, Anzahl Sampling-Punkte bzw. Schrittweite. Insgesamt wurden 261 unterschiedliche Konfigurationen erstellt und mehrmals ausgeführt. In einigen Fälle wurden spezielle Konfigurationen erstellt, z.B. für den DistanceAlgorithm mehr Sampling Punkte, da die Laufzeit sehr gering ist.

Um die Daten aus den vielen Log-Files in einen Zusammenhang zu bringen, wurden Auswertungsscripte in Python mit regulären Ausdrücken geschrieben und anschließend dazu die Plots erstellt.

5.4.1. Laufzeit

Als Laufzeit wurde gemessen, wie viel Zeit der Algorithmus benötigt, um eine initiale Packung zu erzeugen. Die Abbildung 5.5 zeigt die Entwicklung der Laufzeit in Abhängigkeit zur Anzahl von Sampling Punkten. Die Laufzeit steigt bei den meisten Algorithmen fast linear mit den Sampling Punkten an. Beim DistanceAlgorithm scheint die Laufzeit besonders linear zu sein. Das liegt vermutlich daran, dass der Algorithmus keine Überlappungen erlaubt. Das heißt, es findet keine Kollisionsauflösung statt, welche zufällig länger oder kürzer dauern kann.

Ganz anders ist der Verlauf beim HybridAlgorithm, hier wirkt die Laufzeit konstant bzw. unabhängig von den Sampling Punkten. Der HybridAlgorithm arbeitet fast vollständig in einem kontinuierlichen Raum. Nur vor dem Fallenlassen wird ein diskreter Punkt im Raum benötigt. So kommt eine die Unabhängigkeit von den Sampling-Punkten zustande.

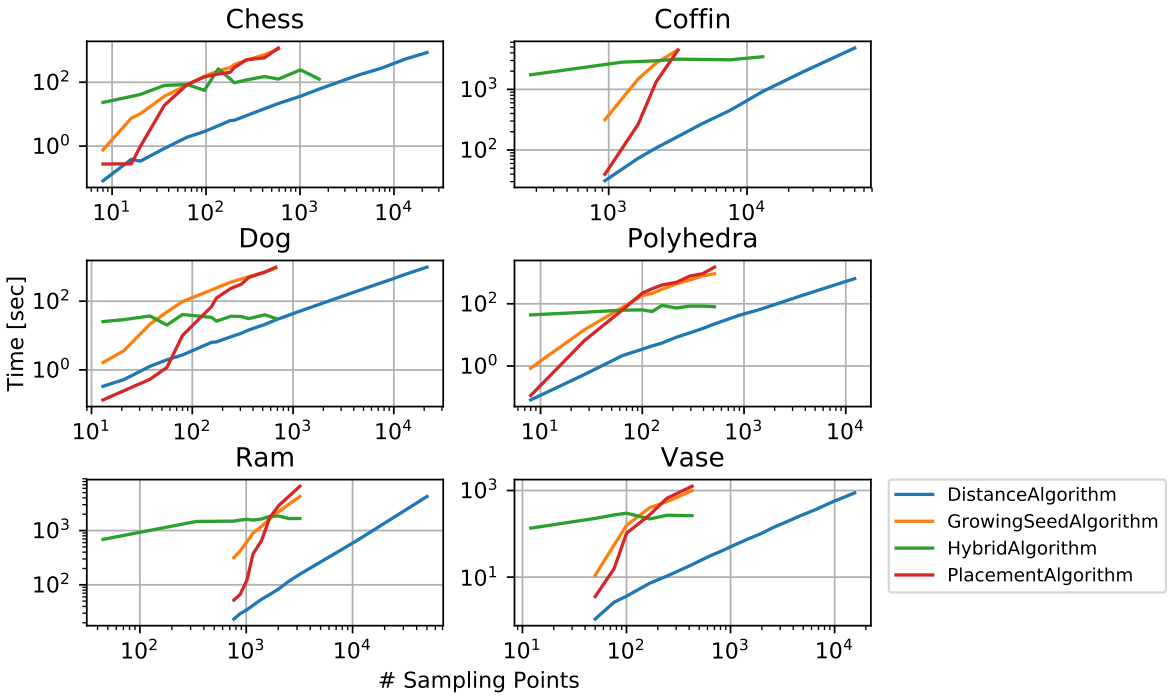


Abbildung 5.5.: Laufzeit in Abhängigkeit zu Sampling Punkten

5.4.2. Packungsdichte

Packungen mit möglichst hoher Dichte zu erzeugen ist ein Ziel dieser Arbeit. In den erzeugten Packungen wird die Dichte als ein prozentualer Wert berechnet. Der Wert ergibt sich wie folgt: $\sum_{o \in P} v(o)/v(c) \cdot 100$. Dabei ist P die Menge der platzierten Objekte innerhalb des Containers C und die Funktion v berechnet das Volumen.

In der Abbildung 5.6 ist Dichte im Verhältnis zur Laufzeit einer Ausführung dargestellt. Grundsätzlich lässt sich erkennen, dass die Dichte sich bei einer geringen Laufzeit stark erhöht. Aber bei einer höheren Laufzeit bzw. vielen Sampling Punkten steigt die Packungsdichte nur noch sehr wenig an. Eine etwas eigenen Verlauf hat hier der HybridAlgorithm. Das liegt daran, dass der Algorithmus weniger Abhängig ist von den Sampling Punkten.

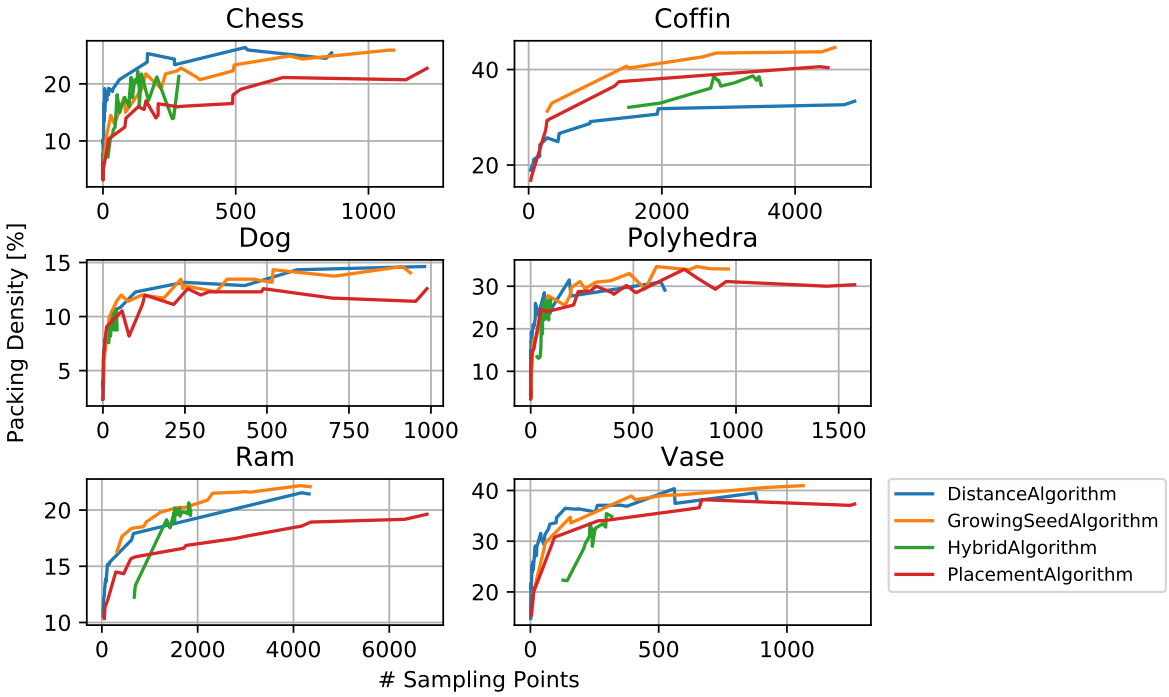


Abbildung 5.6.: Packungsdichte in Abhängigkeit zu Sampling Punkten

5.4.3. Abweichung der Verteilung

Zu Beginn wird in der Konfiguration eine prozentuale Verteilung von Objekt-Typen festgelegt. Für jedes Problemszenario ist die gewählte Verteilung fest. Das Ziel ist, dass die erzeugten Packungen die vordefinierte Verteilung weitestgehend einhalten.

Um die Verteilung auszuwerten, wird eine Abweichung berechnet. Diese besagt wie nah oder weit sich die Verteilung in der Packung von der festgelegten Verteilung befindet. Dazu wird der mittlere absolute Fehler verwendet, da dieser den genannten Zweck erfüllt und einfache Interpretation erlaubt: $MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$. Dabei ist n die Anzahl von Objekttypen i, $\hat{y}_i$ der angestrebte Prozentsatz und y_i der gemessene Prozentsatz. Dazu sind die Ergebnisse in der Abb. 5.7.

Die Abweichung beim Torus/Dog Beispiel ist trivial, da nur ein Objekttyp vorhanden ist und immer mindestens ein Objekt platziert wird. Am schlechtesten wirkt der PlacementAlgorithm bei der Verteilung und am besten der GrowingSeedsAlgorithm.

Auffallend sind die Verläufe in Coffin und Ram. Diese haben eine Ähnlichkeit, was scheinbar mit der hohen Anzahl an Objekten zusammenhängt. Wenn viele Objekte in einen Container passen, dann ist die Abweichung von der Zielverteilung gering. Bei wenigen Objekten, ist die Abweichung größer und schwankt mehr.

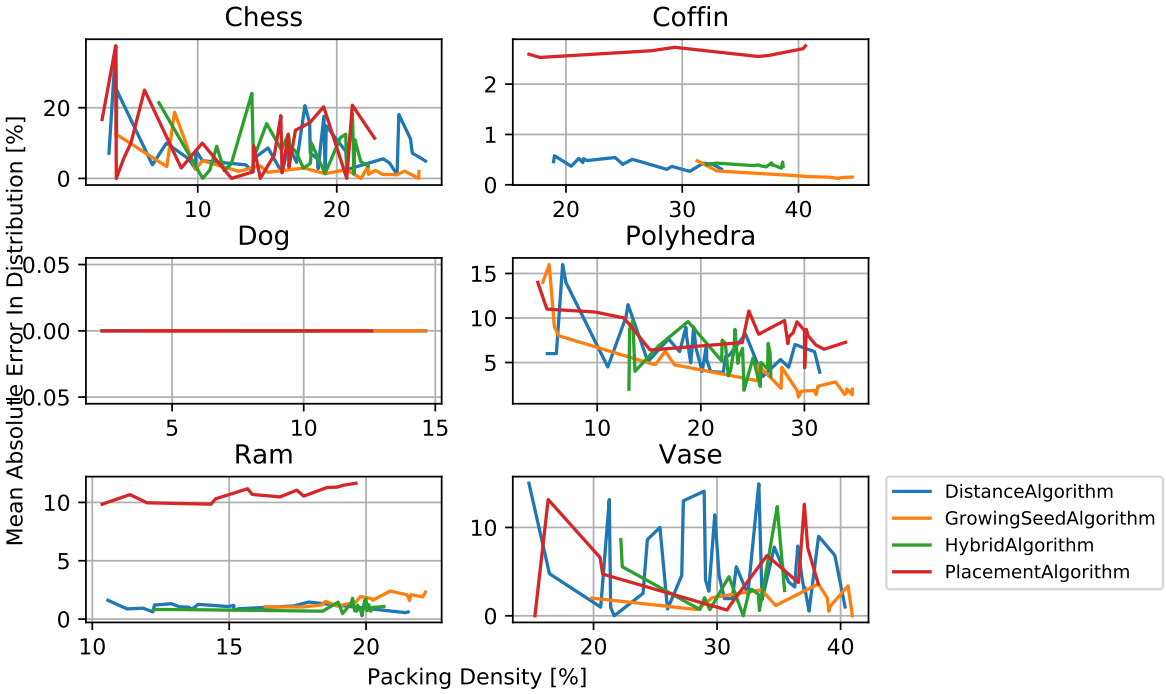


Abbildung 5.7.: Abweichung zwischen prozentualer Zielverteilung und erzeugter Verteilung. Berechnet als mittlerer absoluter Fehler.

5.4.4. Tendenz zu Clustern

Ein weiteres Ziel ist eine räumliche heterogene Verteilung von Objekttypen. Im Idealfall würden für einen Objekttyp, die Objekte einen Mindestabstand haben wie beim *Poisson-disc sampling*. Dazu könnte man dann versuchen eine Abweichung zu berechnen. Nur lässt sich so ein Mindestabstand sehr schwer bestimmen. Daher wurde nach einer anderen Bewertung für die räumliche Verteilung gesucht. Mit Hilfe der Hopkins Statistik misst man die Tendenz zum Clustern innerhalb einer Datenmenge. Die Nullhypothese ist, dass die Daten eine zufällige Verteilung besitzen[Jiawei Han et al., 2012]. Es muss lediglich eine Metrik für Daten vorhanden sein. Hierfür wird Distanzfunktion *checkDistance* verwendet. Die Idee bei der Hopkins Statistik ist relativ einfach. Es muss eine zufällige künstliche Verteilung erzeugt werden. Anschließend werden die Distanzen zum nächsten Nachbarn berechnet. Einerseits werden die Distanzen für die Daten innerhalb der originalen Menge (w_i) und andererseits für die Daten in künstliche Menge hin zu den Daten in der originalen Menge (u_i) bestimmt. Die Distanzen werden summiert:

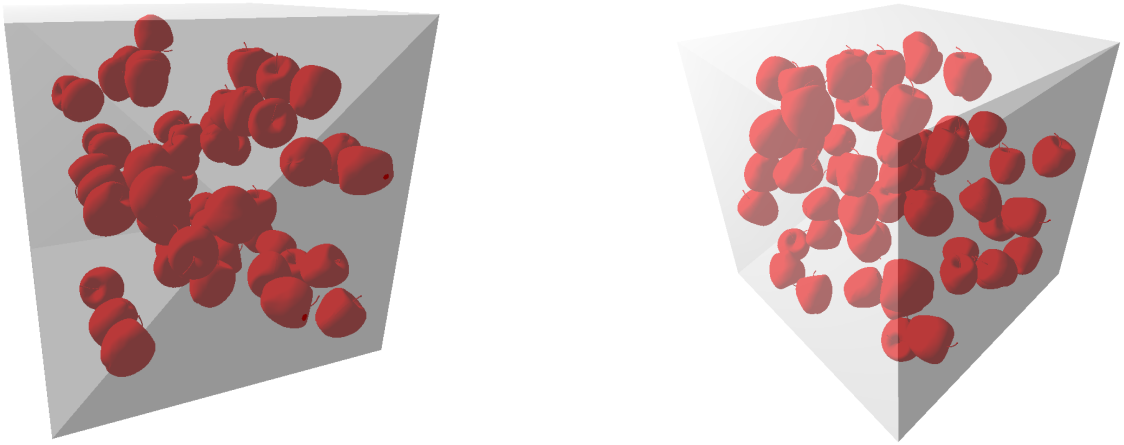
$$H = \frac{\sum_{i=1}^{m} u_i}{\sum_{i=1}^{m} w_i + \sum_{i=1}^{m} u_i} \tag{5.1}$$

Wenn beide Mengen zufällig sind, dann liegt der Hopkins Wert um 0.5 herum, ein Wert der in Richtung 1.0 geht, ist als starkes clustering zu interpretieren. Hingegen stellt ein Wert in Richtung 0 eine uniforme Verteilung dar.

In den Packungen mit wenig Objekten wurde die Hopkins Statistik für die gesamte Menge berechnet und 10 mal wiederholt, um ein konstantes Ergebnis zu bekommen. Für Coffin und Ram wurde 10% der Objektmenge verwendet und die Hopkins Statistik 3 mal berechnet. Aus den Hopkins Werten wurde dann der Durchschnitt gebildet, welcher in der Abb. 5.9 zu sehen ist. Dort sieht man, dass die räumliche Verteilung in allen Beispielen zufällig ist bzw. keine starken Cluster vorhanden sind. Die Unterschiede zwischen den einzelnen Algorithmen sind hier wohl eher vernachlässigbar.

Letztlich wird von den Algorithmen versucht, eine zufällige Verteilung zu erzeugen[2], um Cluster zu vermeiden bzw. eine annähernde gleichmäßige Verteilung zu erzeugen. Dennoch könnte es evtl. nach Verschiebungen von Objekten zu Clustern kommen, was hier jedoch nie aufgetreten ist.

[2]Zufällige Wahl und Platzierung von Objekten.



(a) Schlechte Verteilung künstlich erzeugt. Hopkins V.: 0.83

(b) Zufällige Verteilung. Hopkins V.: 0.52

Abbildung 5.8.: Durchschnittlicher Hopkins Value, für eine schlechte und zufällige räumliche Verteilung.

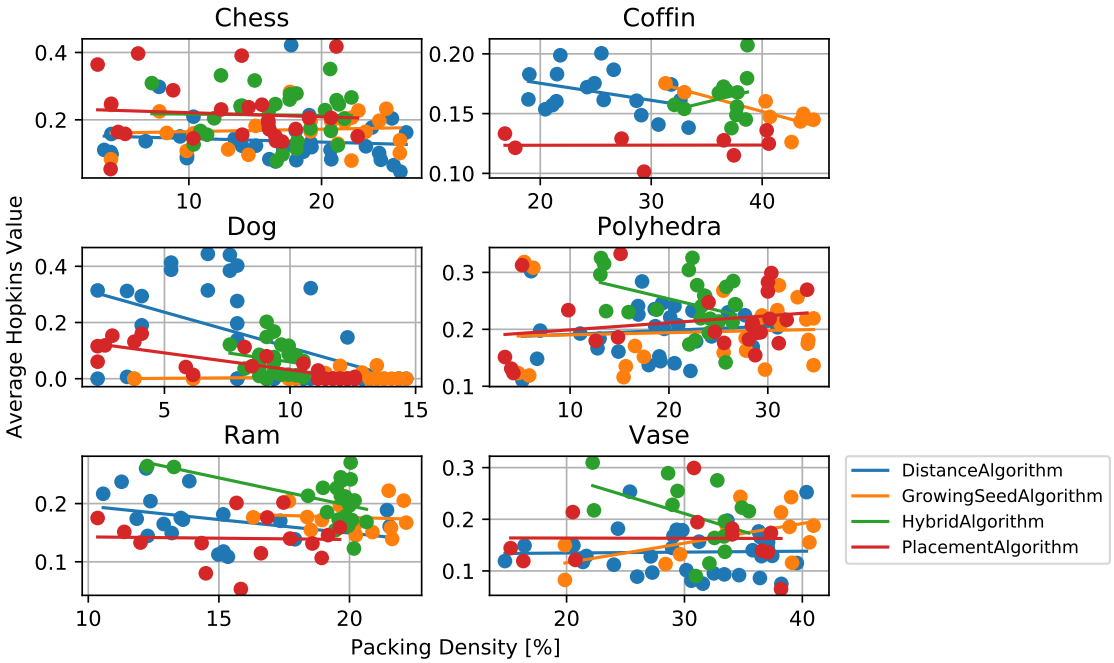


Abbildung 5.9.: Durschnittlicher Hopkins Value in Abhängigkeit zur Packungsdichte. Bedeutung Hopkins Vaule (0-1): 1 Daten clustern sehr stark, 0.5 Verteilung zufällig, 0 uniform Verteilt.

5.4.5. Oberflächendichte

Ein spezielles Kriterium, was im Laufe dieser Arbeit hinzugezogen wurde ist die Oberflächendichte. Dabei wird die Dichte für eine äußere Schicht mit der Tiefe ϵ bestimmt. Das ϵ ist hierbei relativ zum Container. Angenommen der Container ist gefüllt und hat die Skalierung s, dann wird der Container kleiner mit dem Faktor $1 - \epsilon$. Anschließend wird überlappende Volumen zwischen Objekten und Container aufaddiert und durch das Volumen der „dünnen“ Schicht dividiert.

Eine hohe Oberflächendichte wäre vermutlich interessant für Peter Coffin, da die Form der Container mit höherer Oberflächendichte präziser wird. Es gab kleinere Ansätze dies zu optimieren, jedoch wurde das Vorhaben aus zeitlichen Gründen verworfen. Dennoch wird hier die Oberflächendichte ausgewertet. In der Abb. 5.10 ist Oberflächendichte in Abhängigkeit zur Packungsdichte dargestellt. Dabei ist zu sehen, dass die Oberflächendichte steigt, wenn sich die Packungsdichte erhöht. Die Oberflächendichte und die Packungsdichte wirken linear zusammenhängend.

Dass die Form der Objekte eine Rolle spielt, wird hier auch erkennbar. Für Chess, Dog, Polyhedra wird eine relativ niedrige Oberflächendichte erreicht. Die Kombination aus Container und Objekte macht hier eine höhere Oberflächendichte schwierig. Wobei das Polyhedra-Beispiel eher als Sonderfall zu betrachten ist. Die Polyeder haben dort große ebene Flächen, die man für eine optimale Lösung möglichst Dicht an dem Container platzieren würde, um keinen Freiraum zu lassen. Daher schneidet wahrscheinlich das Ergebnis dieser Arbeit im Vergleich zu den Arbeiten in dem Fall verhältnismäßig schlecht ab (siehe Tabelle 5.1). Die höchste Oberflächendichte wird im Coffin-Beispiel erreicht, bei dem viele Objekte Kugelförmig sind.

5.5. Verbesserungsheuristik

In Unterkapitel werden die Verbesserungsheuristiken untersucht. Dazu wurde eine initiale Packung des GrowingSeedsAlgorithm genommen. Dabei wurde die Packung gewählt, bei der die Differenz zwischen Packungsdichte und Laufzeit am größten war. Es wurde also ein sog. Sweet Spot ermittelt und als Packung zur Verbesserung herangezogen. Ein Beispiel-Evaluierung dazu ist in der Abb. 5.11. Dort sind 3 Achsen mit jeweils einem eigenen Graphen eingezeichnet. Da die Laufzeit und die Packungsdichte unterschiedliche Einheiten haben, wurden die Werte vorher normalisiert mit

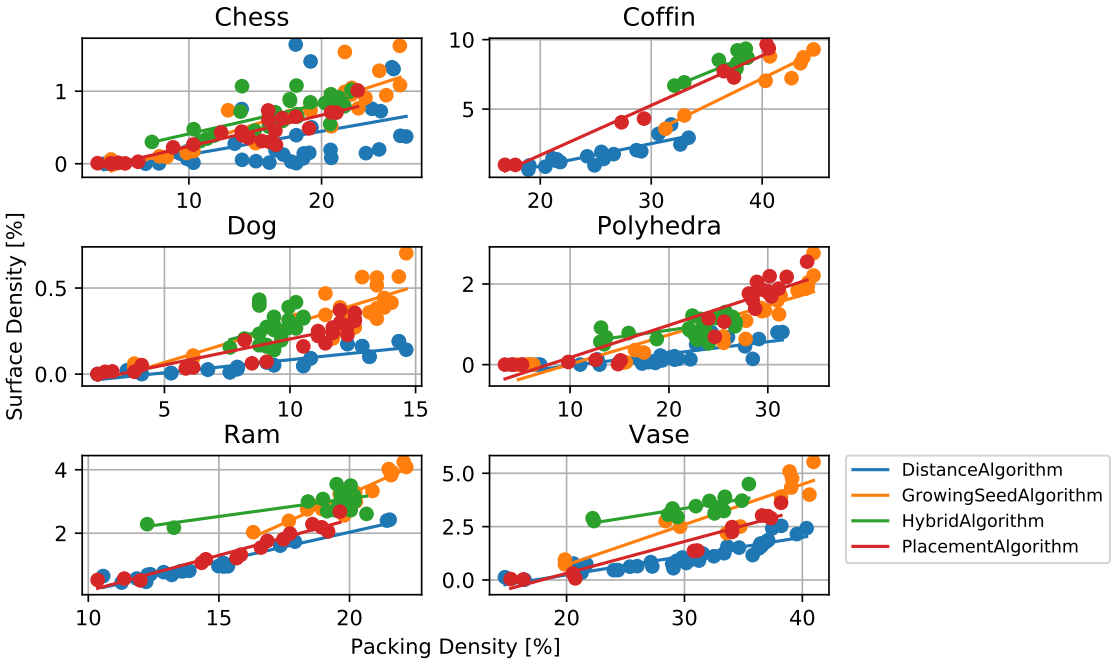


Abbildung 5.10.: Oberflächendichte mit $\epsilon = 0.05$ in Abhängigkeit zur Packungsdichte. Trendlinie berechnet aus den Punkt-Daten.

dem z-score [Jiawei Han et al., 2012]:

$$v_i' = \frac{v_i - \overline{A}}{\sigma_A} \tag{5.2}$$

Hierbei werden die Werte v_i aus der Menge A werden auf v_i' abgebildet. Dabei wird der Durchschnitt und die Standardabweichung der Menge A benötigt. Die Menge mit den Laufzeiten wurde neu abgebildet $max(A) - v_i$, sodass ein hoher Wert gut ist. Anschließend wurden aus den Werten ein Score berechnet, bei dem Laufzeit t und Packungsdichte p gleich viel Wert haben: $0.5 \cdot t_i' + 0.5 \cdot p_i'$. An der Stelle wo dieser Score maximal ist, befindet sich der Sweet Spot. Analog könnte man noch weitere Kriterien hinzuziehen oder auch anders Gewichten.

5.5.1. Lokale Optimierung

Der CavityAlgorithm führt eine lokale Optimierung durch. Es wird versucht nur die Lücken in einer bereits vorhandenen Packung zu schließen. Die Ergebnisse für 6 verschiedene Beispiele sind in der Abbildung 5.12. Dabei wurden in jedem Problemszenario analysiert, wie sich die Kriterien: Dichte, Oberflächendichte, Hopkins

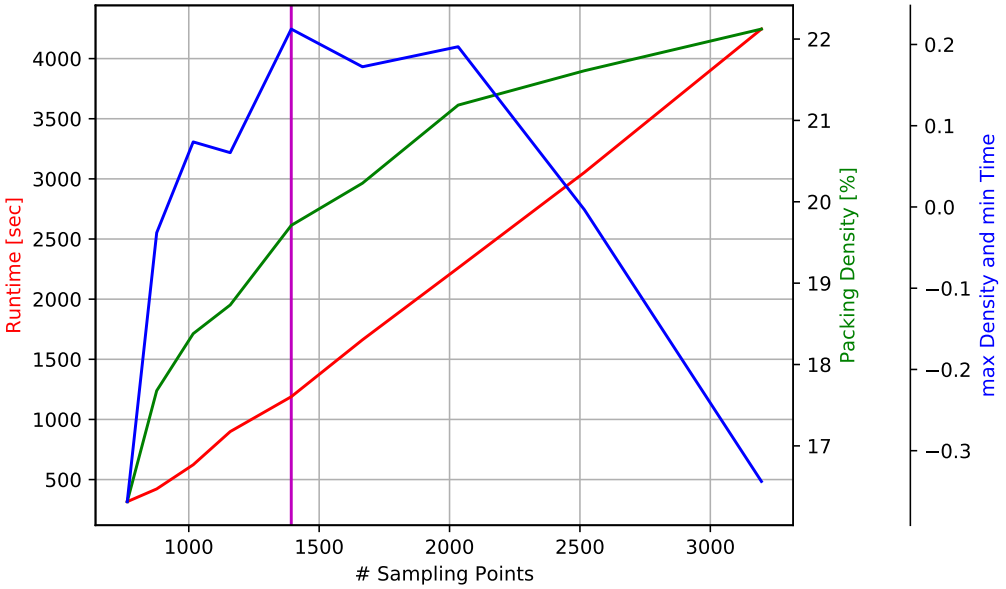


Abbildung 5.11.: Sweet Spot (vertikale Linie) für Dichte und Laufzeit am Beispiel Ram mit GrowingSeedsAlgorithm.

Value und prozentuale Verteilung verändert haben nach der Verbesserung. Um die Veränderung zu messen wurde die Differenzen für die einzelnen Kriterien k_i berechnet: $k_i^{new} - k_i^{old}$. Für Dichte und Oberflächendichte ist eine positive Differenz gut bzw. eine Verbesserung. Dagegen stellt eine positive Differenz bei Hopkins und der prozentualen Abweichung eine Verschlechterung dar.

In allen Beispielen konnte die Dichte noch weiter verbessert werden. Nur in dem Chess-Beispiel ist die Oberflächendichte gesunken, obwohl die Packungsdichte gestiegen ist. Der Hopkins-Wert hat sich nur sehr wenig verändert, wobei in 4 Beispielen sich die räumliche Verteilung minimal verbessert hat (negativer Wert). Die prozentuale Verteilung hat sich in allen Beispielen verschlechtert mit Ausnahmen vom Dog-Beispiel (trivial).

Hierbei wurde auch eine prozentualen Verschlechterung erwartet, da der CavityFillingAlgorithm die Dichte stärker bevorzugt als die Verteilung. Auch hätte man erwarten können, dass die räumliche Verteilung sich verschlechtert, weil evtl. sich viele kleine Objekte in den Lücken anhäufen. Dies ist scheinbar nirgendwo der Fall.

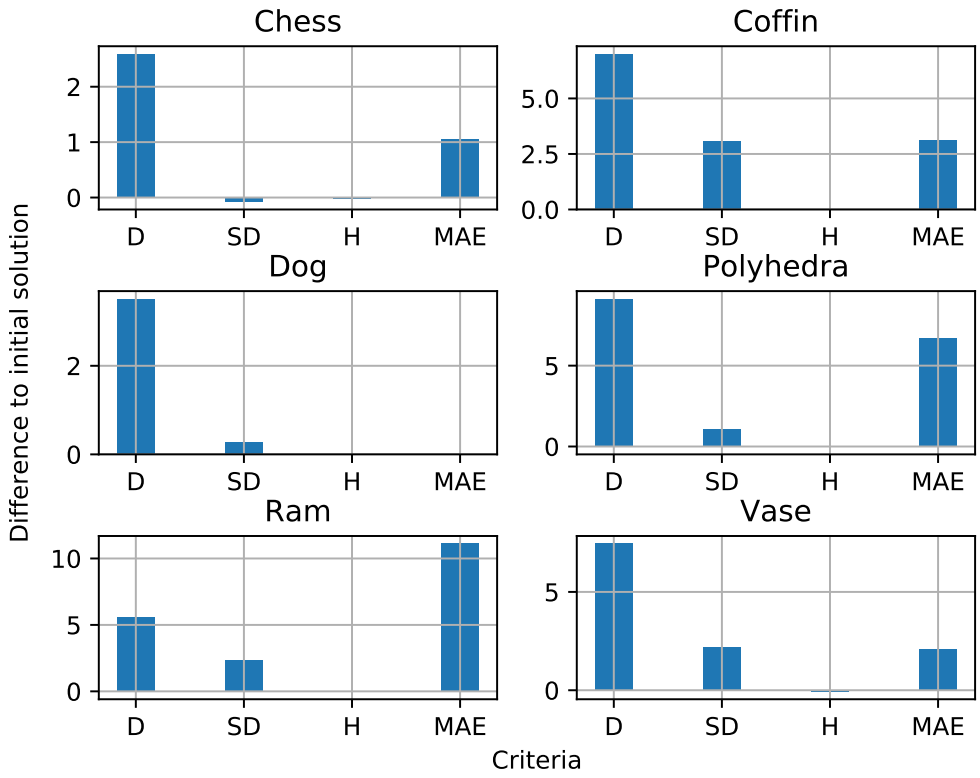


Abbildung 5.12.: Differenzen zur initialen Packung nach lokaler Verbesserung bei den Kriterien: D-Density, SD-Surface Density, H-Hopkins, MAE-Abweichung der Verteilung.

5.5.2. Globale Optimierung

Für eine globale Optimierung wurde der ShuffleAlgorithm entwickelt. Dabei verändern alle Objekte stetig ihre Position oder Rotation und es wird versucht mit den hinzugefügten Objekten eine neue Packungs-Struktur zu erzeugen.

Es wurden die gleichen initialen Packungen wie im vorherigen Kapitel 5.5.1 verwendet und analog eine Verbesserung mit anschließender Auswertung vorgenommen. Ins Auge fällt sofort das Chess-Beispiel. In dem Fall konnte keine Verbesserung erreicht werden und es wurden die initiale Packung ausgegeben. Der Hopkins Wert unterliegt selbst dem Zufall, daher kann es unter Umständen zu kleineren Abweichungen kommen. Alle anderen Hopkins Werte sind fast gleich geblieben.

Für die Dichte und Oberflächendichte konnte in allen anderen Beispielen (außer Chess) eine Verbesserung der Dichte und Oberflächendichte erzielt werden. Im Vergleich zur lokalen Optimierung konnten hier in einigen Fällen ähnlich hohe Packungsdichten und sogar höhere Oberflächendichten erreicht werden. Die prozentuale Verteilung hat sich weniger weniger verschlechtert, als bei der lokalen Verbesserung.

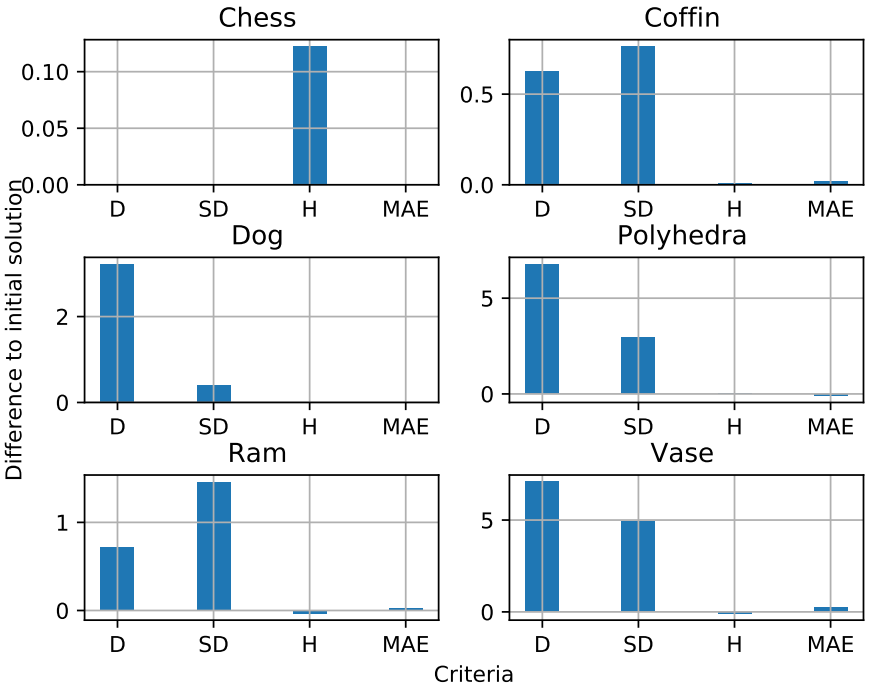


Abbildung 5.13.: Differenzen zur initialen Packung nach globaler Verbesserung bei den Kriterien: D-Density, SD-Surface Density, H-Hopkins, MAE-Abweichung der Verteilung.

5.5.3. Unterschiede durch initiale Packung

In diesem Kapitel wird untersucht, ob die von Konstruktionsheuristiken generierten initialen Packungen einen Einfluss auf die Verbesserungsheuristiken haben. Dazu wurden die Sweet Spot Packungen der unterschiedlichen Algorithmen betrachtet: DistanceAlgorithm, GrowingSeedsAlgorithm, HybridAlgorithm und PlacementAlgorithm. Auf diese Packungen wurde eine lokale und globale Optimierung angewandt. Die Ergebnisse nach lokaler und globaler Optimierung sind in der Abbildung 5.14.

Für den Hopkins-Wert und die Abweichung der prozentualen Verteilung können hier wieder diese gleichen Schlüsse gezogen werden. Also unabhängig von der initial generierten Packung: Hopkins-Wert verändert sich nur minimal und die prozentuale Verteilung wird bei der globalen Optimierung besser eingehalten. Was man auch erkennen kann ist, dass die lokale Optimierung zuverlässiger ist. In einem Fall konnte das Chess-Beispiel (erzeugt von GrowingSeedAlgorithm) nicht optimiert werden und in diesem vorliegenden Fall konnte das Vase-Beispiel (HybridAlgorithm) nicht optimiert werden.

Dennoch sticht hier die globale Optimierung in den Kategorien Dichte und Oberflächendichte hervor. In zwei Fällen (DistanceAlg. PlacementAlg.) konnte eine besseres Ergebnis erzielt werden als von der lokalen Optimierung, was vorherigen Ergebnissen nicht der Fall war.

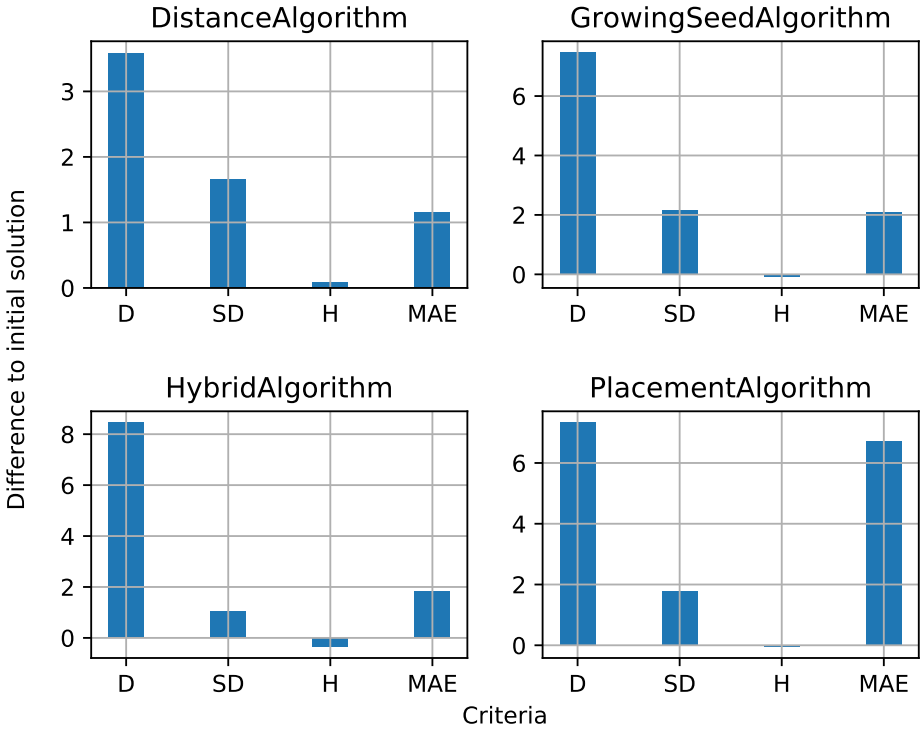
5.5.4. Zusammenfassung

Bei den Konstruktionsheuristiken schneidet der GrowingSeedsAlgorithmen im Vergleich in allen Kategorien sehr gut ab. Dies wäre die Empfehlung, aber alternativ könnte man auch den DistanceAlgorithm oder HybridAlgorithm in Betracht ziehen. Der PlacementAlgorithm wäre überhaupt nicht zu empfehlen.

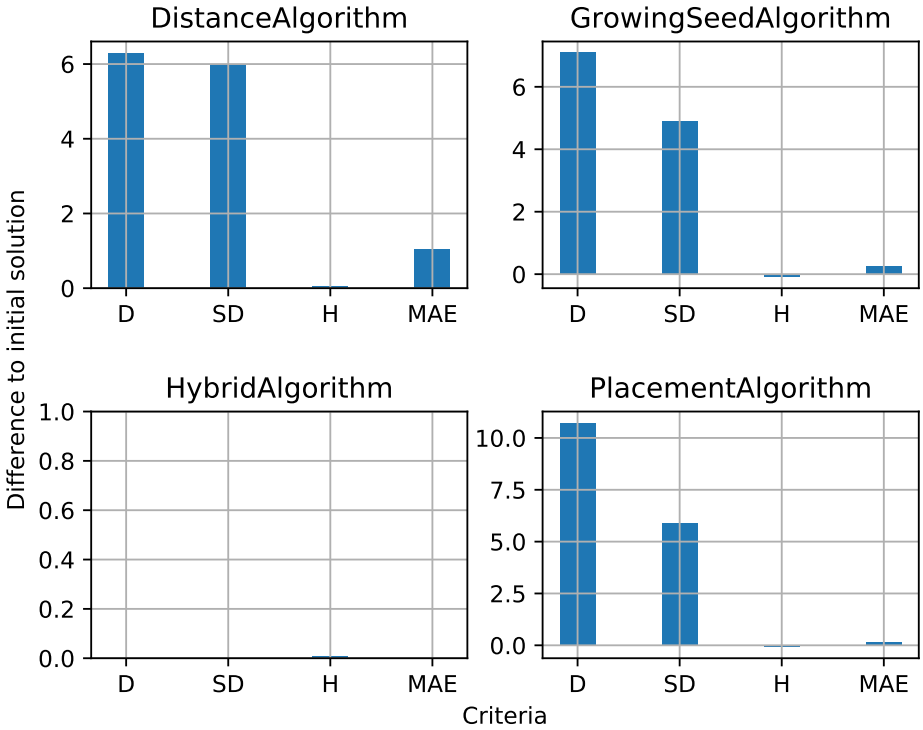
Bei den Verbesserungsheuristiken ist die lokale Optimierung zu empfehlen, wenn Wert auf Zuverlässigkeit gelegt wird und die prozentuale Abweichung weniger wichtig ist. Mit relativ wenigen Objekten (<100) hat die globale Optimierung das potenziell eine bessere Lösung zu finden oder die fast gleiche Dichte mit besserer prozentualen Verteilung.

Eine Empfehlung für eine bestimmte Kombination gestaltet sich schwierig. Man konnte anhand der letzten Ergebnisse erkennen, dass die initiale Packung einen Einfluss auf die globale Optimierung hat. Hierfür müsste noch viel mehr Daten

aus verschiedenen Kombinationen erzeugen und auswerten. Daher ist nach bestem Wissen ist die Kombination aus GrowingSeedsAlgorithm und CavityFilligAlgorithm zu empfehlen.



(a) Lokale Optimierung (CavityFillingAlgorithm)



(b) Globale Optimierung (ShuffleAlgorithm)

Abbildung 5.14.: Differenzen zu initialen Packungen nach globaler und lokaler Verbesserung für das Beispiel Vase. Kriterien: D-Density, SD-Surface Density, H-Hopkins, MAE-Abweichung der Verteilung.

6. Fazit

Das Ziel dieser Arbeit war es, beliebige 3D Objekte in einem beliebigen Container zu platzieren. Dabei sollten sich die Objekte untereinander nicht Überlappen und die Objekte sollten sich innerhalb des Containers befinden. Dazu gehörten noch weitere Ziele wie hohe Packungsdichte, gleichmäßige räumliche Verteilung und eine prozentuale Verteilung von Objekttypen.

Um dieses Problem zu lösen wurden insgesamt 6 verschiedene Algorithmen in dieser Arbeit entwickelt. Wobei eine Unterteilung unternommen worden ist, zwischen Algorithmen, welche eine initiale Packung erzeugen (Konstruktionsheuristik) und Algorithmen, die als Eingabe eine Packung erhalten und diese noch weiter verbessern (Verbesserungsheuristik). Zu den initialen Algorithmen gehören: PlacementAlgorithm, DistanceAlgorithm. HybridAlgorithm und der GrowingSeedsAlgorithm.

Beim PlacementAlgorithm ist die Idee, die Objekte einfach direkt zu platzieren und Überlappungen aufzulösen. Ähnlich ist es beim GrowingSeedsAlgoritm, hier entstehen jedoch die Überlappungen nur langsam, da die Objekte von klein bis zur originalen Größe skaliert werden. Der HybridAlgorithm verfolgt die Idee, die Objekte innerhalb des Container an bestimmten Punkten einfach fallen zu lassen. Dafür wurde eine Rigid-Body-Simulation implementiert, wobei vieles davon auch verwendet wird, um Überlappungen allgemein aufzulösen. Der DistanceAlgorithm lässt keine Überlappungen zu. Dort wird versucht ein Objekt an ein anderes Objekt anzuheften mit Hilfe einer Distanzfunktion für den minimalen Abstand zwischen zwei Objekten.

Bei den Verbesserungsheuristiken wurden zwei unterschiedliche Strategien verfolgt und implementiert. Bei der ersten Strategie handelt es sich um eine lokale Optimierung. Es wird versucht vorhandene Hohlräume in einer Packung zu schließen. Hingegen wird bei der globalen Optimierung versucht die aktuelle Packung neu zu Strukturieren und somit evtl. mehr Objekte einzufügen als es eine lokale Optimierung könnte.

Zur Umsetzung der Algorithmen wurde eine Architektur entworfen, die es ermög-

licht die Algorithmen dynamisch auszuführen bzw. die in den Konfigurationen spezifizierten Algorithmen auszuführen. Die Architektur erlaubt es relativ einfach neue Algorithmen hinzuzufügen. In einer Konfiguration wird Container, Objekte und die Verteilung spezifiziert. Das Ergebnis kann in Dateien geschrieben und auch wieder geladen werden. Auch wurden einfaches Rendering in OpenGL geschrieben, dass es zumindest ermöglicht viele hoch aufgelöste Modelle relativ flüssig zu rendern.

Am Ende dieser Arbeit wurden die Algorithmen evaluiert. Hierzu wurden 6 verschiedene Beispiele aus Objekten und Containern betrachtet. Mit dabei war ein Beispiel bei dem die Modelldaten aus einem 3D-Scanner stammen und eine hohe Auflösung beinhalten. Die Modelldaten wurden vom Künstler Peter Coffin zur Verfügung gestellt, der gerne eine Hand-Statue aus Früchten, in der echten Welt als Kunstobjekt aufstellen würde. Alle Beispiele wurden auf verschiedene Kriterien untersucht und es wurde versucht am Ende eine Empfehlung zur Wahl der Algorithmen zu geben. Als eine gute und zuverlässige Kombination lässt sich GrowingSeedsAlgorithm und die lokale Optimierung empfehlen.

Da die Kollision mit dem Container am ungenausten ist, könnte man sich überlegen es als Sonderfall zu betrachten und sich dazu eine besondere Lösung überlegen (ähnlich wie die Idee in der Evaluation mit einem BSP-Tree).

In einem weiteren Schritt könnte man sich Gedanken machen das Sampling adaptiv durchzuführen. Da in bestimmten Bereiche eine kleinere Schrittweite vom Vorteil wäre und wiederum in anderen Bereichen eine etwas größere. So wäre man etwas weniger Abhängig vom Sampling-Auflösung. Oder man könnte sich grundsätzlich die Frage stellen, ob ein Sampling in der vorhandenen Form noch überhaupt noch notwendig ist. Da im Laufe der Arbeit festgestellt worden ist, dass es kaum Laufzeit kostet. Entsprechend würden sich auch andere und vll. sogar bessere Ideen für Algorithmen ergeben.

Für die globale Optimierung könnte man eine populäre Meta-Heuristik wie Simulated annealing (SA) zu verwenden. Jedoch müsste man sich dort noch genauer Gedanken machen wie eine gute und schnelle Nachbarlösung generiert werden kann (statt einer komplett neuen Lösung). Das war letztlich der Grund, weshalb SA in einer bestimmten Phase dieser Arbeit nicht ausprobiert worden ist bzw. die Laufzeit dafür zu intensiv schien.

Es wäre auch interessant einige erzeugte Packungen mit besserem Rendering zu betrachten. Wo also ein echter Schattenwurf da ist. Dann würden die erzeugten Packungen vermutlich noch viel besser aussehen.

Literaturverzeichnis

[Burke et al., 2007] Burke, E., Hellier, R., Kendall, G., and Whitwell, G. (2007). Complete and robust no-fit polygon generation for the irregular stock cutting problem. *European Journal of Operational Research*, 179(1):27 – 49.

[Dyckhoff, 1990] Dyckhoff, H. (1990). A typology of cutting and packing problems. *European Journal of Operational Research*, 44(2):145 – 159. Cutting and Packing.

[Egeblad, 2008] Egeblad, J. (2008). Heuristics for multidimensional packing problems. *KøbenhavnsUniversitetKøbenhavns Universitet, Det NaturvidenskabeligeFakultetFaculty of Science, Datalogisk InstitutDepartment of Computer Science*.

[Egeblad et al., 2009] Egeblad, J., Nielsen, B. K., and Brazil, M. (2009). Translational packing of arbitrary polytopes. *Computational Geometry*, 42(4):269 – 288.

[Gan et al., 2004] Gan, M., Gopinathan, N., Jia, X., and Williams, R. A. (2004). Predicting packing characteristics of particles of arbitrary shapes. *KONA Powder and Particle Journal*, 22:82–93.

[Gomes and Oliveira, 2002] Gomes, A. M. and Oliveira, J. F. (2002). A 2-exchange heuristic for nesting problems. *European Journal of Operational Research*, 141(2):359–370.

[Jia and Williams, 2001] Jia, X. and Williams, R. (2001). A packing algorithm for particles of arbitrary shapes. *Powder technology*, 120(3):175–186.

[Jiawei Han et al., 2012] Jiawei Han, Micheline Kamber, and Jian Pei (c 20122012). *Data mining : concepts and techniques*. The Morgan Kaufmann series in data management systems. Elsevier/Morgan Kaufmann, Amsterdam [u.a.], 3. ed. edition. Online-Ressource (XXXV, 703 S.) : Ill.

[Li et al., 2010] Li, S., Zhao, J., Lu, P., and Xie, Y. (2010). Maximum packing densities of basic 3d objects. *Chinese Science Bulletin*, 55(2):114–119.

[Lutters et al., 2012] Lutters, E., ten Dam, D., and Faneker, T. (2012). 3d nesting of complex shapes. *Procedia CIRP*, 3:26 – 31. 45th CIRP Conference on Manufacturing Systems 2012.

[Ma et al., 2018] Ma, Y., Chen, Z., Hu, W., and Wang, W. (2018). Packing irregular objects in 3d space via hybrid optimization. In *Computer Graphics Forum*, volume 37, pages 49–59. Wiley Online Library.

[Prasad, 1997] Prasad, L. (1997). Morphological analysis of shapes. *CNLS newsletter*, 139(1):1997–07.

[Romanova et al., 2018] Romanova, T., Bennell, J., Stoyan, Y., and Pankratov, A. (2018). Packing of concave polyhedra with continuous rotations using nonlinear optimisation. *European Journal of Operational Research*, 268(1):37 – 53.

[Rosato et al., 1987] Rosato, A., Strandburg, K. J., Prinz, F., and Swendsen, R. H. (1987). Why the brazil nuts are on top: Size segregation of particulate matter by shaking. *Physical Review Letters*, 58(10):1038.

[Si, 2015] Si, H. (2015). Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Transactions on Mathematical Software (TOMS)*, 41(2):11.

[Si and Gärtner, 2005] Si, H. and Gärtner, K. (2005). Meshing piecewise linear complexes by constrained delaunay tetrahedralizations. In *Proceedings of the 14th international meshing roundtable*, pages 147–163. Springer.

[Weller, 2013] Weller, R. (2013). *Inner Sphere Trees*, pages 113–144. Springer International Publishing, Heidelberg.

[Whitwell, 2004] Whitwell, G. (2004). *Novel heuristic and metaheuristic approaches to cutting and packing*. PhD thesis, University of Nottingham.

[Wäscher et al., 2007] Wäscher, G., Haußner, H., and Schumann, H. (2007). An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109 – 1130.

Abbildungsverzeichnis

Tabellenverzeichnis

A. Anhang

A.1. Quellcode

Der Quellcode ist in der Programmiersprache C++ geschrieben. Abhängigkeiten bestehen nur zu den Bibliotheken CollDet und OpenGL. Das Projekt liegt als cmake-Projekt vor, woraus dann Makefile- und andere Projekt-Dateien erzeugt werden können. Im *lib* Ordner müssen die kompilierten CollDet-Bibliotheken abgelegt werden und die Header-Dateien im *include* Ordner. Da Erweiterungen an CollDet vorgenommen wurden, wird eine angepasste Version mitgeliefert.

A.2. Parameter zum Konfigurieren

Zum Aufruf von *AutoPacking* muss eine Konfigurationsdatei angegeben werden. Die Parameter zum Starten sehen wie folgt aus:

```
AutoPacking [-c path config.cfg default] [-v packingFile]
```

Mit dem Parameter *-v* kann eine Packung nur zum Betrachten geladen werden.

In der Konfigurationsdatei existieren Kategorien in eckigen Klammern die allein in einer Zeile stehen. Ansonsten symbolisieren eckige Klammern einen optionalen Parameter. In spitzen Klammen in ein Pflicht-Parameter. Konstruktionsheuristiken zur Wahl: *PlacementAlgorithm, DistanceAlgorithm, HybridAlgorithm, GrowingSeedAlgorithm.* Verbesserungheuristik zur Wahl: *CavityFillingAlgorithm, ShuffleAlgorithm.*+

Unter Preprocessing kann das Volumen der Kugel für die Kugel beim Sampling festgelegt werden.

```
[Name]
[Optional_Name] // Used for evaluation
[Construction Heuristic]
```

```
<Algorithm> [write_or_load_packing_file.txt]
[Improvement Heuristic]
<Algorithm> [write_improved_packing_file.txt]
[Grid]
<i i i>                // Gridsize int e.g. 2 2 2
[SurfaceDensity Epsilon]
<f>                    // Epsilon real e.g. 0.05
[Hopkins Statistic]
<f> <i>                // 0-1 use percentage/subset of objects, int repeat
[Preprocessing]
<write_load_file> (smallest|average|(real)) <StepSize> <NumberOfThreads>
[ModelPath]
<PrefixPath>           // e.g. models/example1/
[Container]
<ModelName> (<f> <f> <f>|rand) <f> // 0-1 RGB or rand + Alpha
[Objects]
<ModelName> <f> (<f> <f> <f>|rand) <f> // 0-100 Distribution, like above
more objects ..!
```

A.3. Bilder

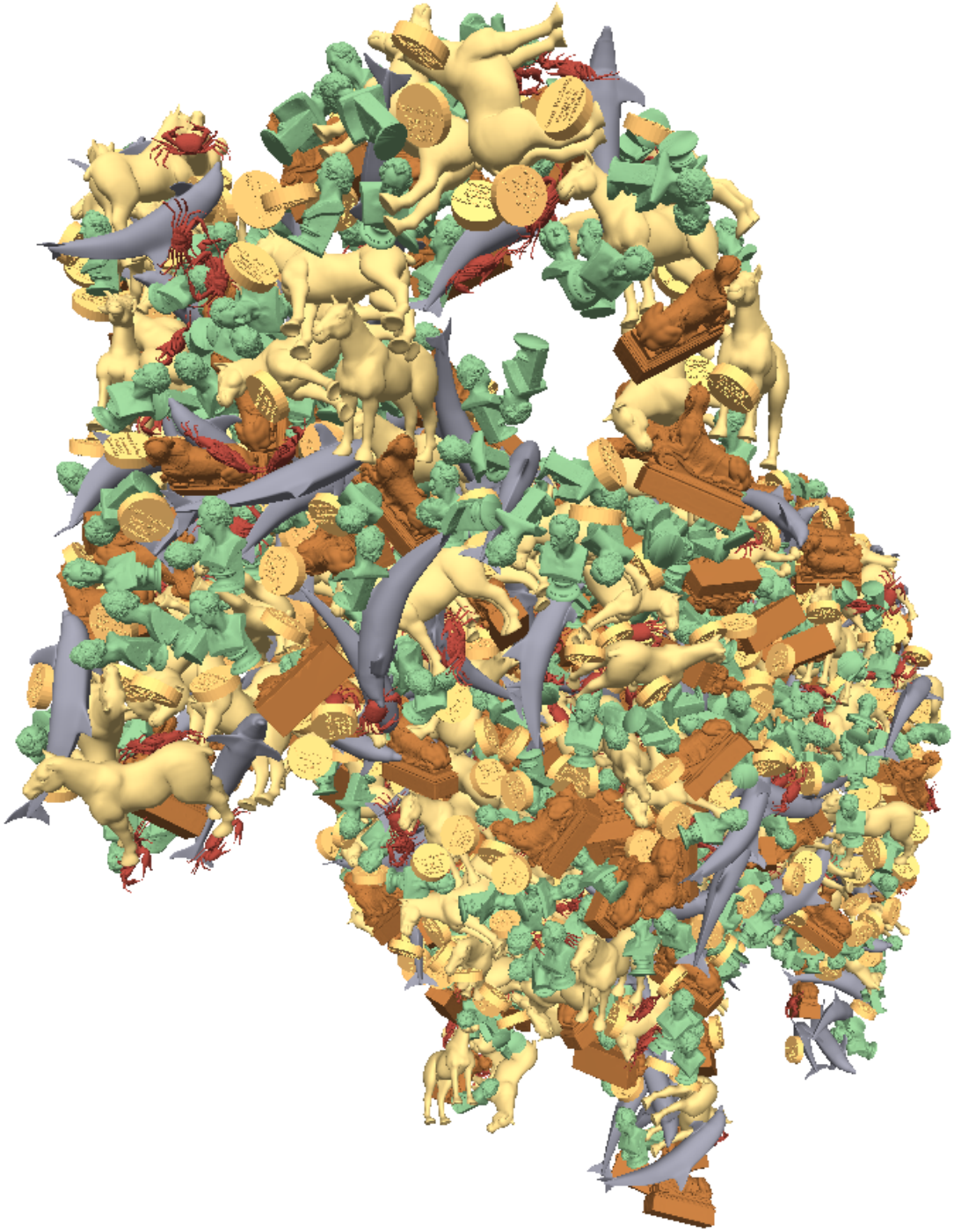


Abbildung A.1.: Ziegenbock ohne Container

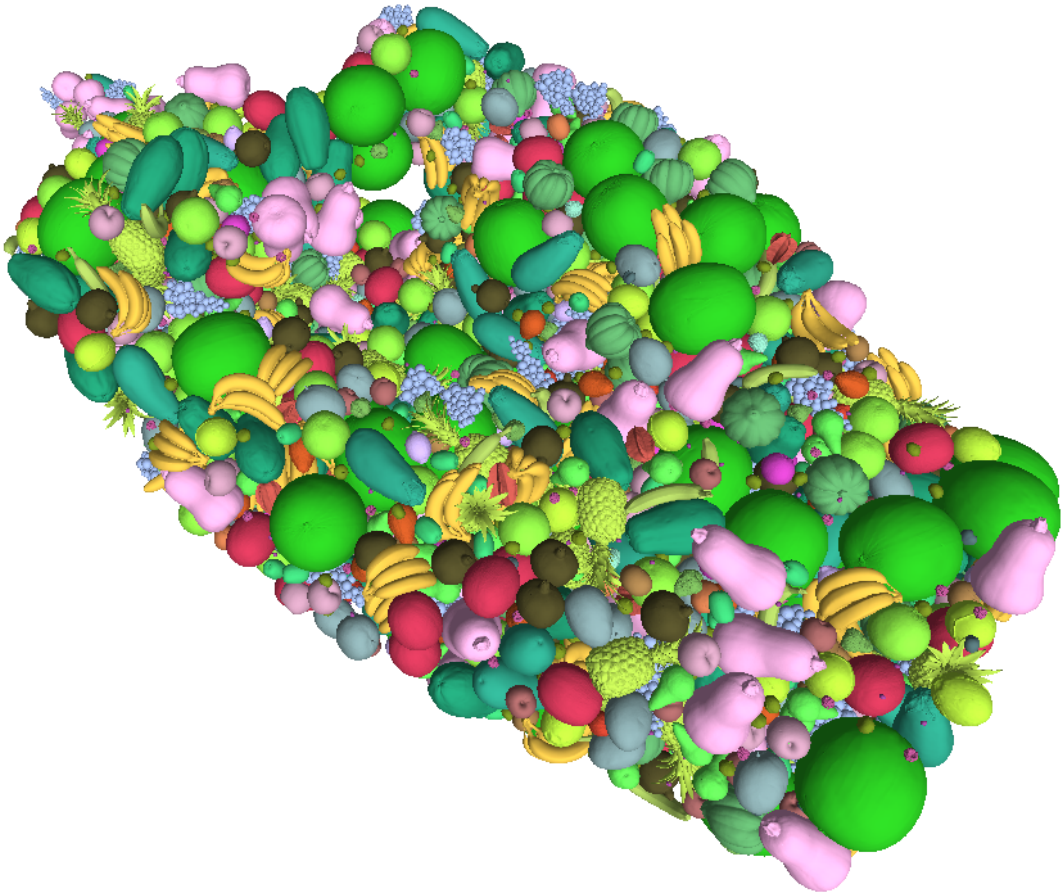


Abbildung A.2.: Hand ohne Container mit zufälligen Farben (Peter Coffin)