Universität Bremen
Fachbereich 3 - Informatik/Mathematik

Master-Thesis

3D Container Packing

Optimierungen für AutoPacking

Derk Sönke Akkermann
Matrikelnummer: 4013914
vorgelegt am: 14.12.2021
1. Gutachter: Prof. Dr. Gabriel Zachmann
2. Gutachter: Dr. René Weller
Betreuer: M.Sc. Hermann Meißenhelter

Eidesstattliche Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe und dass ich alle Stellen, die ich wörtlich oder sinngemäß aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe. Die Arbeit hat bisher in gleicher oder ähnlicher Form oder auszugsweise noch keiner Prüfungsbehörde vorgelegen.

Bremen, den 14.12.2021

Derk Sönke Akkermann

Danksagung

Ich danke Prof. Gabriel Zachmann dafür, dass er mir das Thema überlassen hat und bei Fragen immer ansprechbar war.
Ich danke Dr. René Weller dafür, dass er sich bereit erklärt hat, diese Arbeit als Zweitgutachter zu bewerten.
Besonderer Dank gilt Hermann Meißenhelter dafür, dass er mich während der Bearbeitungszeit betreut hat und mir für Fragen und Anregungen immer zu Verfügung stand.
Ich danke meinen Eltern für die Hilfe während der Bearbeitung und für das Korrekturlesen der Arbeit.

Zusammenfassung

Die vorliegende Masterarbeit befasst sich mit Packungsproblemen, speziell dem Fall, 3D Objekte in einen 3D Container zu packen. Ziel war es dabei, das Programm *AutoPacking* im Hinblick auf die Laufzeit und die Packungsqualität zu verbessern.
Dazu wurde *AutoPacking* so angepasst, so dass verschiedene Arten der Containerrepräsentation unterstützt werden. Neben der ursprünglichen Repräsentation über eine Kugelpackung für die Kollisionserkennung lässt sich der Container nun mittels beliebig vieler Kugelpackungen oder direkt über das Mesh darstellen.
Außerdem wurde die für die Kollisionserkennung verwendete Bibliothek *CollDet* dahingehend geändert, dass die Möglichkeit besteht, die Kollisionserkennung zwischen Objekten und Container auf der GPU zu berechnen.
Die verschiedenen Ansätze wurden evaluiert und verglichen. Es zeigt sich, dass sowohl die Aufteilung des Containers als auch die Kollisionserkennung auf der GPU Laufzeitvorteile bringen. Die Darstellung des Containers über das Mesh oder über mehrere Kugelpackungen verbessern zudem die Packungsqualität.
Mittels einer im Rahmen dieser Arbeit neu entwickelten Verbesserungsheuristik konnte die Oberflächendichte der Packungen in den allermeisten Konfigurationen deutlich gesteigert werden, was für manche Anwendungsfälle von besonderem Interesse ist.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

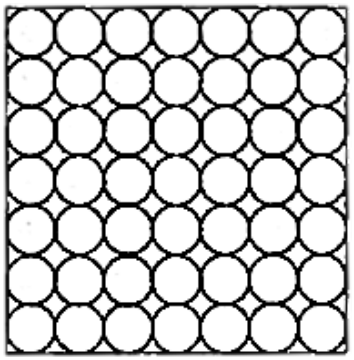
Einleitung

1.1 Motivation

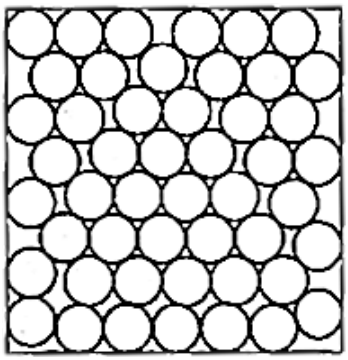
Diese Arbeit basiert auf der Masterarbeit „Auto Packing für beliebige 3D Objekte und Container“ von Hermann Meißenhelter. Ziel war es, beliebige 3D Objekte in Container packen zu können. Die Idee entspringt dem Wunsch Peter Coffins[1], eines amerikanischen Künstlers, eine Statue einer Hand aus Bronze zu fertigen. Die Form der Hand sollte dabei nur durch den Inhalt repräsentiert werden, der in Anlehnung an Bilder von Füllhörner durch Früchte dargestellt werden sollte.
Packungsprobleme finden sich jedoch in vielen weiteren Anwendungen: in den letzten Jahren vor allem im 3D-Druck, aber auch in der Logistik. Dabei ist die Zielsetzung immer eine Unterschiedliche. Geht es bei dem 3D-Druck darum, dass die gedruckten Teile eine gewisse Qualität aufweisen, so ist es in der Logistik wichtig, dass z.B. Container so in Schiffe geladen werden, dass diese zum einen stabil stehen, zum anderen Container, die früher entladen werden, oben platziert werden müssen. Aber auch andere Probleme können als Packungsprobleme aufgefasst werden. So zum Beispiel die Verteilung von Rohstoffen auf verschiedene Werksstandorte eines Industrieunternehmens.
Dabei ist es nicht einfach, eine optimale Packung zu finden. Dies beginnt schon bei der Betrachtung des Problemes, Kreise in ein möglichst kleines Rechteck zu packen, wie Ian Stewart in seinem Buch „Neue Wunder aus der Welt der Mathematik“ darstellt. Für den Menschen nicht erkennbar ist das Quadrat in Abbildung 1.1b minimal kleiner als das in Abbildung 1.1a. Lange war angenommen worden, dass für alle Quadratzahlen n <64 gilt, und somit diese Anordnung die optimale Lösung sei [Nurmela and Östergård, 1997]. Jedoch konnten die Autoren durch die gefundene Packung aus Abbildung 1.1b zeigen, dass dies nur für Quadratzahlen bis 36 gilt.

Mit zunehmenden Dimensionen wird der Suchraum für Lösungen deutlich größer. Hinzu kommt, dass die Zielfunktionen zum Teil sehr komplex sein können, wodurch das Finden einer geeigneten Lösung erschwert wird. Daher ist vor allem die Geschwindigkeit bei der Generierung von möglichen Lösungen sowie deren Überprüfung ein wichtiges Kriterium

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



(a) Quadratische Anordnung



(b) Aktuell optimale Anordnung

Abbildung 1.1: Vergleich der offensichtlichen Packung von 49 Kreisen in ein Quadrat, mit der optimalen Lösung. Entnommen aus [Stewart, 2009].

für Algorithmen, die Packungsprobleme lösen sollen. Je mehr mögliche Lösungen innerhalb eines Zeitintervalls generiert und geprüft werden können, desto höher ist die Wahrscheinlichkeit, dass eine gute Lösung gefunden wird. Allerdings ist es schwierig, optimale Lösungen zu finden.

1.2 Ziele

Das Ziel dieser Arbeit ist es, dass vorliegende Programm *AutoPacking* in den folgenden Bereichen zu verbessern:

- Laufzeit
- Herausragen der Objekte aus dem Container
- Oberflächendichte der erzeugten Packungen

Das Programm benötigt zum Packen der Objekte Zeit. Meißenhelter stellte in seiner Arbeit eine Abhängigkeit von der Anzahl der zu packenden Objekte fest. Ein Ziel dieser Arbeit ist es nun, die Laufzeit bei gleichbleibender Anzahl der Objekte zu beschleunigen, so dass dieselbe Anzahl an Objekten in kürzerer Zeit gepackt werden kann. Dabei sollen die ursprünglichen Ziele, vor allem die Packungsdichte, nicht verschlechtert werden.
Außerdem fällt auf, dass in den erzeugten Packungen die Objekte zum Teil sehr stark aus dem Container ragen. Daher ist ein weiteres Ziel, dies zu verbessern bzw. ganz zu verhindern. Auch hier soll wieder darauf geachtet werden, dass damit keine Verschlechterung der ursprünglichen Ziele, vor allem der Packungsdichte, einhergeht.
Der letzte Verbesserungsbereich ist spezifisch für Peter Coffin, da es bei der Darstellung der Hand durch Früchte darauf ankommt, dass der Containerrand möglichst gut angenähert wird. Dazu müssen mehr Objekte am Rand des Containers gepackt werden, wohingegen das Innere des Containers weniger relevant ist.

1.3 Überblick

Im Folgenden wird kurz der Aufbau der Arbeit umrissen.
In Kapitel 2 werden die theoretischen Grundlagen zu Packungsproblemen erläutert. Dazu wird definiert, was Packungsprobleme sind und wie sich diese klassifizieren lassen. Darüber hinaus werden Stärken und Schwächen der einzelnen Klassifzierungsmöglichkeiten benannt. Des Weiteren werden verschiedene bei der Lösung von Packungsproblemen zu beachtende Probleme aufgezeigt und jeweils mögliche Lösungsansätze dargestellt. Dabei wird der Umgang mit diesen Problemen anhand verschiedener bereits existierender Arbeiten zu Packungsproblemen beschrieben.
Kapitel 3 beschreibt die verschiedenen im Rahmen dieser Arbeit entwickelten Lösungsansätze für die Ziele aus Abschnitt 1.2.
Das Kapitel 4 umfasst die Evaluation der entwickelten Ansätze und vergleicht diese mit der ursprünglichen Arbeit.
Kapitel 5 enthält schlussendlich das Fazit sowie einen Ausblick auf Fragestellungen, die sich aus den Ergebnissen dieser Arbeit ergeben haben.

Kapitel 2

Grundlagen

Packungsprobleme werden seit langem erforscht [Dowsland and Dowsland, 1992]. Im folgenden Kapitel wird erläutert, was Packungsprobleme sind und wie sich diese klassifizieren lassen. Anschließend wird auf verschiedene Kernprobleme, die bei der Entwicklung von Algorithmen zur Lösung von Packungsproblemen zu beachten sind, eingegangen. Dabei werden verwandte Arbeiten auf ihren Umgang mit diesen Problemen hin betrachtet.

2.1 Packungsprobleme

Nach [Wäscher et al., 2007, S. 1110] lassen sich Packungsprobleme wie folgt definieren:

> ,,Given two sets of elements, namely
>
> - a set of large objects (input, supply) and
> - a set of small items (output, demand),
>
> which are defined exhaustively in one, two, three or an even larger number (n) of geometric dimensions. Select some or all small items, group them into one or more subsets and assign each of the resulting subsets to one of the large objects such that the geometric condition holds, i.e. the small items of each subset have to be laid out on the corresponding large object such that
>
> - all small items of the subset lie entirely within the large object and
> - the small items do not overlap,
>
> and a given (single-dimensional or multi-dimensional) objective function is optimised. We note that a solution of the problem may result in using some or all large objects, and some or all small items, respectively. “

2.1.1 Optimierungsprobleme

Aus der Definition geht hervor, dass Packungsprobleme Optimierungsprobleme sind. Diese Probleme werden beschrieben durch eine Menge von möglichen Lösungen L und einer Zielfunktion $f(l)$, die jedem Elemen $l \in L$ einen Wert zuweist [Crama et al., 1995]. Der Wert,

den die Funktion einer Lösung zuweist, kann z.B. die Menge an Objekten im Container oder das eingenommene Volumen des Containers durch die Objekte sein.
Man unterscheidet zwischen Minimierungs- und Maximierungsproblemen. Bei Minimierungsproblemen soll eine Instanz $l \in L$ gefunden werden, so dass gilt $f(l) < f(l')\forall l' \in L$. Für Maximierungsprobleme gilt entsprechend $f(l) > f(l')\forall l' \in L$.

2.1.2 Komplexität

Mit Komplextität wird der Ressourcenverbrauch zur Lösung eines Problems beschrieben. Als Ressourcen werden Rechenzeit und Speicherverbrauch betrachtet [Lutz, 2021]. Komplexität wird in der Theroretischen Informatik jedoch nur für Entscheidungsprobleme definiert. Aus Entscheidungsproblemen lassen sich Optimierungsprobleme bilden, indem dem Entscheidungsproblem noch eine Schranke für ein Optimierungskriterium hinzugefügt wird. Dieses Entscheidungsproblem kann nun nicht komplexer sein als das Optimierungsproblem, wenn die Berechnung der Zielfunktion effizient ist [Garey and Johnson, 1990].
Bezogen auf Containerpackungen könnte eine mögliche Formulierung für ein Entscheidungsproblem wie folgt lauten (in Anlehnung an [Meißenhelter, 2019] und [Egeblad et al., 2009]):
Gegeben sei eine Menge von Objekten und ein Container. Gibt es eine Packung von Objekten in dem Container, ohne dass sich die Objekte untereinander überlappen oder aus dem Container herausragen?
Entscheidungsprobleme werden den Komplexitätsklassen zugeordnet. Die Klassen bilden sich dabei nach maximal benötigter Rechenzeit oder maximal benötigtem Speicherplatzverbrauch des bestmöglichen Algorithmus zur Lösung des Problems. Wichtig sind dabei die beiden Klassen *P* und *NP*. Effizient lösbare Probleme liegen in *P*, nicht effizient lösbare Probleme liegen in *NP*.
Packungsprobleme liegen in der Regel in *NP*, Einschränkungen bei 1-dimensonalen Problemtypen können jedoch dafür sorgen, dass diese mit polynomiell beschränktem Ressourcenverbrauch gelöst werden können [Garey and Johnson, 1990]. Für höhere Dimensionen gilt dies nicht mehr [Fowler et al., 1981].

2.2 Typologie

Packungsprobleme sind sehr vielfältig, wie die Definition zeigt. Zur genaueren Klassifikation von Packungsproblemen wurden daher verschiedene Typologien entwickelt. Die Kerncharakteristiken von Packungsproblemen sind dabei die folgenden:

- Dimensionality
- Kind of Assignment
- Assortment of large objects

- Assortment of small items

Diese Charakteristiken wurden erstmals in der Typologie in [Dyckhoff, 1990] verwendet, um Packungsprobleme zu klassifizieren. Dabei beschreibt *Dimensionality* die räumlichen Dimensionen der Objekte und des Containers. *Kind of assignment* beschreibt das Ziel des Problems. Bei Dyckhoff ist dies entweder alle vorhandenen Container zu füllen, dabei müssen aber nicht alle Objekte verwendet werden oder alle Objekte in die vorhandenen Container zu packen, wobei auch leere Container übrig bleiben dürfen. *Assortment of large objects* und *Assortment of small items* beschreibt dann die Beschaffenheit der Container und Objekte. Er gewichtet dabei *Dimensionality* am stärksten. Allerdings gibt es in seiner Typologie das Problem, dass sie nicht eindeutig ist [Dyckhoff, 1990], [Wäscher et al., 2007]. Außerdem werden nach [Wäscher et al., 2007] Probleme, die in der Praxis unterschieden werden, derselben Klasse zugeordnet. Die Autoren schlagen daher eine neue, dreistufige Typologie vor. Die erste Stufe beschreibt dabei die sogenannten *Basic Problem Types*, wobei *kind of assignment* das wichtigste Kriterium ist. Sie unterscheiden zwischen *output maximisation*, also möglichst viele Objekte zu packen und *input minimisation*, also möglichst wenig Container zu verwenden. Das zweite Kriterium zur Bestimmung der *basic problem types* ist das *assortment of small items*. Hierbei unterscheiden die Autoren *identical*, *weakly heterogenous*, *strongly heterogenous* und *arbitrary*. Die zweite Stufe konkretisiert dies zu sogenannten *intermediate problem types* durch die Hinzunahme des Kriteriums *assortment of large objects*. Dieses Kriterium unterscheidet, ob es ein, mehrere gleiche oder mehrere unterschiedliche Container gibt. Die dritte Stufe bildet dann über die Kriterien *Dimensionality* sogenannte *refinded problem types*. Sollte das Problem nicht eindimensional sein, so wird zusätzlich das Kriterium *shape of the small itmes* herangezogen. Dabei wird zwischen *regular* (einfache geometrische Formen) und *irregular* unterschieden. Araújo et al. schlagen in [Araújo et al., 2019] eine weitere Typologie vor, die speziell das 3-dimensionale Packungsproblem mit verschiedenen Objekten klassifizieren soll. Als Anwendungsfall nennen sie die Klassifizierung der verschiedener Packungsproblemtypen des 3D-Drucks. Als Problem der Typologien von Dyckhoff und Wäscher et al. sehen sie, dass diese bei der Klassifzierung der Objekte stark von der Bedeutung der gewählten Wörter(viel vs.wenig oder stark vs. schwach) abhängen. Außerdem ließen die Typologien wichtige Informationen zu geometrischen Eigenschaften der Objekte außer Acht. Im Gegensatz zu den anderen Typologien unterscheiden sie zwischen *Model* und *Part*. Ein *Model* ist die Beschreibung eines Objektes, während ein *Part* ein tatsächliches Objekt ist. Sie schlagen daher die folgenden Charaktersitiken vor.

- Dimensionality
- Criteria for optimisation
- Build volume types
- Attributes/features of the assortment of parts

Da ihr Anwendungsfall der 3D-Druck ist, geben sie zu bedenken, dass die Probleme meistens dreidimensional sind, es jedoch auch Lösungsansätze gibt, die die Problemdimension verringern. Unter *criteria for optimisation* werden vier Ziele definiert, die ein Problem haben kann. Zu den beiden bereits beschriebenen Zielen kommen hinzu *cost-minimisation parallel production*, die Minimierung der Anzahl der Druckvorgänge und *time-minimisation parallel production*, die Minimierung der Zeit für die einzelnen Druckvorgänge. Beide Ziele sind sehr spezifisch für den beschriebenen Anwendungsfall.
Alle Lösungsansätze haben dabei die beiden folgenden Probleme zu beachten:

- Auswahl eines Containers und bei mehrdimensionalen Problemen eines Platzes im Container
- Auflösung bzw. Vermeidung von Kollisionen bei mehrdimensionalen Problemen

Im folgenden werden verschiedene Lösungsansätze für diese Probleme vorgestellt. Dabei sind die beiden Probleme stark miteinander verbunden.

2.3 Platzieren von Objekten

Ein offensichtliches Problem ist, dass die Objekte im Container bzw. in den Containern platziert werden müssen. Eine Auswahl des Containers ist nur bei Problemen mit mehreren Containern nötig. Bei eindimensionalen Problemen spielt die Anordnung der Objekte innerhalb des Containers keine Rolle. Hier wird eine Summe gebildet, z.B. der Gewichte oder Längen der Objekte, die den Grenzwert des Containers nicht überschreiten darf.

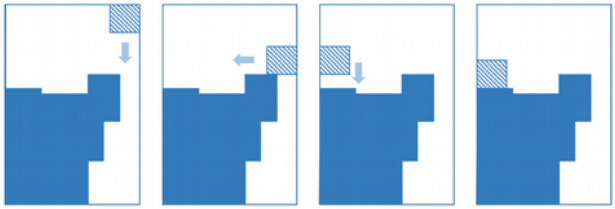
2.3.1 First Fit Decreasing

Dies ist ein Ansatz zur Lösung des eindimensonalen Bin-Packing-Problems. Ziel ist es die Anzahl der benötigten Container für eine gegebene Menge von Objekten zu minimieren. Der Lösungsansatz beruht dabei auf dem *first fit* Prinzip, bei dem ein Objekt in den ersten Container gepackt wird, in dem es Platz hat. Laut [Garey and Johnson, 1990] hat *first fit* die schlechteste Laufzeit, wenn kleine Objekte vor den großen ausgewählt werden. *First fit decreasing* löst dieses Problem, indem die Objekte absteigend anhand ihres Wertes sortiert werden. Dies kann z.B. das Gewicht oder das Volumen sein. Die so enstandene Liste wird dann mittels des *first fit* auf die Container aufgeteilt.

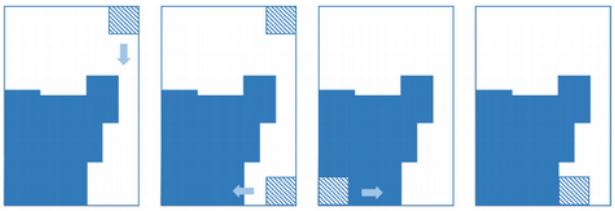
2.3.2 Bottom-Left-Front

Bottom-Left-Front (BLF) ist ein Ansatz zum Platzieren von Objekten in dreideimensionalen Packungsproblemen. Bei diesem Ansatz wird das zu platzierende Objekt von oben, hinten rechts nach unten, vorne links durch den Container geschoben [Wu et al., 2014].

Kommt es zu einer Kollision, so wird zur letzten kollisionsfreien Position zurückgegangen und das Objekt dort platziert. Die Autoren beschreiben, dass dieser Ansatz in lokalen Minima stecken bleiben kann. Daher schlagen sie vor, dass die Bewegungsrichtungen fixiert werden und immer eine Achse optimiert wird. Für den zweidimensionalen Fall wird zuerst die x-Achse fixiert und geprüft, ob auf der y-Achse an der fixierten x-Stelle Platz ist bei $y = 1$. Ist hier Platz, so wird dieser Wert fixiert und die x-Achse wird auf 1 gesetzt. Ist hier kein Platz, so wird das Objekt in Richtung des alten x-Wertes geschoben, bis es passt. Im dreidimensionalen Fall wird zuerst die z-Achse, dann die y-Achse und zuletzt die x-Achse optimiert. Dadurch werden Lücken in der Platzierung geschlossen. Zu sehen sind die beiden Ansätze für zwei Dimensionen in Abbildung 2.1.



(a) Klassisches *Bottom-Left*



(b) Verbessertes *Bottom-Left*

Abbildung 2.1: 2D Darstellung des *Bottom-Left*-Ansatzes. Entnommen aus [Wu et al., 2014].

2.3.3 Platzierung nach Ojektvolumen

Hierbei werden die Objekte zufällig gewählt, die in den Container gepackt werden [Ma et al., 2018]. Der Container wird dann anhand der Volumina der Objekte geteilt, so dass jedes Objekt einen Platz hat, der seinem Volumen entspricht. Die Autoren merken jedoch an, dass hierbei bereits bei der Platzierung Kollisionen zwischen den Objekten entstehen können und, wie auch schon bei *BLF*, lokale Minima Ergebnis der initialen Platzierung sein können.

2.4 Kollisionserkennung

Beim Platzieren der Objekte kann es zu Kollisionen kommen, welche aufgelöst werden müssen. Dabei hängt die Geschwindigkeit der Erkennung von Kollisionen stark von der Art der gewählten Repräsentation der Objekte und des Containers ab [Bennell and Oliveira,

2009]. Im Folgenden werden einige Arten zur Repräsentation der Objekte und Container betrachtet sowie ihre Vor- und Nachteile herausgestellt.

2.4.1 Rasterung

Bei der Rasterung werden die Objekte und der Container in gleichmäßige Zellen geteilt [Eisenbrand et al., 2003], [Bennell and Oliveira, 2008]. Es gibt nun verschiedene Möglichkeiten, wie dieses Raster zum Platzieren von Objekten genutzt werden kann. Zum einen kann man beim Platzieren eines Objektes die Zellen markieren, in denen das Objekt ganz oder teilweise liegt. Je nach Markierungsansatz können Kollisionen durch einfaches Auswerten des Zellwertes festgestellt werden.
In [Eisenbrand et al., 2003] nutzen die Autoren einen Graphen, um Packungen zu codieren. Die Knoten codieren dabei Startposition und Dimension des Objektes innerhalb des Rasters des Containers. Kollisionen zwischen Objekten werden über Nachbarschaften der Knoten im Graphen erkannt.
Vorteile der Darstellung von Objekten und Container durch ein Raster sind, dass sich auch komplexe Objekte einfach darstellen lassen und die Überprüfung auf Kollisionen schnell ist, da nur die Zellen betrachtet werden müssen. Allerdings sind die Rastermethoden in der Darstellung der Objekte ungenau, da die Objekte nicht jede Zelle komplett füllen. Dies kann durch eine Verkleinerung der Rasterzellen verbessert werden, jedoch steigt dann der Speicherverbrauch. Die Komplexität für das Finden von Kollisionen zwischen zwei Polygonen liegt dabei bei $O(n)$ [Egeblad, 2008]. Für das Überprüfen einer kompletten Lösung liegt die Komplexität $O(n^2)$, wobei n die Anzahl der Rasterzellen ist [Bennell and Oliveira, 2008].

2.4.2 Nofit Polygone

Eine weitere Möglichkeit 2D-Objekte zu repräsentieren und auf Kollisionen zu überprüfen stellen die songenannten *nofit polygone (NFP)* dar. Um diese zu konstruieren, werden Paare gebildet. Ein Objekt wird dann im Ursprung fixiert, während für das andere ein Referenzpunkt auf dessen Kante gewählt wird. Dieses Polygon wird dann so um das andere Objekt geschoben, dass die beiden sich berühren, aber nie überlappen. Der Pfad des Referenzpunktes stellt dann das *nofit polygon* dar [Bennell and Oliveira, 2008]. Das geschobene Objekt darf dabei allerdings nicht rotiert werden. Zu sehen ist dies in Abbildung 2.2. Überlappungen zwischen zwei Polygonen enstehen nur, wenn der Referenzpunkt des einen Polygons im *NFP* des Paares liegt. Liegt er auf der Kante des *NFP*, so berühren sich die beiden Polygone. Der Container lässt sich dabei als sogennates *inner-fit polygon* darstellen. Das Verfahren zur Erstellung ist analog zum *NFP*, jedoch wird hier das zu packende Objekt innen durch den Container geschoben, so dass sich die beiden berühren [Bennell and Oliveira, 2009].

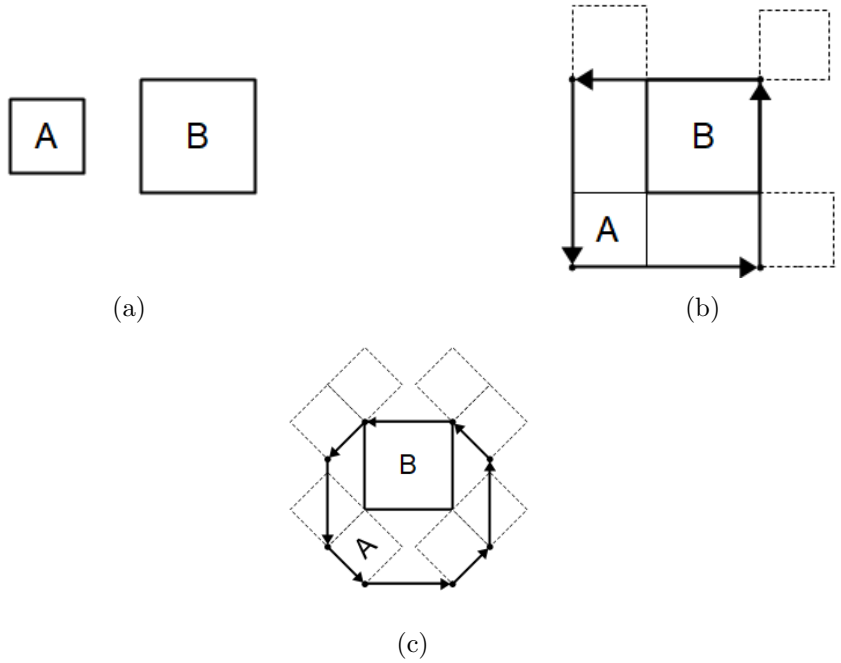


Abbildung 2.2: (a) zeigt die beiden Polygone, für die das *NFP* erstellt werden soll. (b) zeigt das Schieben von Polygon A um Polygon B. Die Pfeile stellen das *NFP* dar. (c) zeigt das *NFP*, wenn Polygon A um 45° gedreht wurde. Angelehnt an [Bennell and Oliveira, 2008]

Die Nachteile von *NFPs* sind, dass sie für jedes Objekt-Paar erstellt werden müssen. Dies kann zwar bei der Initialisierung geschehen und muss nicht während der Laufzeit passieren, allerdings dürfen die Objekte dann nicht rotiert werden [Egeblad, 2008]. Die Probleme bei der Rotation sind im Unterschied der *NFPs* in Abbildung 2.2b und Abbildung 2.2c zu sehen. Eine Drehung um 45° verändert das *NFP* signifikant, so dass dieses für jede mögliche Drehung erneut erstellt werden müsste.
Für 3D-Objekte wird diese Art der Repräsentation als *no-fit polyhedron* bezeichnet. Ihre Erzeugung ist jedoch wesentlich schwieriger als die der *NFPs* [Verkhoturov et al., 2016]. Die Autoren erwähnen zwar einen Ansatz mittels Tiefensuche, erläutern diesen jedoch nicht weiter.

2.4.3 Bounding Volumes

Polytope werden in der Regel als *Meshes* dargestellt. Der naive Ansatz, eine Packung auf Kollisonen zu überprüfen, hat allerdings einen exponentiellen Zeitaufwand in Abhängigkeit der Anzahl der Kanten der einzelnen Polygone [Bennell and Oliveira, 2008]. Daher werden häufig sogenannte *bounding volumes (BV)* eingesetzt. Dabei wird ein Polygon über eine simplere, geometrische Form angenähert. Dies kann zum Beispiel ein Kreis oder eine Kugel sein oder eine Box. Zur Kollisionserkennung werden nun in einem ersten Schritt nur die jeweileigen *BVs* gegeneinander getestet. Nur wenn diese kollidieren, werden die Poly-

gone geprüft um, eine genaue Kollisionsermittlung zu erreichen. Ein Verfeinerung dieses Ansatzes ist es, sogenannte *bounding volume hierarchies (BVH)* zu bilden. Diese werden häufig zur Kollisionserkennung verwendet, da sie besonderes performant sind [Weller and Zachmann, 2009]. Der Aufwand für die Kollisionserkennung hängt dann vom Aufwand für den Test zweier *BVs* und der Anzahl der zu testenden *BVs* ab [Zachmann, 1998].
Wichtig für diese Arbeit sind die Datenstrukturen *Inner Sphere Trees* und *DOP-Trees*, welche im Folgenden näher erläutert werden.

2.4.3.1 Inner Sphere Trees

Bei *Inner Sphere Trees (IST)* handelt es sich um eine hierarchische Datenstruktur auf Basis von Kugelpackungen. Im Gegensatz zu anderen *bounding volumes* liegt diese Datenstruktur innerhalb des Objektes [Weller and Zachmann, 2009]. Die Autoren beschreiben nun einen Ansatz, um aus einer Kugelpackung einen *IST* zu erstellen. Dazu wird für alle Kugeln der Kugelpackung eine Boundingsphere berechnet. Diese wird als Wurzel der Hierarchie gewählt. Mittels des *batch neural gas* Algorithmus werden dann Cluster der enthaltenen Kugeln gebildet. Dieser Vorgang läuft rekursiv, bis eine Abbruchbedingung erfüllt wird. Daraus ergibt sich ein Baum, bei dem die Kugeln der Kugelpackung die Blätter sind und die inneren Knoten die berechneten *Boundingspheres*.
Statt einer *layered hierarchy*, bei der garantiert ist, dass jeder Knoten alle seine Kindknoten vollständig enthält, wird eine *wrapped hierarchy* verwendet [Weller and Zachmann, 2009]. Hierbei ist nur garantiert, dass ein Knoten alle Blätter umfasst, die sich über seine Kindknoten erreichen lassen. Dadurch sollen die Volumina der inneren Knoten so klein wie möglich gehalten werden. Zu sehen ist dies in Abbildung 2.3.

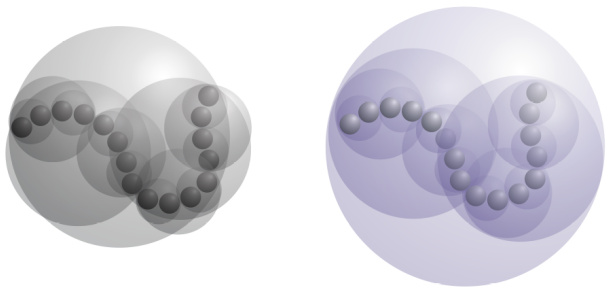


Abbildung 2.3: *Wrapped hierarchy* links, *layered hierarchy* rechts. Entnommen aus [Sobottka and Weber, 2005]

Kollisionen werden nun über die Traversierung der Hierarchien der Objekte erkannt. Die Traversierung kann dann abgebrochen werden, wenn die Distanz zweier Knoten größer ist als die Summe ihrer Radien.

In [Kaluschke et al., 2020] beschreiben die Autoren einen Ansatz, die Kollisionserkennung mittels *ISTs* auf der Graffikkarte zu parallelisieren. Ihr Anwendungsfall ist dabei die Simulation verschiedener medizinischer Eingriffe in *virtual reality*. Der Parallelisierungsansatz, den sie beschreiben, nutzt statt der Hierarchie beider Objekte nur die Hierarchie eines Objektes und testet parallel alle Kugeln der Kugelpackung des anderen Objektes gegen diese Hierarchie.

2.4.3.2 DOP-Trees

Eine weitere Möglichkeit, Objekte mittels einer *BVH* darzustellen, sind *discrete orientation polytopes (DOP)*. Diese werden in [Kay and Kajiya, 1986] eingeführt und umschließen ein Objekt dabei mit Flächen, die durch einen Normalenvektor und eine Entfernung zum Ursprung beschrieben werden. Dabei werden immer zwei Flächen zu einem Normalenvektor gewählt. Um den Speicherverbrauch zu optimieren schlagen die Autoren vor, die Normalenvektoren im Vorfeld zu wählen und für alle Objekte beizubehalten, so dass nur die Entfernungen der einzelnen Flächen zum Ursprung für jedes Objekt gespeichert werden muss. Wie eng das Objekt dabei vom *DOP* umschlossen wird, hängt von der Anzahl der gewählten Normalenvektoren ab. Zu sehen ist dies in Abbildung 2.4.

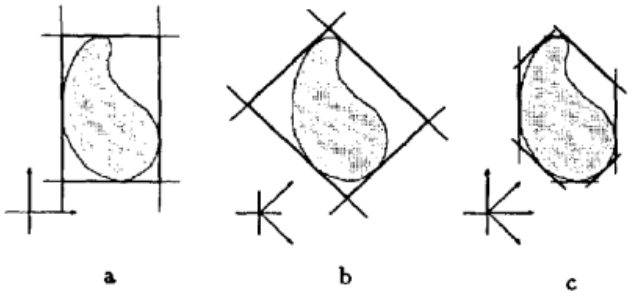


Abbildung 2.4: Zu sehen sind die verschiedenen DOPs für unterschiedliche Normalenvektoren. Für a und b wurden jeweils zwei Normalenvektoren genutzt, c wird von einem DOP umschlossen, welches alle Normalenvektoren aus a und b nutzt. Entnommen aus [Kay and Kajiya, 1986]

Der in [Zachmann, 1998] beschriebene Ansatz nutzt zum Aufbau der Hierarchie nun *DOPs*. Dabei wird für jedes Polygon eines Objektes ein *DOP* berechnet. In einem rekursiven Verfahren werden nun die zwei *DOPs* aus der Menge der übergebenen *DOPs* ermittelt, bei denen die Baryzentren des eingeschlossenen Polygons den größten Abstand haben. Die übrigen *DOPs* werden diesen beiden zugeordnet, sodass zum einen das dadurch enstehende *BV* möglichst klein ist. Falls das Hinzufügen eines Polygons zu einer Menge keinen Effekt auf das Volumen hat, dann wird die Menge gewählt, die weniger Polygone enthält.

2.5 Kollisionsauflösung

Werden bei der Packung Kollisionen zwischen den Objekten zugelassen, so müssen diese aufgelöst werden. Dazu werden die Objekte verschoben und rotiert. Verschiebungen von Objekten werden als Translation entlang einer oder mehrer Koordinatenachsen beschrieben. Rotationen werden in der Regel als eulersche Rotationen betreachtet. Auf diese wird im Folgenden genauer eingegangen.

2.5.1 Rotation in 3D

Rotationen im dreidimensionalen Raum lassen sich auf verschiedene Weisen darstellen. Eine gängige Art Rotationen zu beschreiben ist mittels dreier Winkeln, von denen jeder die Rotation um eine Koordinatenachste des Koordinatensystems beschreibt. Diese Winkel werden auch eulersche Winkel genannt. Dabei wird jede Rotation um eine Koordinatenachse mit Hilfe einer eigenen Matrix beschrieben. Diese Rotationen können mittels Matrixmultiplikation verkettet werden um jede mögliche Orientierung eines Objektes im dreidimensionalen Raum zu erhalten. Die Rotation mittels eulerscher Winkel wird auch eulersche Rotation genannt und hat die folgenden zwei Probleme.
Zum einen ist die Rotation nicht kommutativ. Stellen die drei Matrizen R_x, R_y und R_z eine Rotation um die x-, y- bzw. z-Achse dar und x einen Punkt, der rotiert werden soll, so gilt $R_x R_y R_z \cdot x \neq R_z R_y R_x \cdot x$. Zu sehen ist dies in Abbildung 2.5.

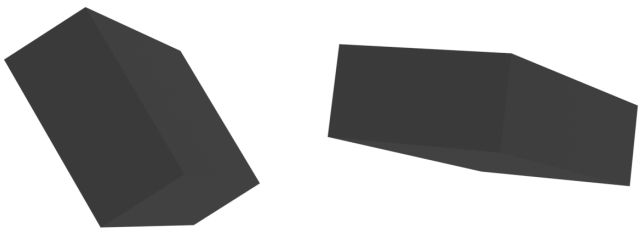


Abbildung 2.5: Beide Quader wurden jeweils 45° um die x-, y- und z-Achse rotiert. Der linke in der Reihenfolge XYZ, der rechte in der Reihenfolge ZYX. Deutlich zu sehen ist, dass die beiden Quader unabhängig von der räumlichen Verschiebung unterschiedlich rotiert sind.

Das zweite Problem ist das sogenannte *Gimbal Lock*. Dabei geht ein Freiheitsgrad verloren. Das heißt, dass eine Achse so gedreht wurde, das eine Rotation um diese Achse der Rotation um eine andere Achse gleicht. Sehen lässt sich dies in Gleichung 2.1(angelehnt an [Vince, 2011]), bei dem 90° um die y-Achse rotiert wird. Die Rotation um die x-Achse wird mittels des Winkels α, die Rotation um die z-Achse mittels des Winkels γ angegeben. Man sieht in der finalen Rotationsmatrix, dass eine Änderung des Winkels α oder γ

die selben Einträge der Matrix ändert. Einer der beiden Winkel hat also nicht mehr den gewünschten Einfluss auf die Rotation des Objektes. Eine Änderung der Reihenfolge der Matrizen in der Multiplikation ist dabei keine Lösung, da sie das Ergebnis ändert, wie in Abbildung 2.5 gezeigt. Außerdem verlagert sich das Problem so nur, so dass ein *Gimbal Lock* auftreten würde, wenn die Drehung um 90° in der dann mittleren Matrix auftritt.

$$
\begin{aligned}
rotation &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\alpha & -sin\alpha \\ 0 & sin\alpha & cos\alpha \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} cos\gamma & -sin\gamma & 0 \\ sin\gamma & cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 0 & 0 & 1 \\ sin(\alpha+\gamma) & cos(\alpha+\gamma) & 0 \\ -cos(\alpha+\gamma) & sin(\alpha+\gamma) & 0 \end{bmatrix}
\end{aligned} \tag{2.1}
$$

2.5.2 Quaternionen

Quaternionen sind 4-dimensionale Zahlen, durch die sich Rotationen im dreidimensionalen Raum beschreiben lassen. Quaternionen bestehen dabei aus einem Skalar und einem 3D-Vektor (siehe Gleichung 2.2) [Vince, 2011]. Hierbei sind s, v_x, v_y und v_z relle und i, j und k imaginäre Zahlen. Eine Einheitsquaternion ist dabei eine Quaternion für die gilt $\sqrt{s^2 + v_x^2 + v_y^2 + v_z^2} = 1$.

$$
q = s + v = s + v_x i + v_y j + v_z k \tag{2.2}
$$

Die Rotation eines Punktes um eine beliebige Achse mit Winkel α lässt sich mittels Quaternionen beschreiben als qpq^{-1}. Hierbei ist $q = cos(\frac{\alpha}{2}) + sin(\frac{\alpha}{2})i + sin(\frac{\alpha}{2})j + sin(\frac{\alpha}{2})k$ eine Einheitsquaternion und beschreibt die Achse, um die rotiert werden soll. q^{-1} ist die Inverse der Quaternion. Der Punkt, der rotiert werden soll, wird mittels $p = 0 + p_x i + p_y j + p_z k$ beschrieben. Sind die Achse, die durch die Quaternion beschrieben wird und der Punktvektor nicht rechtwinklig zueinander, so wird der Punktvektor durch die Multiplikation skaliert. Die Multiplikation mit der inversen Quaternion macht diese Skalierung rückgängig, rotiert den Punkt allerdings ein weiteres Mal um die angegeben Achse um den Winkel α.

2.5.3 Rotation in verwandten Arbeiten

Der Umgang mit der Rotation von Objekten wird verschieden gehandhabt, jedoch lassen sich drei Arten identifizieren:

- Freie Rotation

- Festgelegte Rotation
- Keine Rotation

In [Egeblad et al., 2009] werden keine Rotationen zugelassen. Kollisionen zwischen Objekten werden hier nur durch Bewegungen entlang einer Achse aufgelöst. Die Autoren stellen allerdings eine Erweiterung vor, die dann Rotationen der Objekte erlaubt. Diese ist auf 90° Schritte um die Achsen beschränkt.
Auch in [Eisenbrand et al., 2003] wird die Anzahl der möglichen Rotationen der Objekte eingeschränkt. Die Autoren lassen sechs verschiedene Rotationen zu. Diese werden durch die Platzierung der Boxen innerhalb des Grids erreicht, wobei eine gepackte Box dann nur Zellen einnehmen darf, die nicht diagonal zueinander liegen.
In [Gogate and Pande, 2008] beschreiben die Autoren ein System, welches Objekte in die Baukammer eines 3D-Druckers packt. Ein wichtiger Aspekt bei manchen 3D-Drucktechniken ist die Orientierung der Objekte, da dies Einfluss auf die Qualität der Oberflächen hat. Die erlaubten Rotationen werden vom Nutzer in einem ersten Schritt gewählt, wobei das Programm einen Gütewert für die Rotation berechnet. Da eine Rotation um die z-Achse (die Druckrichtung) keinen Einfluss auf die Qualität hat, erlauben die Autoren hier, dass das System in 45° Schritten um diese Achse rotiert um eine Lösung zu finden.
In [Ma et al., 2018] erlauben die Autoren beliebige Rotationen um die Achsen des Koordinatensystem und nutzen dafür eulersche Rotationen. Sie begrenzen die Rotationen der Objekte für die initiale Packung jedoch auf kleine Anpassungen. Dies führt dazu, dass mehr Zeit benötigt wird, damit die Objekte eine geeignete Position finden.

Kapitel 3

Umsetzung

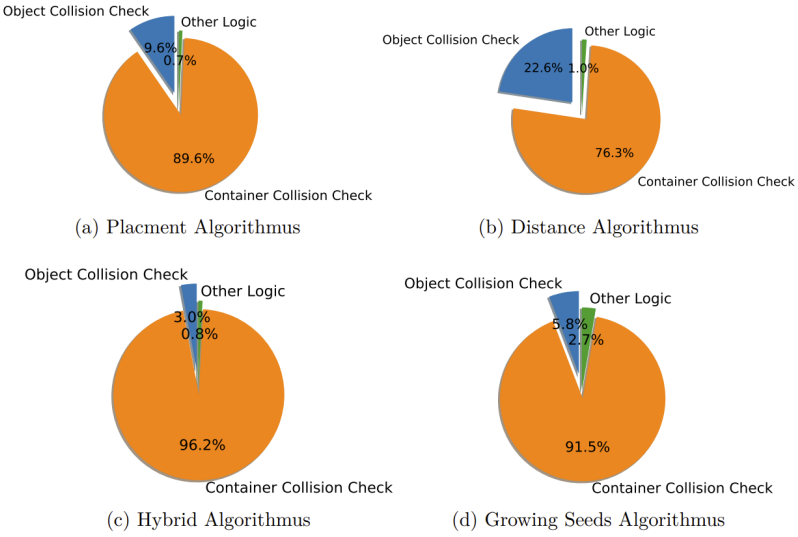
In diesem Abschnitt werden die verschiedenen Anpassungen an *AutoPacking* erklärt. Sie basieren dabei alle auf dem *Growing Seeds Algorithmus*, der in der Masterthesis entwickelt wurde. Diese Wahl wurde getroffen, da der Algorithmus als bester ausgewiesen wird, was das Verhältnis Packungsdichte zu Laufzeit betrifft.
Wie auch schon in Meißenhelters Masterthesis wird die Bibliothek *CollDet*[1] zur Kollisionserkennung verwendet. Hier muss eine Pipeline innerhalb des Codes erstellt werden, in der alle Objekte registriert und auf Kollisionen untereinander überprüft werden. Über sogenannte *callbacks* wird dann definiert, was im Falle eine Kollision geschehen soll. *Colldet* bietet dabei zur Objektrepräsentation *ISTs*, *DOP-Trees* und *BoxTrees* an.
Die Kugelpackungen zur Repräsentation der Objekte und des Containers werden mit Hilfe von Protosphere[2] erstellt.

3.1 Problembeschreibung

Meißenhelter beschreibt in seiner Arbeit, dass vor allem die Auflösung bzw. das Erkennen von Kollisionen zwischen Objekten und Container einen signifikanten Anteil der Laufzeit ausmacht. Diese unterscheidet sich je nach gewähltem Algorithmus, liegt jedoch bei mindestens 76,3%. Für den in dieser Arbeit betrachteten *Growing Seeds Algorithmus* liegt sie bei 91,5%. Zu sehen ist die Verteilung in Abbildung 3.1. Es erscheint also sinnvoll, die Zeit zur Kollisionserkennung zwischen Objekten und Container zu verringern, da diese durchgehend den größten Anteil an der Laufzeit hat.

Ein weiteres Problem bei der Erkennung von Kollisionen zwischen Objekt und Container ist die Genauigkeit der Darstellung des Containers. Abbildung 3.2 zeigt stark aus dem Container ragende Objekte, die auf eine ungenaue Repränsentation des Container zurückzuführen sind. Da ein Objekt nur platziert wird, wenn alle erkannten Kollisionen aufgelöst wurden, scheint hier die Repräsentation der Containerform über eine einzelne Kugelpackung nicht genau genug zu sein.

[1] `https://cgvr.cs.uni-bremen.de/research/colldet/`, Abgerufen am: 09.10.2021
[2] `https://cgvr.cs.uni-bremen.de/research/protosphere/index.shtml`, Abgerufen am 23.10.2021



(a) Placment Algorithmus (b) Distance Algorithmus

(c) Hybrid Algorithmus (d) Growing Seeds Algorithmus

Abbildung 3.1: Aufschlüsselung der Laufzeit der verschiedenen Algorithmen. Entnommen aus [Meißenhelter, 2019, S. 41]

3.2 Aufteilung des Containers

Für die in [Meißenhelter, 2019] beschriebene Kollisionserkennung, muss der Container in eine ihn vollständig umschließende Box gepackt werden. Zu sehen ist dies in Abbildung 3.3. Protosphere erzeugt dann eine Kugelpackung, wobei Kugeln sowohl in der Box, als auch im eigentlichen Container platziert werden. Diese Kugelpackung wird dann über Graphen analysiert, wobei die Kugeln die Knoten des Graphen darstellen. Eine Kante wird nur dann zwischen zwei Knoten gesetzt, wenn die Kugeln nicht durch Dreiecke getrennt sind. Meißenhelter nutzt nur die Packung für die Repräsentation des Containers, die durch den Graphen beschrieben wird, welcher alle Kugeln außerhalb des eigentlichen Containers umfasst. Ein Problem bei der Erstellung der Kugelpackung für den Container über die Beschreibung als Graphen beschreibt Meißenhelter wie folgt: „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.“ [Meißenhelter, 2019, S. 33].

In *Colldet* wird aus dieser Kugelpackung ein *IST* gebildet. Ein weiteres Problem dabei ist, dass dieser *IST* eigentlich eine Repräsentation des Objektes von innen darstellt, das innere des Containers jedoch leer sein muss, damit dort Objekte platziert werden können. Bei der Bildung der Hierarchie geht der in Unterunterabschnitt 2.4.3.1 grob beschriebene Algorithmus allerdings davon aus, dass die Kugeln innerhalb des darzustellenden Objektes liegen. Dies führt dazu, dass auch bei der Nutzung einer *wrapped hierarchy* der gesamte Container innerhalb dieser Kugeln liegt. Bei der Traversierung werden also zwischen den

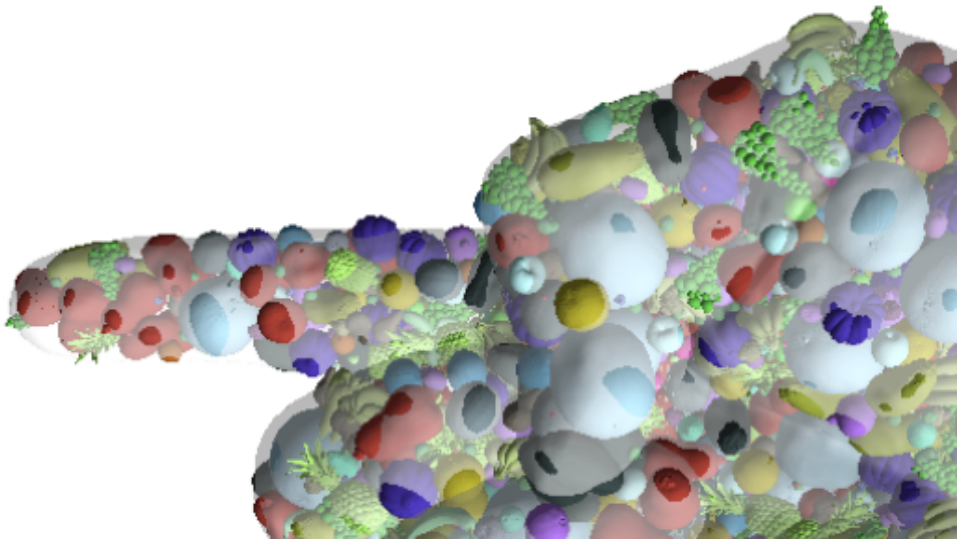


Abbildung 3.2: Packungsergebnis der Hand aus [Meißenhelter, 2019]. Die farblich kräftigen Stellen zeigen, wo die Objekte aus dem Container ragen.

inneren Knoten der *ISTs* Kollisionen festgestellt, obwohl die Blätter der Hierarchie nicht miteinander kollidieren.
Beide Probleme werden durch die Aufteilung in acht Teilboxen umgangen. Da für jede Box eine separate Kugelpackung erzeugt wird, müssen keine Graphen mehr berechnet werden. Daher werden keine Kugeln mehr aussortiert. Außerdem liegt nicht mehr der ganze Container komplett im *IST* des Container, sondern nur noch Abschnitte. Zu sehen ist die Aufteilung in Abbildung 3.4a, welche beispielhaft eine Hälfte zeigt. Die andere Hälfte des Containers wird analog dazu geteilt. Abbildung 3.4b zeigt die entstehende Kugelpackung für diese Teilcontainer. Man erkennt hier, dass die vier Kugelpackungen voneinader getrennt sind.

Die urpsrüngliche Variante von *AutoPacking* kann nur eine Kugelpackung für den Container verarbeiten. Das Programm wurde dahingehend angepasst, dass in den Konfigurationsdateien für das jeweilige Szenario beliebig viele Kugelpackungen für den Container angegeben werden können. Für jede Kugelpackung wird dann ein eigenes Containerobjekt erstellt und in der Kollisionspipeline registriert. Es werden dann *Callbacks* für jedes Objekt und jeden Container angelegt.
In Tests hat sich gezeigt, dass bei der gleichmäßigen Aufteilung acht Teilboxen bessere Ergebnisse liefern als zwei oder vier Teilboxen.

3.3 Parallelisierung auf der Grafikkarte

Ein weiterer Ansatz, der verfolgt wurde, war, die Kollsionserkennung zwischen Objekten und dem Container auf die Grafikkarte auszulagern. Dazu wurden in *CollDet* die zwei

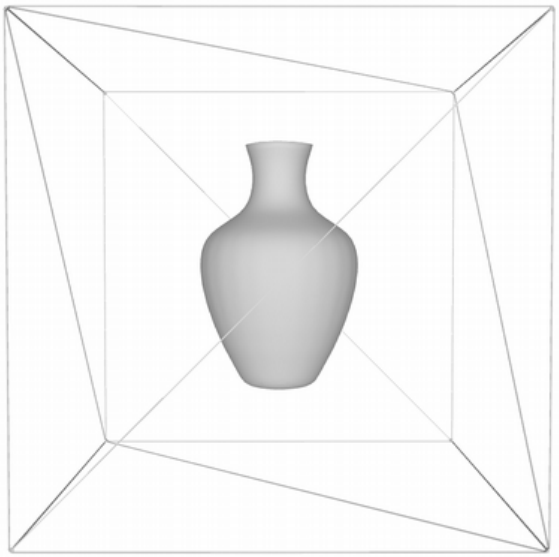


Abbildung 3.3: Vasenmodell in der Box

Varianten *GPUSpheres* und *GPUForce* implementiert. Sie unterscheiden sich in der Art der Rückgabedaten an *AutoPacking*. Es gibt jedoch auch Teile, die in beiden Varianten gleich funktionieren. Im Folgenden werden zuerst die gleichen Teile vorgestellt und dann die jeweilige Erzeugung der Rückgabedaten. In beiden Varianten wird die Hierarchie des Containers nicht genutzt. Dadurch soll das Problem der vermuteten langen Traversierungszeit, falls keine Kollision zwischen den Kugelpackungen zu finden ist, umgangen werden. Beide Implementierungen sind darauf angewiesen, dass die Hierarchien der Objekte sowie die Kugeln des Containers im Speicher der Grafikkarte zu finden sind. Damit nicht für jede Kollisionserkennung die *ISTs* der Objekte sowie die Kugeln des Containers neu in den Speicher der Grafikkarte geladen werden müssen, passiert dies in einem Schritt während der Initialisierung von *AutoPacking*. Zu sehen ist dies in Algorithmus 1.

Algorithmus 1 Initialisierung GPU Speicher

Input: Objects, Container
1: **for all** $o \leftarrow Objects$ **do**
2: copyHierarchyToGPU(o)
3: **end for**
4: **for all** $c \leftarrow Container$ **do**
5: copySpheresToGPU(c)
6: **end for**

Für die Speicherung sowohl der Kugeln als auch der *ISTs* während eines Programmdurchlaufes auf der GPU wurde eine eigene Klasse (*CUDASphereNode*) implementiert. Diese enthält das Zentrum sowie den Radius der entsprechenden Kugel und Pointer auf die Kindknoten. Außerdem wird eine Methode bereit gestellt, um zu überprüfen, ob es sich bei der Kugel um ein Blatt in der Hierarchie handelt.

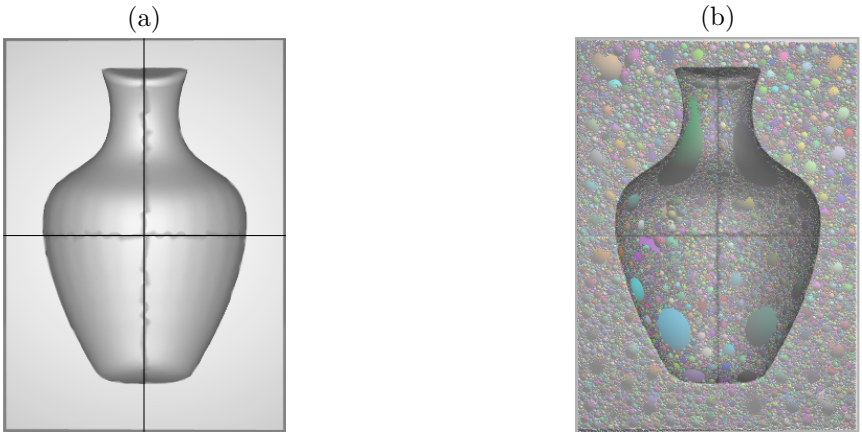


Abbildung 3.4: Abbildung a zeigt vier Teilcontainer des Containers aus dem Szenario *Vase mit Herzen*. Die hellgrauen Bereiche stellen die Teile dar, die über die Kugelpackung den Container represäntieren. Die Vase im Inneren ist der eigentliche Container. Abbildung b zeigt die sich ergebende Kugelpackung für diese Aufteilung.

Auch die Traversierung der Hierarchie der Objekte ist in beiden Methoden gleich. Der Code ist dabei eine Anpassung des Codes von Tero Karras zu Traversierung von *BVHs*, welchen er in [Karras, 2012a], [Karras, 2012b] und [Karras, 2012c] vorstellt sowie eine Anpassung des Codes von Maximilian Kaluschke, der im Rahmen von [Kaluschke et al., 2020] entwickelt wurde.
In beiden Varianten wird dabei die Hierarchie des Objektes mittels Breitensuche traversiert. Dies verhindert laut [Karras, 2012b] sogenannte *execution divergence*, welche sich negativ auf die Laufzeit auswirkt. Zu sehen ist die Traversierung der Hierarchie in Algorithmus 2.

Die Funktion *checkOverlap* überprüft dann, ob es für die jeweilige Kugel in der Hierarchie und die Kugel des Containers eine Kollision gibt. Sie ist für die beiden Varianten unterschiedlich implementiert, bricht jedoch ab, sollte keine Kollision gefunden werden. Diese Überprüfung ist in beiden Varianten gleich. Dazu wird die Kugel des Containers in das Koordinatensystem des Objektes transformiert. Eine Kollision zwischen den beiden Kugeln gibt es nur dann, wenn gilt $dist < r_1 + r_2$, wobei d die Distanz der beiden Kugelmittelpunkte zueinander beschreibt und r_1, r_2 die Radien der jeweiligen Kugeln.

3.3.1 GPUSpheres

Ähnlich zur Kollisionerkennung auf der CPU findet dieser Ansatz alle Kugelpaare, die tatsächlich kollidieren, und gibt diese an *AutoPacking* zurück. Da vor der Kollision nicht bekannt ist, wie viele Kugelpaare es bei der Kollision geben wird, wird beim Initialisieren des Programmes Arbeitsspeicher sowie Speicher auf der Grafikkarte reserviert. Sollte es

Algorithmus 2 Traversierung der Objekthierarchie

Input: $objectRoot, containerSphere, containerToObjectMatrix$

```
1: CUDASphereNode* stack[64]
2: CUDASphereNode** stackPtr = stack
3: CUDASphereNode* node = objectRoot
4: *stackPtr++ = NULL
5: do
6:     nodeChildren* = node-¿children
7:     bool overlap[4]
8:     for i ← 0, 3 do
9:         overlap[i] = checkOverlap(nodeChildren[i], containerSphere, containerToObjectMatrix)
10:    end for
11:    for i ← 0, 3 do
12:        if overlap[i] AND NOT nodeChildren[i]-¿isLeaf() then
13:            *stackPtr++ = nodeChildren[i]
14:        end if
15:    end for
16:    node = *–stackPtr
17: while node != NULL
```

sich bei der Kugel der Hierarchie um ein Blatt handeln und die Kugeln kollidieren, so werden die Zentren und die Radien der beiden Kugeln in diesen Speicher geschrieben. Ein Zähler erfasst die Anzahl der festgestellten Kollisionen, damit nur der Speicherbereich, der tatsächlich kollidierende Kugelpaare enthält, von der Grafikkarte wieder in den Arbeitsspeicher kopiert werden muss.

Algorithmus 3 Überprüfung auf Kollisionen

Input: objectNode, containerSphere, containerToObjectMatrix, outputBuffer, bufferSize, collisionCounter

```
1: transformedContainerSphere = containerToObjectMatrix · containerSphere
2: distance = dist(objectNode, transformedContainerSphere)
3: if distance >objectNode.radius + transformedContainerSphere.radius then
4:     return false
5: end if
6: if objectNode.isLeaf() then
7:     placeInBuffer = atomicAdd(collisionCounter, 1)
8:     if placeInBuffer <bufferSize then
9:         Add data to buffer
10:    end if
11: end if
12: return true
```

3.3.2 GPUForce

Die Callbacks, die die Bewegungsrichtung der Objekte berechnen, müssen sowohl bei der Erkennung der Kollision auf der CPU als auch bei der Erkennung mittels der Variante *GPUSpheres* alle Kugelpaare auswerten. Zu sehen ist dies in Algorithmus 4. Allerdings liegen diese Daten der Kugelpaare bereits auf der GPU vor, wenn auf Kollisionen geprüft wird.

Algorithmus 4 Berechnung der Kollisionspunkte

Input: *collidingSpheres, obj1ToWorld, obj2ToWorld*

```
1: averageNormal = Vector3(0,0,0)
2: averageCenter = Vector3(0,0,0)
3: nrAverageNormals = 0
4: for all spherePair ← collidingSpheres do
5:     obj1InWorld = obj1ToWorld · spherePair.obj1Center
6:     obj2InWorld = obj2ToWorld · spherePair.obj2Center
7:     averageNormal += normalize(obj1InWorld - obj2InWorld)
8:     averageCenter += obj1InWorld + obj2InWorld
9:     nrAverageNormals += 1
10: end for
```

Die Idee hinter dieser Variante ist es, deshalb die entsprechenden Daten auf der GPU zu berechnen und an *AutoPacking* zurückzugeben. So müssen nur noch die Bewegungsrichtungen berechnet werden. Dazu wurde der Code zu Kollisionserkennung aus Algorithmus 3 so angepasst, dass, statt die Kugelpaare in einen Ausgabepuffer zu schreiben, nun die Daten aus Algorithmus 4 berechnet und in entsprechende Puffer geschrieben werden. Zu sehen ist dies in Algorithmus 5. Ein möglicher Nachteil dieser Variante ist, dass sie viele *atomicAdd*-Aufrufe hat, welche andere Threads potentiell blockieren können.

Algorithmus 5 Berechnung der Kollisionsvektoren auf der GPU

Input: objectNode, containerSphere, containerToObjectMatrix, outputBuffer, bufferSize, collisionCounter

```
1: transformedContainerSphere = containerToObjectMatrix · containerSphere
2: distance = dist(objectNode, transformedContainerSphere)
3: if distance >objectNode.radius + transformedContainerSphere.radius then
4:     return false
5: end if
6: if objectNode.isLeaf() then
7:     obj1InWorld = obj1ToWorld · spherePair.obj1Center
8:     obj2InWorld = obj2ToWorld · spherePair.obj2Center
9:     atomicAdd(averageNormal, normalize(obj1InWorld - obj2InWorld))
10:    atomicAdd(averageCenter, obj1InWorld + obj2InWorld)
11:    atomicAdd(nrAverageNormals, 1)
12: end if
13: return true
```

3.4 DOP-Tree Verbesserung

Wie beschrieben war ein Problem, dass die gepackten Objekte mehr oder weniger stark aus dem Container ragen. Dies liegt an einer ungenauen Kollisionserkennung am Rand des Containers. Eine Idee, die in [Meißenhelter, 2019] genannt wurde, ist es, den Container nicht mittels einer Kugelpackung zu repräsentieren, sondern eine andere Datenstruktur zu wählen. Daher wurde implementiert, dass die Kollisionserkennung für den Container und Objekte auch über einen *DOP-Tree* stattfinden kann. Dabei wird für jeden Objekttyp und den Container ein *DOP-Tree* erstellt. Diese Datenstruktur wird, wie in Unterunterabschnitt 2.4.3.2 beschrieben, bereits von *CollDet* unterstützt. Der Vorteil ist, dass diese Datenstruktur mit den tatsächlichen Meshdaten der Objekte und des Containers arbeitet und daher eine genauere Darstellung der Objekte ermöglicht als es über *ISTs* möglich ist. Damit nicht für jedes einzelne Objekt ein eigener *DOP-Tree* angelegt werden muss, wurde *CollDet* so erweitert, dass für jeden Objekttyp ein *DOP-Tree* erstellt und in jedem Kollisionsobjekt referenziert wird. Diese Implementierung ist analog zur instanziierten Speicherung der *ISTs*. Wird ein neues Objekt in der Kollisionspipeline von *CollDet* registriert, so bekommt es nur einen Pointer auf den *DOP-Tree*. Die *DOP-Trees* werden erzeugt, wenn die einzelnen Objekttypen von *AutoPacking* geladen werden.

3.5 Border Filling Algorithm

Für das Ziel, die Oberflächendichte der Packungen zu erhöhen, wurde eine neue Verbesserungsheuristik implementiert. Diese orientiert sich an der lokalen Suchvariante *Cavity-Filling* aus [Meißenhelter, 2019]. Auch sie versucht, Objekte an passenden Orten wachsen zu lassen, erzeugt jedoch auf Grundlage der Samplingpunkte für die initiale Packung neue Samplingpunkte am Containerrand. Sie arbeitet ähnlich zu dem in Unterabschnitt 2.3.1 beschriebenen Prinzip und versucht auf jedem Samplingpunkt das größtmögliche Objekt zu platzieren.
Die Heuristik arbeitet in vier Phasen:

- Generierung neuer Samplingpunkte
- Klassifizierung der Objekte
- Probing
- Platzierung der Objekte

Die ersten beiden Phasen laufen dabei einmalig ab, die letzten beiden Phasen wechseln sich gegenseitig ab. Im folgenden werden die einzelnen Phasen näher erläutert.

3.5.1 Generierung neuer Samplingpunkte

Der Algorithmus erzeugt nicht im ganzen Container neue Samplingpunkte, sondern berechnet diese auf Basis der alten Samplingpunkte. Zu sehen ist dieser Ablauf in Abbildung 3.5. Zuerst sucht der Algorithmus die Randpunkte der urpsrünglichen Samplingpunkte (zu sehen in Abbildung 3.5b). Dazu filtert er die ursprünglichen Samplingpunkte nach denen, die in einer oder mehreren Koordinaten-Richtungen keinen Nachbarn mehr haben. Ein Punkt hat dann keinen Nachbarn in einer Richtung, wenn alle anderen Punkte in dieser Richtung mindestens $stepSize \cdot 1.1$ entfernt liegen. Zu sehen ist dieses Vorgehen in Algorithmus 6. Es werden alle Punkte so behandelt, als würden sie am Rand liegen, und sollte festgestellt werden, dass für den betrachteten Punkt diese Bedingung nicht gilt, so wird er entsprechend markiert.
Von den so gefunden Punkten werden neue Samplingpunkte in der gleichen Art und Weise erzeugt, wie dies auch im initialen Sampling der Fall ist. Dabei wird eine kleinere *stepSize* gewählt und es werden nur Punkte erzeugt, für die in der entsprechenden Richtung gilt: $P_{new} \leqslant P_{start} + oldStepSize$. So wird verhindert, dass Punkte erzeugt werden, die hinter dem vom intialen Sampling aussortierten direkten Nachbarn liegen. Abbildung 3.5d zeigt die so neu erzeugten Punkte in grün. In schwarz sind die Randpunkte dargestellt und in grau die Punkte, die durch das intiale Sampling aussortiert wurden, weil sie zu nah am Rand oder außerhalb des Containers lagen. Unter Umständen wurden diese Punkte durch das initiale Sampling auch gar nicht erst erzeugt, da nur Punkte innerhalb der Koordinaten $AABB$ des Containers erzeugt werden.
Diese neuen Punkte werden nochmals gestreut, um ein feineres Sampling zu erreichen. Dabei wird der Koordinatenwert der Achse fixiert, auf der der neue Punkt Richtung Containerrand erzeugt wird. Die beiden anderen Koordinatenwerte werden anhand der neuen *stepsize* verändert. Zu sehen ist das Ergebnis in Abbildung 3.5e, wobei die gestreuten Punkte grün sind.
Die so erzeugten Punkte werden nun gefiltert. Zuerst werden Duplikate entfernt, da es beim Streuen der Punkte passieren kann, dass ein Punkt zweimal erzeugt wird. Nachfolgend werden wie beim initialen Sampling alle Punkte mittels *CollDet* überprüft, ob diese in Objekten oder im Container liegen. Dazu wird eine Kugel mit dem kleinem Volumen auf jedem Punkt platziert und überprüft, ob es eine Kollision mit dem Container oder einem Objekt gibt. Bei Kollisionen mit dem Container wird der Punkt verworfen, bei Kollisionen mit Objekten wird geprüft, ob das Kollisionsvolumen einen Schwellwert übersteigt. Anschließend werden die übriggebliebenen Punkte noch nach ihrer Distanz zueinander gefiltert. Dabei werden Punkte aussortiert, die näher beieinander liegen als der Radius, den eine Kugel mit dem durchschnittlichen Volumen aller bereits gepackten Objekte hat.

Algorithmus 6 Filtern der alten Samplingpunkte

Input: *SamplingPoints*
Output: *BorderPoints*

```
1:  for all s ← SamplingPoints do
2:      xN, xP, yN, yP, zN, zP ← true
3:      for all s' ← SamplingPoints do
4:          if abs(s.x - s'.x) ⩽ stepSize · 1.1 AND s.y = s'.y AND s.z = s'.z then
5:              if s.x > s'.x then
6:                  xN ← false
7:              else if s.x < s'.x then
8:                  xP ← false
9:              end if
10:         end if
11:         if s.x = s'.x AND abs(s.y - s'.y) ⩽ stepSize · 1.1 AND s.z = s'.z then
12:             if s.y > s'.y then
13:                 yN ← false
14:             else if s.y < s'.y then
15:                 yP ← false
16:             end if
17:         end if
18:         if s.x = s'.x AND s.y = s'.y AND abs(s.z - s'.z) ⩽ stepSize · 1.1 then
19:             if s.z > s'.z then
20:                 zN ← false
21:             else if s.z < s'.z then
22:                 zP ← false
23:             end if
24:         end if
25:     end for
26:     if xN OR xP OR yN OR yP OR zN OR zP then
27:         calculateNewSamplingPoints(s, BorderPoints)
28:     end if
29: end for
```

Algorithmus 7 Streuung der neuen Punkte

```
1: stepSizeRatio ← oldStepSize/newStepSize
2: if Axis = xPositive then
3:     for i ← 1, stepSizeRatio do
4:         p ← (s.x + newStepSize · i, s.y, s.z)
5:         BorderPoints.add(p)
6:         scatterPoint(Axis, p, newStepSize, stepSizeRatio, BorderPoints)
7:     end for
8: end if
9: if Axis = xNegative then
10:    for i ← 1, stepSizeRatio do
11:        p ← (s.x − newStepSize · i, s.y, s.z)
12:        BorderPoints.add(p)
13:        scatterPoint(Axis, p, newStepSize, stepSizeRatio, BorderPoints)
14:    end for
15: end if
16: if Axis = yPositive then
17:    for i ← 1, stepSizeRatio do
18:        p ← (s.x, s.y + newStepSize · i, s.z)
19:        BorderPoints.add(p)
20:        scatterPoint(Axis, p, newStepSize, stepSizeRatio, BorderPoints)
21:    end for
22: end if
23: if Axis = yNegative then
24:    for i ← 1, stepSizeRatio do
25:        p ← (s.x, s.y − newStepSize · i, s.z)
26:        BorderPoints.add(p)
27:        scatterPoint(Axis, p, newStepSize, stepSizeRatio, BorderPoints)
28:    end for
29: end if
30: if Axis = zPostive then
31:    for i ← 1, stepSizeRatio do
32:        p ← (s.x, s.y, s.z + newStepSize · i)
33:        BorderPoints.add(p)
34:        scatterPoint(Axis, p, newStepSize, stepSizeRatio, BorderPoints)
35:    end for
36: end if
37: if Axis = zNegative then
38:    for i ← 1, stepSizeRatio do
39:        p ← (s.x, s.y, s.z − newStepSize · i)
40:        BorderPoints.add(p)
41:        scatterPoint(Axis, p, newStepSize, stepSizeRatio, BorderPoints)
42:    end for
43: end if
```

3.5.2 Klassifizierung der Objekte

Die Objekte werden anhand ihres Volumes absteigend sortiert, da der Algorithmus nach dem *first fit decreasing* Ansatz arbeitet und so unter Umständen viele Objekte ausprobieren muss. Für große Objektmengen kann diese Überprüfung zeitaufwendig werden, da im schlimmsten Fall auf jedem Samplingpunkt jedes Objekt getestet wird. Für die Worst-Case-Laufzeit ergibt sich also:

$$runtime = |objects| \cdot |SamplingPoints| \cdot time_{objectCheck} \tag{3.1}$$

In Tests stellte sich heraus, dass eine Auswahl der Objekte nach dem vermutlich passenden Volumen die Laufzeit deutlich reduziert, dass Ergebnis jedoch nicht verschlechtert. Eine weitere Optimierung dieses Ansatzes war es, die Objekte anhand ihres Volumens zu klassifizieren. Die Klassen werden durch das Volumen des größten der enthaltenden Objekte charakterisiert. Ein Objekt wird einer Klasse zugeordnet, wenn sein Volumen mindestens einem bestimmten Prozentsatz des charakterisierenden Volumens entspricht. Ist das Volumen kleiner als dieser Prozenzsatz, so wird für das Objekt eine neue Klasse gebildet und alle Objekte mit kleinerem Volumen, welchen jedoch dem Kriterium entsprechen, werden dieser Klasse zugeordnet. Tests ergaben, dass 80% hier einen guten Wert für den Prozentsatz darstellt, ab dem eine neue Klasse gebildet wird.

3.5.3 Probing

Das Probing dient dazu, das vermutete maximale Volumen, welches ein Objekt auf einem bestimmten Samplingpunkt besitzen kann, zu ermitteln. Dazu wird ein Objekt des größten zur Verfügung stehenden Objekttyps auf diesem Samplingpunkt platziert und langsam größer skaliert, ähnlich der Vorgehensweise wie im *Growing Seeds Algorithmus* aus der Masterarbeit von Meißenelter [Meißenhelter, 2019]. Wird festgestellt, dass das Objekt nicht mehr wachsen kann, so wird das Volumen nach der Formel $V = V_{Obj} \cdot s^3$ berechnet, die Meißenhelter angibt. Außerdem werden die letzten Kollisionspunkte gespeichert.
Mit Hilfe der Kollisionspunkte versucht der Algorithmus nun, eine mögliche Rotation zu bestimmen. Dafür werden die Kollisionspunkte in das Weltkoordinatensystem transformiert. Anschließend werden drei linear voneinander unabhängige Vektoren zwischen diesen Punkten gesucht, wobei der Ursprung dieser Vektoren immer derselbe Kollisionspunkt sein muss. Sollte für einen Punkt ein Vektor gefunden werden, der länger ist als der längste Vektor aus bereits drei gefundenen linear voneinander unabhängigen Vektoren, so werden diese verworfen. Dies geschieht auch, wenn sich für den neuen Vektor keine zwei weiteren Vekoren finden lassen, so dass diese drei Vektoren linear voneinander unabhängigen sind. So wird sichergestellt, dass die Länge der Vektoren für die neue Basis den Raum optimal nutzt.

Sind drei linear unabhängige Vektoren gefunden worden, so wird aus diesen eine Orthonormalbasis mit Hilfe des *Gram-Schmidtsches Orthogonalisierungsverfahrens* gebildet [of Mathematics, 2011]. Das Verfahren ist in Gleichung 3.2 gezeigt, wobei $v \bullet w$ das Skalarprodukt zweier Vektoren beschreibt und $|v|$ den Betrag eines Vektors. Die Vektoren x_n, y_n und z_n stellen dann die normierten Vektoren der neuen Basis dar.

$$\begin{aligned} x_n &= \frac{v_1}{|v_1|} \\ y &= v_1 - (x_n \bullet v_1) \cdot x_n \\ y_n &= \frac{y}{|y|} \\ z &= v_2 - (x_n \bullet v_2) \cdot x_n - (y_n \bullet v_2) \cdot y_n \\ z_n &= \frac{z}{|z|} \end{aligned} \tag{3.2}$$

Die Objekte sollen dann mithilfe der neu gefundenen Basis rotiert werden. Dafür wird eine Matrix gebildet, die in der ersten Spalte den Vektor x_n, in der zweiten Spalte den Vektor y_n und in der dritten Spalte den Vektor z_n in den oberen Einträgen enthält. Da die x-Achse der neuen Basis immer entlang des längsten gefundenen Vektors zeigt, wird für jeden Objekttyp im *Preprocessing* von *AutoPacking* eine Rotationsmatrix berechnet, die das Objekt so rotiert, dass die längste Ausdehnung der *AABB* in x-Richtung und die zweitlängste Ausdehnung in y-Richtung des Weltkoordinantensystems liegt.
Ein Ansatz, Winkel zwischen den verschiedenen Vektoren und den Koordinatenachsen zu berechnen, wurde aufgrund der in Unterabschnitt 2.5.1 beschriebenen Probleme bei der Rotation mittels Eulerscher Winkel verworfen. Auch die Rotation über Quaternionen wurde nicht weiter verfolgt, da auch hier das Problem besteht, eine geeignete Rotationsachse finden zu müssen.

3.5.4 Platzieren der Objekte

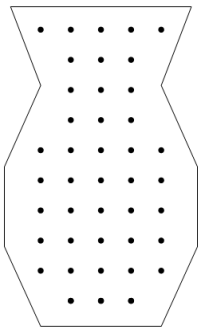
Das Platzieren der Objekte ist in Algorithmus 8 dargestellt. Bei der Platzierung der Objekte ist zu beachten, dass beim Probing das Probingobjekt im Container leicht verschoben werden kann und auch die bereits gepackten Objekte verschoben werden. Daher wird das Objekt nicht auf dem Samplingpunkt platziert, sondern auf der letzten Position des Probingobjektes. Auch werden die Objekte erst an ihre ursprüngliche Position geschoben, wenn kein neues Objekt plaziert werden konnte. Andernfalls muss das neu zu platzierende Objekt die vom Probingobjekt bereits verschobenen Objekte erneut verschieben. Hierbei kann es passieren, dass durch das Probing genug freier Platz erkannt wurde und das ausgewählte Objekt eigentlich passen würde, aber durch die Dauer bei der Kollisionsauflösung nicht platziert werden kann, da diese den Timeout überschreitet.

Wurde eine neue Basis gefunden, so werden Objekte bevorzugt, deren $AABB$ die Dimensionen der gefundenen Basis nicht überschreitet. Sollte für die aktuelle Objektklasse kein so rotiertes Objekt passen, so wird ein zufälliges Objekt aus der Klasse mit zufälliger Rotation gewählt.

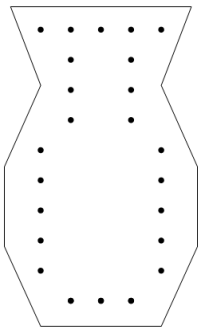
Algorithmus 8 Platzieren von Objekten

Input: *samplingPoints, objectClassBuffer*

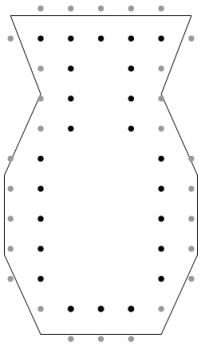
```
 1: for all s ← SamplingPoints do
 2:     probe(s)
 3:     if Probingobject fit then
 4:         Fill buffer
 5:         Move to next sampling point
 6:     end if
 7:     for all objectClass ← objectClassBuffer do
 8:         if Basis was found then
 9:             for all object ← objectClass do
10:                 if Object dimensions fit basis dimensions then
11:                     if Object fit then
12:                         Fill buffer
13:                         Move to next sampling point
14:                     end if
15:                 end if
16:             end for
17:         end if
18:         if ObjectClassVolume <  ProbingVolume then
19:             if Fit random object then
20:                 Fill buffer
21:                 Move to next sampling point
22:             end if
23:         end if
24:     end for
25: end for
```



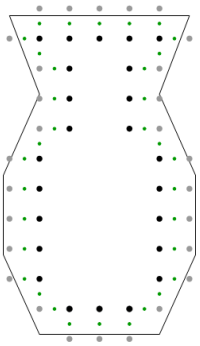
(a) Die initialen Samplingpunkte



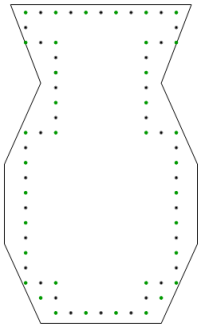
(b) Die Randpunkte der initialen Samplingpunkte



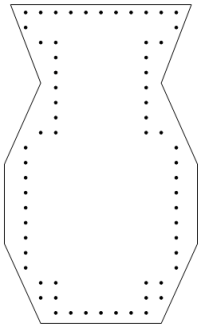
(c) Die Randpunkte in schwarz, in grau die Samplingpunkte, die durch das initiale Sampling aussortiert oder gar nicht erzeugt wurden



(d) In grün die neu erzeugten Samplingpunkte mit der neuen Schrittweite



(e) Die von den neuen Samplingpunkten(schwarz) aus gestreuten Punkte(grün)



(f) Die neuen, gefilterten Samplingpunkte

Abbildung 3.5: Das Resampling entlang des Containerrandes auf Basis der alten Samplingpunkte

Kapitel 4

Evaluation

Die verschiedenen in dieser Arbeit entwickelten Ansätze werden miteinander sowie mit dem Ursprungszustand aus Meißenhelters Masterarbeit verglichen. Für den Vergleich wurden die zwei Szenarien *Vase mit Herzen*[1] und die *Hand mit Früchten*[2] gewählt, die schon Meißenhelter in seiner Arbeit betrachtete. Für beide Szenarien wurden jeweils vier Konfigurationen angelegt. Dabei wurde die Zielverteilung der Objekte für beide Szenarien in jeder Konfiguration gleich gelassen, lediglich die Parameter zur Erzeugung der Samplingpunkte wurden angepasst. Dabei wurde darauf geachtet, dass beide Szenarien eine Konfiguration haben, die ungefähr so viele Samplingpunkte erzeugt, wie in Vortests Objekte in den entsprechenden Container gepackt werden können. Bei der Hand entspricht dies ungefähr 1.000 Objekten, bei der Vase sind es ungefähr 100 Objekte. Darauf aufbauend wurden drei weitere Konfigurationen erstellt, eine mit 1,5 mal so vielen, eine mit 2 mal so vielen und eine mit 3 mal so vielen Samplingpunkten. Zu sehen sind die Konfigurationen und ihre Namen in Tabelle 4.1.

Name	Samplingpunkte Vase	Samplingpunkte Hand
LSP	≈100	≈1.000
LMSP	≈150	≈1.500
HMSP	≈200	≈2.000
HSP	≈300	≈3.000

Tabelle 4.1: Benennung und Samplingpunktanzahl für die verschiedenen Konfiugrationen und die beiden Szenarien *Vase mit Herzen* und *Hand mit Früchten*.

Um auch die Auswirkungen der Anzahl der Kugeln in den Kugelpackungen zu untersuchen, wurden für alle Modelle und die entsprechenden Teilcontainer neue Kugelpackungen erzeugt. Für die zu packenden Objekte jeweils mit 2.500, 5.000 und 10.000 Kugeln, für die Teilcontainer mit 25.000, 50.000 und 100.000 Kugeln. Für manche Objekte liegt die Anzahl der Kugeln für die Packung mit 10.000 Kugeln niedriger, da *Protosphere* nicht in der Lage war, mehr Kugeln zu packen.

[1] Original aus der Arbeit [Ma et al., 2018]
[2] Zur Verfügung gestellt von Peter Coffin

Alle Evalutionstestläufe liefen auf demselben Rechner mit folgender Hardware: Intel Core i7-7800X, NVIDIA GeForce RTX 2080 Ti, 64GB Arbeitsspeicher. Als Betriebssystem kam Windows 10 zum Einsatz.
Tabelle 4.2 listet die verschiedenen evaluierten Varianten auf. Die Kollisionserkennung zwischen den Objekten wird bei jeder Variante über Kugelpackungen und *ISTs* als *BVH* realisiert und auf der CPU berechnet.

Name	Containerrepräsentation	Kollisionserkennung auf
GT	Kugelpackung aus [Meißenhelter, 2019]. *IST* als *BVH* .	CPU
CPUEight	Acht Kugelpackungen. Trennung zwischen diesen entlang der Koordinatenachsen. *IST* als *BVH*.	CPU
GPUSpheres	Kugelpackung aus [Meißenhelter, 2019].	GPU
GPUForce	Kugelpackung aus [Meißenhelter, 2019].	GPU
CPUDopOne	Mesh des Containers. *DOP-Tree* als *BVH*.	CPU
CPUDopEight	Mesh des Containers getrennt in acht Teile entlang der Koordinatenachsen. *DOP-Tree* als *BVH*.	CPU
GPUSpheresEight	Acht Kugelpackungen. Trennung zwischen diesen entlang der Koordinatenachsen. *IST* als *BVH*..	GPU
GPUForceEight	Acht Kugelpackungen. Trennung zwischen diesen entlang der Koordinatenachsen. *IST* als *BVH*.	GPU

Tabelle 4.2: Beschreibung der Varianten mit ihrer Containerrepräsentation und auf welcher Recheneinheit der Kollisionscheck mit dem Container ausgeführt wird.

Sowohl *GPUSpheresEight*, als auch *GPUForceEight* werden nur im Kapitel Abschnitt 4.4 genutzt.

4.1 Laufzeit

Im Folgende werden die verschiedenen Laufzeiten für die beiden Szenarien miteinander verglichen. Dabei werden die Gesamlaufzeiten für jede Konfiguration der einzelnen Methoden betrachtet. Es wird hier auch immer die erreichte Packungsdichte[3] als Qualitätskriterium mit einbezogen, um zu verhindern, dass ein Ansatz zwar schnellere, qualitativ aber schlechtere Ergebnisse liefert.
Für das Szenario *Vase mit Herzen* sind die durchschnittlichen Gesamtlaufzeiten der verschiedenen Ansätze in Abbildung 4.1 gezeigt, Abbildung 4.2 zeigt die dabei erreichten

[3]Berechnet wie in [Meißenhelter, 2019].

durchschnittlichen Packungsdichten. Es stellt sich heraus, dass alle Ansätze geringere Gesamtlaufzeiten haben als der Vergleichswert *GT*. Dabei ist der Unterschied für alle Varianten stark ausgeprägt. Auffallend ist, dass die *CPUDopOne* und *CPUDopEight* in den Konfigurationen *HMSP* und *HSP* schneller laufen als alle anderen Ansätze. Nur für die Konfiguration *LSP* können alle Ansätze bis auf *GT* ähnlich gute Laufzeiten wie *CPUDopOne* erzielen.
Die beiden GPU-Varianten erzeugen die höchsten Packungsdichten und haben dabei sehr kurze Laufzeiten. Bei beiden *DOP-Tree*-Ansätzen fällt auf, dass die erreichten Packungsdichten niedriger liegen als bei den anderen Ansätzen. Zu sehen ist, dass die Varianten, die einen *IST* benutzen, zum Teil bessere Packungsdichten für dieselbe Konfiguration wie *GT* erreichen. Bei den Varianten *CPUDopOne* und *CPUDopEight* fällt auf, dass diese für die Konfiguration *HSP* deutlich schlechtere Ergebnisse liefern als der Vergleichswert. *CPUDopOne* ist auch für die Konfiguration *LMSP* schlechter.
Betrachtet man den Zuwachs der Packungsdichten zwischen den Konfigurationen *LSP* und *HSP*, so liegen hier die Varianten *GT* und *CPUEight* vorne. Alle anderen Varianten erreichen niedrigere Zuwächse. Bei den GPU-Varianten ist zu erkennen, dass diese auch mit wenigen Samplingpunkten bereits sehr hohe Packungsdichten erzielen, dann aber nach oben begrenzt sind. Dadurch fällt der Zuwachs nicht so stark aus. Die *DOP-Tree*-Varianten beginnen zwar mit ähnlichen Packungsdichten wie *GT*, erreichen aber auch mit vielen Samplingpunkten nicht so hohe Packungsdichten, wodurch der Zuwachs nicht so stark ausfällt.
Es zeigt sich das generelle Bild, welches auch von Meißenhelter in seiner Arbeit beschrieben wurde, dass sowohl Laufzeit als auch Packungsdichte von der Anzahl der Samplingpunkte abhängen, wobei mehr Samplingpunkte eine längere Laufzeit und eine höhere Packungsdichte bedeuten.

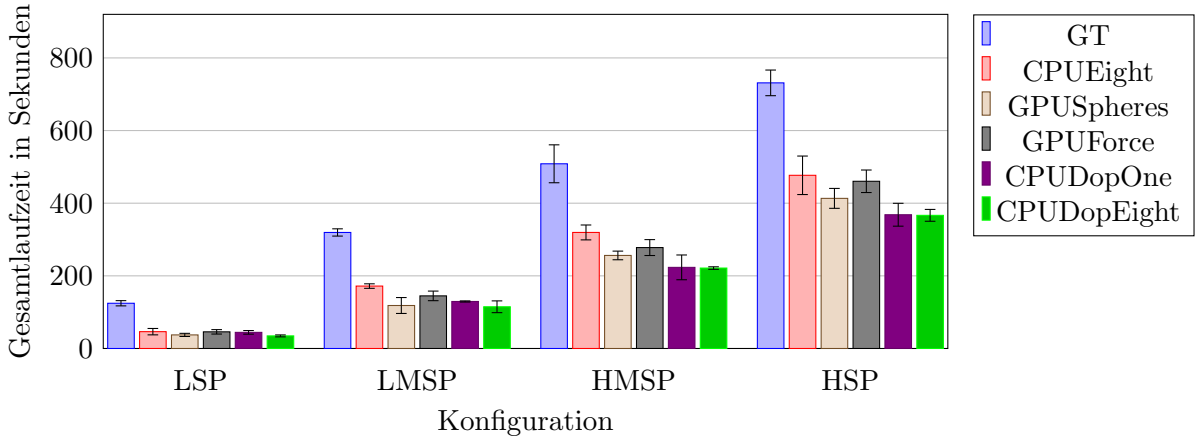


Abbildung 4.1: Vergleich der durchschnittlichen Laufzeiten und Standardabweichungen jedes Ansatzes für die verschiedenen Konfigurationen des Szenarios *Vase mit Herzen*. 3 Durchläufe pro Ansatz und Konfiguration.

Die Abbildung 4.3 und Abbildung 4.4 zeigen den Laufzeitenvergleich und den Packungs-

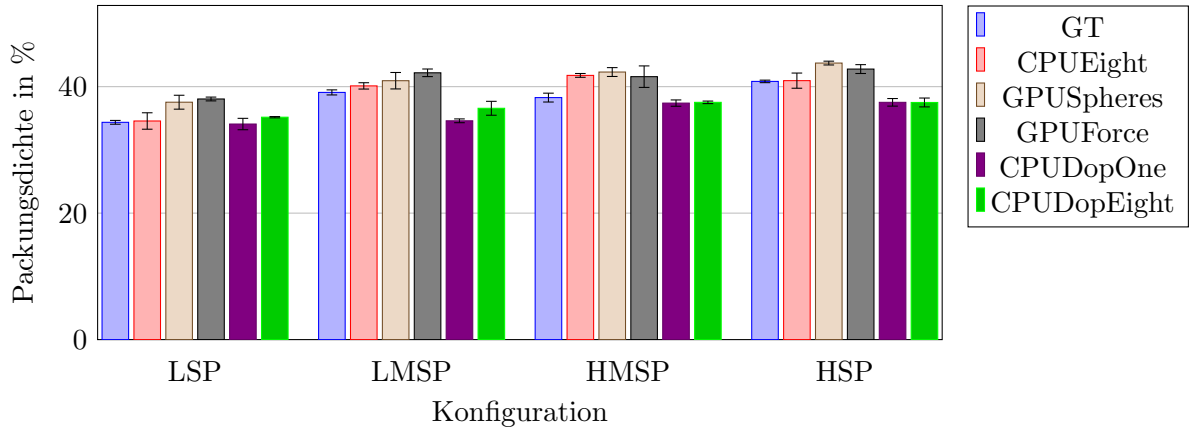


Abbildung 4.2: Vergleich der durchschnittlichen Packungsdichte für die Durchläufe aus Abbildung 4.1.

dichtenvergleich für das Szenario *Hand mit Früchten*. Es zeigt sich, dass alle Varianten, bis auf die Kollisionserkennung zwischen Objekten und Container mittels *DOP-Tree* ohne geteiltes Mesh, schneller sind als der Vergleichswert. Auch hier sind die Unterschiede wieder deutlich ausgeprägt. Dabei sind alle Varianten schneller als der Vergleichswert *GT*, bis auf die Variante *CPUDopOne*. Diese ist deutlich langsamer.
Im Vergleich der Packungsdichten schneiden die Ansätze *CPUDopOne* und *CPUDopEight* wieder am schlechtesten ab. Die beiden GPU-Ansätze prodzieren auch hier die höchsten Packungsdichten. Dabei unterscheiden sich diese nicht wesentlich von der Packungsdichte des Vergleichswertes. Auffallend ist, dass die Varianten *CPUEight*, *CPUDopOne* und *CPUDopEight* deutlich schlechtere Packungsdichten für alle Konfigurationen bis auf *LSP* erzeugen.
Anders als im Szenario *Vase mit Herzen* stellt sich der Vergleich zwischen *CPUEight* und *CPUDopEight* dar. Es fällt auf, dass *CPUDopEight* für das Szenario *Hand mit Früchten* keinen Geschwindigkeitsvorteil gegenüber den Kugelpackungen hat, wie es im anderen Szenario noch der Fall war.
Auch in diesem Szenario liefern die beiden GPU-Ansätze die besten Laufzeiten und Packungsdichten. Der Zuwachs der Packungsdichten zwischen den Konfigurationen liegt bei allen Konfigurationen bei ca. 30%. *CPUDopEight* und *GPUForce* liegen hier mit 26,6% bzw. 27% leicht darunter. Anders als im Szenario *Vase mit Herzen* produzieren die GPU-Varianten hier allerdings nicht bereits bei niedriger Samplingpunktanzahl deutlich höhere Packungsdichten.

4.2 Kollisionserkennung

Im folgenden werden die Zeiten für die Kollisonserkennung der verschiedenen Ansätze verglichen. Dazu werden die Laufzeiten für einen check()-Aufruf in der Kollisionspipeline

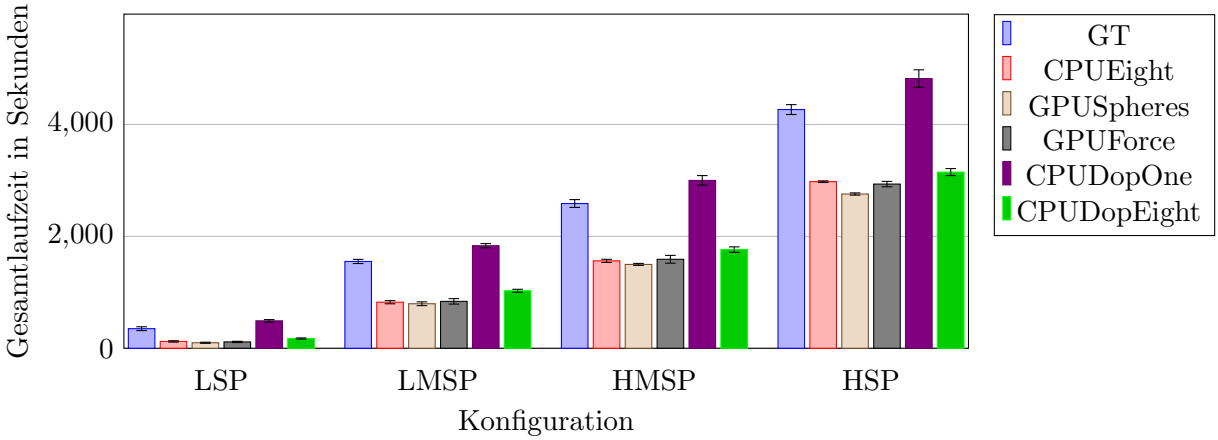


Abbildung 4.3: Vergleich der durchschnittlichen Laufzeiten und Standardabweichungen jedes Ansatzes für die verschiedenen Konfigurationen des Szenarios *Hand mit Früchten*. 3 Durchläufe pro Ansatz und Konfiguration.

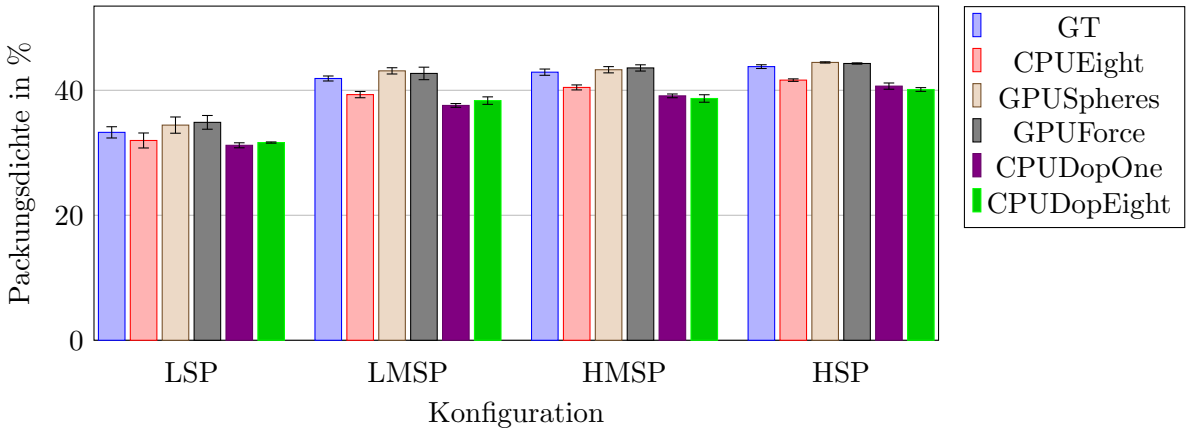


Abbildung 4.4: Vergleich der durchschnittlichen Packungsdichte für die Durchläufe aus Abbildung 4.3.

von *CollDet* betrachtet. Darüber hinaus wird verglichen, wieviel Zeit das Erkennen einer Kollision zwischen einem Objekt und dem Container beansprucht.
In Abbildung 4.5 sind die Durchschnittszeiten für einen check()-Aufruf für das Szenario *Vase mit Herzen* gezeigt. Zu sehen ist, dass die beiden *DOP-Tree*-Varianten deutlich am schnellsten sind. Für die *SphereTree*-Varianten schneiden die GPU-Varianten besser ab als *CPUEight*. Dieser Ansatz ist aber immer noch schneller als die urpsrüngliche Lösung mit nur einem Container. Die Unterschiede im Vergleich zu *GT* für die Durchschnittszeiten eines check()-Aufrufes sind für alle Varianten stark ausgeprägt. Zu erkennen ist, dass es auch hier eine Steigerung in Abhängigkeit von der Anzahl der Samplingpunkte gibt. Je mehr Samplingpunkte generiert und damit auch mehr Objekte im Container platziert werden, desto länger dauert die Überprüfung auf Kollisionen. Dies ist dadurch zu erklären, dass die Packungsdichten steigen und somit im Verlauf der Packung mehr Objekte im Container platziert sind. Bei mehr Objekten müssen potentiell mehr Objekte gegeneinander

geprüft werden, es werden aber auch mehr Objekte gegen den Container geprüft. Darum gilt auch hier, wie schon bei der Gesamtlaufzeit, dass mehr Samplingpunkte und damit mehr Objekte längere Zeiten bedeuten. Auffallend ist hierbei, dass *GT* den geringsten Zuwachs zwischen den Konfigurationen *LSP* und *HSP* hat mit gerade einmal 38%. Alle anderen Varianten verzeichnen hier einen Zuwachs von mehr als 60%.

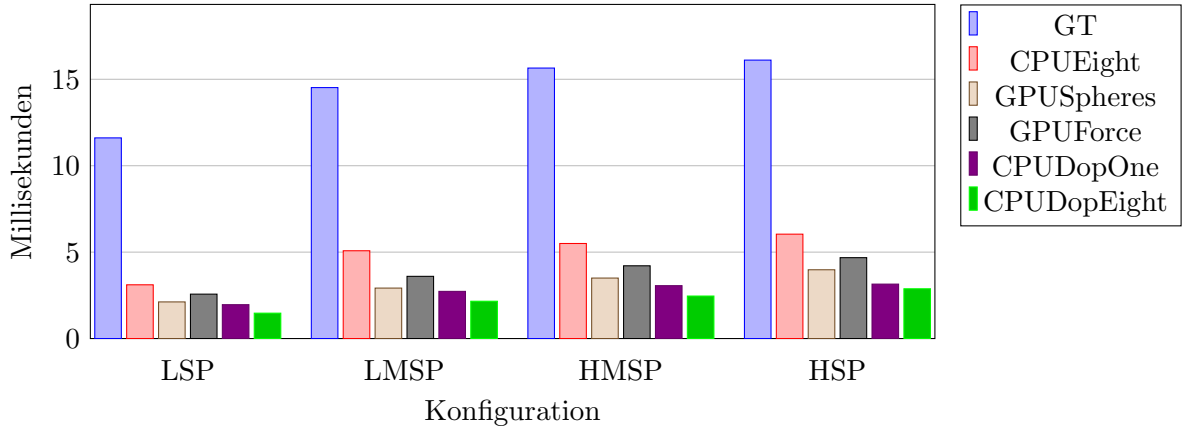


Abbildung 4.5: Durschnittszeiten für eine Überprüfung auf Kollisionen (ein check()-Aufruf auf der Pipeline) für das Szenario *Vase mit Herzen*.

Für das Szenario *Hand mit Früchten* ist dieser Vergleich in Abbildung 4.6 dargestellt. Es zeigt sich, dass die Repräsentation des Containers über nur einen *DOP-Tree* deutlich langsamer ist als alle anderen Varianten. Die Kollisionserkennung zwischen Objekten und Container auf der GPU sorgt für die schnellsten Laufzeiten. *CPUEight* liegt in diesem Szenario auf Rang drei noch vor *CPUDopEight*, anders als im anderen Szenario. Auch hier sind die Unterschiede im Vergleich zu *GT* wieder deutlich, wobei *CPUDopOne* langsamer ist, während alle anderen Varianten schneller sind.
Hier stellt es sich so dar, dass *CPUDopOne* den geringsten Zuwachs bei der Dauer in Abhängigkeit von der Samplingpunktanzahl hat. Bei allen anderen fällt der Zuwachs stärker aus, auch im Vergleich zu *GT*.

Da die Kollisionserkennung zwischen Containern und Objekten den größten Anteil an der Gesamtzeit zur Erkennung aller Kollisionen hat, wird diese im Folgenden näher betrachtet. Aus Platzgründen finden sich die Diagramme in Abschnitt A.1. Für das Szenario *Vase mit Herzen* ist dieser Vergleich in Abbildung A.1 gezeigt. Zu sehen ist, dass alle Varianten deutlich schneller sind, als der Vergleichswert *GT*. Besonders gut schneiden die Ansätze ab, bei denen der Container als *DOP-Tree* repräsentiert wird. Es zeigt sich, dass auch die Aufteilung des Meshes in acht Teilstücke hier die Zeit zur Erkennung einer Kollision mit dem Container noch einmal mehr als halbieren kann. Bei den drei Varianten, die einen *IST* benutzen, schneiden die GPU-Varianten besser ab als *CPUEight*. Bei allen Varianten zeigt sich, dass der stärkste Anstieg der Dauer zur Erkennung einer Kollision zwischen einem Objekt und dem Container zwischen den Konfigurationen *LSP* und *LMSP* liegt. Danach stellt sich diese Steigerung ein und es gibt nur kleinere Schwankungen in beiden

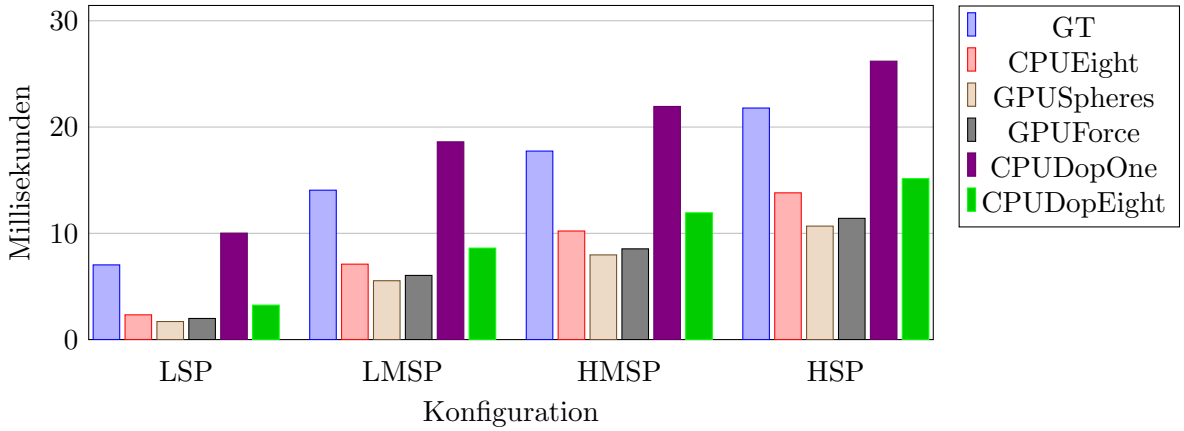


Abbildung 4.6: Durschnittszeiten für eine Überprüfung auf Kollisionen (ein check()-Aufruf auf der Pipeline) für das Szenario *Hand mit Früchten*.

Richtungen. Die durchschnittliche Zeit zur Erkennung einer Kollision zwischen einem Objekt und dem Container ist augenscheinlich nicht abhängig von der Anzahl der Objekte bzw. der Menge der Samplingpunkte. Sie hängt vermutlich eher damit zusammen, wie viele Objekte am Rand des Containers platziert werden. In Abbildung A.2 ist der Vergleich für das Szenario *Hand mit Früchten* gezeigt. Wie auch schon bei der Laufzeit für die check()-Aufrufe ist bei diesem Szenario die *CPUDopOne*-Variante deutlich am langsamsten. Ein Vergleich der beiden GPU-Ansätze zeigt, dass *GPUSpheres* fast doppelt so schnell ist wie *GPUForce*. *CPUEight* ist hier, anders als im anderen Szenario, schneller als *CPUDopEight*. Aber auch für dieses Szenario zeigt sich, dass die Unterschiede in den Zeiten stark ausgeprägt sind. Dabei gilt für *CPUDopOne* wieder, dass die Variante langsamer ist, wohingegen alle anderen Varianten schneller sind. Für die Varianten *GPUSpheres*, *GPUForce* und *CPUDopEight* gilt auch hier, dass keine nennenswerte Steigerung der Zeiten zwischen den Konfigurationen *LMSP* und *HSP* zu erkennen ist. Für die Varianten *GT*, *CPUEight* und *CPUDopOne* ist eine leichte Steigerung festzustellen.
In beiden Szenarien ist auffällig, dass bei der Kollisionserkennung zwischen einem Objekt und dem Container deutlich häufiger der Fall eintritt, dass die beiden Objekte nicht miteinander kollidieren. Dadurch lässt sich der Anstieg in den Durchschnittszeiten zwischen den Konfiugrationen *LSP* und *LMSP* erklären. Bei *LSP* werden etwas weniger Samplingpunkte erzeugt als Objekte in den Container passen. Dadurch kommt es zu weniger Kollisionen zwischen den Objekten und dem Container. Dies reduziert die durchschnittliche Zeit zur Erkennung einer Kollision zwischen Objekt und Container und somit auch die Laufzeit eines check()-Aufrufes.

4.3 Packungsqualität Containerrand

Im Folgenden werden die Ergebnisse hinsichtlich der Qualität am Containerrand betrachtet. Dies lässt sich zum einen durch die Anzahl der Objekte messen, die nach Abschließen das Packprozesses noch mit dem Container kollidieren. Dazu wird eine Kollisionsüberprüfung mittels *DOP-Tree* durchgeführt und jedes Objekt markiert, wenn es noch mit dem Container kollidiert.

Für die verschiedenen Konfigurationen des Szenarios *Vase mit Herzen* ist dieser Vergleich in Tabelle 4.3 zu sehen. Es ist zu erkennen, dass die beiden Ansätze, die den *DOP-Tree* zur Repräsentation des Containers verwenden, am besten abschneiden. Von den beiden GPU-Ansätzen schneidet *GPUSpheres* leicht besser ab. *CPUEight* erzeugt Packungen, bei denen weniger Objekte mit dem Container kollidieren als *GPUSpheres*, *GPUForce* und *GT*.
Da die Zahlen jedoch keine Auskunft darüber geben, wie stark die Objekte tatsächlich aus dem Container ragen, ist in Abbildung 4.7 ein beispielhafter visueller Vergleich zu sehen. Es fällt auf, dass die herausragenden Stellen der Objekte bei *CPUEight* deutlich kleiner sind als die von *GT*, *GPUSpheres* und *GPUForce*. Auf Bilder der Varianten *CPUDopOne* und *CPUDopEight* wird an dieser Stelle aus Platzgründen verzichtet. Diese Bilder sowie alle in der Abbildung gezeigten Bilder finden sich in größerer Darstellung in Abschnitt A.3.

Ansatz \ Konfiguration	LSP	LMSP	HMSP	HSP
GT	31	46,7	43,3	48,7
CPUEight	23,7	34,7	36,3	33,3
GPUSpheres	40,3	51,3	52,3	52,6
GPUForce	43	51,3	53	55
CPUDopOne	0	0	0	0
CPUDopEight	0	0	0	0

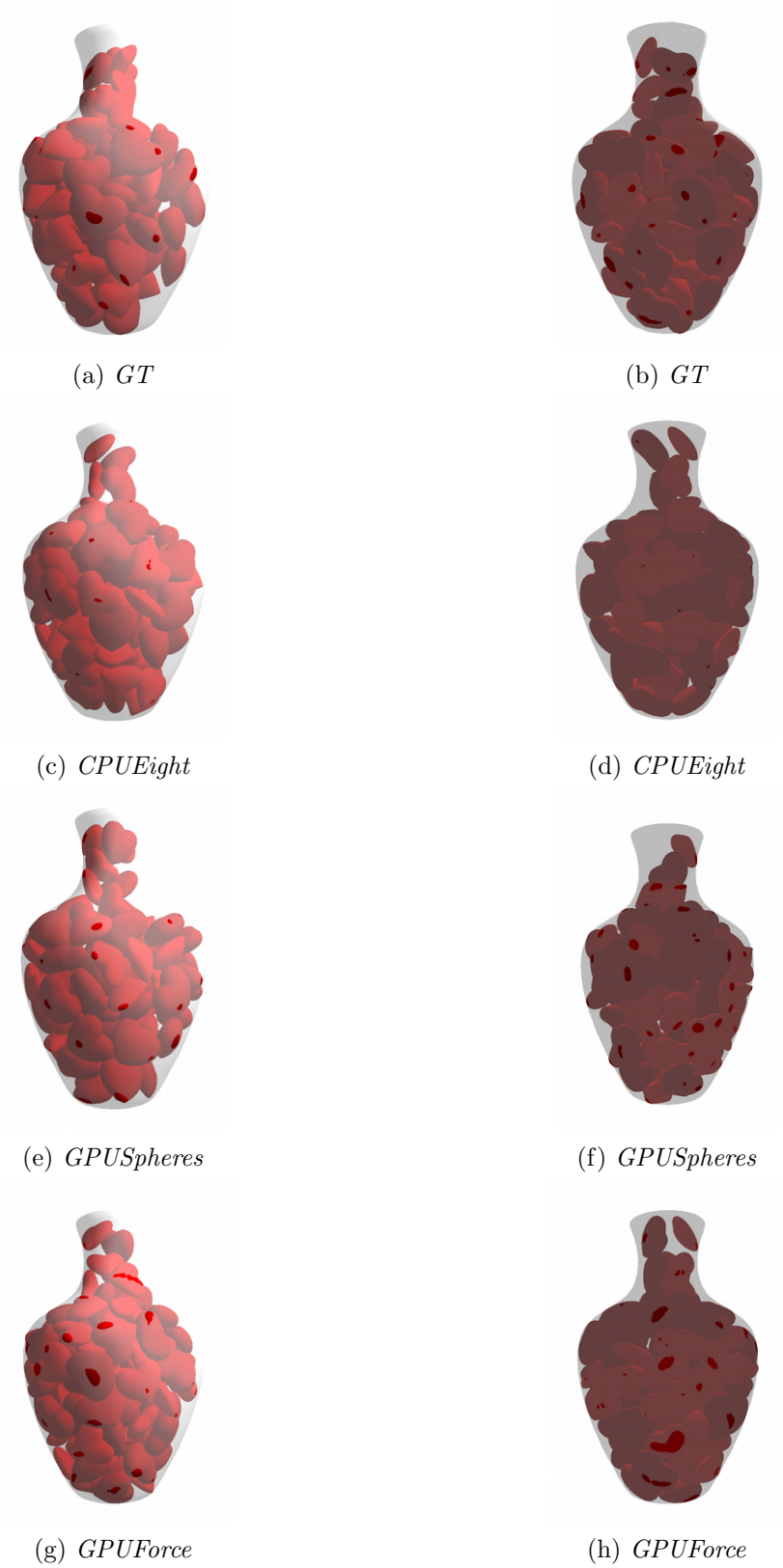
Tabelle 4.3: Durschnittliche Anzahl an Objekten, die in der finalen Packung noch mit dem Container kollidieren für das Szenario *Vase mit Herzen*.

Tabelle 4.4 zeigt die durchschnittliche Anzahl an Objekten, die in der finalen Packung noch mit dem Container kollidieren für das Szenario *Hand mit Früchten*. Es zeigt sich ein ähnliches Bild wie für das andere Szenario. Die Aufteilung des Containers sorgt dafür, dass weniger Objekte noch mit diesem kollidieren. Auch hier schneiden *GT*, *GPUSpheres* und *GPUForce* schlechter ab als *CPUEight*. Auffallend ist außerdem, dass für beide Varianten, die den Container mittels *DOP-Tree* repräsentieren, Packungen erzeugt werden, bei denen eine Überprüfung mittels *DOP-Tree* noch Kollisionen findet.
In Abbildung 4.8 ist der visuelle Vergleich von beispielhaft ausgwählten Packungen zu sehen. Wie auch im anderen Szenario ist das Herausragen der Objekte aus dem Container

in den Varianten *GT*, *GPUSpheres* und *GPUForce* deutlich stärker ausgeprägt als bei *CPUEight*.

Konfiguration / Ansatz	LSP	LMSP	HMSP	HSP
GT	204	354,3	384,3	413
CPUEight	100,3	177,3	191,3	227,7
GPUSpheres	220,3	398,7	412	435
GPUForce	226,3	384,4	415	440,7
CPUDopOne	0	0	0,3	0
CPUDopEight	0,3	0,3	0	0,3

Tabelle 4.4: Durschnittliche Anzahl an Objekten, die in der finalen Packung noch mit dem Container kollidieren für das Szenario *Hand mit Früchten*.



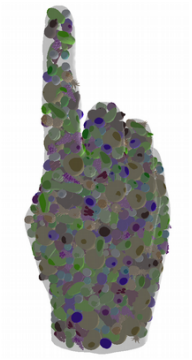
(a) *GT* (b) *GT*

(c) *CPUEight* (d) *CPUEight*

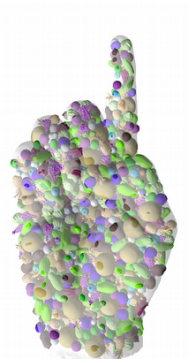
(e) *GPUSpheres* (f) *GPUSpheres*

(g) *GPUForce* (h) *GPUForce*

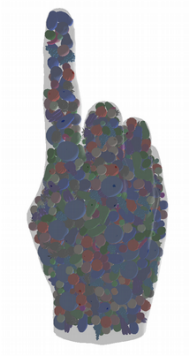
Abbildung 4.7: Visueller Vergleich der verschiedenen Ansätze. Kräftig gefärbte Stellen an den Objekten stellen ein Herausragen aus dem Container dar. Alle Bilder aus Packungen, die durch die Konfiguration HMSP erzeugt wurden, jeweils mit Front- und Rückansicht.



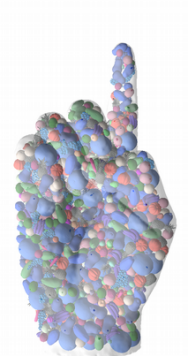
(a) *GT*



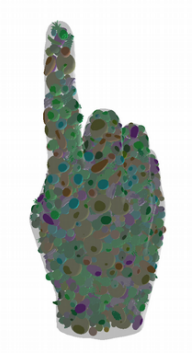
(b) *GT*



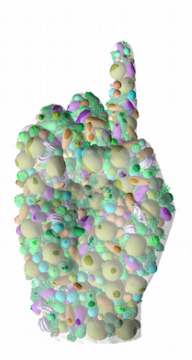
(c) *CPUEight*



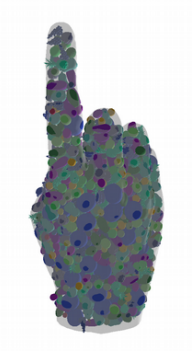
(d) *CPUEight*



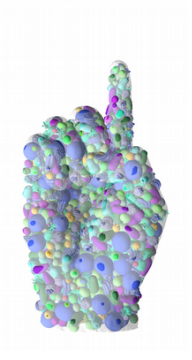
(e) *GPUSpheres*



(f) *GPUSpheres*



(g) *GPUForce*



(h) *GPUForce*

Abbildung 4.8: Visueller Vergleich der verschiedenen Ansätze. Kräftig gefärbte Stellen an den Objekten stellen ein Herausragen aus dem Container dar. Alle Bilder aus Packungen, die durch die Konfiguration HMSP erzeugt wurden, jeweils mit Front- und Rückansicht.

4.4 Einfluss der Kugelpackungsdichte auf die Laufzeit

Im Folgenden wird der Einfluss der Anzahl der Kugeln in den Kugelpackungen für Objekte und Container betrachtet. Hierzu wurden die Konfigurationen für beide Szenarien gewählt, die für die Vase ca. 200 Samplingpunkte und für die Hand ca. 2.000 Samplingpunkte erzeugen. Auch für die GPU-Varianten wurde die Aufteilung des Containers gewählt, im Gegensatz zu den Betrachtungen in Abschnitt 4.1 und Abschnitt 4.2, in denen die GPU-Varianten jeweils die Darstellung des Containers über nur einen *IST* nutzten.
Tabelle 4.5 zeigt die Bezeichnungen und die Anzahl der Kugeln für die Kugelpackungen mit denen der Einfluss getestet wurde. Dabei wurden alle Kugelpackungen für die Teilcontainer mit allen Kugelpackungen für die Objekte kombiniert, so dass sich für jedes Szenario insgesamt neun Konfigurationen ergeben.

Bezeichnung	Anzahl der Kugeln	Kugelpackung für
C25	≈25.000	Teilcontainer
C50	≈50.000	Teilcontainer
C100	≈100.000	Teilcontainer
O25	≈2.500	Objekt
O50	≈5.000	Objekt
O100	≈10.000	Objekt

Tabelle 4.5: Bezeichnungen der Kugelpackungen sowie die Anzahl der Kugeln in der jeweiligen Kugelpackung zur Evaluierung des Einflusses der Anzahl der Kugeln auf die Laufzeit.

4.4.1 CPUEight

Die Abbildung 4.9 zeigt die durchschnittlichen Laufzeiten für die unterschiedlichen Kugelpackungsgrößen. Die durchschnittlichen Packungsdichten liegen dabei im Bereich von 39,86% bis 42,39%. Wobei die Kombination aus 5.000 Kugeln für die Objekte und 25.000 Kugeln für die Teilcontainer den höchsten Wert erreicht. Am niedrigsten ist der Wert für die Kombination 10.000 Kugeln für die Objekte und 50.000 für den Container. Allerdings schwanken die Werte der Packungsdichten für die einzelnen Läufe zum Teil stark, so dass die Unterschiede der Laufzeiten dadurch begründet sein können.

Ein anderes Bild zeigt sich in Abbildung 4.10. Hier erkennt man, dass die Anzahl der Kugeln in den Objekten einen Einfluss auf die Gesamtlaufzeit hat. Die Anzahl der Kugeln im Container allerdings nicht. Die durchschnittlichen Packungsdichten reichen hier von 39,57% für 5.000 Kugeln in den Objekten und 100.000 Kugeln in jedem Teilcontainer bis zu 40,79% für 5.000 Kugeln für die Objekte und 25.000 Kugeln für die Teilcontainer.

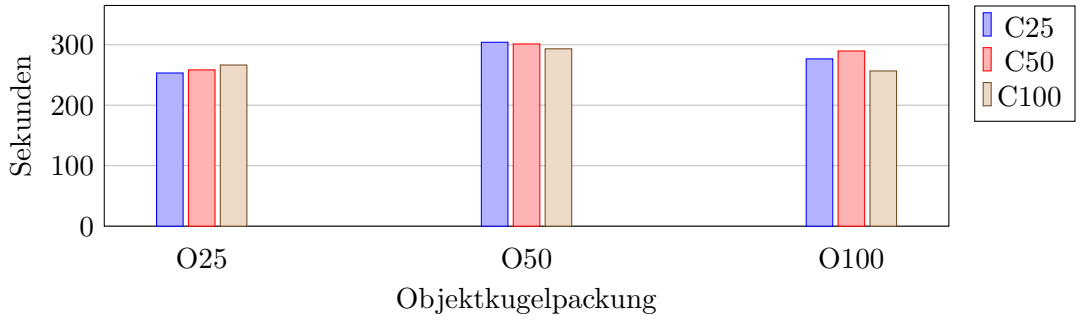


Abbildung 4.9: Durchschnittliche Gesamtlaufzeiten für die verschiedenen Kombinationen der Kugelpackung des Szenarios *Vase mit Herzen* für die Variante *CPUEight*.

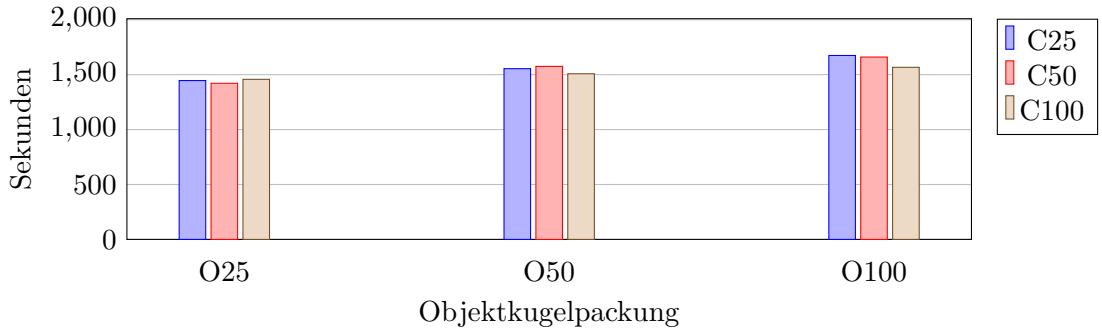


Abbildung 4.10: Durchschnittliche Gesamtlaufzeiten für die verschiedenen Kombinationen der Kugelpackung des Szenarios *Hand mit Früchten* für die Variante *CPUEight*.

Wie in Abschnitt 4.2 gezeigt wurde, hängt die Gesamlaufzeit stark von der Zeit zur Erkennung von Kollisionen zwischen den Objekten und dem Container ab. Im Folgenden werden daher die Durchschnittszeichen verglichen. Die Diagramme Abbildung 4.11 und Abbildung 4.12 zeigen für die beiden Szenarien, wie lange das Überprüfen eines Objektes gegen den Container dauert. Hierbei fällt auf, dass die Dauer von der Anzahl der Kugeln in den Objekten abhängig ist, nicht jedoch von der Anzahl der Kugeln innerhalb der Teilcontainer. Es ergibt sich das Bild, dass die Überprüfung auf eine Kollision umso länger dauert, je mehr Kugeln in der Kugelpackung des Objekte sind.

4.4.2 GPUSpheresEight

Für die Variante *GPUSpheresEight* sind die durschnittlichen Gesamlaufzeiten für das Szenarion *Vase mit Herzen* für die verschiedenen Kombinationen aus Anzahl der Kugeln für die Objektkugelpackungen und Containerkugelpackungen in Abbildung 4.13 gezeigt. Auch hier fällt, wie schon in Unterabschnitt 4.4.1 auf, dass die Unterschiede der Laufzeiten durch Schwankungen innerhalb der Werte verursacht werden. Die durchschnittlichen Packungsdichten liegen zwischen 38,23% für die Kombination aus 10.000 Kugeln für die Objekte

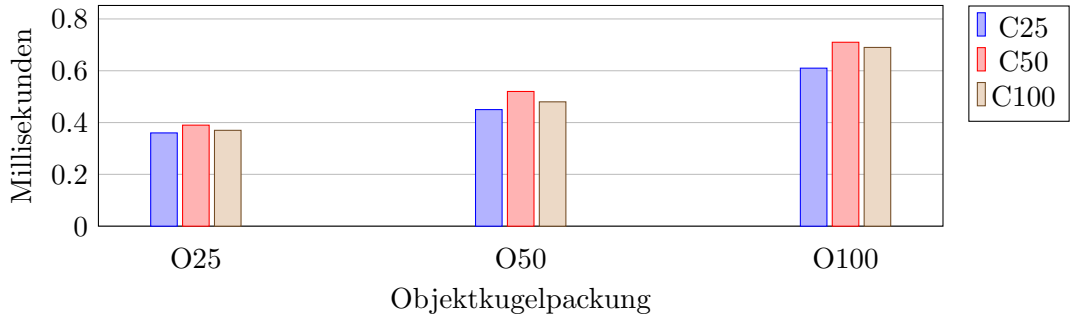


Abbildung 4.11: Durchschnittliche Kollisionserkennungszeiten zwischen Container und einem Objekten für die verschiedenen Kombinationen der Kugelpackung des Szenarios *Vase mit Herzen* für die Variante *CPUEight*.

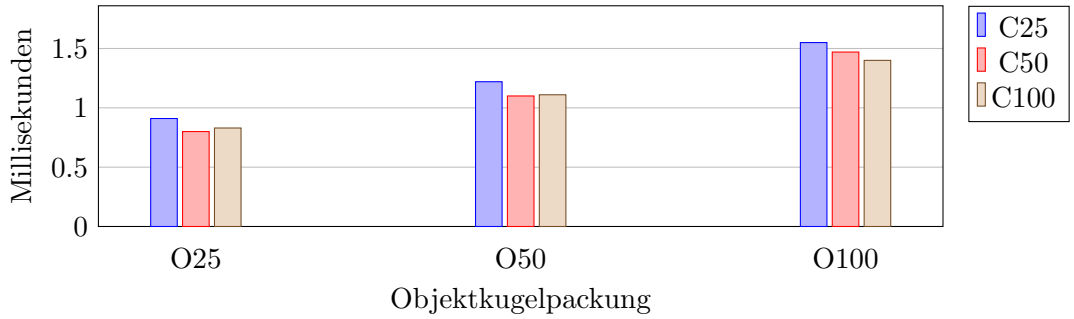


Abbildung 4.12: Durchschnittliche Kollisionserkennungszeiten zwischen Container und einem Objekten für die verschiedenen Kombinationen der Kugelpackung des Szenarios *Hand mit Früchten* für die Variante *CPUEight*.

und 25.000 Kugeln für die Teilcontainer und 40,68% für die Kombination 25.000 Kugeln für die Teilcontainer und 2.500 Kugeln für die Objekte.

Für das Szenarion *Hand mit Früchten* sind die durchschnittlichen Gesamtlaufzeiten in Abbildung 4.14 aufgezeigt. Hier zeigt sich ein ähnliches Bild wie schon im anderen Szenario. Die Packungsdichten liegen hier zwischen 39,3% für 100.000 Kugeln in den Teilcontainern und 10.000 Kugeln in den Objekten und 40,83% für 100.000 Kugeln in den Teilcontainern und 2.500 Kugeln in den Objekten.

Die Diagramme Abbildung 4.15 und Abbildung 4.16 zeigen die Zeiten für das Überprüfen eines Objektes gegen den Container der beiden Szenarien. Zu erkennen ist, dass sowohl die Anzahl der Kugeln für die Teilcontainer einen Einfluss auf die Zeit hat als auch die Anzahl der Kugeln in den Objekten. Allerdings liegen zwischen der langsamsten und der schnellsten Durchschnittszeit lediglich 0,2 Millisekunden für das Szenario *Vase mit Herzen* und 0,21 Millisekunden für das Szenario *Hand mit Früchten*.

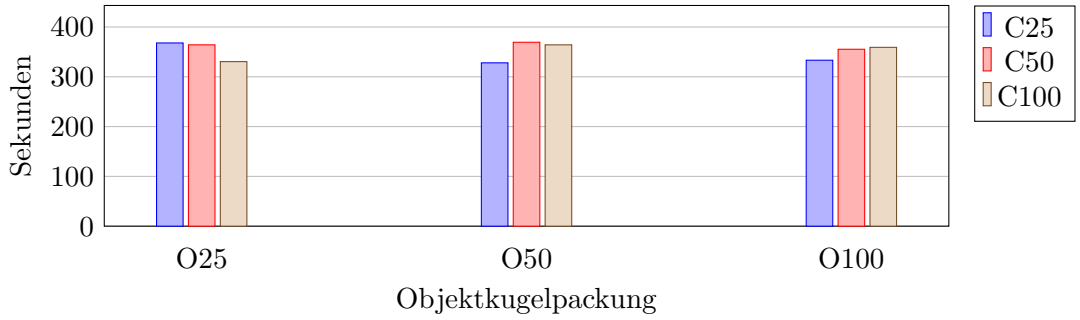


Abbildung 4.13: Durchschnittliche Gesamtlaufzeiten für die verschiedenen Kombinationen der Kugelpackung des Szenarios *Vase mit Herzen* für die Variante *GPUSpheresEight*.

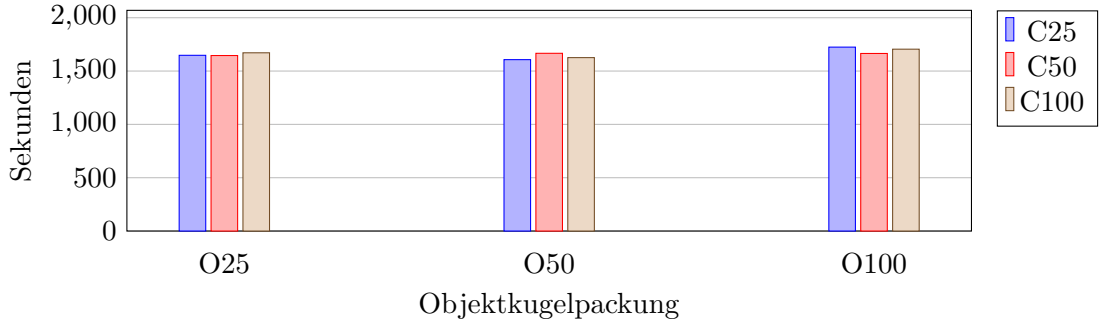


Abbildung 4.14: Durchschnittliche Gesamtlaufzeiten für die verschiedenen Kombinationen der Kugelpackung des Szenarios *Hand mit Früchten* für die Variante *GPUSpheresEight*.

4.4.3 GPUForceEight

In Abbildung 4.17 sind die Laufzeiten für die verschiedenen Konfigurationen des Szenarios *Vase mit Herzen* zu sehen. Es fällt auf, dass die verschiedenen Containerpackungen sehr ähnliche Laufzeiten unabhängig davon haben, wie viele Kugeln die Objekte repräsentieren. Ein Anstieg der Laufzeit in Abhängigkeit von der Anzahl der Kugeln für die Teilcontainer ist zu erkennen. Die Packungsdichten schwanken von 37,26% für die Kombination 50.000 Kugeln in den Teilcontainern und 5.000 Kugeln in den Objekten und 40,41% für 50.000 Kugeln in den Teilcontainern und 5.000 Kugeln in den Objekten.

Abbildung 4.18 zeigt ein ähnliches Bild für das Szenario *Hand mit Früchten*. Bei der Packungsdichte schneiden 50.000 Kugeln in den Teilcontainern und 5.000 Kugeln in den Objekten mit durchschnittlich 39,74% am schlechtesten ab. Den besten Wert erreichen 25.000 Kugeln pro Teilcontainer und 2.500 Kugeln für die Objekte mit 40,92%.

In Abbildung 4.19 und Abbildung 4.20 sind die durchschnittlichen Zeiten, wie lange es dauert, ein Objekt gegen alle acht Teilcontainer zu prüfen, für die verschiedenen Konfigurationen dargestellt. Es zeigt sich ein ähnliches Bild wie bei der Variante *GPUSpheresEight*. Jedoch ist *GPUForceEight* für jede der Konfigurationen langsamer als die andere Variante.

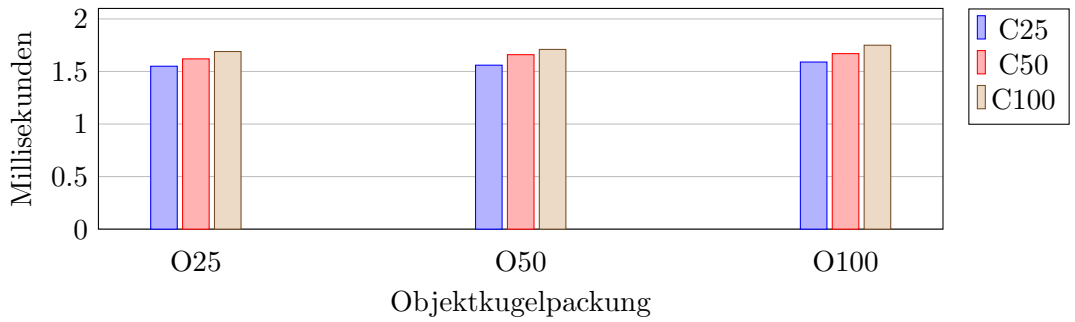


Abbildung 4.15: Durchschnittliche Kollisionserkennungszeiten zwischen Container und Objekten für die verschiedenen Kombinationen der Kugelpackung des Szenarios *Vase mit Herzen* für die Variante *GPUSpheresEight*.

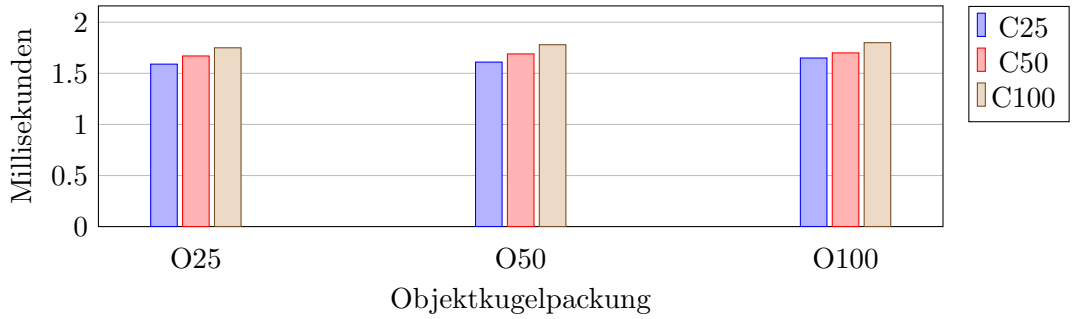


Abbildung 4.16: Durchschnittliche Kollisionserkennungszeiten zwischen Container und Objekten für die verschiedenen Kombinationen der Kugelpackung des Szenarios *Hand mit Früchten* für die Variante *GPUSpheresEight*.

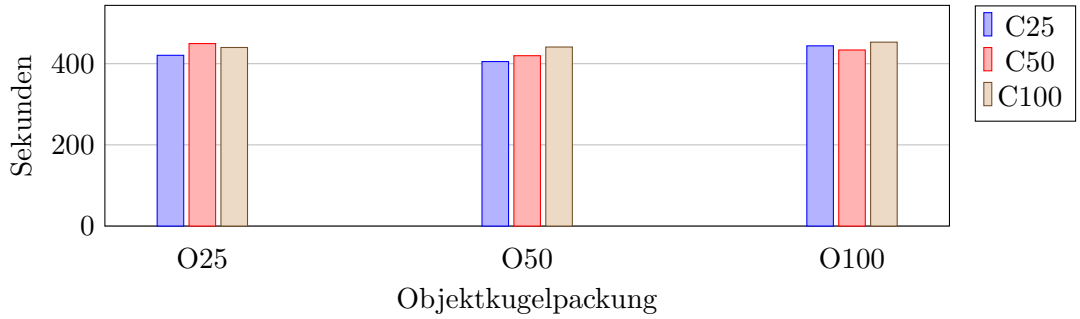


Abbildung 4.17: Durchschnittliche Gesamtlaufzeiten für die verschiedenen Kombinationen der Kugelpackung des Szenarios *Vase mit Herzen* für die Variante *GPUForceEight*.

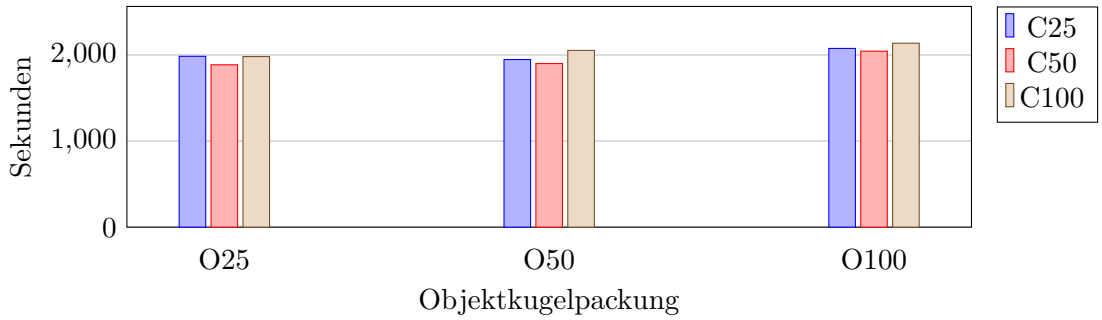


Abbildung 4.18: Durchschnittliche Gesamtlaufzeiten für die verschiedenen Kombinationen der Kugelpackung des Szenarios *Hand mit Früchten* für die Variante *GPUForceEight*.

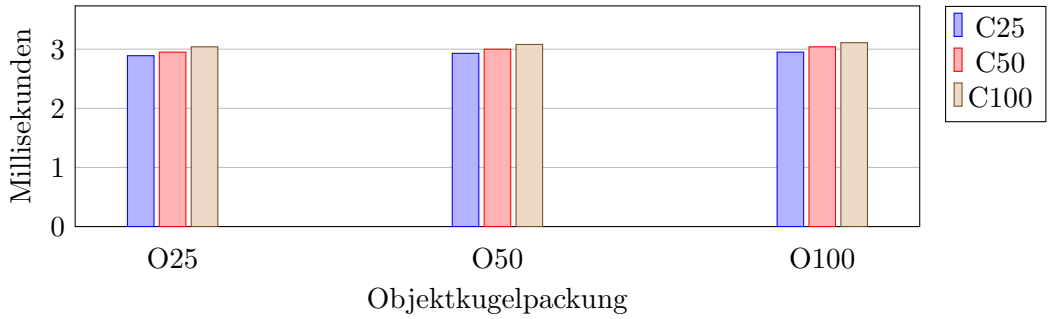


Abbildung 4.19: Durchschnittliche Kollisionserkennungszeiten zwischen Container und Objekten für die verschiedenen Kombinationen der Kugelpackung des Szenarios *Vase mit Herzen* für die Variante *GPUForceEight*.

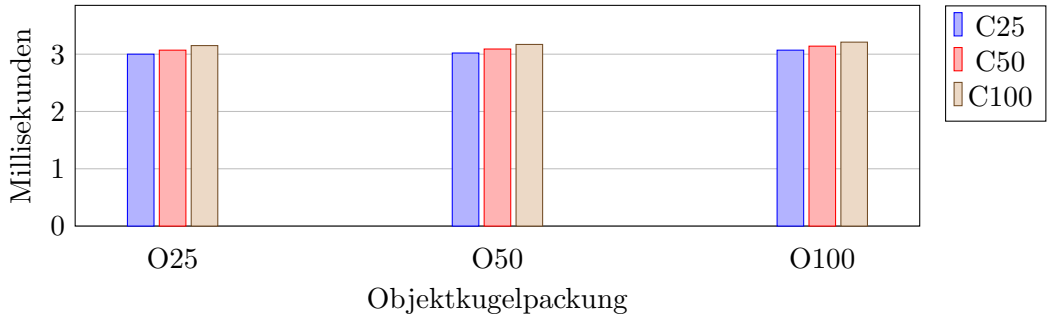


Abbildung 4.20: Durchschnittliche Kollisionserkennungszeiten zwischen Container und Objekten für die verschiedenen Kombinationen der Kugelpackung des Szenarios *Hand mit Früchten* für die Variante *GPUForceEight*.

4.5 Verbesserungsheuristiken

Die entwickelte Verbesserungsheuristik *BorderFillingAlgorithm (BFA)* wurde mit der existierenden Verbesserungsheuristik *CavityFillingGrowingAlgorithm (CFGA)* verglichen. Dazu wurden drei Konfigurationen für die beiden Szenarien evaluiert. Die Ausgangsdaten der Konfigurationen sind in Tabelle 4.6 gezeigt. Für beide Szenarien wurde jeweils eine Packung mit niedriger Packungsdichte gewählt und eine Konfiguration mit der gleichen Einstellung für das Sampling wie in der ursprünglichen Konfiguration gewählt. Darüber hinaus wurde für dieselbe Packung eine Konfiguration angelegt, so dass doppelt so viele Samplingpunkte generiert werden. Außerdem wurde für jedes Szenario eine initiale Packung mit sehr hoher Packungsdichte gewählt. Für diese Packung wurde eine Konfiguration mit denselben Einstellungen für das Sampling wie bei ihrer Erzeugung erstellt.
Als Kriterien zum Vergleich werden im Folgenden die Packungsdichte, die Oberflächendichte[4] und die Abweichung von der gewünschten Verteilung betrachtet. Die Abweichung von der gewünschten Verteilung wird dabei mittels des mittleren absoluten Fehlers (MAE) beschrieben, der sich wie folgt berechnet: $\frac{1}{n}\sum_{i=1}^{n}|\overline{x}_i - x_i|$. Dabei steht n für die Anzahl der zu packenden Objekttypen, $\overline{x}_i$ für den gewünschten prozentualen Anteil des Objekttyps i und x_i für den tatsächlich erreichten prozentualen Anteil des Objekttyps i an der erzeugten Packung.

Name	Samplingpunkt-Faktor	Initiale Packungsdichte	Initale Oberflächendichte	Initaler MAE
Konfiguration V1	1	33,88%	4,65%	3,49
Konfiguration V2	2	33,88%	4,65%	3,49
Konfiguration V3	1	41,54%	8,04%	0,96
Konfiguration H1	1	31,98%	5,89%	0,38
Konfiguration H2	2	31,98%	5,89%	0,38
Konfiguration H3	1	40,87%	9,06%	0,21

Tabelle 4.6: Beschreibung der Konfigurationen, mit denen die Verbesserungsheuristiken evaluiert wurden. Die oberen drei Zeilen sind die Konfiguration für *Vase mit Herzen*, die unteren drei für *Hand mit Früchten*. Der Samplingpunktfaktor beschreibt das Verhältnis der Samplingpunkte der neuen Konfiguration zu der Anzahl der Samplingpunkte der Konfiguration, mit der die initiale Packung erzeugt wurde.

Die beiden Abbildung 4.21 und Abbildung 4.22 stellen den Vergleich der beiden Verbesserungsheuristiken für die drei Konfigurationen pro Szenario dar. Erkennbar ist, dass *BFA* die Packungsdichte in allen sechs Szenarien erhöhen konnte. Nur für die Vase mit der hohen Packungsdichte lag die Steigerung bei etwas unter einem Prozentpunkt. In allen anderen Fällen lag sie bei mindestens vier Prozentpunkten. *CFGA* konnte im Vergleich zur ursprünglichen Konfiguration lediglich bei den Konfigurationen mit erhöhter Samp-

[4] Berechnet wie in [Meißenhelter, 2019].

lingpunktanzah deutliche Verbesserungen erzielen. Bei der Oberflächendichte zeigt sich ein ähnliches Bild. Einzig bei der Abweichung von der gewünschten Verteilung verhält sich *BFA* schlechter und sorgt für eine größeren MAE. Diese liegen darin begründet, dass kleine Objekte häufiger an den Stellen passen, wo neue Samplingpunkte erzeugt wurden. Besonders ausgeprägt ist dies im Szenario *Hand mit Früchten*, da hier der Größenunterschied unter den Objekten deutlich ausgeprägter ist.
Heraus sticht die Konfiguration V2, bei der beide Verbesserungsheuristiken eine sehr ähnliche Steigerung bei der Packungsdichte erreichen konnten. Für diese wurde die Oberflächendichte durch *CFGA* im Gegensatz zu *BFA* deutlich verbessert. Dieses Ergebnis stellt sich jedoch nicht für die analoge Konfiguration H2 dar. Hier liegt *BFA* in diesen beiden Punkten deutlich vorne. Durch die beiden Verbesserungsheuristiken optimierte Packungen sind als Abbildungen in Abschnitt A.4 zu sehen.

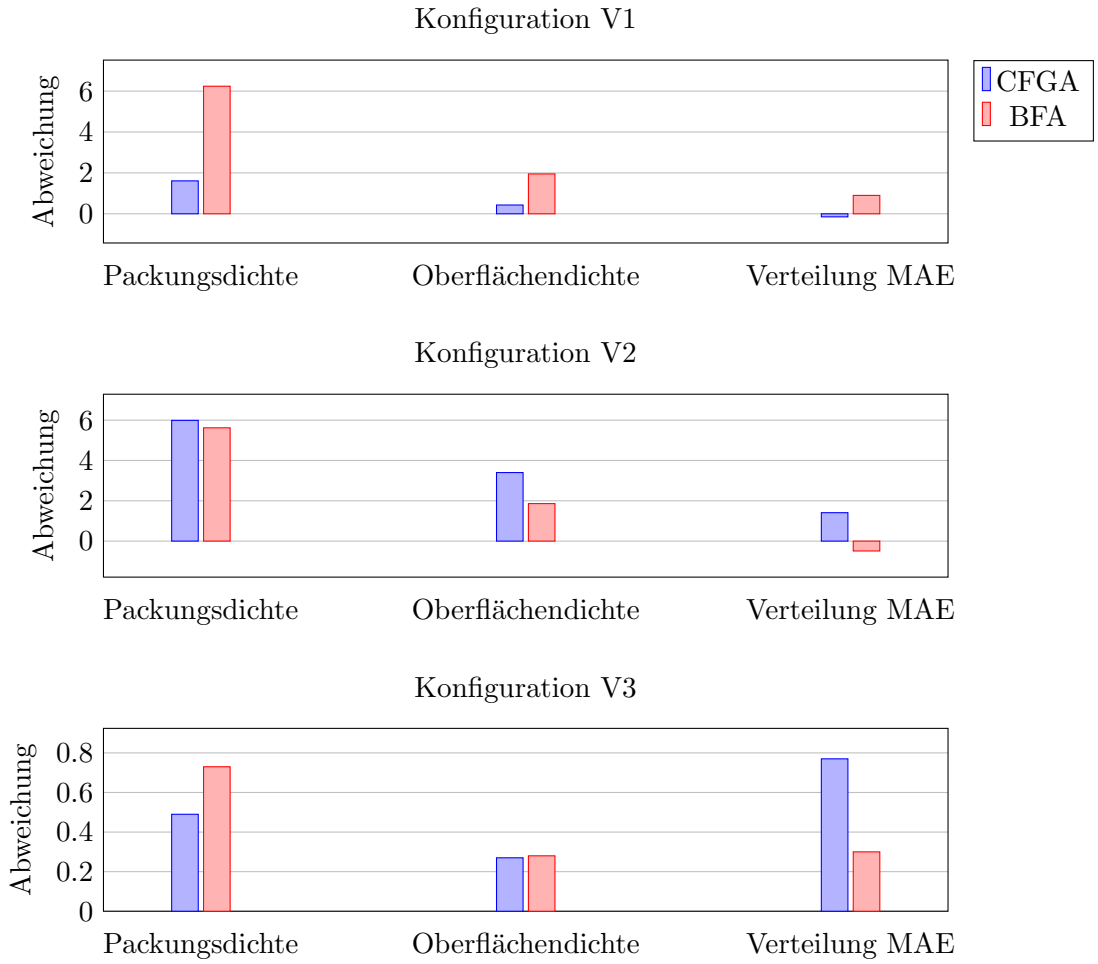


Abbildung 4.21: Vergleich der Auswirkungen der Verbesserungsheuristiken *CavityFilling-GrowingAlgorithm (CFGA)* und *BorderFillingAlgorithm (BFA)* auf die Kriterien Packungsdichte, Oberflächendichte und der mittlere absoulute Fehler zur gewünschten Verteilung für das Szenario *Vase mit Herzen*.

Betrachtet man die durchschnittlichen Laufzeiten, dargestellt in Tabelle 4.7, so fällt auf,

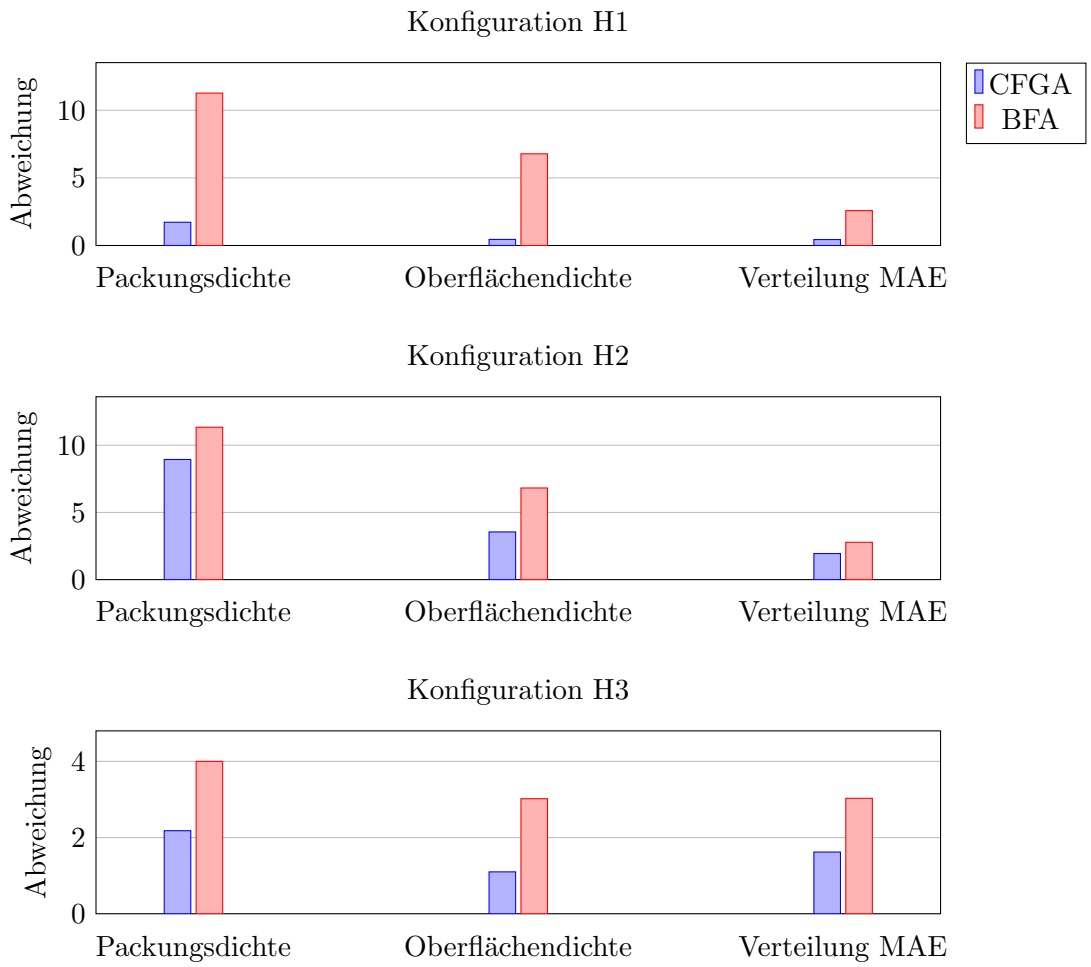


Abbildung 4.22: Vergleich der Auswirkungen der Verbesserungsheuristiken *CavityFilling-GrowingAlgorithm (CFGA)* und *BorderFillingAlgorithm (BFA)* auf die Kriterien Packungsdichte, Oberflächendichte und der mittlere absoulute Fehler zur gewünschten Verteilung für das Szenario *Hand mit Früchten*.

dass *CFGA* in allen Konfigurationen des Szenarios *Vase mit Herzen* schneller ist. Zu erklären ist dies für die Konfiguration *V1* damit, dass kaum neue Objekte platziert werden, was den größten Teil der Laufzeit ausmacht. Die Verbesserungsheuristik *CFGA* sortiert viele Samplingpunkte nach einer kurzen Überprüfung vollständig aus, was vergleichsweise wenig Zeit benötigt. Interessant ist, dass für die Konfiguration *V2* die Laufzeit besser ist als die von *BFA*. Dieser Algorithmus hat hier weniger Samplingpunkte, benötigt im Schnitt aber deutlich mehr Zeit. Dabei unterscheiden sich die Verbesserungen der Packungsdichte kaum voneinander.
Für die Konfiguration *H1* zeigt sich ein ähnliches Bild, wie für *V1*. *CFGA* ist hier deutlich schneller, da die Packung kaum verbessert wird. Interessant ist, dass hier *BFA* für die beiden übrigen Konfigurationen mehr Zeit benötigt, was damit zu erklären ist, dass er deutlich weniger Samplingpunkte erzeugt und somit nicht neue Positionen für Objekte testet.

Bei beiden Algorithmen fällt auf, dass die benötigte Zeit sowohl zwischen *V2* und *V3*, als auch zwischen *H2* und *H3* abnimmt. Dies scheint darin begründet zu liegen, dass die Packungsdichte in *V3* und *H3* bereits sehr hoch ist und nicht deutlich verbessert werden kann. Somit werden häufiger Samplingpunkte aufgrund zu geringen freien Volumens übersprungen.

Konfiguration	Durchschnittliche Laufzeit in Sekunden	
	CFGA	BFA
V1	14,7	254,1
V2	153,9	271,3
V3	117,1	228,3
H1	462,1	4073
H2	5748,2	4399,5
H3	4384,6	3625

Tabelle 4.7: Vergleich der durchschnittlichen Laufzeiten der beiden Verbesserungsheuristiken für die unterschiedlichen Konfigurationen in Sekunden.

4.6 Zusammenfassung & Diskussion

Es zeigt sich, dass eine einzelne Kugelpackung schlecht dazu geeignet ist, den Container zu repräsentieren. Dies äußert sich sowohl in langen Laufzeiten, wenn die Kollisionserkennung auf der CPU berechnet wird, als auch in stark aus dem Container herausragenden Objekten. Ähnliches ist auch zu beobachten, wenn die Kollisionserkennung zwischen Objekten und Container auf die GPU ausgelagert wird. Es zeigt sich, dass die Repräsentation des Containers über acht Kugelpackungen, von denen jede nur einen Teil repräsentiert, deutlich bessere Ergebnisse produziert. Dies ist zum einen an der verkürzten Laufzeit zu erkennen als auch an der niedrigeren Zahl von Objekten, die in der finalen Packung noch mit dem Container kollidieren, und dem Grad des Herausragens dieser Objekte. Die Packungsdichte wird dabei für das Szenario *Vase mit Herzen* nicht sehr stark verringert, so dass ausgeschlossen ist, dass die Beschleunigung hier durch weniger gepackte Objekte zustanden kommt. Das bessere Abschneiden ist in diesem Fall durch die bessere Repräsentation des Containers zu begründen.
Im Szenario *Hand mit Früchten* werden die Packungsdichten durch die Aufteilung des Containers zum Teil deutlich schlechter. Dies kann zwei Gründe haben. Zum einen werden weniger Samplingpunkte bei denselben Einstellungen für das Sampling generiert, es werden also initial bereits weniger Objekte für die Packung in Betracht gezogen, zum anderen sieht man, dass die Objekte bei der Repräsentation des Containers über eine Kugelpackung deutlich stärker aus dem Container herausragen als bei acht Kugelpackungen. Dadurch werden unter Umständen große Objekte am Containerrand in den Varianten *GT*, *GPUSpheres* und *GPUForce* nicht so häufig verworfen wie bei *CPUEight*. Es zeigte sich

außerdem, dass das Volumen der einzelnen Objekte einen starken Einfluss auf die Packungsdichte hat. Dies äußerte sich in Packungen, die deutlich weniger Objekte als andere Packungen enthielten, deren Packungsdichte jedoch vergleichbar war.
Das initial mehr Samplingpunkte aussortiert werden, ist ein Hinweis darauf, dass der Container über acht Kugelpackungen besser repräsentiert wird als über eine, weil die Kugelpackung am Rand des Containers dann weniger Lücken hat. Dies schlägt sich auch in der Anzahl und dem Grad des Herausragens der Objekte nieder.
Jedoch muss ein Teil des Geschwindigkeitsgewinns im Szenario *Hand mit Früchten* diesem Umstand zugerechnet werden, da für dieselbe Konfiguration insgesamt weniger Objekte überprüft werden müssen. Für das Szenario *Vase mit Herzen* unterscheidet sich die Anzahl der erzeugten Samplingpunkte kaum. Somit festigt sich auch das Ergebnis, dass die Containerform einen starken Einfluss auf die Wirkung der verschiedenen Ansätze hat.
Wird der Container durch das tatsächliche Mesh repräsentiert und die Kollisionserkennung nutzt *DOP-Trees*, so zeigt sich, dass hier die Geschwindigkeit von der Komplexität des Containers abhängt. Bei der Vase als weniger komplexem Container waren beide Varianten am schnellsten, was die sowohl Gesamlaufzeit angeht, als auch die Überprüfung auf Kollisionen aller Objekte untereinander sowie mit dem Container. Besonders hervor sticht die Zeit, die benötigt wird, um in diesem Szenario ein Objekt gegen den Container zu prüfen. Für die Hand sieht das Ganze anders aus. Hier zeigt sich, dass diese als Container deutlich komplexer ist. Dies führt zu deutlich längeren Laufzeiten, wenn das Mesh als ganzes für die Kollisionserkennung genutzt wird. Wird das Mesh geteilt, so sinkt die Laufzeit deutlich. Im Vergleich mit den acht Kugelpackungen zeigt sich allerdings, dass hier die Kollisionserkennung mittels *SphereTree* minimal schneller ist. Allerdings muss auch gesagt werden, dass die beiden Varianten für beide Szenarien die schlechtesten Packungsdichten erzeugen, dabei jedoch am besten abschneiden bezüglich des Herausragens der Objekte aus dem Container.
Bei den Ansätzen, die die Kollisionserkennung zwischen Objekten und Container auf die GPU verlagern, überzeugt *GPUSpheres*. Dieser Ansatz ist, bezogen auf die Gesamtlaufzeit, schneller als *GPUForce* und produziert Packungen mit gleicher oder besserer Packungsdichte. Dies ist darauf zurückzuführen, dass die Zeiten für einen check()-Aufruf auf der Pipeline schneller sind. Außerdem werden die Kollisionschecks zwischen Objekt und Container schneller ausgeführt. Dies bestätigt die getroffene Annahme, dass die *atomicAdds* in den Funktionen von *GPUForce* einen Einfluss auf die Laufzeit haben. Dieser ist größer als das Abarbeiten der Rückgabewerte von *GPUSpheres* mittels Schleife auf der CPU. Ein Grund hierfür ist, dass die tatsächliche Anzahl der kollidierenden Kugelpaare häufig im niedrigen zweistelligen Bereich liegt.
Die GPU-Ansätze profitieren nicht von der Aufteilung des Containers, wenn nur die Laufzeiten verglichen werden. Sie profitieren jedoch bezüglich des Herausragens der Objekte aus dem Container. Da der Container über die Teilcontainer besser dargestellt wird, ragen die gepackten Objekte hier, wie auch bei der Variante *CPUEight*, nicht mehr so stark

heraus.
Einen sehr kleinen Einfluss auf die Laufzeiten der Kollisionserkennung zwischen Objekt und Container beider GPU-Varianten haben die Anzahl der Kugeln in den Kugelpackungen. Bei allen Varianten dauert die Kollisionserkennung zwischen den Objekten im Schnitt gleich lang. Unterschiede sind hier nur für die beiden Szenarien festzustellen.
Die Varianten, bei denen die Kollisionserkennung schneller ist, generieren Packungen mit höheren Packungsdichten. Dies liegt daran, dass Objekte verworfen werden, wenn durch ihre Platzierung entstandende Kollisionen nicht innerhalb einer bestimmten Zeit aufgelöst werden können. Hier können schnellere Varianten also mehr Lösungen auf ihre Gültigkeit überprüfen. Es fällt auf, dass in den Varianten, in denen der Container durch einen *DOP-Tree* repräsentiert wird, die Packungsdichten allerdings niedriger liegen als bei den Varianten, die den Container mittels *IST* darstellen. Dies liegt daran, dass auch Objekte in der Mitte des Containers häufiger verworfen werden, wenn ihr Platzieren dazu führt, dass andere Objekte mit dem Container kollidieren. Da der Container deutlich genauer dargestellt ist als mittels *IST*, werden hier mehr check()-Aufrufe benötigt, um diese Kollisionen aufzulösen. Dadurch wird der Geschwindigkeitsvorteil, den diese Varianten vor allem im Szenario *Vase mit Herzen* haben, aufgehoben. Für die Variante *CPUDopOne* im Szenario *Hand mit Früchten* kommt neben der genaueren Darstellung des Containers hinzu, dass das Erkennen von Kollisionen deutlich länger dauert. Daher werden hier deutlich mehr Objekte verworfen, da das Zeitlimit für die Auflösung der Kollisionen häufiger nicht eingehalten werden kann.
Zusammenfassend lässt sich feststellen, dass die Aufteilung des Containers unabhängig von der Repräsentationsform die besten Ergebnisse liefert. Für sehr genaue Packungen, bei denen die Packungsdichte eine zweitrangige Rolle spielt, ist der *DOP-Tree* besser geeignet, bei Packungen, in denen die Packungsdichte wichtig ist und kleine Überschneidungen zwischen Container und Objekte tollerabel sind, ist *CPUEight* die bessere Variante.

Bei den Verbesserungsheuristiken zeigt sich, dass vor allem *CFGA* von der Anzhal der Samplingpunkte abhängt. Hinzu kommt, dass eine Abhängigkeit von der initialen Packung besteht. Wurden die Objekte hier bewegt, befinden sich also nicht auf dem Samplingpunkt, auf dem sie ursprünglich platziert wurden, so verbessert auch *CFGA* die Packung merklich. *BFA* hat hier den Vorteil, dass es weniger von den ursprünglichen Samplingpunkten abhängig ist, da neue Samplingpunkte generiert werden. Da diese Punkte ausschließlich am Rand liegen, erhöht es die Oberflächendichte deutlich stärker als *CFGA*. Jedoch werden keine neue Objekte in Lücken platziert, die im Inneren des Containers liegen. Es ist also davon auszugehen, dass hier Lücken bestehen bleiben, die noch geschlossen werden können. *BFA* verursacht außerdem durch die Auswahl der Objekte eine deutlich stärkere Abweichung von der gewünschten Verteilung als *CFGA*. Bei den Laufzeiten ist *BFA* schneller, wobei dies daran liegt, dass die Samplingpunktanzahl durch das Filtern geringer ist. Eine weniger strenge Filterung würde hier zu längeren Laufzeiten führen. Auffällig ist außerdem, dass *CFGA* bei Konfigurationen, bei denen die urpsrüngliche Samplingpunkte

den neuen Samplingpunkten entsprechen, deutlich schneller ist. Dies liegt daran, dass auf fast keinem Punkt neue Objekte platziert werden.

Kapitel 5

Fazit & Ausblick

Ziel der Arbeit war es, das Programm *AutoPacking* in den Bereichen Laufzeit, Kollisionserkennung mit dem Container, sowie für Peter Coffin die Oberflächendichte zu erhöhen.
Für die Verbesserung der Laufzeit wurden verschiedene Ansätze entwickelt, die alle darauf abzielen, die Kollisionserkennung von Objekten mit dem Container zu beschleunigen, da dies den größten Anteil an der Gesamtlaufzeit einnimmt. Hierzu wurde der Container nicht durch eine einzelne sondern durch acht Kugelpackungen repräsentiert, die jeweils einen Teil des Containers repräsentieren. Außerdem wurden zwei Ansätze entwickelt, die alle Kugeln der Kugelpackung des Containers gegen den *IST* eines Objektes prüfen. Festzustellen ist, dass alle Varianten deutlich schneller sind als der ursprüngliche Ansatz. Rein laufzeittechnisch schneiden die beiden Varianten auf der GPU nochmals deutlich besser ab als die Aufteilung des Containers und Überprüfung auf der CPU.
Um die Kollisionserkennung mit dem Container genauer zu gestalten, wurde implementiert, dass neben der Repräsentation über einen *IST* auch ein *DopTree* verwendet werden kann. Hier zeigt sich, dass dies eine deutlich bessere Repränsentation des Containers darstellt und das Ergebnis mit Blick auf noch mit dem Container kollidierende Objekte in der finalen Packung besser ist. Interessant ist, dass für einfache Container die Repräsentation mittels *DopTree* schnellere Ergebnisse liefert als die mittels *IST*. Für komplexere Container gilt dies nicht mehr, aber auch hier führt die Aufteilung des Containers zu einem Geschwindigkeitsgewinn. Es hat sich außerdem gezeigt, dass acht Kugelpackungen für den Container ebenfalls deutlich bessere Ergebnisse liefern, was das Herausragen der Objekte aus dem Container betrifft.
Ein Nachteil der *DopTree* Methoden ist jedoch, dass in manchen Fällen Packungen erzeugt werden, bei denen in einer nachgelagerten Prüfung noch Kollisionen zwischen einzelnen Objekten und dem Container festgestellt werden. Es konnte abschließend nicht ergründet werden, was zu diesem Verhalten führt. Daher eignet sich der Einsatz des *DopTrees* nicht für die Verbesserungsheuristiken, da in einem solchen Fall das Platzieren neuer Objekte verhindert wird.
Als Ergebnis zeigt sich, dass eine Aufteilung des Containers sowohl für die Laufzeit als auch für die Qualität der Packung die meisten Vorteile bietet, wenn *ISTs* für den Container verwendet werden.

Um die Oberflächendichte einer bestehenden Packung zu erhöhen, wurde die Verbesserungsheuristik *BorderFillingAlgorithm* implementiert. Diese erzeugt neue Samplingpunkte entlang des Randes des Containers. Auf diesen werden dann ähnlich wie schon beim *GrowingSeedsAlgorithm* neue Objekte platziert. Die Evalution zeigt, dass dieser Algorithmus deutlich unabhängiger von den Einstellungen zum Sampling des Containers ist, als der *CavityFillingGrowingAlogrithm* und bessere Ergebnisse liefert, was Packungsdichte und Oberflächendichte angeht. Ein Nachteil ist jedoch, dass die Abweichung von der gewünschten Verteilung der Objekte deutlich stärker ausfällt.

Verbesserungsmöglichkeiten ergeben sich vor allem bei dem Einsatz der beiden GPU-Ansätze. Diese profitieren mit Hinblick auf die Laufzeit nicht von der Aufteilung des Containers, allerdings wird in der akutellen Implementierung ein Objekt sequentiell gegen alle Kugelpackungen der Teilcontainer geprüft. Hier wäre es interessant zu sehen, wie sich das Laufzeitverhalten ändert, wenn nicht auf das Ergebnis einer Kollisionserkennung zwischen Objekt und Teilcontainer gewartet würde, sondern ein Objekt parallel gegen alle Teilcontainer geprüft wird. Dazu müsste *CollDet* so angepasst werden, dass es möglich ist, die Ergebnisse einer Kollisionserkennung zwischen zwei Kollisionsobjekten asynchron erhalten zu können.
Weitere interessante Frage sind, welche Aufteilung des Containers am besten ist und ob die Form der Teilcontainer eine Rolle spielt. In der vorliegenden Arbeit wurden acht quaderförmige Teilcontainer verwendet, wobei der Raum um den Container gleichmäßig aufgeteilt wurde. Kurze Tests haben ergaben, dass weniger Teilcontainer keinen nennenswerten Geschwindigkeitsgewinn liefern. Interessant wäre nun, ob andere Formen eventuell zu besseren Ergebnissen führen und ob eine unregelmäßige Aufteilung des Containers eventuell besser geeignet ist.

Literaturverzeichnis

[Araújo et al., 2019] Araújo, L. J., Özcan, E., Atkin, J. A., and Baumers, M. (2019). Analysis of irregular three-dimensional packing problems in additive manufacturing: a new taxonomy and dataset. *International Journal of Production Research*, 57(18):5920–5934.

[Bennell and Oliveira, 2008] Bennell, J. A. and Oliveira, J. F. (2008). The geometry of nesting problems: A tutorial. *European Journal of Operational Research*, 184(2):397–415.

[Bennell and Oliveira, 2009] Bennell, J. A. and Oliveira, J. F. (2009). A tutorial in irregular shape packing problems. *Journal of the Operational Research Society*, 60(sup1):S93–S105.

[Crama et al., 1995] Crama, Y., Kolen, A. W. J., and Pesch, E. J. (1995). Local search in combinatorial optimization. In Braspenning, P. J., Thuijsman, F., and Weijters, A. J. M. M., editors, *Artificial Neural Networks: An Introduction to ANN Theory and Practice*, pages 157–174, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Dowsland and Dowsland, 1992] Dowsland, K. A. and Dowsland, W. B. (1992). Packing problems. *European Journal of Operational Research*, 56(1):2–14.

[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. *Københavns UniversitetKøbenhavns Universitet, Det Naturvidenskabelige FakultetFaculty 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.

[Eisenbrand et al., 2003] Eisenbrand, F., Funke, S., Reichel, J., and Schömer, E. (2003). Packing a Trunk. In Di Battista, G. and Zwick, U., editors, *Algorithms - ESA 2003*, pages 618–629, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Fowler et al., 1981] Fowler, R. J., Paterson, M. S., and Tanimoto, S. L. (1981). Optimal packing and covering in the plane are NP-complete. *Information Processing Letters*, 12(3):133–137.

[Garey and Johnson, 1990] Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA.

[Gogate and Pande, 2008] Gogate, A. S. and Pande, S. S. (2008). Intelligent layout planning for rapid prototyping. *International Journal of Production Research*, 46(20):5607–5631.

[Kaluschke et al., 2020] Kaluschke, M., Weller, R., Hammer, N., Pelliccia, L., Lorenz, M., and Zachmann, G. (2020). Realistic Haptic Feedback for Material Removal in Medical Simulations. `https://cgvr.cs.uni-bremen.de/papers/haptics_symposium20/paper.pdf`, Abgerufen am: 23.09.2021.

[Karras, 2012a] Karras, T. (2012a). Thinking Parallel, Part I: Collision Detection on the GPU. `https://developer.nvidia.com/blog/thinking-parallel-part-i-collision-detection-gpu/`, Abgerufen am: 07.10.2021.

[Karras, 2012b] Karras, T. (2012b). Thinking Parallel, Part II: Tree Traversal on the GPU. `https://developer.nvidia.com/blog/thinking-parallel-part-ii-tree-traversal-gpu/`, Abgerufen am: 07.10.2021.

[Karras, 2012c] Karras, T. (2012c). Thinking Parallel, Part III: Tree Construction on the GPU. `https://developer.nvidia.com/blog/thinking-parallel-part-iii-tree-construction-gpu/`, Abgerufen am: 07.10.2021.

[Kay and Kajiya, 1986] Kay, T. L. and Kajiya, J. T. (1986). Ray Tracing Complex Scenes. *SIGGRAPH Comput. Graph.*, 20(4):269–278.

[Lutz, 2021] Lutz, P. D. C. (2021). Theoretische Informatik 1 + 2; Skript zu den Lehrveranstaltungen. `http://www.informatik.uni-bremen.de/tdki/lehre/ws21/theoinf/TheoInfSkript.pdf`, Abgerufen am: 11.10.2021.

[Ma et al., 2018] Ma, Y., Chen, Z., Hu, W., and Wang, W. (2018). Packing Irregular Objects in 3D Space via Hybrid Optimization. *Computer Graphics Forum*, 37(5):49–59.

[Meißenhelter, 2019] Meißenhelter, H. (2019). Auto Packing für beliebige 3D Objekte und Container. Master's thesis, Universität Bremen. `https://cgvr.cs.uni-bremen.de/theses/finishedtheses/3d-autopacking/Thesis_Meissenhelter.pdf`, Abgerufen am: 05.10.2021.

[Nurmela and Östergård, 1997] Nurmela, K. J. and Östergård, P. R. J. (1997). Packing up to 50 Equal Circles in a Square. *Discrete & Computational Geometry*, 18:111–120. `https://link.springer.com/article/10.1007/PL00009306`.

[of Mathematics, 2011] of Mathematics, E. (2011). Orthogonalization. `http://encyclopediaofmath.org/index.php?title=Orthogonalization&oldid=17050`, Abgerufen am: 13.11.2021.

[Sobottka and Weber, 2005] Sobottka, G. and Weber, A. (2005). Efficient Bounding Volume Hierarchies for Hair Simulation. In *The 2nd Workshop in Virtual Reality Interactions and Physical Simulations (VRIPHYS '05)*.

[Stewart, 2009] Stewart, I. (2009). *Neue Wunder aus der Welt der Mathematik*. Piper, München Zürich. Aus dem Englischen von Helmut Reuter.

[Verkhoturov et al., 2016] Verkhoturov, M., Petunin, A., Verkhoturova, G., Danilov, K., and Kurennov, D. (2016). The 3D Object Packing Problem into a Parallelepiped Container Based on Discrete-Logical Representation. *IFAC-PapersOnLine*, 49(12):1–5. 8th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2016.

[Vince, 2011] Vince, J. (2011). *Rotation Transforms for Computer Graphics*. Springer-Verlag London, London.

[Weller and Zachmann, 2009] Weller, R. and Zachmann, G. (2009). Inner sphere trees for proximity and penetration queries. `https://cgvr.cs.uni-bremen.de/papers/rss09/ist-rss.pdf`, Abgerufen: 29.09.2021.

[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.

[Wu et al., 2014] Wu, S., Kay, M., King, R., Vila-parrish, A., and Warsing, D. (2014). Multi-objective optimization of 3D packing problem in additive manufacturing. *IIE Annual Conference and Expo 2014*, pages 1485–1494.

[Zachmann, 1998] Zachmann, G. (1998). Rapid collision detection by dynamically aligned dop-trees. In *Proceedings. IEEE 1998 Virtual Reality Annual International Symposium (Cat. No.98CB36180)*, pages 90–97.

Anhang A

A.1 Diagramme Kollisionserkennung Objekt Container

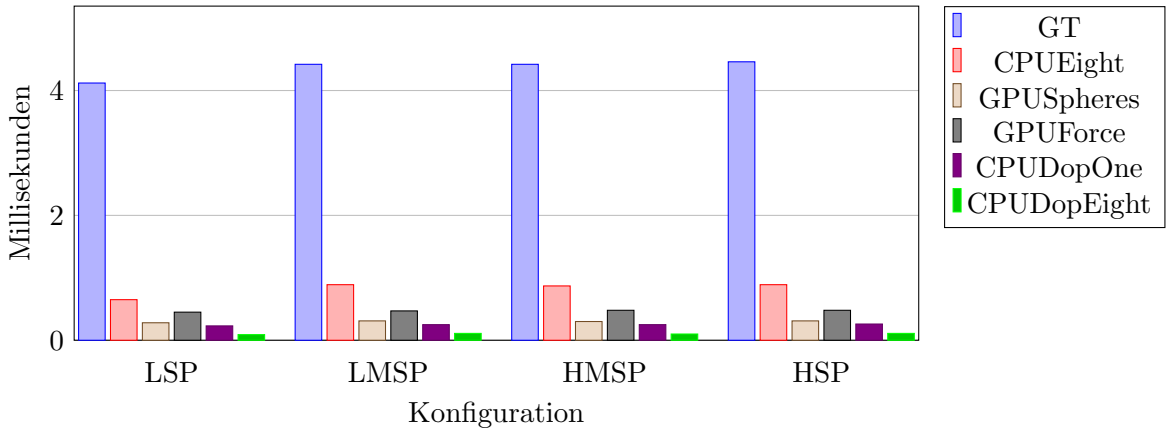


Abbildung A.1: Durchschnittszeiten für die Prüfung auf eine Kollision eines Objektes mit dem Container für das Szenario *Vase mit Herzen*.

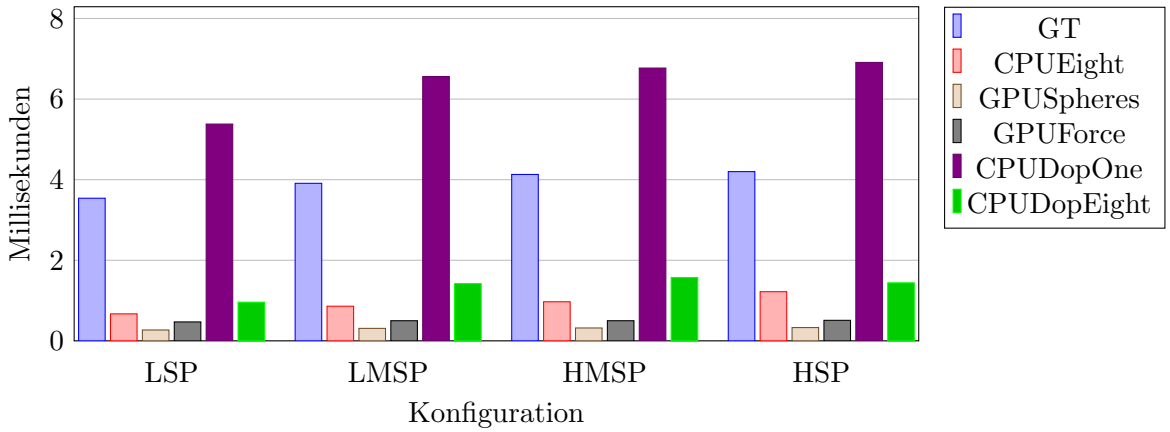


Abbildung A.2: Durchschnittszeiten für die Prüfung auf eine Kollision einses Objektes mit dem Container für das Szenario *Hand mit Früchten*.

A.2 Programmstart und Konfigurationsdateien

Im Folgenden wird beschrieben, wie das Programm gestartet werden kann und wie die Konfigurationsdateien aufgebaut sind.
Das Programm wird über die Kommandozeile gestartet.

```
AutoPacking -c ConfigurationName [-r] [-v PackingName]
```

Der Parameter `-c ConfigurationName` dient dazu, eine Konfiguration für eine Packung zu laden. Mittels `-r` wird gesteuert, ob während des Packungsprozesses ein Fenster gerendert wird oder nicht. Wird der Parameter weggelassen, so wird kein Fenster angezeigt. Über den Parameter `-v PackingName` kann eine Packung zum Betrachten geladen werden. Hierfür muss dann der Parameter `-r` zwingend gesetzt werden.

Im Folgenden wird der Aufbau der Konfigurationsdateien beschrieben.
Die Überschriften für die einzelnen Kategorien dürfen nicht verändert werden. Unter dem Punkt `[Construction Heuristic]` wird der zu verwendende Algorithmus für die initiale Packung angegeben. Darüber hinaus kann optional ein Dateinname angegeben werden, unter dem die erzeugte Packung gespeichert wird oder die bereits eine zu ladende Packung enthält. Als Algorithmen stehen zur Auswahl:

- GrowingSeedsAlgorithm
- DistanceAlgorithm
- HybridAlgorithm
- PlacementAlgorithm

Unter `[Improvement Heuristic]` kann optional eine Verbesserungsheuristik gewählt werden. Auch hier kann ein Dateiname angegeben werden, unter dem die erzeugte Packung gespeichert wird. Hier stehen als Algorithmen zur Auswahl:

- CavityFillingGrowingAlgorithm
- ShuffleAlgorithm
- BorderFillingAlgorithm

Die Verbesserungen aus Kapitel 3 sind derzeit nur für *GrowingSeedsAlgorithm* implementiert. Aussschließlich *CavityFillingGrowingAlgorithm* und *BorderFillingAlgorithm* funktionieren mit den aufgeteilten Kugelpackungen für den Container.
Sowohl für den Container unter `[Container]` als auch für die Objekte unter `[Objects]` besteht die Möglichkeit, Farben auszuwählen. Diese können spezifisch über RGB-Werte mit einem zusätzlichen Alpha-Wert im Bereich `[0,1]` angegeben werden, oder die Farbe wird zufällig, bei der Verwendung von `rand`, vom Programm generiert. Für die Objekte

kann in einem zusätzlichen Parameter noch ein Name für die zu verwendende Kugelpackung angegeben werden. Dazu muss das Programm jedoch speziell kompiliert werden. Ansonsten muss die Kugelpackung den gleichen Namen haben wie die .obj-Datei, die das Mesh des Objektes enthält. Unter `[Spheres]` werden die für den Container zu ladenden Kugelpackungen angegeben. Unter `[Dops]` können die zu verwendenden Meshes für den Container angegeben werden, falls die Kollisionserkennung zwischen Objekten und Container über einen *DOP-Tree* realisiert werden soll. Dabei müssen dann unter `[Spheres]` genauso viele Kugelpackungen angegeben werden. Dieser Eintrag ist optional.

```
[Name]
Name
[Construction Heuristic]
PackingAlgorithm [packingFile]
[Improvement Heuristic]
[ImprovementAlgorithm] [packingFile]
[Grid]
i i i //Gridsize als Integer
[SurfaceDensity Epsilon]
f //Epsilon für die Berechnung der Oberflächendichte als float
[Hopkins Statistic]
f i //f = [0,1]; i = Anzahl der Wiederholungen
[Preprocessing]
pointFile (smalles | average | REAL) StepSize NumberOfThreads
[ModelPath]
path //Pfad zum Verzeichnis mit den Modelldaten
[Container]
ContainerName ((f f f) | rand) f
[Spheres]
SpherePackingNames
[Dops]
MeshNames
[Objects]
ObjectName [SpherPackingName] f ((f f f) | rand) f
...
```

A.3 Packungsbilder

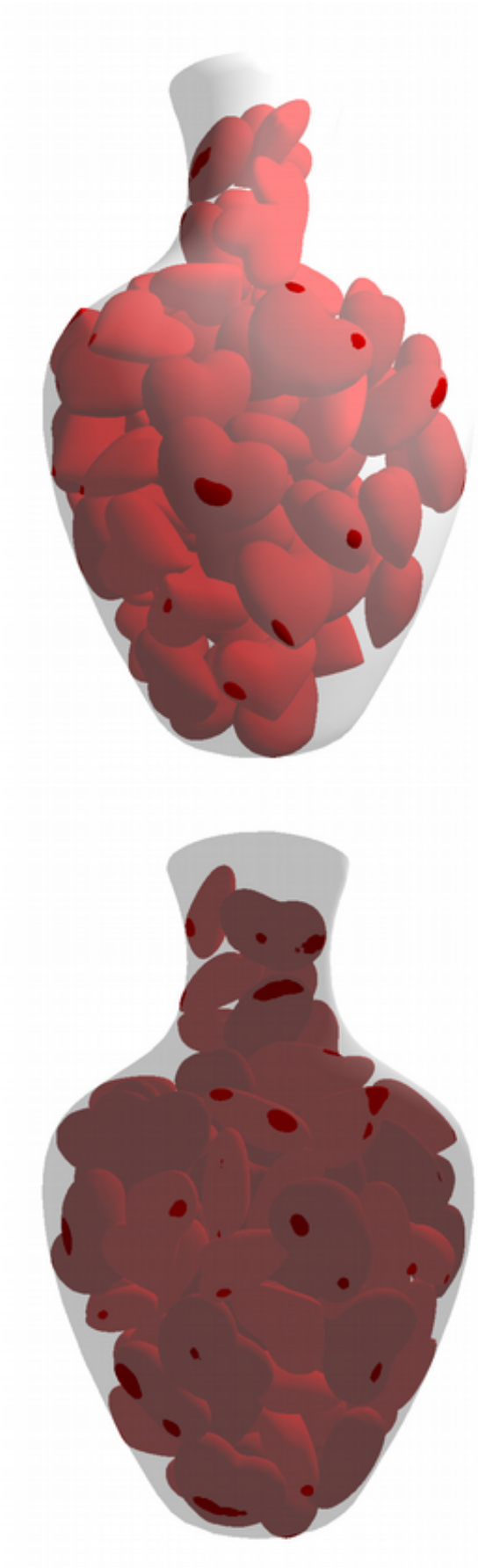


Abbildung A.3: GT Vase

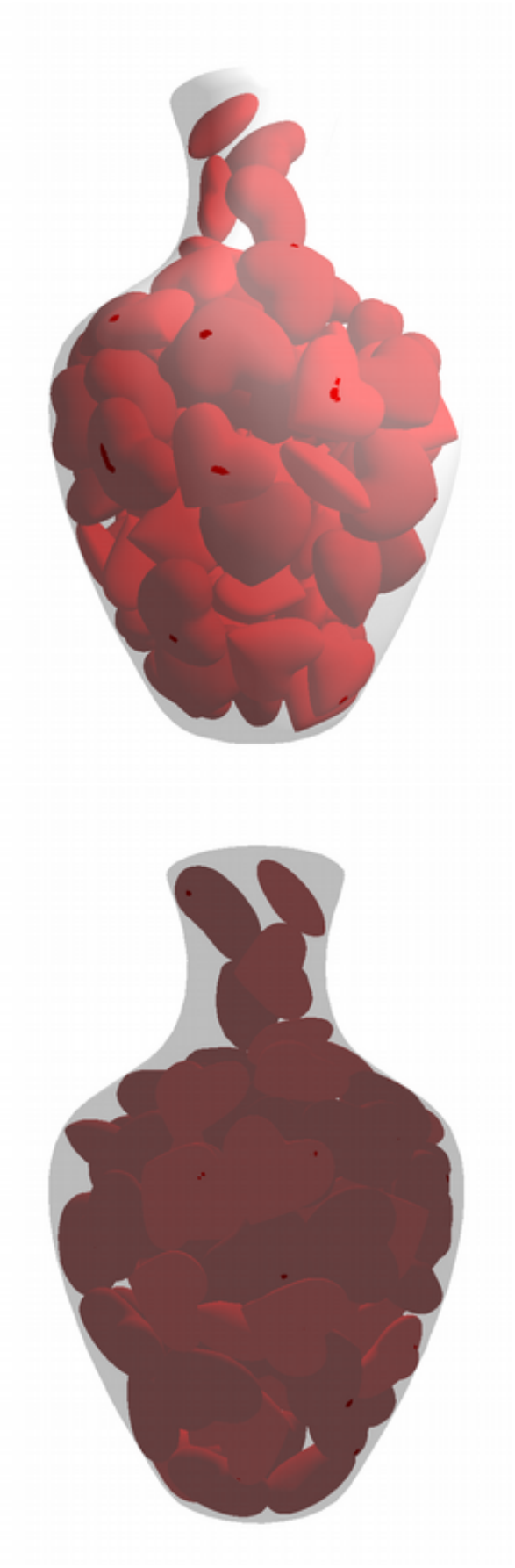


Abbildung A.4: CPUEight Vase

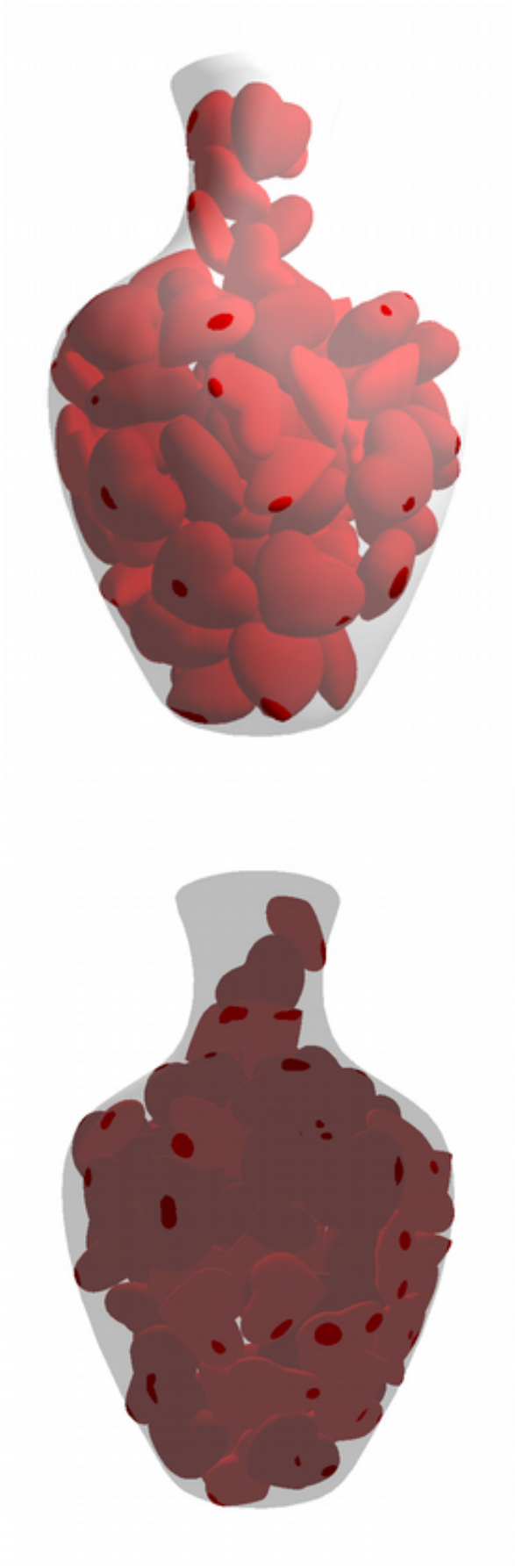


Abbildung A.5: GPUSpheres Vase

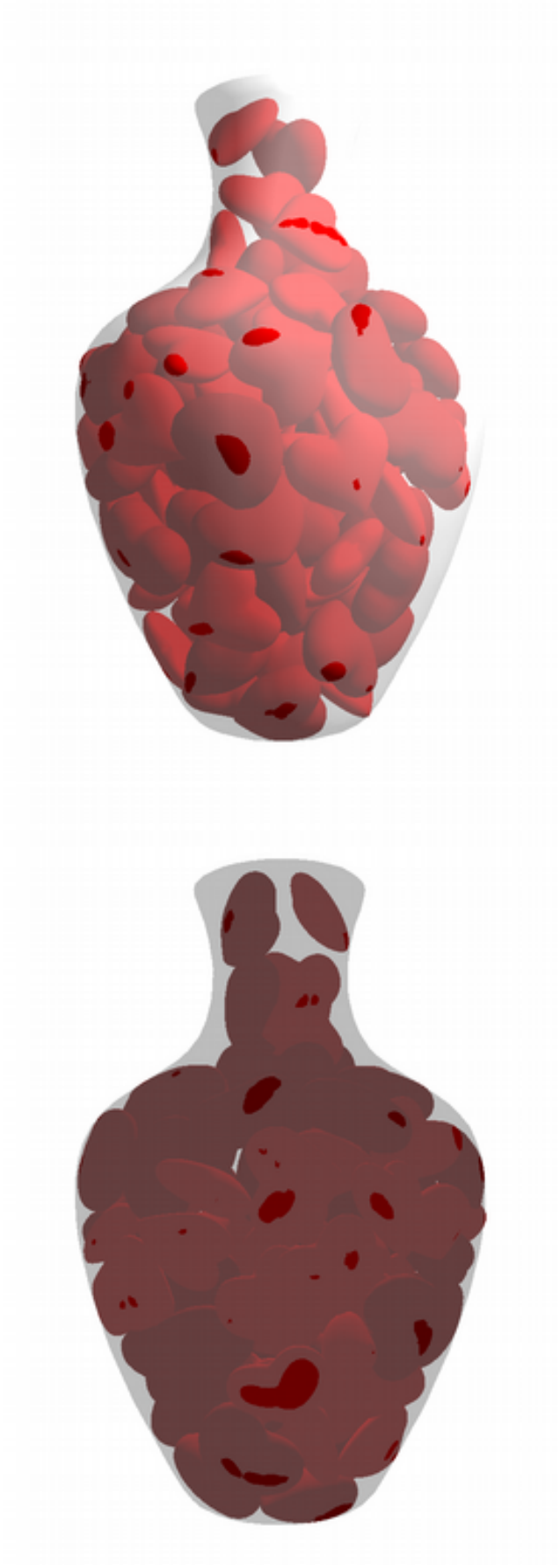


Abbildung A.6: GPUForce Vase

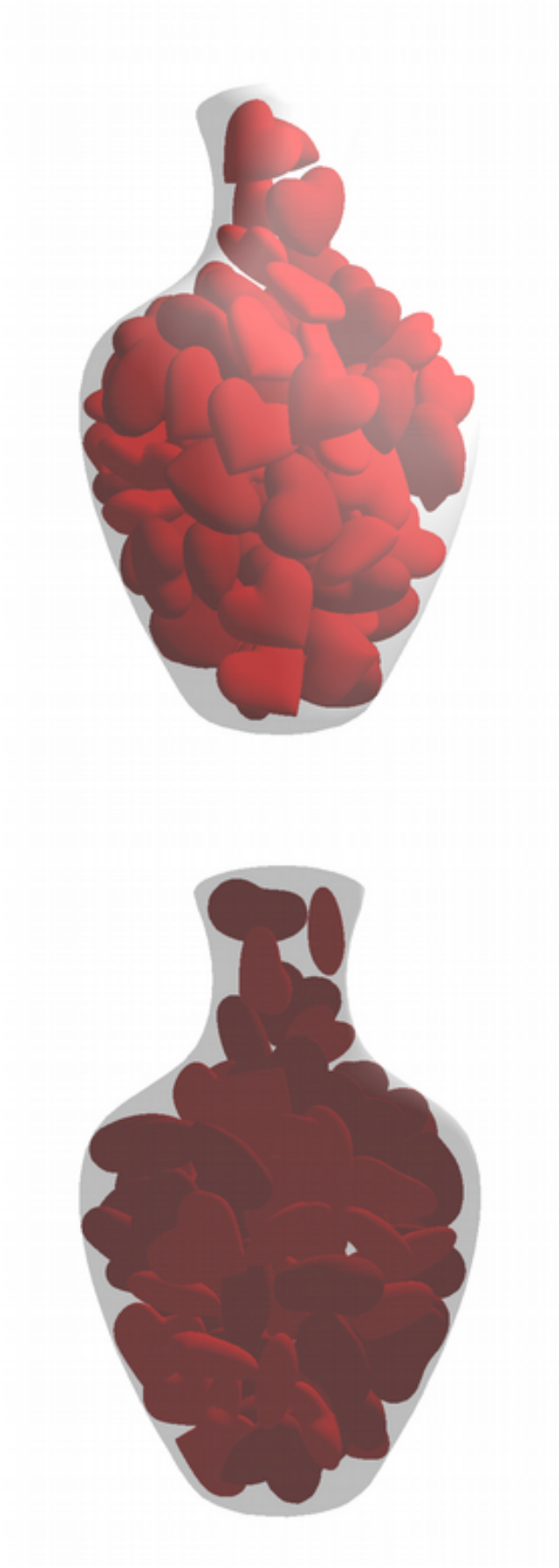


Abbildung A.7: DOP Vase

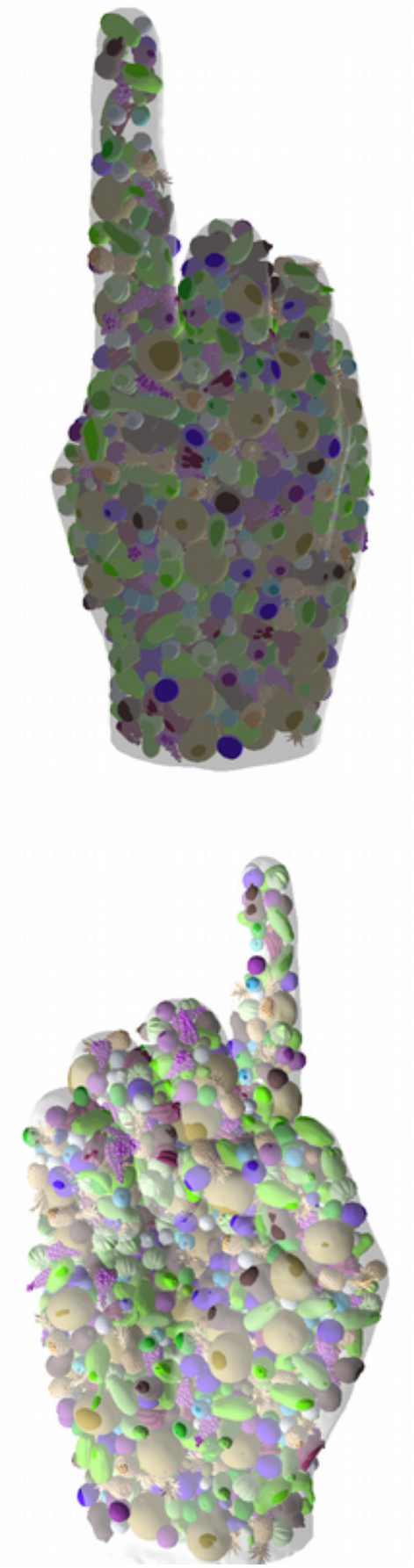


Abbildung A.8: GT Hand

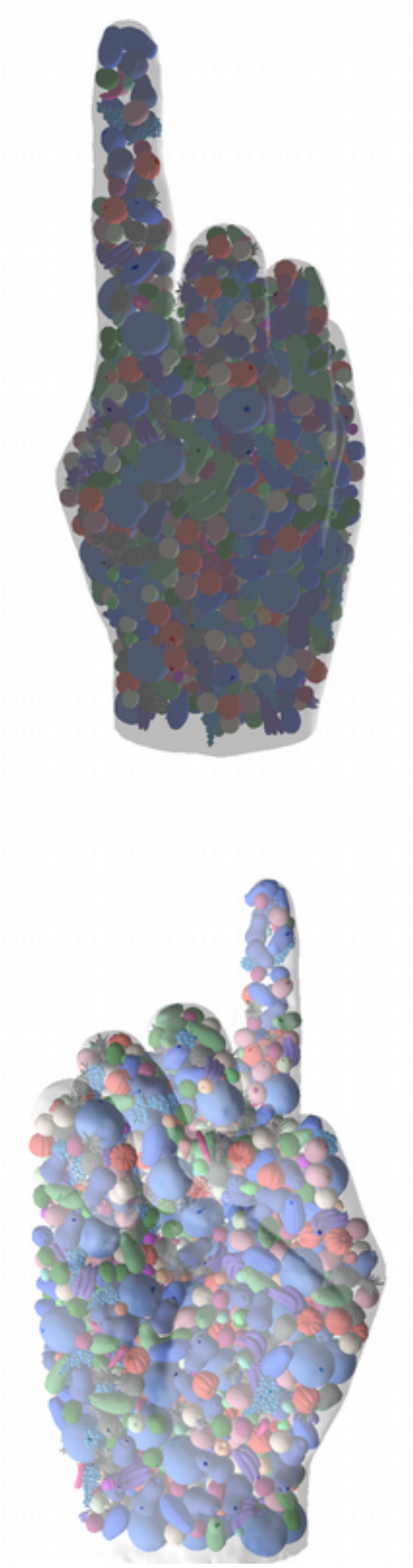


Abbildung A.9: CPUEight Hand

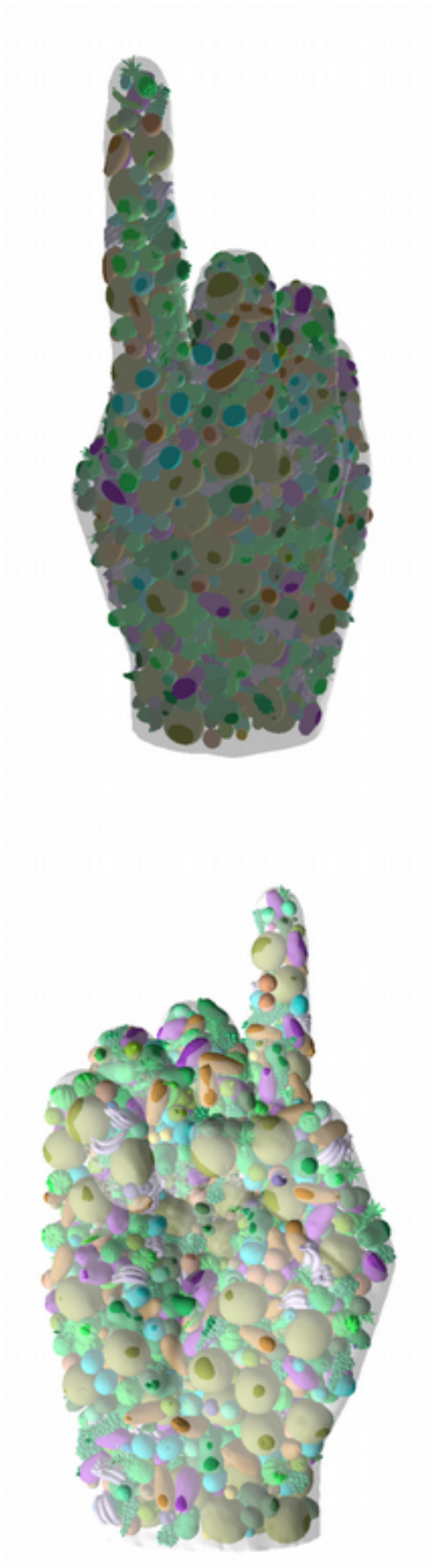


Abbildung A.10: GPUSpheres Hand

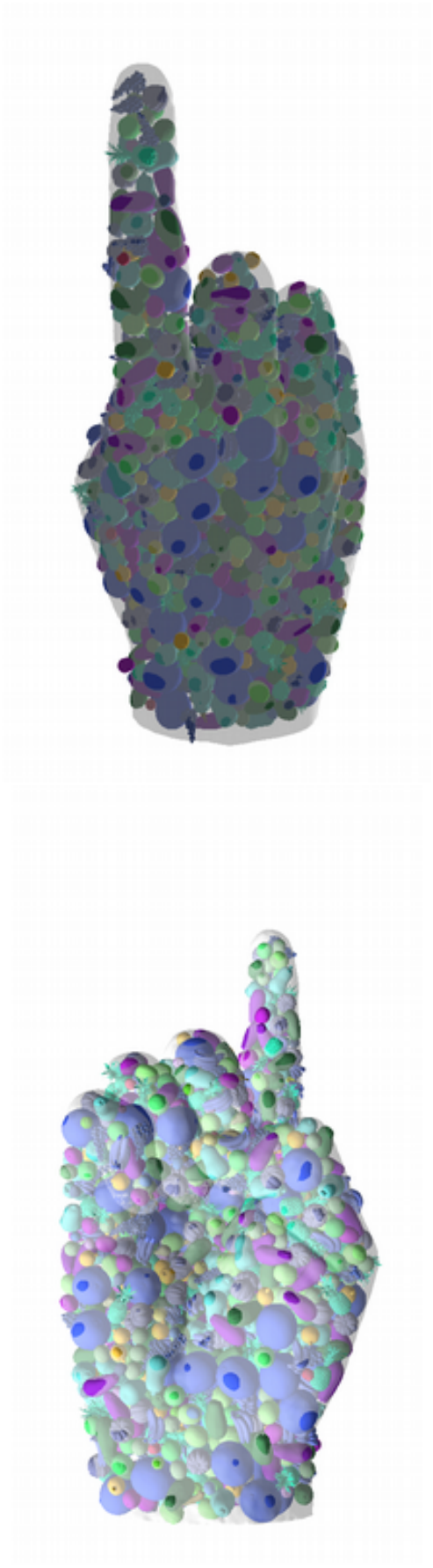


Abbildung A.11: GPUForce Hand

Abbildung A.12: DOP Hand

A.4 Packungsbilder Verbesserungsheuristiken

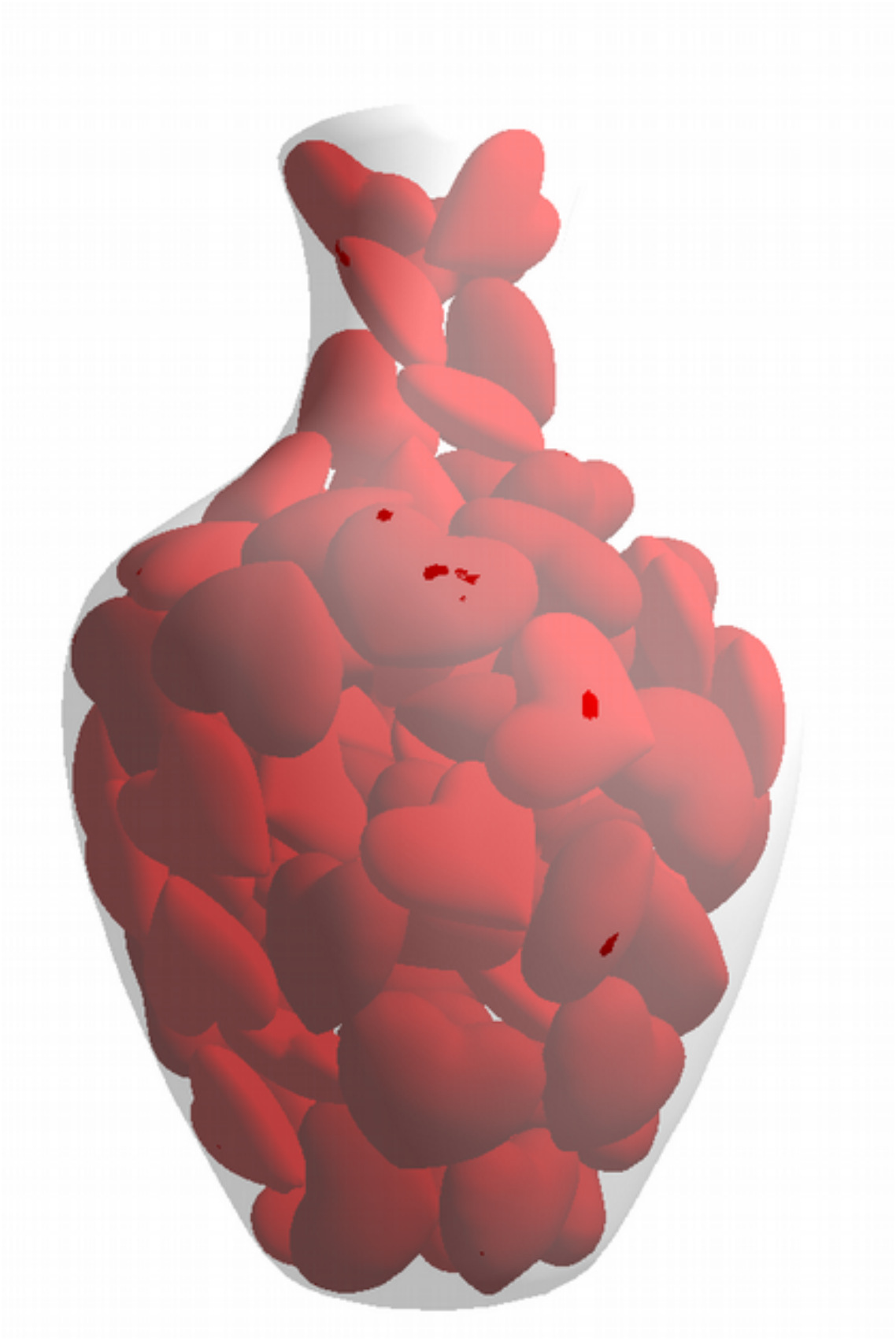


Abbildung A.13: Packung des Szenarios *Vase mit Herzen* verbessert durch *BFA*.

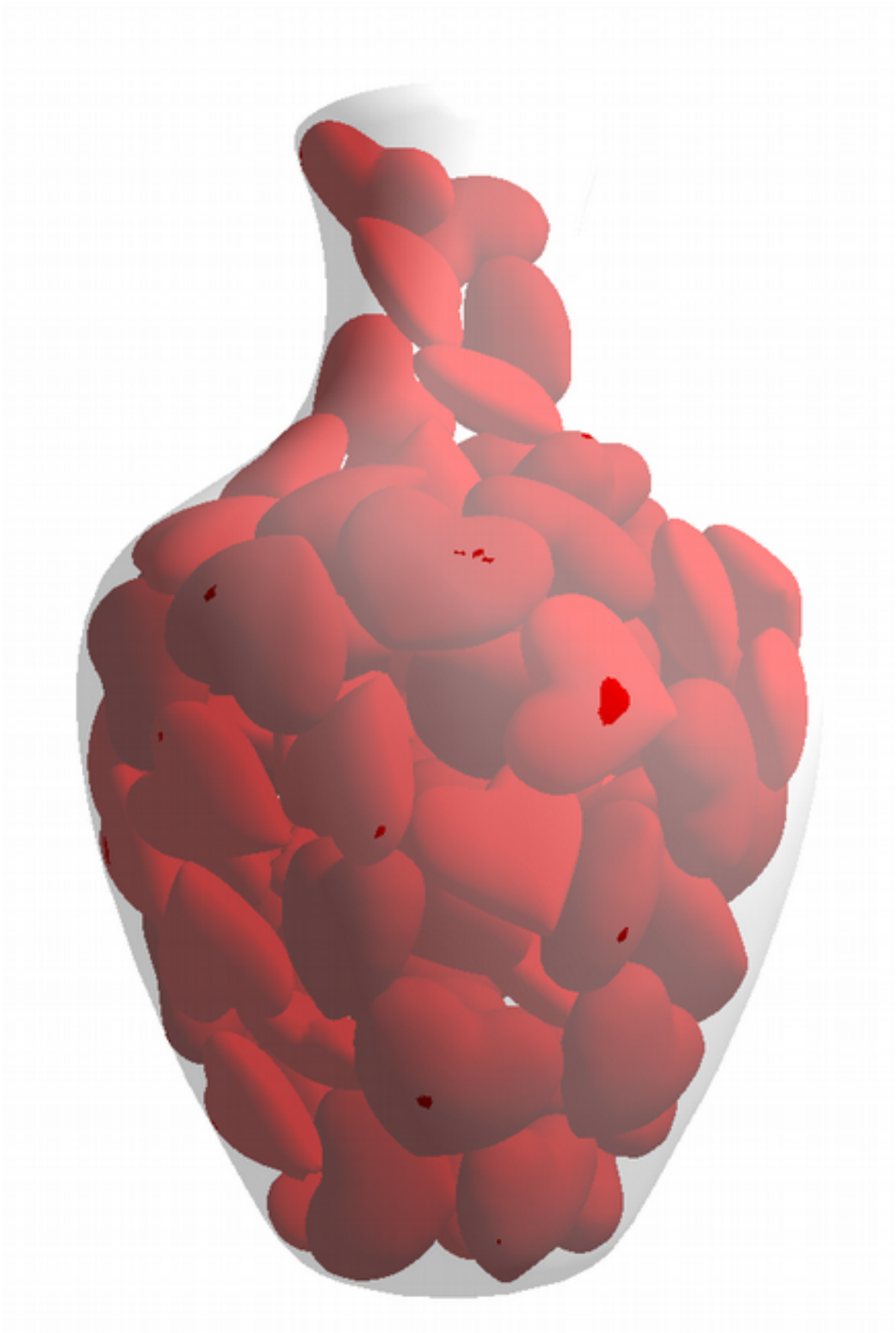


Abbildung A.14: Packung des Szenarios *Vase mit Herzen* verbessert durch *CFGA*.

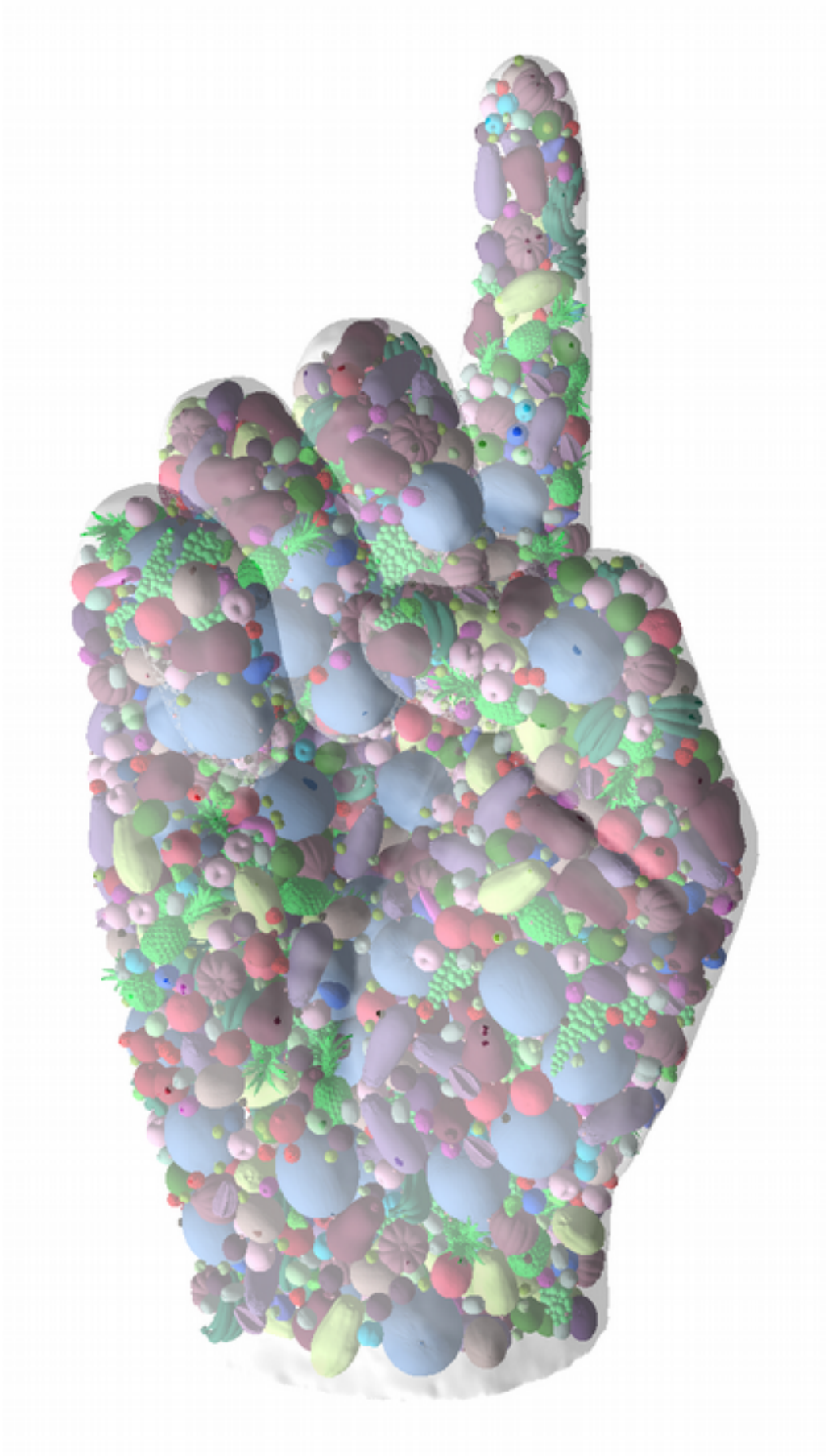


Abbildung A.15: Packung des Szenarios *Hand mit Früchten* verbessert durch *BFA*.

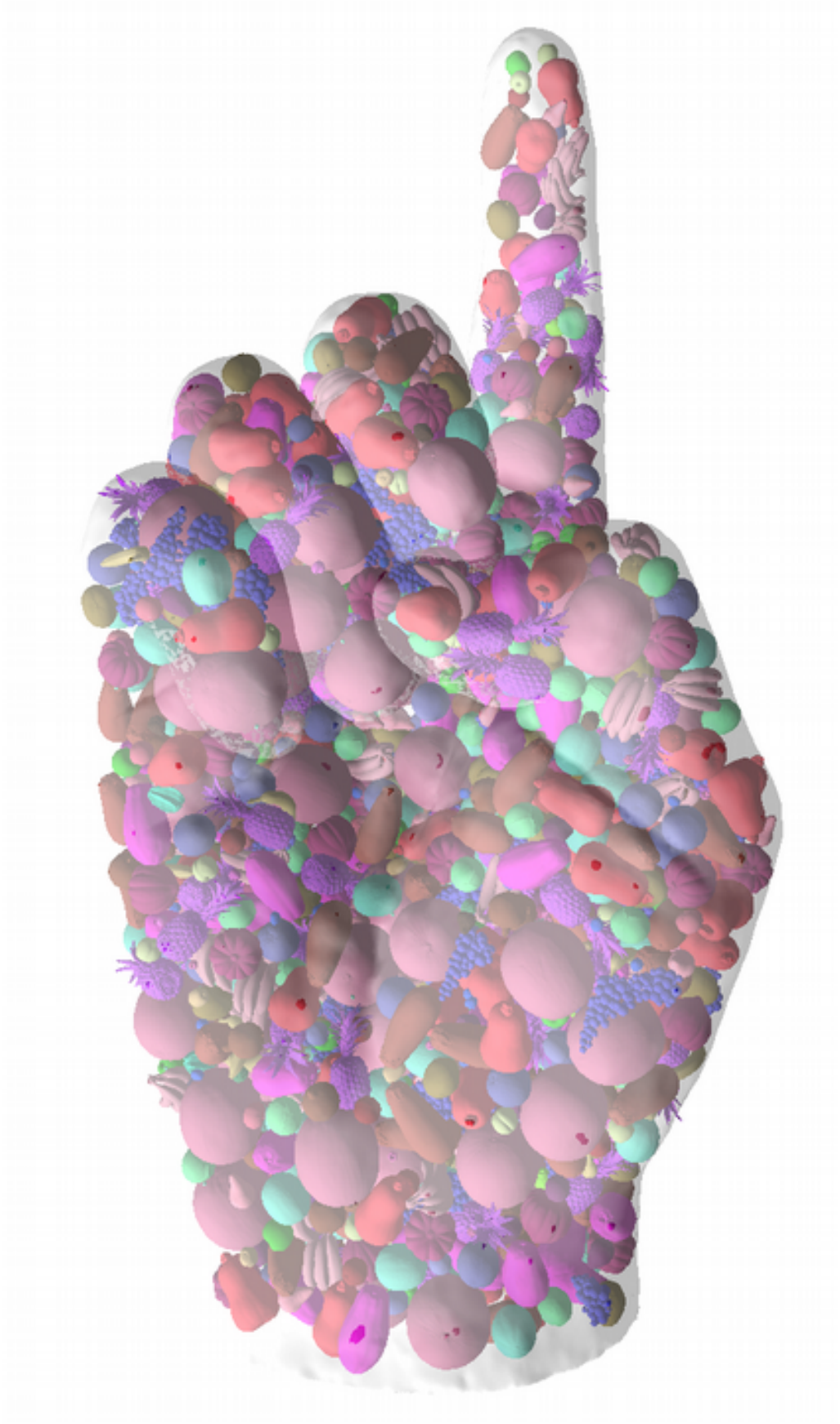


Abbildung A.16: Packung des Szenarios *Hand mit Früchten* verbessert durch *CFGA*.