Dense, Interlocking-Free and Scalable Spectral Packing of Generic 3D Objects

QIAODONG CUI, Inkbit, USA
VICTOR RONG, Massachusetts Institute of Technology, USA
DESAI CHEN, Inkbit, USA
WOJCIECH MATUSIK, MIT CSAIL, Inkbit, USA

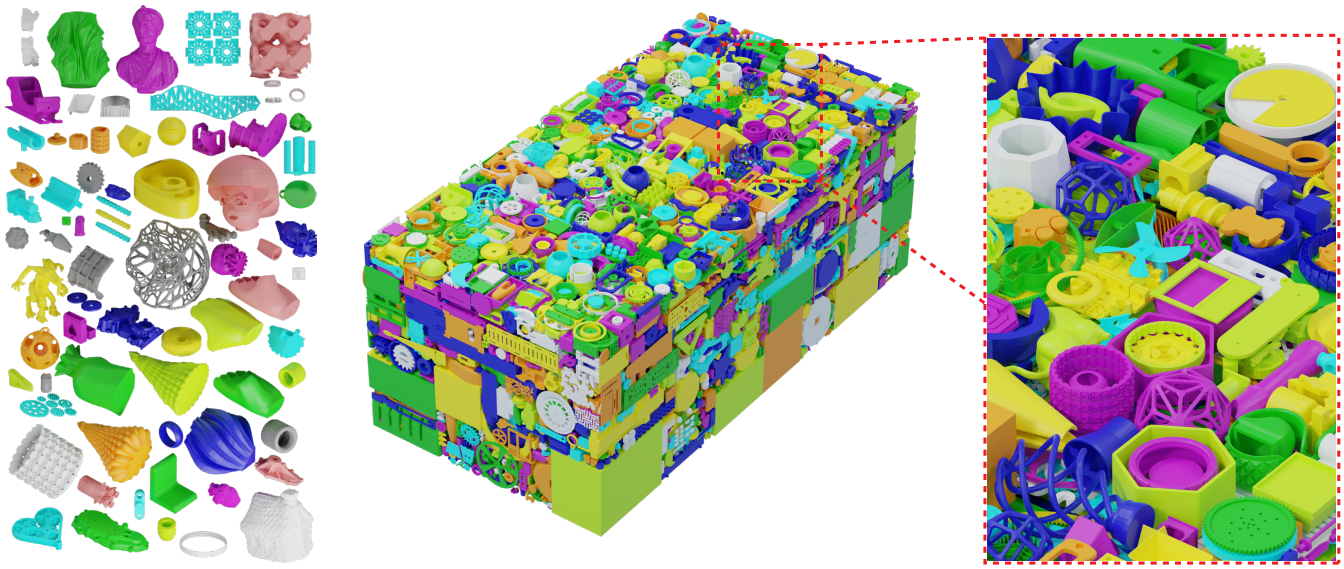

Fig. 1. **Left**: We select over 6000 objects from Thingi10K as our benchmark, which consists of many challenging geometries. **Middle**: We densely pack the benchmark into a cuboid with a packing density of 35.77%. The packing is free of interlocking. **Right**: An in-focused view highlighting densely packed objects.

Packing 3D objects into a known container is a very common task in many industries such as packaging, transportation, and manufacturing. This important problem is known to be NP-hard and even approximate solutions are challenging. This is due to the difficulty of handling interactions between objects with arbitrary 3D geometries and a vast combinatorial search space. Moreover, the packing must be *interlocking-free* for real-world applications. In this work, we first introduce a novel packing algorithm to search for placement locations given an object. Our method leverages a discrete voxel representation. We formulate collisions between objects as correlations of functions computed efficiently using Fast Fourier Transform (FFT). To determine the best placements, we utilize a novel cost function, which is also computed efficiently using FFT. Finally, we show how interlocking detection and correction can be addressed in the same framework resulting in interlocking-free packing. We propose a challenging benchmark with thousands of 3D objects to evaluate our algorithm. Our method demonstrates state-of-the-art performance on the benchmark when compared to existing methods in both density and speed.

CCS Concepts: • **Computing methodologies** → **Shape analysis**; *Volumetric models*; Collision detection.

Additional Key Words and Phrases: 3D Packing

ACM Reference Format:
Qiaodong Cui, Victor Rong, Desai Chen, and Wojciech Matusik. 2023. Dense, Interlocking-Free and Scalable Spectral Packing of Generic 3D Objects. *ACM Trans. Graph.* 42, 4, Article 141 (August 2023), 14 pages. https://doi.org/10.1145/3592126

1 INTRODUCTION

The packing problem is an optimization problem aiming to best arrange a set of objects into a container. It has been studied for hundreds of years due to its practical importance. For example, the famous Kepler conjecture was inspired by tightly stacking cannonballs in the 17th century. Since then, packing of regular objects such as boxes [Martello et al. 2000], ellipses [Stoyan et al. 2016], and tetrahedra [Chen et al. 2010] has been extensively studied, as well as packing of irregular 2D objects [Leao et al. 2020]. However, packing of generic 3D objects has received less attention despite

Authors' addresses: Qiaodong Cui, qcui@inkbit3d.com, Inkbit, Medford, USA; Victor Rong, Massachusetts Institute of Technology, USA, vrong@mit.edu; Desai Chen, Inkbit, USA, dchen@inkbit3d.com; Wojciech Matusik, MIT CSAIL, Inkbit, USA, wojciech@csail.mit.edu,wojciech@inkbit3d.com.

0730-0301/2023/8-ART141
https://doi.org/10.1145/3592126

its practical importance, such as in robotic packaging [Wang and Hauser 2019], 3D printing [Chen et al. 2015; Yao et al. 2015], transportation [Egeblad et al. 2010], and layout and pattern generation [Fanni et al. 2022; Reinert et al. 2013].

We focus on the problem of maximum density packing given a fixed container and a list of 3D objects. Packing density is the total volume of all packed objects divided by container volume. In this scenario, there may be more objects than what can fit into a single container. We later extend our base algorithm to multi-tray packing where the goal is to pack all objects using as few containers as possible. We also require the packing to be *interlocking-free*: parts must be physically disassembled without breaking or deforming them. The disassembly is allowed to move a part or a group of parts through one or more steps along possibly different directions. We gear our algorithm toward 3D printing [Sitthi-Amorn et al. 2015; Vanek et al. 2014]. In this case, the container is virtual and only serves to specify the printing volume. Objects can be disassembled in any direction after printing. This is different from packing in robotics such as Hu et al. [2020], where the objects are inserted through a physical opening.

The first challenge of packing generic 3D objects is addressing the complicated collision constraints that stem from arbitrary geometries. Many algorithms have been proposed previously [Lamas-Fernandez et al. 2022; Liu et al. 2015; Ma et al. 2018; Romanova et al. 2018], but it remains difficult to scale these algorithms to pack more than a few dozen complex objects. Inspired by works in protein docking [Katchalski-Katzir et al. 1992], we formulate collision constraints as correlations between discretized objects, which are computed in the spectral domain with FFT. This is efficient because a single convolution using FFT computes collision detection results for an object at all voxel locations. This algorithm is scalable to thousands of complex 3D objects. We then introduce a proximity metric at every voxel location, also computed using FFT. The resulting algorithm is extremely efficient at finding object placements for tight packing.

Another challenge is to avoid interlocking because objects must follow a non-colliding path when they are assembled or disassembled. Checking for disassembly is known to be difficult [Goldwasser et al. 1996]. We show that the same spectral framework comes to the rescue yet again, allowing interlocks to be both detected and resolved. The key insight is that interlocking-free locations can be computed efficiently with a flood-fill on the collision metric.

Expanding on the base algorithm, we employ a continuous search to refine placements at sub-voxel locations. We also propose a ray casting disassembly method to efficiently handle easy-to-disassemble parts. Lastly, we extend the algorithm to multi-tray packing. To evaluate our algorithm, we generate a set of diverse benchmarks consisting of thousands of generic 3D objects. In summary, our contributions are:

- An efficient placement search based on FFT
- A disassembly algorithm in the same unified framework
- A continuous placement refinement method
- A broad-phase ray casting disassembly
- An extension to multi-tray packing
- A set of challenging benchmarks for future evaluations

2 RELATED WORK

Packing problems have been studied many times in the context of computational geometry and optimization [Wang et al. 2021]. Previous approaches often use expensive geometric queries [Hang 2015; Liu et al. 2015; Ma et al. 2018] to enforce collision constraints. Our algorithm only uses a minimal amount of geometry processing by working with a voxel representation computed at a rate of 10 ms per million triangles on a GPU [Schwarz and Seidel 2010]. Our FFT-based collision metric takes 3 ms to identify all collisions on a grid with 24 million voxels.

Our algorithm is inspired by previous work in molecular biology. Katchalski-Katzir et al. [1992] proposed an efficient FFT-based algorithm to evaluate the quality of all ligand-protein docking configurations for a given orientation. More recent works have modeled proteins' shapes with spherical harmonics to include rotations [Padhorny et al. 2016; Ritchie and Kemp 2000; Ritchie and Venkatraman 2010]. Compared to these works, we pack hundreds of objects rather than two proteins and ensure the interlocking-free constraint. In our setting, it is difficult to approximate man-made CAD models using spherical harmonics. This is because CAD models usually contain sharp edges and corners, requiring many high-order spherical harmonics bases to accurately resolve. Therefore, we choose to perform 3D Cartesian FFTs for higher speed and accuracy, while limiting the number of possible rotations. We briefly summarize other lines of work on packing in the following paragraphs.

No-fit Polygon. A no-fit polygon (NFP, also known as a Minkowski difference) is a compact representation for the set of positions that would cause collisions between two polygons [Art Jr 1966]. In 2D, NFPs are computed by sliding polygon A along the edges of polygon B and tracing the center of polygon A, which requires careful treatment of edge cases [Burke et al. 2007]. Exact NFPs in 3D do not scale well even for convex polyhedra as the algorithms have a quadratic complexity in the number of vertices.

Voxel-based Methods. Vanek et al. [2014] and Chen et al. [2015] demonstrated a method to decompose and pack objects for 3D printing. Lamas-Fernandez et al. [2022] proposed no-fit voxels (NFV) as a 3D analogy to NFPs. NFVs are computed in the spatial domain by sliding one object across all voxel locations and checking for any overlapping voxels. The authors used bounding boxes to accelerate detection of non-overlapping positions. Our FFT-based collision metric serves the same function as NFVs but the computation is much more efficient with linearithmic runtime in the number of voxels.

Mathematical Models. Packing can be modeled as a nonlinear optimization problem [Pankratov et al. 2020; Romanova et al. 2018]. Collision constraints are encoded using phi-functions between pairs of objects, which can be viewed as an analytical version of NFPs for 3D objects. For triangular meshes, phi-functions are derived from pairwise distance functions between vertices and triangles [Chernov et al. 2010]. Thus the complexity grows quadratically. This limits the results to a small number of simple objects (typically fewer than 100). We refer the readers to Leao et al. [2020] for a more in-depth review of mathematical formulations for packing problems.

Learning to Pack. Goyal and Deng [2020] used deep reinforcement learning (DRL) to train a neural network for picking the next shape to pack. DRL has been used to learn selection and placement policies for box packing with robot arms [Hu et al. 2020; Yang et al. 2021; Zhao et al. 2021]. To use trained policies, a placement step is still needed to compute the packed configuration, which is often performed using rigid body simulation software [Coumans and Bai 2021; Lan et al. 2022]. Our work primarily focuses on improving placement speed and preventing interlock rather than designing policies for selecting the next shape to pack.

Disassembly Planning. Song et al. [2012] demonstrated an algorithm to design interlocking puzzles using voxels. Wang et al. [2018] used Directional Blocking Graphs to efficiently construct puzzles. We use the same representation to accelerate disassembly. Zhang et al. [2020] proposed a path planning algorithm that considers both translation and rotation. Chen et al. [2022] presented an algorithm for high-level puzzles where a piece can only be removed after moving several pieces. Tian et al. [2022] used rigid body simulation to disassemble mechanical assemblies. These works focus on tightly coupled assemblies which necessitate complicated paths with rotations. Our packed assemblies are usually loosely coupled but consist of hundreds of objects. Thus we develop disassembly algorithms that are more suitable to packing. We also only allow translations in the disassembly process to keep it simple for users.

Packing Benchmarks. Packing benchmarks are rare [Araújo et al. 2019]. Goyal and Deng [2020] constructed a dataset with a large number of packs, however, each pack only contains an average of 23 objects. Our benchmark constructed from Thingi10k [Zhou and Jacobson 2016] contains hundreds of diverse shapes per pack.

3 SCALABLE SPECTRAL PACKING

We present our packing algorithm in two parts. We start with the placement step which is performed efficiently in the spectral domain. Then, we describe a greedy algorithm that packs objects densely and without collision. Interlocking prevention is addressed in §4.

The placement search step attempts to find a position for a given object A in the container (tray), such that A is fully contained in the tray and not colliding with the set of existing objects Ω. Efficient collision detection is critical for this step. To effectively search in the placement space, we perform the search using a discrete step followed by a continuous refinement step.

3.1 Discrete Placement Search with FFT

We begin the discrete placement search by formulating collision detection as a correlation between voxel grids of A and Ω [Katchalski-Katzir et al. 1992]. We refer to the result of the correlation computation as the collision metric. For n voxels, computing the collision metric at every voxel location by brute force is impractical at $O(n^2)$ complexity. Instead, we compute the collision metric in spectral domain with FFT spending only $O(n \log n)$ time. For example, the brute force method on a $240 \times 123 \times 100$ grid takes 9.4 seconds on average, while FFT only takes 0.003 seconds, which is three orders of magnitude faster.

Next, we propose a proximity metric to score the collision-free placements by how well A fits within Ω. Here a better fit means a smaller gap between A and Ω. The proximity metric is also formulated as a correlation. Lastly, to encourage a lower packing height, a simple height penalization term is added to the score.

Collision Metric. We represent an object $A \subset \mathbb{R}^3$ and a set of existing objects $\Omega \subset \mathbb{R}^3$ with indicator functions $s_A(\mathbf{x})$ and $s_\Omega(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^3$ is a point in 3D space:

$$s_A(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in A, \\ 0 & \text{if } \mathbf{x} \notin A. \end{cases} \qquad s_\Omega(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in \Omega, \\ 0 & \text{if } \mathbf{x} \notin \Omega. \end{cases} \tag{1}$$

The indicator functions over the voxel grid s_A and s_Ω are computed using a conservative voxelization [Schwarz and Seidel 2010]. Then, we compute the collision metric $\zeta_{A,\Omega}(\mathbf{q}), \mathbf{q} \in \mathbb{R}^3$ as a correlation between $s_A(\mathbf{x})$ and $s_\Omega(\mathbf{x})$ for each displacement $\mathbf{q}$ of A:

$$\zeta_{A,\Omega}(\mathbf{q}) = \int s_A(\mathbf{x}) s_\Omega(\mathbf{x} - \mathbf{q}) d\mathbf{x}. \tag{2}$$

The collision metric $\zeta_{A,\Omega}(\mathbf{q})$ is non-negative. Zero values of $\zeta_{A,\Omega}(\mathbf{q})$ indicate collision-free positions for the object A while positive values correspond to colliding positions for A. A description of computing the collision metric $\zeta_{A,\Omega}$ is shown in Fig. 2.

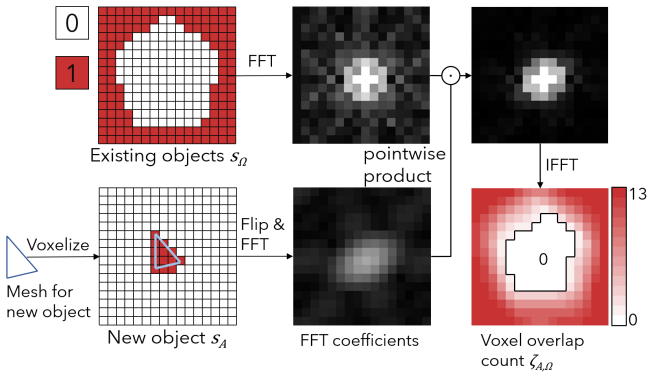

Fig. 2. Computing collision metric $\zeta_{A,\Omega}$ using Fast Fourier Transform (FFT). Existing objects including the container have already been voxelized (top left). The new mesh is voxelized and padded to the background grid size (bottom left). To compute the collision metric, we assign 0 to empty voxels and 1 to occupied voxels. We then perform FFT on both grids (center). We show the magnitude of the complex Fourier coefficients where the low frequency values are grouped in the center. We then perform inverse FFT on the pointwise product of the Fourier coefficients (right). The final output is a per-voxel overlap count where zero values indicate collision-free offsets for the new object.

Proximity Metric. To define the proximity metric, we first compute an unsigned distance function of the set of objects Ω:

$$\phi_\Omega(\mathbf{x}) = \begin{cases} d(\mathbf{x}, \Omega) & \text{if } s_\Omega(\mathbf{x}) = 0, \\ 0 & \text{if } s_\Omega(\mathbf{x}) = 1, \end{cases} \tag{3}$$

where $d(\mathbf{x}, \Omega) = \min_{\forall \mathbf{p} \in \partial\Omega} |\mathbf{x} - \mathbf{p}|$ is the unsigned distance from point $\mathbf{x}$ to Ω. Then, the proximity metric is defined as:

$$\rho_{A,\Omega}(\mathbf{q}) = \int s_A(\mathbf{x})\phi_\Omega(\mathbf{x} - \mathbf{q})d\mathbf{x}. \tag{4}$$

When object A moves closer to Ω, proximity metric $\rho_{A,\Omega}$ becomes smaller. Thus, the proximity metric $\rho_{A,\Omega}(\mathbf{q})$ measures how closely the object A fits into the set of objects Ω at the location $\mathbf{q}$.

Height Penalization. A major application of our algorithm is raster-based 3D printing, which prints in a layer by layer fashion [Sitthi-Amorn et al. 2015]. Because each layer takes a constant amount of time, the print time is proportional to the height of the packing. To minimize print time, we add a height penalization term:

$$h(\mathbf{t}) = p\mathbf{q}_z^3, \tag{5}$$

where $\mathbf{q}_z$ is the z coordinate of the placement normalized to the range $[0, 1]$, and p is a non-negative parameter. We choose the cubic exponent to discourage placement at a taller height. The choice for p is discussed in §A.2.

We add the proximity metric and height penalization into a single cost function $\sigma_{A,\Omega}(\mathbf{q}) = \rho_{A,\Omega}(\mathbf{q}) + h(\mathbf{q})$ and search for the collision-free location $\mathbf{q}^*$ with minimum cost:

$$\mathbf{q}^* = \arg\min_{\mathbf{q}} \{\sigma_{A,\Omega}(\mathbf{q}) \mid \zeta_{A,\Omega}(\mathbf{q}) = 0\}. \tag{6}$$

Since $\zeta_{A,\Omega}$ and $\sigma_{A,\Omega}$ are efficiently computed for all voxels, the optimization can be solved by iterating over all voxels whose collision metric is zero and finding the global minimum.

Orientations Sampling. To obtain a better placement, we rotate the object using a list of orientations. For each orientation $\mathbf{r}_j$, we find the best translation $\mathbf{q}_j$ and the minimal cost $c_j = \sigma_{A,\Omega}(\mathbf{q}_j)$. We then select the orientation with the minimum cost. Orientations can be sampled by uniformly dividing Euler angles or using subdivisions of an icosahedron. The entire discrete placement search is described in Algorithm 1 with more implementation details in §A.1.

Algorithm 1 Placement Search with FFT

function FFTSearchPlacement($A, \Omega, m, \mathbf{l}, dx$)
  Input: An object A to be placed, existing set of placed objects Ω, the number of orientations to search m, tray dimension $\mathbf{l} = (l_x, l_y, l_z)$ along the x, y, z axis, voxel size dx
  Output: Placement successful (True or False), Placement position $\mathbf{q}$, rotation $\mathbf{r}$
  $s_\Omega \leftarrow$ Voxelize($\Omega, \mathbf{l}, dx$)
  $\widetilde{s_\Omega} \leftarrow$ FFT(s_Ω) ▷ s_Ω in spectral domain
  $\phi_\Omega \leftarrow$ ComputeDistance(s_Ω)
  $\widetilde{\phi_\Omega} \leftarrow$ FFT(ϕ_Ω) ▷ ϕ_Ω in spectral domain
  $j \leftarrow 1, c \leftarrow \infty, \mathbf{q} \leftarrow \mathbf{0}, \mathbf{r} \leftarrow \mathbf{I}$
  while $j \le m$ **do** ▷ Search over orientations
    $\mathbf{r}_j \leftarrow$ SampleEulerAngle(j) ▷ Sample orientations
    $A_j \leftarrow$ Rotate($A, \mathbf{r}_j$)
    $s_{A_j} \leftarrow$ Voxelize($A_j, \mathbf{l}, dx$)
    $\widetilde{s_{A_j}} \leftarrow$ FFT(s_{A_j})
    $\zeta_{A_j,\Omega} \leftarrow$ IFFT($\widetilde{s_{A_j}} \cdot \widetilde{s_\Omega}$) ▷ Collision metric
    $\rho_{A_j,\Omega} \leftarrow$ IFFT($\widetilde{s_{A_j}} \cdot \widetilde{\phi_\Omega}$) ▷ Proximity metric
    $\sigma_{A_j,\Omega}(\mathbf{x}) \leftarrow \rho_{A_j,\Omega}(\mathbf{x}) + h(\mathbf{x})$ ▷ Add height penalty
    $\mathbf{X} \leftarrow \{\mathbf{x}_i \mid \zeta_{A_j,\Omega}(\mathbf{x}_i) = 0\}$ ▷ Non-colliding locations
    if $\mathbf{X} \neq \emptyset$ **then**
      $\{c_j, \mathbf{x}_j\} \leftarrow \min \sigma_{A_j,\Omega}(\mathbf{x}), \mathbf{x} \in \mathbf{X}$ ▷ Find minimum
      if $c_j < c$ **then**
        $c \leftarrow c_j, \mathbf{q} \leftarrow \mathbf{x}_j, \mathbf{r} \leftarrow \mathbf{r}_j$
      end if
    end if
    $j \leftarrow j + 1$
  end while
  if $c < \infty$ **then**
    Return {True, $\mathbf{q}, \mathbf{r}$} ▷ Successful placement
  else
    Return {False, $\mathbf{0}, \mathbf{I}$} ▷ Unsuccessful placement
  end if
end function

3.2 Continuous Placement Refinements

The discrete placement search is performed on voxel positions, where the minimal spacing is determined by the voxel size dx. This leaves unnecessary spacing between objects, as shown on the left of Fig. 3. However, reducing the voxel size dx is expensive as the size of the voxel grid scales cubically. To address this issue, we apply a continuous refinement step to reduce the spacing between objects. In this step, objects are represented by their original geometries for accuracy. We then progressively update the placement position $\mathbf{q}$ of the object A without colliding with Ω, until it is within a desired margin from Ω.

Specifically, we implement the continuous refinement as a binary search along a direction. At each binary search step, a collision detection is performed between A and Ω. To accelerate, we use axis-aligned bounding box trees [Jacobson et al. 2018]. We then use the separation axis theorem [Ericson 2004] to detect if two triangles are within the margin. The movement direction is the discrete gradient $\nabla\sigma_{A,\Omega}$ at voxel $\mathbf{q}$. The maximum movement range is $2dx$. In practice, we apply continuous refinement three times along x, y, z axis separately. For each axis, the corresponding component of $\nabla\sigma_{A,\Omega}$ determines the movement direction. In practice, this process is effective because many objects are axis aligned.

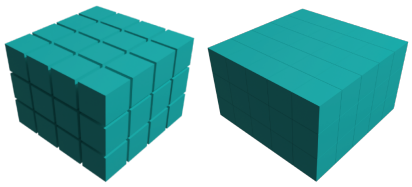

Fig. 3. **Left**: Packing cubes with discrete placement search using voxel size $dx = 2$ mm leaves undesirable spacing. **Right**: With the addition of continuous placement refinements, the spacing between cubes can be reduced to a user-specified value, in this case 0.2 mm.

3.3 Packing Objects Greedily

We use a greedy strategy that orders all objects from largest to smallest according to bounding box volumes. The objects are then

sequentially placed into the container with the placement search. This is described in Algorithm 2. This heuristic is inspired by the commonly used first-fit-decreasing algorithm [Johnson 1973]. While a combinatorial search over object ordering may yield better results (§B.3), we leave it as future work due to the exponential search space. We extend the greedy strategy to multi-tray packing shown in §A.3. The complexity of the packing algorithm described thus far is $O(Nm(n \log n))$, where N is the number of objects, n is the number of voxels, and m is the number of rotations.

Algorithm 2 Greedy Packing with Placement Search

function GreedyPackObjects($\mathbf{A}, m, \mathbf{l}, dx$)
  Input: A list of objects $\mathbf{A}$, the number of orientations to search m, tray dimension $\mathbf{l}$, voxel size dx.
  Output: A list of successfully packed objects $\mathbf{P}$. A list of translations $\mathbf{Q}$ and rotations $\mathbf{R}$ of objects in $\mathbf{P}$. A list of unsuccessfully packed objects $\mathbf{U}$.
  $\mathbf{P} \leftarrow \emptyset, \mathbf{Q} \leftarrow \emptyset, \mathbf{R} \leftarrow \emptyset, \mathbf{U} \leftarrow \emptyset, i \leftarrow 1$
  $\mathbf{A} \leftarrow$ Sort($\mathbf{A}$) ▷ Sort all objects from largest to smallest
  while $i \leq$ A.size() **do**
    $\{b, \mathbf{q}_i, \mathbf{r}_i\} \leftarrow$ FFTSearchPlacement($\mathbf{A}_i, \mathbf{P}, m, \mathbf{l}, dx$)
    if b **then** ▷ Placement successful
      P.add($\mathbf{A}_i$)
      Q.add($\mathbf{q}_i$)
      R.add($\mathbf{r}_i$)
    else ▷ Placement unsuccessful
      U.add($\mathbf{A}_i$)
    end if
    $i \leftarrow i + 1$
  end while
  Return $\{\mathbf{P}, \mathbf{Q}, \mathbf{R}, \mathbf{U}\}$;
end function

4 INTERLOCKING-FREE PACKING WITH DISASSEMBLY

The algorithm described in §3 may produce interlocking. To address this, we introduce a novel flood-fill disassembly algorithm to detect any interlocks. Once interlocks are detected, one or more objects are removed from the packing until all interlocks are resolved. Next, we derive a placement method based on flood-fill disassembly, which is used to reinsert removed objects into the packing without introducing new interlocks. Finally, to improve the efficiency of disassembly, we propose a fast ray-casting disassembly.

4.1 Flood-Fill Disassembly

The collision metric $\zeta_{A,\Omega}$ already contains enough information for the disassembly. Specifically, an object A can physically move from a start location $\mathbf{q}$ to another location $\mathbf{q}'$ if there is a path connecting them such that $\zeta_{A,\Omega}$ is zero along the path. Thus, given a set of locations $\mathbf{X}$ where A is considered disassembled, the disassembly check can be performed as a flood-fill on the voxel grid $\zeta_{A,\Omega}$, which is described in Algorithm 3. Fig. 4 shows examples of determining reachable spaces using flood-fill.

An object is fully disassembled when its bounding box is outside of the tray. Therefore, at any feasible starting location $\mathbf{x} \in \mathbf{X}$, the bounding box of A should not intersect Ω. This can be achieved by zero-padding the voxel grid $\zeta_{A_j,\Omega}$ with the size of the object's voxel grid s_A. Then, the feasible starting locations $\mathbf{X}$ are the voxels on the six boundary faces of the grid $\zeta_{A_j,\Omega}$. Since the size of grid s_A depends on the object's geometry and orientation, the grid $\zeta_{A_j,\Omega}$ must be dynamically resized. This is usually 1.2-4x slower than the metric computation in the FFT placement algorithm, depending on the size of the object.

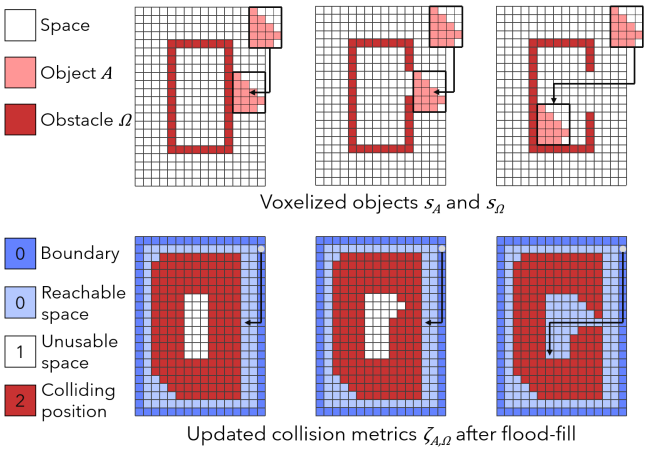

Fig. 4. Flood-fill for computing all reachable positions. The top row shows three cases of rectangular boxes. The bottom row shows the collision metrics $\zeta_{\Omega,A}$ and flood-filled labels indicating valid packing positions for the object A. The boundary voxels (dark blue) are initialized to 0 indicating that the object is completely outside of the assembly. The light blue voxels are reachable positions computed by flood-fill. In the first column, the box is water-tight so nothing can enter or exit. In the middle column, the opening is too narrow to fit the object A. In the last case, there is a perfect opening for the object to enter.

Algorithm 3 Disassembly of One Object with Flood-Fill

function FloodFillDisassembly($\zeta_{A,\Omega}, \mathbf{q}, \mathbf{X}$)
  Input: Collision metric $\zeta_{A,\Omega}$, placement location $\mathbf{q}$, a set of feasible starting locations $\mathbf{X}$.
  Output: A Boolean indicates whether A can be physically disassembled from the placement location $\mathbf{q}$.
  $\zeta_{A,\Omega}(\mathbf{x}_i) \leftarrow 2,\ \forall \mathbf{x}_i \in \{\mathbf{x}_i \mid \zeta_{A_j,\Omega}(\mathbf{x}_i) \neq 0\}$ ▷ Colliding voxel
  $\zeta_{A,\Omega}(\mathbf{x}_i) \leftarrow 1,\ \forall \mathbf{x}_i \in \{\mathbf{x}_i \mid \zeta_{A_j,\Omega}(\mathbf{x}_i) = 0\}$
  $\zeta_{A,\Omega}(\mathbf{x}_i) \leftarrow 0,\ \forall \mathbf{x}_i \in \mathbf{X}$ ▷ Feasible voxel
  while not converged **do**
    $\zeta_{A,\Omega}(\mathbf{y}) \leftarrow 0$ If $\zeta_{A,\Omega}(\mathbf{y}) = 1$ and $\exists \mathbf{y}_i, \zeta_{A,\Omega}(\mathbf{y}_i) = 0$, and $\mathbf{y}_i$ is a neighbor of $\mathbf{y}$ ▷ Flood-fill 1 with 0
  end while
  if $\zeta_{A,\Omega}(\mathbf{q}) = 0$ **then** ▷ $\mathbf{q}$ is feasible
    Return True;
  else
    Return False;
  end if
end function

We apply the flood-fill check to disassemble a packed tray. The disassembly algorithm for the packed tray is described in Algorithm 4 and visualized in Fig. 5. Specifically, we examine each object sequentially and disassemble objects that pass the check. This procedure is repeated until either all objects are disassembled, or until the remaining objects cannot be disassembled due to interlocking. If the remaining objects are interlocked, we remove a subset of them from the packed tray, and add them to the list of unsuccessfully packed objects. The SelectRemove step of Algorithm 4 is used to choose a subset of parts to explicitly remove. We first compute the total bounding volume of $\mathbf{A}_m$. Then, starting from the smallest object, we remove objects until the bounding volume of all removed objects exceeds 10% of the total bounding volume of $\mathbf{A}_m$.

Our flood-fill disassembly favors simpler disassembly plans by only allowing translations. This is not a drawback but an important aspect for our applications. For example, this makes it easier to physically disassemble a 3D printed tray. Algorithm 4 is very conservative in removing objects to preserve packing density. This may result in a quadratic number of flood-fill operations in the worst case. To address this, we developed a ray-casting-based disassembly (§4.4) to quickly disassemble a large fraction of the objects first. Then Algorithm 4 is used to examine the remaining objects. In summary, we find the flood-fill disassembly strikes a good balance between the packing density and the disassembly complexity.

Algorithm 4 Disassembly for a List of Objects

function DisassemblyObjectsFloodFill($\mathbf{A}_p, \mathbf{Q}, \mathbf{R}, \ll, dx$)
  Input: A list of packed objects $\mathbf{A}_p$, the list of translations $\mathbf{Q}$, the list of rotations $\mathbf{R}$, tray dimension $\ll$, voxel resolution dx
  Output: A list of removed objects $\mathbf{A}_r$ to resolve interlocking.
  $\mathbf{A}_r \leftarrow \emptyset$ ▷ List of removed objects
  $\mathbf{A}_m \leftarrow \mathbf{A}_p$ ▷ List of remaining objects
  while $\mathbf{A}_m \neq \emptyset$ **do**
    $\mathbf{A}_d \leftarrow \emptyset$ ▷ List of disassembled objects
    for A_i in $\mathbf{A}_m$ **do**
      $\Omega \leftarrow \mathbf{A}_m - A_i$
      $A_j \leftarrow$ Rotate($A_i, \mathbf{R}[A_i]$)
      $\zeta_{A_j,\Omega} \leftarrow$ ComputeMetric($\Omega, A_j, \ll, dx$)
      $\mathbf{X} \leftarrow$ ExtractBoundary($\zeta_{A_j,\Omega}$)
      if FloodFillDisassembly($\zeta_{A_j,\Omega}, \mathbf{Q}[A_i], \mathbf{X}$) **then**
        $\mathbf{A}_d$.add(A_i)
      end if
    end for
    if $\mathbf{A}_d = \emptyset$ **then** ▷ Everything is interlocked
      $\mathbf{A}_{\text{tmp}} \leftarrow$ SelectRemove($\mathbf{A}_m$)
      $\mathbf{A}_r$.add($\mathbf{A}_{\text{tmp}}$)
      $\mathbf{A}_m$.remove($\mathbf{A}_{\text{tmp}}$)
    else
      $\mathbf{A}_m$.remove($\mathbf{A}_d$)
    end if
  end while
  Return $\mathbf{A}_r$
end function

4.2 Flood-Fill Disassembly with Refinements

Algorithm 3 only checks if an object A can be disassembled from discrete voxel locations. With continuous refinement, the placement $\mathbf{q}$ may not lie exactly on a voxel. Moreover, A may share voxels with other objects so that the initial position would be marked as colliding. To address this, we snap A to a voxel once flood-fill has found all reachable grid locations. In particular, we compute the distance of every feasible grid location ($\zeta_{A_j,\Omega}(\mathbf{x}) = 0$) to $\mathbf{q}$, and select the top ten candidates closest to $\mathbf{q}$. For each candidate location $\mathbf{x}_c$, we define a snapping vector $\mathbf{y} = \mathbf{x}_c - \mathbf{q}$. Then, we check the path from $\mathbf{q}$ to $\mathbf{x}_c$ by building a swept triangle mesh of object A along $\mathbf{y}$. Using AABB trees and the separating axis theorem, we can determine if the swept mesh collides with any other remaining objects [Ericson 2004]. The object is successfully disassembled if it snaps to any of the candidates.

4.3 Interlocking-Free Placement

The flood-fill disassembly check can be easily reformulated to enforce interlocking-free placement. This can be achieved by performing flood-fill on the collision metric before searching for placements in Algorithm 1. Voxels marked as zero on the collision metric will be physically feasible. As a result, any placements produced algorithm produces will not cause interlocking.

Interlocking-free placement can be used to avoid the disassembly step, but it is slower as we need to perform flood-fill for *every* object orientation. This strategy also lowers packing density in most cases, because flood-fill prevents objects from being placed inside cavities of the packing, as shown in §5.2. Thus we prefer to only use the interlocking-free placement for reinsertion.

4.4 Ray-Casting Disassembly

We observe a large fraction of objects can be removed following a straight line in our assemblies. Inspired by this, we develop a fast ray-casting-based disassembly. First, we cast rays on the assemblies along a given direction. Then, we extract the precedence information along each ray, which can be encoded as directed edges on a graph where nodes represent objects. The resulting graph is also referred to as a Directional Blocking Graph (DBG) [Wang et al. 2018; Wilson and Latombe 1994]. We use a graph analysis tool similar to Wang et al. [2018] for disassembly.

Specifically, after obtaining the DBG along a ray direction, we extract strongly connected components (SCCs) from the graph. The SCCs represent a group of objects that cannot be separated along that ray direction. However, different SCCs can be separated following a topological order on the graph. We then remove any SCCs with only one node. We repeat this process 2 times along each x, y, z direction, which usually removes a large fraction of objects with no interlocking. The algorithm is described in Fig. 6. The remaining SCCs can still potentially be disassembled. Therefore, we use flood-fill disassembly in §4.1 to examine each SSC. This greatly reduces the number of input objects for the flood-fill disassembly and improves its efficiency. If the largest SCC contains S objects, where $S \ll N$, then the complexity of the disassembly step is $O(SN(n \log n))$. The total time complexity of placement and disassembly is $O((m + S)N(n \log n))$. The number of voxels n scales

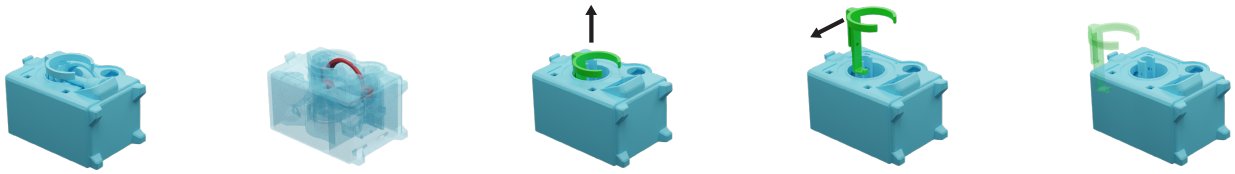

Fig. 5. Flood-fill disassembly. From left to right: The algorithm checks if each object can be physically disassembled. An object may be removed if no other objects can be disassembled. For example, the ring object marked with red is removed in the second figure. Once the interlocking is resolved, the algorithm iterates over the remaining objects, and removes them following a physically valid path.

cubically with the inverse voxel size $1/dx$. We evenly sample Euler angles along three axes to produce rotations. Thus, m scales cubically with the angular resolution. For best results, one should choose the finest dx within the memory limit and sample rotations every 90°, as in appendix B.2.

5 RESULTS

We first describe the benchmarks used throughout this section. Next, we show results using different interlocking prevention strategies and continuous refinement. Then we explore different algorithm parameter choices. We also demonstrate the result on multi-tray packing. Lastly, we compare our algorithm against a variety of commercial software and other techniques.

All of our experiments are run on a desktop equipped with an RTX3090Ti GPU and an i7-13700K CPU. We implemented our algorithm in C++ and CUDA programming language [NVIDIA 2022a]. We use the cuFFT library [NVIDIA 2022b] to perform FFT on the GPU. Unless otherwise specified, all experiments use a tray of dimension 480 mm × 245 mm × 200 mm, which is the maximum print volume of our 3D printer.

5.1 Benchmark

To evaluate our algorithm comprehensively, we produce a large benchmark from Thingi10K [Zhou and Jacobson 2016]. The exact details are discussed in §B.1. Additionally, we produced three other benchmarks for evaluation. The first benchmark, PIECES, contains many medium and small objects, many of which are axis-aligned. The second dataset, MIXTURES, contains a mixture of large and small objects selected from Thingi10K. Both of the benchmarks represent typical packing scenarios. The third dataset, LOCKS, contains many ring-like geometries to test the robustness of the algorithm against interlocking. A preview of the benchmarks, as well as packing using our algorithm, is shown in Fig. 7 and Fig. 8. We also 3D printed the packed MIXTURES dataset as shown in Fig. 9.

5.2 Different Strategies to Avoid Interlocking

We compare two strategies for avoiding interlocking, namely flood-fill at the placement stage versus disassembly as a post-processing step. The result is shown in Table 1. Without checking interlocking, we may achieve a higher packing density, but interlocking can easily appear as shown in Fig. 10. Generally, the two-phase post-processing disassembly packs objects more densely and in less time. Therefore, we choose it as the default method.

Enabling flood-fill at the placement stage requires more steps as shown in §4.3, which makes the overall algorithm slower. This is shown in the PIECES and MIXTURES dataset of Table 1. An exception is the LOCKS benchmark, where the flood-fill placement is faster. For this benchmark, the default placement algorithm usually produces large interlocking assemblies such that the ray-casting disassembly is less effective. The more expensive flood-fill disassembly has to be used to examine large interlocking assemblies, which becomes slower as the number of objects in the assembly grows.

Table 1. We compare the packing densities and timings of our packing algorithm using different disassembly methods. "No" means using no disassembly, and therefore the packing may contain interlocking. "Flood-Fill" uses the interlocking-free placement in §4.3 for every orientation at the placement stage. "Post" is using the disassembly as a post-process, where the ray-casting and flood-fill disassembly are combined in §4.

Dataset	Disassembly	2 mm 90°	Timing	1 mm 90°	Timing
PIECES	No	50.48%	104.05s	51.11%	427.77s
	Flood-Fill	48.14%	421.81s	49.12%	1879.54s
	Post	50.45%	102.64s	50.48%	475.0s
MIXTURES	No	36.06%	40.51s	41.62%	178.50s
	Flood-Fill	29.05%	177.54s	34.97%	838.75s
	Post	34.71%	67.99s	39.77%	408.05s
LOCKS	No	8.25%	14.50s	9.79%	62.63s
	Flood-Fill	6.91%	71.47s	7.59%	346.56s
	Post	6.89%	123.84s	8.67%	469.17s

5.3 Continuous Refinement

We demonstrate the efficacy of continuous refinements on three benchmarks. The result is shown in Table 2. Using refinements consistently improves packing density at a modest increase of timing. At 2 mm voxel resolution, enabling refinements achieves a similar packing density as using 1mm voxel resolution, while having a much lower computational cost. As shown in Fig. 11, continuous refinement saves a small amount of space for each object insertion with FFT. This accumulates and allows some objects to squeeze into new spaces, leading to different, denser packing configurations than without refinement.

5.4 Multi-Tray Packing

In this example, we pack all 6596 meshes into a number of trays using Algorithm 5. We limit the number of objects to pack for each tray

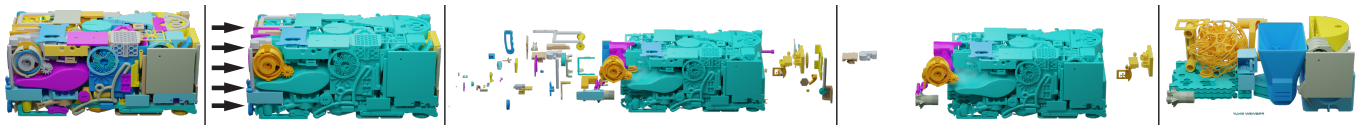

Fig. 6. Ray-casting disassembly. From left to right: Starting with a packed tray, we cast rays along the x direction shown with the black arrow. A DBG is obtained, where SCCs are rendered with the same color. Then, every SCCs with only one object are disassembled from the packing, as shown in the third and fourth figure. This broad-phase disassembly is repeated several times along different directions. The remaining SCCs are shown in the fifth figure.

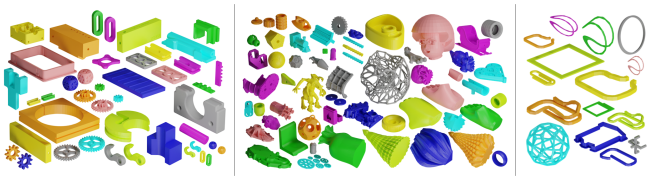

Fig. 7. Our benchmark data sets. Left: meshes in the PIECES data set. Middle: a selection of meshes from the MIXTURES data set. Right: Meshes in the LOCKS data set.

Fig. 8. Example packs obtained from three benchmarks. **Left**: A packing of PIECES. **Middle**: A packing of MIXTURES. **Right**: A packing of LOCKS.

Table 2. Comparison of our packing algorithm with and without continuous refinement. Post-processing disassembly is used for all experiments.

Dataset	Continuous Refinement	2 mm 90°	Timing	1 mm 90°	Timing
PIECES	No	48.41%	78.50s	50.45%	402.96s
	Yes	50.45%	102.64s	53.32%	475.00s
MIXTURES	No	34.19%	57.50s	36.51%	384.20s
	Yes	34.71%	67.99s	39.77%	408.05s
LOCKS	No	6.86%	100.30s	8.24%	334.24s
	Yes	6.89%	123.84s	8.67%	469.17s

to 1000. We use 1 mm voxel size and 90° angle sampling. Interlock checking and refinement are enabled as well. We show the results of multi-tray packing in Fig. 13. The timing of this example is shown in the third row of Table 3.

To scale up the evaluation, we pack these meshes into a single tray with 2 mm voxel size and 90° angle sampling. Without interlock checking, the benchmark is packed at a density of 40.28% in a bounding box of 1430 mm × 730 mm × 538 mm. With interlock checking, the benchmark is packed at 35.77% density with a bounding box of 1430 mm × 730 mm × 597 mm (see the teaser figure). Timing for these examples is shown in Table 3 as well.

5.5 Parameter Choices

We study our algorithm with different parameter choices, especially the height penalization coefficient. As shown in Fig. 12, when height penalization is set to zero, the proximity metric will try to find placements that best fit existing objects, resulting in a taller pack. With a non-zero penalization, the placement method will prioritize a lower height for each object. We also study the result of searching over different numbers of orientations in §B.2. Experiments on placement order and combinatorial strategies are also shown in §B.3. The results of packing in irregular containers in Fig. 14, with the implementation details in §B.4.

5.6 Comparison

Comparison with Commercial Software. We compare our algorithm with commercial software Netfabb [Autodesk 2023], Fabpilot [Sculpteo 2023], and Polydevs [UnionTech 2018] on our benchmarks. We select the best settings from each software, which are shown in §B.5. Table 4 shows that our algorithm achieves the highest packing densities and runs the fastest on all three benchmarks. A visualization of packing on PIECES is shown in Fig. 15. Our algorithm results in tighter margins between objects and places small objects into gaps more efficiently. For the LOCKS dataset, our result is both interlocking-free and denser. No interlocking is found in the result of Netfabb and Polydevs, but we found many interlocking instances in the results of Fabpilot. We also provide timing for a CPU implementation of our algorithm that achieves competitive speeds despite the lack of optimization. Compared to our GPU version, the CPU implementation loses more performance during disassembly where FFT and flood-fill are used extensively.

Comparison with Other Techniques. We compare our algorithm with four prior techniques that provided datasets or source code. PackMerger [Vanek et al. 2014] decomposes a given shape into a small number ($\leq$ 20) of segments and then packs them. It uses a heightmap-based packing and a tabu search to minimize the bounding box volume. We compare packing performance on decomposed segments produced by PackMerger. We enforce the same 1 mm separation between objects as in their experiment. Our packing method improves density significantly even with a simple greedy ordering, whereas PackMerger used a tabu search (Fig. 16). For this small-scale problem, our algorithm remains efficient, averaging 1-2 seconds per object. We also evaluate our algorithm on 80 polyhedra from Example 5 of Romanova et al. [2018] (Fig. 17 left). Our algorithm is two orders of magnitude faster at a slightly lower packing density. We pack 77 polyhedra to compare with Ma et al. [2018] in Fig. 13 of their paper (Fig. 17 right). Our algorithm is one order of magnitude

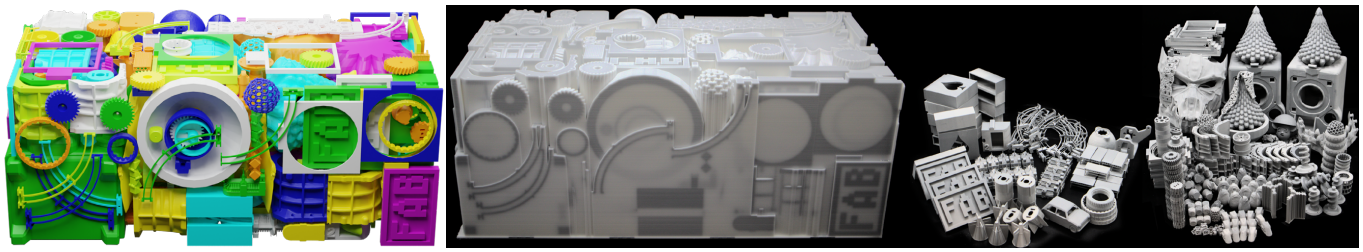

Fig. 9. **Left**: A packing with a density 34.25% produced by our algorithm. **Middle**: The packed tray printed with an inkjet printer. **Right** The disassembled parts.

Table 3. The packing statistics on the Thingi benchmark. The benchmark consists of 6596 meshes. The benchmark is packed in multiple trays of dimensions 480 mm×245 mm×200 mm. †: Packing of dimensions 1430 mm×730 mm×597 mm with interlock checking. ‡: Packing of dimensions 1430 mm×730 mm×538 mm without interlock checking. All timing is in seconds.

Setting	Interlock Checking	Number of Trays	Packing Density	Total Time	Packing	Disassembly	Re-Insert	FFT	Voxelization	Collision Detection
2 mm/90°	Yes	28	35.06%	1456.03s	1207.95s	90.16s	157.91s	34.64%	38.66%	7.84%
	No	26	37.77%	1010.27s	1010.27s	0s	0s	32.69%	39.58%	7.71%
1 mm/90°	Yes	26	37.30%	7212.97s	5606.00s	504.99s	1101.98s	55.29%	24.25%	2.18%
	No	24	40.64%	5103.02s	5103.02s	0s	0s	54.88%	23.59%	2.46%
2mm/90°	Yes	1†	35.77%	14478.80s	6595.30s	2524.58s	5358.87s	71.33%	17.35%	0.64%
	No	1‡	40.28%	6662.98s	6662.98s	0s	0s	82.20%	1.50%	1.41%

Fig. 10. **Left**: Locks packed without interlock checking, notice significant number of interlocked parts as highlighted on the bottom. **Right**: Locks packed with interlock checking, notice parts are free of interlocking, as highlighted on the bottom.

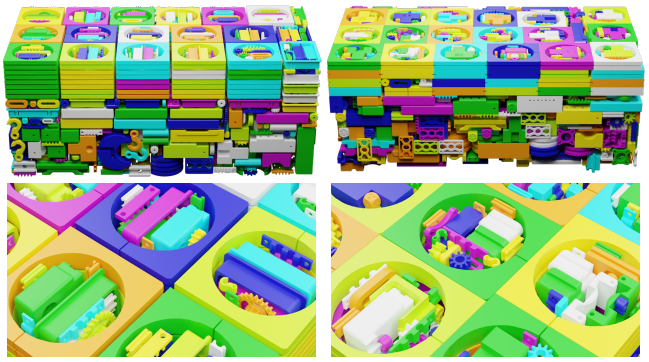

Fig. 11. **Left**: Pieces packed without refinements. Notice the gaps between parts due to voxelization. **Right**: Pieces packed with refinements. The gaps are much smaller with continuous refinements, as highlighted on the bottom. This leads to higher packing density with very little overhead.

faster and achieves a slightly higher packing density. For this example, we use a voxel size of 0.25 and uniformly sample Euler angles at 30°. Continuous refinement is enabled and the margin is set to zero between parts. Pack3D [Fogleman 2019] is an open source 3D packing software. We compare with it on mixtures in §B.6.

6 CONCLUSION AND FUTURE WORK

We have introduced a scalable packing algorithm that packs thousands of generic 3D objects densely and without interlocking. Our method highlights the importance of using the objects' detailed geometries without oversimplifying in order to identify a richer set of placements. By using high-resolution voxelization and computing correlations in the spectral domain, we can efficiently place each object into a pack. This approach is extended to guarantee that the objects can be disassembled in the context of 3D printing.

In this work, we only consider interlocking-free packing. In practice, many other physical constraints can be added, such as robotic motion planning [LaValle 2006] and packing stability [Wang and Hauser 2019]. Another interesting direction is to pack articulated or deformable objects. Finally, a better combinatorial search algorithm could substantially improve packing density.

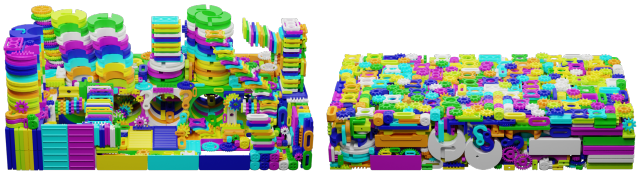

Fig. 12. **Left**: 1500 parts from PIECES packed with zero height penalization, which has a taller total height. **Right**: The same set of objects packed with height penalization $p = 10^8$, which results in a lower total height.

Table 4. Comparison of packing densities between our algorithm and other software packages. We use 2 mm voxel size, 90° orientation sampling, and post-processing disassembly for all examples. Continuous refinement is also enabled. †: CPU timing is recorded on the same platform. ‡: Timing reported on the cloud platform. We were unable to pack MIXTURES on Fabpilot due to upload limits. We report both GPU and CPU timing of our algorithm.

	Methods	PIECES	MIXTURES	LOCKS
Density	Netfabb†	28.23%	25.48%	3.46%
	Fabpilot‡	35.59%	-	4.56%
	PolyDevs	40.87%	23.16%	3.13%
	Ours	50.45%	34.71%	6.89%
Timing	Netfabb†	1810s	1560s	610s
	Fabpilot‡	684s	-	241s
	PolyDevs	3598s	1180s	184s
	Ours (GPU)	103s	68s	124s
	Ours (CPU)	313s	342s	425s

ACKNOWLEDGMENTS

This work was conducted at Inkbit LLC and is part of Inkbit's patented and patent pending commercial 3d printing solutions. We thank the anonymous reviewers for their helpful comments in revising the paper.

REFERENCES

Luiz JP Araújo, Ender Özcan, Jason AD Atkin, and Martin Baumers. 2019. Analysis of irregular three-dimensional packing problems in additive manufacturing: a new taxonomy and dataset. *International Journal of Production Research* 57, 18 (2019), 5920–5934.

Richard Carl Art Jr. 1966. *An approach to the two dimensional irregular cutting stock problem.* Ph. D. Dissertation. Massachusetts Institute of Technology.

Autodesk. 2023. Autodesk Netfabb. https://www.autodesk.com/products/netfabb/Build: 47, Release: 2023.1.

Edmund K Burke, Robert SR Hellier, Graham Kendall, and Glenn Whitwell. 2007. Complete and robust no-fit polygon generation for the irregular stock cutting problem. *European Journal of Operational Research* 179, 1 (2007), 27–49.

Elizabeth R Chen, Michael Engel, and Sharon C Glotzer. 2010. Dense crystalline dimer packings of regular tetrahedra. *Discrete & Computational Geometry* 44, 2 (2010), 253–280.

Rulin Chen, Ziqi Wang, Peng Song, and Bernd Bickel. 2022. Computational design of high-level interlocking puzzles. *ACM Trans. Graph.* 41, 4 (2022), 1–15.

Xuelin Chen, Hao Zhang, Jinjie Lin, Ruizhen Hu, Lin Lu, Qi-Xing Huang, Bedrich Benes, Daniel Cohen-Or, and Baoquan Chen. 2015. Dapper: decompose-and-pack for 3d printing. *ACM Trans. Graph.* 34, 6 (2015), 213–1.

Nikolai Chernov, Yuriy Stoyan, and Tatiana Romanova. 2010. Mathematical model and efficient algorithms for object packing problem. *Computational Geometry* 43, 5 (2010), 535–553.

Erwin Coumans and Yunfei Bai. 2016–2021. PyBullet, a Python module for physics simulation for games, robotics and machine learning. http://pybullet.org.

Jens Egeblad, Claudio Garavelli, Stefano Lisi, and David Pisinger. 2010. Heuristics for container loading of furniture. *European Journal of Operational Research* 200, 3 (2010), 881–892.

Christer Ericson. 2004. *Real-time collision detection.* Crc Press.

Filippo Andrea Fanni, Fabio Pellacini, Riccardo Scateni, and Andrea Giachetti. 2022. PAVEL: Decorative Patterns with Packed Volumetric Elements. *ACM Transactions on Graphics (TOG)* 41, 2 (2022), 1–15.

Michael Fogleman. 2019. *Pack3d.* https://github.com/fogleman/pack3d

Michael Garland and Paul S Heckbert. 1997. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques.* 209–216.

Michael Goldwasser, J-C Latombe, and Rajeev Motwani. 1996. Complexity measures for assembly sequences. In *Proceedings of IEEE International Conference on Robotics and Automation*, Vol. 2. IEEE, 1851–1857.

Ankit Goyal and Jia Deng. 2020. PackIt: A Virtual Environment for Geometric Planning. In *International Conference on Machine Learning.*

Si Hang. 2015. TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw* 41, 2 (2015), 11.

Ruizhen Hu, Juzhan Xu, Bin Chen, Minglun Gong, Hao Zhang, and Hui Huang. 2020. TAP-Net: transport-and-pack using reinforcement learning. *ACM Trans. Graph.* 39, 6 (2020), 1–15.

Alec Jacobson, Daniele Panozzo, et al. 2018. libigl: A simple C++ geometry processing library. https://libigl.github.io/.

David S Johnson. 1973. *Near-optimal bin packing algorithms.* Ph. D. Dissertation. Massachusetts Institute of Technology.

Ephraim Katchalski-Katzir, Isaac Shariv, Miriam Eisenstein, Asher A Friesem, Claude Aflalo, and Ilya A Vakser. 1992. Molecular surface recognition: determination of geometric fit between proteins and their ligands by correlation techniques. *Proc Natl Acad of Sci* 89, 6, 2195–2199.

Carlos Lamas-Fernandez, Julia A Bennell, and Antonio Martinez-Sykora. 2022. Voxel-Based Solution Approaches to the Three-Dimensional Irregular Packing Problem. *Operations Research* (2022).

Lei Lan, Danny M Kaufman, Minchen Li, Chenfanfu Jiang, and Yin Yang. 2022. Affine body dynamics: Fast, stable & intersection-free simulation of stiff materials. *arXiv preprint arXiv:2201.10022* (2022).

Steven M LaValle. 2006. *Planning algorithms.* Cambridge university press.

Aline AS Leao, Franklina MB Toledo, José Fernando Oliveira, Maria Antónia Carravilla, and Ramón Alvarez-Valdés. 2020. Irregular packing problems: A review of mathematical models. *European Journal of Operational Research* 282, 3 (2020), 803–822.

Xiao Liu, Jia-min Liu, An-xi Cao, and Zhuang-le Yao. 2015. HAPE3D—a new constructive algorithm for the 3D irregular packing problem. *Frontiers of Information Technology & Electronic Engineering* 16, 5 (2015), 380–390.

Yuexin Ma, Zhonggui Chen, Wenchao Hu, and Wenping Wang. 2018. Packing irregular objects in 3D space via hybrid optimization. *Comp. Graph. Forum (SGP)* 37, 5 (2018), 49–59.

Silvano Martello, David Pisinger, and Daniele Vigo. 2000. The three-dimensional bin packing problem. *Operations research* 48, 2 (2000), 256–267.

NVIDIA. 2022a. CUDA, release: 12.0. https://developer.nvidia.com/cuda-toolkit

NVIDIA. 2022b. CUDA, release: 12.0. https://developer.nvidia.com/cufft

Dzmitry Padhorny, Andrey Kazennov, Brandon S Zerbe, Kathryn A Porter, Bing Xia, Scott E Mottarella, Yaroslav Kholodov, David W Ritchie, Sandor Vajda, and Dima Kozakov. 2016. Protein–protein docking by fast generalized Fourier transforms on 5D rotational manifolds. *Proc Natl Acad Sci* 113, 30, E4286–E4293.

Alexander Pankratov, Tatiana Romanova, and Igor Litvinchev. 2020. Packing oblique 3D objects. *Mathematics* 8, 7 (2020), 1130.

Bernhard Reinert, Tobias Ritschel, and Hans-Peter Seidel. 2013. Interactive by-example design of artistic packing layouts. *ACM Transactions on Graphics (TOG)* 32, 6 (2013), 1–7.

David W Ritchie and Graham JL Kemp. 2000. Protein docking using spherical polar Fourier correlations. *Proteins: Structure, Function, and Bioinformatics* 39, 2 (2000), 178–194.

David W Ritchie and Vishwesh Venkatraman. 2010. Ultra-fast FFT protein docking on graphics processors. *Bioinformatics* 26, 19 (2010), 2398–2405.

Tatiana Romanova, Julia Bennell, Yuriy Stoyan, and Aleksandr Pankratov. 2018. Packing of concave polyhedra with continuous rotations using nonlinear optimisation. *European Journal of Operational Research* 268, 1 (2018), 37–53.

Michael Schwarz and Hans-Peter Seidel. 2010. Fast parallel surface and solid voxelization on GPUs. *ACM transactions on graphics (TOG)* 29, 6 (2010), 1–10.

Sculpteo. 2023. Sculpteo Fabpilot. https://www.fabpilot.com/.

Pitchaya Sitthi-Amorn, Javier E Ramos, Yuwang Wangy, Joyce Kwan, Justin Lan, Wenshou Wang, and Wojciech Matusik. 2015. MultiFab: a machine vision assisted platform for multi-material 3D printing. *Acm Transactions on Graphics (Tog)* 34, 4 (2015), 1–11.

Peng Song, Chi-Wing Fu, and Daniel Cohen-Or. 2012. Recursive interlocking puzzles. *ACM Trans. Graph.* 31, 6 (2012), 1–10.

Yuriy Stoyan, Aleksandr Pankratov, and Tatiana Romanova. 2016. Quasi-phi-functions and optimal packing of ellipses. *Journal of Global Optimization* 65, 2 (2016), 283–307.

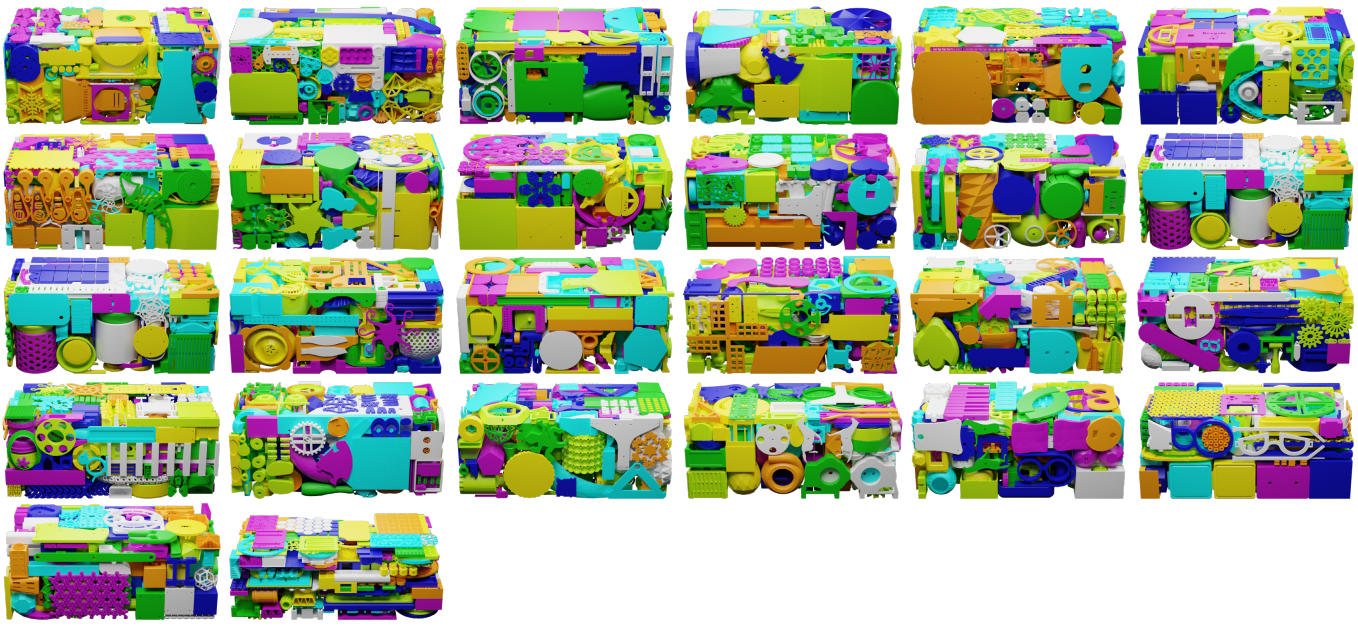

Fig. 13. We packed 6596 objects from the Thingi10K benchmark into 26 trays.

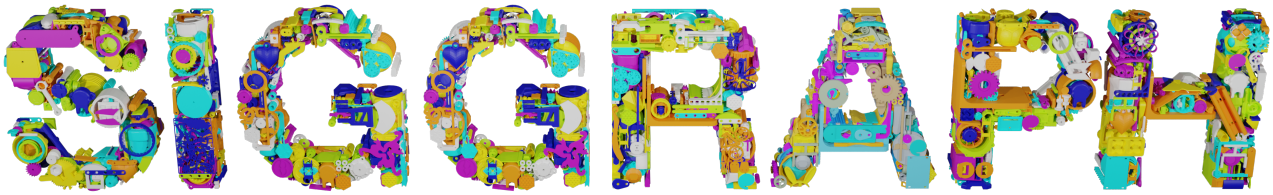

Fig. 14. We packed the MIXTURES benchmark into SIGGRAPH letters.

Fig. 15. Comparison between our algorithm and commercial software on PIECES. **Left**: Netfabb. **Middle**: Fabpilot. **Right**: Our algorithm. The bottom row shows in-focus views of the result. Both Netfabb and Fabpilot produce wide gaps between objects as highlighted with red dashed line. Also notice our algorithm is able to fill void space more efficiently as shown in the black square.

Yunsheng Tian, Jie Xu, Yichen Li, Jieliang Luo, Shinjiro Sueda, Hui Li, Karl DD Willis, and Wojciech Matusik. 2022. Assemble Them All: Physics-Based Planning for Generalizable Assembly by Disassembly. *ACM Transactions on Graphics (TOG)* 41, 6 (2022), 1–11.

UnionTech. 2018. Polydevs. https://polydevs3d.com/ Version 2.2.5.33.

Juraj Vanek, JA Garcia Galicia, Bedrich Benes, R Měch, N Carr, Ondrej Stava, and GS Miller. 2014. Packmerger: A 3d print volume optimizer. In *Computer Graphics Forum*, Vol. 33. Wiley Online Library, 322–332.

Fan Wang and Kris Hauser. 2019. Stable bin packing of non-convex 3D objects with a robot manipulator. In *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 8698–8704.

Ziqi Wang, Peng Song, and Mark Pauly. 2018. DESIA: A general framework for designing interlocking assemblies. *ACM Transactions on Graphics (TOG)* 37, 6 (2018), 1–14.

Ziqi Wang, Peng Song, and Mark Pauly. 2021. State of the Art on Computational Design of Assemblies with Rigid Parts. *Eurographics 2021 STAR* 40 (2021).

Randall H Wilson and Jean-Claude Latombe. 1994. Geometric reasoning about mechanical assembly. *Artificial Intelligence* 71, 2 (1994), 371–396.

Zifei Yang, Shuo Yang, Shuai Song, Wei Zhang, Ran Song, Jiyu Cheng, and Yibin Li. 2021. PackerBot: Variable-Sized Product Packing with Heuristic Deep Reinforcement Learning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 5002–5008.

Miaojun Yao, Zhili Chen, Linjie Luo, Rui Wang, and Huamin Wang. 2015. Level-set-based partitioning and packing optimization of a printable model. *ACM Transactions on Graphics (TOG)* 34, 6 (2015), 1–11.

Cha Zhang and Tsuhan Chen. 2001. Efficient feature extraction for 2D/3D objects in mesh representation. In *Proceedings 2001 International Conference on Image Processing (Cat. No. 01CH37205)*, Vol. 3. IEEE, 935–938.

Xinya Zhang, Robert Belfer, Paul G Kry, and Etienne Vouga. 2020. C-Space tunnel discovery for puzzle path planning. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 104–1.

Hongkai Zhao. 2005. A fast sweeping method for eikonal equations. *Mathematics of computation* 74, 250 (2005), 603–627.

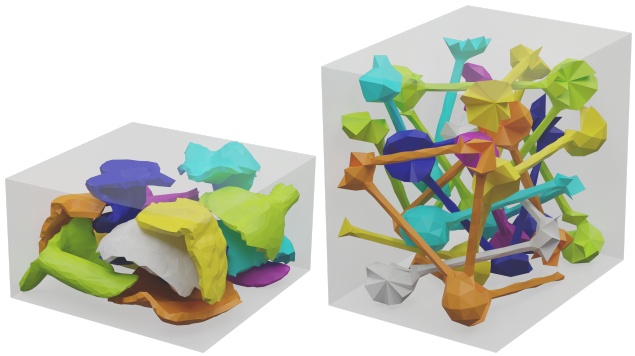

Fig. 16. Our packed examples corresponding to Figures 1 and 11 of [Vanek et al. 2014]. **Left**: We pack 13 bunny parts into a bounding box of volume 2.90×10^5 mm^3, a 27.8% improvement over Vanek et al. [2014]'s result of 4.01×10^5 mm^3. This packing takes 19.2s. **Right**: We pack 12 molecule parts into a bounding box of volume 2.09×10^5 mm^3, a 43.0% improvement over Vanek et al. [2014]'s result of 3.67×10^5 mm^3. This packing takes 27.4s.

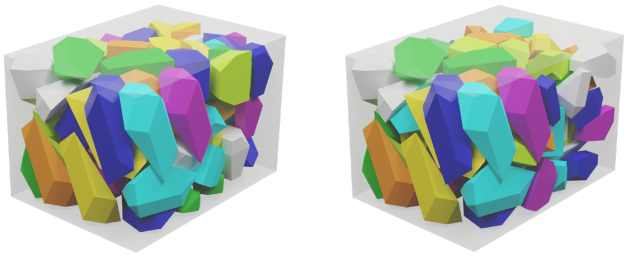

Fig. 17. Our packing of Example 5 in [Romanova et al. 2018]. **Left**: We pack 80 polyhedrons inside a cube of size $36.98 \times 41.98 \times 56.37$, with a packing density of 51.27%, and 329s. Romanova et al. [2018] achieved a packing density of 53.66% with 42950s. **Right**: We pack 77 polyhedrons inside a cube of size $36.98 \times 40.98 \times 56.55$, with a packing density of 51.53%, and 335s. Ma et al. [2018] achieved a packing density of 51.3% with 2700s.

Hang Zhao, Qijin She, Chenyang Zhu, Yin Yang, and Kai Xu. 2021. Online 3D bin packing with constrained deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 741–749.

Qingnan Zhou and Alec Jacobson. 2016. Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797* (2016).

A PACKING ALGORITHM IMPLEMENTATION

In this section, we present further details about the packing algorithm's implementation.

A.1 Metrics Computation and Discussion

We use a fixed grid to store the indicator function s_Ω. Instead of re-voxelizing the entire grid for every object placement, we update s_Ω with the object voxelization s_A, should it be placed into the tray successfully. To compute the proximity metric, we allocate another grid and initialize it with ∞, then mark every voxel where $s_\Omega = 1$ with 0. Based on this grid, we compute ϕ_Ω using the Fast Sweeping Method [Zhao 2005]. We compute the Manhattan distance instead of the Euclidean distance, and find it to be both faster and easier to implement on GPUs, while yielding reasonable results. We use circular convolutions to calculate both metrics. This allows us to use fixed voxel grids and FFT plans [NVIDIA 2022b] that can be pre-allocated with the size of tray dimension $\mathbf{l}$ divided by voxel resolution dx. In contrast, computing zero-padded convolutions will require padding the voxel grid of s_Ω with the size of voxel grid s_A. This requires reallocating the voxel grid as well as resizing FFT plans for every object orientation, which can incur significant overhead. (Note that computing zero-padded convolution becomes necessary with flood-fill disassembly, which is discussed in §4.1.) We then discard any voxel location that the object clips out of the tray.

Katchalski-Katzir et al. [1992] combined collision detection and proximity search into a single metric, which is achieved by setting a penalty term on $\phi_\Omega(\mathbf{x})$ where $s_\Omega(\mathbf{x}) = 1$. With a small penalization term, overlaps between objects can still occur. With a very large penalization term, the sharp transition of ϕ_Ω at the boundary of Ω introduces ringing artifacts on the results of FFT due to numerical precision limits. In contrast, we compute collision metric and proximity metric separately and avoid using penalty terms for collision. This improves the numerical accuracy because the input to FFT is smoother.

A.2 Choosing Height Penalization

We use a simple heuristic to determine the penalization coefficient p by estimating how likely the tray will be fully filled. We add up the volume of bounding boxes of all objects that need to be packed, and divide the result by the volume of the tray. If the ratio is larger than 1.2, the tray is likely to be fully filled, and so we use a small penalization $p = 4$. Otherwise, we use a large penalization $p = 10^8$ to minimize the total height.

A.3 Greedy and Multi-Tray Packing

We extend our algorithm to multi-tray packing with a simple greedy approach. Given a large list of objects and assuming each of them can fit into a tray, we select a number of objects to attempt for each tray and pack them with Algorithm 2. Then, we remove the objects that are successfully packed from the list, and proceed to the next empty tray. This process is repeated until all objects are packed, as described in Algorithm 5.

B ADDITIONAL RESULTS

In this section we present additional results and clarifications of experiments in the main paper.

B.1 Benchmark Generation

To generate our benchmark, we extract meshes from Thingi10K dataset that are both manifold and consistently-oriented. We compute the mesh volume with the signed volume [Zhang and Chen 2001] and prune any meshes with a negative volume. Note that some of the meshes we retain have multiple components which can intersect and so the signed volumes in such cases may not be completely accurate. The packing density is computed by adding up the signed volume of all meshes contained in the tray and dividing that by the volume of the bounding box. We use quadratic mesh simplification [Garland and Heckbert 1997] on the extracted meshes to reduce the data size without any impact on the packing. Then,

Algorithm 5 Greedy Multi-Tray Packing

function MULTITRAYPACKING($\mathfrak{A}$, n, m, $\mathbf{l}$, dx)
  Input: A large list of objects $\mathfrak{A}$, maximum number of objects to attempt for each tray n, the number of orientations to search m, tray dimension $\mathbf{l}$, voxel resolution dx
  Output: A list of packed trays $\mathfrak{P}$, a list of translations $\mathfrak{Q}$, a list of rotations $\mathfrak{R}$
  $\mathfrak{P} \leftarrow \emptyset, \mathfrak{Q} \leftarrow \emptyset, \mathfrak{R} \leftarrow \emptyset$
  while $\mathfrak{A} \neq \emptyset$ **do**
    $\mathbf{A} \leftarrow$ SELECT($\mathfrak{A}$, n) ▷ Select most n objects to attempt
    $\{\mathbf{P}, \mathbf{Q}, \mathbf{R}, \mathbf{U}\} \leftarrow$ GREEDYPACKOBJECTS($\mathbf{A}$, m, $\mathbf{l}$, dx)
    $\mathfrak{A}$.remove($\mathbf{P}$)
    $\mathfrak{P}$.add($\mathbf{P}$)
    $\mathfrak{Q}$.add($\mathbf{Q}$)
    $\mathfrak{R}$.add($\mathbf{R}$)
  end while
  Return $\{\mathfrak{P}, \mathfrak{Q}, \mathfrak{R}\}$;
end function

we uniformly scale down any meshes that have a bounding box larger than $480 \times 245 \times 200$, and scale up any meshes that are smaller than $2 \times 2 \times 2$. This results in a benchmark containing 6596 meshes, which consists of many diverse and challenging geometries.

B.2 Placement Orientation Search

Sampling more orientations allows for more options per object and a better local optimum. It is reasonable to assume this would lead to better overall packing density. As shown in Table 5, sampling Euler angles at 30° and 45° results in a moderate increase in packing density in the MIXTURES and LOCKS dataset. However, this is not always the case. As shown in Table 5, packing PIECES by searching over 30° produces a worse result than searching over 90°. This is because most objects in PIECES are axis aligned. Even though a search over 30° may result in a better solution for a partial tray packing than searching over 90°, it may cause subsequent objects to be misaligned, resulting in a worse final packing. This counter-intuitive result is visualized in Fig. 18.

Table 5. Comparisons of packing density and timing on three datasets with different number of orientation sampling. A voxel size of 2 mm and post-processing disassembly are used for all the following experiments.

Dataset	Angle Sample	PIECES	MIXTURES	LOCKS
Density	90°	50.48%	34.71%	6.89%
	45°	50.02%	35.27%	7.88%
	30°	45.39%	35.27%	7.57%
Timing	90°	104.05s	67.99s	123.84s
	45°	603.28s	315.97s	226.19s
	30°	2099.65s	1052.08s	429.57s

B.3 Placement Order and Combinatorial Search

As our algorithm is greedy, the order of objects selected for placements will affect the results. We compare different selection strategy with the benchmark MIXTURE. As shown in Fig. 19, selecting objects at random produces sub-optimal results compared to selecting objects from largest to smallest. We also show a result of beam search over placement ordering with a beam width of 5. While the packing density is slightly higher, it is 4× slower.

Fig. 18. **Left**: PIECES packed by evenly searching 30° over Euler angles. It produces a sub-optimal result with a density of 44.91%. Notice misaligned parts as highlighted. **Right**: The same dataset packed by only searching 90°. This results in a better density of 50.81% where parts are axis aligned.

Fig. 19. **Left**: Placing parts in random order produces sub-optimal result with a density of 29.9% **Middle**: Placing parts from largest to smallest produces a better packing density of 34.71% with the same computational cost. **Right**: Beam search results in a packing density of 35.06%, but spends 4× more time.

B.4 Packing in Irregular Containers

Our algorithm can be easily extended to packing in irregular containers. In this case, the geometry of the container is voxelized to initialize the grid for the placement computation. In the example, we pack the MIXTURES benchmark into SIGGRAPH letters. We use a fixed tray size of 322 mm × 322 mm × 80 mm and a voxel size of 1 mm for all the letters. Interlock checking is disabled and only one orientation is attempted. Packing each letter takes on average 12.73 seconds.

B.5 Comparison Settings

We clarify the settings used for the comparison with commercial software. For Netfabb [Autodesk 2023], we choose size sorting 3D packing, which is its best-performing setting on our benchmark. The algorithm is a 3-phase packing algorithm. In the first two phases the algorithm packs large and medium-sized objects with its built-in Monte Carlo packer. In the final phase, the algorithm uses the scan-line packer to pack small objects. We use highest-density/highest-quality settings for all three phases, and 1 mm voxel size (the highest quality setting) for the third phase. We also enable avoiding interlocking option in Netfabb. For Fabpilot [Sculpteo 2023], area averaging packing is disabled to produce the best overall results. However, there is no interlock detection feature. For Polydevs [UnionTech 2018], we use high accuracy setting with 1 mm (lowest) part interval and 30° orientations for both MIXTURE and LOCKS dataset. For

Fig. 21. **Top left**: The same set of objects in Fig. 20 are packed with our algorithm with no interlocking. **Top right**: The printed tray containing the set of objects on the left. **Bottom**: The disassembled printed objects.

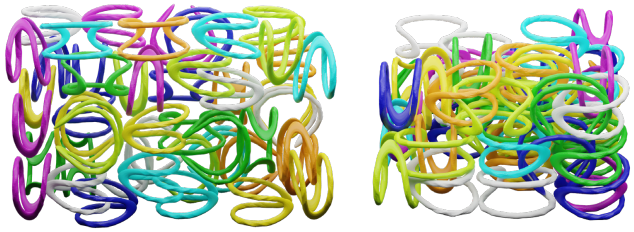

Fig. 22. The packing of 48 distorted tori with our algorithm and pack3d. **Left**: The result of pack3d. While there is no interlocking, the packing density is 3.96% with bounding box size 156.49 mm × 339.11 mm × 220.49 mm. This example takes 1200s. **Right**: The result of our algorithm, which is both interlocking-free and with a higher packing density of 5.82%. Our result takes 32s, with bounding box size 239.98 mm × 239.98 mm × 138.13 mm.

Fig. 23. **Left**: The packing result of MIXTURE with pack3d, where it fails to converge with a packing density of 3.47% after 1200s. **Right**: Our algorithm achieves a packing density of 34.71% in only 67.99s.

PIECES, we use low accuracy setting with 90° orientations to avoid excessive run-time.

We set the same build volume (480 mm × 245 mm × 200 mm), select the same objects and the number of copies for each object to pack. After packing, we remove any objects that are outside of the build volume and compute the packing density. We run Netfabb and Polydevs on the same platform as ours and record the timing, though they both only use CPU for packing. Fabpilot is cloud-based and we are unable to determine the hardware platform. We report the timing when it converged.

B.6 Additional Comparisons

We used Fabpilot to pack a set of ring-like objects. As shown in Fig. 20, the result contains many interlocked assemblies. We prepared the same set of objects with our algorithm and printed on an inkjet deposition 3D printer (Fig. 21). The parts are successfully disassembled without interlocking.

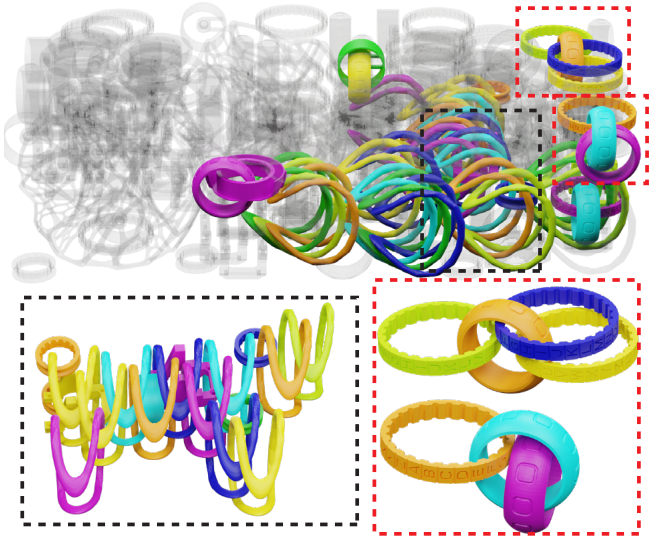

Fig. 20. A packing produced by Fabpilot with multiple interlocking assemblies highlighted in the red box and the black box.

We also compare our algorithm with an open source 3d packing software pack3D [Fogleman 2019] which uses a simulated-annealing-based approach. It first produces a collision-free initial configuration by spreading out all the objects. Then it locally optimizes one object at a time by updating its position and orientation, in order to minimize the volume of the bounding box. The software uses bounding volume hierarchies (BVH) to detect collisions between objects, and is able to achieve reasonable results for a small number of objects. We compare our algorithm with pack3d by packing 48 ring-like objects. The results are in Fig. 22. Our algorithm is able to achieve higher packing density with less time. Furthermore, as pack3d only optimizes one object at a time, it is not scalable to a large set of objects. As shown in Fig. 23, it fails to converge while packing the MIXTURES benchmark.