Computational Design of High-level Interlocking Puzzles

RULIN CHEN, Singapore University of Technology and Design, Singapore
ZIQI WANG, EPFL, Switzerland and ETH Zürich, Switzerland
PENG SONG, Singapore University of Technology and Design, Singapore
BERND BICKEL, IST, Austria

1 INTRODUCTION

Interlocking puzzles are intriguing geometric games where the puzzle pieces are held together based on their geometric arrangement, preventing the puzzle from falling apart. High-level-of-difficulty, or simply high-level, interlocking puzzles are a subclass of interlocking puzzles that require multiple moves to take out the first subassembly from the puzzle. Solving a high-level interlocking puzzle is a challenging task since one has to explore many different configurations of the puzzle pieces until reaching a configuration where the first subassembly can be taken out. Designing a high-level interlocking puzzle with a user-specified level of difficulty is even harder since the puzzle pieces have to be interlocking in all the configurations before the first subassembly is taken out.

In this paper, we present a computational approach to design high-level interlocking puzzles. The core idea is to represent all possible configurations of an interlocking puzzle as well as transitions among these configurations using a rooted, undirected graph called a disassembly graph and leverage this graph to find a disassembly plan that requires a minimal number of moves to take out the first subassembly from the puzzle. At the design stage, our algorithm iteratively constructs the geometry of each puzzle piece to expand the disassembly graph incrementally, aiming to achieve a user-specified level of difficulty. We show that our approach allows efficient generation of high-level interlocking puzzles of various shape complexities, including new solutions not attainable by state-of-the-art approaches.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2022 Association for Computing Machinery.
0730-0301/2022/07-ART150 $15.00
https://doi.org/10.1145/3528223.3530071

ACM Trans. Graph., Vol. 41, No. 4, Article 150. Publication date: July 2022.
In this paper, we develop a computational approach for designing new high-level interlocking puzzles according to user specifications, including the puzzle shape, number of puzzle pieces, and level of difficulty. To make the problem tractable, we assume that the puzzle shape is represented as a voxelized model, and disassembly motion is limited to translation along each of the major axes. Our core idea to address the problem is to encode all possible configurations of a given puzzle using a rooted graph data structure, where the level of difficulty corresponds to the length of a shortest path from the root node (i.e., assembled puzzle configuration) to a target node (i.e., configuration where a single subassembly is removed) in the graph. At the design stage, we iteratively construct each puzzle piece to expand the graph data structure incrementally such that potential target nodes can be as far from the root node as possible, aiming to increase the level of difficulty.

Contributions. Specifically, we make the following contributions:

- We propose a graph-based disassembly planner to compute the exact level of difficulty of an interlocking puzzle, defined by a non-monotone and linear/non-linear disassembly plan that requires a minimal number of moves to take out the first subassembly.
- We present a computational framework for constructing the geometry of each voxelized puzzle piece iteratively, guided by the graph-based disassembly planner, to achieve the user-specified level of difficulty.
- We formulate and solve a shape optimization problem to deform an input shape slightly such that we can generate high-level interlocking puzzles with smooth appearance and structurally sound pieces.

Thanks to our shape optimization, our approach is able to design puzzles with both voxelized and smooth appearance. We demonstrate the effectiveness of our computational approach on a variety of shapes, compare it with a state-of-the-art approach [Gontier 2020], and fabricate some of our designed puzzles to validate their level of difficulty; see Figure 1 for an example. Code and data of this paper are at https://github.com/Linsanity81/High-LevelPuzzle.
2 RELATED WORK

Puzzle design. Motivated by recent advances in digital fabrication, the graphics community has raised a great interest in research on computational design for stylized fabrication [Bickel et al. 2018]. Among this line of research, a number of computational methods and tools have been developed for personalized design and fabrication of various kinds of geometric puzzles, including 3D jigsaw puzzles [Elber and Kim 2022], polyomino puzzles [Kita and Miyata 2020; Lo et al. 2009], dissection puzzles [Duncan et al. 2017; Li et al. 2018; Tang et al. 2019; Zhou and Wang 2012], interlocking puzzles [Song et al. 2012; Xin et al. 2011], centrifugal puzzles [Kita and Saito 2020], and twisty puzzles that generalize the Rubik’s cube mechanism [Sun and Zheng 2015].

Interlocking assemblies. In an interlocking assembly, there is only one movable part called the key, while all other parts, as well as any subset of the parts, are immobilized relative to one another [Song et al. 2012]. A number of computational methods have been developed to construct interlocking assemblies for different applications, including puzzles [Song et al. 2012; Xin et al. 2011], 3D printed objects [Song et al. 2016, 2015; Yao et al. 2017], furniture [Fu et al. 2015; Song et al. 2017], architecture [Wang et al. 2019], and robotic assembly [Zhang and Balkcom 2016; Zhang et al. 2021]. In particular, Wang et al. [2018] developed a unified framework to design interlocking assemblies of different forms by leveraging a graph-based representation.

The main objective of the above works is to make the assembly structurally stable based on interlocking of component parts. Since the parts are preferred to be easily assembled to form the final structure, all these works assume a monotone and linear (dis)assembly plan. Due to this reason, the resulting interlocking assemblies typically have a level 1 difficulty; i.e., the key can be taken out directly with one move. In this paper, we focus on high-level interlocking assemblies. Different from level-1 interlocking assemblies that are interlocking only at the final configuration, high-level interlocking assemblies have to be “interlocking” (i.e., no removable subassembly) in all the configurations before the first subassembly is removed.

High-level interlocking puzzles. Designing high-level interlocking puzzles is much more difficult than designing the above level-1 interlocking assemblies, due to the coupling of two complex subproblems: disassembly planning and geometric design. The design of a new high-level interlocking puzzle is extremely hard for humans, even for skilled professionals, which perhaps explains why there are not many known high-level interlocking puzzles [Coffin 2006]. Cutler [1988] proposed to use computers to exhaustively try and discover six-piece interlocking holey burr puzzles. He performed a complete analysis of 13,354,991 essentially different puzzle assemblies, among which the highest level found was level-10. Other than exhaustive search, some 3D puzzle designers took a trial-and-error approach by using computer software such as BurrTools by Röver [2013] as a puzzle solver to test if their puzzle designs can be assembled as well as to compute their level of difficulty. Recently, Gontier [2020] developed a genetic algorithm to search for high-level interlocking cubes on supercomputers, and reported four results of interlocking cubes with level 5, 9, 10, and 13, respectively.

The above approaches compute the level of difficulty of a given interlocking puzzle based on a feasible disassembly plan found by a specific disassembly planner such as BurrTools [2013]. In contrast, we define the level of difficulty in a more strict way as the minimal number of moves to take out the first subassembly, which is intrinsic to the puzzle. Moreover, we develop a new graph-based disassembly planner that is able to compute this intrinsic level of difficulty and a computational framework that can construct the geometry of puzzle pieces to achieve the level of difficulty. Due to this reason, our approach can create high-level interlocking puzzles much more efficiently and flexibly, and generate results that cannot be achieved by the state-of-the-art approaches; see Section 7 for a quantitative comparison with [Gontier 2020].

Assembly planning. The goal of assembly planning is to find a sequence of operations to assemble the parts (i.e., assembly sequencing [Jiménez 2013]), and determine the motions that insert each part into the assembly (i.e., assembly path planning [Ghandi and Masehian 2015]). A bijection usually exists between assembly planning and disassembly planning [Halperin and Masehian 2007]. Hence, a common strategy to assembly planning is assembly-by-disassembly, where an assembly plan is obtained by computing a disassembly plan and then reversing its order and path.

To address the assembly sequencing problem, a number of data structures have been proposed to enumerate all possible assembly sequences, including Bourjault tree [Bourjault 1984], directed graph of assembly states [Fazio and Whitney 1987], and And/Or graph [Mello and Sanderson 1990]; readers are referred to the surveys [Jiménez 2013; Wolter 1991] for more details. All these data structures assume monotone assembly plans and cannot represent assembly states with intermediate placements of component parts. We address this issue by proposing a new graph data structure augmented with spatial information of the assembly such that it can enumerate all non-monotone assembly plans for high-level interlocking puzzles.

In the literature, there are very few works that deal with non-monotone assembly planning as pointed out in [Masehian and Ghandi 2020]. Tsao and Wolter [1993] proposed a method to generate a feasible non-monotone and non-linear assembly plan by assuming that intermediate states are given as input. Le et al. [2009] searched for a feasible non-monotone and linear disassembly plan by extending a sampling-based path planner. Masehian and Ghandi [2020] proposed a planner/replanner for monotone and non-monotone assembly planning, with advantages of supporting obstacles in the workspace and allowing translational and rotational movements of paths. Later, this approach was extended to support non-monotone assembly planning with both rigid and flexible parts [Masehian and Ghandi 2021]. All these existing approaches aim to find a feasible non-monotone (dis)assembly plan, making them inapplicable to our problem of finding an optimal non-monotone disassembly plan that requires a minimal number of moves to take out the first subassembly.

In the graphics community, (dis)assembly planning is typically formulated as an optimization problem to find a desired (dis)assembly plan that maximizes certain objectives. For example, the objective can be visibility of parts for creating visual assembly instructions [Agrawala et al. 2003], stability of incomplete assemblies for
we define the level of difficulty as follows:

\[ L_{\text{exact}} = \min_i N(D_i) \]  

Note that our definition of level of difficulty is unique and intrinsic to the puzzle, compared with the level computed by existing tools like Burr Tools [Röver 2013] that depends on the employed disassembly planner.

Design requirements. Given the user input, our designed puzzles should satisfy the following requirements:

1. Fabricability. Each generated puzzle piece should be a single piece of connected geometry \(1\) that can be fabricated.
2. Puzzle piece size. All the puzzle pieces should have similar sizes (i.e., similar number of voxels) to avoid fragmented pieces.
3. Level of difficulty \(L\). There should exist at least one disassembly plan that takes out the first subassembly with \(L\) moves. And there should not exist any disassembly plan that takes out the first subassembly with fewer than \(L\) moves.
4. Disassemblability. The puzzle can be completely disassembled into individual puzzle pieces.

In case the input voxelized shape is convex, we can only generate level-1 interlocking puzzles since any movable puzzle piece can be directly removed; see Figure 2(a). To resolve this issue, we allow users to create small holes inside the convex shape by introducing a few hole voxels; see Figure 2(h) for an example. The purpose of these hole voxels is to enable intermediate moves (yet not removal) of the puzzle pieces.

Overview of our approach. In Section 4, we propose a disassembly planner that uses a rooted graph to represent all possible valid configurations of the puzzle as well as transitions among these configurations, until the first subassembly is taken out. To compute the exact level of difficulty, our planner finds a path with the shortest length from the initial puzzle state (i.e., root node) to any state where a single subassembly has been taken out. We also extend our planner to check whether a given puzzle can be disassembled into individual pieces.

In Section 5, we propose a computational approach to design high-level interlocking puzzles with voxelized shape. Our approach consists of two key components: an iterative design framework that ensures the generated puzzles satisfy the design requirements, and an algorithm to construct the geometry of each puzzle piece guided by the disassembly planner.

In Section 6, we introduce methods to extend our design approach for generating high-level interlocking puzzles with smooth appearance and structurally sound pieces. The idea is to formulate and solve a shape optimization problem to ensure that a sufficiently large subvolume of each voxel is covered by the optimized shape. By this, we can generate puzzle pieces with smooth appearance by simply performing CSG intersection between each voxelized puzzle piece and the optimized shape.
4 DISASSEMBLY PLANNER

The purpose of our disassembly planner is twofold. First, it should be able to compute the exact level of difficulty for a given interlocking puzzle (Section 4.2). Second, it should be able to identify if the puzzle can be disassembled into individual pieces (Section 4.3). To facilitate understanding of our approach, we first define a set of relevant concepts in Section 4.1.

4.1 Definitions

Given an interlocking puzzle $P$, we denote its pieces as $\{P_1, ..., P_K\}$, $K \geq 3$, where $K$ is the number of pieces in the puzzle. We assume that each puzzle piece is represented as a polycube, and the whole puzzle forms a voxelized shape.

To disassemble the puzzle, we assume that each puzzle piece can only translate along one of the three major axes following a certain order. Denote the side length of a voxel as $\mu$. We call that a translation of a puzzle piece along one major axis for a distance $d = h \cdot \mu$ as a translation of $h$ steps, where $h \geq 1$ and $h \in \mathbb{Z}$. We only consider these discrete translations for disassembling the puzzle pieces, due to the discrete nature of the voxelized shape.

**State.** A state for a puzzle piece $P_i$ is defined as either the piece’s initial position or other positions induced by a set of operations. We represent a puzzle piece $P_i$’s state $s_i$ using the displacement from its initial position. Hence, the initial state of each $P_i$ is always $s_i = (0, 0, 0)$; see Figure 4(a). The state $s_i$ consists of integers only due to the discrete nature of the voxelized shape. For example, the state of $P_1$ in Figure 4(b) is $(1, 0, 0)$ since it translates along $+x$ for one step from its initial state. We introduce the infinity state, denoted as $\infty$, to represent the state of a piece that is disassembled such as $P_1$ in Figure 4(g).

**Operation.** We define a disassembly operation for a puzzle piece as a change of state by translating along one of the major axes for one or multiple steps. That is, for each puzzle piece $P_i$, an operation is denoted by four variables $o_j = (s_i^u, s_i^v, d_j, h)$, where $s_i^u$ is the source state that $P_i$ moves from, $s_i^v$ the destination state that $P_i$ moves to, $d_j \in \{-x, +x, -y, +y, -z, +z\}$ is the moving direction of $P_i$, and $h$ is the number of moving steps. We classify disassembly operations into the following two types, i.e. temporary operations and removal operations. A temporary operation is defined as an operation that moves a puzzle piece from a non-infinity state to another non-infinity state (e.g., moving $P_1$ along $+x$ in Figure 4(e) to reach the state in Figure 4(f)) whereas a removal operation moves a puzzle piece from a non-infinity state to the infinity state (e.g., moving $P_1$ along $+y$ in Figure 4(f) to remove it).

**Move.** In order to support non-linear disassembly planning, we define a move $m_i$ to be a non-empty set of collision-free operations where the associated puzzle pieces are moved simultaneously along the same direction for the same distance; see the move of $P_1$ and $P_3$ along $-x$ in Figure 4(c). Obviously, a move is just a generalization of an operation from a single puzzle piece to a subset of puzzle pieces. Here, we require that the subset of puzzle pieces have to contact each other to form a connected piece of geometry. Such subset of puzzle pieces is called a **subassembly**, denoted as $S_j$. Each single puzzle piece is considered as a special case of a subassembly.

**Configuration.** We define a puzzle configuration as a set of puzzle pieces together with their states, i.e., $C = \{(P_1, s_1), ..., (P_K, s_K)\}$. A configuration is valid if there is no overlap among the puzzle pieces in the configuration (i.e., collision-free). According to the states of the puzzle pieces, we classify a configuration $C$ into three types:

1. **Full configuration.** In a full configuration, the state $s_j = \infty$ for none of the $K$ puzzle pieces; see Figure 4(a-f).

2. **Partial configuration.** In a partial configuration, the state $s_j = \infty$ for $k$ puzzle pieces, where $1 \leq k \leq K - 2$, meaning that $k$ puzzle pieces have been disassembled; see Figure 4(g-m);
Algorithm 1 Algorithm to build a kernel disassembly graph $G$ for a given interlocking puzzle $P$.

1. function BuildKernelDisassemblyGraph($P$)
2. $G = (V, E) \leftarrow \emptyset$
3. let $C_1$ be the initial puzzle configuration
4. mark $C_1$ as the root node
5. $V$.push_back($C_1$)
6. let $Q$ be queue
7. $Q$.enqueue($C_1$)
8. mark $C_1$ as unvisited
9. while $Q$ is not empty do
10. $C = Q$.dequeue()
11. if $C$ has been visited then
12. continue
13. if $C$ is not a full configuration then
14. mark $C$ as a target node
15. else
16. $C_{neighbor} = \text{ComputeNeighborConfigs}(C)$
17. for each $C_k$ in the list $C_{neighbor}$ do
18. if $C_k \in V$ then
19. if $(C, C_k) \notin E$ then
20. $E$.push_back($(C, C_k)$)
21. $V$.push_back($C_k$)
22. $E$.push_back($(C, C_k)$)
23. mark $C_k$ as unvisited
24. $Q$.enqueue($C_k$)
25. mark $C$ as visited
26. return $G$

(3) Final configuration. In a final configuration, the state $s_i = \infty$ for at least $K - 1$ puzzle pieces; i.e., the puzzle has been completely disassembled into individual puzzle pieces; see Figure 4(h);

Kernel disassembly plan. We define a kernel disassembly plan $D_{kern} = \langle m_1, m_2, \ldots, m_l \rangle$ as an ordered sequence of moves to take out the first subassembly, where $l$ is the number of moves in $D_{kern}$; see Figure 4(a-g). This kernel disassembly plan can also be represented as a set of configurations of the puzzle, i.e., $D_{kern} = \langle C_1, C_2, \ldots, C_{l+1} \rangle$, where $C_1$ is the initial puzzle configuration and $C_{l+1}$ is the configuration obtained by applying move $m_l$ onto the configuration $C_j$. We require that each $C_j$, where $1 \leq j \leq l$, is a full configuration while $C_{l+1}$ is a partial configuration with exactly one removed subassembly.

Complete disassembly plan. We define a complete disassembly plan $D_{comp} = \langle m_1, m_2, \ldots, m_q \rangle$ as an ordered sequence of moves to disassemble the puzzle into individual pieces, where $q$ is the number of moves in $D_{comp}$; see Figure 4. Similarly, the complete disassembly plan can be represented as a set of configurations of the puzzle, i.e., $D_{comp} = \langle C_1, C_2, \ldots, C_{l+1}, \ldots, C_{q+1} \rangle$, where $C_{q+1}$ is the final configuration. From the definition, we can see that there exists $l < q$ such that the first $l$ moves in a complete disassembly plan form a kernel disassembly plan.

Fig. 5. The kernel disassembly graph of a 4-piece level-4 interlocking puzzle. The graph has three target nodes, and the exact level (i.e., 4) of the puzzle is defined by the shortest path from the root node to the target node 1 or 2 (the path is colored in orange or cyan respectively).

4.2 Computing Level of Difficulty
To compute the level of difficulty, we have to enumerate all possible kernel disassembly plans to take out the first subassembly; see Equation 1. However, directly enumerating all these plans has significant redundancy in computation since many plans share the same configurations with different orders. To avoid the ordering problem [Wu et al. 2019], our idea is to enumerate all the valid configurations of the puzzle as well as valid transitions among these configurations instead. To this end, we propose a rooted graph data structure called kernel disassembly graph to encode all possible kernel disassembly plans, in which each node represents a puzzle configuration and each edge represents a move of a subassembly. A kernel disassembly graph is an undirected graph since a move is bidirectional according to its definition; see Figure 5 for a simple example.

In a kernel disassembly graph, the root node is the initial puzzle configuration $C_1$, and nodes with one subassembly removed are called target nodes. All the nodes except the target nodes in the graph are full configurations. A kernel disassembly plan is a path from the root node to one target node in the graph, where the length of the path is the number of moves to take out the first subassembly.
We need to extend the algorithm to check if a given puzzle can be disassembled into individual puzzle pieces, which is known to be an NP-hard problem [Kavraki et al. 1993]. To speed up the planning process, our algorithm aims to find a feasible (i.e., collision-free) complete disassembly plan instead of enumerating all possible complete disassembly plans; see Algorithm 3. Hence, our algorithm cannot guarantee that the puzzle is not disassemblable if it does not find a complete disassembly plan. In detail, we first modify Algorithm 1 to find a feasible kernel disassembly plan that removes a single subassembly from a puzzle by terminating the graph expanding process once it finds a target node; see RemvSubassembly in Algorithm 3. Our disassembly planning algorithm calls this function recursively for both the removed subassembly $S_{remv}$ and the remaining puzzle $P_{rema}$. This recursion terminates when $P_{rema}$ and each $S_{remv}$ consist of a single puzzle piece respectively (i.e., $P$ is disassemblable) or when there is no subassembly that can be taken out from $P_{rema}$ or $S_{remv}$ (i.e., $P$ is not disassemblable).

Besides the first removed subassembly, it may also require multiple moves to take out some of the subsequent subassemblies. For example, in Figure 4, it takes 6 moves to take out the firstly removed piece $P_1$ and 3 moves to take out the secondly removed piece $P_2$. An alternative way to measure an interlocking puzzle’s level of difficulty is to use the total number of moves to completely disassemble the puzzle into individual pieces, denoted as $L_{total}$. However, since our disassembly planner can only find a feasible complete disassembly plan due to the huge search space, our computed $L_{total}$ cannot be guaranteed to be the smallest total number of moves to disassemble the puzzle.

### 4.3 Disassembly Planning

A given puzzle is possible to be not disassemblable, which can be classified into three cases: 1) the whole puzzle is deadlocking; 2) the puzzle is deadlocking after taking out a few puzzle pieces; and 3) at least one removed subassembly is deadlocking; see Figure 6 for examples. Algorithm 1 can only identify the first case of deadlocking (i.e., no target node in the graph) but not the other two cases. Hence, we need to extend the algorithm to check if a given puzzle can be disassembled into individual puzzle pieces, which is known to be an NP-hard problem [Kavraki et al. 1993].

To speed up the planning process, our algorithm aims to find a feasible (i.e., collision-free) complete disassembly plan instead of enumerating all possible complete disassembly plans; see Algorithm 3. Hence, our algorithm cannot guarantee that the puzzle is not disassemblable if it does not find a complete disassembly plan. In detail, we first modify Algorithm 1 to find a feasible kernel disassembly plan that removes a single subassembly from a puzzle by terminating the graph expanding process once it finds a target node; see RemvSubassembly in Algorithm 3. Our disassembly planning algorithm calls this function recursively for both the removed subassembly $S_{remv}$ and the remaining puzzle $P_{rema}$. This recursion terminates when $P_{rema}$ and each $S_{remv}$ consist of a single puzzle piece respectively (i.e., $P$ is disassemblable) or when there is no subassembly that can be taken out from $P_{rema}$ or $S_{remv}$ (i.e., $P$ is not disassemblable).

Besides the first removed subassembly, it may also require multiple moves to take out some of the subsequent subassemblies. For example, in Figure 4, it takes 6 moves to take out the firstly removed piece $P_1$ and 3 moves to take out the secondly removed piece $P_2$. An alternative way to measure an interlocking puzzle’s level of difficulty is to use the total number of moves to completely disassemble the puzzle into individual pieces, denoted as $L_{total}$. However, since our disassembly planner can only find a feasible complete disassembly plan due to the huge search space, our computed $L_{total}$ cannot be guaranteed to be the smallest total number of moves to disassemble the puzzle.

Fig. 6. Three cases of deadlocking puzzles. From left to right, the whole puzzle, the puzzle after removing the green piece, and the removed subassembly, are deadlocking.
To address the puzzle design problem formulated in Section 3, a straightforward approach would be a trial-and-error approach that iterates between randomly assigning piece IDs (from 1 to N) to each voxel in the input shape and checking whether the resulting puzzle satisfies the requirements in Section 3. However, this approach is very inefficient and hard to generate a desirable result since each puzzle piece can be easily disconnected and the puzzle can be easily non-interlocking or non-disassemblable, especially when the input shape has a large number of voxels; see Section 7 for the complexity analysis of our puzzle design problem.

In this section, we propose a computational approach to design a K-piece level-L interlocking puzzle by constructing the geometry of each puzzle piece iteratively; see Figure 7. Section 5.1 introduces our computational framework as well as a set of requirements on constructing each puzzle piece. Section 5.2 presents an algorithm that constructs the geometry of each puzzle piece to satisfy the requirements in Section 5.1.

### 5.1 Computational Framework

Given the input voxelized shape denoted as $R_0$, we iteratively construct the geometry of each puzzle piece one by one; see Figure 7 for an example. This forms a sequence of constructed puzzle pieces, $P_1, P_2, ..., P_{K-1}, P_K$:

$$[R_0] \rightarrow [P_1, R_1] \rightarrow [P_1, P_2, R_2] \rightarrow ... \rightarrow [P_1, ..., P_{K-1}, R_{K-1}]$$

where $R_i, 1 \leq i \leq K-1$, is the remaining part of the shape and $R_{K-1} = P_K$ is the last puzzle piece. Here we denote each intermediate assembly $[P_1, ..., P_i, R_i]$ as $A_i$ ($0 \leq i \leq K-1$), and its kernel disassembly graph as $G(A_i)$.

To design a high-level interlocking puzzle, our idea is that the construction of each puzzle piece has to increase or at least preserve the potential level of difficulty for the resulting puzzle. To this end, we require that there is no removable subassembly in any of the intermediate puzzles $A_i$, $1 \leq i \leq K-2$; see Figure 7(b-e). In other words, each node in each kernel disassembly graph $G(A_i)$, $1 \leq i \leq K-2$, should be a full puzzle configuration. And only the resulting puzzle $P = A_{K-1}$ is disassemblable after we construct the last two puzzle pieces by decomposing $R_{K-2}$ into $P_{K-1}$ and $P_K$; see Figure 7(f). Otherwise, the level of difficulty of the resulting puzzle will be the same as or smaller than that of the first intermediate assembly $A_i$ with removable subassemblies, no matter how we construct the geometry for the subsequent puzzle pieces. To increase the level of difficulty, we further require that each kernel disassembly graph $G(A_i)$ should have as many nodes as possible, especially those far away from the root node.

To implement the above idea, we propose the following requirements when decomposing $R_{i+1}$ into $P_i$ and $R_i$, where $1 \leq i \leq K-1$:

(i) **Connected geometry.** The geometry of $P_i$ and $R_i$ should be connected respectively, making them fabricable.

(ii) **Puzzle piece size.** The number of voxels of $P_i$ should be within the range $[(1 - \delta)(M/K), (1 + \delta)(M/K)]$, where $M$ is the total number of voxels in the input shape.

![Fig. 7. Overview of our design approach. (a) Taking a voxelized shape as an input, (b-f) we generate the geometry of each puzzle piece iteratively by partitioning the shape, during which we build (bottom) a kernel disassembly graph for (top) each intermediate puzzle. To increase the level of difficulty, we require that (b-e) each intermediate puzzle should not have any removable subassembly and (f) only the resulting puzzle is disassemblable when the last two puzzle pieces $P_i$ and $P_{i+1}$ are constructed. The shortest path in the graph to take out the first subassembly (i.e., $P_1$) is colored in blue.](image-url)
We propose an approach to construct each puzzle piece \( P \) with \( K \) voxels (marked as \( T \)) in one direction in the set as \( \delta \) denoted as \( d \). According to the requirements in Section 3. Moreover, our framework aims at augmenting it with more voxels to balance the number of voxels in the input shape \( R_0 \) and \( \delta \in [0, 1] \) is a user-specified parameter.

(iii) **Movable puzzle piece.** \( P_1 \) should be movable in a configuration that is furthest away from the root configuration in \( G(A_{i-1}) \), aiming to extend the graph in a depth-first manner.

(iv) **No removable subassembly in \( A_i \), \( 1 \leq i \leq K-2 \).** There is no removable subassembly in each intermediate puzzle \( A_i \) after constructing \( P_1 \), where \( 1 \leq i \leq K-2 \).

(v) **Disassemblable puzzle \( A_{K-1} \).** The resulting puzzle \( A_{K-1} \) becomes disassemblable after constructing \( P_{K-1} \).

Our computational framework guarantees that the resulting puzzle \( A_{K-1} \) satisfies the fabricability, puzzle piece size, and disassemblability requirements in Section 3. Moreover, our framework aims to generate an interlocking puzzle with an as-high-as-possible level of difficulty, although it does not have exact control over the level of difficulty of the resulting puzzle. To this end, we allow some modification operations on the resulting puzzle to satisfy the level of difficulty requirement in Section 3.

### 5.2 Puzzle Piece Construction

We propose an approach to construct each puzzle piece \( P_i \), \( 1 \leq i \leq K-1 \), to satisfy the requirements in Section 5.1. We first present our approach to construct the first piece \( P_1 \) and then the other pieces \( P_i \), \( 2 \leq i \leq K-1 \). Lastly, we introduce the modification operations on the pieces \( P_i \) to achieve the user-specified level of difficulty \( L \).

**Constructing \( P_1 \).** Constructing the first puzzle piece \( P_1 \) is relatively simple since the kernel disassembly graph \( G(A_{0}) \) only has a single node, which is the input shape \( R_0 \). According to the requirements in Section 5.1, \( P_1 \) should be movable but not removable in the input shape \( R_0 \). Moreover, \( P_1 \) should be movable along a single direction denoted as \( d_1 \) in \( R_0 \), aiming to make the resulting puzzle more stable. We construct \( P_1 \) with the following steps:

(i) **Pick the moving direction \( d_1 \).** We first identify the set of non-height-field directions for the input shape \( R_0 \); e.g., the set should be \( \{+x,-x\} \) for the input shape in Figure 7. We randomly choose one direction in the set as \( P_1 \)'s moving direction \( d_1 \). Height-field directions cannot be selected since they will make \( P_1 \) removable along the direction.

(ii) **Pick a seed voxel.** For a selected moving direction \( d_1 \), a seed voxel should not contact any voxel along \( d_1 \), to ensure mobility of \( P_1 \) along \( d_1 \). To ensure that \( P_1 \) is not removable along \( d_1 \), there should exist some voxels that stop the movement of \( P_1 \) along \( d_1 \). We identify all voxels satisfy the two conditions; see Figure 8(a&b) for examples. We randomly choose one voxel from the set as a seed voxel; see Figure 8(c).

(iii) **Ensure blocking.** If the seed voxel is also movable along a direction that is not \( d_1 \), we prevent such motion by identifying a pair of blocking and blockee voxels and connecting the seed voxel with the blockee voxel using a shortest path [Song et al. 2012].

(iv) **Expand \( P_1 \).** Since \( P_1 \) usually has a few voxels at this moment, the goal of this step is to augment it with more voxels to balance the size of the puzzle piece. To this end, we add voxels one by one to \( P_1 \), without violating the mobility and blocking conditions that have been satisfied; see Figure 8(d&e).

**Constructing \( P_i \), \( 2 \leq i \leq K-1 \).** Constructing the subsequent puzzle piece \( P_i \) is more complex since it has to satisfy the requirements for all the configurations in \( G(A_{i-1}) \). To increase the level of difficulty, we identify configurations in \( G(A_{i-1}) \) that are furthest away from the root node in \( G(A_{i-1}) \). We randomly choose one as a primary configuration for constructing \( P_i \), denoted as \( C_{prim} \); see Figure 7(b-e). Our idea is to make \( P_i \) movable but not removable in the configuration \( C_{prim} \) of \( G(A_{i-1}) \) such that we can create new configurations in \( G(A_i) \) that are even further away from the root node than \( C_{prim} \) in \( G(A_{i-1}) \).

(i) **Pick the moving direction \( d_i \) in configuration \( C_{prim} \).** We identify the set of non-height-field directions for the assembly \( A_{i-1} \) at configuration \( C_{prim} \), and randomly choose one direction in the set as \( P_i \)'s moving direction \( d_i \); see Figure 9(a&d).

(ii) **Pick a seed voxel in configuration \( C_{prim} \).** A seed voxel is selected such that \( P_i \) is movable but not removable along direction \( d_i \) in the
Algorithm 4 Algorithm to modify a puzzle $P$ to increase its level of difficulty to $L$.

1: function ModifyPuzzle($P$)
2:     for $m=0$; $m<$IterNum; $m++$ do
3:         randomly select a voxel $V_k$ in $P$ using reachability
4:         randomly select a neighboring puzzle piece $P_j$ of $V_k$
5:         let $P_i$ be the puzzle piece that has $V_k$
6:         assign $V_k$ from $P_i$ to $P_j$
7:         let $\tilde{P}$ be the modified puzzle
8:         if $P_j$ in $\tilde{P}$ is not connected then
9:             continue
10:         if $\tilde{P}$ is not disassemblable then
11:             continue
12:         if $L_{\text{exact}}(\tilde{P}) == L$ then return $\tilde{P}$
13:         if $(L_{\text{exact}}(\tilde{P}) > L_{\text{exact}}(P)) \land ((L_{\text{exact}}(\tilde{P}) == L_{\text{exact}}(P)) \land (L_{\text{total}}(\tilde{P}) > L_{\text{total}}(P)))$ then
14:             $P \leftarrow \tilde{P}$
15:     return NULL

primary configuration $C_{\text{prim}}$ using the same approach for picking a seed voxel for $P_i$.

iii) Ensure blocking in all configurations in $G(A_{i-1})$. If the seed voxel is movable along a direction that is not $d_1$ in $C_{\text{prim}}$ or movable along any direction in other nodes in $G(A_{i-1})$, we will summarize all these unwanted mobilities, choose a minimal number of pairs of blocking and blockee voxels, and connect the seed voxel with each blockee voxel using a shortest path to get rid of the mobilities; see Figure 9(b&c).

iv) Expand $P_i$. We expand $P_i$ by adding voxels one by one to $P_i$, without violating the mobility and blocking conditions that have been satisfied for any configuration in $G(A_{i-1})$; see Figure 9(c&f).

The above puzzle piece construction process may not be always successful. When it fails, our approach backtracks until all the puzzle pieces can be successfully constructed and disassembled. We compute the level of difficulty of our generated puzzle by running the disassembly planner in Section 4. In case that the generated puzzle’s level of difficulty is different from the user-specified level $L$, we will re-generate another puzzle design by repeating the random construction process of the puzzle pieces. Our approach terminates when the puzzle’s level of difficulty is the same as the user-specified level $L$ or the computation time exceeds a user-specified threshold. In practice, we find that our approach is efficient when the user-specified level $L$ is not that large. However, when $L$ is large, our approach may fail to generate such puzzles within a user-specified time threshold. To address this limitation, we introduce additional modification operations on a puzzle generated by our approach, aiming to increase its level of difficulty to the user-specified $L$.

Modifying $\{P_i\}$, $1 \leq i \leq K$. We found that an interlocking puzzle’s level of difficulty is possible to be increased by a slight modification on the puzzle geometry in our experiments. Inspired by this observation, we perform the slight puzzle geometry modification iteratively using Algorithm 4 to increase the level of difficulty. Our algorithm consists of the following steps: 1) randomly choose a voxel on the puzzle according to its reachability [Song et al. 2012] (i.e., we prefer to choose a voxel with fewer number of neighbors); 2) assign the voxel to a neighboring puzzle piece that is randomly selected; 3) compute the modified puzzle’s level of difficulty $L_{\text{exact}}$ and total number of moves $L_{\text{total}}$; and 4) update the puzzle geometry if the modification leads to a larger $L_{\text{exact}}$ and/or $L_{\text{total}}$. The process terminates when finding a level-$L$ interlocking puzzle or exceeding the maximum number of iterations.

6 SHAPE OPTIMIZATION AND VOXELIZATION

Our computational design approach in Section 5 can generate high-level interlocking puzzles with voxelized shape. In this section, we will extend our computational design approach to support generating puzzles with smooth appearance. To this end, a straightforward approach is to directly perform a CSG intersection operation between each voxelized puzzle piece and the input smooth shape; see Figure 10(a-d). However, this simple approach may result in structurally weak puzzle pieces that can easily break during fabrication or playing; see Figure 10(e). Such puzzle pieces typically include voxels that contain only a tiny part of the input smooth shape, called problematic voxels.

To address this issue, our idea is to minimize the number of such problematic voxels by allowing slight deformation on the input smooth shape during the voxelization. Then, we take the voxelized shape without the problematic voxels to generate a high-level interlocking puzzle, which guarantees the structural soundness of each puzzle piece. In the end, we attach the tiny shape contained in the problematic voxels (if any) back to the corresponding puzzle piece following the approach in [Song et al. 2015]. We formulate our problem of shape optimization for voxelization in Section 6.1 and then present a method to solve the problem in Section 6.2.

![Fig. 10. (a) Voxelizing a given smooth shape. (b) Taking the voxelized shape (with two hole voxels) as an input, we generate a (c) 5-piece level-2 interlocking puzzle by using our design approach in Section 5. (d) A puzzle with smooth appearance is obtained by performing CSG intersection between each voxelized puzzle piece in (c) and the input shape in (a). (e) However, the generated puzzle pieces are possible to be structurally weak for fabrication and playing (see red circles).]
We define the energy $E$ where $E_{M}$ is the number of problematic voxels, $E_{\lambda}$ is the weight for the energy $E_{\lambda}$, and $\lambda$ is the weight for the energy $E_{\lambda}$. We define the energy $E_{\lambda}(V)$ as an as-rigid-as-possible shape preservation energy [Sorkine and Alexa 2007].

6.1 Shape Optimization Problem

The input smooth shape is represented as a mesh surface $M$ with vertices $V$ and faces $F$. Assuming the voxel size $\mu$ is specified by the user, we obtain an initial voxelized shape by voxelizing the mesh $M$ with the voxel size $\mu$ [Nooruddin and Turk 2003]. We classify voxels in the voxelized shape into three classes according to the position of each voxel relative to the mesh $M$: full voxel that is totally inside the mesh $M$, empty voxel that is totally outside of the mesh $M$, and partial voxel that intersects the mesh $M$. We further identify problematic voxels as partial voxels that contain a tiny amount of local shape of $M$; see Figure 11(a). In our experiments, problematic voxels are partial voxels with less than 0.1$\mu^3$ volume filled.

The goal of our shape voxelization is to minimize the number of problematic voxels while preserving the input shape $M$ as much as possible. Our search space includes: 1) a translational vector $t$ that defines the position of the mesh $M$ with respect to the voxelized shape; 2) a uniform scale factor $w$ of the mesh $M$; and 3) vertices $V$ of the mesh $M$. We formulate our shape voxelization problem as an optimization problem:

$$\min_{t, w, V} E_{\text{voxel}}(t, w, V) + \lambda E_{\text{shape}}(V)$$

where $E_{\text{voxel}}$ is the number of problematic voxels, $E_{\text{shape}}$ is the shape preservation energy, and $\lambda$ is the weight for the energy $E_{\lambda}$. We define the energy $E_{\text{shape}}(V)$ as an as-rigid-as-possible shape preservation energy [Sorkine and Alexa 2007].

6.2 Optimization Solver

Solving the optimization problem in Equation 2 is challenging due to two reasons. First, computing the gradient of $E_{\text{voxel}}$ is complex, which typically involves differentiation on the CSG intersection operation between each voxel and the mesh $M$. Second, the search space is large, and contains both the transformation $(t, w)$ and geometry $V$ of the input shape $M$.

To address the first challenge, our observation is that the boundary surface $M_{\text{full}}$ of the set of full voxels is a good approximation of the input surface $M$; see Figure 11(a). Hence, we define a shell $S$ whose outer and inner surfaces are obtained by offsetting the boundary surface $M_{\text{full}}$ by $\mu$ and $\alpha \cdot \mu$ ($0.5 \leq \alpha \leq 0.9$ in our experiments) distance, respectively; see Figure 11(b). If the input mesh $M$ can be optimized such that it is contained in the shell $S$, then there is no problematic voxel since the local shape volume covered in each partial voxel is at least $(\alpha \cdot \mu)^3$; see Figure 11(c). Moreover, the closer the mesh $M$ is to the shell $S$, the more likely there is a fewer number of problematic voxels. Hence, we define:

$$E_{\text{voxel}}(t, w, V) = \sum_{F_i \in F} (\text{dist}(F_i(t, w, V), S))^2$$

where $F_i$ is a face of the mesh $M$ and $\text{dist}(F_i, S)$ is the distance between the face $F_i$ and the shell $S$. When a face $F_i$ is contained in the shell, $\text{dist}(F_i, S) = 0$.

To address the second challenge, we use a two-stage approach to explore the search space effectively. First, the transformation stage fixes the vertex positions $V$ and searches for an optimal transform $(t, w)$ to minimize the energy $E_{\text{voxel}}$, during which we perform voxelization for each transformed mesh to compute the energy $E_{\text{voxel}}$. Since the transform $(t, w)$ is only 4 degrees of freedom, we can uniformly sample the variables’ space of $(t, w)$ to find the optimal one. Then, the deformation stage fixes the mesh transformation $(t, w)$ and deforms the input mesh $M$ to minimize the energy of Equation 2, assuming the voxelization is fixed. We use the L-BFGS algorithm to solve the mesh deformation problem. We iterate between the shape transformation stage and shape deformation stage until the ratio between the number of problematic voxels and the total number of voxels cannot be further reduced.

Figure 12 shows an example before and after our shape optimization. It took our algorithm 30 minutes to reduce the problematic voxel ratio from 23.3% to 11.7%. Please refer to the supplementary material for details about our optimization solver.

7 RESULTS

We implemented our computational design tool in C++ and libigl [Jacobson et al. 2018] on a desktop computer with 3.6 GHz 8-Core Intel processor and 16 GB RAM. Our tool allows user control in several aspects (see Table 1):

- **Voxelization resolution.** Users can specify the input voxelized shape’s resolution, typically fewer than 1000 voxels. Low-resolution voxelized shapes are preferred since they ensure that the move...
Fig. 13. High-level interlocking puzzles generated by our approach. From left to right and then top to bottom: CUBE FRAME, SHELF, SPIDER, MARIO, BUNNY, MOAI, ANGRY BIRD, OWL, PUMPKIN, and SQUIRREL.

of each puzzle piece for disassembly is significant relative to the whole puzzle size.

- **Number of hole voxels** $E$. For convex input shapes such as CUBE, hole voxels are needed to generate high-level interlocking puzzles. By default, our tool inserts a single hole voxel at the center of the shape. Users can override this by specifying the number of hole voxels $E$ as well as their locations in the input shape.

- **Number of puzzle pieces** $K$. The number of puzzle pieces $K$ is typically between 3 and 8.

- **Target level of difficulty** $L$. The target level of difficulty $L$ is typically between 4 to 30. When $L$ is too large, our tool may not be able to generate such a puzzle. In this case, our tool will output the puzzle whose level of difficulty is closest to $L$.

**Results.** Our design tool allows generating high-level interlocking puzzles with a variety of voxelized shapes and topologies, including CUBE FRAME with hollows, SHELF with four large cavities, SPIDER with elongated features, and MARIO with a 2.5D shape; see Figure 13 (top). The variety of 3D shapes that can be represented by a low-resolution voxelization is limited. Thanks to our shape optimization, our tool can generate high-level interlocking puzzles with smooth appearance, which significantly extends the resulting puzzles’ shape complexity and variety; see Figure 13 (bottom). Our tool can generate interlocking puzzles with a very high level of difficulty; see the 5-piece level-27 interlocking CUBE in Figure 15. We refer to the accompanying video for the disassembly animation of the results. We provide the puzzle piece 3D models and the kernel disassembly graph of each result in the supplementary material.

Table 1 provides the statistics of all the results shown in the paper. To generate high-level interlocking puzzles efficiently, it is usually necessary to modify the puzzle pieces using Algorithm 4; see the second column from right in the table. For example, we could not generate a level-15 SHELF by running the puzzle piece construction algorithm for 24 hours. Instead, we were able to generate a level-15 SHELF within 1.23 hours by first generating a level-6 SHELF using the puzzle piece construction algorithm (0.95 hours) and then modifying the puzzle’s geometry using Algorithm 4 (0.28 hours).

**Complexity and performance.** The complexity of our puzzle design problem is $O(K^N \cdot N^{2K})$, where $K$ is the number of puzzle pieces and $N$ is the number of voxels. For a given input shape, the total number of possible puzzle designs (including invalid ones) is $K^N$, since each voxel can be assigned to any puzzle piece. For each design, we need to compute its level of difficulty by building a kernel disassembly graph using a BFS-based algorithm with complexity $O(V + E)$, where $V$ and $E$ are the number of vertices and edges in the graph respectively. In the worst case, $V = N^K$ since each puzzle...
Table 2. Comparing our approach with a baseline [Gontier 2020] for generating high-level interlocking Cubes with a single hole voxel at the center. The numbers in the table are the highest level of difficulty that can be achieved by both approaches, given the same amount of computation time (i.e., 12 hours).

<table>
<thead>
<tr>
<th></th>
<th>Cube_4x4x4_E1</th>
<th>Cube_5x5x5_E1</th>
<th>Cube_6x6x6_E1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K=3 K=4 K=5 K=6</td>
<td>K=3 K=4 K=5 K=6</td>
<td>K=3 K=4 K=5 K=6</td>
</tr>
<tr>
<td>[Gontier 2020]</td>
<td>8 5 5 1</td>
<td>9 8 8 1</td>
<td>3 1 8 1</td>
</tr>
<tr>
<td>Our Approach</td>
<td>8 6 8 5</td>
<td>11 16 14 13</td>
<td>15 20 27 17</td>
</tr>
</tbody>
</table>

Table 3. Performance of our approach for generating K-piece level-L interlocking 5 × 5 × 5 Cubes with a single hole voxel. The computational time of each result is in minutes.

<table>
<thead>
<tr>
<th></th>
<th>K=4</th>
<th>K=6</th>
<th>K=8</th>
</tr>
</thead>
<tbody>
<tr>
<td>L=4</td>
<td>0.028</td>
<td>0.057</td>
<td>1.463</td>
</tr>
<tr>
<td>L=8</td>
<td>4.540</td>
<td>3.909</td>
<td>62.011</td>
</tr>
<tr>
<td>L=12</td>
<td>68.272</td>
<td>45.643</td>
<td>365.675</td>
</tr>
<tr>
<td>L=16</td>
<td>70.757</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Despite the high complexity of the problem, our design approach is able to find a solution in the huge design space using a reasonable amount of time. We evaluate the performance of our design approach on a 5 × 5 × 5 Cube with a single hole voxel at the center; see Table 3. The computation time of our approach increases with respect to the number of puzzle pieces K since a larger K means each puzzle piece has fewer number of voxels, making them harder to interlock. The computation time of our approach also increases with respect to the target level of difficulty L since a larger L means that the puzzle has to be interlocking in a larger number of configurations. Our approach is efficient when K ≤ 6 and L ≤ 8, e.g., less than 5 minutes for the 5 × 5 × 5 Cubes.

Comparison with [Gontier 2020]. We compare our design approach with a baseline approach [Gontier 2020]. The baseline approach uses a genetic algorithm to generate high-level interlocking puzzles, in which each puzzle candidate (encoded as a 1D array) is an individual and the fitness function is the level of difficulty. To ensure a fair comparison, we re-implemented the baseline approach in C++ and replaced the disassembly planner with ours to compute the exact level of difficulty. We ran both approaches for 12 hours to generate high-level interlocking Cubes with different resolutions, and recorded the puzzle with the highest level of difficulty; see Table 2. When the Cube puzzle has a low resolution (i.e., 4 × 4 × 4) and a small number of pieces (i.e., K ≤ 4), the two approaches have comparable performance. However, our approach has significantly better performance than the baseline when the puzzle has a higher resolution and/or a larger number of pieces, demonstrating its good scalability to a large design space.

Physical puzzles. We fabricated five of our designed puzzles using a Stratasys J750 multi-color 3D printer; see Figure 14. When playing these puzzles, we found that a larger number (K) of puzzle pieces makes the puzzle harder to play; e.g., the 5-piece AIRPLANE is harder to play than the 4-piece SOFA. This is because one needs more effort to hold the puzzle pieces steadily so they remain in the correct position of each configuration in order to make the next move. We
also found that the time to solve the puzzle increases significantly with the level of difficulty \( L \). We argue that this is because a large \( L \) usually means a large kernel disassembly graph, requiring one to explore a large space of puzzle configurations to solve the puzzle.

**User study.** We conducted a user study to learn how difficult for general users to play our high-level interlocking puzzles. Besides the level-16 Cube, level-8 Sofa, and level-7 Owl shown in Figure 14, we prepared two more puzzles for the user study, a level-4 Cube and a level-8 Cube. We recruited 8 participants, 4 males and 4 females, to play the five 3D printed puzzles, and recorded the time to solve each puzzle for each participant. We found that the average time to solve the puzzles increases with the puzzles’ level of difficulty (e.g., 0.30 mins, 0.35 mins, and 7.02 mins for the level-4, -8, and -16 Cubes respectively). In particular, four participants failed to solve the level-16 Cube, due to its very high level of difficulty. One interesting observation is that the time to solve a puzzle depends not only on the level of difficulty but also on the kernel disassembly graph’s size. For example, all the participants took a much longer time to solve the level-8 Sofa than the level-8 Cube (i.e., 1.87 mins vs 0.35 mins on average), likely because that the Sofa (307 graph nodes) has a much larger kernel disassembly graph than the Cube (11 graph nodes). After the user study, seven participants chose the Owl as the most attractive puzzle because of its cute appearance. Our user study confirms the necessity of using our computational tool to design interlocking puzzles such that their level of difficulty matches the ability of the user and their appearance can be attractive to the user. Please refer to the supplementary material for more details about our user study.

**8 CONCLUSION**

This paper presents a computational approach for designing high-level interlocking puzzles from an input voxelized shape. For this purpose, we propose a disassembly planner that is able to compute a given puzzle’s exact level of difficulty by enumerating all possible non-monotone, linear/non-linear disassembly plans to take out the first subassembly using a rooted graph data structure. We also present a computational framework that constructs the geometry of each puzzle piece iteratively guided by the disassembly planner, aiming to achieve a user-specified level of difficulty. To extend our approach for designing puzzles with smooth appearance, we formulate and solve a new shape optimization problem for creating voxelizations with a minimal number of problematic voxels. We demonstrate the effectiveness of our approach by designing puzzles with various shapes and topologies, show the advantages of our approach by comparing it with a baseline, and evaluate the difficulty of playing our designed puzzles in a user study.

**Limitations and future work.** First, our shape optimization focuses on minimizing the number of problematic voxels yet does not consider other requirements such as aesthetics and shape symmetry. Second, our puzzle design approach requires a large internal volume of the input shape for making the puzzle pieces interlocking in multiple configurations. Thus, it may fail for input shapes with no or small internal volume such as architectural shells and tree-like shapes. Third, we found that a high-level interlocking puzzle may be hard to play if the puzzle is not stable in some configurations. One possible way to address this limitation is to incorporate structural stability analysis [Wang et al. 2021] into our computational design.
framework. Lastly, our puzzle design approach assumes that the input shape is represented as a voxelization. In the future, we want to extend our approach to support designing high-level interlocking puzzles with other geometric forms such as Japanese Puzzle Boxes with planar parts and integral joints as well as Excalibur Puzzles with a big cube frame that holds the other small puzzle pieces.

ACKNOWLEDGMENTS
We thank the reviewers for the valuable comments, David Gontier for sharing the source code of the baseline design approach, Christian Hafner for proofreading the paper, Keenan Crane for the 3D model of Cow, and Thingiverse for the 3D models of Moai and Owl. This work was supported by the SUTD Start-up Research Grant (Number: SRG ISTD 2019 148), the Swiss National Science Foundation (NCCR Digital Fabrication Agreement #51NF40-141853), and the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (Grant Agreement No 715767 – MATERIALIZABLE).

REFERENCES

Computational Design of High-level Interlocking Puzzles • 150:15

naeis.pdf

ACM Trans. Graph., Vol. 41, No. 4, Article 150. Publication date: July 2022.