UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

# CS/IT Honours
# Final Paper 2019

**Title:** Generating 3D Interlocking Puzzle Pieces from 3D Arrays

**Author:** Nkosingiphile Gumede

**Project Abbreviation:**
PUZLOK

**Supervisor(s):** Prof.
James Gain

| **Category** | *Min* | *Max* | Chosen |
|---|---|---|---|
| Requirement Analysis and Design | *0* | *20* | 20 |
| Theoretical Analysis | *0* | *25* | 0 |
| Experiment Design and Execution | *0* | *20* | 0 |
| System Development and Implementation | *0* | *20* | 20 |
| Results, Findings and Conclusion | *10* | *20* | 10 |
| Aim Formulation and Background Work | *10* | *15* | 10 |
| Quality of Paper Writing and Presentation | *10* | | 10 |
| Quality of Deliverables | *10* | | 10 |
| Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*) | *0* | *10* | 0 |
| **Total marks** | | **80** | |

# Generating 3D Interlocking Puzzle Pieces from 3D Arrays

Nkosi Gumede
University of Cape Town
Department of Computer Science 2019

## ABSTRACT

This paper focuses on a technique for generating 3D (3-dimensional) interlocking puzzles. The technique is adapted from a research paper published in 2012 entitled 'Recursively Interlocking Puzzles' [10] which contains a 3D puzzle piece generating algorithm written by Peng Song, Chi-Wing Fu and Daniel Cohen-Or. Together they solved the mystery of the governing mechanics behind recursively interlocking puzzles. Their algorithm follows a top-down constructive approach which offers a much-needed speedup over pre-existing exhaustive search approaches. Such 3D puzzles have a large number of (potential) applications. For example, they can be used to create games and furniture which can be assembled and/or disassembled. This paper focuses specifically on the application of 3D interlocking puzzles to the fast-growing field of 3D printing.

Using 'Recursively Interlocking Puzzles' [10] and an existing framework; this paper unpacks the process of generating 3D recursively interlocking puzzles from 3D models. Our focus is primarily centred on coding the algorithm contained in section 5 of 'Recursively Interlocking Puzzles'. We discuss the process of implementing the algorithm and aim to offer suggestions to assist in speeding up the execution time of the algorithm as a whole. The method for evaluation is comparing their algorithm to our implementation in terms of both speed and accuracy. The necessity to measure speed comes from our desire to speed up the source algorithm. The necessity to measure accuracy comes from our desire to ensure that valid puzzle pieces are generated consistently.

## Keywords

Recursive interlocking; 3D puzzles; algorithms; computational geometry

## 1. INTRODUCTION

3D interlocking puzzles are puzzles with the following two attributes: 1) Interlockable (assembly and disassembly are required to solve the puzzle). 2) 3-Dimensional (the third dimension should constrain all generated puzzle pieces). The term 'voxel' is important to understand as it will be used throughout this paper. A voxel can is a volume element – a cube with all six sides being of equal length and breadth. Finding a valid interlocking puzzle given an enclosing volume for a target shape is a computational geometry problem that has already been solved by Song et al [10]. However, their solution requires a blocky outer surface aligned on a regular grid that could take up to 10 hours to generate given large input sizes. Since 2012, computing power has generally increased according to Moore's Law. This means that generating 3D interlocking puzzle pieces is a lot more feasible. This paper explores the Song et al [10] algorithm in detail; going from a relatively high-level of pseudocode to a lower-level which is easier to understand. 3D interlocking puzzles are relevant because they have many applications. Not only are they fun and interesting, but they can also be used to generate useful objects such as tables and chairs. We are only in the early stages of discovering their usefulness. Objects consisting of various components do not have a single point of failure. If the engine of a vehicle failed, for example, you would only have to replace the engine. Given a lower-level description of the code, we will discuss the parts of the algorithm which are computationally expensive and suggest ways to execute such programs faster using multiple cores and other high-performance computing techniques.

## 2. PREVIOUS WORK

There are three main general approaches to solving the problem of generating interlocking puzzles. Namely, the exhaustive search approach, the construction approach and the geometric design approach.

### 2.1 Exhaustive Search

The exhaustive search approach describes the naive method of generating puzzle pieces. The set of valid puzzle pieces are generated iteratively by going through the set of all possibilities. This approach was commonplace in the early days of puzzle generation. Due to its inefficiency, exhaustive searching has since been replaced by more efficient approaches in recent times. Little is known about the origins of interlocking puzzles. Many believe that they were invented by the Chinese, who used wood to build earthquake-resistant homes without nails. Edwin Wyatt, author of Puzzles in Wood [13], coined the term 'burr' to describe puzzles which resemble a burr seed. In 1997, IBM Research [3] launched the burr puzzles site. IBM Research defines burr puzzles as "consisting of at least three rods intersecting at right angles". Burr puzzle algorithms attempt to discover new interlocking puzzles based on exhaustive search. There are various types of three-piece and six-piece burr puzzles. The burr puzzles site extensively covers Bill Cutler's contributions to burr puzzles. Cutler [2] also covers previous attempts to discover new interlocking puzzles based on exhaustive search. Bill Cutler introduced a large number of six-piece interlocking structures. He shows that there are 314 ways to assemble 25 notchable pieces to make a six-piece burr puzzle by assessing all permutations through exhaustively searching for puzzle pieces which meet the criteria for interlocking. There are two approaches for constructing all assemblies via a computer program: 1) the program determines every set of six pieces and possible assemblies, then checks for a solution. 2) The program constructs each assembly of six pieces, analyzing it immediately. Approach 2 was selected as it saves time. Rover [7] created the Burr Tools computer software to help solve certain puzzle designs by trial-and-error (an exhaustive search approach). The program supports "square or dice shaped units, spheres, prisms with an equilateral triangle as base or 2 grids that use tetrahedra". The program allows users to construct puzzle pieces and target shapes visually. Users can then use the puzzle

solver to check for the number of assemblies, solutions and the time taken to complete a generated puzzle.

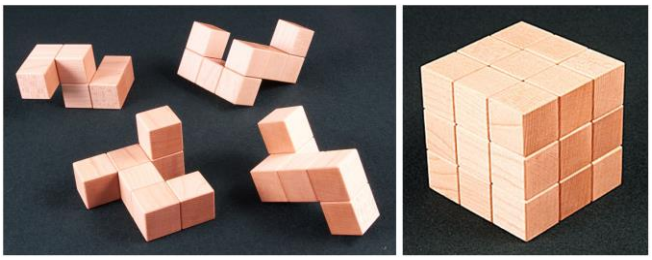

**Figure 1:** *Coffin [1]'s four-piece interlocking cube.*

## 2.2 Construction Approach

The construction approach refers to a method of generating interlocking puzzle pieces recursively from a source object. This can be done by dividing the source object into its constituent puzzle pieces (top-down approach) or by creating new puzzle pieces from the source object (bottom-up approach). Song et al. [10] explore a recursive top-down approach for devising new interlocking geometries that directly guarantee the validity of the interlocking instead of exhaustive testing it. The algorithm takes a voxelized shape as input and iteratively extracts puzzle pieces until the remaining part is the last piece. The algorithm requires local interlocking among the key piece (the first piece to be extracted), the secondary piece, and the remaining part. Wang et al. [12] present a general framework for designing interlocking structures which use Direction Blocking Graphs and analysis tools. They use an algorithm outlined in section 4 of their paper to design an interlocking assembly from a given shape by generating parts puzzle pieces and a remaining volume. Much like Song et al. [10], they generate the key piece and then generate the other pieces from the remaining volume. The main difference between the two approaches is the design of the graphs used to ensure interlocking – all previous parts are used as opposed to using only the most recently added part. Song et al. [11] delve deeper into the matter of generating interlocking puzzle pieces while focusing on their ability to be 3D printed. They take a watertight (consisting of no gaps/ holes) surface mesh as input and place a 3D grid within the object's space to voxelize it. They deform the geometry locally to cater to fragments. Neighbouring voxel shape connection strength is calculated by analysis local shapes and a graph is built. The important features of the model are realized on saliency graph. The construction of puzzle pieces is guided by the graphs, reassigning boundary voxels (for aesthetics) and constructive solid geometry (CSG) intersection. Interlocking does not necessarily have to be produced with voxelized pieces. This is demonstrated by Lau et al. [5] who converted furniture models into parts and connectors using lexical and structural analysis. Their algorithm uses lexical analysis to identify primitive shapes (processed as tokens). Then, structural analysis to generate fabricatable parts and connectors automatically. The output of their algorithm can also generate user manuals on how to assemble furniture from constituent components via the bottom-up approach.

## 2.3 Geometric Design Approach

Stewart Coffin is widely regarded as the world's best designer of interlocking puzzles. In 1990, he produced a book [10], which described interlocking cubes. His book motivated the study of geometric mechanics producing interlocking. The geometric design approach refers to using lines and curves generated by a mathematical equation to generate puzzle pieces. Lo et al. [6] used the geometric design approach to generate 3D polyomino puzzles. They created shell-based (comprising of an interlocking surface) 3D puzzles with polyominoes as the component shape of the puzzle pieces. First, apply quad-based surface parametrization to the input solid and tile the parametrized surface with polyominoes. Then, construct a tiled surface inside the parameterized shape to fit inside a thick shell volume. Finally, they use associated geometric techniques to construct puzzle pieces (polyominoes generated via Procedure 1 outlined in section 2 of their paper). Xin et al. [13] also explored the governing mechanics of interlocking puzzles using geometric methods. They replicated and connected pre-defined six-piece burr structures to create larger interlocking puzzles from 3D models. First, they embed a network of knots into a given 3D model. Then, the 3D model is split according to its geometry. The method requires one to first manually design and construct a grid-based graph inside a given 3D shape. The method can put a six-piece burr puzzle at each grid point and connect them to guarantee the interlocking. Their paper describes both a single-knot and multi-knot burr puzzle. Zhang and Balkcom [14] explore a solution to assemble voxelized interlocking structures using joints (pairs of male-female connectors on adjacent voxels) to guarantee interlocking and assembly order. The algorithm breaks models into layers and sequentially builds layers using block types to restrict the mobility of puzzle pieces.

# 3. SYSTEM FRAMEWORK

The general flow of our system consists of the following seven steps: **1) Obtain a triangle mesh object.** This triangle mesh object is a 3D model of an object such as a sphere or bunny represented in a '.stl' file. Such a file can be obtained for free online. **2) Voxelize the object.** Voxelization describes the process of representing an object as a collection of voxels. The process of voxelization is accomplished by my project partner, Dominic Ngoetjana, with the assistance of a 3D model renderer supplied by our supervisor. The process of voxelization is accomplished by aligning a 3D object on a 3D grid and setting voxels where space in the 3D grid is predominantly covered by the 3D object. **3) Apply the relevant puzzle generating algorithm.** In this case, the relevant puzzle generating algorithm has been determined to be Song et al. [10]. Section 5 of this paper details how this algorithm is applied in the context of our system. **4) Generate multiple voxel pieces.** The output of the previous step (3) is a set of puzzle pieces which must be served back into the renderer for processing. **5) Triangulate the voxel pieces**. Each voxelized puzzle piece is served back into the renderer and must be triangulated for the sake of 3D printing. Triangulation is the process of converting the surface of any 3D object to many tiny connected triangles. This must be accomplished for the purpose of 3D printing. **6) Add a triangulated outer surface (optional).** A triangulated outer surface is added on top of the voxelized puzzle piece to give our final 3D object a smooth (as opposed to blocky) outer surface. **7) 3D-print the puzzle pieces.** Finally, we represent the triangulated 3D object in a '.stl' file format which is recognized by 3D printers and 3D print all puzzle pieces. All steps mentioned above are to be integrated into one final system. Part 3 (applying the relevant puzzle generating algorithm) is the focus of this paper while the rest of the work is allocated to

Dominic Ngoetjana. Figure 2 below provides an overview of the entire system.

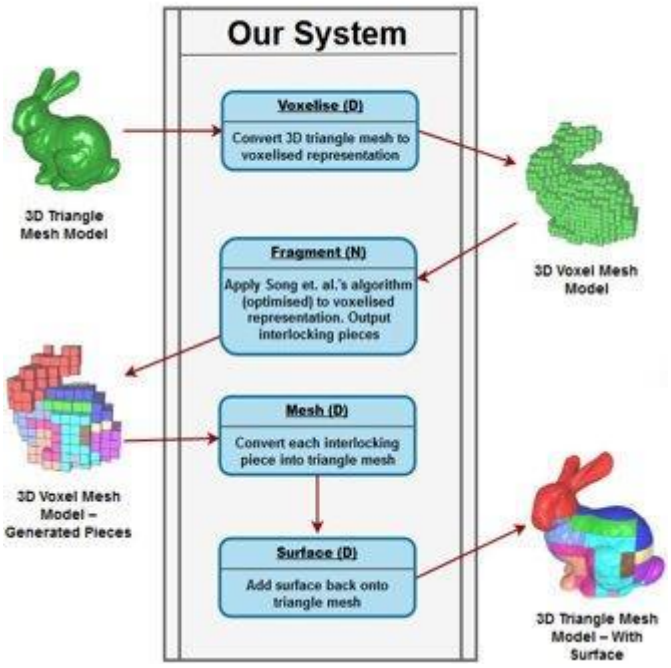


**Figure 2:** *A visual representation of the entire system. D (Dominic) represents the responsibilities of my project partner. N (Nkosi) represents my responsibilities.*

# 4. SOURCE ALGORITHM

Section 5 of Song et al. [10] features an innovative approach for generating recursively interlocking puzzle pieces. Their work follows two main procedures: 1) Extracting a key piece from an input voxelized 3D mesh. 2) Iteratively extracting all of the other puzzle pieces from the remaining volume.

## 4.1 Extracting of Key Piece

The first part of the algorithm described by Song et al [10] follows five main procedures: 1) Pick a seed voxel. 2) Compute voxel accessibility. 3) Ensure blocking and mobility. 4) Expand the key piece. 5) Confirm the key piece. These five produces are explained in detail below.

### 4.1.1 Pick a seed voxel

The first step is to identify a set of candidate seed voxels which meet the following criteria: 1) they must be at the top exterior of the puzzle. 2) They must have exactly two exterior and adjacent faces. From that set of candidate voxels, we either randomly select a seed or let the user make a choice.

### 4.1.2 Compute voxel accessibility

To avoid fragmenting the remaining volume after extracting a key piece, we compute accessibility values for every voxel. The process of computing accessibility values is done via three passes. In the first pass, we set the accessibility value of each voxel to the sum of its neighbours. That is, for each neighbour a voxel has (left, right, up, down, forward or backward), its neighbour count is increased by one. Once the first pass is over and all voxel accessibility values have been calculated, we will execute two more subsequent passes through all voxels in the object. In the subsequent passes, the accessibility values of each voxel are equal to the current accessibility value of the current voxel plus the sum of the accessibility values of its neighbours. The accessibility values indicate how each it is to visit a voxel via its neighbours.

### 4.1.3 Ensure blocking and mobility

At this point, we should be ready to develop the key piece. A key requirement is that this piece should be removable in one direction; we will call this the normal direction. The key piece should be blocked in all other directions. The normal direction will be set to the direction of the outward-facing exterior which is adjacent to the upward-facing exterior. After this, we do a breadth-first traversal from the seed to all other voxels to find 50 pairs of voxels closest to the seed. Each pair will consist of a blocking and blockee voxel. The blocking voxel is a voxel which will prevent the puzzle piece from moving out of the puzzle. The blockee voxel is a voxel in the puzzle piece which is adjacent to the blocking voxel which allows the piece to be blocked. The blocking and blockee voxel pairs will be on the positive and negative sides of the normal direction respectively. Among these 50 pairs, we will collect 10 voxel pairs whose blockee has the smallest accessibility value. Next, the goal will be to expand the piece in a way that it is blocked from moving out of the puzzle along the normal direction. We will accomplish this with the following three steps. 1) Via breadth-first traversal, determine the shortest paths from the seed voxel to each blockee voxel without crossing the blocking voxel or any other voxels which are positioned beneath it. This path will represent the voxels in the key piece. 2) Make this key piece removable by adding all the voxels above the path. Ignore the shortest path candidates generated from step 1 which lead to creating a removable key piece with an excessive number of voxels. This is done by presetting a variable (m) to the average number of voxels per puzzle piece. Alternatively, the variable m can be calculated by determining the number of puzzle pieces we wish to generate. The number of voxels per puzzle piece is equal to the total number of voxels in the puzzle divided by the number of puzzle pieces. The removable key piece should contain less than m voxels. 3) Now that we have a removable key piece with less than m voxels and might mobilize the piece by adding voxels in previously blocked directions; we will identify an anchor voxel for each one of the directions blocking the seed. The purpose of the anchor voxels is to ensure we do not expand the piece using a voxel which would make it removable in any other directions. The anchor voxel for each direction is set to the voxel furthest away from the seed along that direction, keeping all other axes constant. Anchor voxels should be added to the key piece.

### 4.1.4 Expand the key piece

At this point, we should be in a position to expand the key piece using information yielded from the afore-mentioned three-step procedure. From the shortest path candidates, we will sum the accessibility values of all voxels in each shortest path and set key piece to the shortest path with the smallest sum of voxel accessibility values. Since it is still expected to be less than m voxels in size, the following three steps should be applied iteratively to expand the key piece until it is the appropriate size (m): 1) Store an additional anchor voxel relative to the blocking voxel in the same manner as before; it will be the furthest voxel directly-connected along the direction blocked by blocking voxel. If such a voxel does not exist, set the additional anchor voxel to the blocking voxel itself. 2) Store the set of voxels which neighbour the current key piece but are neither the anchors nor below the anchors. For each voxel, also store the set of voxels above it on the same axes. If the voxels added are in excess of the number required by m, remove them. This stored set of voxels represents the candidate voxels. 3) Finally, we will add candidate voxels to the key piece on the basis of probability until the key piece is the appropriate size. First, we take the sum of each candidate voxel and the voxels above it. Then, we set a Beta value to any integer between 1 and 6. We use this Beta value to get a normalized sum of accessibility values (Pi) by calculating negative Beta raised to the power of the sum of accessibility values. As we calculate each Pi, we add it to a cumulative total. The probability value from each candidate voxel is equal to its Pi divided by the sum of all Pis (i.e. the cumulative total). Now we can randomly pick voxels to add to the key piece by repeating the three steps outlined above until there are roughly m voxels.

#### 4.1.5 Confirm the key piece

To ensure the validity of the key piece, we must ensure the remaining voxel (voxels not in the key piece) are simply connected. This can be done by applying a flooding algorithm which checks if all voxels are reachable via neighbouring voxels. If so, given that the key piece generation process has been followed correctly, we can confirm that the key piece is valid.

### 4.2 Extracting the Other Puzzle Pieces

Much like to first part of Song et al. [10], the second part of the algorithm also follows 5 main procedures: 1) Select candidate seed voxels. 2) Create an initial next piece. 3) Ensure local interlocking. 4) Expand the next piece. 5) Confirm the next piece.

#### 4.2.1 Select candidate seed voxels

The next piece should be blocked by the key piece as required by the property of being recursively interlocking. The candidate seed voxels, in this case, are voxels which neighbour the key piece in a direction perpendicular to the direction in which the key piece is removable. The set of candidate seeds will be reduced to a maximum of ten via the two equally-weighted criteria: 1) voxels with smallest accessibility values. 2) The shortest distance between the candidate seed and furthest voxel along the perpendicular direction.

#### 4.2.2 Create an initial next piece

To determine which seed voxel will be picked, we must determine the cost of making the proposed next piece removable along the perpendicular direction by following three steps: 1) Identify all voxels (in the remaining volume) from the candidate seed to furthest-away voxel along the perpendicular direction. 2) Connect the candidate seed to the closest voxel among those identified in step 1. 3) Add all the other voxels required to make the next piece removable along the perpendicular direction. Store all the voxels identified by the steps above and calculate the sum of their accessibility values according to the same formula used in section 4.1.2. The chosen seed candidate should have the smallest sum of accessibility values. The seed and all the other voxels identified above constitute our next piece.

#### 4.2.3 Ensure local interlocking

Since the next piece is already blocked in one direction, we must now ensure that we block the other five directions. We must also ensure that the absence of the key piece does not render the next piece removable in more than one direction. If any voxel from either the key piece or the new remaining volume neighbour the next piece along the perpendicular direction, the next piece is considered blocked (or not removable). If the next piece is removable along an undesired direction (a direction other than the perpendicular direction), we apply the strategy (of using anchor, blocking and blockee voxels) described in section 4.1.3.

#### 4.2.4 Expand and Confirm the next piece

The process of expanding the next piece to roughly m voxels and confirming it is identical to that of the key piece. Both processes are previously described in sections 4.1.4 and 4.1.5 respectively.

## 5. IMPLEMENTATION OF THE SOURCE ALGORITHM

Our implementation of Song et al. [10] was coded from scratch using Java. We created two driver classes to called Puzlock and Puzlock2 to represent the implementation of subsections 5.1. and 5.2 respectively. We use the 3D integer array data structure to represent a voxelized input object. Credit goes to my Dominic Ngeotjana for adding the functionality to input and output text files consisting of 1s and 0s (based on a 3D model) to the renderer. The idea was to use a text file as the input to the program. This text file would then be used to generate dynamic 3D arrays of integers. We created an IO (Input-Output) class to convert input text files to 3D integer arrays and vice-versa. We used the renderer to visualize the output of my program. The output was a set of text files written in the same format as the input test files. These output text files were used to represent either puzzle pieces or the remaining volume. Visualization via the renderer aided the process of debugging, allowing me to compare my expectations with the actual output. The class diagram below provides a visual representation of the entire Puzlock system.

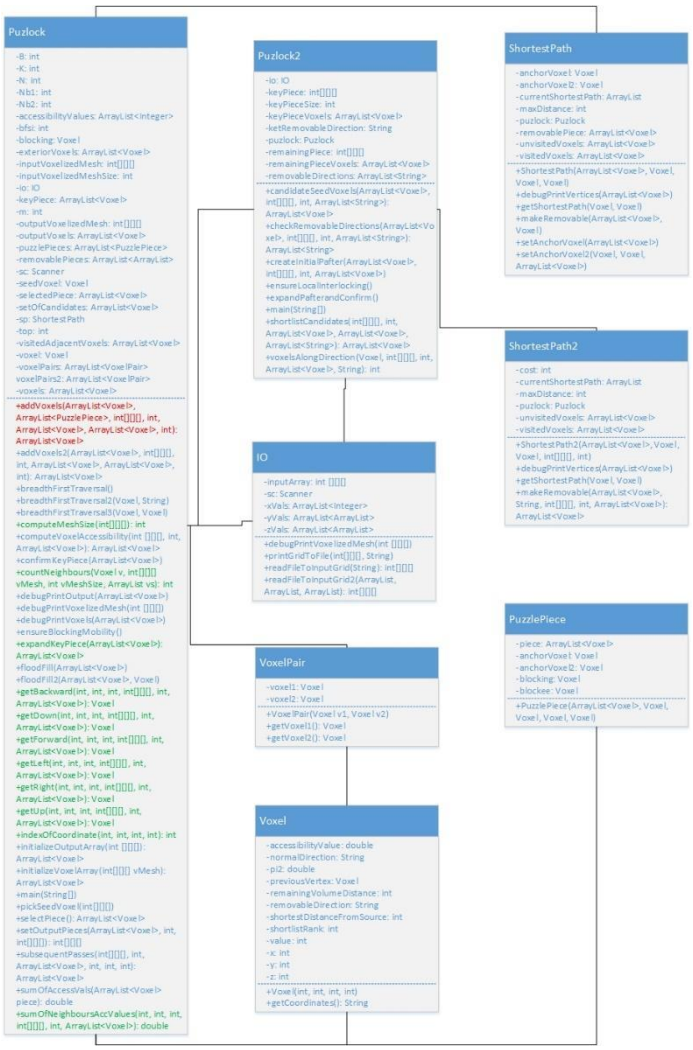


**Figure 3:** *Class diagram showing all classes in the Puzlock java program and their associations. The methods written in green represent methods which passed Junit testing and the methods written in red represent methods which failed Junit testing.*

### 5.1 Extraction of Key Piece

The Java program we created consists of a method which reads in a file consisting of 0s and 1s to initialize a 3D array of integers. This 3D array of integers is used to initialize an ArrayList of voxels. To represent a voxel, we created a Voxel class which stores the x, y and z coordinates of the voxel as well as its value. The value 1 represents the presence of a voxel and 0 represents its absence. We use the main method to initialize all global variables, including those which are generated from user input such as the average size of a puzzle piece (m) and Beta. Please note that the voxel at the most extreme left, top and backward position is always at position 0, 0, 0 (x-coordinate, y-coordinate and z-coordinate respectively) when represented on the 3D array. The voxel at the most extreme right, bottom and forward position is always at position n, n, n; where the integer n is the size of the 3D array minus one. We use a 4x4x4 3D array is the default input mesh. The main calls each method described in the five sections below.

### 5.1.1 Pick a seed voxel

We begin the process of extracting a key piece by picking a seed voxel. To do so, we created a method called pickSeedVoxel which iterated through all voxels in the ArrayList and stores the all voxels which meet the following criteria: 1) the voxel must be at the top-most position of the 3D array (which means it should have the lowest possible y-coordinate). 2) The voxel must not have a neighbour at the either left, right, forward or backward direction. If the voxel does not have a neighbour in the direction we are currently checking, we set its normal direction to the current direction. To check whether there is a neighbouring voxel in one of the six directions (left, right, up, down, forward or backward), we use six methods which take in a voxel's coordinates as parameters and return either the neighbouring voxel (with a value set to 1 if present or 0 if absent) or null (if a voxel object was not created in that direction). These methods are appropriately named 'getLeft', 'getRight', 'getUp', 'getDown', 'getForward' and 'getBackward'. We will hereafter refer to these six methods as 'get<Direction>'. If a voxel meets both criteria, we check to see if it is already contained in the candidate set of seed voxels (called exteriorVoxels). If it is already contained, it is set to be removed from the candidate set of voxels by adding it to an ArrayList called duplicates. If it is not already contained, it is added to the candidate set of seeds. From the candidate set of seed voxels, we randomly pick a seed by generating a random number limited by the range of indices in the candidate set of seed voxels.

### 5.1.2 Compute voxel accessibility

We compute an accessibility value for each voxel in the input mesh (represented with a 3D array) to reduce the chance of having a fragmented remaining volume after the key piece has been extracted. We do so with three (as suggested by Song et al. [10]) passes. In the first pass, we use a triply nested for loop to go through all positions in a 3D array. We use a method called 'countNeighbours' to count how many neighbours each voxel has using get<Direction> for one of the six 3D directions. In the first pass, the accessibility value of each voxel is set to the number of neighbours it has. A method called 'subsequentPasses' is then called to compute the accessibility value of each voxel in the remaining passes. The subsequent passes method contains a quadruply nested for loop. The outer for-loop is for the current pass. The inner triply nested for-loop is used to iterate through the positions of the 3D array. Within this loop, we iteratively (for each subsequent pass, denoted by 'p') calculate a new accessibility value for each voxel which is equal to its current accessibility value plus the product of two variables called 'power' and 'sum'. 'Power' is equal to the weight factor (set to 0.1 in Song et al. [10]) to the power of 'p' (the current subsequent pass number). 'Sum' is calculated by a method called sumOfNeighboursAccValues. It is the sum of the accessibility values of neighbouring voxels (set during the previous pass). It is important to note that the accessibility values should always be set using the accessibility values from the previous pass; we distinguish between the accessibility values of the current pass and the previous pass using two ArrayLists of voxels called 'newvoxels' and 'oldvoxels' respectively.

### 5.1.3 Ensure blocking and mobility

Now we have to ensure that the key piece is removable in one direction and blocked in the other five directions. We use a 'breadthFirstTraversal' method to store fifty blocking and blockee voxel pairs. The blocking voxel is used to block the key piece in the removable ('normal') direction. The blockee voxel is the voxel in the key piece which is blocked by the blocking voxel. These two voxels are therefore always positioned next to each other such that the blocking voxel has at one exterior. To conduct the breadth-first traversal, we keep track of the voxels which have already been visited in an ArrayList called 'visitedAdjacentVoxels'; making to traverse from the neighbours which have not already been visited in each iteration. The current voxel is initially set to the seed voxel. We use the normal direction to determine which direction the blockee voxel should be in relative to the blocking voxel. For example, if the normal direction is left, then the blockee voxel is always set to the right of the blocking voxel. A class called VoxelPair is used to represent each blocking/ blockee voxel pair. Once we have stored our fifty voxel pairs, we create another ArrayList called 'accVals' to store the accessibility of each blockee voxel in each voxel pair. We sort the 'accVals' list from the smallest accessibility to the largest and then set the tenth element of 'accVals' to the maximum accessibility value. We then iterate through the fifty voxel pairs again and store the first ten voxel pairs whose blockee voxel has an accessibility value less than or equal to the maximum accessibility value in yet another ArrayList called 'accessibleVoxelPairs'.

Now that we have our ten least accessible voxel pairs (in 'accessibleVoxelPairs), we can begin the process of ensuring blocking and mobility. To do so, we use the 'ShortestPath' class to determine the shortest path from the seed to each blockee voxel contained in 'accessibleVoxelPairs' without crossing the blocking voxel. The ShortestPath class uses Djikstra's single-source shortest path algorithm to determine the next voxel each voxel should visit we wish to go from the seed to the blockee voxels. Once the shortest path is determined, we also add all voxels above each voxel in the shortest path (using a method called 'makeRemovable') to make the path (which represents the proposed key piece) removable in the upward direction. We store this removable piece in an ArrayList of voxels called 'removablePiece'. To prevent the removable piece from being mobile in more than just the upward direction, we use the 'setAnchorVoxel' method of the 'ShortestPath' class to set an anchor voxel. The anchor voxel is a voxel in the remaining volume which is directly-connected and furthest-away from the seed voxel along each direction initially blocked direction by the seed. The idea is that each anchor voxel should prevent the key piece from being removable in more than just the upward direction. Please note that our current implementation of this part is incorrect; although the newer versions of our program attempt to fix it. We use the same strategy to determine an additional anchor voxel to block the direction blocked by the blocking voxel. If such a voxel cannot be determined among the set of voxels available along the normal direction, we set the additional anchor voxel 'anchorVoxel2' to the blocking voxel itself. Once the removable piece, as well as the set of anchor voxels, have been determined for each one of our ten voxel pairs, we use the PuzzlePiece class to store a new puzzle piece with the associated parameters (such as the set of anchors as well as the blocking and blockee voxels) in the Puzlock class. To select a piece among our ten removable pieces, we use the 'selectPiece' method. This method returns the removable piece with the smallest sum of voxel accessibility values. The 'selectPiece' method is called in the main method to allow for the selected piece to be passed as a parameter to the 'expandKeyPiece' method.

### 5.1.4 Expand the key piece

At this point, we wish to expand the selected piece such that it contains the appropriate number of voxels. We use a method called 'addVoxels' to determine a set of candidate voxels to be added to the key piece. For each voxel in the selected piece, an added candidate is a neighbouring voxel not already in the key piece nor the set of candidates. The 'addVoxels2' method adds voxels to the selected piece on the basis of probability. Firstly, it calculates the sum of accessibility values of each voxel as well as the neighbours above it using the 'sumOfAccessVals' method. This sum is raised to the power of negative 'beta' (where 'beta' has been set to three as per Song et al's [10] recommendation of setting it to an integer between one and six) and stored in a variable called 'pi'. The result of each iteration is added to a cumulative total called 'sumOfPis'. The set of each voxel and the voxels above it are stores in an ArrayList of voxels called 'setOfUpNeighbours'. Then, we iterate through the 'setOfUpNeighbours' to determine set a probability value for selected a voxel among the candidate set of voxels. The probability of each candidate is calculated by taking 'pi' and dividing it by the 'sumOfPis'. Finally, we add a voxel to the selected piece by generating a random integer in the between zero and 'sumOfPis' and add the candidate at the index of the random integer value to the selected piece iteratively until we have either have a selected piece with m voxels or reach our iteration limit of fifty.

### 5.1.5 Confirm the key piece

We confirm the key piece by determining a remaining volume (all voxels in the input mesh which are not in the key piece) and ensuring that all voxels contained in the remaining volume are reachable from source voxel using a flood fill algorithm. The 'floodFill' method stores the set of voxels which have not been visited in an ArrayList of voxels called 'unvisitedVoxels'. It uses 'get<Direction>' to visit neighbouring voxels iteratively until no unvisited voxels remain. It there are no unvisited voxels, we can confirm the key piece as this means that the remaining volume is not fragmented. At this point, we also print the remaining volume to a file called 'setOfNeighbours' for the purpose of visualization. The key piece is then stored in a 3D array called 'outputVoxelizedMesh' and printed out to a file called 'keyPiece' for the purpose of visual debugging. Visualization aids the debugging process and is accomplished on the renderer due to functionality added by Dominic Ngoetjana. The output files can be used by Dominic in the subsequent stages of the processes accomplished by our system. This concludes the process of generating a key piece from an input mesh represented as a 3D array.

## 5.2 Extraction of Secondary Pieces

The extraction of secondary pieces can be seen as a separate process entirely. Ideally, we wish to use the valid key piece and remaining volume, generated from the first part as input this part of the program. However, since we could not confirm a valid key piece in the previous section, we hard-coded a valid key piece aa well as the remaining volume. The implementation of this section is contained in the class called 'Puzlock2'. Each one the five steps below are called from the main method.

### 5.2.1 Select candidate seed voxels

The first step in determining the set of the candidate seed voxels is determining the set of directions in which the key piece is removable. To do so, we use a method called 'checkRemovableDirections'. The checkRemovableDirections method works by going through each voxel in the key piece iteratively and using 'get<Direction>' to check whether there the voxel to the direction we are checking (left, right, up, down, forward and backward) is either null or already contained in the key piece. If this is the case for all voxels, we add the direction we checking to the list of removable directions via storing the direction is an ArrayList of strings called 'movingDirections'. If the key piece is removable in only one direction (meaning there is only one string in 'movingDirections'), it can be confirmed to be valid. This direction is stored in a variable called 'keyRemovableDirection'. The set of candidate seed voxels are voxels which neighbour the key piece in a direction perpendicular to the 'keyRemovableDirection'. We use the 'candididateSeedVoxels' method to store the candidate seeds which meet these criteria in an ArrayList called 'setOfNeighbours'. For each voxel in the key piece, we check if there is no key piece set in each direction perpendicular to the 'keyRemovableDirection'. If this is true, we set its removable direction to the direction opposite the direction we are checking and add the neighbouring voxel to the 'setOfNeighbours'. Each voxel's removable direction is stored in a variable called 'removableDirection'.

To reduce the set of ten neighbours to ten candidate seed voxels, we use a method called 'shortlistCandidates'. Within this method, we compute the accessibility value of all voxels in the remaining volume and store the accessibility value of all ten corresponding candidate seed voxels in an ArrayList of doubles called 'accVals'. We then sort 'accVals' from smallest to largest. We then iterate through each shortlisted seed voxel and count the number of voxels from the current voxel to last voxel along its 'removableDirection'. The count of voxels along each voxel's 'removableDirection' is stored in a voxel attribute called 'remainingVolumeDistance' and in an ArrayList of integers called 'distances'. We then sort 'distances from smallest to largest. The shortlist of ten candidates is determined by equally-weighted criteria such than each shortlisted candidate must have the smallest accessibility value as well as the shortest distance to the furthest-away voxel along its 'removableDirection'. We, therefore, rank each candidate as follows: 1) we iterate through 'distances' and store the index of 'distances' according to the current voxel's 'remainingVolumeDistance' in yet another voxel attribute called 'shortlistRank'. 2) We update 'shortlistRank' by iterating through the indices of 'accVals' and adding the index of the corresponding voxel. We store the ranking of each voxel as per its 'shortlistRank' in an ArrayList of integers called 'rankings'. We sort 'rankings' from smallest to largest. Finally, we iterate through our set of 'rankings', adding the voxels with a corresponding 'shotlistRank' to our shortlist of ten candidate seed voxels.

### 5.2.2 Create an initial next piece

In creating the initial next piece, the goal is to pick the next piece with the smallest sum of accessibility values among a set of candidates. For each voxel in our shortlisted set of seed voxels, we traverse through the voxels in the remaining volume along the current voxel's 'removableDirection' and get the distance ('cost') of the path between the seed and current neighbour. This is accomplished with a doubly-nested for-loop. The 'ShortestPath2' class is used to calculate the distance and stores it in a variable called 'cost'. We iteratively set the current 'cost' to the lowest cost if it is less than the lowest cost which is initialized to an integer greater than the size of the input mesh. We also store the seed and neighbouring voxel associated with the lowest cost. We then determine the path between the stored seed voxel and neighbouring voxel with yield the lowest possible cost. This path is made removable along the stored seed voxel's 'removableDirection' to generate an ArrayList of voxels which is appropriately named 'removablePiece'. This piece represents our initial next piece. We print this removable piece to a file for the purpose of visual debugging.

### 5.2.3 Ensure local interlocking

Using the 'checkRemovableDirections' method, we check to see if the other five directions are blocked. The only difference is that in this case, we pass the next piece as well as a combination of both the key piece and the remaining volume as parameters. Since the next piece should not be co-movable with the key piece, we also check to ensure that the next piece is not removable in the removable direction of the key piece. If the next piece is removable along an undesired direction, we use the strategies described in subsection 5.1.3 to ensure blocking and mobility.

### 5.2.4 Expand and Confirm the next piece

The process of expanding each subsequent piece to the appropriate size and confirming it is conducted in a process which is identical to the process described in subsection 5.1.4. From the 'Puzlock2' class, we use a 'Puzlock' object to make calls to the 'expandKeyPiece' and 'confirmKeyPiece' methods of the 'Puzlock' class.

## 5.3 Limitations

During the process of coding my implementation of the algorithm contained in section 5 of Song et al [10], I discovered a set of limiting factors. These limitations included the inability to manipulate a visual representation, lack of time (or perhaps misused time spent on implementing elements which were out of scope such as writing pseudocode and C# code to visualize puzzle pieces in the Unity 3D IDE), and the inability to handle different input types. Each one of the limiting factors is further explained below. Although I could visualize puzzle pieces using the renderer, I was unable to manipulate the visualization of the output. This inability made it difficult to determine the relative positioning of extracted puzzle pieces. My project partner and I spent a few days implementing a C# method consisting of a triply nested for loop to visualize a voxelized input object represented in a 3D array of integers. I spent a week more than expected creating unit tests for every method with a return type contained in the Puzlock class. This unit testing proved valuable though as it allowed me to debug minor errors and refactor my code. From unit testing, I learned, for example, that it was best practice to parse variable into methods as opposed to adjusting static variables which may not have been initialized. Should time permit, I will

implement more unit testing to ensure my program produces what is expected in every constituent method. Since I had initially started coding the algorithm with a preset 3D array consisting only of 1s. My program cannot handle inputs with 0s padded on any side of the 1s. My program expects the starting position of the voxelized input object (represented as 1s) to be at the top and left ends of the input text file. Due to time constraints, I continued my implementation and testing with the initial 4x4x4 3D integer array of 1s. Ambiguity hindered my understanding of the paper. For certain issues, I had to ask my supervisor for clarification on what a previously unexplained symbol represented for example.

## 6. RESULTS

There are multiple ways in which the algorithm outlined and documented in [10] can be implemented and optimized. From the feedback of my demo presentation, I was confronted with the decision to choose among a variety of programming languages. C++ is commonly known as one of the fastest programming languages currently. Although Python uses external libraries extensively, it is arguably the simplest, which helps in terms of speed of development. The Unity 3D IDE (integrated development environment) utilizes C#. Coding in C# would help me visualize the output of my program during the development phase. Ultimately I settled on coding the algorithm in Java. Java's familiarity would assist me to write code quickly as opposed to having to endure the process of learning a relatively unfamiliar programming language. Despite the speed of development, there exists the problem of the speed of the algorithm itself. Java is said to be slower than C++ but its code can be made to run quicker by using multithreading on multicore computer architectures. The idea was to spend the last portion of my allocated time parallelizing the algorithm using known procedures and consult on how to run the parallelized algorithm on a high-performance computer using OpenMP. Figure 1 shows the necessity to speed up the process of generating recursively interlocking puzzle pieces. When K (the number of puzzle pieces) is 150 and N (the number of voxels) is 30x29x23, Song et al. [10]'s algorithm takes 534.43 minutes to run. The time complexity of this algorithm was not provided.

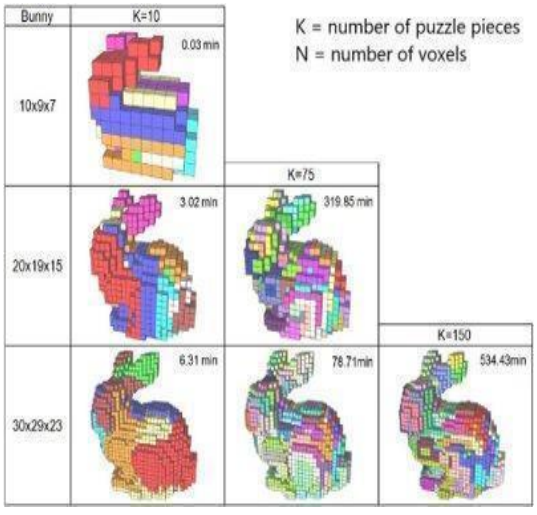


**Figure 4:** *A table adapted from Song et al. [10] showing how the time taken to run their algorithm grows with an increase in the number of puzzle pieces (K) and voxels.*

## 6.2 Accuracy

A 'PuzlockJUnitTests' class is contained within the 'test' folder. This class contains twelve unit tests for the seventeen unique methods with return types contained in the 'Puzlock' class. JUnit testing is the basis for determining the accuracy of our implementation relative to what is expected from Song et al [10]. Unit testing allows us to check for any discrepancies between what a program produces and what it is expected to produce. Out of our twelve unit tests, eleven passed and only one failed. The variable used as input and output to the program were hard-coded in each case. These variables serve the purpose of allowing out to understand what to expect as output when our program is given certain variables as input. If a unit test fails, it is most likely due to an error in our implementation of the algorithm or an error in understanding how the algorithm works. Ideally, a finished program would contain a unit test for every single method which returns an object. However, since time did not permit, we could not develop comprehensive and successful unit tests for each one of our five classes which require them.

Through a combination of unit testing and acquiring an improved understanding of how Song et al [10] works over time, we have established the reasons as to why the Puzlock class fails to derive a valid key piece successfully. The problems affecting the accuracy of our program are as follows: 1) in subsection 5.1.3, when attempting to ensure the blocking and mobility of a key piece, we are yet to adjust the code such that the removable piece is determined before the anchor voxels are set. The anchor voxels are part of the key piece although they should not be. We are also yet to implement the functionality to ensure that excessive voxels are not added to the removable piece. 2) In subsection 5.1.3, our program only checks for blockage along the seed voxel's removable direction. We are yet to recognize that there should be a set of anchor voxels, one for each removable direction. The set of removable directions can be established using the 'checkRemovableDirections' method of the 'Puzlock2' class, however, complications arose when attempting to utilize this functionality. The program began to behave unexpectedly, requiring further debugging. 3) When expanding the key piece, we are yet to add a for-loop which allows us to repeat the steps contained in subsection 5.1.4 to make the key piece the appropriate size. 4) Due to a previous misunderstanding, the general sequence of events contained in our implementation of subsections 5.1.3 and 5.1.4 is incorrect. The 'Puzzle Piece' class should not contain an 'anchorVoxel2' variable should not be initialized at this point. 5) Subsection 5.2.3 and 5.2.4 of this paper are yet to be realized. Although they predominantly consist of utilizing functionality already contained in the program, we failed to implement these methods on time. The five points above tremendously affect the accuracy of our implementation as a whole. However, corrections are not expected to change the time complexity of our program. It helps that we realize the shortcomings of our implementation. We could make the program increasingly accurate over time. Below, is a screenshot of a key piece generated by our program given a 4x4x4 3D array as input. Our program normally generates a different key piece with each run as expected.

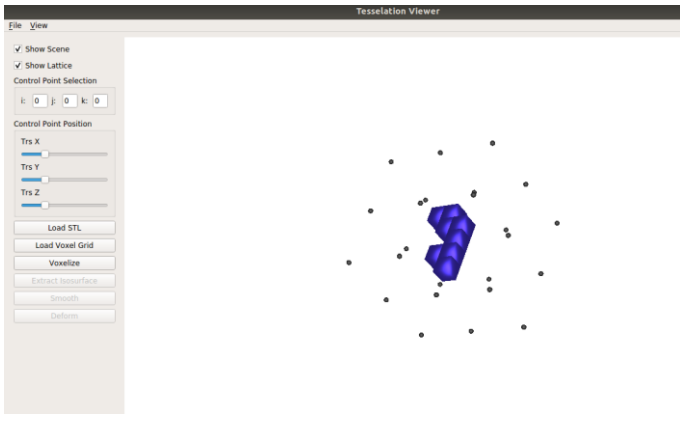


**Figure 5:** *A visualization of a key piece generated by our implementation on the renderer supplied. This key piece was generated from a 3D array of 1s and 0s, later represented in a file format recognized by the renderer.*

## 6.3 Speed

The most expensive series of computations contained in our program is a quadruply nested for-loop. This for-loop is used in the 'subsequentPasses' method of the Puzlock class when calculating the accessibility value of every voxel in the input 3D array. This means that

our program has a time complexity of O($n^4$). Unfortunately, the Song et al. [10] paper contains no mention of time complexity in their results. The only metric left for us to use as a basis for comparison is time taken to generate puzzles. The table below shows the execution times (in minutes) of the Puzlock and Puzlock2 classes relative to identical cubes of varying sizes contained in Song et al. [10]:

| **Cube Dimensions** | **Song et al. [10]** | **Puzlock** | **Puzlock2** |
|---|---|---|---|
| **5x5x5 (K=10)** | 0.7 minutes | 0.0167 minutes | 0.033 minutes |
| **15x15x15 (K=100)** | 3.03 minutes | StackOverflowError at 'breadthFirstTraversal' method | n/a |
| **25x25x25 (K=500)** | 110.7 minutes | StackOverflowError at 'breadthFirstTraversal' method | n/a |

**Table 1**: *A table showing the execution times (in minutes) of Song et al [10]'s implementation, our implementation, and the difference between them where K is the number of puzzle pieces in the puzzle.*

It is important to more than the speed of execution is likely to have been affected by Moore's Law. There is no mention of the computer used to run the Song et al. [10] algorithm. The computer used to run our implementation is an Acer Travelmate laptop with an Intel Core 2 Duo processor and four gigabytes of RAM. Faster computers are expected run our implementation is less time. To accurately compare the speed of two or more programs, one should run them on the same computer at least five times and get the average of their execution times. Due to implementation being incomplete and us not having access to metrics for a fair comparison, any comparison of speed is not useful. We cannot conclude on whether we have accomplished any speedup over the original algorithm.

## 7. CONCLUSIONS

Interlocking puzzles are a largely unexplored topic with many possible applications. This paper describes the coding process undertaken to derive 3D interlocking puzzles using 3D arrays as the primary data structure. It helps bridge the gap between a high-level and lower-level algorithm which is easier to understand from a programming perspective. The Song et al. [10] technique of generating recursively interlocking puzzles is fairly complex and requires a lot of explanation. One cannot understand it by simply reading their paper once or twice. Although we failed to generate a set of valid key and secondary puzzle pieces, we are well aware of the errors we made in understanding and implementing their technique. Lack of time was the main drawback. The primary objective of 3D printing valid puzzle pieces can still be accomplished by hard-coding puzzle pieces in 3D arrays. The IO class would assist us in using these 3D arrays to generate files which are served as input to the renderer for post-processing. Going forward, we hope to improve the accuracy of the implementation described in section 5 of this paper. Once this implementation yields valid sets of key and secondary puzzle pieces, one may focus on the efficiency aspect. The efficiency aspect requires that one should accomplish speedup over the original algorithm. We hope this paper inspires more insight into the topic of recursively interlocking 3D puzzles. May we continue to "do the other things, not because they are easy, but because they are hard". May we continue to recursively unlock a future of possibilities.

## 8. ACKNOWLEDGEMENTS

Many thanks to my supervisor, Professor James Gain, who suggested this interesting yet challenging honours topic along with its many perks. His guidance and support cannot be understated. Thanks to my project partner, Dominic Ngoetjana, for reaching out to me and motivating the selection of this project. Even during tough times, we managed to stay positive and maintain a productive working relationship. Thanks to Sea Monster for allowing us to complete a two-week internship at their offices. This two-week period was undoubtedly our most productive. Thanks to Peng Song, Chi-Wing Fu and Daniel Cohen-Or for writing the research paper this paper is based on. Thanks to every other researcher who has worked on developing our understanding of 3D interlocking puzzles.

## 9. REFERENCES


[1] COFFIN, S. T. 1990. The Puzzling World of Polyhedral Dissections. Oxford University Press.

[2] CUTLER, B. 2007. A Computer Analysis of All 6-Piece Burrs. Available online: http://billcutlerpuzzles.com/docs/CA6PB/index.html. [Accessed 23 April 2019]

[3] IBM RESEARCH, 1997. The burr puzzles site. Available online: http://www.research.ibm.com/BurrPuzzles/. [Accessed 23 April 2019]

[4] KILIAN, M., FLÖERY, S., CHEN, Z., MITRA, N. J., SHEFFER, A., AND POTTMANN, H. 2008. Curved folding. ACM Tran. on Graphics (SIGGRAPH) 27, 3.

[5] LAU, M., OHGAWARA, A., MITANI, J., AND IGARASHI, T. 2011. Converting 3d furniture models to fabricatable parts and connectors. ACM Tran. on Graphics (SIGGRAPH) 30, 4. Article 85.

[6] LO, K.-Y., FU, C.-W., AND LI, H. 2009. 3D Polyomino puzzle. ACM Tran. on Graphics (SIGGRAPH Asia) 28, 5. Article 157.

[7] RÖVER, A. 2011. Burr tools. Available online: http://burrtools.sourceforge.net/. [Accessed 23 April 2019]

[8] SKOURAS, M., COROS, S., GRINSPUN, E., AND THOMASZEWSKI, B. 2015. Interactive surface design with interlocking elements. ACM Transactions on Graphics, vol. 34, issue 6, Article 224.

[9] SLOCUM, J. 2001. Mechanical puzzles their history and their challenge. In Katonah Museum of Art. The art of the puzzle: astounding and confounding.

[10] SONG. P., FU. C., AND COHEN-OR. D. 2012. Recursive Interlocking Puzzles. ACM Transactions on Graphics, Vol. 31, No. 6, Article 128.

[11] SONG, P., FU, Z., LIU, L., AND FU, C. 2015. Printing 3D objects with interlocking parts. Computer Aided Geometric Design, Vol. 35 Issue C, 137-148.

[12] WANG, Z., SONG, P., AND PAULY, M. 2018. DESIA: a general framework for designing interlocking assemblies. ACM Transactions on Graphics, Vol. 37 Issue 6, No. 191.

[13] WYATT, E. 1981. Puzzles in Wood. Woodcraft Supply Corp.

[14] XIN, S.-Q., LAI, C.-F., FU, C.-W., WONG, T.-T., HE, Y., AND COHEN-OR, D. 2011. Making burr puzzles from 3D models. ACM Tran. on Graphics (SIGGRAPH) 30, 4. Article 97.

[15] ZHANG, Y., AND BALKCOM, D. 2016. Interlocking structure assembly with voxels. 2173-2180. 10.1109/IROS.2016.7759341.