UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Final Paper 2019

Title: Recursive interlocking puzzles: rendering and 3D printing

Author: Dominic Ngoetjana

Project Abbreviation: PuzLok

Supervisor(s): James Gain

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	15
Theoretical Analysis	*0*	*25*	0
Experiment Design and Execution	*0*	*20*	0
System Development and Implementation	*0*	*15*	15
Results, Findings and Conclusion	*10*	*20*	15
Aim Formulation and Background Work	*10*	*15*	15
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*	
Total marks		**80**	80

Recursive interlocking puzzles: rendering and 3D printing

Dominic Ngoetjana
Department of Computer Science
University of Cape Town
NGWKGA001

ABSTRACT
Computationally-generated and interlocking 3D puzzles are a fascinating concept—particularly in how there are various methods that attempt to as efficiently, accurately, and speedily as possible generate a variety of 3D puzzles that are not only challenging but incite more creation and excitement for the type of puzzles that are offered.
In this paper, we detail the development process involved in generating computational 3D models and 3D printing them. This process involves merging, shrinking, and rendering mesh models, flowing through the system as either a triangle mesh model or a voxelized grid, depending on the type of functionality invoked. We also observe how merging and shrinking mesh models is necessary for 3D printed models to interconnect without locking together due to the compactness of the printing material.

CCS Concepts
• **Mathematics of computing → Information Theory;** • **Theory of computation → Design and analysis of algorithms → Mathematical optimization →** Continuous optimization; • **Theory of Computation → Theory and algorithms for application domains → Algorithmic game theory and mechanism design →** Algorithmic game theory

Keywords
Interlocking Puzzle(s); 3D Printing; 3D Models; Computationally Generated; Approach;

1. INTRODUCTION
Three–dimensional (3D) puzzles are geometric problems that challenge how we think and the extent of our problem-solving skills. These are also recreational objects and typically involve a certain level of dis/assembly of a 3D shape. The same applies to non-puzzle 3D objects that not only closely emulate puzzles but also real-world objects that may be intended for actual real-world use, such as furniture. Three-dimensional puzzles involve solving complex geometric problems across various axes, which involves some kind of transformation of the object, e.g., translation along one or more axes, rotation of pieces at an angle, etc.

We focus on computationally-generated 3D puzzles and non-puzzles. This refers to objects that are, in the general case, presented as input to a computational program as a 3D triangle mesh (a graphical model built of triangle polygons, to approximate the design of that model) or a voxelized representation (a *blocky* representation of the model). These objects are then processed by a given algorithm that determines how to split the object into pieces that interlock. This object is then either reconstructed using external materials (such as wood), or 3D printed (as in our case); after which, the object can be dis/assembled and is interlocking. Figure 1 shows a representation of a standard 3D puzzle (Kong Ming Lock on the left) and a 3D recursive interlocking puzzle (on the right)—which is a fair representation of our focus in this paper.

Interlocking assemblies have a long history in the design of puzzles, furniture, architecture, and other complex geometric structures. The key defining property of interlocking assemblies is that all component parts are immobilized by their geometric arrangement, preventing the assembly from falling apart [16].

Contribution. Our purpose here is to record the development process that enabled fully graphically-rendered and ultimately 3D printed voxelized representations of input triangle mesh models. The contribution that this makes is in the rendering and (subsequently) the creation of various types of models that are 3D printed and physically dis/assembled as puzzles.

2. RELATED WORK
We have identified several categories that 3D puzzles and non-puzzles fall into. These are categories of algorithms based on the type of approach used to generate objects.

2.1 3D Puzzles
2.1.1 3D Jigsaw Puzzle-esque
In the early stages (and times) of the creation of interlocking puzzles, computational generation was not existent, thus puzzles were merely produced but not yet artificially generated for editing on computer systems. Some earlier examples of these types of puzzles include 3D jigsaw-esque puzzles; these are puzzles composed of pieces that resemble 2D jigsaw puzzles, only they contain larger depths and depending on the approach, the pieces may be planar or non-planar, i.e., flat. One such method is one accomplished with two types of puzzle elements—one that can be folded into a corner configuration through a hinge that unites the planar segments of that element, and another that is generally planar, and (much like the first) does not have a specific design but rather differs per piece. The two element types interlock through a dovetail-type joint [2]. Taking this type of method further by using non-planar elements and different interlocking mechanics, this concept evolves further to emulate more than fundamentally flat structures. An example of this is one that uses a support structure as a basis for a known object, with the non-planar elements clipping onto the support structure to form a resembling outer surface of the

known object, and held together by release pins in the interior of the support structure, such as in [13].

2.1.2 Disc and Extrusion

Similarly to the aforementioned approach, this category of 3D puzzles also has some roots in physically-produced objects composed of likewise pieces. Before computability was applied to this type of interlocking puzzle creation, there existed one in which discs were used to interlock and form larger objects. This concept was later adapted to computation and enhanced with the process of

Figure 1: Kong Ming Lock (left), recursive interlocking puzzle (right) (source: [10])

extrusion. The earlier method adopted for this type of puzzle made use of specifically and purposefully-designed discs. The puzzle is considered complete/solved once the total set of pieces are joined to form a sphere-like structure, as shown in [9]. With the power of computation added, later on, a similar concept was applied using canonical six-piece burr puzzles (or knots) [14]. This burr puzzle acts as a basis for even larger puzzles composed of just of one of these, or a multitude. In the base case, this is achieved when the outer pieces of one knot are extruded using an anisotropic (axial) scaling until they go beyond the 3D model. The next step is to apply a Constructive Solid Geometry intersection between the extruded six pieces and the given 3D model to produce the puzzle "skeleton."

2.1.3 Layers

This approach is one in which a realised object is built up layer by layer till completion. Each layer is built from the bottom up, as it would naturally be to assemble a real-world object with that type of layout of segments. Each layer may consist of several segments joined together. Each segment is composed of several pieces. The first type of method that implements this kind of approach is one that uses polyominoes as basic building blocks for constructing the shapes used in the puzzle assembly. A polyomino is a generalization of a domino constructed by connecting *n* squares edge-to-edge instead of the two squares of an ordinary domino [5]. Another type of method that follows this approach is one that uses a voxelized model, with each voxel split into 8 equal-size blocks. To construct the model, the method first builds long flat chains of the squares (named segments) by connecting joints (male and female connectors on the blocks that permit movement in a certain direction depending on the type of block) of previous squares to new squares. Then the next step is to build layers, where a layer is a set of blocks with the same *z*-coordinate. This process is outlined in [15].

2.1.4 Voxelisation

This approach highlights methods that rely on the voxelization process in order to produce pieces that interlock when assembled. The first method takes in a general voxelized shape as input and iteratively extracts puzzle pieces from it to generate the puzzle. First is the key piece, then the next piece adjacent to that, and so on. The method ensures that every three consecutive pieces interlock, and therefore ensuring that the whole puzzle interlocks without having to exhaustively verify this. The end result is a voxelized interlocking puzzle that can be assembled and disassembled, with one piece as the key. This is detailed in [10]. The second method draws on concepts from the first, but still offers its own procedure, and outputs an object that matches the geometry of the input object, along with ensuring that this object can be 3D printed (by partitioning it into 3D parts that can be assembled) without concerns of its size. This method takes in a 3D watertight (well-defined exterior boundary and closed volume) model as input, converts it into a voxelized grid, followed by a shape connection graph; this is used to generate interlocking parts, after which CSG intersection is used to add back the surface of the mesh. This is process is detailed in [12]. The third method in this approach also draws on the concepts and techniques in the first method. To start off with, the input expected is the final shape of the assembly, from which the component parts are either constructed from scratch as in [10] or explicitly initialized. The computational process for creating an interlocking assembly starts with the full input model, then iteratively splits off successive parts for disassembly. At each iteration, it first identifies a set of suitable blocking relations to be generated between the current assembly and the new part such that the interlocking property is maintained. Then it searches for the part geometry that satisfies these blocking relations.

2.2 3D Non-Puzzles

2.2.1 Furniture

Furniture typically consists of assemblies of elongated and planar parts that are connected together by glue, nails, hinges, screws, or other means that do not encourage disassembly and re-assembly [3], so new approaches were developed that would rid the process of these additional means. The first method presents a computational solution to support the design of a network of interlocking joints that form a globally-interlocking furniture assembly. The key idea is to break the furniture complex into an overlapping set of small groups, where the parts in each group are immobilized by a local key, and adjacent groups are further locked with dependencies. The dependency among the groups saves the effort of exploring the immobilization of every subset of parts in the assembly, thus allowing the intensive interlocking computation to be localized within each small group [3]. The second method presents an interactive tool for designing intrinsic joints. Users draw the visual appearance of the joints on the surface of an input furniture model as groups of two-dimensional regions that must belong to the same part. The method automatically partitions the furniture model into a set of solid 3D parts that conform to the user-specified 2D regions and assemble into the furniture. This is outlined in [4]. The third method presents computational methods as tools to assist the design and construction of reconfigurable assemblies (i.e., consists of a common set of parts that can be assembled into different forms for use in different situations), typically for furniture, as shown in [11].

2.2.2 General

Unlike the previous approach (to a certain extent) there are approaches that were designed to be generally applicable, i.e., generate or guide in creating a variety of different objects. The first

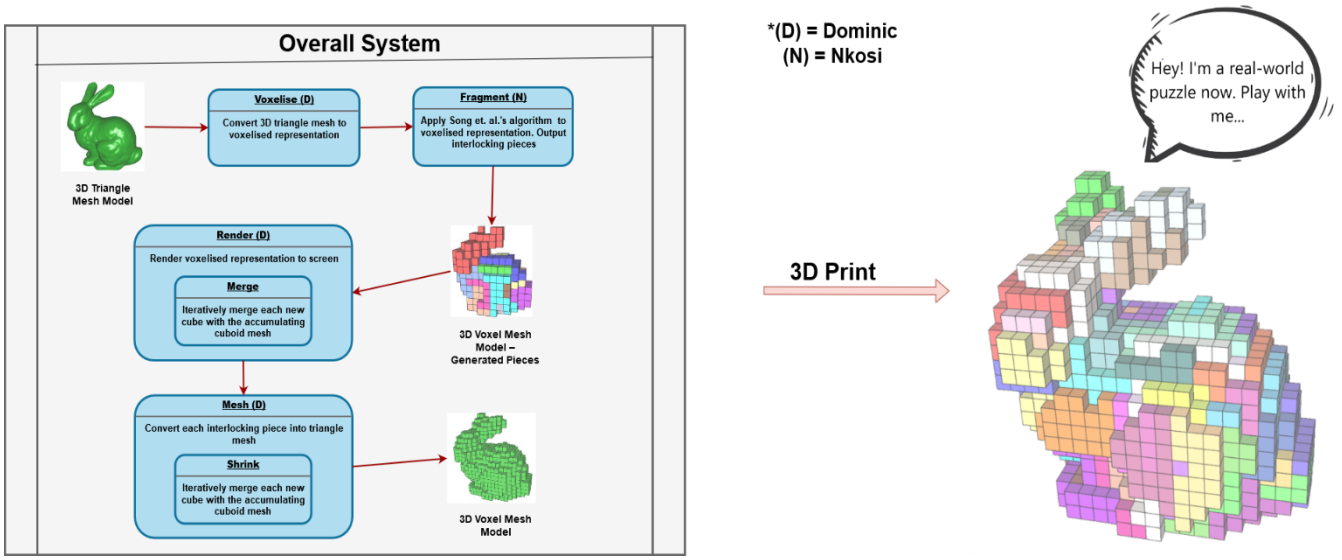

Figure 2: Overall system

method is one that presents a software environment intended to support the fluid interactive design of reconfigurables, featuring tools that identify, visualize, monitor and resolve infeasible configurations [1]. The second method one that allows the computation of aesthetically pleasing structures that are structurally stable, efficiently fabricatable with a 2D wire bending machine, and assemblable without the need of additional connectors [8].

2.2.3 Novelty

In addition to the specialized and purposed approaches stated earlier, there also exist approaches that merely facilitate in creating/designing novelty objects, i.e., cheap or unusual objects. One such method in this approach is one that presents a computational system to design an interlocking structure of a partitioned shell model, which uses only male and female connectors to lock shell pieces in the assembled configuration [7].The second method is one that presents an interactive tool for designing physical surfaces made from flexible interlocking quadrilateral elements of a single size and shape. With the element shape fixed, the design task becomes one of finding a discrete structure—i.e., element connectivity and binary orientations—that leads to the desired geometry. Paralleling principles from conventional modeling software, this approach leverages a library of base shapes that can be instantiated, combined, and extended using two fundamental operations: merging and extrusion [6].

3. SYSTEM FRAMEWORK

3.1 Overall system

The project consists of a contiguous system that works dependently (each part relies on another) and, in the ideal scenario, works autonomously from beginning to end. Refer to Figure 2 for a visual representation of this entire system from 3D triangle mesh model input to 3D printed recursive interlocking puzzle.

The system initially takes in a 3D triangle mesh model as input; the mesh must be watertight, i.e., it must consist of one closed surface, without any holes, and the insides must be clearly defined. This property is a requirement for the next part of the system. The next part is the voxelisation process that is applied to the mesh; This essentially means the system takes in the mesh, then places it in a virtual 3D grid (*blocks* in virtual space), and then iterates through all the *blocks* and for each *block* that *touches* (similar coordinates in 3D space) the mesh, it becomes part of the grid representing the triangle mesh. When the iteration is done, what is identified is a set of *blocks* that approximate the shape of the original triangle mesh model, which is then fed into the next part of the system. Figure 2 shows a *blocky* approximation (on the right) of a bunny model fed in as input (on the left).

The full representative voxel grid is then processed through a fragmentation program (with the algorithm detailed in [10]). This algorithm subdivides the grid into various pieces, which have certain properties, e.g., the pieces must interlock with each other, one piece must immobilize its adjacent pieces, etc. The fragmentation program outputs a series of separate pieces generated from the input grid. Each of these pieces is then fed into the next part of the system (as voxel grids).

Since each input piece is still represented as a voxel grid, the next part of the system generates meshes for each grid point that is enabled, and then merges, shrinks, and renders them. This part of the system is the primary focus of this paper and detailed below.

The last part of the system, after each puzzle piece is ultimately rendered, is to save each piece as a triangle mesh, and 3D print them; the physically-realised pieces can then be dis/assembled into one interlocking puzzle.

3.2 Rendering framework

The framework used was developed by Professor James Gain from the University of Cape Town. The framework's visual layout was created with the Qt5 package, with the entirety of the framework written in the C++ programming language.

The processes to follow in the next sections are implementations that are added onto this existing framework, with some of the functionality required for the focal points of this paper already implemented.

The rendering framework already consists of a graphical user interface for viewing/rendering 3D models (with additional functionality, such as zoom and rotate), along with functionality for voxelising, extracting an isosurface (a 3D surface representation of

points with equal values in a 3D data distribution), smoothing, and deforming. The framework also has the ability to save meshes that are rendered.

We added the functionality to load in specific (.stl) mesh models at runtime from local resources that are then rendered on the screen, along with the functionality of loading in a voxel grid from local resources, which then puts it through the cuboid merging process, and renders it to the screen. The last added functionality applies the cuboid shrinking process to the model on display.

4. MESH RENDERING

Although the rendering framework was already able to render 3D triangle meshes loaded from file, it did not have the functionality of loading in either another triangle mesh or a grid representation and rendering it to the screen. A part of the task was adding this functionality and ensuring that the correct model is displayed.

The framework is modeled such that each rendered model is represented by a 'scene' method within the code, where each scene is configured for the kind of model it is purposed to display, e.g., a voxel scene—for rendering a voxel grid. Such an approach provided the opportunity to simply adopt one of the sample scenes to render a model loaded from file instead. This essentially became a template for other scenes that were added. Rendering a voxel grid required a different strategy, i.e., in addition to using the *template* scene structure, the idea with this type of scene was to use one cube mesh loaded in from file as a basis for constructing a voxel grid representation. The process continues by loading in a new cube mesh for each voxel grid element (i.e., a value set to 1, meaning a voxel exists at those coordinates) and *merging* it with the existing mesh, i.e., initially, translate the new cube relative to the grid element's coordinates, and then change the new cube's vertex indices so they're unique relative to the previous, and finally copy over the new cube's vertices and triangles to the respective data structures of the accumulating mesh (initially one cube, as a basis model) and discard the new mesh. Algorithm 1 provides pseudocode for this process. *Line 15* is where the bulk of the process takes place and a variation of this method will be explored in the next section.

One of the main challenges with creating both the default and voxel scenes was loading in models from file rather than through pop-up dialogs (as done through the aforementioned loading buttons on the graphical user interface), as the framework has a layered directory structure.

5. CUBOID MERGING

As shown in Algorithm 1, merging various cube meshes is one way to obain a fully-connected mesh. The process of merging includes copying vertices and triangles of a particular secondary mesh to a primary one, but goes beyond the level of detail highlighted above. To explain further, some fundamentals are required.

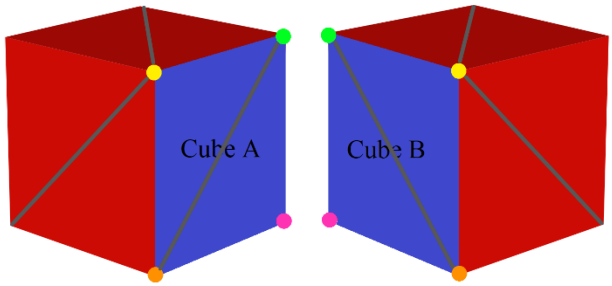

Figure 3: Cube A (left) and Cube B (right). Blue faces will be merged

Fundamentally, and in the general case, triangle meshes are composed of triangles; these are themselves constructed from the indices used to identify the unique vertices of a mesh. In the case of Cube A and Cube B in Figure 3 respectively, there are vertices at the corner of each cube, but for the purpose of our explanation we will only focus on the vertices coloured yellow (Y), green (G), pink (P), and orange (O)—which occur on both cubes. Unless otherwise stated, the right-hand rule will apply in terms of determining the winding of each triangle along the cubes' surfaces, i.e., for the normal vector of a given face of a cube to point to the outside and perpendicular to the face of the cube, the right hand will be wound in a counterclockwise direction along each of the vertices, and the thumb will point in the direction of the normal vector. For instance, one triangle along the vertices of Cube A could be G-Y-O or O-P-G forming the blue face of Cube A, and similarly for Cube B, you could have G-P-O or O-Y-G forming its blue face. Each face of a cube has exactly two triangles, with a total of eight vertices and twelve triangles in one cube.

When Cube A and Cube B merge (join vertices and triangles), the two blue faces join together and are then removed (Cube A on its right and Cube B on its left) along the adjacent vertices, i.e., green to green, yellow to yellow, etc., from Cube A to B. Since Cube A and B share the same vertices along Y, G, P, and O, the duplicate vertices would have to be removed from the collective set; a similar concept applies with the triangles as well. When the cubes merge, the two faces between them must be removed since they longer form part of the exterior of a fully-connected mesh (the resulting cuboid), and so the four triangles (two from each face) must be removed from the collective set. The total number of vertices is reduced from sixteen to twelve.

In the general case, each new cube that is merged into an existing accumulative mesh requires that the two relevant faces are removed. For each new cube that is read in, an entry count property is maintained; the property is used in calculating the indices of the four triangles to be removed. The framework now includes the functionality to read in a grid, iterate through it along the three axes, and for each true value in the grid, a new cube is added, merged, and the two merging cubes' faces are removed. Since the axis iterations are incremental, each new cube (apart from the first) need only consider any pre-existing cubes on its left, bottom, and back for face removal. The merging operation only happens once the entire grid is checked; this helps maintain a consistent data structure when triangles have to be systematically removed with each new cube read in.

A notable challenge with this implementation lies in determining how to pinpoint the specific indices of the triangles to remove,

depending on the new cube's position, its faces, and the faces of adjacent pre-existing cubes, taking into account the adjusting triangle collection at each iteration when a true grid value is found. Algorithm 1 provides pseudocode for this and the rendering process shown above:

Algorithm 1 Cuboid Merging algorithm

```
1: procedure MERGE SCENE (filename)
2:     voxels←readGrid(filename)
3:     accCube←readSTL('cube.stl')
4:     entryCount←0
5:     for {nested loop in z, y, x} do
6:         if voxels[x][y][z]==1 then
7:             if entryCount==0 then
8:                 accCube←setPosition(x,y,z)
9:             else
10:                newCube←readSTL('cube.stl')
11:                newCube←setPosition(x,y,z)
12:                newCube←setNewVertices()
13:                newCube←setNewTriangleIndices()
14:                accCube.vertices←accumulateFrom(newCube.vertices)
15:                accCube.triangles←accumulateFrom(newCube.triangles)
16:                if cube found on left then remLeftFace(newCube),
remRightFace(accCube) end if
17:                if cube found behind then remBackFace(newCube),
remFrontFace(accCube) end if
18:                if cube found below then remBottomFace(newCube),
remTopFace(accCube) end if
19:            end if
20:            entryCount←entryCount+1
21:        end if
22:    end for
23:    accCube←identifyDuplicateVertices()
24:    accCube←reIndexTriangles()
25: end procedure
```

Lines 14-15 transfer the vertices and triangles of the new cube to the accumulating mesh of cuboids after the new cube's vertices are translated relative the position of the valid grid element and its triangles are uniquely re-indexed relative to the exisiting set of accumulating triangles in *Lines 12-13. Lines 16-18* are triggered when a pre-existing cubes are identified on the respective sides of the new cube, which then means the adjoining faces of the new cube and the identified cube must be removed through erasing the triangles representing those faces. *Lines 23-24* activate the last merging process which identifies and removes duplicate vertices (i.e., vertices shared by two or more cubes) and re-indexes the accumulated triangles to compensate for the change in vertex assignment.

6. CUBOID SHRINKING

This next part of the process comes after the merging is complete and the full voxel grid representation exists. Simply put, cuboid shrinking is the process of translating all vertices such that each vertex moves towards the interior of the mesh, at an angle that is determined by the average of the unique inward-facing normals of the triangles connected to that particular vertex, and by a magnitude determined by the user. This process is necessary in order to cater for the 3D printing of pieces. For the physical puzzle pieces to properly interlock without forced dis/assembly due to incompatible sizes among the pieces and an overtight fit when joined, the pieces must all shrink inward. Figure 4 illustrates how the shrinking process would work and why it is a viable approach for this type of operation. Suppose the two pieces in the figure (black and grey, respectively) can join together prior to shrinking (vertically as they are now); it is reasonable to assume that on a computational display this is no cause for concern, except that for 3D printing, the physical material must be taken into consideration; that extra bit of depth is enough to prevent the connectivity of these two pieces. With the cuboid shrinking process applied, each piece would shrink inward and effectively be represented by the black pieces in Figure 4b and Figure 4c. Figure 4a represents two pieces in their default (unshrunken) state, Figure 4b represents the resulting pieces of the shrinking process, and Figure 4c shows how the former is transformed into the latter.

In addition to this shrinking algorithm, two alternatives were considered; the first was affine scaling, which would essentially have all of the vertices translate towards a central point. A caveat with this approach is that the central point would have to be determined, but the main one is that there would be no net change in the spacing of components of the piece. Another approach was to have a smaller sized grid and then map each new cube to that smaller grid. The caveat with this approach is that the overall grid would misalign with other pieces, and would cause chaos if the grid size varied per piece, i.e., each piece would have to be comparatively evaluated with other pieces in order to determine its grid size factor. Our chosen approach offers no such caveats and does not suffer from any known vulnerabilities.

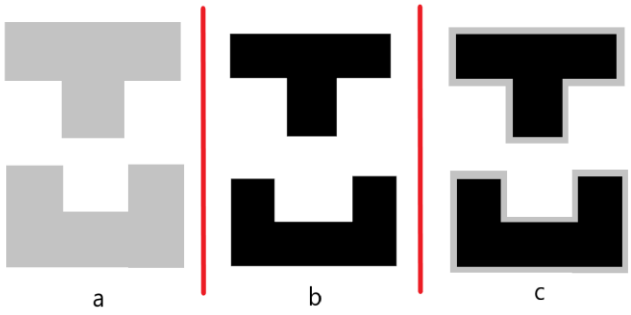

Figure 4: (a) Non-modified shapes, (b) shrunken shapes, (c) shape (b) superimposed onto (a)

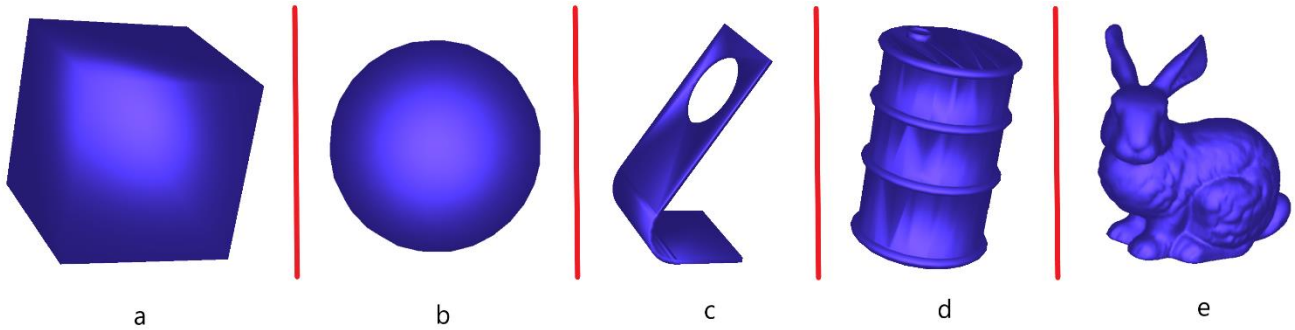

Figure 5: (a) Cube, (b) sphere, (c) hygrometer, (d) barrel, (e) bunny

Algorithm 2 Cuboid Shrinking algorithm

```
1: procedure SHRINK SCENE (filename)
2:     for each vertex in vertexList do
3:         adjTris←findAdjacentTriangles()
4:         for each triangle in adjTris do
5:             normalVec←deriveAndCrossVectors()
6:             normalVec←invert()
7:             normalVec←normalize()
8:             vertextNormals←addFrom(normalVec)
9:         end for
10:        avgNormal←avgVertNormals(vertexNormals)
11:        avgNormal←normalize()
12:        netVectors←addFrom(avgNormal)
13:    end for
14:    shrinkBy←0.5
15:    vertex←nextItem(vertexList)
16:    for each vec in netVectors do
17:        vec←snapVector()
18:        vec←normalize()
19:        vec←specificSnap()
20:        translateBy←multiply(netVectors, shrinkBy)
21:        vertex←add(translateBy)
22:        vertex←nextItem(vertexList)
23:    end for
24: end procedure
```

For each vertex represented by its index in the vertetx list, the set of adjactent triangles is determined; these are triangles with a vertex element that is equal to the vertex list index. This is shown on *Lines 2-3* of Algorithm 2. Then for each adjacent triangle of a vertex, we derive its vectors and cross them. The resulting vector is then inverted (components multiplied by -1), normalized, and then added to a list of vector normals of the vertex. After the vertex normals are calculated, an average normal is calculated from that, and is then normalized. The resulting vector is added to a list of net vectors, which represents the overall movement (-1, 1, or 0) in each component of the vertex. This is done on *Lines 10-12.*

For each vector in the net vectors list, the vector is first snapped, i.e., each component is evaluated against and is, if approximately equal to, set to the value 0, 1, -1. This is so the net vector can be normalized with a set of components in the set {0, 1, -1}, and is then *specific snapped*, i.e., for each component, if the value is positive (greater than 0) then it's set to 1, or -1 if negative, or it remains as 0 if previously so. Each net vector is multiplied by a shrinking factor; this is a value that determines how much each component of a vertex should translate by. The vertex is then translated inward according to the resulting product of the net vector. This process is represented on *Lines 16-23.*

7. RESULTS

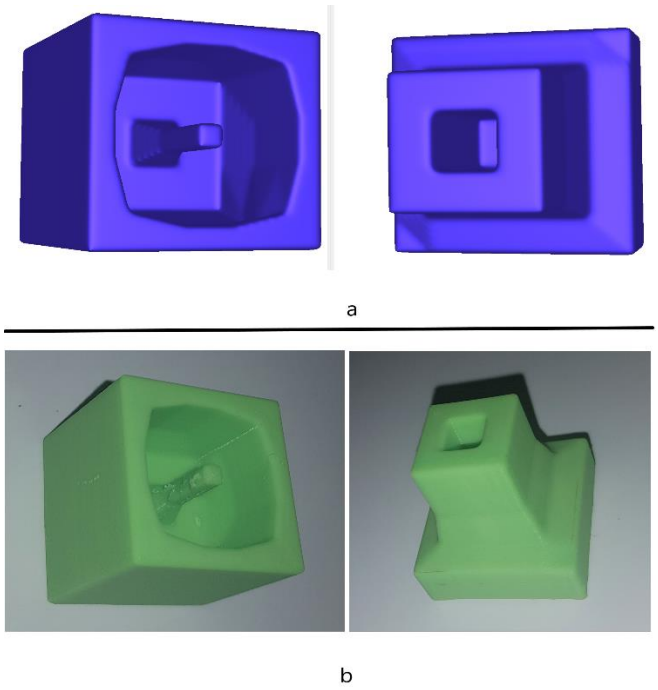

Figure 6: Exterior piece (left), interior piece (right)

As previously mentioned, triangle meshes used as input into the system must be watertight; examples of these are Figures 5a-d. The exception to this is Figure 5e, which was used to confirm that the

process of converting the mesh to its voxelized representation results in an inaccurate representation of the original mesh. Figure 1a (the cube) was predominantly used throughout the development process. A set of merged cubes is used to represent a voxel grid, where a cube is rendered for each valid element of a voxel grid. Cuboids (two or more cubes joined together) are used to illustrate merging (including removal of faces) and the shrinking process (although an individual cube can be used for this purpose).

Figure 6a shows graphical models used to demonstrate all of the core principles highlighted in this paper, i.e., voxelisation, merging, shrinking, and overall rendering. The models shown are the shrunken versions, compensating for the size of 3D printing materials. Figure 6b shows the 3D printed version of Figure 6a. The two pieces interconnect without obstruction, and affirm the viability of generating more complex pieces.

8. CONCLUSIONS

Algorithm design. Each of the algoritms highlighted above are well designed and perform as intended. Algorithm 1 is designed to iteratively merge new cubes to an accumulating mesh composed of previously-rendered cubes. The algorithm takes into account that a new cube will have to merge with at least one other cube in the accumulating mesh and, therefore, adjoining triangles have to be rremoved and duplicate vertices must be removed. Algorithm 2 is designed to iteratively translate each of the vertices of the accumulating mesh. The algorithm covers the whole process from identifying surrounding triangles, to normalizing, snapping, and finally translating the vertex.

Implementation. The implementation of each algorithm is relatively straightforward. The difficulty lies in checking for accuracy in each of the intermediary phases of the implementation of each functionality. The primary method of testing is tracing, which involves standard output error printing of the relevant information required for verifying the accuracy of a particular process. This method is coupled with a paper-written set of tests used to visually confirm the tracing outputs.

9. ACKNOWLEDGMENTS

We would like to thank Prof. James Gain and Dr. Josiah Chavula for their contributions and guidance as supervisor and second reader, respectively.

We'd also like to thank Sea Monster for providing us with working space, project focus, and some guidance.

10. REFERENCES

[1] Akash, G., Alec, J., And Eitan, G. 2016. Computational Design of Reconfigurables. ACM Trans. on Graph. (SIGGRAPH) 35, 4 (2016). Article No. 90.

[2] Benoit, P. And Gareau, D. (2000). *Three-dimensional Puzzle*. US 6,086,067, United States Patent and Trademark Office, 11 July 2000.

[3] Fu, C.W., Song, P., Yan, X., Yang, L.W., Jarayaman, P.K., And Cohen-Or, D. 2015. Computational interlocking furniture assembly. ACM Trans. Graph. 34, 4, Article 91 (July 2015), 11 pages.

[4] Jiaxian, Y., Danny, K., Yotam, G., And Maneesh, A. 2017. Interactive Design and Stability Analysis of Decorative Joinery for Furniture. ACM Trans. on Graph. 36, 2 (2017). Article No. 20.

[5] Lo, K.-Y., Fu, C.-W., And Li, H. 2009. 3D Polyomino puzzle. ACM Tran. on Graphics (SIGGRAPH Asia) 28, 5. Article 157.

[6] Mélina, S., Stelian, C., Eitan, G., And Berhnhard, T. 2015. Interactive Surface Design with Interlocking Elements. ACM Trans. Graph. (SIGGRAPH Asia) 34, 6 (2015). Article No. 224.

[7] Miaojun, Y., Zhili, C., Weiwei, X., And Huamin, W. 2017a. Modeling, Evaluation and Optimization of Interlocking Shell Pieces. Comp. Graph. Forum 36, 7 (2017), 1–13.

[8] Miguel E., Lepoutre M., Bickel B.: Computational design of stable planar-rod structures. *ACM Transactions on Graphics 35*, 4 (July 2016), 86:1–86:11.

[9] Miller, Jr., J. (1998). *Three Dimensional Interlocking Puzzle*. US 5,762,336, United States Patent and Trademark Office, 9 June 1998.

[10] Peng, S., Chi-Wing, F., And Daniel, C. 2012. Recursive Interlocking Puzzles. ACM Trans. on Graph. (SIGGRAPH Asia) 31, 6 (2012). Article No. 128.

[11] Peng, S., Chi-Wing, F., Yueming, J., Hongfei, X., Ligang, L., Pheng-Ann, H., And Daniel, C. 2017. Reconfigurable Interlocking Furniture. ACM Trans. On Graph. (SIGGRAPH Asia) 36, 6 (2017). Article No. 174.

[12] Peng, S., Zhongqi, F., Ligang, L., And Chi-Wing, F. 2015. Printing 3D Objects with Interlocking Parts. Comp. Aided Geom. Des. 35-36 (2015), 137–148.

[13] Simmons, T. (2006). *Three-dimensional Puzzle*. US 7,021,625 B2, United States Patent and Trademark Office, 4 April 2006.

[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] Yinan, Z., And Devin, B. 2016. Interlocking Structure Assembly with Voxels. In IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems. 2173–2180.

[16] Ziqi, W., Peng, S., And Mark, P. "DESIA: A General Framework for Designing Interlocking Assemblies". In: ACM Transactions on Graphics (SIGGRAPH Asia) 37.6 (2018). Article No. 191.