To appear in ACM TOG 34(4).

Computational Design of Twisty Joints and Puzzles

Timothy Sun   Changxi Zheng
Columbia University [*]

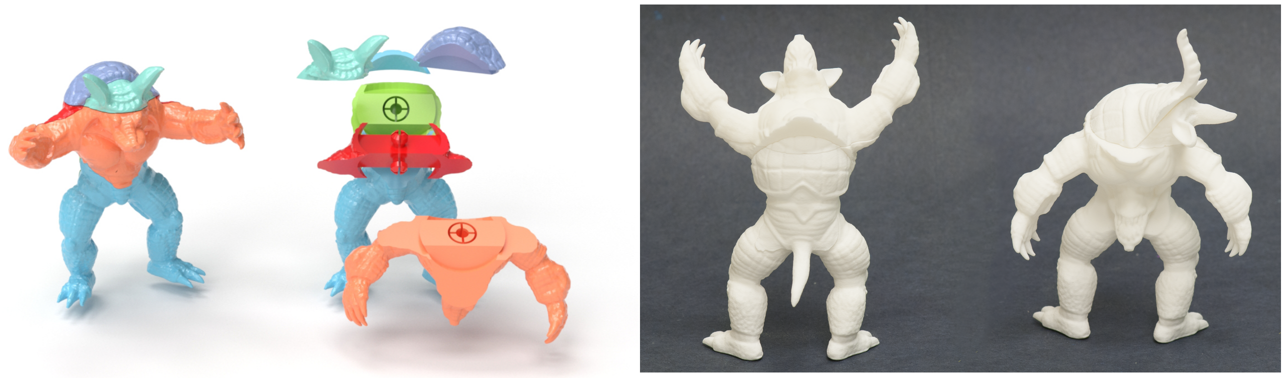

Figure 1: Twisty Armadillo. *(left) A twisty puzzle in the shape of an* Armadillo *whose rotation axes are placed along a triangular prism. (right) The output of our algorithm was fabricated, assembled, and scrambled into contorted poses. The different parts of the model, such as the arms and legs, were deformed so that they do not collide with one other regardless of the configuration of the puzzle.*

Abstract

We present the first computational method that allows ordinary users to create complex twisty joints and puzzles inspired by the Rubik's Cube mechanism. Given a user-supplied 3D model and a small subset of rotation axes, our method automatically adjusts those rotation axes and adds others to construct a "non-blocking" twisty joint in the shape of the 3D model. Our method outputs the shapes of pieces which can be directly 3D printed and assembled into an interlocking puzzle. We develop a group-theoretic approach to representing a wide class of twisty puzzles by establishing a connection between non-blocking twisty joints and the finite subgroups of the rotation group SO(3). The theoretical foundation enables us to build an efficient system for automatically completing the set of rotation axes and fast collision detection between pieces. We also generalize the Rubik's Cube mechanism to a large family of twisty puzzles.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems

Keywords: Computational design, 3D fabrication, twisty puzzles, Rubik's Cube, group theory, interlocking

1 Introduction

Perhaps the most familiar example of a twisty joint is *Rubik's Cube* (Figure 2(a)), a 3D puzzle composed of 26 separate pieces attached to a core with six rotation axes, each of which can be rotated independently. Rubik's Cube and its variants, which are known as *twisty puzzles*, are enormously popular around the world: there are hundreds of "speedcubing" competitions for solving these puzzles every year, and new puzzle designs are being mass-produced for those seeking new challenges (Figure 2(b)). Beyond their recreational popularity, twisty joints that share similar mechanics with Rubik's Cube have found applications in many areas, including mechanical joints for robotics [Ding et al. 2011] and omnidirectional security cameras [Khoudary 2000].

The elegant and ingenious design of Rubik's Cube addresses two seemingly conflicting mechanical requirements. On one hand, it needs to be rotatable around different axes. Every rotation permutes the puzzle's pieces, which need to be aligned so that they can be rotated around other axes. On the other hand, all the pieces are interlocked such that they never fall apart in any configuration. Both goals are realized through precisely aligned rotation axes (Figure 3(a)) and the special shapes of the pieces, each with a hidden internal structure that interlocks with other pieces (Figure 3(b)). Designing new puzzles and joints is a sophisticated task, requiring expert knowledge in twisty puzzle design and often many iterations of trial and error. For example, even with CAD software, a user needs to manually check the internal mechanism for tiny undesired parts that result from misaligned cutting surfaces.

In this paper, we propose a computational approach that enables a non-expert user to easily design new and customized twisty joints and puzzles. We model the structure of a twisty puzzle from an algebraic point of view. The cornerstone of our foundation is a connection between the rotation axis assignment for twisty puzzles and the finite subgroups of the 3D rotation group SO(3). Using this relationship, our method can take an arbitrary set of candidate rotation axes and automatically generate a similar set of axes that yields a workable twisty joint. Our interactive and semi-automatic generation process in essence amounts to aligning finite symmetry groups to any set of axes (§5). The user

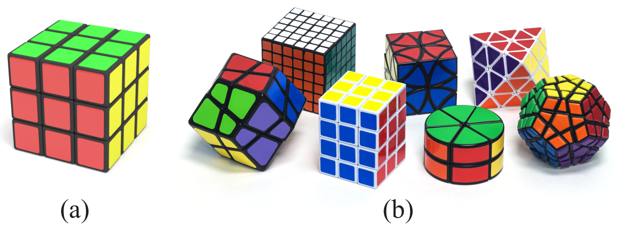

Figure 2: Twisty puzzles. *Rubik's Cube (a) and its variants (b).*

[*]e-mail: {tim, cxz}@cs.columbia.edu

only has to specify a small set of rotation axes and the algorithm automatically adds other cuts and produces fabricable 3D models of the resulting puzzle.

We allow the twisty puzzle to have complex and user-customized shapes (Figure 1), in contrast to the traditional twisty puzzles which are mainly symmetric and convex (Figure 2). To incorporate irregular shapes, an additional challenge arises: different pieces may collide with one another while rotating around an axis (see Figure 14(a)). We detect all these collisions and deform the shape to avoid collisions. With the help of our group-theoretic foundation, we can enumerate all potential collisions between pieces. We also reduce the 3D collision detection to a 2D problem, significantly improving the runtime efficiency (§6).

Finally, we develop an algorithm to automatically generate the geometry of all the pieces, which, once assembled, interlock with one another and form a working puzzle (§7). Furthermore, we exploit the elasticity of the printing material and design a snapping mechanism to ease the assembly process, eliminating the need for other hardware such as screws and springs (§7.3).

Contributions. Our core contribution is the first computational approach for interactively designing customized 3D twisty joints and puzzles, featuring two major technical contributions:

- We develop an algebraic treatment of twisty joints, and model the design of new puzzles using subgroups of the rotation group SO(3) (§4).
- We generalize the Rubik's Cube mechanism to many other sets of rotation axes, and our designs can be 3D printed and assembled into interlocking puzzles (§7).

2 Related Work

3D puzzle design. With the advent of affordable rapid prototyping techniques such as 3D printing, one of the most popular hobbyist uses of rapid prototyping is puzzle design. The creation of such puzzles has been greatly enhanced by computational techniques. The design of polyomino puzzles on the surface of arbitrary objects using quadrilateral meshing was introduced by Lo et al. [2009]. Song et al. [2012] devised an algorithm for generating a large variety of interlocking puzzles made from voxelizations. Our approach resembles that of Xin et al. [2011], who used constructive solid geometry algorithms for generalizing the six-piece burr to arbitrary shapes.

In this paper, we focus on an entirely different class of puzzles. We might describe twisty puzzles as "dynamically interlocking" since pieces can move around while interlocked. Compared to the aforementioned "statically interlocking" puzzles, the requirement that our pieces need to be movable *after* being assembled adds an extra challenge to the design process. Our puzzles rely on twisty joints to produce a large variety of puzzle piece combinations. Zhou et al. [2014] also introduced a type of transformable puzzle which "folds" an arbitrary object into a box using hinge joints between pieces. Rubik's Cubes have been made in other shapes such as character's heads [Scherphuis 2015], but those shapes have been mostly convex or star-shaped.

Transformable objects. One area of fabrication is concerned with creating mechanical assemblies that can move, usually those in the shapes of characters. Ceylan et al. [2013] created mechanical characters whose motions approximate motion-capture sequences. Concurrent work by Coros et al. [Coros et al. 2013; Thomaszewski et al. 2014] used a large set of linkages to generate a wider variety of trajectories for their mechanisms. Bächer et al. [2012] and Calì et al. [2012] fabricated articulated meshes by adding ball joints at key locations. Their models were printed in one piece, and the latter introduced friction to enable static posing. However, all this previous work on transformable objects considers mechanisms where all the pieces are connected by joints. In our setting, we need to consider that some pieces of twisty puzzles "float." Pieces that are not connected to joints are held in place by *interlocking alone*. The transformable objects we study have found applications in mechanical design and robotics [Khoudary 2000; Ding et al. 2011].

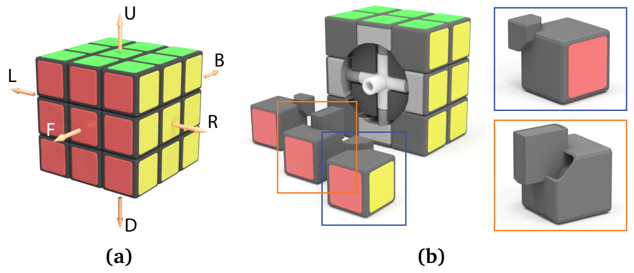

Figure 3: Rubik's Cube. *(a) The rotation axes of Rubik's Cube and (b) a peek at the internal mechanism. The pieces (inset) have little attachments and cavities that cause them to interlock.*

Interactive design. One approach to designing fabricable objects is to make the process semi-automatic: with minimal user interaction, the desired product is generated via an optimization process. Mori et al. [2007] and Skouras et al. [2014] introduced interfaces for designing plush toys and inflatable objects, respectively, by specifying the seams between patches of the surface. Interactive design has even been applied to garment [Umetani et al. 2011] and furniture design [Umetani et al. 2012].

Mesh simplification is a common tool for fabricating an object with as few pieces as possible. Chen et al. [2013] introduce a mesh simplification algorithm which generates large planar pieces suitable for fabrication, and Igarashi et al. [2012] generated beadwork by simplifying a model into a uniform hex-dominant mesh. In the latter work, the user can interactively edit the shape of the simplified mesh. Our work also follows the same philosophy of requiring minimal user interaction: the user specifies a few initial rotation axes and our algorithm generates a fabricable puzzle.

Group theory and Rubik's Cube. Both computer and human solving techniques for Rubik's Cube have benefited from treating Rubik's Cube as an abstract group. Thistlethwaite's algorithm [1981] restricts the state of the puzzle to smaller and smaller subgroups until the puzzle is solved. Rokicki et al. [2010] showed that no position requires more than 20 moves by breaking up the problem into cosets of one of Thistlethwaite's subgroups. Our application of group theory serves a different purpose: generalizing Rubik's Cube to other puzzles. While there exist variants of Rubik's Cube that agree with the theory presented in this paper (such as the ones in Figure 2(b)), our main theoretical result is new to our knowledge.

3 Background and Overview

The goal of this work is to automatically generate a 3D twisty puzzle given a 3D model and some approximate rotation axes specified by the user. A special and simple case of the twisty puz-

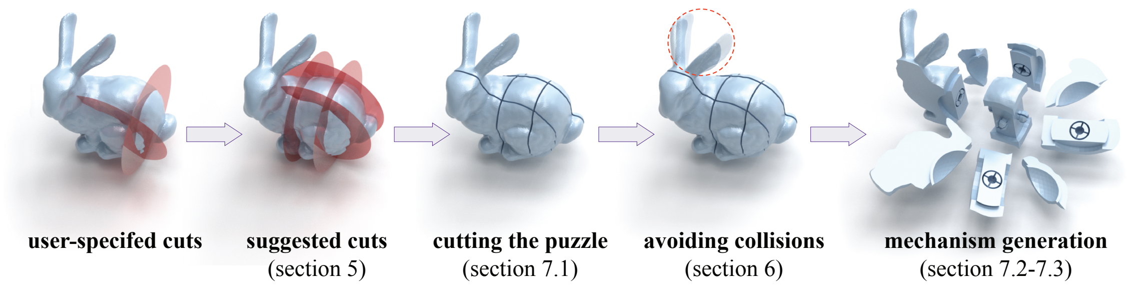

Figure 4: Pipeline. *Starting from simple user input in the form of planes, our algorithm finds a nearby set of non-blocking rotation axes (§5), slices up the puzzle according to those axes (§7.1), deforms the resulting pieces to avoid any external collisions (§6), and generates a fabricable internal mechanism using those pieces (§7.2-7.3).*

zles that our method can generate is Rubik's Cube. Throughout this paper, we constantly refer to it for illustration purposes, thus we first introduce some of its terminology (§3.1) and then give an overview of our pipeline (§3.3).

3.1 Basics of Rubik's Cube

Rubik's Cube has six possible "moves," each corresponding to a rotation axis. These axes are referred to as U(p), D(own), F(ront), B(ack), L(eft), and R(ight) as in Figure 3(a). A move along one rotation axis applies a 90° clockwise rotation to all the pieces on one side of a plane perpendicular to that rotation axis. We say that Rubik's Cube is *non-blocking* because any sequence of moves can be applied to the puzzle without the mechanism getting stuck. An example of a puzzle with blocking is illustrated in Figure 10(b). Each rotation axis has a piece called the *center* which rotates in place, and internally, these centers are connected to a *core*. The pieces with two and three stickers are called *edges* and *corners*, respectively. Figure 3(b) illustrates the different kinds of pieces and the mechanism that holds all the pieces together.

3.2 Jaap's sphere.

While it is natural to visualize Rubik's Cube in its cubic geometry, in order to generalize to other twisty joints, we instead interpret the intrinsic structure of a puzzle using a sphere (Figure 5). This is sometimes called *Jaap's sphere* [Scherphuis 2003]. Jaap's sphere is wholly contained inside the puzzle, and we call the center of the sphere the *core position* of the puzzle. The sphere intersects with some *cutting planes* or *cuts*, each of which separates the sphere into two parts that can be independently rotated. In other words, each cutting plane defines a rotation axis which is perpendicular to the plane and passes through the core position. In the context of our problem, by looking at a sphere, we can more easily examine puzzles produced by arbitrary sets of rotation axes, and we can produce a unified method for designing the internal mechanisms of puzzles produced by those axes (as detailed in §7).

3.3 Pipeline

We incorporate our pipeline (Figure 4) into an interface that takes user-supplied 3D models and rotation axes, and displays the resulting puzzles interactively. Our pipeline generates 3D models which can be directly fabricated and assembled (see the supplemental video). Specifically, out pipeline consists of the following steps:

1. **Positioning Jaap's sphere.** The pipeline is initialized by a 3D model. We aim to make Jaap's sphere as large as possible.

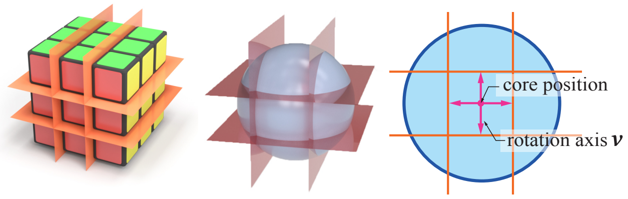

Figure 5: Cuts of a Twisty Puzzle. *Some cuts of a Rubik's Cube (left) and its Jaap's sphere (middle), and the 2D projected view (right). Each cut can be described by a single perpendicular vector inside the sphere.*

Since all cutting planes have to pass through Jaap's sphere, a larger sphere means a wider variety of cutting planes. The algorithm computes the optimal core position $\boldsymbol{c}$ by solving the optimization problem

$$\boldsymbol{c} = \underset{\boldsymbol{p} \in \text{int}(S)}{\arg\max} \left(\min_{\boldsymbol{q} \in S} \|\boldsymbol{p} - \boldsymbol{q}\| \right)$$

where S denotes the surface of the 3D model and int(S) denotes its interior region. A solution to this optimization procedure is illustrated in Figure 6. Our implementation performs simulated annealing [Kirkpatrick et al. 1983] starting from the center of mass of the model. We then set the radius of Jaap's sphere to be $\min_{\boldsymbol{q} \in S} \|\boldsymbol{c} - \boldsymbol{q}\|$.

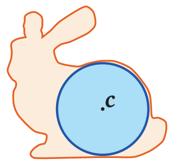

Figure 6: Fitting Jaap's sphere *inside the Stanford bunny with the optimal core position* $\boldsymbol{c}$. *We make the sphere as large as possible inside the object.*

2. **Axis auto-completion (§5).** Using the graphical interface, the user specifies cuts that pass through the 3D model and Jaap's sphere. These cuts define a subset of rotation axes, but they may generate a puzzle that blocks. Our algorithm perturbs the user-specified cuts and adds other rotation axes to produce a non-blocking puzzle. The good sets of cuts are derived using results on rotation groups (§4).
3. **Resolving collisions (§6).** The suggested cuts then separate the provided 3D model into individual pieces. Inside Jaap's sphere, these pieces can freely rotate around the axes. However, for complex shapes, the resulting pieces may collide with one another in certain positions of the puzzle. We enumerate all possible configurations of pairs of pieces by defining a group structure on the pieces. A 2D collision detection algorithm is introduced for this problem. Once collisions are detected, they are resolved using standard Laplacian mesh deformation algorithms.

4. **Internal mechanism (§7).** Finally, for every piece of the puzzle, we create an interlocking internal mechanism which rotates according to the rotation axes. The mechanism is a generalization of the Rubik's Cube mechanism; its generation uses only constructive solid geometry operations on simple geometric primitives.

4 Theory

In this section, we operate on Jaap's sphere, ignoring for now the 3D model surrounding the sphere. We will consider the shapes of 3D models in §6 and §7. Without loss of generality, we assume that the sphere has unit radius and is centered at the origin. The goal of this section is to develop a relationship between valid sets of rotation axes and an algebraic result on rotations in $\mathbb{R}^3$ and define an algebraic group structure on the positions of a puzzle. We refer to a Rubik's Cube throughout for illustrating the concepts, even though we consider a much wider class of twisty joints and puzzles.

4.1 Abstract Puzzle Representation

A *rotation axis* $\boldsymbol{r} = (\boldsymbol{v}, n)$ is specified by a nonzero vector $\boldsymbol{v}$ whose norm is strictly less than 1 and an integer n. The direction of $\boldsymbol{v}$ indicates the orientation of the rotation axis, and the length of $\boldsymbol{v}$ indicates where the sphere is cut (Figure 7). The cutting plane is the plane that passes through $\boldsymbol{v}$ and is perpendicular to $\boldsymbol{v}$, i.e. the locus of points $(\boldsymbol{p} - \boldsymbol{v}) \cdot \boldsymbol{v} = 0$. Thus, in our presentation, we refer to $\boldsymbol{v}$ as a vector and as a plane interchangeably. The rotation axis can be turned by angles of $2\pi/n$. For example, the axes of Rubik's Cube have $n = 4$. Furthermore, we call the two halfspaces

$$\boldsymbol{r}^0 = \{\boldsymbol{p} \mid (\boldsymbol{p} - \boldsymbol{v}) \cdot \boldsymbol{v} < 0\} \text{ and } \boldsymbol{r}^1 = \{\boldsymbol{p} \mid (\boldsymbol{p} - \boldsymbol{v}) \cdot \boldsymbol{v} > 0\}$$

the *near* and *far* sides or halfspaces of the rotation axis, respectively.

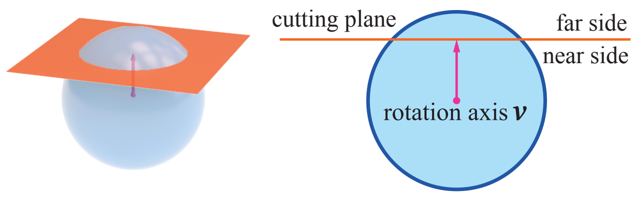

Figure 7: A rotation axis and its cutting plane *on a Jaap's sphere. The cutting plane forms two halfspaces that we use to describe pieces.*

A set of rotation axes partitions the sphere into *pieces*. A piece $p = \{\boldsymbol{r}_1^{s_1}, \ldots, \boldsymbol{r}_k^{s_k}\}$ is specified by the halfspaces which form it, where $s_i, i = 1, \ldots, k$ is either 0 or 1, depending on which side of the axis $\boldsymbol{r}_i$ the piece is on. For instance, a corner of a Rubik's Cube is specified by three far halfspaces, and an edge is composed of two far halfspaces and two near halfspaces (Figure 8). Turning a rotation axis $\boldsymbol{r} = (\boldsymbol{v}, n)$ by its angle $2\pi/n$ rotates each of the pieces on the far halfspace by $2\pi/n$ clockwise about the axis $\boldsymbol{v}$. The (unique) piece containing the origin is called the *core*, which is often concealed by the other pieces. The pieces that lie on the far side of exactly one rotation axis are called *centers*. Note that every center touches the core. For the purpose of designing the internal mechanism in §7, we need a core and a center for each rotation axis: having the centers affixed to the core is the basis of the interlocking design. By requiring that the length of the rotation axis is positive (i.e., $\|\boldsymbol{v}\|_2 \neq 0$), we ensure that no plane cuts through the origin, guaranteeing the existence of the core piece. To make sure that each rotation axis $\boldsymbol{r}_i$ has an associated center piece, for every other rotation axis $\boldsymbol{r}_j$, $i \neq j$, $\boldsymbol{v}_i$ is on the near halfspace of $\boldsymbol{r}_j$, i.e. $(\boldsymbol{v}_i - \boldsymbol{v}_j) \cdot \boldsymbol{v}_j < 0$. On Rubik's Cube, the rotation axis passes through the center piece, but on the right of Figure 9, the "center" piece of the green rotation axis is actually on the far side of two rotation axes.

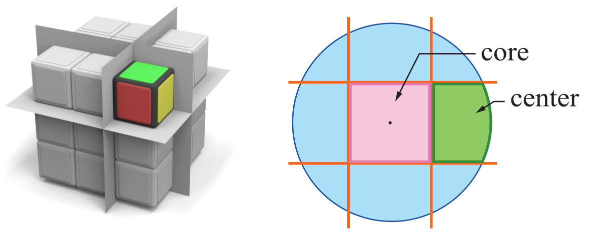

Figure 8: Piece Types. *(left) A corner piece of Rubik's Cube is specified by three cuts (or equivalently, three rotation axes). (right) A 2D illustration of the core and center pieces.*

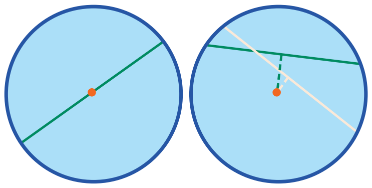

Figure 9: *Because we need core and center pieces for the mechanism design, we do not allow these types of rotation axis configurations.*

4.2 Non-Blocking Puzzles

One crucial property of twisty joints and puzzles is that they are non-blocking. An illustrative example of how to make a blocking puzzle non-blocking is the "two-generator" Rubik's Cube, where only two adjacent faces can turn. If we only cut the cube using the two rotation axes U and R (Figure 3(a) and Figure 10(a)), it is clear that this puzzle will lock—after turning R clockwise (Figure 10(b)), the U axis is blocked. To remove this blocking, we can cut along the U axis's plane again (Figure 10(c)). Applying this process repeatedly to our example will produce a non-blocking puzzle.

However, for most sets of rotation axes, this process never terminates and we will end up with an infinite number of pieces. Therefore, a physically meaningful puzzle needs to have carefully arranged rotation axes such that the number of resulting pieces is finite. Our characterization of sets of rotation axes that are non-blocking is based on finite subgroups of SO(3), the group of rotations of the sphere. In particular, for each element of one of these subgroups, we make a cut perpendicular to that element's axis of rotation. As a result, the number of resulting pieces is guaranteed to be finite. The justification of this connection between rotation axes and finite subgroups of SO(3) is detailed in Appendix A and is, to our knowledge, new. In fact, there is a succinct characterization of the finite subgroups of SO(3):

> **Theorem.** [Thurston and Levy 1997] Any finite subgroup of SO(3) is isomorphic to a cyclic group, dihedral group, or a rotational symmetry group of a Platonic solid.

We visualize these families of rotation axes using polyhedra (see Figure 11)—when we refer to a rotation axis corresponding to a facet of a polyhedron, we mean the rotation axis that is parallel to its centroid. The sets of rotation axes correspond to

1. the square faces and/or the n-gonal faces of the n-prism (cyclic, dihedral).
2. the faces, edges, and/or vertices of the Platonic solids (tetrahedral, octahedral, icosahedral).

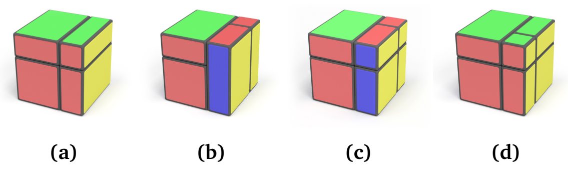

Figure 10: "Two-generator" Rubik's Cube. *Cutting a cube using only two rotation axes (a) will block after just one turn (b). Blocking can be resolved by cutting the puzzle further (c-d).*

As an example, Rubik's Cube is an instance of the octahedral class. Subsets of these (like the two-generator example) can be achieved by "fusing" centers to the core to block their motions. Because of the group-theoretic derivation of these sets of rotation axes, we will refer to them simply as the *finite rotation groups*.

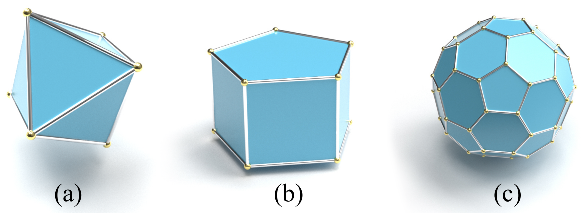

Figure 11: Examples of polyhedra *which have finite symmetry groups. These have octahedral, dihedral, and icosahedral symmetries, respectively, and we place rotation axes along the fixed points of a rotational symmetry.*

4.3 The Puzzle Group

A *permutation* $\tau : X \rightarrow X$ is a one-to-one correspondence of the elements of a set X. In the context of our problem, applying moves to Rubik's Cube shuffles pieces around (Figure 12(a)), so these moves can be expressed as permutations. However, the permutation is not on the pieces, since pieces can have different *orientations*. For example, a corner piece can be "twisted" in the correct position. We therefore define the *flag* $(p, \boldsymbol{r})$, which is a pair of a piece p and one of its rotation axes $\boldsymbol{r}$. The piece's flags encode the orientation. For example, each corner and edge of a Rubik's Cube has three and four flags, respectively (Figure 12(b)). The flags of a core piece and one of each center's flags (i.e. the flag $(p, \boldsymbol{r})$ where $\boldsymbol{r}^1 \in p$) stay fixed under the action of any rotation axis, so we ignore them.

Rotating the puzzle along a rotation axis applies a permutation on the flags. We refer the reader to Appendix B for details on how to compute these permutations. Arbitrary compositions of these permutations form the *puzzle group* of the puzzle, which describes all the possible states of the puzzle.

Remark. The puzzle group defined here is general enough for describing other sets of rotation axes, such as the one shown in Figure 13. However, in our user interface and examples, we only consider the rotation axes of the finite rotation groups (recall Figure 11). Furthermore, we do not have a general classification of all possible sets of rotation axes that have a well-defined puzzle group.

5 Auto-Completion of Rotation Axes

As developed in §4.2, a non-blocking twisty puzzle is constructed by choosing rotation axes derived from a finite rotation group. To ease the user in specifying valid sets of rotation axes on an

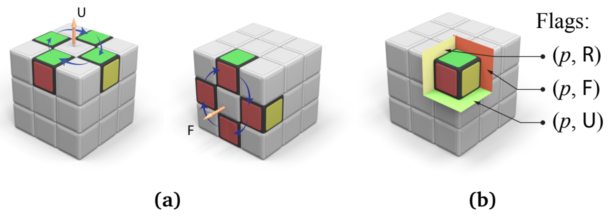

Figure 12: Flags. *(a) Each rotation axis applies a permutation on the pieces. (b) The flags of a corner of Rubik's Cube. Flags can intuitively be thought of as different stickers of pieces.*

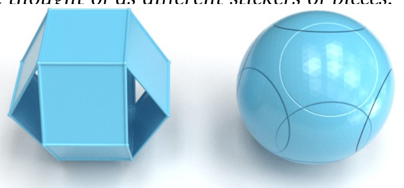

Figure 13: *A set of axes that does not come from a finite rotation group, but still can be described by a puzzle group. The rotation axes are the faces of this "polyhedron" which consists of two hexagonal cycles of square faces. The centers do apply a permutation to the two adjacent lune-shaped pieces.*

arbitrary shape, we provide an interface to allow the user to sketch a few planes that cut through the 3D model and the Jaap's sphere inside (see the supplemental video). Every user-specified plane defines a rotation axis. Our algorithm then automatically adjusts these cuts and suggests other cuts to form a valid puzzle.

In general, user-specified cuts will not be perfectly aligned with any of the Platonic solids or n-prisms. Instead, we take each of the finite subgroups and rotate it to fit the user-specified rotation axes as closely as possible. There is an infinite number of dihedral groups D_q, so in practice we set a threshold by only considering $q \leq 5$. Suppose the user specifies n rotation axes $\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_n$ and we have a subgroup with m rotation axes $\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_m$, where $m \geq n$. For each pair of rotation axes $\boldsymbol{r}_i$ and $\boldsymbol{r}_j$, we compute the angle θ_{ij}. Meanwhile, for all ordered subsets of size n of $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m$, we similarly compute angles θ'_{ij} and choose the set which minimizes

$$\sum_{1 \leq i < j \leq n} \|\theta'_{ij} - \theta_{ij}\|^2.$$

Here we can reduce the number of subsets by accounting for rotational symmetries. Let $\boldsymbol{s}'_1, \ldots, \boldsymbol{s}'_n$ denote the optimal ordered subset of rotation axes drawn from a particular subgroup with the axes $\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_m$. We wish to rotate these axes to align with user-specified axes as closely as possible. To this end, we find a rotation matrix $\boldsymbol{R}$ that minimizes

$$\sum_{i=1}^{n} \|\boldsymbol{r}_i - \boldsymbol{R}\boldsymbol{s}'_i\|^2.$$

In graphics, this is known as shape matching [Müller et al. 2005; Rivers and James 2007] (and in other fields it is called the orthogonal Procrustes problem [Gower and Dijksterhuis 2004] and Wahba's problem [Wahba 1965]). The standard solution, as seen in [Müller et al. 2005], is to consider the following matrix and its polar decomposition:

$$\boldsymbol{B} = \sum_{i=1}^{n} \boldsymbol{r}_i \boldsymbol{s}_i'^T = \boldsymbol{R}\boldsymbol{S},$$

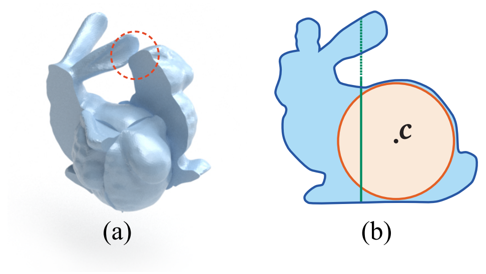

Figure 14: Collisions of Puzzle Pieces. *Even though a puzzle is non-blocking, pieces may collide with one another (a). These collisions can occur when at least one of the pieces has overhang (b). If there is no overhang, then each piece is confined to an intersection of halfspaces, and these regions are all distinct from one another.*

where $\boldsymbol{S}$ is a positive-semidefinite matrix and $\boldsymbol{R}$ is the desired rotation matrix. Both matrices can be computed using the singular value decomposition [Golub and Van Loan 2012]. Finally, we apply $\boldsymbol{R}$ to all the rotation axes $\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_m$ to obtain the rotation axes (i.e., $\boldsymbol{R}\boldsymbol{s}_1, \ldots, \boldsymbol{R}\boldsymbol{s}_m$) for puzzle generation. We scale the vectors so that their lengths equal the average length of the user-specified rotation axes.

6 Avoiding Collisions

After completing the rotation axes based on user-specified cuts, we use the associated planes to cut the given 3D model into individual pieces (details in §7.1). Unfortunately, since we are cutting arbitrary geometries, pieces of the puzzle may collide with one another after applying several turns (Figure 14(a)). Such collisions can be considered as another type of blocking. In this section, we first present an approach to detect all possible collisions without using any temporal collision detection techniques such as a direct simulation of puzzle rotations (§6.1). The 3D collision detection problems are reduced into 2D problems, drastically increasing the efficiency of our algorithm. These two simplifications allow us to check only a finite set of configurations, and symmetries reduce the number even further. We then resolve the collisions using simple Laplacian-based mesh deformations (§6.2).

6.1 Collision Detection

A piece cut out from a 3D model is said to have *overhang* if part of the piece intersects one of its rotation axes, as in Figure 14(b). We observe that if two pieces are free of overhang, then it is impossible for them to ever collide, while the existence of overhang may not necessarily lead to collisions. It suffices to check only the pairs where at least one of the pieces has overhang. One simple approach is to deform every piece until there is no overhang regardless of the actual collisions. While this can guarantee there are no collisions, we often only need moderate deformations or none at all.

For each pair of pieces, we check if there exists some configuration of the puzzle, including during moves, where they collide. One case does not need to be checked because it can always be avoided: if two pieces collide only while applying two moves simultaneously (for example, manipulating two parallel rotation axes), we can simply apply the two moves sequentially. As a result, we only need to check cases where one piece is being rotated while the other piece stays fixed.

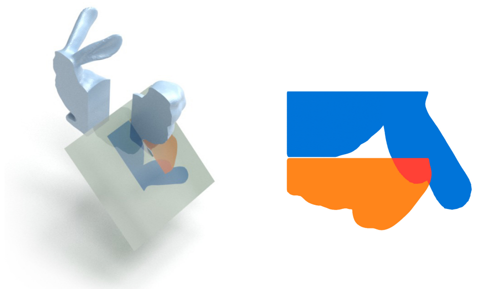

Figure 15: Collision Detection in 2D. *We reduce the collision detection problem from 3D to 2D by projecting each mesh onto a halfplane. The projection removes the angular component from the meshes, which is equivalent to rotating the halfplane about the rotation axis and finding the intersection with the two meshes (see the supplemental video).*

Orbits of flags. We need to enumerate all the possible locations and orientations these pieces can be in. In between moves, there are a finite number of positions and orientations a piece can be in. Since the orientation of a piece is encoded by one of its flags, these positions constitute the *orbit* of the flag. For example, the orbit of a corner's flag on Rubik's Cube has 24 flags: three flags for each orientation of the eight corners.

To compute the orbit of each flag, we create a graph where each flag is a vertex. There is a directed edge from f to f' if there is a rotation axis $\boldsymbol{r}$ that brings f to f' (this information can been determined from the puzzle group). On the directed edge, we store a rotation corresponding to $\boldsymbol{r}$. The connected component containing a flag f is its orbit, and it can be found by a depth-first search. The transformation from f to another flag f' in its orbit can be computed by tracing the path of edges from f to f' in the depth-first search tree and composing the resulting sequence of rotations. By abuse of notation, we write the orbit of a flag of piece p_i as $O(p_i)$, because the orbits of different flags of a single piece are all isomorphic to one another.

Given two pieces p_1 and p_2, we compute the orbits $O(p_1)$ and $O(p_2)$ of one of their flags. For each pair of flags $(p_i, \boldsymbol{r}_i) \in O(p_1)$ and $(p_j, \boldsymbol{r}_j) \in O(P_2)$ in the two orbits, we first move p_1 and p_2 to p_i and p_j, respectively, using the previously computed rotations. We then test for collisions by rotating each rotation axis that revolves exactly one of p_i and p_j. We reduce the total number of pairs of flags to check by considering rotational and reflectional symmetries which can be determined from the finite rotation group the axes came from.

Collision detection in 2D. Because the only motions are pieces rotating around a single axis, we can reduce this collision detection into a 2D problem. Let $\boldsymbol{z}$ denote the unit vector in the direction of the rotation axis we are rotating (Figure 15). We express the coordinates of the pieces' mesh vertices in cylindrical coordinates $(\boldsymbol{z}, r, \theta)$. We are not interested in the angular component, because we need to detect collisions in any state of rotation. Thus, we discard it to get a two-dimensional collision detection problem in $(\boldsymbol{z}, r)$. In practice, this method can be implemented by first constructing a tree of bounding spheres around the original mesh in 3D. Since projecting out the angular component in any direction $\boldsymbol{z}$ yields a bounding circle in 2D, the hierarchy only needs to be constructed once per piece. Finally, we note that in this problem, we care about capturing all the collisions, but not about the exact positions at which these collisions occur, because a vertex may collide at any revolved state around the rotation axis. Therefore, at the end of this collision detection step, we only store the indices of vertices that are in collision, and the

Algorithm 1 Internal Mechanism

```
Require: Mesh $M$, piece $p = \{\boldsymbol{r}_1^{s_1}, \ldots, \boldsymbol{r}_k^{s_k}\}$
 1: procedure PIECE-MAKER($M, p$)
 2:     Cut $M$ along each $\boldsymbol{r}_i$                              ▷ §7.1
 3:     Create inner sphere $S$.
 4:     for all $\boldsymbol{r}_i^{s_i} \in p$ such that $s_i = 1$ do         ▷ §7.2
 5:         Create new halfspace $(\boldsymbol{r}'_i)^1 = (\boldsymbol{v}'_i, n_i)^1$
 6:         Intersect $S := S \cap (\boldsymbol{r}_i^1)'$.
 7:     end for
 8:     Union $M := M \cup S$.
 9:     for all $\boldsymbol{r}_i^{s_i} \in p$ such that $s_i = 0$ do
10:         Create cylinder $C_i$.
11:         Subtract $M := M \setminus C_i$.
12:     end for
13:     if $f(p) = 1$ then                                              ▷ §7.3
14:         Make shaft $S_i$ along $\boldsymbol{r}_i$, where $\boldsymbol{r}_i^1 \in p$.
15:         Subtract $M := M \setminus S_i$.
16:     end if
17:     if $f(p) = 0$ then
18:         for all $\boldsymbol{r}_i^{s_i} \in p$ do
19:             Create appendage $A_i$.
20:             Union $M := M \cup A_i$.
21:         end for
22:     end if
23: end procedure
```

corresponding rotation axes that introduce the collisions. In the worst case, every piece has overhang, so every possible pair of pieces needs to be considered, and for each pair, every pair of elements from their respective orbits must be checked.

6.2 Deformation

After detecting vertices in each mesh which collide with the meshes of other pieces, we deform the mesh to resolve collisions. For each vertex, we wish to move it in the direction of the rotation axis $\boldsymbol{z}$, because deforming a colliding vertex in a direction on the plane of rotation would not help resolve the collisions—since we revolve the piece entirely around the axis, the collision would still occur. Our deformation algorithm is an iterative process. At each iteration, for every vertex $\boldsymbol{v}$ in collision, our algorithm aims to decrease the angle between $\boldsymbol{v}$ and $\boldsymbol{z}$ by a fixed small amount θ. That is, we slightly deform $\boldsymbol{v}$ toward $\boldsymbol{z}$. In practice, we use $\theta = 5°$ at each iteration. The collision detection and deformation steps are repeated until all collisions are resolved.

In addition, we deform the rest of the mesh smoothly while preserving the adjusted positions of the vertices in collision. Let V and V' denote the set of all vertices of a puzzle piece's mesh and the subset of vertices in collision, respectively. Let $\boldsymbol{v}$ be the undeformed vertex position of the mesh. We minimize the following energy to find a deformed mesh described by vertex positions $\boldsymbol{v}^*$:

$$E(V) = \gamma \sum_{\boldsymbol{v} \in V} \|\Delta \boldsymbol{v}^* - \Delta \boldsymbol{v}\|^2 + (1-\gamma) \sum_{\boldsymbol{v} \in V'} \|\boldsymbol{v}^* - \boldsymbol{v}'\|^2, \qquad (1)$$

where Δ is the discrete Laplace-Beltrami operator [Meyer et al. 2003] and γ is a scalar weight balancing both terms. This is a standard Laplacian mesh editing scheme [Sorkine et al. 2004], for which the underlying motivation is to use the Laplacian to preserve the local "shape" of the object in the deformation. The first term of (1) is to preserve the shape of the original mesh, while the second term is to match the desired deformation. If we let $\boldsymbol{d}$ be the vector stacking all the vertex displacement (i.e., $\boldsymbol{d}_i = \boldsymbol{v}_i^* - \boldsymbol{v}_i$ for all $\boldsymbol{v}_i \in V$), then taking the gradient of the energy and setting it to 0 yields the linear system

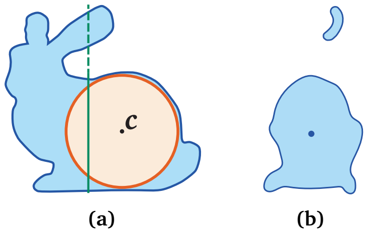

Figure 16: Cutting a Puzzle. *The intersection of a plane and the surface of the 3D model (a) can be disconnected, as seen in a side view of the intersection (b). We only cut the model along the essential polygon, which contains the origin of the plane.*

$$\gamma \Delta \boldsymbol{d} = -(1-\gamma)\boldsymbol{d}.$$

Solving this equation results in a deformed mesh with the vertex positions $\boldsymbol{v}_i^* = \boldsymbol{v}_i + \boldsymbol{d}_i$. We then move on to the next iteration step until the collisions are fully resolved, or we reach a maximum number of iterations. We do not prove convergence for this algorithm in theory, although all our examples ended up being collision-free. We believe checking for convergence is expensive as it means to repeatedly check all possible permutations.

7 Design of Inner Mechanism

The last step of our pipeline is to generate the internal mechanism for every piece of the puzzle. The geometry it automatically creates can be directly 3D fabricated. After assembling the fabricated pieces together, they interlock with one another while rotating. Our mechanism generation is general for all puzzles that we defined in §4, including those not derived from finite subgroups of SO(3), and involves only constructive solid geometry (CSG) operations on the pieces and primitives such as spheres and cylinders. The full algorithm is outlined in Algorithm 1. To our knowledge, there has not been an automatic approach for generating the internal mechanism.

Our method starts by cutting the provided 3D model into pieces using cutting planes defined by rotation axes (§7.1). It then generates the internal interlocking mechanism for each piece (§7.2) as well as the core (§7.3). As an extension, our method can also incorporate curved cutting surfaces (§7.4), thus producing more complex and interesting twisty puzzles.

7.1 Cutting the Puzzle

We assume that the input 3D model is a well-tessellated (e.g. isotropically remeshed) topological sphere. Let $\boldsymbol{r} = (\boldsymbol{v}, n)$ be a rotation axis, as introduced in §4.1. When there is no ambiguity, we also use it to refer to the cutting plane related to the rotation axis. The *origin* of the cutting plane is the point closest to the center of Jaap's sphere. Unfortunately, we can not simply cut the 3D model using planes, since some pieces might have overhang (Figure 14(b)). In general, the intersection of a plane and the surface mesh results in multiple polygons. We wish to only cut along what we call the *essential polygon*, the one whose interior contains the origin of the plane because its interior intersects Jaap's sphere (Figure 16). To robustly cut the puzzle along the plane, we take a ray on the plane starting at its origin and intersect it with the 3D model. We find the first intersecting face and subdivide it into two faces and continue onto the neighboring faces that also intersect the cutting plane. We continue along the mesh until we return to the original face, tracing out the essential polygon on the 3D model. The mesh is split into two new meshes along the newly-created edges. We fill in the resulting holes using a standard constrained Delaunay triangulation algorithm [Shewchuk 1996].

The situation becomes more complicated when we slice the model with multiple planes to make individual pieces. A piece $p = \{\boldsymbol{r}_1^{s_1}, \ldots, \boldsymbol{r}_k^{s_k}\}$ is formed by applying the tracing algorithm sequentially on each of its halfspaces. We can interpret this process as forming a sequence of intermediate pieces $p_j = \{\boldsymbol{r}_1^{s_1}, \ldots, \boldsymbol{r}_j^{s_j}\}$, for $j = 0, \ldots, k$. The ray intersection step produces a random point on the essential polygon of $\boldsymbol{r}_j$, but that point might not be on the intermediate piece p_{j-1}. Instead of choosing a random point, we first find a point $\boldsymbol{v}_j$ on the essential polygon of each rotation axis that is on p_{j-1}, as shown in Figure 17(b). Once these points are computed, we can form each intermediate piece p_j by the tracing algorithm on p_{j-1}, starting at the precomputed point $\boldsymbol{v}_j$.

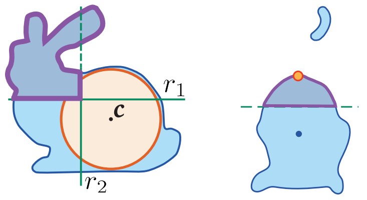

Figure 17: *If we wish to make a piece $p = \{\boldsymbol{r}_1^1, \boldsymbol{r}_2^1\}$ (left), we first compute a point on the essential polygon of $\boldsymbol{r}_2$ that is part of the piece $p' = \{\boldsymbol{r}_1^1\}$ (right). After cutting by $\boldsymbol{r}_1$ using the tracing algorithm, we can trace out what is left of the essential polygon of $\boldsymbol{r}_2$ by starting at the computed point.*

7.2 Interlocking Internals

We now generate the internal interlocking mechanism for each piece. We examine the internals of our BUNNY example (Figure 18(a)). The BUNNY model has four of the six rotation axes of Rubik's Cube, so the mechanisms are similar. The center pieces are attached to the core so that they can only rotate in place. Thus, by construction these piece can not come apart (see §7.3). The remaining pieces have a spherical extension that allow them to latch onto the center pieces.

Consider the case of a single rotation axis $\boldsymbol{r} = (\boldsymbol{v}, n)$. The previous step cuts the puzzle into two parts by slicing along the plane $\boldsymbol{r}$. The two pieces can be made interlocking by attaching part of a smaller sphere to the piece lying on the far halfspace of $\boldsymbol{v}$ and subtracting the same attachment from the other piece (Figure 18(b)). In practice, we carve out a cylinder instead of a sphere so that the puzzle is easier to assemble. Note that we still need to connect the two pieces together so that they do not move in a direction perpendicular to the cut—this extra step is described in §7.3 and justifies why we need center pieces for each rotation axis. Repeating this process for the other rotation axes results in the mechanism in Figure 18(a).

Caution needs to be taken when choosing the radius r' of the smaller sphere and the depth of the plane which cuts through it to make the attachment. We choose the radius to be large enough so that every piece intersects the sphere (Figure 19(a)). In fact, we noticed that in our fabricated examples, a larger sphere radius made for a more stable puzzle. We slice the spherical attachment by a plane $\boldsymbol{v}_i' = \boldsymbol{v}_i - d\hat{\boldsymbol{v}}_i$, where $\hat{\boldsymbol{v}}$ is $\boldsymbol{v}$ normalized, and d is a constant. An additional constraint is that the spherical attachments from different rotation axes must not collide with one another.

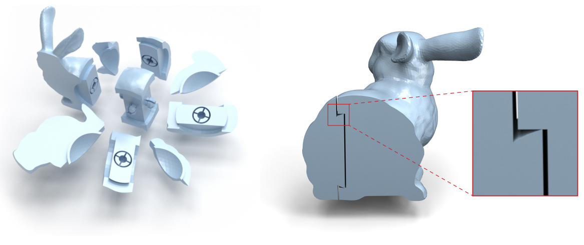

Figure 18: Puzzle Mechanism. *A look into the BUNNY mechanism (a), and isolating one rotation axis (b). The smaller piece has a spherical attachment that fits into the cylindrical hole of the larger piece, allowing it to rotate along the cylinder's axis.*

Therefore, for any non-intersecting rotation axes $\boldsymbol{r}_i = (\boldsymbol{v}_i, n_i)$ and $\boldsymbol{r}_j = (\boldsymbol{v}_j, n_j)$, we select d to be small enough such that $\boldsymbol{v}_i'$ and $\boldsymbol{v}_j'$ do not intersect inside the Jaap's sphere (Figure 19(b)).

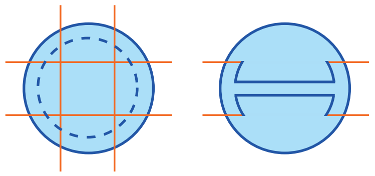

Figure 19: *The inner sphere should intersect all the pieces, and the attachments created from it should not collide with one another.*

We examine the shape of the internals of one of the edges of Rubik's Cube (Figure 3(b)). When the cylinder is carved out of the edge, the spherical attachments are carved out as well. Thus, we first create the spherical attachment by slicing the sphere by $\boldsymbol{v}_i'$, for all far halfspaces $\boldsymbol{r}_i^1$ (Lines 3-8 of Algorithm 1). Then, we carve out a cylinder of radius $\sqrt{(r')^2 - (\boldsymbol{v}_i')^2}$ at a depth of $\boldsymbol{v}_i'$ for each near halfspace $\boldsymbol{r}_i^0$ (Lines 9-12 of Algorithm 1). The radius of the cylinder is chosen to be exactly the radius of the circle where the sphere is sliced by $\boldsymbol{v}_i'$. Because the spherical attachment and the cylindrical hole are radially symmetric, there will not be any internal collisions when the puzzle is being turned.

7.3 Core Design

While 3D printers are now sophisticated enough to print interconnected pieces, there needs to be clearance between pieces to prevent them from fusing together. We found that printing twisty puzzles in this manner is undesirable: the mechanism is too loose to turn smoothly and sometimes pieces can fall out. Since centers do not interlock with the core, we need another mechanism to join them together. While most mass-produced twisty puzzles use screws to fix the center pieces to the core, we substitute in a snapping mechanism that can be 3D printed. The rotation joint exploits the fabricated material's flexibility and is illustrated in Figure 20. On the core piece, we add an appendage for each rotation axis, and on each center, we hollow out a shaft for the appendage to fit into (Lines 13-22 of Algorithm 1). The slits that cut through both the core and center pieces facilitate assembly by allowing the appendages to flex inwards and the shafts to flex outwards. As a result, the entire puzzle can be fully 3D printed and directly assembled.

When a puzzle has multiple cutting planes along the same rotation axis (such as a $5 \times 5 \times 5$ Rubik's Cube), there are multiple center pieces per rotation axis (some of which may be hidden entirely). Each of these needs to rotate independently, so they will all have the aforementioned joints. This compound design is illustrated by the CURVY5 example in §8.

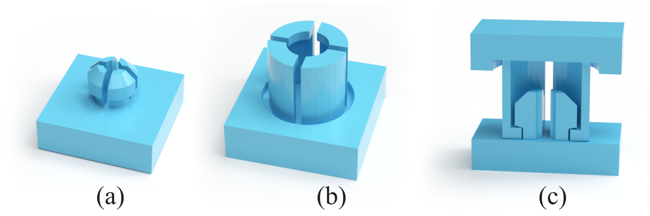

Figure 20: Center Mechanism. *Centers snap into the core piece by flexing the appendages and shafts, and once assembled they can only rotate in place. Each piece was created using CSG operations on cylinders, cones, and cubes.*

7.4 Extension: Curved Cuts

In all of the preceding discussion, rotation axes were specified by a plane that cuts through Jaap's sphere. However, we can use other surfaces for cutting up the puzzle. Instead of using a plane, we can use radially symmetric surface, such as a cone or a sphere (Figure 21). Concretely, when described using cylindrical coordinates (z, r, θ) where z coincides with the rotation axis, points on the cutting surface must have a z-coordinate independent of θ, that is, $z(r, \theta) = z(r)$. Because of the radial symmetry, we can rotate a piece along the rotation axis without internal collisions. To use the same mechanism design as presented above, we further require that the cutting surface is a plane inside of the Jaap's sphere, but can be curved outside of the sphere. In particular, curved cuts only affect the aesthetics of the puzzle, and not the combinatorics.

Although even-order Rubik's Cubes (like the $2 \times 2 \times 2$ Cube) have cuts which pass through the origin, these puzzles can be simulated using curved cuts on the next-highest odd-order cube to hide extraneous pieces. We also note that the idea of using a radially-symmetric curve to cut out pieces enabled Verdes [2004] to construct higher-order Rubik's Cubes.

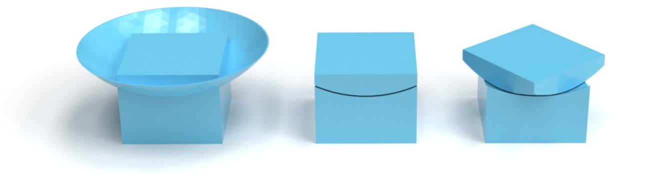

Figure 21: A spherical cut *passing through a cube.*

8 Results

We incorporated our pipeline into an interface where the user can draw cuts through the 3D model. Some of our results were fabricated using selective laser sintering (SLS) in a strong nylon material. Table 1 lists our examples with some properties such as the number of pieces and whether it uses curved cuts. The ARMADILLO example is shown in Figure 1. The total computation time in the pipeline was under a minute on a single-core processor for all the examples, though the step of the user drawing cuts had to be repeated a few times to get the desired output. For the examples with symmetric pieces (TETRA (Figure 23), CURVY5, and RHOMBI (Figure 26)), we generated just one of each unique piece and duplicated them to produce the whole puzzle.

The twisty flowerpot in Figure 25 is an example of our method generating transformable objects beyond just puzzles. In particular, these twisty joints can be used for reconfiguring or repositioning objects, such as the flowers inside the pot.

In our examples with curved cuts, the cutting surfaces we used were either spherical or conical. In general, the cutting surfaces

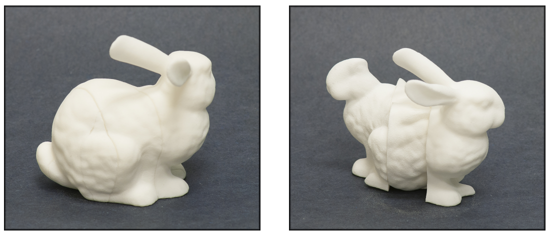

Figure 22: *Two configurations of the* BUNNY *puzzle.*

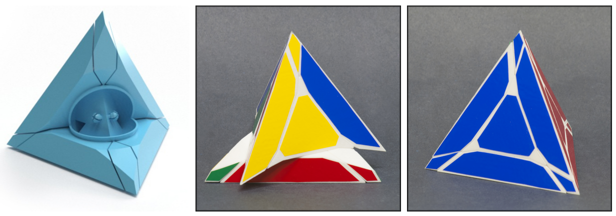

Figure 23: *While* TETRA *is a traditional, stickered twisty puzzle, the curved cut made up of two frustrums.*

can be generated by revolving a curve about the rotation axis, and the intersection of the cutting surface with the surface of the 3D model can be traced out in a manner similar to the one we described in §7.1 for cutting planes.

9 Discussion and Future Work

We proposed a method for interactively designing twisty puzzles based on Rubik's Cube, and even though our algorithm produces a wide variety of puzzles, there are still many extensions to explore. In our pipeline, we want to make Jaap's sphere as large as possible, so the input 3D model needs to have a large, round region for placing the sphere. If the puzzle is not round enough, there will be pieces that are completely outside of Jaap's sphere, and our method has no way of handling that case. Furthermore, not all rotation mechanisms have their rotation axes passing through the same origin, so one future direction is investigating the design of these more general mechanisms.

The classification of finite rotation groups does not encompass all non-blocking sets of rotation axes. For example, the group generated by the faces of the cube (i.e. rotations of 90° along the coordinate axes) is actually the entire octahedral group—there exist other sets of rotation axes which have a puzzle group structure (e.g., the puzzle in Figure 13) but which generate an infinite subgroup of SO(3). A classification of these more general sets of axes would enlarge the space of twisty puzzles our pipeline can produce. Furthermore, we restricted the possible sets of rotation axes only because we wanted to ensure non-blocking, but this is not a necessary property. Pieces that block one another can even enhance the puzzle's difficulty.

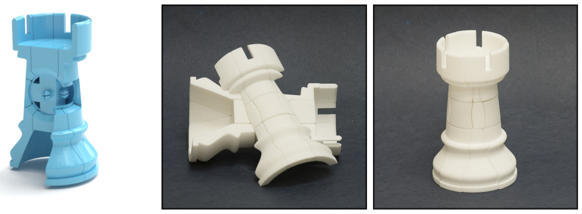

Figure 24: *Only one kind of curved cut is used in the* ROOK *puzzle, but some cuts appear flat because the model is radially symmetric.*

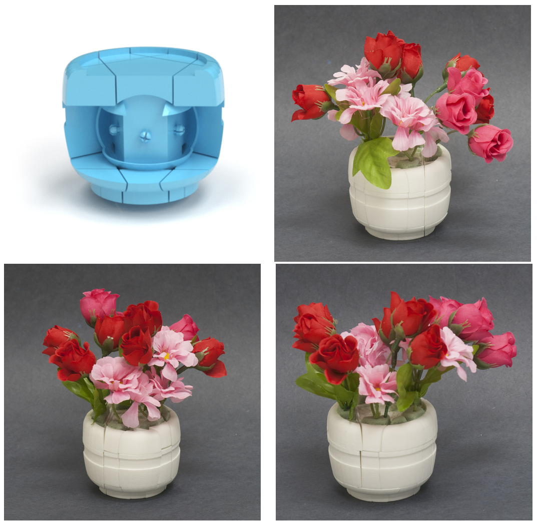

Figure 25: *Multiple arrangements of flowers in the* FLOWERPOT. *The twisty joint has* 14400 *possible arrangements for the flower pieces.*

We wanted to produce puzzles that are handheld and entirely 3D printed. However, the limited resolution of the 3D printer introduced errors in the models. The automatic placements of the rotation axes does not take into account the shape of the 3D model, so some of our models had thin and fragile parts near the cutting planes. Zhou et al. [2014] avoided printing small pieces by deforming the mesh, and perhaps something similar can be applied to twisty puzzles. Additionally, the joints connecting the core to the centers sometimes break in the printing process, and other times the puzzle is too loose and difficult to turn. One improvement to our design would be a joint design robust to printing errors that is just as easy to assemble. Assembly is sometimes a difficult task when many pieces are involved, and perhaps our method could benefit from automatically generated assembly instructions in the style of Xin et al. [2011] or an algorithm which designs the pieces in a clever way to simplify the assembly process.

Acknowledgements

We thank the anonymous reviewers for their feedback. We also thank Yun Fei for assistance in preparing the video, Dingzeyu Li for help on making figures, and Akash Garg, Yonghao Yue, and Shuang Zhao for their comments on earlier drafts. This research was supported in part by the National Science Foundation (CAREER-1453101) and generous gifts from Intel. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies or others.

A Good Rotation Axes

In the "two-generator" example of Figure 10, let $\psi_{\mathsf{U}}, \psi_{\mathsf{R}} \in \mathrm{SO}(3)$ be 90° rotations along the U and R axes, respectively. We observe that cutting along the U axis's plane after turning R clockwise is equivalent to cutting by an imaginary rotation axis defined by rotating the U axis by ψ_{R}^{-1} (Figure 10(d)). This axis is the result of composing the two rotations to get $\psi_{\mathsf{U}}\psi_{\mathsf{R}}$. In general, we could try to cut the puzzle after an arbitrary number of turns, so the axes we would be cutting along would be elements of the subgroup of SO(3) generated by ψ_{U} and ψ_{R}. In general, this subgroup will be infinite and Jaap's sphere will be cut by an infinite number of planes, leading to an infinite number of pieces. Since we want the number of pieces to be finite, we require that for a set of rotation axes $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_k$, the subgroup generated by $\psi_{\boldsymbol{r}_1}, \ldots, \psi_{\boldsymbol{r}_k}$ needs to be finite. Therefore, a set of desired rotation axes needs to correspond to a finite subgroup of SO(3), leading to the theorem described in §4.2.

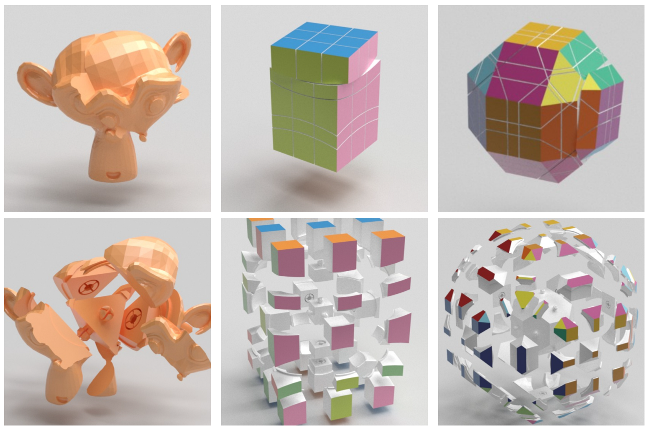

Figure 26: SUZANNE*,* CURVY5*, and* RHOMBI *and their internal mechanisms.*

Examples	Cuts	Pieces	Curved?	SO(3)
BUNNY	4	9	No	O
ARMADILLO	3	7	No	D_3
TETRA	4	15	Yes	T
ROOK	6	27	Yes	O
FLOWERPOT	7	33	No	D_5
SUZANNE	4	11	No	T
CURVY5	8	45	Yes	D_4
RHOMBI	20	99	No	O

Table 1: *A list of our examples with the number of cuts and pieces, whether it uses curved cuts, and the subgroup of SO(3) the cuts are derived from.* D_n*,* T*, and* O *refer to the dihedral, tetrahedral, and octahedral groups, respectively.*

B Flag Permutations

Two rotation axes are said to be *neighbors* if their planes intersect inside Jaap's sphere. The key property of the sets of rotation axes we consider is that the neighbors of a rotation axis $\boldsymbol{r} = (\boldsymbol{v}, n)$ can be permuted by rotating the entire Jaap's sphere around $\boldsymbol{r}$. For example, if we rotate Rubik's Cube around the U axis by 90°, we are effectively applying a permutation F → L → B → R → F to the neighbors of U. For each rotation axis $\boldsymbol{r}$, let τ_r be the permutation. For simplicity, we use τ_r to denote a permutation on axes, pieces, and flags. The permutation on rotation axes provides a convenient way of determining how pieces are permuted. Let $p = \{\boldsymbol{r}_1^{s_1}, \ldots, \boldsymbol{r}_k^{s_k}\}$ be a piece, and let $\boldsymbol{r}$ be a rotation axis such that p is on the far side of $\boldsymbol{r}$, i.e., $\boldsymbol{r}^1 \in p$. Then the move corresponding to $\boldsymbol{r}_i$ will move p to another piece's position, namely $\tau_r(p) = \{\tau_r(\boldsymbol{r}_1)^{s_1}, \ldots, \tau_r(\boldsymbol{r}_k)^{s_k}\}$. That is, we apply the permutation on rotation axes to each of the halfspaces that form p. An example of this permutation on Rubik's Cube is given in Figure 27.

As stated earlier, permutations on pieces are not enough to describe the state of the puzzle because of possible orientations of pieces. We extend the permutations on axes and pieces to a permutation on flags. The corresponding permutation is just

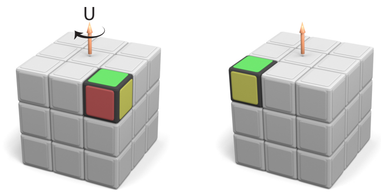

Figure 27: *The* UFR *corner is moved to* ULF *after turning the* U *axis clockwise by 90°. These permutations are computed by defining a group structure on the rotation axes.*

$\tau_r(p, \boldsymbol{r}_i) = (\tau_r(p), \tau_r(\boldsymbol{r}_i))$.

References

Bächer, M., Bickel, B., James, D. L., and Pfister, H. 2012. Fabricating articulated characters from skinned meshes. *ACM Trans. Graph. (Proc. SIGGRAPH) 31*, 4.

Calì, J., Calian, D. A., Amati, C., Kleinberger, R., Steed, A., Kautz, J., and Weyrich, T. 2012. 3d-printing of non-assembly, articulated models. *ACM Transactions on Graphics (TOG) 31*, 6, 130.

Ceylan, D., Li, W., Mitra, N. J., Agrawala, M., and Pauly, M. 2013. Designing and fabricating mechanical automata from mocap sequences. *ACM Trans. Graph. 32*, 6 (Nov.), 186:1–186:11.

Chen, D., Sitthi-amorn, P., Lan, J. T., and Matusik, W. 2013. Computing and fabricating multiplanar models. *Computer Graphics Forum 32*, 2, 305–315.

Coros, S., Thomaszewski, B., Noris, G., Sueda, S., Forberg, M., Sumner, R. W., Matusik, W., and Bickel, B. 2013. Computational design of mechanical characters. *ACM Trans. Graph. 32*, 4 (July), 83:1–83:12.

Ding, X., Lv, S., and Yang, Y. 2011. Configuration transformation theory from a chain-type reconfigurable modular mechanism-rubik's snake. In *13th World Congress in Mechanism and Machine Science*.

Golub, G. H., and Van Loan, C. F. 2012. *Matrix computations*, vol. 3. JHU Press.

Gower, J. C., and Dijksterhuis, G. B. 2004. *Procrustes problems*, vol. 3. Oxford University Press Oxford.

Igarashi, Y., Igarashi, T., and Mitani, J. 2012. Beady: interactive beadwork design and construction. *ACM Transactions on Graphics (TOG) 31*, 4, 49.

Khoudary, S., 2000. Mechanism for independently moving segments of a three-dimensional object and applications thereof, May 11. WO Patent App. PCT/GB1999/003,643.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. 1983. Optimization by simulated annealing. *Science 220*, 4598, 671–680.

Lo, K.-Y., Fu, C.-W., and Li, H. 2009. 3d polyomino puzzle. *ACM Trans. Graph. 28*, 5 (Dec.), 157:1–157:8.

Meyer, M., Desbrun, M., Schröder, P., and Barr, A. H. 2003. Discrete differential-geometry operators for triangulated 2-manifolds. In *Visualization and mathematics III*. Springer.

Mori, Y., and Igarashi, T. 2007. Plushie: an interactive design system for plush toys. *ACM Transactions on Graphics (TOG) 26*, 3, 45.

Müller, M., Heidelberger, B., Teschner, M., and Gross, M. 2005. Meshless deformations based on shape matching. *ACM Trans. Graph. 24*, 3 (July), 471–478.

Rivers, A. R., and James, D. L. 2007. Fastlsm: Fast lattice shape matching for robust real-time deformation. *ACM Trans. Graph. 26*, 3 (July).

Rokicki, T., Kociemba, H., Davidson, M., and Dethridge, J., 2010. God's number is 20. `http://cube20.org/`.

Scherphuis, J., 2003. Sphere. `http://www.jaapsch.net/puzzles/sphere.htm`.

Scherphuis, J., 2015. Mini cube, the 2×2×2 rubik's cube. `http://www.jaapsch.net/puzzles/cube2.htm`.

Shewchuk, J. R. 1996. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *Applied computational geometry towards geometric engineering*. Springer, 203–222.

Skouras, M., Thomaszewski, B., Kaufmann, P., Garg, A., Bickel, B., Grinspun, E., and Gross, M. 2014. Designing inflatable structures. *ACM Transactions on Graphics (TOG) 33*, 4, 63.

Song, P., Fu, C.-W., and Cohen-Or, D. 2012. Recursive interlocking puzzles. *ACM Trans. Graph. 31*, 6 (Nov.), 128:1–128:10.

Sorkine, O., Cohen-Or, D., Lipman, Y., Alexa, M., Rössl, C., and Seidel, H.-P. 2004. Laplacian surface editing. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, ACM, New York, NY, USA, SGP '04, 175–184.

Thistlethwaite, M. B., 1981. The 45-52 move strategy. `http://www.jaapsch.net/puzzles/thistle.htm`.

Thomaszewski, B., Coros, S., Gauge, D., Megaro, V., Grinspun, E., and Gross, M. 2014. Computational design of linkage-based characters. *ACM Trans. Graph. 33*, 4 (July), 64:1–64:9.

Thurston, W. P., and Levy, S. 1997. *Three-dimensional geometry and topology*, vol. 1. Princeton University Press.

Umetani, N., Kaufman, D. M., Igarashi, T., and Grinspun, E. 2011. Sensitive couture for interactive garment modeling and editing. In *ACM SIGGRAPH 2011 Papers*, ACM, New York, NY, USA, SIGGRAPH '11, 90:1–90:12.

Umetani, N., Igarashi, T., and Mitra, N. J. 2012. Guided exploration of physically valid shapes for furniture design. *ACM Trans. Graph. 31*, 4 (July), 86:1–86:11.

Verdes, P., 2004. Cubic logic toy, Feb. 12. WO Patent App. WO/2004/103497.

Wahba, G. 1965. A least squares estimate of satellite attitude. *SIAM review 7*, 3, 409–409.

Xin, S., Lai, C.-F., Fu, C.-W., Wong, T.-T., He, Y., and Cohen-Or, D. 2011. Making burr puzzles from 3d models. In *ACM Transactions on Graphics (TOG)*, vol. 30, ACM, 97.

Zhou, Y., Sueda, S., Matusik, W., and Shamir, A. 2014. Boxelization: Folding 3d objects into boxes. *ACM Trans. Graph. 33*, 4 (July), 71:1–71:8.