Boxelization: Folding 3D Objects into Boxes

Yahan Zhou[1] Shinjiro Sueda[1] Wojciech Matusik[2] Ariel Shamir[3,1]

[1]Disney Research Boston [2]MIT CSAIL [3]The Interdisciplinary Center

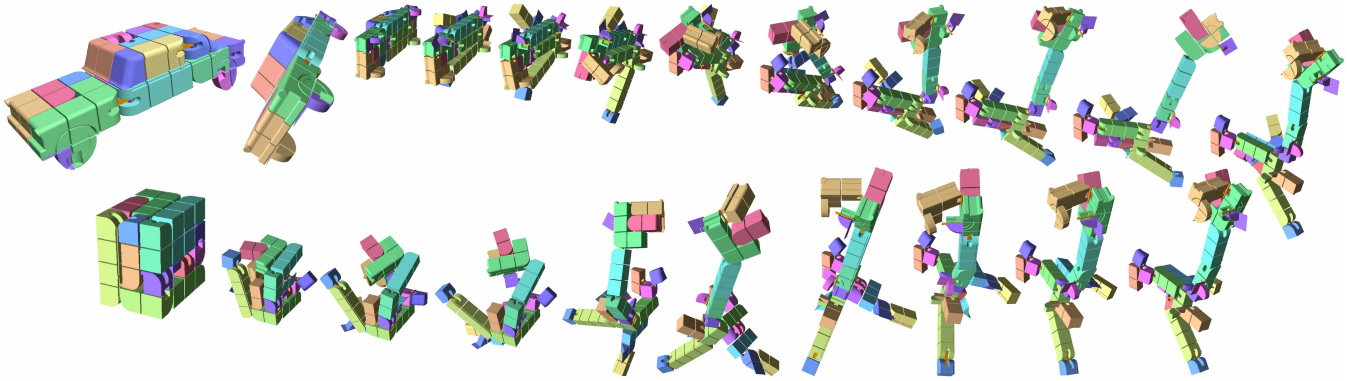


Figure 1: *Folding a car into a cube. Our system finds a collision-free folding sequence.*

Abstract

We present a method for transforming a 3D object into a cube or a box using a continuous folding sequence. Our method produces a single, connected object that can be physically fabricated and folded from one shape to the other. We segment the object into voxels and search for a voxel-tree that can fold from the input shape to the target shape. This involves three major steps: finding a good voxelization, finding the tree structure that can form the input and target shapes' configurations, and finding a non-intersecting folding sequence. We demonstrate our results on several input 3D objects and also physically fabricate some using a 3D printer.

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

Keywords: puzzle, folding, fabrication, interactive physics

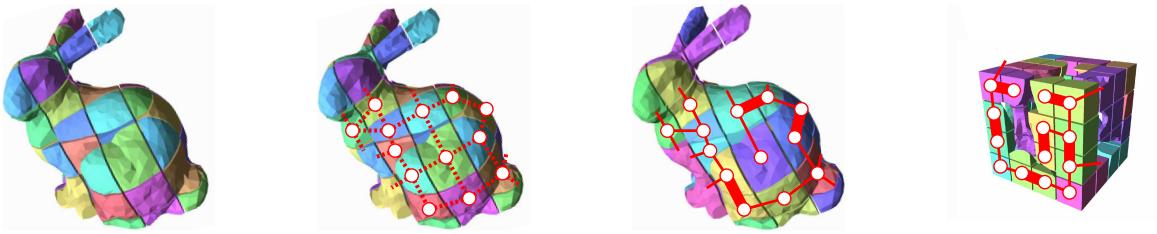
Links: DL PDF

1 Introduction

Humans are fascinated by objects that have the ability to transform into different shapes. Our interest is especially piqued when these shapes are dissimilar. Image-morphing and mesh-morphing have this type of appeal [Wolberg 1998; Lazarus and Verroust 1998], but they captivate us even more because watching the process of transformation is often the most compelling part. This has recently been exploited in motion pictures such as Transformers [2007]. Nevertheless, such transformations are applied in the virtual world and are often physically implausible. In contrast, recent works on the creation of 3D puzzles concentrate on physically creating objects composed of building blocks. These captivate us arguably for a similar reason—the building blocks do not resemble or hint as to the final shape [Lo et al. 2009; Xin et al. 2011; Song et al. 2012], but on top of that, they can be physically assembled and taken apart.

In this paper, we tackle both of these challenges together: creating *transformations* of 3D shapes that are *physically achievable*. We focus on one specific type of shape transformation: folding 3D objects into a cube or a box-like shape (Fig. 1). A cube is considered to be a special shape as it is highly symmetric and regular (one of the platonic polyhedra). Cubes and boxes are often seen as the most basic 3D shape that does not resemble any specific object. They can be stacked, stored and transported more easily, and used as "building blocks" for other shapes. Our work presents a method to create a fabricated 3D object that can physically fold between the input 3D shape and a box. Unlike previous works in computer-assisted fabrication that create disjoint pieces [McCrae et al. 2011; Luo et al. 2012; Hildebrand et al. 2012; Schwartzburg and Pauly 2013; Chen et al. 2013], our method produces a *single, connected* object that can be folded. Along with the visual appeal and functional advantages of stacking and transporting, our technique allows for reduced printing times and cost, due to the compactness and regularity of the shape.

Given the input 3D shape and the target box dimensions, finding a physically achievable folding sequence is a challenge as it involves many sub-problems that are interdependent. The input shape needs to be segmented into parts, and these parts need to be connected in a pattern that can fold into two configurations—the source and the target shapes. Both the static configurations as well as the dynamic folding sequence need to be physically achievable. This means that parts should be able to fold, joints should be sturdy, and self intersections or interlocking should not occur at both configurations and each step of the folding sequence. Any segmentation choice affects the connectivity pattern, which in turn affects the folding. This creates an intractable search space for possible solutions, and in general, this space is not continuous—for example, changing the segmentation by a small amount can lead to a drastically different connectivity solution.

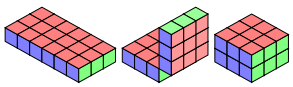


(a) *Voxelization* **(b)** *Connectivity Graph* **(c)** *Joint Tree* **(d)** *Folded result*

Figure 2: *An illustration of our method. (a) We first find the best voxelization of the input shape. (b) Geometric neighbors define the connectivity graph with nodes as voxels and edges as potential hinge locations. (c) We turn the graph into a tree. Some edges are removed, and some edges are turned into rigid links. The rest are assigned a joint type and a folding angle. (d) Once we compute the locations of the joints and their angles, the shape can transform into a box.*

Theoretically these problems can be shown to be very difficult. For instance, we examine the two subproblems of computing a segmentation with a connectivity structure (the joints), and finding physically achievable folding sequences for a given structure. For the first one, there exists an algorithm for placing joints given the common dissection between two shapes, but finding common dissection itself is an open problem [Abbott et al. 2008]. In addition, this algorithm tends to cut the shape into a large number of tiny structures, which are implausible for actual 3D printing. For the second subproblem, one can prove it is PSPACE-complete (more difficult than NP problems), by reducing it from the 2D linkage tree reconfiguration problem [Alt et al. 2004]. It is also well known in the protein folding community that just finding the minimum energy state given a set of joints is NP-complete [Berger and Leighton 1998].

To make the search space tractable, and to find a plausible solution we make some underlying design choices. Instead of using arbitrary segmentation and arbitrary joint angles, we use voxels as our folding primitives with a discrete set of joint angles between them (see Fig. 2a). Hence, our segmentation problems turns into a voxelization problem. Next, we must choose the connectivity structure for the voxels. Joints will be placed only between connected pieces that need to move during folding. Since the whole object must be connected, such a pattern forms a connectivity graph on the voxels (Fig. 2b). Connectivity loops in this graph are plausible and could potentially increase the stability of the static configuration. However, since they typically cause complex locking patterns in the folding sequence, we choose to constrain this graph to a tree structure (see Fig. 2c). Each tree edge represents a connection between neighboring voxels. If these voxels must move relative to each other during the folding sequence, a joint must exist. Our problem is therefore to choose the location of these joints and then compute the angles so that the initial shape will fold into the target shape (Fig. 2d).



In some cases, instead of using a box directly as the target shape, we will use a template that can be easily folded into a box (see example on the right). Using such a template not only makes the search for solution easier but also reduces the printing time, since we can print the object in a compact, flattened state.

Even after limiting the scope to voxels, the size of the search space is still too large, and therefore we cannot hope to exhaustively search through all possible folding patterns. Finding a solution manually is possible only for small examples with a handful of pieces (e.g., cubebots [Weeks 2013]). We want to be able to produce outputs with as many as 125 pieces, as shown in Fig. 7f. We use simulated annealing [Kirkpatrick et al. 1983] along with *beam search* [Lowere 1976] to search the space of solutions.

We seek a solution that optimizes a number of objectives:

1. **Geometric fit**: The folded object must match the target shape.
2. **Compactness**: The space wasted in the folded shape must be minimized.
3. **Fabricability**: All the joints and connectors must be printable. Small pieces must be avoided.
4. **Foldability**: There must be a physically achievable sequence of moves to fold/unfold the shape with no intersections.

Trying to solve all of these at once imposes a major challenge. The key to our solution is the separation of the problem into three stages: defining the shape, finding the connectivity structure, and finding the folding sequence. In the first step, we search for a good voxelization pattern of the input 3D shape following the first three objectives above (§3). Because the voxels in the input shape are packed, it is difficult to search for a solution that already maintains all the objectives. In the second step, we simultaneously build a connectivity tree between the voxels and search for a folding sequence that transforms the input shape into the target shape by following only the first objective above (§4). This step only defines the connectivity *structure* of the object that can fit the source and target configurations. Only in the third step we follow the fourth objective and search for a non-intersecting folding sequence. However, instead of searching for a folding sequence from the source shape to the target, we utilize a physical simulator to *unfold* both configurations and match them. This provides a valid sequence of folding moves that will transform the object from the input shape to the target shape in a plausible manner (§5).

2 Related Work

There is a large body of work on each of the sub-problems we face: segmentation (or voxelization), joints placement, and folding. We are not aware of a work that combines these to solve a folding problem similar to ours.

Shape segmentation is an active area of research [Shamir 2008; Chen et al. 2009]. More specifically, voxelization of 3D objects is useful for physical simulation and analysis, for medical imaging and visualization, and for computer graphics and games [Varadhan et al. 2003; Pantaleoni 2011; Loop et al. 2013; Chang et al. 2013]. In our setting, the constraints on the voxelization shape and size arise from the fabricability and geometric-fit objectives, which were not used explicitly before.

Foldable designs have long been created for furniture and other useful objects (umbrellas, chairs, tents etc.). Our domain is closer to recreational puzzles and art forms such as popup books [Li et al. 2010; Li et al. 2011], papercraft toys [Mitani and Suzuki 2004], and cubebots [Weeks 2013]. Recently, several works have presented

methods to create puzzles of various types from 3D objects. These include polyominoes [Lo et al. 2009], burr puzzles [Xin et al. 2011], interlocking puzzles [Song et al. 2012], dissection puzzles [Zhou and Wang 2012], or sliding planar slices [Hildebrand et al. 2012]. However, all these create disjoint-pieces puzzle, while we seek a single connected object folding into two shapes. The addition of joint constraint to keep the pieces connected presents new challenges not encountered in previous methods.

Folding of paper to create various shapes (Origami) has been studied extensively [O'Rourke 2011]. More recently this has been extended to developable surfaces with curved folding [Kilian et al. 2008], and to the creation of polyhedral surfaces [Tachi 2010]. Our work can be seen as a type of voxel-Origami (or "ori-voxel") since, once we find a solution, we can begin from simple boxes and fold them into various 3D-shapes.

As mentioned earlier, finding a folding pattern can become a very challenging problem [Alt et al. 2004] and in some cases even present an intractable search space. This complexity also appears in related fields such as protein folding [Berger and Leighton 1998; Istrail and Lam 2009]. Some very nice mathematical results for linkages, planes, and polyhedra are summarized by Demaine and O'Rouke [2007]. Our specific problem is close in spirit to linkages, but in our case, the parts, configuration, and structure of links are unknown as well.

Computer assisted fabrication of objects is a new area of research emerging from graphics, CAD, and design [Séquin 2012]. Fabrication in-parts create tangible, physical artifacts either by using shape proxies such as planar boundary pieces [Chen et al. 2013] or planar slices [Schwartzburg and Pauly 2013; Hildebrand et al. 2012; McCrae et al. 2011], or by segmenting the object to pieces for assembly [Luo et al. 2012; Lau et al. 2011]. In all these works, the object is cut into disjoint parts and reassembled, while our work searches for a single foldable object. A somewhat similar problem in terms of printing a single model, but for the creation of articulated models, was presented recently [Bächer et al. 2012; Calì et al. 2012]. Their challenge is more to assure pieces will function in a single configuration, while ours is to find a shape that can take on two different configurations. Similar to ours, most fabrication methods allow minor shape modifications to comply with some given constraints. Shape modifications were also used to increase stability [Prévost et al. 2013; Bächer et al. 2014] or allow stackability [Li et al. 2012]. In our case, we optimize a small warp of the shape so that small voxel pieces are avoided.

3 Voxelization

The first step in our approach is to find a voxelization of the input shape that will meet our objectives. Voxelization is performed by placing a grid around the object and marking the voxels that contain any part of the object. By intersecting the voxels with the object mesh we create the set of pieces for folding. For convenience, we continue to call these pieces "voxels," even though some of them are only partially filled voxels.

We use cube-shaped voxels as they allow full freedom of movement in folding and placing of hinges. Hence, the free parameters for voxelization are the dimensions of the grid and its position and orientation in space. Because our target shape is a box or a template that can fold into a box, we fit the dimensions of the grid so that the number of pixel pieces will be equal or smaller than those of the box (we used several box sizes from $3\times4\times4$ to $5\times5\times5$). We therefore search only for the orientation and position of the grid. In addition, we allow small deformations of the input object to optimize the fit into the voxels as will be described below. In general, the voxelization grid can also be defined and positioned manually by the user.

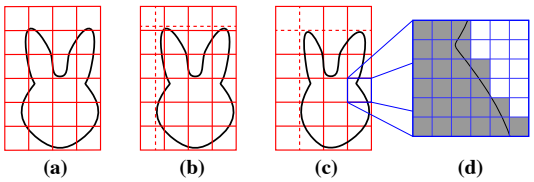
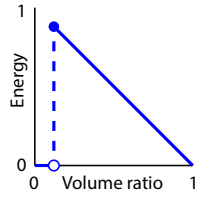


Figure 3: *(a) The original, uniform voxelization may contain some small parts. (b) We apply a small offset to each of the planes to minimize the voxelization energy. (c) We move the planes back to their original locations, which deforms the mesh parts in the voxels. (d) We further divide each voxel into sub-voxels.*

To meet the printability criterion, our main goal in voxelization is to make sure that the actual volume of each final voxel piece is large enough to support and hold the connecting hinges and be printable. Moreover, the closer the shapes of the pieces are to full voxels, the easier it would be to fill a target box shape with little waste of space (compactness). Hence, we define the "fullness" objective function as follows:

$$E_{\text{vox}} = \sum_{v \in V} \begin{cases} 0 & \text{if } M \bigcap v = \emptyset, \\ 1 - \frac{\text{volume}(M \cap v)}{\text{volume}(v)} & \text{otherwise,} \end{cases} \quad (1)$$

where M is the input mesh, V is the voxelization, and v is a voxel. $M \bigcap v$ is the intersection of M and v. Although mesh intersection can be used to compute the volume, we instead use a voxelization approach once again. After the grid is chosen, we subdivide the grid further so that each voxel is composed of 20x20x20 subvoxels (Fig. 3(d)). The volume of intersection between the mesh and a voxel, $M \bigcap v$, can then be efficiently approximated by counting the subvoxels occupied by the mesh inside each voxel. These subvoxels also used for non-uniform voxelization and the evaluation of the folding objective function, described below.



The graph of Eq. 1 is shown in the inset figure. This function penalizes voxels occupied by a small portion of the input mesh but does not penalize empty voxels. We do not need a threshold since the volume computation using subvoxels means the volume ratio takes on discrete values. To optimize this function we choose different randomized rotation and translation of the grid, and keep the best results after applying the non-uniform voxelization step.

Non-uniform voxelization To lower the objective function further we allow slight deformation to the input shape by locally offsetting the grid planes, as shown in Fig. 3(b-c). We limit the offset of the grid planes so that the grid spacing does not change by more than 10-20% along the normal of the plane, to ensure that the distortion to the input shape would be small. This corresponds to each of the planes having the freedom to move $\pm2 - 4$ subvoxels. The X, Y, and Z planes are adjusted using a block coordinate descent approach; we hold two of the directions fixed and adjust the planes in the remaining direction. Optimal solution in terms of the energy function (1) is found using dynamic programming in each direction. We iterate between the three directions (X, Y, Z, X, Y, Z, ...) until convergence, which in our examples tend to be around 3-4 iterations per direction. Once the optimal grid plane offsets are found, they are moved back to their original positions, carrying along with them the input mesh, which results in a slightly deformed mesh with fewer small voxel pieces.

4 Tree Fitting

After the object is segmented into voxels, we need to find the connectivity between the voxels so that the resulting object can be folded into the target shape. The voxels created in the previous step do not yet have any joints between them. However, they do provide the geometric neighborhood information that defines the potential joint locations—we can only add joints between voxels that contain part of the object along their shared face (Fig. 2b). Our final goal is to define an undirected tree to represent the connectivity between the voxels: nodes correspond to the voxels, and edges correspond to joints that connect the voxels (Fig. 2c). As mentioned earlier, we do not allow loops, as they almost always create over-constrained configurations. The objective of the fitting step is to find a low energy tree that spans all the voxels by assigning a joint type to each pair of neighboring voxels. The energy we use is defined in §4.1. The joints are parameterized by the following types:

- ***N***ull: No joint is added between the voxels and they can be separated. These correspond to the dotted edges in Fig. 2b that were removed in Fig. 2c.
- ***R***igid: The nodes, and the voxels that they represent, are attached rigidly. This means there is no hinge between these voxels and they move together. These correspond to the thick edges in Fig. 2c.
- ***S***ingle hinge: A simple hinge that connects the voxels with a single axis of rotation, as shown in Fig. 4a (top left). There are 4 types of single hinges, corresponding to the 4 rotation directions of the child voxel with respect to the parent voxel.
- ***D***ouble hinge: A hinge with two axes of rotation connecting the voxels. This joint type provides a rich set of transformations of the child with respect to the parent, some of which are shown in Fig. 4.

Using this parameterization, the search boils down to assigning a joint type to each graph edge in Fig. 2b so that the end result is a tree, as in Fig. 2c.

The double hinge provides a rich set of transforms for the tree fitting stage while still being simple enough for physical printing. We parameterize the double hinge by the two axes of rotation it provides: the 1st axis between the parent voxel (shown in pink in Fig. 4a) and the link body (green), and the 2nd axis between the link body and the child voxel (purple). The parameterization can be described compactly as "*[axis][sgn]:[axis][sgn]*", where *[axis]* can be X, Y, or Z, and *[sgn]* can be -, - -, +, or ++. We use "-" to indicate a -90°rotation, "- -" for -180°, "+" for +90°, and "++" for +180°. For example, the 3 double hinges in the figure are Z-:Z-, Z-:Y-, and X+:Z-. A sample transform is shown in Fig. 4b—with respect to the parent voxel, the child voxel translates to the +Z position and rotates by -90°around the Z-axis. With a double hinge, a child voxel can be transformed to a total of 78 distinct axis-aligned configurations in $SE(3)$, after all the double counting has been accounted for (e.g., Y++:Y++ and Y- -:Y- - give the same transform).[1] This is in contrast to the single hinge, which only provides 4.

We can now define the search space formally. Let x_i be the joint type of the i^{th} edge. Then the assignment of edge types can be expressed as

$$x_i \in \{N, R, S_{Z+}, \ldots, D_{Z+:Z+}, \ldots\}, \qquad i = 1, \ldots, n, \quad (2)$$

where n is the number of edges, and N, R, S, and D correspond to the joint types listed above.

[1]The total number of axis-aligned configurations is 144. There are 6 different positions for the child with respect to the parent: $\pm X, \pm Y, \pm Z$. For each of these positions, there are 6 different ways in which the X-axis of the child can point, and after that 4 more choices for the Y-axis.

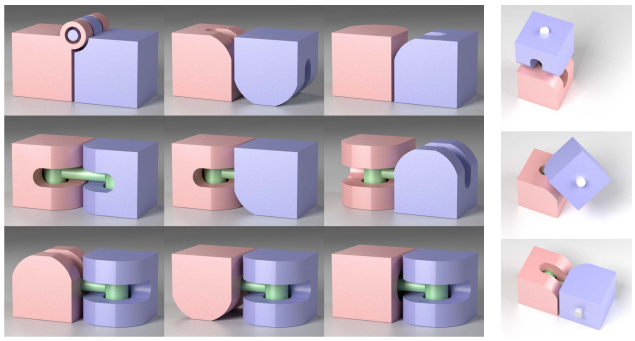


Figure 4: *(a) Examples of hinge types. The X-axis is to the right, Y is into the paper, and Z is up. Top row: "Y-", "Y-:Y++" & "Z++:Y-". Middle row: "Z-:Z-", "Z-:Y+" & "Z- -:Y- -". Bottom row: "Y- -:Z- -", "Y++:Z- -" & "X+:Z- -". (b) Example motion sequence of a double hinge.*

Let V_0 be the transform of the root node of the tree, which is chosen randomly. Given a sequence of joint types, $[x_1, x_2, \ldots]$, starting from the root transform, we can compute the transformation of each voxel, V_i, by traversing the tree from the root to the voxel.

$$V_i = \text{VoxelTransform}(V_0, [x_1, x_2, \ldots]). \quad (3)$$

We use V_i to denote the transformation of the i^{th} voxel, i.e., the 4x4 $SE(3)$ matrix that transforms from voxel's local coordinates to world coordinates. Depending on the context, we also use V_i to denote the final position of the i^{th} voxel in $\mathbb{R}^3$. We also use a similar traversing function to compute the position and orientation of the i^{th} joint.

$$J_i = \text{JointTransform}(V_0, [x_1, x_2, \ldots]). \quad (4)$$

If we randomly assign values to the edges, then the resulting folded configuration will almost always suffer from collisions. Instead, we build a collision free configuration incrementally using a tree search. Starting from a randomly chosen root node, the fitting step advances on the graph using beam-search, an extension of best-first search that sorts and keeps the top partial solutions whenever a new search path is explored. Unlike breadth-first search and its variants, beam search keeps the memory footprint small by throwing away paths that look to be the least promising. The tree is expanded one edge at a time while keeping the resulting partial configuration collision free. The search ends when the tree spans the voxels and all edge types have been determined.

4.1 Fitting Energy

The energy is a function of the root transform and the sequence of edge types: $E(V_0, [x_1, x_2, \ldots])$. As we build the tree, we evaluate the energy whenever the tree is expanded by adding an edge. Initially, the tree only contains the root node, so the energy is $E(V_0, [\,])$, and only the transform of the root is known. Then, the edges incident to the root, which is the current *frontier*, are evaluated, and the most promising ones are added to the frontier. For brevity, we use $E(\boldsymbol{x})$ to indicate $E(V_0, [x_1, x_2, \ldots])$.

The energy function has four terms.

$$E = E_{\text{collision}} + E_{\text{template}} + E_{\text{surface}} + E_{\text{count}}. \quad (5)$$

The first two terms are hard constraints, and the last two are energy objectives.

Collision The collision term constrains the folded shape from placing voxels or joints at the same location in space: $E_{\text{collision}} = E^V_{\text{collision}} + E^J_{\text{collision}}$. We do, however, allow for two partially-filled voxels to be at the same location if their meshes do not overlap when placed at the same location. The joint collision term is required to prevent two single hinges to reside on the same edge of the voxel, or from two double joints to originate from the same voxel.

$$E^V_{\text{collision}}(\boldsymbol{x}) = \begin{cases} \infty & \text{if } V_i(\boldsymbol{x}) = V_j(\boldsymbol{x}), \\ 0 & \text{otherwise,} \end{cases}$$
$$E^J_{\text{collision}}(\boldsymbol{x}) = \begin{cases} \infty & \text{if } J_i(\boldsymbol{x}) = J_j(\boldsymbol{x}), \\ 0 & \text{otherwise,} \end{cases} \tag{6}$$

for some i and j. The equality in this equation only checks for the positions of voxels V_i and V_j and not their orientations. Voxel collisions are trivial to compute using the subvoxels computed in the voxelization step from §3.

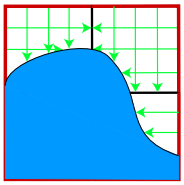
Template The template term constrains the folded shape to match the target template and is again composed of two subterms that correspond to voxels and joints: $E_{\text{template}} = E^V_{\text{template}} + E^J_{\text{template}}$. A template defines sets of positions, T_V and T_J, that the folded voxels and joints, respectively, are allowed to take.

$$E^V_{\text{template}}(\boldsymbol{x}) = \begin{cases} \infty & \text{if } V_i(\boldsymbol{x}) \notin T_V, \\ 0 & \text{otherwise,} \end{cases}$$
$$E^J_{\text{template}}(\boldsymbol{x}) = \begin{cases} \infty & \text{if } J_i(\boldsymbol{x}) \notin T_J, \\ 0 & \text{otherwise,} \end{cases} \tag{7}$$

Constraints on the placement of joints imposed by partially filled voxels are included in the joint template term. For instance, a single hinge cannot be constructed on a voxel piece unless the edge it is assigned to contains a large enough part of the object so as to position the hinge geometry on. Since most voxels are only partially filled, this constrains the search considerably.

Note that the template and collision energy terms are hard constraints. If they are violated, then the tree search prunes off the branch and searches down another branch. The following two terms are used as soft constraints to differentiate between feasible solutions.

Surface Because our goal is to create a box whose faces should be as planar as possible, we want the outside faces of boundary voxels in the target configuration to be filled. We use the surface energy to encourage this behavior. A 2D illustration is given in the inset figure. The red voxel edges form the boundary surface of the template. Rays, shown in green, are shot from the boundary until they hit the surface or the edge of the voxel. The ray distances are integrated to give the energy for that voxel. The energy is minimized when the shape matches the boundary and is maximized when the voxel location is unoccupied.



$$E_{\text{surface}}(\boldsymbol{x}) = \sum_i \int \text{ray distance}. \tag{8}$$

In the inset figure, the top two voxels have high energy, the lower right voxel has low energy, and the lower left voxel has zero energy. For interior voxels that do not contain a border, we set the energy to be zero. Instead of actually shooting rays and calculating distances we use the subvoxels from §3.

Counting The final energy term counts the number of joints. Whenever possible, we prefer solutions with a fewer number of joints as it will make folding simpler. Furthermore, some joint types are preferred over others since they require less modification to the input mesh. Each joint type is given a weight, and we simply sum the weights to compute the energy.

$$E_{\text{count}}(\boldsymbol{x}) = \sum_i \|x_i\|, \tag{9}$$

where $\|\cdot\|$ denotes the numerical weight given to each joint type listed in Eq. 2.

4.2 Simulated Annealing

The tree fitting step returns a list of solutions ordered by the energy value. Since the first two energy terms are hard constraints, these solutions are guaranteed to be collision free and to fit inside the template. Usually, however, just one tree search does not give a satisfactory solution—some solutions have poor surface energy, and others have too many joints to be printable. Therefore, we combine the tree search with simulated annealing. Initially, the annealing temperature is set to be high, which means that the tree search is run many times with random position and orientation of the root voxel. This portion of the algorithm is embarrassingly parallelizable. After we have a certain number of solutions, we lower the temperature gradually, so that whenever a good solution is found, we start the search using a partial subtree from that solution.

4.3 Geometric Post-processing

Once the joint types are determined, we must modify the voxels to include the geometry of the joints. We are guaranteed not to have any hinges on an empty edge of a voxel, because of the hard constraints applied in the tree search. We must also carve out some geometry from the voxels to enable proper motion of the joints. As shown in Fig. 4a, a single hinge is less obtrusive than a double hinge to the voxel geometry, requiring less of the voxel to be carved out. At this stage, we only look at neighboring voxels. Sometimes, it is necessary to carve out the corners of the voxels due to global contacts, and this is addressed in the next section.

5 Interactive Folding

The tree search only considers collisions in the folded state and not during the movement of the voxels in the folding sequence. This means that the computed solution may not be physically foldable when manufactured. We mitigate this problem by disallowing loops in the connectivity graph, but we must still verify that the computed solution can be folded without collisions. We use a semi-automatic approach that combines a physical simulator and user interactions. The key idea here is that physics is quite effective at *unfolding* even though it does not work well for *folding*.

The process of folding the original shape (Shape A) into the target shape (Shape B, for "B"ox) is broken up into two steps as shown in Fig. 5: unfolding and matching. First, the simulator simultaneously tries to *unfold* both the original shape (A) and the folded shape (B) by applying a repulsive force ($\propto 1/r^2$) between all pairs of voxels within the shapes. The simulator can be any off-the-shelf rigid body dynamics engine that supports joint constraints and collisions. At any time, the user can guide the system by supplying additional external forces or by pinning certain voxels strategically.

After both shapes have been unfolded adequately, the repulsive forces are removed and attractive forces between the corresponding voxels from A and B are added to *match* their shapes. Once A and

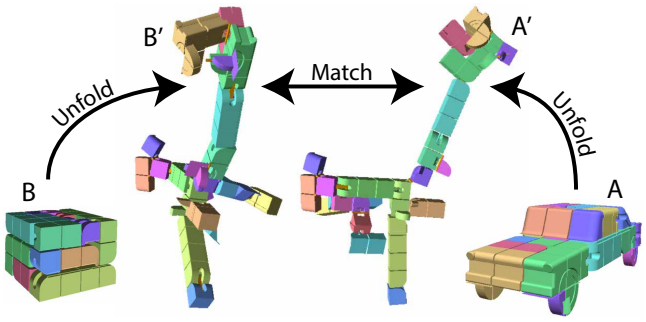


Figure 5: *Using physics to find a folding sequence. Using repulsive forces and some user interactions, A and B can be unfolded into A' and B'. (The reverse is not easy to do with physics.) Then A' and B' are matched to each other using attractive forces. Once A' and B' coincide, we have a collision-free path from A to B and vice-versa.*

B take on the same configuration, we have a valid folding sequence from A to B, passing through the intermediate unfolded configuration. Note that the order of first unfolding and later matching is very important. Without unfolding first, the matching force will almost always cause the shapes to get stuck due to collisions and will not be able to cause A and B to reach the same intermediate configuration.

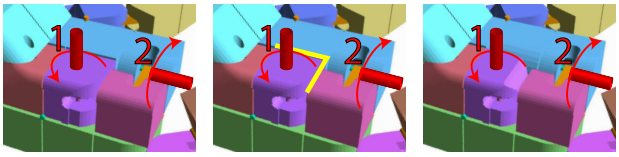
Interaction Fig. 6 shows a stage in the unfolding process where the physics simulator has managed to unfold most of the joints but is not able to untangle a small portion of the shape. The user intervenes and decides that the correct ordering to unfold is to rotate the purple voxel about the axis labeled "1." However, before the purple voxel can be rotated, its corners must be carved out, since otherwise collisions will constrain the rotation physically (Fig. 6b). After carving the appropriate corners, the physics simulator can continue to unfold the remaining voxels (Fig. 6c). It took around 5-10 minutes of interaction to obtain a physically foldable solution for all results in this paper.

The interactive physics simulator acts as a filter that semi-automatically removes physically unfoldable solutions. If we find a valid folding sequence using the interactive simulation process, then we know that the solution is valid. Note however, that if we cannot find it, we cannot guarantee that there is no solution. Also, if there is a valid solution, we are not guaranteed to find it. In practice, this simulator did assisted in filtering out some implausible solutions found in the tree search.

6 Results

We used our system to create a foldable bunny (Fig. 7a), kitten (Fig. 7b), car (Fig. 7c-d), dragon (Fig. 7e), and elephant (Fig. 7f). For all our examples, we used the following energy weights: $w_{\text{surface}} = 0.3$, $w_{\text{count(N)}} = 0$, $w_{\text{count(R)}} = 0$, $w_{\text{count(S)}} = 0.1$, and $w_{\text{count(D)}} = 0.12$. The first three results were physically manufactured using the Objet500 Connex 3D printer. For the tree search, we run the search in parallel using a cloud computing service. We ran the tree search for up to ∼30 hours with random restarts for each object. Then we sort the generated solutions by the energy (all hard constraints are satisfied), and starting from the best solution, we interactively folded for around 5-10 minutes until we found a good working solution.

Table 1 lists the running time of the tree search, the number of joints in the computed solution, and the number of voxels carved during the interactive folding session. "Wall" is the wall-clock time to find the solution used in the examples, "CUs" is the number of normalized



(a) (b) (c)

Figure 6: *(a) The user sees that the correct unfolding sequence is to rotate the purple voxel around 1 and then the pink voxel (which is the purple voxel's parent) around 2. (b) But the corner (marked in yellow) on the purple voxel prevents rotation. (c) The user carves some edges of the purple voxel and continues unfolding.*

compute units employed (roughly equivalent to a single 1GHz core), and "total" is the product of these two numbers, which is the total number of core hours. Note that these numbers represent the amount of time the tree search took to find the solutions used for the results, not the total time we ran the tree search (up to ∼30 hours). As can be seen, a reasonable solution can be found in the 5-15 hour range, depending on the example. As this process is embarrassingly parallelizable, these numbers can be further reduced by using more processing power.

We found the animation produced by the interactive simulation, as shown in Fig. 1, to be extremely useful as a guide for folding and unfolding the 3D-printed prototypes (car, bunny, and kitten). Even for the simplest example, it is non-trivial to fold the shape from one to the other without the aid of the provided animation.

The U-car example shown in Fig. 7d demonstrates that shapes other than cubes can be used as the target template, as long as it is a voxelized shape.

For the dragon shown in Fig. 7e, the best results in terms of energy were generated for a voxelization that has some voxels that cover disconnected pieces. In these cases, we added "struts" between disjoint pieces.

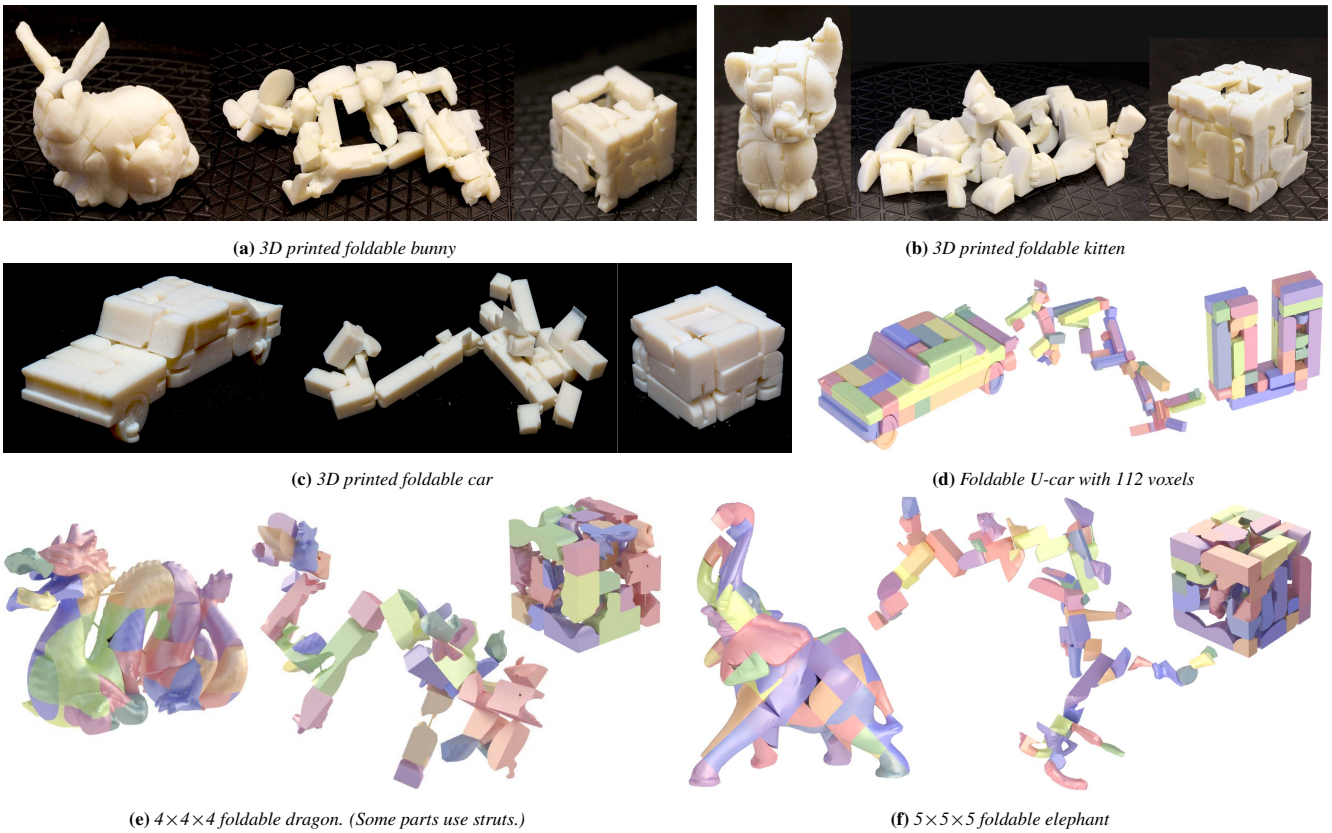
The final example, the elephant shown in Fig. 7f, uses a 5×5×5 voxelization.

Table 1: *"Size" is the voxelization resolution. "wall" is the number of wall-clock hours to find the solution used in the examples. "CUs" is the number of compute units used (roughly equivalent to a single 1GHz core). "Total" is the total number of compute hours, i.e. the product of wall and CUs. "Joints" is the number of joints that are added to the shape. "Carves" is the number of voxels that are carved with the interactive simulator.*

	size	wall (h)	CUs	total (h)	joints	carves
bunny	4×4×4	4.9	50	245	29	7
kitten	4×4×4	6.9	50	345	34	3
car	3×4×4	11.5	14	161	22	3
U-car	112	16.1	50	805	45	19
dragon	4×4×4	4.3	50	215	32	3
elephant	5×5×5	13.4	75	1005	62	7

7 Conclusion & Future Work

We have introduced a method for folding 3D shapes into cubes and boxes. Objects designed with our method can be physically printed and folded from one shape to the other without collisions. We segment the input shape into a set of voxels and find a tree that connects these voxels with joints. We make the problem tractable by dividing the algorithm into three major steps: voxelization, tree fitting, and



(a) *3D printed foldable bunny*

(b) *3D printed foldable kitten*

(c) *3D printed foldable car*

(d) *Foldable U-car with 112 voxels*

(e) *4×4×4 foldable dragon. (Some parts use struts.)*

(f) *5×5×5 foldable elephant*

Figure 7: *The first three objects (a-c) are physically manufactured using a 3D printer. (d) The U-car demonstrates that the target does not need to be a box. (e) The dragon contains some struts due to the challenging geometry. (f) The 5×5×5 elephant is the largest example produced.*

interactive folding. In the voxelization step, we find a good segmentation of the input shape that reduces small pieces. In the tree fitting step, we use beam search and simulated annealing to find the joint types and locations that minimize our energy function, temporarily ignoring collisions during folding. Finally, in the interactive folding step, we use a physics simulator to unfold both the source and target shapes in order to validate that a collision-free folding sequence can be generated for the computed solution.

Currently, it is difficult to fabricate tight-fitting joints with no play, and this causes our final output, which is printed in a single piece, to be weaker than desired. One potential way around this problem is to modify the geometry of the joints [Bächer et al. 2012; Calì et al. 2012], but such techniques are designed for larger joints and are difficult to apply to our intricate results. Fortunately, digital manufacturing technologies are constantly improving, and so we expect our framework to be more and more practical in the future. Also, because the joints are very lose, we resort to applying a small amount of glue or a putty to hold parts together. To produce more robust models, we would need to design, either automatically or semi-automatically, hooks, pegs, or other types of retention system.

The qualities of the computed solution and the fabricated result are not perfect. There is a trade-off between the amount of inner void and the completeness of the outer surface of the folded shape. This can be changed, if desired, by modifying the surface energy (Eq. 8).

Our automatic voxelization method does not honor important features of the model, such as the eyes and wheels. A user interface for specifying which parts of the model to not segment would be useful. Note that because of the way we divided the algorithm into three stages, we can also run voxelization with user-specified constraints in an interactive manner, without affecting the optimization stages.

We only included hinge joint in our examples. Other joints, such as prismatic, cylindrical, or even linkages, would add more rich set of transforms. A telescoping joint would be very interesting to add, since this would enable us to hide a piece inside another larger piece.

We showed in Fig. 7d that the template does not necessarily need to be a box, but because of our formulation, the target shape must be composed of voxels. However, it is possible to explore ways to carve away the inside of the shapes to make them transform to other shapes as well. Also, it would be useful to add another energy objective that allows the designer to specify where in the target shape each segmented voxel maps to. This could potentially allow the shape to transform more naturally to the target shape.

Our physics simulation result serves as a useful guide for folding and unfolding the shape. However, it would be better if we could generate a step-by-step manual rather than a continuous animation.

Finally, it is possible to connect multiple outputs from our system to create one big output. For example, it would be amusing to create a robot where the head is made of a bunny, the torso from an elephant, the arms from kittens, and the legs from dragons.

References

ABBOTT, T. G., ABEL, Z., CHARLTON, D., DEMAINE, E. D., DEMAINE, M. L., AND KOMINERS, S. D. 2008. Hinged dissec-

tions exist. In *Proc. 24th Annual Symposium on Computational Geometry*, 110–119.

ALT, H., KNAUER, C., ROTE, G., AND WHITESIDES, S. 2004. On the complexity of the linkage reconfiguration problem. *Contemporary Mathematics 342*, 1–14.

BÄCHER, M., BICKEL, B., JAMES, D. L., AND PFISTER, H. 2012. Fabricating articulated characters from skinned meshes. *ACM Trans. Graph. 31*, 4 (July), 47:1–47:9.

BÄCHER, M., WHITING, E., SORKINE-HORNUNG, O., AND BICKEL, B. 2014. Spin-it: Optimizing moment of inertia for spinnable objects. *ACM Trans. Graph. 33*, (to appear).

BERGER, B., AND LEIGHTON, T. 1998. Protein folding in the hydrophobic-hydrophilic (HP) model is NP-complete. *Journal of Computational Biology 5*, 1, 27–40.

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 Trans. Graph. 31*, 6 (Nov.), 130:1–130:8.

CHANG, H.-H., LAI, Y.-C., YAO, C.-Y., HUA, K.-L., NIU, Y., AND LIU, F. 2013. Geometry-shader-based real-time voxelization and applications. *The Visual Computer* (July), 1–14.

CHEN, X., GOLOVINSKIY, A., AND FUNKHOUSER, T. 2009. A benchmark for 3D mesh segmentation. *ACM Trans. Graph. 28*, 3 (July), 73:1–73:12.

CHEN, D., SITTHI-AMORN, P., LAN, J. T., AND MATUSIK, W. 2013. Computing and fabricating multiplanar models. *Computer Graphics Forum 32*, 2pt3, 305–315.

DEMAINE, E. D., AND O'ROURKE, J. 2007. *Geometric Folding Algorithms: Linkages, Origami, Polyhedra*. Cambridge Univ. Press, New York, NY.

HILDEBRAND, K., BICKEL, B., AND ALEXA, M. 2012. crdbrd: Shape fabrication by sliding planar slices. *Computer Graphics Forum 31*, 2pt3, 583–592.

ISTRAIL, S., AND LAM, F. 2009. Combinatorial algorithms for protein folding in lattice models: A survey of mathematical results. *Communications in Information and Systems 9*, 4, 303.

KILIAN, M., FLÖRY, S., CHEN, Z., MITRA, N. J., SHEFFER, A., AND POTTMANN, H. 2008. Curved folding. *ACM Trans. Graph. 27*, 3 (Aug.), 75:1–75:9.

KIRKPATRICK, S., JR., D. G., AND VECCHI, M. P. 1983. Optimization by simmulated annealing. *Science 220*, 4598, 671–680.

LAU, M., OHGAWARA, A., MITANI, J., AND IGARASHI, T. 2011. Converting 3D furniture models to fabricatable parts and connectors. *ACM Trans. Graph. 30*, 4 (July), 85:1–85:6.

LAZARUS, F., AND VERROUST, A. 1998. 3D metamorphosis: a survey. *The Visual Computer 14*, 8–9.

LI, X.-Y., SHEN, C.-H., HUANG, S.-S., JU, T., AND HU, S.-M. 2010. Popup: Automatic paper architectures from 3D models. *ACM Trans. Graph. 29*, 4 (July), 111:1–111:9.

LI, X.-Y., JU, T., GU, Y., AND HU, S.-M. 2011. A geometric study of v-style pop-ups: Theories and algorithms. *ACM Trans. Graph. 30*, 4 (July), 98:1–98:10.

LI, H., ALHASHIM, I., ZHANG, H., SHAMIR, A., AND COHEN-OR, D. 2012. Stackabilization. *ACM Trans. Graph. 31*, 6 (Nov.), 158:1–158:9.

LO, K.-Y., FU, C.-W., AND LI, H. 2009. 3D polyomino puzzle. *ACM Trans. Graph. 28*, 5 (Dec.), 157:1–157:8.

LOOP, C., ZHANG, C., AND ZHANG, Z. 2013. Real-time high-resolution sparse voxelization with application to image-based modeling. In *Proc. ACM SIGGRAPH Symposium on High Performance Graphics*, 73–79.

LOWERE, B. 1976. The HARPY speech recognition system. *Ph.D. Thesis, Carnegie Mellon University*.

LUO, L., BARAN, I., RUSINKIEWICZ, S., AND MATUSIK, W. 2012. Chopper: Partitioning models into 3D-printable parts. *ACM Trans. Graph. 31*, 6 (Nov.), 129:1–129:9.

MCCRAE, J., SINGH, K., AND MITRA, N. J. 2011. Slices: A shape-proxy based on planar sections. *ACM Trans. Graph. 30*, 6 (Dec.), 168:1–168:12.

MITANI, J., AND SUZUKI, H. 2004. Making papercraft toys from meshes using strip-based approximate unfolding. *ACM Trans. Graph. 23*, 3 (Aug.), 259–263.

O'ROURKE, J. 2011. *How to Fold It: The Mathematics of Linkages, Origami and Polyhedra*. Cambridge Univ. Press, New York, NY.

PANTALEONI, J. 2011. Voxelpipe: A programmable pipeline for 3D voxelization. In *Proc. ACM SIGGRAPH Symposium on High Performance Graphics*, 99–106.

PRÉVOST, R., WHITING, E., LEFEBVRE, S., AND SORKINE-HORNUNG, O. 2013. Make it stand: Balancing shapes for 3D fabrication. *ACM Trans. Graph. 32*, 4 (July), 81:1–81:10.

SCHWARTZBURG, Y., AND PAULY, M. 2013. Fabrication-aware design with intersecting planar pieces. *Computer Graphics Forum 32*, 2pt3, 317–326.

SÉQUIN, C. H. 2012. Interactive 3D rapid-prototyping models. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 210–210.

SHAMIR, A. 2008. A survey on mesh segmentation techniques. *Computer Graphics Forum 27*, 6, 1539–1556.

SONG, P., FU, C.-W., AND COHEN-OR, D. 2012. Recursive interlocking puzzles. *ACM Trans. Graph. 31*, 6 (Nov.), 128:1–128:10.

TACHI, T. 2010. Origamizing polyhedral surfaces. *IEEE Transactions on Visualization and Computer Graphics 16*, 2, 298–311.

TRANSFORMERS, 2007. `http://transformersmovie.com`.

VARADHAN, G., KRISHNAN, S., KIM, Y. J., DIGGAVI, S., AND MANOCHA, D. 2003. Efficient max-norm distance computation and reliable voxelization. In *Proc. 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, 116–126.

WEEKS, D. 2013. Cubebots. `http://tweekstudio.com`.

WOLBERG, G. 1998. Image morphing: a survey. *The Visual Computer 14*, 8, 360–372.

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 Trans. Graph. 30*, 4 (Aug.), 97:1–97:8.

ZHOU, Y., AND WANG, R. 2012. An algorithm for creating geometric dissection puzzles. In *Proceedings of Bridges 2012: Mathematics, Music, Art, Architecture, Culture*, 49–56.