Fold and Fit: Space Conserving Shape Editing

Mohamed Ibrahim, Dong-Ming Yan

Abstract

We present a framework that reduces the physical space occupied by a man-made object when it is not in use. That is, given a segmented 3D mesh of a man-made object, our framework jointly optimizes for joint locations, folding order, and folding angles for each part of the model, enabling it to transform into a more space efficient configuration while keeping its original functionality as intact as possible. That is, if a model is supposed to withstand several forces in its initial state to serve its functionality, our framework places the joints between the parts of the model such that the model can withstand forces with magnitudes that are comparable to the magnitudes applied on the unedited model. Furthermore, if the folded shape is not compact, our framework proposes further segmentation of the model to improve its compactness in its folded state.

Keywords: Folding, Shape Editing, Segmentation, Functional Feasibility.

1. Introduction

Real-estate in major cities like London, New York, and Hong Kong is extremely expensive. Therefore, many new developments are concerned with building apartments with a small area. Efficient space usage in these apartments is critical and it is important that the available space can serve multiple functions. We are therefore interested to compute alternative configurations of man-made objects that use less space. We introduce a system that adds joints to pieces of furniture so they could be folded into a more space-efficient form, occupying less space when stored or transported while partially retaining their original functionality. Furthermore, if the folded furniture piece is not compact, our system proposes further segmentation steps of the model so that it folds into a compact form.

Over the past few years there were several research projects that focused on joint-aware shape processing. Examples include the works of Xu et al. [2] in shape editing, Mitra et al. [3] in motion illustration for mechanical assemblies, Luo et al. [4] in segmentation for 3D printing, Bächer et al. [5] in fabrication, Zheng et al. [6] in shape synthesis and the works of Li et al. [7] and Zhou et al. [8] in space saving.

The most closely related work to ours was recently published by Zhou et al. [8]. They *boxelize* an input 3D shape by first voxelizing the shape, then using a beam search method, they solve for the voxels that should be connected and for the types of joints between them that would enable the model to transform into a target 3D box. Finally, they use a physics simulator to find the path that the initial shape takes to reach the final folded form.

Our work differs from that of Zhou et al. [8] in two main aspects. First, we formulate the folding problem as a combinatorial optimization problem that optimizes for the folding order, folding angle and folding axis of each part of the model while they optimize for the joint type and folding angle and then, use a physics simulator to generate a valid folding path from the initial configuration of the model to its folded configuration. In addition, our framework places the joints between the parts such that even after the joints are placed, the edited model retains its functionality up to a user-defined value, while Zhou et al. [8] perform no functionality analysis on the edited models. Figure 1 shows an example of a model folded using our system.

Recently, Li et al [9] independently proposed a solution for the furniture folding process. While their method is faster than ours, we have a more extensive treatment of functional feasibility in our algorithm.

2. Related Work

Motion Analysis & Synthesis. Motion analysis and synthesis have been active areas of research in different fields for many years. For instance, in the field of *robotics*, the planning of robot motion from a start point to an end point has been heavily studied. For a good summary of motion planning algorithms, we refer the reader to the work of Steven et al. [10]. Other examples of research projects concerned with motion analysis include the works of Driskill et al[11], Agrawala et al.[12] and Lambert et al.[13] that focus on assembly design and visualization. Moreover, in the field of computer graphics, Mitra et

Figure 1: Furniture folding example: The top row is an example of a real life *bunk bed* that transforms to a space conserving form of a couch obtained from [1]. The bottom row shows our system's attempt to fold a 3D model of the same bed. Our system assigns the joints and folds the bunk bed to a configuration that is very similar to that of the real life example. Selected stages that our system takes to fold the model are shown in a), b) and c) respectively. The reader is referred to Figure 17 for more details regarding the folding of this model.

al. [3] introduce a novel system that illustrates the motion of mechanical assemblies by analyzing the geometric properties of each individual part and the joint-types between each adjacent pair of parts. Zhu et al. [14] built on this work by synthesizing the motion system from the user specifications, which is the reverse procedure of Mitra et al. [3]. Another extention of the work of Mitra et al. [3] is the work of Guo et al. [15] as they demonstrate the disassembly sequences of assemblies based on shape analysis. Finally, in a more recent work, Shao et al.[16] interpret the functionality of 3D shapes by analyzing corresponding 2D sketches, and they introduce an interactive system for this purpose.

Joint-Aware Editing & Fabrication. There are multiple recent papers that deal with joint-aware editing and fabrication. Xu et al. [2] extract the joint structures from 3D input shapes, and deform the 3D shapes based on the extracted joints. Moreover, Bächer et al. [5] aim at fabricating articulated deformable models for computer animation. A skinned mesh is taken as input and the joints are extracted and analyzed to satisfy the specific kinematics for fabrication. Furthermore, Ceylan et al. [17] approximate the motion of 3D automata by designing a kinematic system.

Given the 3D model of a man-made object, Lau et al. [18] define a formal grammar to automatically generate the parts and connectors needed to physically fabricate the model. To print large 3D objects, Luo et al. [4] propose *chopper*, a method that segments an object into printable size pieces, and joints are added to connect the pieces to each other. The robustness of the assembled objects are verified by physical simulations. Xin et al. [19] and Song et al. [20] study the design of 3D puzzles for fabrication. Moreover, Coros et al. [21] developed an easy to use framework that enables non-expert users, through sketching motion curves for the different parts of the model, to create animated mechanical characters. Finally, Cali et al. [22] intuitively add printable joints to 3D meshes such that, when printed, the model would exhibit functional articulation without the need for manual intervention. That is, the parts of the model would be able to move to satisfy the function that the model was intended to perform.

Structure-Aware Shape Editing. Several approaches have been recently presented in the field of structure-aware shape editing. For instance, Gal et al. [23] describe man-made shapes using 1D wires and their mutual relations. They keep these relations consistent while deforming the shape. Moreover, Wang et al. [24] analyze the hierarchical symmetry structure of shapes.

Zheng et al. [25] propose a framework to edit a 3D object while preserving the structure between its components. Recently, Li et al. [7] derive a stackability measure of a pair of objects as the gap between the lower and the upper envelopes of the objects being stacked. Then, they deform the models to reach a certain stackability measure and thus can be stacked tighter. Furthermore, Zheng et al. [6] generate variations of shapes from a set of input parts based on a functionality analysis system. Finally, for more details, we refer the readers to the most recent

survey by Mitra et al. [26].

3. Overview

Given a segmented 3D mesh of a man-made object, locations and directions of forces that it should withstand, our system goes through several steps to transform it to a plausible, space saving form that retains its functionality up to a user-defined threshold after editing. In what follows, we briefly explain each step.

Initialization. In this step, our system processes the model to decide which of its parts are symmetric and which parts should be connected together. Consequently, the algorithm places joints between each pair of parts that is connected. These joints are then utilized to generate a relational graph between the parts of the model and decide on the axes about which the parts are allowed to rotate. In addition, for each given direction and location of a user specified force, our framework analyzes the model to determine the maximum magnitude of the forces that the model can handle without collapsing. The initialization step performed by our system is explained in detail in Section 4.1.

Folding. In order to fold a given model, our system jointly solves for the order in which the parts should be folded, the axes about which each part should be folded and the angle by which each part should be folded while keeping the functional feasibility of the model up to a user-defined threshold. Using the order, the axes, and the angles generated in this step, our system is able to produce a collision free folding trajectory for each part of the model. The order optimization and the folding mechanism are explained in detail in Sections 4.2.2 and 4.2.3, respectively.

Segmentation. After the model is folded, the system checks the compactness of the folded configuration, and based on that, it decides whether or not further segmentation of the model is needed. If parts should be further segmented, the algorithm determines a location and a direction in which the parts should be split. The system then proceeds to fold the model with the new segmentation. This process is outlined in detail in Section 4.2.4.

4. Methodology

In this section, we describe the procedure followed by our system to fold a given segmented shape. First, we discuss the required initialization steps. Then, we list the steps taken by our system to fold the shape.

4.1. Initialization

Segmentation. Our system takes as input a segmented mesh, where the segments form a set of parts, P, that one would find in a place such as Ikea; a leg, a table top, etc. That is, we rely on semantic segmentation, i.e., each segment is an actual part that one would expect to be whole. Models with such segmentation can be obtained from many online 3D mesh repositories. For the models used in this paper, we obtain them from *Google 3DWarehouse*, *GrabCAD*, or we design them manually using *SketchUp*. An example of a semantically segmented shape that we obtained from *Google 3DWarehouse* is the pergola example shown in Figure 2.

Convex Decomposition. Most collision detection libraries operate faster when testing collision against a number of convex shapes. Therefore, we decompose each part of the model into a set of convex parts that are then used when we test for collision with other parts. We used the HACD technique introduced by Mamou et al. [27] for convex decomposition.

Symmetry Groups. Our system utilizes symmetry groups in the initialization, the folding and the segmentation steps. Symmetry groups are groups of parts that are reflectively symmetric about some plane. For each part, we search for another part that is reflectively symmetric to it about some plane, and group both parts in a symmetry group. To check whether two parts, P_i and P_j, are reflectively symmetric, we first obtain their barycenters $\mathbf{c}_i$ and $\mathbf{c}_j$ respectively. Our symmetry plane is defined by a normal $\mathbf{n} = \frac{\mathbf{c}_j - \mathbf{c}_i}{\|\mathbf{c}_j - \mathbf{c}_i\|}$ and a point on the plane $\mathbf{p} = \mathbf{c}_i + 0.5 \cdot (\mathbf{c}_j - \mathbf{c}_i)$. Two parts P_i and P_j are symmetric about this plane if for each point $\mathbf{p}_i$ on P_i, there exists a point $\mathbf{p}_j$ on P_j such that $\mathbf{p}_j = \mathbf{p}_i + \|\mathbf{p}_j - \mathbf{p}_i\| \cdot \mathbf{n}$. An example of the result of this step on the *Pergola* model is shown in Figure 2.

Joints Assignment. Once a segmented mesh has been loaded by the user, our system determines the joints to be placed on each part. Our system first determines contact regions between pairs of segments, then it places joints at these regions. Since the input model is segmented semantically, a pair of parts are in contact at a polygon. Therefore, to determine the contact regions between a pair of parts the algorithm first finds the polygons that make up each part, then it checks whether polygons from each part are in contact. An example of joints assignment between a pair of parts is shown in Figure 3.

- **Composing Polygons.** To obtain the composing polygons of a part, we group all the neighboring faces of the part that have a common normal into a structure that we call a *Composing Polygon (CP)*. We then obtain a 2D Oriented Bounding Box (OBB) of the vertices of the faces that make up the CP. Each *CP* is made up of the four vertices that make up the OBB and the common normal of the group of faces. Finally, with each *CP* we associate the area of its OBB to be used in the connection point assignment.

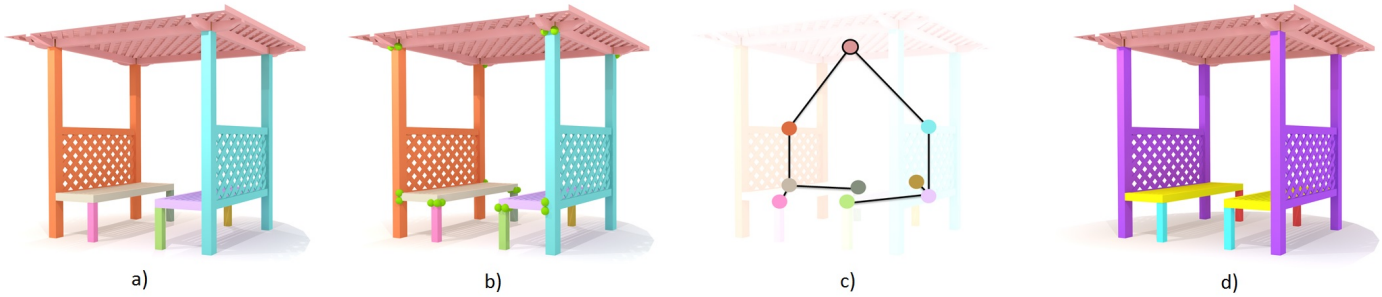

Figure 2: Initialization: Selected initialization steps performed by our system, a) input: a segmented mesh of the *Pergola* example, b) joints assigned by our system are displayed as green spheres, c) our system picks the topmost part to be the root node, r, enclosed in a black circle, of the relational graph and generates the relational graph using r and the joint information obtained in b, d) our system detects reflectively symmetric parts and adds them to symmetry groups to be used later in the folding and the segmentation steps, parts that have the same color belong to the same symmetry group.

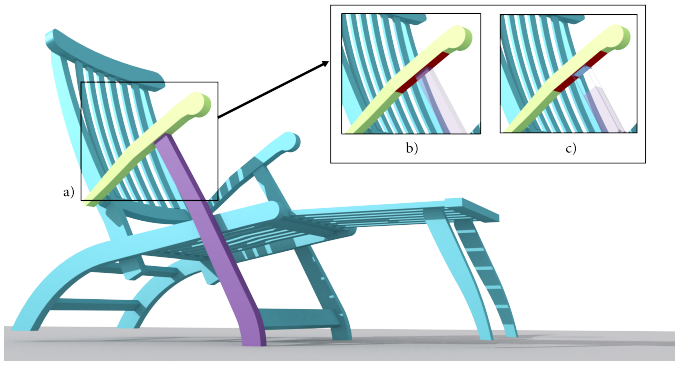

Figure 3: Joints Assignment: a) our system investigates the possibility of placing joints between the green and the purple parts of the teak chair model. b) a composing polygon of the green part is marked in red. c) a composing polygon of the purple part is marked in blue, both CPs are co-planar, and the CP of the purple part lies inside the CP of the green part. Therefore, we assign joints between both parts with their locations being that of the vertices of the OBB of the CP of the purple part.

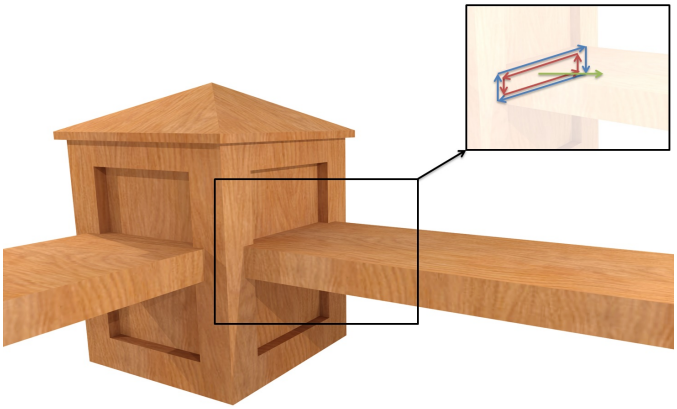

Figure 4: Axes assignment example: For this model, we show the nine possible axes, marked in blue, red and green, about which the seat can rotate with a positive angle. The seat can only rotate about four of the possible eight axes (marked in blue and red) without colliding with the parent part. Therefore, as a form of computation reduction, our algorithm enables the rotation of the part only about four of these eight axes in addition to the axis perpendicular to them, marked in green.

- **Contact.** To determine whether or not two parts, P_a and P_b, are in contact, we search for CPs of P_b that have a normal opposite to CPs of P_a. For each pair of polygons that have opposite normals, we find the CP with the smaller area and check if this polygon is on and inside the bigger CP or not. If so, we take the four points that make up the smaller polygon and add them as connection points for both parts. Else, if the smaller CP is on, but not inside the bigger CP, then P_a and P_b are in contact with their connection points being the 2D OBB of the common region between both CPs.

In few cases, further manual intervention might be required from the user's side to adjust the locations of the joints. For instance, if the model is not clean, that is the parts are missing some triangles, or the parts are not in a co-planar contact or the parts are intersecting, the user intervenes to either fix the model or adjust the joint locations manually.

Axes Assignment. Once the joints for each part have been generated, each part will have a multiple of four joints connecting it to other parts in the model. For each set of four joints, the part can rotate about nine different axes. Eight of these axes are the vectors between each consecutive pair of joints. That is two possible axes between each consecutive pair of joints, with these axes being parallel and opposite. A part can only rotate about four of the possible eight axes without causing collisions. Our algorithm rotates the part about the eight possible axes to pick the four axes about which the part can rotate without causing collisions. The last possible axis of rotation is the normal of the composing polygon. A demonstration of the possible axes of rotation is shown in Figure 4.

Relational Graph. Our framework employs a hierarchical transformation of the parts of the model to fold the model. It relies on a relational graph of the model where the nodes represent the parts and the edges represent the joints between them. The structure of the graph determines the parent-child relationships between the parts of the model needed to hierarchically transform a part.

Our system first picks a part of the model to be the root of the graph. Then, it generates a relational graph of the model using the connectivity information obtained from the joint generation step.

- **Root Picking.** Our system utilizes the symmetry groups obtained earlier to pick the root node, r, of the relational graph. That is, it searches for parts that do not belong to a symmetry group and nominates them as potential candidates to be the root of the graph. Amongst these parts, it searches for the part with the largest OBB volume and chooses it to be the root of the graph. If multiple parts have the same, largest, OBB volume, then we pick a part amongst these parts at random to be the root.

- **Graph Generation.** We use a *Breadth First Search (BFS)* method that starts from the root node and utilizes the connectivity information obtained earlier to generate the relational graph. The reason we chose BFS over other search methods, such as Depth First Search (DFS), is that unlike DFS, BFS correctly captures the parent/child relationships as they appear in the model. For example, for a model such as the shelf in Figure 10, the root node would be one of the sides, if we follow a DFS search for example, that would cause only one of the planks connected to the root to be the child of the root, while the root becomes the child of the other plank. However, if we use a BFS method, we will have both planks connected to the root as its children. We favor the result obtained by the BFS method as our algorithm attempts to fold a child node on its parent as described in Section 4.2.3, which will result in folding the planks on the side, instead of folding one plank on the side and the side on the other plank, which will result in a worse solution compared the one obtained by folding both planks on the side.

Maximum Force Magnitudes. Our framework analyzes the model to calculate the maximum magnitude of the forces that the model can handle without collapsing. That is, for each user-defined force's direction and location, we search for the maximum magnitude that will not cause it to collapse. We encode the collapsing of a model as the rotation of one of its parts by some angle that is above some user set rotation threshold ϵ_r. Ideally, we do not want the model to move at all,

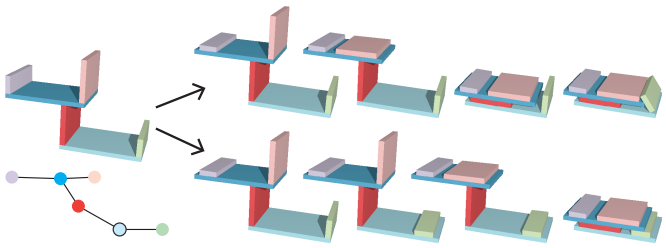

Figure 5: Folding Order: A small shelf could be folded in different ways. With the root node being enclosed in a black circle, the top folding uses a different folding order compared to the bottom folding. Moreover, the bottom folding order produces a more plausible folded form for this model.

however, since there is an error between the simulation of applying forces and physically applying the forces, we introduce this threshold. For all of our experiments, we set $\epsilon_r = 3°$. This maximum magnitude will be utilized in Section 4.2.3 to compare the maximum force magnitudes that the initial model and the edited model can withstand without collapsing. We use the *Bullet Collision Detection and Physics SDK* [28] to simulate the effect of applying forces on the model.

We refer the reader to Figure 2, for a demonstration of semantically segmented shape, and for the *Symmetry Groups*, *Joints Assignment*, and the *Relational Graph* generation steps performed by our system.

4.2. Folding Algorithm

We interpret folding a shape as minimizing the difference between the folded shape and its convex hull. That is, at its folded state, the shape should be as convex as possible. The folding problem is a combinatorial optimization problem. That is, in order to transform the input model to a space conserving configuration, our algorithm should solve three queries.

- What order should be followed to fold the parts of the model?
- How to fold a part?
- Would further splitting of some parts result in a folded model that consumes less space compared to the original folded model?

Throughout this text, we refer to these three problems as the *Order*, *Folding*, and *Segmentation* problems. In what follows, we present our solutions to these queries.

4.2.1. Algorithm Overview

Our algorithm solves the three problems above as demonstrated in Algorithm 1. First, it solves simultaneously for the *Order* and the *Folding* problems in the *Fold* function. After a solution for these problems is obtained, the algorithm calls the *ProposeSegmentation* function to

check whether it is worthwhile to solve the *Segmentation* problem or not. If so, it solves the *Segmentation* problem by calling the *Segment* function, then solves again for the *Order* and the *Folding* problems for the new parts resulting from the *Segmentation* step.

Algorithm 1 Algorithm Overview

```
1: procedure FoldAndFit(Model)
2:     repeat
3:         InitializeModel(Model)
4:         FoldedModel ← Fold(Model)
5:         if ProposeSegmentation(FoldedModel) then
6:             Model ← Segment(FoldedModel)
7:             Refold ← True
8:         else
9:             Refold ← False
10:    until !Refold
```

4.2.2. Order Optimization

The folding order of the parts of the model is of significant importance when it comes to folding a model. That is, for many models, deciding to fold part P_a first then part P_b can have a significant impact on the space consumed by the final, folded shape. A simple example of this is shown in Figure 5. An optimal folding order is an order that would result in an optimal folding, a folding that transforms the model to a plausible, space conserving form. Our task for this step of the algorithm, is to find such an order.

Folding Order. We define a folding order, O, as a permutation of the parts of the model. For example, for a model with four parts, and the root node being the first part, P_1, a possible order could be, $O = [P_3, P_2, P_4]$, meaning that P_3 should be folded first, then P_2, then P_4. The root node, r, is not taken into consideration in the folding order since it remains fixed throughout the folding process. Therefore, for a model with n parts, the number of possible orders is $(n-1)!$. Therefore, to find a folding order that would lead to an optimal folding, one could use a brute-force algorithm to search over the $(n-1)!$ possible folding orders, which would be costly in terms of speed for large values of n. Instead, we used a Genetic Algorithm (GA) based search to find an optimal folding order.

The population used by the GA is a population of possible orders, permutations, of the parts of the model. Each order O is a sequence of subsets, $S_i \subset P$. Our first attempt was to limit S_i to only contain a single part of P. However, experimenting with different models lead us to the conclusion that for some models, a plausible, folded output can only be obtained if we allow multiple parts to fold simultaneously. An example of such case is demonstrated in Figure 8. Therefore, we allow each S_i to contain one or two parts. Throughout our trials we found that if S_i is to contain two parts, it is better if the two parts are in a child-parent relationship or belong to the same symmetry group.

Genetic Algorithm. To fold a given model, the algorithm proceeds as demonstrated in Algorithm 2. Below is a description of the Algorithm.

- **Parameters.** The user of our system sets some initial parameters to be used by the GA. The user sets ps, the size of the population of orders $\mathbf{O}$. In addition, the user sets n_e, n_c and n_m, which set the size of the *Elite Population*, the number of pairs to be chosen for the *Crossover* operation, and the number of orders to be considered for the *Mutation* operation respectively. Finally the user sets the term sat, which states the percentage of the population that should be examined to check for the saturation of the population. We refer the reader to Table 1 for the values set for these parameters for all of our results.

- **Initialization.** The algorithm starts by initializing a population of orders $\mathbf{O} = \{O_1, ..., O_{ps}\}$. This is done by first generating a population of random orders of size $10ps$ each of which is composed of either subsets of cardinality one, subsets of cardinality two or a mix of both. For each subset of cardinality two, the two elements of the subset must have a parent-child relationship or belong to the same symmetry group. From all of these orders, a random set of orders of size ps is selected to be the initial population pop.

- **Evaluation.** As mentioned earlier, we interpret folding an object as transforming it to a configuration that is as close as possible to its convex hull. Therefore, the algorithm proceeds to evaluate each order of the population using the fitness function expressed by Equation 1.

$$E = V(ch(P')) \tag{1}$$

  where V is a function that calculates the volumes, ch is the convex hull function and P' is the set of all parts in the model after the model was folded by the order being evaluated.

- **Elite Population.** After the initialization and the evaluation steps, the algorithm proceeds to sorting the population according to the fitness function described in Equation 1 and it extracts the best n_e elements of the population to be considered in the new population.

- **Crossover and Mutation.** The algorithm then calls the crossover function to generate new elements from existing elements in the population. First it selects a pair of elements using a *roulette-wheel* selection method. Then it selects an index of the order at random after which it swaps the remainder of both orders to generate a new pair of orders. The algorithm further processes

the order to ensure it is valid. This operation is repeated for a user specified number of times, n_c. After the crossover stage, the algorithm proceeds to *mutate* a user specified number of elements of the population n_m. Order mutation is done by interchanging the locations of two subsets of the order.

- **Saturation Check and Termination.** After new elements are generated via the crossover and the mutation stages, a new population is formed by combining the elite population and the crossover, mutated population. This population is checked against the old population to check whether or not the algorithm reached a saturation stage where the new population does not vary significantly from the original population. That is, the algorithm checks a percentage, *sat*, of the current and the old populations, and compares them against each other. If both percentages in the old and the current populations remain equal for more than three iterations, then a saturation stage is reached. Once a saturation stage is reached, the algorithm terminates, picks the best folding order, folds the model according to that order and returns the folded model.

Algorithm 2 Fold Model

```
1: function FOLD(Model)
2:     Saturated ← False
3:     Pop ← InitalizePopulation(ps)
4:     Pop ← EvaluatePopulation(Pop, Model)
5:     while !Saturated do
6:         EPop ← ExtractElitePopulation(Pop, n_e)
7:         CPop ← CrossOverPopulation(Pop, n_c)
8:         MPop ← MutatePopulation(CPop, n_m)
9:         MPop ← EvaluatePopulation(MPop, Model)
10:        NPop ← EPop + MPop
11:        Saturated ← IsSaturated(Pop, NPop)
12:        Pop ← NPop
13:    O ← GetBestOrder(Pop)
14:    FoldedM ← FoldInOrder(O, Model)
15:    return FoldedModel
16:
17: function EVALUATEPOPULATION(Pop)
18:    for i = 1 → i = size(Pop) do
19:        FoldedM ← FoldInOrder(Pop[i], Model)
20:        Pop[i].fitness ← EvaluateFitness(FoldedM)
    return Pop
```

4.2.3. Folding

After a folding order is specified, our algorithm folds the model according to this order. This step constitutes the function *FoldInOrder* mentioned in Algorithm 2. In what follows we describe how to fold each subset S_i of an order **O**.

Folding Energy. Folding a part amounts to bringing it as close as possible to another fixed part in the model so that the new folded configuration consumes less space compared to the original one. Therefore, folding the elements of S_i amounts to bringing them very close to some fixed subset of parts, S_{i_f}, in the model. That is, for each element of S_i, we need find an axis/angle combination such that the angle is the maximum, collision-free angle that would bring the elements of S_i to S_{i_f}. If $|S_i| = 1$, then $|S_{i_f}| = 1$, and will contain the parent of the element in S_i. However, if $|S_i| = 2$, and the elements in S_i are in a parent-child relationship then $|S_{i_f}| = 1$, and will contain the grandparent of the child node in S_i. However, if the elements in S_i belong to the same symmetry group, then $|S_{i_f}| = 2$, and will contain the two parents of the two elements. To optimally fold a subset S_i we must consider the effect that this folding will have on the model both locally and globally. That is, the folding of S_i should minimize the space occupied by S_i along with S_{i_f}. In addition, the folding must minimize the space occupied by the model as a whole.

We define the folding measure of folding a set of parts S_i as a weighted sum objective function that is a variant of the fitness function described in Equation 1. The folding energy is proportional to the volume of the convex hull containing the parts in S_i along with their fixed set, S_{i_f}, and the convex hull containing all the parts of the model as stated in Equation 2.

$$E_1 = w_g \cdot V(ch(P')) + w_l \cdot V(ch(S_i, S_{i_f})) \qquad (2)$$

where w_g and w_l are weights set by the user, for all of the results presented in this paper, we set $w_g = w_l = 0.5$.

Therefore, to fold the parts in S_i together, we solve for a set of transformations for the parts in S_i that minimizes the folding measure stated above. These transformations must also posses two other characteristics. First, they should not allow the parts in S_i to penetrate each other or other parts of the model. In addition, the path taken by the parts in S_i to their folded configurations must be continuous and collision free.

Transformation. Since we only allow rotational joints in our system, transforming a part, P_i amounts to rotating it about the center of the joint connecting it to its parent part, Par_{P_i}. To transform a part, we first analyze its position in the relational graph, and depending on the path that it lies on, we either transform it using a simple hierarchical approach or using a more complex motion constraint solvers.

- **Hierarchical Approach.** If a part, P_i does not belong to a cycle in the relational graph, then its motion will only affect its descendants in the graph and therefore, could be simply transformed using a simple Hierarchical Approach. We denote by t_i the transformation matrix

of P_i. This matrix is computed as shown in Equation 3.

$$t_i = \text{Trans}(cor_i) \cdot R_i \cdot \text{Trans}(-cor_i) \tag{3}$$

where $Trans(cor_i)$ is a translation matrix by the vector cor_i, which is the center of the joint connecting P_i to Par_{P_i}. Moreover, R_i is a rotation matrix that is equivalent to rotating P_i about the origin by a quaternion, $\mathbf{q_i}$. We follow the same conversion found explained by Akenine-Moller et al. [29] to obtain a rotation matrix R from a quaternion $\mathbf{q} = \begin{pmatrix} qx & qy & qz & qw \end{pmatrix}^t$.

Moreover, since we are using a hierarchical representation, the final transformation matrix, T_i of P_i is $T_i = T_{Par_{P_i}} \cdot t_i$. Where $T_{Par_{P_i}}$ is the total transformation of Par_{P_i}.

- **Motion Constraints Approach.** Moving a part, P_i that belongs to a cycle in the relational graph could result in moving nodes that are not in the descendants of P_i, and therefore, a simple hierarchical approach would not transform all the parts correctly. Instead, motion constraints must be set to correctly solve for the motion of all parts in the graph depending on the motion of P_i. Examples of such models are shown in Figures 10 and 13. We use the *Bullet Collision Detection and Physics SDK* [28] to solve for the motion of such parts.

Functional Feasibility Energy. A model that serves a certain function, should still be able to serve the same function after editing, that is after adding joints between its parts. We interpret the functionality of a part as the forces that it can withstand without collapsing. For instance, the functionality of a table, amounts to being able to hold objects of a range of weights on its top without collapsing. We encode this functionality as the maximum magnitude of a force applied to the table top as shown in Figure 6.

Adding joints between different parts of a model weakens the model. That is, it reduces the maximum magnitudes of the forces that the model can handle without collapsing. However, not all joint placements weaken the model equally. That is, for a joint j placed between two parts P_a and P_b, the joint location could dramatically affect the functional feasibility of the model. As shown in Figure 6, placing the joints on the outer sides of the legs makes the table unstable and therefore functionally infeasible, while placing the joints on the inner sides of the legs makes the table functionally feasible. Therefore, in addition to occupying less space, our framework aims to edit the model such that it can withstand forces that are equal in direction, location and are comparable, up to a user defined value, to the maximum magnitudes, calculated in the initialization step, that the initial model can bare without collapsing.

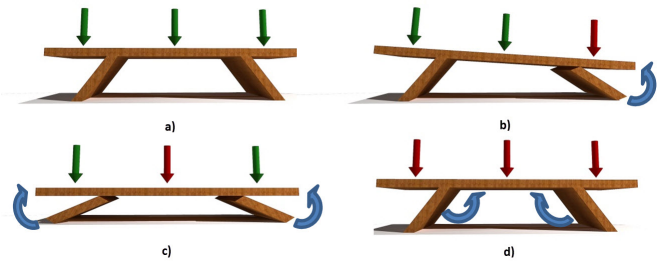

Figure 6: Functional Feasibility: a) Original table model with user specified force locations and directions, b) unstable case, after inserting one joint between the right leg and the tabletop on the outer side of the leg, applying the force in red, with the maximum magnitude obtained in the initialization step, makes the table unstable, c) unstable case, after inserting two joints on the outer sides of both legs, and applying the force in the middle, the table becomes unstable, d) stable case, adding both joints on the inner sides of the legs, makes the table stable, even when all forces are activated.

Users of our framework set the location, and the direction of the forces that they think define the functionality of the model. With this in mind, we define the functional feasibility energy for a force i as stated in Equation 4. Our framework will seek to minimize this energy.

$$E_{2,i} = \begin{cases} \infty, & \frac{\|mf_{c,i}\|_2}{\|mf_{o,i}\|_2} < \epsilon_f \\ 0, & \frac{\|mf_{c,i}\|_2}{\|mf_{o,i}\|_2} \geq \epsilon_f \end{cases} \tag{4}$$

where $\|mf_{o,i}\|_2$ is the maximum magnitude of force i that the model can withstand originally, $\|mf_{c,i}\|_2$ is the maximum magnitude of force i that the model can withstand at a certain configuration, i.e, a certain joint combination.

The aim of this energy term is to direct our algorithm to finding the folding order, folding angles and axes combinations that would produce the most compact model such that it supports a minimum percentage of the maximum forces that the original model can withstand. We define the functional feasibility energy for a force i as the ratio of the maximum force magnitude applied to the current configuration versus the original maximum magnitudes of force i applied to the original, fixed model. This directs the framework to favoring joint placements that would enable the model to withstand a force with a maximum magnitude that is greater than or equal to a percentage, ϵ_f, of the original maximum magnitude of force i. For Instance, if placing a rotational joint reduces the functional stability below ϵ_f, compared to placing a fixed joint, a fixed joint is favored by this energy function.

Combined Energy. Our framework seeks order, joint and angle combinations that would transform the model to a more space convenient configuration while maintaining a minimum functional feasibility for the edited model.

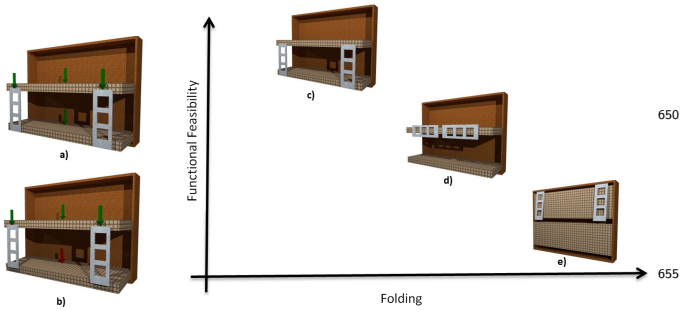

Figure 7: Competing Goals: a) The original bunk bed model with the locations and directions of the user supplied forces shown in green, b) bottom force is active, with magnitude equal to $\epsilon_f = 0.7$, the bottom force moves the bottom bed with an angle above ϵ_r, making the model unstable as it is not supported by anything other than the back of the bunk bed, at $\epsilon_r = 3°$, the same applies for the top bed, the outcome of the folding produced by our algorithm is shown in c) using $\epsilon_f = 1$, in d) $\epsilon_f = 0.7$ and in e) $\epsilon_f = 0.5$. Note that when ϵ_f is set to a high value in 'd', the framework placed joints between the top bed and the ladders, and fixed the top and the bottom beds to the back.

We formulate this goal as a summation of the folding energy, E_1, stated in Equation 2 and the functional feasibility energy for a force i, $E_{2,i}$, stated in Equation 4. This formulation is demonstrated in Equation 5.

$$E = E_1 + \sum_{i=1}^{n_f} E_{2,i} \tag{5}$$

where n_f is the number of user specified forces.

For some cases, as the one shown in Figure 7, E_1 and E_2 become competing goals, at this point it is left up to the user to favor foldability over functional feasibility or vice versa. Adjusting ϵ_f generates results that trace a set similar to a *Pareto Optimal Set*, at one end of the set foldability is preferred, at the other end, functional feasibility is preferred. An example of this set for the *bunk bed* model is shown in Figure 7.

Angle/Axis Optimization. Given a subset of parts S_i to fold, the algorithm optimizes for the angles and axes of the parts in S_i that would minimize the energy described in Equation 5. The objective here is to fold each element of the S_i as much as possible about its different rotation axes and pick the folding that best minimizes the objective function mentioned in Equation 5. An overview of this process is demonstrated by Algorithm 3. Even though the goal of finding the best angle/axis combination that minimizes Equation 5 does not change depending on whether the node at hand belongs to a cycle or not in the relational graph, the search method used is different depending on whether the node belongs to a cycle or not.

- **Node not Belonging to a Cycle.** For nodes that do not belong to a cycle, if the set S_i contains only one element, the algorithm searches for the maximum rotation angle about each of the possible rotation axes of the element at hand. This is done by iterating over all rotation axes and incrementally searching for the maximum angle by which the element can rotate without causing any collisions in the model. That is for each axis, we start by angle=0 and keep incrementing the angle by increments of 5° till we reach 180° or collide with another part, whichever comes first. For each (axis, angle) combination, the objective function is evaluated and the result is stored in a *scores* structure. After all rotation axes have been processed, the best *score* is chosen from the *scores* structure and its respective (axis, angle) combination is applied to the model to fold S_i.

  If S_i contains multiple elements, the algorithm searches for a combination of axes and angles (an axis and angle for each element) that would minimize the objective function. It does so by traversing recursively all possible axes combinations, and calculating their respective maximum angle by which each element can rotate. For all (axes, angles) combinations, the *scores* structure is maintained and updated as in the single element case. The best *score* is then picked from the *scores* structure and its respective (axes, angles) combinations are applied to fold the elements in S_i.

  When we come to folding multiple parts simultaneously, some parts might hinder other parts from folding by larger angles due to collision problems. Therefore, to solve this problem and ensure that the parts are folded to their maximum angles, we repeat the folding multiple times until the outcome of the folding does not change. An example of this case is shown in Figure 8.

- **Node Belonging to a Cycle.** Finally, if we come to searching for the folding axis of a part, P_i that belongs to a cycle in the relational graph, then, in addition to rotating about all the axes of P_i we must also assign axes to all the parts that belong to the cycle in which P_i falls. For example, in the folding bed model in Figure 13, the algorithm will pick either the part of the bed closer to where one would rest his/head, or the part closer to where would rest his/her feet as the root. This will lead to a graph that has a cycle, the sides of the bed and the head rest and feet rest will form this cycle. Therefore to test for the rotation of any of these parts about an axis, one must also assign axes to the remaining parts of the cycle so the part could physically move. Therefore, if the algorithm encounters a node that falls within such a cycle, instead of searching for the rotations over

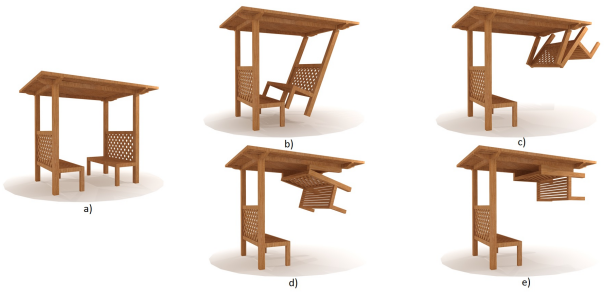

Figure 8: Folding pairs: a) Original model to be folded, b) and c) our algorithm's attempt to fold the seat-back and the planks above it on the right side of the model. In b), the algorithm folds the planks on the right, it stops to avoid colliding with the left part of the model. In c), the algorithm folds the seat-back on the right and stops to avoid colliding with the topmost part of the model. In d) the algorithm folds both the seat-back and the top planks on the right together and reaches a configuration similar to c), however, since we repeat the folding of pairs in e) we reach a plausible, folded configuration of these parts.

its axes, it incrementally searches for all the combinations of the axes of the parts that fall within the cycle.

For each axes combination, we virtually install *hinges* and *motors* supplied by the *Bullet Collision Detection and Physics SDK* [28] to move the nodes that belong to the cycle about their designated axes. We install *hinges* on all the nodes at their respective axes and set the hinge limits to be $[-\pi, \pi]$ to enable all the nodes in the cycle to move freely. We then install a *motor* on the part that we wish to move. The *motor* then moves the part about its designated axis and therefore moves the other parts in the cycles in a motion that is physically correct. The *motor* continues to move the part till a collision is encountered. At this point we get the angle the *motor* stopped at, and set that as the folding angle for P_i.

4.2.4. Segmentation

Our algorithm for solving the *Ordering* and the *Folding* problems is capable of folding a model to a plausible, space conserving configuration if the model is known to be foldable as in Figures 7, 10, 11, 13, and 14. However, if the model is not foldable by design, our algorithm will produce a configuration that is space conserving but not necessarily plausible. This gives rise to the question, can we slightly manipulate the shape so that it would consume less space and look more plausible at its folded state compared to the original folding? In what follows, we present our solution to this problem.

Possible operations to consider to manipulate a shape include scaling, smoothing, clipping and segmenting a mesh. For our proposes, we seek a shape manipulation

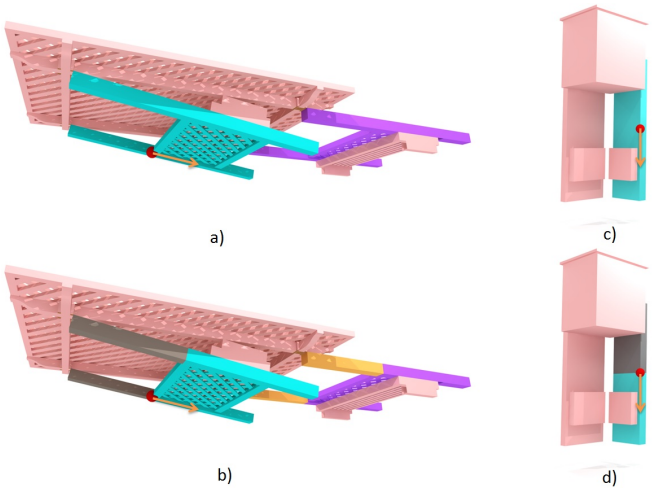

Figure 9: Segmentation: a) the blue part's *Folding Potential* is not maximized after folding due to the seat attached to it, if the blue part is to come any closer to the topmost part of the model, the seat will collide with the topmost part. The blue part is nominated for segmentation, with a segmentation location and direction shown in red and orange respectively. The purple part belongs to the same symmetry group as the blue part, therefore it is nominated for segmentation as well. b) the outcome of the segmentation step done on the blue and the purple parts in a), four segments, with the blue part and the purple part belonging to one symmetry group, and the grey and orange parts belonging to another symmetry group. c) the folding outcome of the *bench* model is of high concavity, and the part responsible the most for this high concavity is the blue part. Therefore the blue part is nominated for segmentation based on *concavity* with a segmentation location and direction shown in red and orange respectively. d) the outcome of the segmentation step done on the blue part in c).

operation that would not change the overall look of the shape, and would coincide with our goal of introducing joints that would make the model consume less space in its folded state. Therefore, we decided to manipulate a shape using segmentation, since it basically amounts to dividing a part into multiple parts and adding joints between them.

Throughout our attempts to fold shapes that are not foldable by design we came to the realization that there are two main features of these shapes that prevent them from having a plausible folded configuration. These features are *Folding Potential* and *Concavity*.

Folding Potential. Throughout our trials, we noted that the parts of a shape that is foldable by design tend to fold over their respective rotation axes by an angle that would bring them very close to their parents. This angle is ninety degrees for all of our test models. Therefore, after folding a shape, we check the folding angle of each part and nominate the part with the smallest folding angle for further segmentation.

Concavity. Since we define folding as bringing a shape to a configuration that is as convex as possible, a folded configuration that is concave is in direct opposition of our goal. Therefore, after folding a shape, we check each part to see if it is causing the folding to be concave. That is, for each part, we shoot a number of rays that is proportional to its volume from the surface of the part to the convex hull containing the folded shape. From these rays, we pick the rays that do not hit any other part before hitting the convex hull. We sum the lengths of these rays and store this sum as the concavity score of the part. Finally, we nominate the part with the largest concavity score for further segmentation.

Segmentation Algorithm. As mentioned earlier, we wish to *slightly* modify the shape so that it posses a plausible folded configuration. Therefore, we choose to only apply one type of segmentation at a time. That is, we check if the *Folding Potential* of a part is greater than zero, if so, the part is segmented and no check for *Concavity* is performed. Otherwise, the *Concavity* check is applied, and depending on the *Concavity* score, the algorithm decides whether to segment the model or not. This process is outlined in Algorithm 4.

After a part is nominated for segmentation, all that remains is to determine how we will segment the part. Moreover, we need to decide into how many parts will we divide the part, at what location, and in what direction.

- **Number.** At each splitting iteration, we split a part into two parts. We chose to split a part into two parts only as we wish to add as few parts as possible to our original model to reduce the complexity of the order generation and folding phases of our algorithm. In ad-

Algorithm 3 FoldInOrder

```
function FoldInOrder(O,Model)
    for i = 1 → i = size(O) do
        S = O[i]
        if S.length()==1 then
            NModel ← FoldSet(0, S, Model)
        else
            NModel ← Model
            repeat
                Model ← NModel
                NModel ← FoldSet(0, S, Model)
            until AreSimilar(NModel, Model)
function FoldSet(ind,S,Model)
    if ind > S.size() then
        Score = PickBestScore(Scores)
        FoldedModel =
  ApplyScoreData(Model, Score)
        return FoldedModel
    mind = S[ind]
    for i = 1 → i = Model[mind].axes.size() do
        a ← Model[mind].axes[i]
        θ ← PickMaxAngle(mind, Model, a)
        Model[mind].a ← a
        Model[mind].θ ← θ
        if ind == S.size() then
            Score = EvaluateFolding(Model)
            Scores.Push(Score)
        else
            return FoldSet(ind + 1, S, Model)
```

Algorithm 4 Segmentation

1: **function** SEGMENT($Model$)
2: $P = C = False$
3: $[P, NModel] = SegmentForPotential(Model)$
4: **if** $!P$ **then**
5: $[C, NModel] = SegmentForConvexity(Model)$
6: **if** P *or* C **then return** $NModel$
7: **else return** $Model$

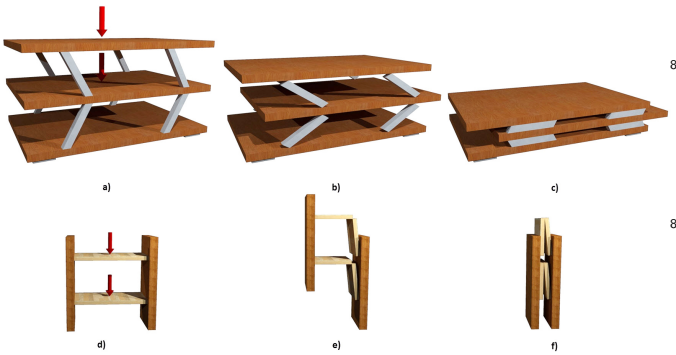

Figure 10: Folding of models with cycles: a) shows the Multi-layer table model with the user specified forces indicated in red, b) and c) shows selected iterations of how our algorithm folds the model at $\epsilon_f = 0$. d) shows the shelf model with the user specified forces indicated in red, e) and f) show selected iterations of how our algorithm folds the model at $\epsilon_f = 0.8$. The motion of the parts in these models were implemented using the 'Motion Constraints Approach' mentioned in Section 4.2.3

.

dition, the algorithm analyzes the concavity and folding potential after a model is folded and though would split the model further if the need arises.

- **Direction.** Splitting a part requires a splitting direction and a splitting point. The splitting direction should be a direction at which the object varies most so that the splitting would contribute to a folding result that is more compact since the the part is split into two parts along this direction. Therefore we run a *Principal Component Analysis* on the part to be split, and choose the splitting direction to be the component with the greatest variance to be the splitting direction. For some examples, it proved better to use the component with the second greatest variance. Therefore we run the algorithm with both choices and choose the result that is more compact. We refer the reader to Table 1 for the choice of PCA vectors used for each model we segmented.

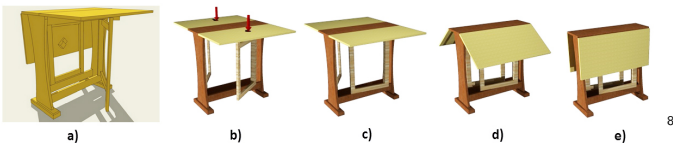

Figure 11: Folding of the table model: a) a picture of real life model that is half folded obtained from [30], b) through e) demonstrate selected iterations that our system takes to fold the table at $\epsilon_f = 0.9$.

- **Location.** We wanted to split a part at a plausible location, a location that splits the part into two semantically meaningful parts that one would find in a place like *Ikea*. Therefore, using the convex decomposition information obtained from the initialization step, we choose locations that are at the intersection of convex clusters to be candidates for the splitting location. We also wish to split the part into two parts that are somewhat similar in terms of size, splitting a part into parts that are not proportional is not desirable. Therefore, among the points present at the intersection of convex clusters, we look for the one closest to the center in the splitting direction obtained above. This point is expressed more formally in Equation 6.

$$\arg\min_{s_p} \|(s_p - c) \cdot d\|^2 \quad s.t.\ s_p \in ccp \tag{6}$$

where s_p is the location that we are looking for, c is the center of the part, d is the splitting direction obtained above and ccp is the set of points present at the intersection of convex clusters. If the part to be split is composed of only one convex cluster as in the seats in Figure 16, then we split at the center of the part.

- **Symmetry Group.** If the part to be split belongs to a symmetry group, then we split the part symmetric to it as well. The splitting point will be reflected over the symmetry plane and the splitting direction would remain the same. For each reflectively symmetric pair of the newly generated parts, a new symmetry group is generated and added to the model. An example of the segmentation step performed by our system based on *Folding Potential* and *Concavity* on the *Pergola* and the *bench* models respectively is demonstrated in Figure 9.

5. Results

We used our system to generate two categories of results. The first category demonstrates the ability of our system to fold models that are foldable by design in a manner that is similar to their folded state in real life. Examples in this category include a bunk bed as in Figure 7, a multi-layer table as in Figure 10, a foldable table as in Figure 11, a folding bed as in Figure 13, and a desk-crate as in Figure 14. The second category demonstrates the ability of our system to transform models that are not foldable by design to foldable models. Furthermore, it proposes segmentation steps that would make the original model

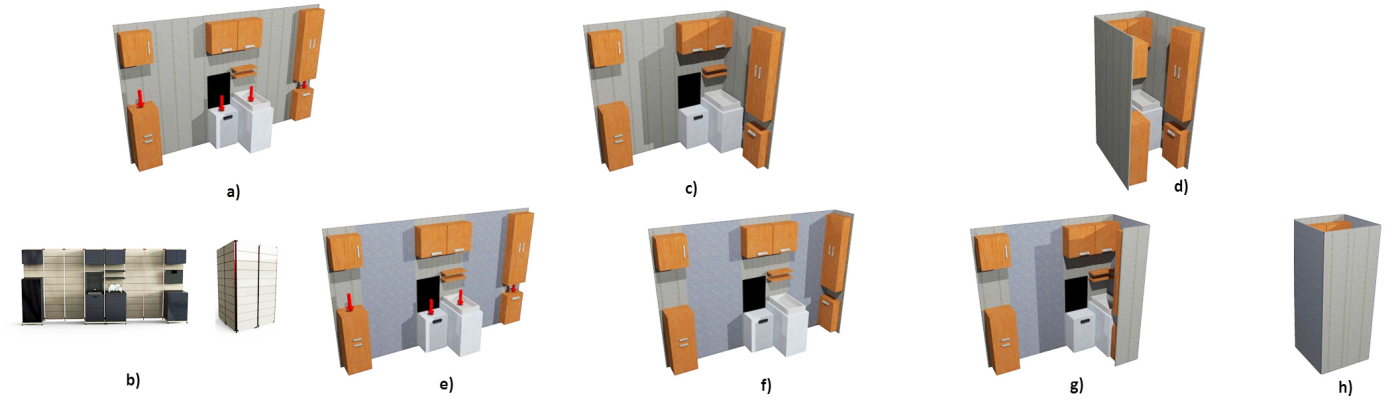

Figure 12: Folding of the kitchen model: a) kitchen model with three segments and the user specified forces indicated as red arrows, b) a real life picture of the model in its original and folded states obtained from [31], c) and d) demonstrate the folding iterations taken by our algorithm to fold the model, e) the result of the segmentation step explained in Section 4.2.4, applied on a), the model is segmented vertically, resulting into a model with five segments, the new segments are displayed using a different texture compared to the older ones. f) through h) demonstrate selected iterations that our algorithm takes to fold the newly segmented model at $\epsilon_f = 0.9$.

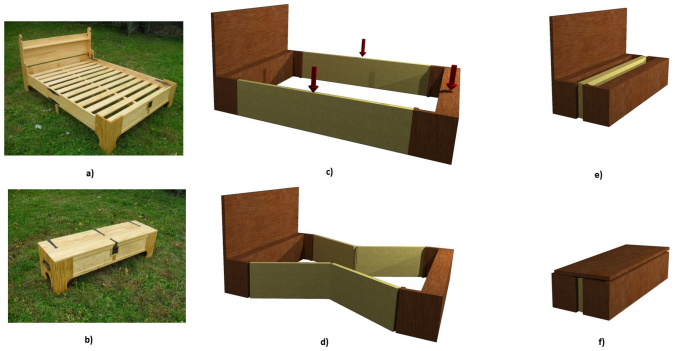

Figure 13: Folding of the bed model: a) and b) show pictures of the real life model in its original and folded states respectively obtained from [32], c) the original model with the user specified forces indicated by red arrows, d) through f) demonstrate selected iterations that our system takes to fold the model at $\epsilon_f = 0.8$.

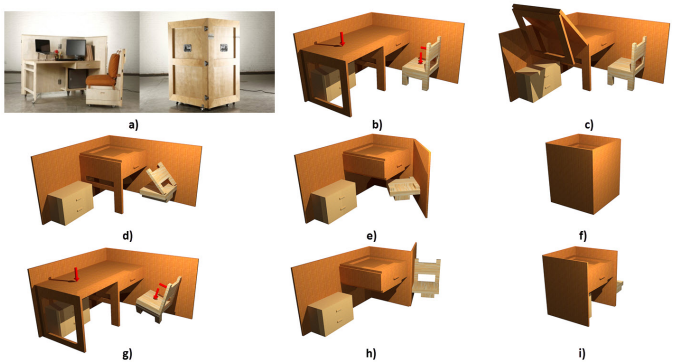

Figure 14: Folding of the desk-crate model: a) real life pictures of the model in its original and folded states obtained from [33], b) Original desk-crate model with the user specified forces indicated by the red arrows. , c) through f) demonstrate selected iterations that our algorithm takes to fold the model. g)the folded model in f) cannot withstand 60% of the maximum magnitudes handled by the original model as the chair begins to move, h) and i) show a folding that can handle 60% of maximum magnitudes of the original forces, that is, at $\epsilon_f = 0.6$. It installs fixed joints for the chair.

foldable. Examples in this category include a shelf as in Figure 10, a kitchen as in Figure 12, a pergola as in Figure 15, a bench as in Figure 16, and a sofa-bed as in Figure 17.

Our algorithm proposes segmentation that is similar to the segmentation of the models in real life. That is, given a segmented model, if we merge some of its segments, and pass the model to the algorithm to fold, the algorithm will recover the original segmentation and fold the model in a manner similar to the way it was folded in real life. This is demonstrated in the sofa-bed and the kitchen examples in Figures 12 and 17 respectively.

Table. 1 documents the figures recorded for each model using our system. We compute the energy of the part before and after folding as described in Equation 1. Furthermore, we compute the saved energy ratio (svr) as *1-(E'/E)*, where E is the energy of the shape at its initial state, and E' is the energy of the shape after it is folded. Our system saved more than 50% of the space occupied by the initial model for most of the models.

As for the animation of the results, since we are using quaternions to represent the rotation of each part, interpolating between the initial state of a part and its folded state was done using SLERP, a quaternion interpolation technique explained in the work of Shoemake et al. [35]. As for the animation of the parts that contain cycles within their respective relational graph, we use the *Bullet Collision Detection and Physics SDK* [28] to animate the folding for these models. Please refer to the supplemen-

Table 1: Results Summary: For each model we state its original energy, E, folded energy, E', saved energy ratio, ser, number of parts, p, number of connection points, c, total time taken for folding, segmentation and refolding, t, the type of segmentation applied to the model st, a *Folding Potential* segmentation is denoted as f and a *Concavity* based segmentation is denoted by c, the number of parts segmented by the algorithm, sn, which PCA vector was used as a segmentation direction pca, size of the population used to search for the folding order ps, the minimum stability ratio ϵ_f, the population saturation percentage, sat, and the number of forces applied to the model n_f. we list multiple results for models, where varying parameter ϵ_f gives interesting and meaningful variations.

model	ϵ_f	E	E'	ser(%)	p	c	t(mins)	st	sn	pca	ps	sat(%)	n_f
Bunk Bed	0	4.23	1.89	55.3	5	16	0.035	na	0	na	1	20	na
	0.5	4.23	1.89	55.3	5	16	2.48	na	0	na	2	20	4
	0.7	4.23	3.39	19.89	5	16	2.53	na	0	na	2	20	4
Table	0	3.86	1.47	61.9	5	16	0.39	na	0	na	7	20	na
	0.9	3.86	1.47	61.9	5	16	5.69	na	0	na	7	20	2
Sofa/Bed	0	6.46	3.73	42.2	6	20	0.54	c	1	2^{nd}	5	20	na
	0.5	6.46	5.34	17.34	6	20	10.94	c	1	2^{nd}	5	20	3
Bench	0	5.39	1.91	64.59	7	24	0.23	c	2	1^{st}	1	20	na
	0.5	5.39	2.60	51.77	7	24	9.61	c	2	1^{st}	4	20	2
Kitchen	0	8.37	4.92	41.22	5	16	0.9682	c	2	2^{nd}	4	20	na
	0.9	8.37	4.92	41.22	5	16	11.42	c	2	2^{nd}	4	20	5
Desk-Crate	0	2.60	0.98	62.10	10	36	10.18	na	na	na	12	40	na
	0.6	2.60	1.27	50.91	10	36	61.37	na	na	na	12	40	3
Bed	0.8	2.42	0.63	74.0	7	28	134.3	na	na	na	10	20	3
Multi-layer Table	0	6.68	2.46	63.15	11	64	95.84	na	na	na	10	20	na
Shelf	0.8	4.02	1.32	66.87	6	24	8.44	f	2	1^{st}	5	20	2
Pergola	0	6.46	3.73	42.25	6	24	8.11	f	2	1^{st}	10	20	na
	0.7	6.46	4.28	33.75	6	24	105.59	na	na	na	10	20	4

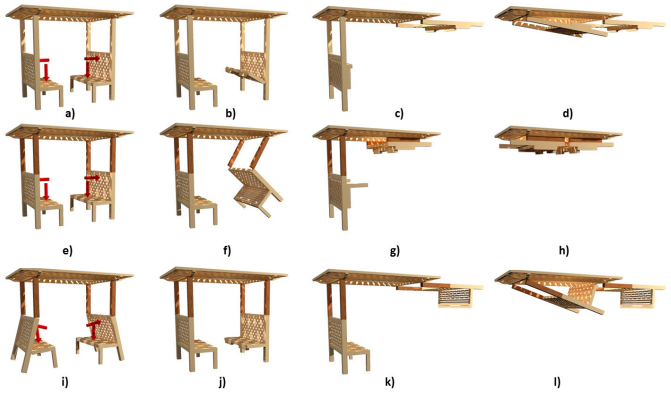

Figure 15: Folding of the pergola model: a) the original pergola model with nine segments, the user set forces indicated by red arrows,b) through d) show selected iterations that our algorithm takes to fold the model, e)the sides of the pergola are segmented into two segments, each with a different texture, resulting in a pergola model with eleven segments, f) through h) show selected iterations that our algorithm takes to fold the newly segmented model using $\epsilon_f = 0$, i) the model becomes unstable if a hinge is placed on the outer side of the sides, or on the top side of the seat. That is, applying a force to the side will result in the side moving, and as a result, the ground will cause the legs of the seat to push the seat upwards, and so the seat moves as well, instead our algorithm places fixed joints for both seats and both sides to fold the model with using $\epsilon_f = 0.7$. j) through l) show selected iterations are shown that fold the model.

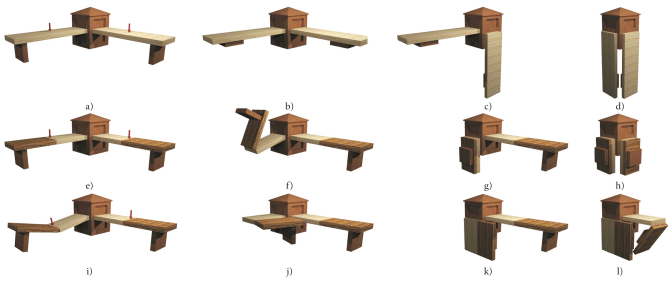

Figure 16: Folding of the bench model: a) the original bench model with five segments, the user set forces indicated by red arrows,b) through d) show selected iterations that our algorithm takes to fold the model, e)the seats of the bench are segmented into two segments, each with a different texture, resulting in a bench model with seven segments, f) through h) show selected iterations that our algorithm takes to fold the newly segmented model using $\epsilon_f = 0$, i) the model becomes unstable if a hinge is placed on the top side of the seat, applying a force on the seat will result in a collapsing seat, instead our algorithm places a joint on the side of one seat, and a joint on the bottom of the other seat to fold the model with using $\epsilon_f = 0.5$. j) through l) show selected iterations are shown that fold the model.

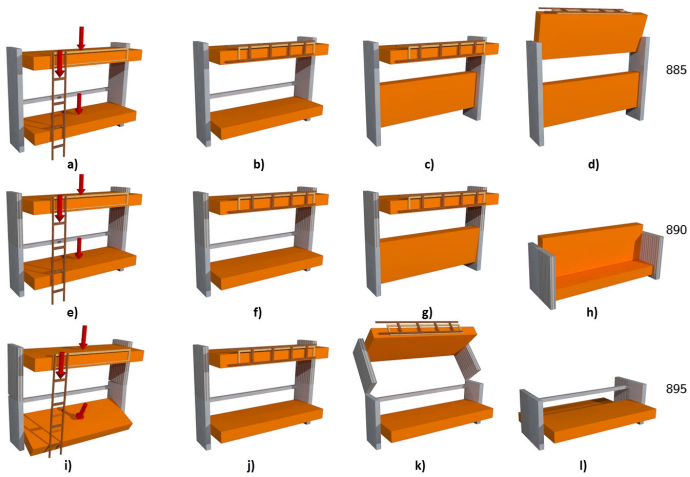

Figure 17: Folding of the sofa/bed model: a) the original bunk bed model with five segments and the user specified forces indicated by red arrows, b) through d) demonstrate the steps our algorithm takes to fold the model in a), e) the result of the segmentation step applied on a), the sides of the bed are split into two, each with a different texture f) through h) demonstrate selected iterations that our algorithm takes to fold the newly segmented model with $\epsilon_f = 0$. i) at $\epsilon_f = 0.5$, the model becomes unstable as the sides and both beds begin to move. j) through l) show selected iterations that our algorithm goes through to fold the model with $\epsilon_f = 0.5$.

Figure 18: Sliding Window: Our framework only handles fixed, and rotational joints, and therefore, can not fold a model such as the one in the photo because the windows are attached to the frame via sliding constraints. Image obtained from [34].

tary material for a video of the folding process for some of these models.

6. Conclusion and Discussion

We present a folding pipeline that solves three main problems to fold a given shape, namely *Order*, *Folding* and *Segmentation*. It solves simultaneously for the folding order and the folding parameters of each part to fold a given shape while keeping its functional feasibility up to a user-defined threshold. Our system is capable of folding a shape that is foldable by design using our solution for the *Order* and the *Folding* problems. However, if the shape is not foldable by design, we incorporate one more step, that is the *Segmentation* step to make it fold into a plausible, space conserving configuration.

Limitations. Our decision choices induce several limitations on our system. In what follows we discuss limitations of our algorithm and how they might impact the results.

- **Genetic Algorithm.** Our algorithm relies on a genetic algorithm approach to obtain an optimal folding order. The speed of this approach relies on the initialization of the initial population. That is, if the optimal order is present in the initial population, or an order that is close to optimal, the algorithm will quickly find it. However, if the optimal order is far from any element in the population, the algorithm will still find it, but it will take more time.

- **Segmentation.** We limit the number of splits to two splits per each segmentation iteration. This choice is mainly to minimize the editing done to the original model and to help keep the complexity of the folding order and the folding angle/axis combinations to a minimum. Of course this comes at the cost of overlooking better segmentation strategies that could help make the model more compact at its folded state.

- **Joint Types.** We chose to limit the joint types used in our algorithm to two types, fixed joints and rotational joints. This limits the types of models that our algorithm can handle as shown in Figure 18. We plan to explore other types of joints, like slider, and multi-axis rotational joints in future work.

Future Work. There are several areas that could be explored further to build on this work.

- **Functional Feasibility.** We proposed a method that determines the locations and the types of the joints to place on the different parts of the model to make the

model foldable while keeping its functionality up to a user-defined threshold. In some cases, both goals do not compete, but in many cases they do. For future work, we plan to introduce a 'stiffness' factor to the joints to make it harder, requiring forces of higher magnitudes, to move some joints compared to others. This factor will decrease the competition between the folding and the functional feasibility objectives as it will allow placing joints that are not fixed to allow folding, and at the same time, will increase their stiffness so they are not easily moved, and therefore the model will be harder to collapse.

- **Other Extensions.** We would like to expand our framework to fold more classes of objects, like cars and construction vehicles. Furthermore, we would like to incorporate a fabrication component with our method to place the physical joints on the actual parts and produce them as real life pieces of furniture.

[1] Sofa bed (2012).
URL http://www.232designs.com/wp-content/uploads/2012/02/sofa-bed03.png

[2] W. Xu, J. Wang, K. Yin, K. Zhou, M. Van De Panne, F. Chen, B. Guo, Joint-aware manipulation of deformable models, in: ACM TOG (Proc. SIGGRAPH), Vol. 28, 2009, pp. 35:1–35:9.

[3] N. J. Mitra, Y.-L. Yang, D.-M. Yan, W. Li, M. Agrawala, Illustrating how mechanical assemblies work, ACM TOG (Proc. SIGGRAPH) 29 (4) (2010) 58:1–58:11.

[4] L. Luo, I. Baran, S. Rusinkiewicz, W. Matusik, Chopper: Partitioning models into 3d-printable parts, ACM TOG (Proc. SIGGRAPH) 31 (6) (2012) 129:1–129:9.

[5] M. Bächer, B. Bickel, D. L. James, H. Pfister, Fabricating articulated characters from skinned meshes, ACM TOG (Proc. SIGGRAPH) 31 (4).

[6] Y. Zheng, D. Cohen-Or, N. J. Mitra, Smart variations: Functional substructures for part compatibility, in: Computer Graphics Forum (Proc. EUROGRAPHICS), 2013.

[7] H. Li, I. Alhashim, H. Zhang, A. Shamir, D. Cohen-Or, Stackabilization, ACM TOG (Proc. SIGGRAPH Asia) 31 (6) (2012) Article 158.

[8] Y. Zhou, S. Sueda, W. Matusik, A. Shamir, Boxelization: Folding 3d objects into boxes, ACM TOG (Proc. SIGGRAPH) 33 (4) (2014) 71:1–71:8.

[9] H. Li, R. Hu, I. Alhashim, H. Zhang, Foldabilizing furniture, ACM Trans. Graph. 34 (4) (2015) 90:1–90:12. doi:10.1145/2766912.
URL http://doi.acm.org/10.1145/2766912

[10] S. M. LaValle, Planning Algorithms, Cambridge University Press, 2006.

[11] E. Driskill, E. Cohen, Interactive design, analysis, and illustration of assemblies, in: Proceedings of the 1995 symposium on Interactive 3D graphics, 1995, pp. 27–34.

[12] M. Agrawala, D. Phan, J. Heiser, J. Haymaker, J. Klingner, P. Hanrahan, B. Tversky, Designing effective step-by-step assembly instructions, ACM TOG (Proc. SIGGRAPH) 22 (3) (2003) 828–837.

[13] A. Lambert, Disassembly sequencing: a survey, Int. Journal of Production Research 41 (16) (2003) 3721–3759.

[14] L. Zhu, W. Xu, J. Snyder, Y. Liu, G. Wang, B. Guo, Motion-guided mechanical toy modeling, ACM TOG (Proc. SIGGRAPH Asia) 31 (6) (2012) 127:1–127:10.

[15] J. Guo, D.-M. Yan, E. Li, W. Dong, P. Wonka, X. Zhang, Illustrating the disassembly of 3D models, Comput. Graph. 37 (6).

[16] T. Shao, W. Li, K. Zhou, W. Xu, B. Guo, N. J. Mitra, Interpreting concept sketches, ACM Transactions on Graphics 32 (4).

[17] D. Ceylan, W. Li, N. J. Mitra, M. Agrawala, M. Pauly, Designing and fabricating mechanical automata from mocap sequences, ACM Transactions on Graphics 32 (6).

[18] M. Lau, A. Ohgawara, J. Mitani, T. Igarashi, Converting 3D furniture models to fabricatable parts and connectors, ACM TOG (Proc. SIGGRAPH) 30 (4) (2011) 85:1–85:6.

[19] S. Xin, C. Lai, C. Fu, T. Wong, Y. He, D. Cohen-Or, Making burr puzzles from 3D models, ACM TOG (Proc. SIGGRAPH) 30 (4) (2011) 97:1–97:8.

[20] P. Song, C.-W. Fu, D. Cohen-Or, Recursive interlocking puzzles, ACM TOG (Proc. SIGGRAPH Asia) 31 (6) (2012) 128:1–128:10.

[21] S. Coros, B. Thomaszewski, G. Noris, S. Sueda, M. Forberg, R. W. Sumner, W. Matusik, B. Bickel, Computational design of mechanical characters., ACM Trans. Graph. 32 (4) (2013) 83.

[22] J. Calì, D. Calian, C. Amati, R. Kleinberger, A. Steed, J. Kautz, T. Weyrich, 3d-printing of non-assembly, articulated models 31 (6) (2012) 130:1–130:8.

[23] R. Gal, O. Sorkine, N. J. Mitra, D. Cohen-Or, iWIRES: an analyze-and-edit approach to shape manipulation, ACM TOG (Proc. SIGGRAPH) 28 (3) (2009) 33:1–33:10.

[24] Y. Wang, K. Xu, J. Li, H. Zhang, A. Shamir, L. Liu, Z. Cheng, Y. Xiong, Symmetry hierarchy of man-made objects, in: Computer Graphics Forum, Vol. 30, 2011, pp. 287–296.

[25] Y. Zheng, H. Fu, D. Cohen-Or, O. K.-C. Au, C.-L. Tai, Component-wise controllers for structure-preserving shape manipulation, in: Computer Graphics Forum (Proc. EUROGRAPHICS), Vol. 30, 2011, pp. 563–572.

[26] N. J. Mitra, M. Wand, H. Zhang, D. Cohen-Or, M. Bokeloh, Structure-aware shape processing, in: EUROGRAPHICS State-of-the-art Report, 2013.

[27] K. Mamou, HACD : Hierarchical approximate convex decomposition.

[28] Bullet physics library.
URL http://bullet.sourceforge.net/

[29] T. Akenine-Möller, E. Haines, N. Hoffman, Real-Time Rendering, Third Edition, Taylor & Francis, 2008.

[30] Foldable dining table (22-13-2014).
URL https://3dwarehouse.sketchup.com/warehouse/getpubliccontent?contentId=10e0608ebb78-4f4a-9b91-d48a85f078ad

[31] minimalist and modern modular kitchen concept (2015).
URL http://modernhouseinsight.com/compact-and-foldable-modern-kitchen-concept-for-small-home-2011minimalist-and-modern-modular-kitchen-concept

[32] Folding bed (2013).
URL http://www.viralnova.com/not-a-box/

[33] The crates (2015).
URL http://naihanli.com/products/the-crates/

[34] Folding sliding (2015).
URL http://www.fbs.co.in/folding-sliding.php

[35] K. Shoemake, Animating rotation with quaternion curves, SIGGRAPH Comput. Graph. 19 (3) (1985) 245–254.