Apparent Layer Operations for the Manipulation of Deformable Objects

Takeo Igarashi*
The University of Tokyo / JST ERATO

Jun Mitani†
University of Tsukuba / JST ERATO

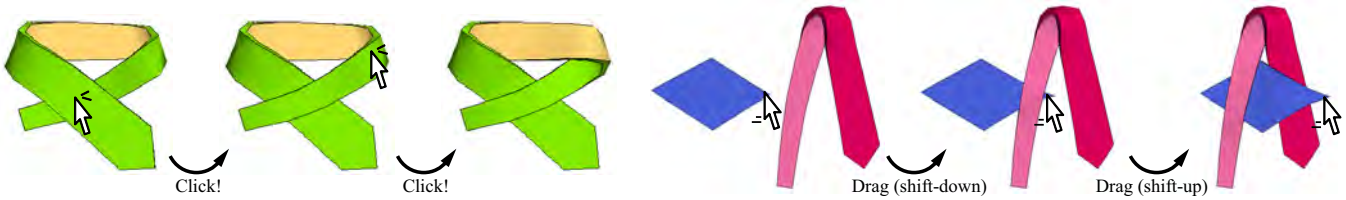

Figure 1: *Apparent layer operations. Left: Layer swap allows the user to swap local layers under the clicked point. Right: Layer-aware dragging allows the user to drag over (shift-up) or under (shift-down) a colliding object in the screen space.*

Abstract

We introduce layer operations for single-view 3D deformable object manipulation, in which the user can control the depth order of layered 3D objects resting on a flat ground with simple clicks and drags, as in 2D drawing systems. We present two interaction techniques based on this idea and describe their implementation. The first technique is explicit layer swap. The user clicks the target layer, and the system swaps the layer with the one directly underneath it. The second technique is layer-aware dragging. As the user drags the object, the system adjusts its depth automatically to pass over or under a colliding object in the screen space, according to user control. Although the user interface is 2.5D, all scene representations are true 3D, and thus the system naturally supports local layering, self-occlusions, and folds. Internally, the system dynamically computes the apparent layer structure in the current configuration and makes appropriate depth adjustments to obtain the desired results. We demonstrate the effectiveness of this approach in cloth and rope manipulation systems.

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction Techniques; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric Algorithms

Keywords: Local layering, 3D user interfaces, deformable objects, physical simulation, modeling interfaces

1 Introduction

Three-dimensional object manipulation using a two-dimensional input device is difficult. Depth information is not directly accessible in a standard 2D view, and the user typically has to switch to another view for depth control. Three-dimensional input devices are available, but depth control (positional control orthogonal to viewing direction) is still difficult. However, in some applications, precise depth is relatively unimportant, and only the relative depth ordering of the objects is of interest. In this case, the user interface can be significantly simplified by allowing the user to discretely control depth ordering, as in 2.5D scene-editing systems (2D objects with layers).

In this paper, we introduce two interaction techniques based on this idea and describe their implementation designed for clothes and ropes resting on a flat ground. The first technique is an extension of local layering [McCann and Pollard 2009] applied to 3D modeling. The user clicks a layered object and changes the stacking order of the layers underneath the mouse cursor (Figure 1, left). Our current implementation swaps the depths of the two topmost layers and adjusts the depths of other areas when necessary to maintain consistency. Layer swapping is more difficult in 3D than in 2D because naive swaps can cause interpenetrations at fold lines. The second technique is layer-aware dragging. As the user drags the object, the system automatically adjusts its depth to pass over or under a colliding object (Figure 1, right). In our system, pressing the shift key toggles the dragging operations between the drag-over and drag-under modes.

Unlike original local layering, our representation uses true 3D models and supports layer operations even when there are self-occlusions and folds. One can also apply various 3D effects that are difficult in 2D representations, such as shading and physical simulation. Layer operations are particularly useful for manipulating interwoven deformable objects, and so we demonstrate the effectiveness of this approach in cloth and rope manipulation systems. It might be possible to develop similar layer operations for other targets, such as stacked rigid objects or flexible objects with more complicated geometries (such as an octopus or a squid). However, the user interface and implementation details would vary with the target application, and thus these are reserved for future research.

Internally, the system dynamically computes the apparent layer structure and makes the necessary depth adjustments to obtain a valid result. In the layer swap, the system swaps the depth order of the layers under the mouse cursor and that of nearby areas (to prevent penetration), as in local layering. The original local layering algorithm uses a greedy algorithm to propagate swaps but is ineffective in our case because of folds and self-occlusions. We therefore enumerate all possible depth orders globally and then search for the valid combination with the minimum amount of change. In the layer-aware drag, the system adjusts the depth of the dragged object so that it becomes shallower (or deeper) than a colliding object

*e-mail:takeo@acm.org
†e-mail:jmitani@gmail.com

while maintaining consistency with the depth order at the previous object position.

Our technique might not be used very frequently, but it will save a significant amount of labor those times when it is applicable. It enables the simple operations shown in Figure 1 to be performed with a few clicks. Performing these same operations using a standard modeling interface can be extremely tedious, because such an interface requires the user to select appropriate vertices to move, change viewing directions, adjust the depth, and then return to the original view to verify the results.

2 Related Work

The traditional approach to 3D scene editing via a 2D interface is to use three separate views. However, various single-view control methods have been proposed. Interactive shadows [Herndon et al. 1992] uses shadows projected onto walls and a floor as a handle for depth control. Through-the-Lens Camera Control [Gleicher and Witkin 1992] uses constraints specified in the projected image to control the camera position and orientation in a 3D scene. Two-handed and multi-touch controls are also used to control additional degrees of freedom [Zeleznik et al. 1997, Reisman et al. 2009]. The Sketch system [Zeleznik et al. 1996] determines the depth of a newly created or dragged object based on the assumption that it rests on top of another object in the scene. Our work builds on these previous attempts and presents specialized depth control methods for interwoven deformable objects.

Our work is inspired by recent progress in advanced representations and operations for 2.5D drawings. Williams [1997] introduced an algorithm for inferring the correct layer structures from contour visibility information. Asente [2007] presented an algorithm for preserving apparent layer structures during the editing of a planar map. Wiley [2006] presented a data structure and algorithm for representing self-intersecting boundary curves in vector graphics. McCann and Pollard [2009] introduced a pixel-based representation and algorithm for local stacking of 2D graphical objects. Our goal is to enable richer expressions by using a 3D representation while providing a simple user interface similar to those of these 2.5D systems.

Automatic inference of 3D geometry from self-occluding 2D input is discussed in some sketch-based modeling systems. Karpenko and Hughes [2006] introduced an algorithm for inflating a 2D region enclosed by a boundary curve that contains local self-intersections. Cordier and Seo [2007] extended this algorithm to the inflation of shapes that contain self-intersecting regions such as knots. The knot plot system [Scharein 1998] provides a sketch mode in which a new 3D knot can be designed by drawing a 2D knot with user-specified depth ordering for each intersecting segment (left-button click requests passing over and right-button click requests passing under). Fabianowski and Dingliana [2008] used pen pressure to control depth. Whereas these systems focus on the initial creation of objects, we focus on the manipulation of existing 3D objects.

Computing layer structure (depth ordering) of objects in a 3D scene has been discussed in a variety of contexts. Snyder and Lengyel [1998] presented an algorithm for sorting geometric objects into layers to accelerate the rendering process. Eiseman et al. [2009] presented a method for converting a 3D scene to 2.5D vector graphics by segmenting the surfaces of self-occluding objects. Our contribution is a method of modifying the given layer structure and reconstructing a new 3D scene from the updated layer structure.

Problems related to folding are a primary subject of study in the field of origami [Demaine and Rourke 2007]. Origami foldability is concerned with the possibility of folding a given crease pattern into a flat shape, and origami design is concerned with how to fold a single sheet of paper into a target shape. The difficulty of finding a globally consistent stacking order of local layers is known to be NP-complete [Bern and Hayes 1996]. Our contribution is the development of an algorithm for making a valid change to the stacking order and the packaging of that algorithm as a tool to assist in 3D modeling tasks.

There are studies on folding a sheet of paper interactively in a virtual space. Miyazaki et al. [1996] presented a geometric origami folding system that maintains the consistency of the stacking order during the user's folding operations. Burgoon et al. [2006] introduced a physical simulation based on a discrete shell model for interactive origami folding. However, these authors did not provide tools for directly changing the stacking order.

Interaction with deformable objects has most often been discussed in the context of simulation systems, such as surgical simulations [Brown et al. 2004]. In such instances, the goal is to faithfully simulate physically realistic behavior for training purposes. However, our goal is to support the creative design process by adding context-aware operations to a modeling system without being restricted to physically realistic behavior. A seminal work by Igarashi and Hughes [2001] addressed a similar objective, but they did not consider the cloth-cloth interaction.

3 The User Interface

Our interaction techniques are designed mainly for the manipulation of deformable objects in a 3D scene. We have implemented two particular systems (cloth and rope manipulation systems) to demonstrate our approach. These two examples show that our method can handle typical 2D and 1D deformable objects in 3D space.

We use a single view for visualization and control of a 3D scene. The user edits the 3D scene with a 2D pointing device by combining clicking and dragging operations. Our technique can be applied from an arbitrary viewing direction in principle by considering the projections of the layers on the screen. However, for clothes and ropes lying on the floor, the layers make the most sense when viewed vertically from the top. We therefore focus on a vertical view (camera pointing downward, screen parallel to the ground) to simplify the explanation in this paper.

3.1 Layer swap

This method allows the user to directly modify the depth order by clicking on a layered object in the 3D scene. The current implementation swaps the two topmost layers. It is not always possible to swap only the first two layers, because penetration at a fold may occur. In this case, the system automatically applies additional swaps to other layers under the clicked position. The system also propagates swaps to the area around the clicked position when necessary to prevent interpenetration. When multiple possibilities exist, the system selects the one that causes the minimum amount of change to the current configuration.

Our current technique of swapping the first two layers is not the only approach, and many other implementations are possible (e.g., pushing the topmost layer to the bottom, popping the bottom layer to the top, or showing all layers in a list and asking the user to specify the desired operation [McCann and Pollard 2009]). We chose the current implementation for several reasons. First, a single click is simple and fast, allowing for fluent interaction. Second, the user can obtain various configurations by successively clicking on different locations (see our user study in the Results section). Finally, deep layers are not immediately visible, so they will probably be

manipulated less frequently. Manipulation of deep layers requires additional operations (such as temporarily hiding top layers), the discussion of which is outside the scope of this paper.

Figure 2ab shows a simple example of a layer swap. Figure 2cf shows an example in which propagation is necessary to prevent penetration. There are three possibilities (d, e, f), and the system returns (d) because it produces the minimum amount of change (see the Algorithm section for a definition). It is sometimes difficult to predict which result will be returned, but the user can quickly explore the other possibilities by clicking on other locations in the returned result until the desired configuration is obtained. Figure 2gh shows an example in which it is not possible to swap the first two layers only because of a fold, and here the system applies an additional swap.

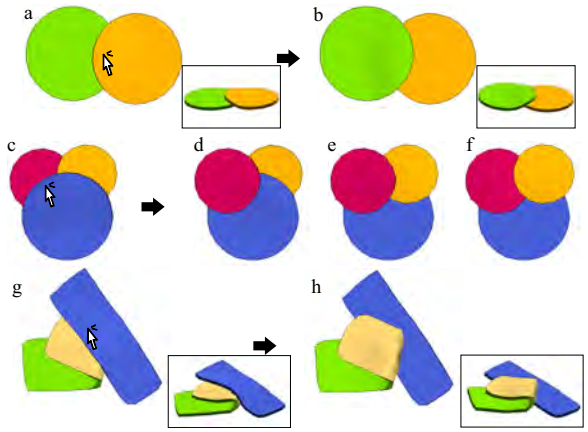

Figure 2: *Basic layer swap operations.*

Figure 3 (top) shows an extreme case (a square napkin folded twice). In this case, two valid configurations satisfy the user request (swap the two layers under the cursor), and the system returns the configuration with the fewest swaps (center). The user can easily obtain the other result (right) by clicking the center of the returned result (center). Figure 3 (bottom) shows another extreme case (a spiral). In this case, the system reverses the order of all the local layers, which is the only valid configuration. Figure 4 shows layer-swap operations applied to a knot.

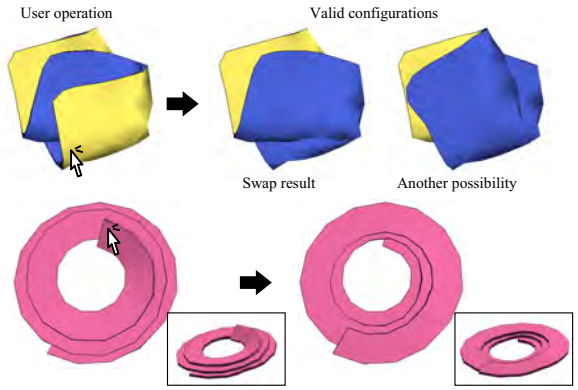

Figure 3: *Complicated layer swaps (napkin and spiral).*

3.2 Layer-aware dragging

In our method, depth is adjusted automatically during the dragging operation to make it easier to control the depth order. If the dragged object is already over or under another object, the system tries to maintain the depth order by adjusting the depth to prevent unexpected penetration. When the dragged object collides with another object in the screen space, the system automatically adjusts the depth of the dragged object so that the object passes over or under the colliding object (shift-down is associated with passing under in our current implementation).

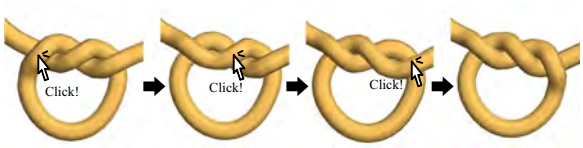

Figure 4: *Layer swap for a rope.*

We currently support two types of drag. One is boundary drag, in which the user drags a boundary vertex, and the rest of the object follows the dragged vertex according to the physical simulation. This is useful for rotating, bending, and folding an object. The other is intact drag, in which the user drags an internal vertex, and all of the other vertices of the object move together with the dragged vertex. This is useful for translating the entire object in one direction. In a boundary drag, the system adjusts the depth of the area around the dragged vertex. In an intact drag, the system adjusts the depth of the area around the advancing front of the object (boundary edges whose direction of motion is outside the object). The physical simulation adjusts the depth of the remaining area and resolves collisions.

Figure 5 (top) shows an example of a boundary drag. The user first drags A over B (in pass-over mode). The user then drags A under C (in pass-under mode). The depth order of A, D, and E is preserved during this procedure. This operation is particularly useful for showing the process of tying a knot (Figure 5, bottom).

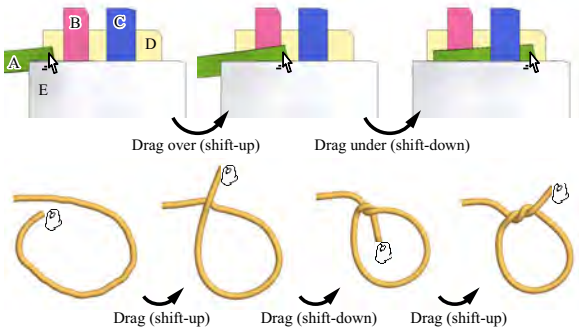

Figure 5: *Layer aware dragging.*

4 Algorithm

We use standard representations for deformable 3D objects. Cloth is represented as a single-layer triangular mesh, and rope is represented as a linear chain of spherical primitives. We use a simple physical simulation based on shape matching [Müller et al. 2005, Rivers and James 2007] with basic collision handling to maintain the integrity of deformable objects during manipulation. We temporarily compute a special data structure (apparent layers) to apply our layer-based operations and discard them once they have been completed. We assume that any interpenetration is resolved by physical simulation before applying the layer operations.

4.1 Layer swap

The layer swap algorithm is an extension of the algorithm used in local layering [McCann and Pollard 2009]. Original local layering uses single-pass propagation of flip-up (-down) operations but does not work for objects containing folds and self-occlusions. Figure 6 shows examples. Suppose that the user wanted to swap A and B at screen location p in Figure 6 left. The system would initially move A under B and then try to move A under C because C is above B. However, this would cause interpenetration at location q. To resolve this interpenetration, the system has to backtrack and move A under D. Figure 6 right is a simplified view of the spiral shown in Figure 3. If the user tried to swap A and B, the system would first move A directly underneath B. However, as propagation proceeded, the system would eventually move B underneath C, nullifying the previous swap of A and B. This example shows that naive propagation of layer swaps may not work even with backtracking. We must therefore examine layer order combinations more globally to find a valid result. The detailed process is described in the following subsection.

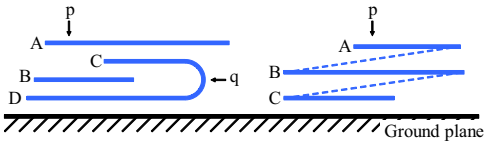

Figure 6: *Cases in which simple propagation does not work. Fold (left) and spiral (right).*

4.1.1 Constructing a list graph

The system first identifies local layer structures by examining overlaps among object fractions. In the case of cloth, the system first divides the folded meshes at silhouette edges as a preprocessing step. The system then projects all fold lines and boundary edges onto the screen, thereby constructing a planar map (Figure 7 b). Each area in the planar map is called a region, and in every region, each connected fraction of a mesh is considered to be a layer. Two separate fractions are generated along a fold line. Each region contains multiple layers sorted according to their depth if meshes overlap inside the region. A layer is associated with a set of mesh vertices inside a region.

In the case of rope, we project the silhouette of the rope(s) onto the screen and obtain a 2D planar map. Each area of the map is called a region (Figure 7 d), and in every region, a connected rope fraction becomes a layer. A layer is associated with spherical primitives whose screen projections intersect the region. A spherical primitive can be associated with multiple layers.

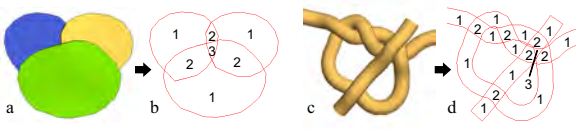

Figure 7: *Examples of planar maps. Numbers indicate the number of layers in each region.*

We then construct a list graph (Figure 8) to represent a global layer structure, as in original local layering [McCann and Pollard 2009]. Each node of the list graph corresponds to a region and contains an ordered list of layers. Each edge of the list graph corresponds to an adjacency relation between regions. If any layer pair between one region and another is connected in the original object representation, the two regions are connected in the list graph. We remove regions that contain only one layer and then remove regions that are not connected to the clicked region. The entries of a list graph in original local layering are the IDs of the original images while they are object fractions in our system for handling self-occlusion.

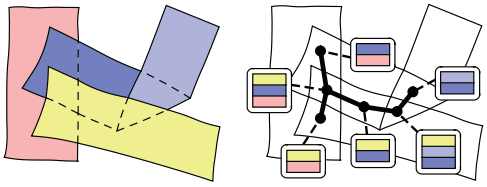

Figure 8: *List graph.*

4.1.2 Updating the list graph

The next task is to apply a valid change to the current list graph. We formulate this as a search problem, in which we examine all valid layer order combinations and return the one that causes the least amount of change to the original configuration while satisfying the user request (the topmost layer at the clicked position moves underneath the layer that was immediately beneath it).

A list graph (a layer order combination) is valid if none of the following violations are present (Figure 9). A type I violation is an inconsistency between adjacent regions. This violation occurs when the layer order is swapped in adjacent regions. In other words, layer A is above layer B in one region, whereas a layer connected to A is underneath a layer connected B in an adjacent region. Connection here means adjacency in the original object representation. A type II violation is an inconsistency inside of a region. This violation occurs when two layers sharing a fold (silhouette edges) are not adjacent in the layer order of the region. This corresponds to a situation in which there is a penetration at a fold, and it occurs only in cloth systems. Type I violations have previously been considered in original local layering, but type II violations are introduced here for the first time.

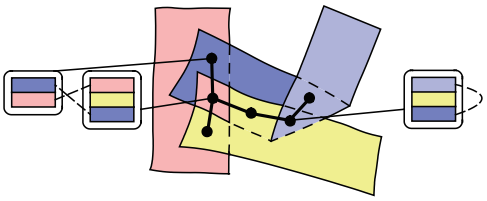

Figure 9: *Type I violation (left) and Type II violation (right).*

We solve this search problem by explicitly enumerating all possible layer order combinations and returning the best one. An alternative approach would be to use a heuristic beam search to save time, but a beam search can miss good solutions. We found that an exhaustive search is fast enough for our examples (see the Results section) and is more reliable in terms of obtaining plausible results. We do accelerate the process by culling unnecessary branches, but this problem is combinatorial in nature [Bern and Hayes 1996], and there is no easy solution.

We first explicitly enumerate all valid layer orders in each region, independently considering type II violations. We enumerate all the permutations and then remove those that contain a type II violation. For the starting region chosen (clicked) by the user, we also remove permutations that do not satisfy the user request (namely that the

first layer is to be moved underneath the second layer). The system sorts the layer orders in each region according to the number of swaps (with 0 swap being the original layer ordering). Figure 10 shows the results of this permutation enumeration process when the user clicks the bottom left region in Figure 9.

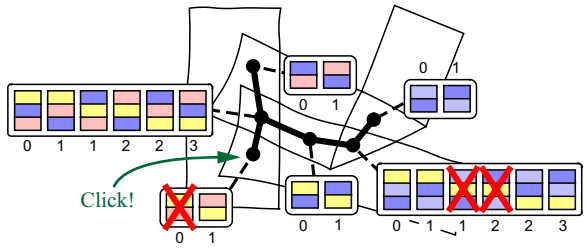

Figure 10: *Valid permutations in each group.*

We then enumerate the valid combinations of these permutations (layer orders) by visiting the regions one by one. We start the search at the clicked region and propagate it to nearby regions in a breadth-first manner. The system first collects valid layer orders in the starting region and retains them as a tentative solution set. The system then visits the next region and examines all combinations between the layer orders in that region and the layer orders in the tentative solution set. Combinations without type I violations are added to the tentative solution set. Figure 11 shows the valid layer order combinations in our example.

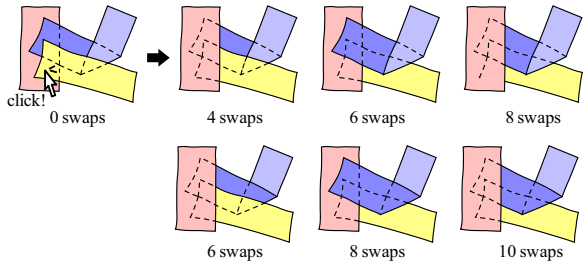

Figure 11: *Enumeration of all valid combinations.*

Finally, the system computes the score of each combination, measuring the amount of change from the original configuration. We currently compute the score as the number of swaps weighted with the size of each region. A swap is a pair of layers whose depth order in the updated layer order is the reverse of what it was in the original layer order. (Thus there are a maximum of $n(n-1)/2$ possible swaps in a region with n layers.) The combination with the lowest score is returned as a result, and all other candidates are discarded. We prototyped and tested an approach that exhibits other candidates to the user as thumbnails [Igarashi and Hughes 2002], but we found that it is difficult to understand the configurations from small thumbnails and tedious to examine them one by one. The user can quickly explore other possibilities by applying layer swap (clicking) at other locations, and this is much faster than selecting a configuration from multiple candidates. Another possibility is to cycle though the multiple candidates with repeated clicks on the same region, but we have not tested this approach yet.

This scoring scheme is an initial experiment, and many other possibilities exist. One possible extension would be to add extra score when the folding direction is changed (from valley fold to mountain fold and vice versa), because this is a perceptually significant change. It might also be worthwhile to consider the distance of each region from the clicked position, because it is better to keep the changes near the clicked position. The precise details will depend on the application and are reserved for future research.

4.1.3 Computing the updated geometry

The result of the search is a collection of valid layer orders in each region. The system must then compute the actual depth of each layer. We accomplish this by solving a least-squares problem, in which the system tries to make the depth differences between adjacent layers in a region equal to the thickness of the object primitives, and the depth differences between connected layers in different regions equal to zero. We also add a positional constraint to fix the center. The system then updates the vertex depths based on the computed layer depths. The depth of a vertex along a fold line in a cloth model is given as the average of the two corresponding vertices in the separated layers. Figure 12 (middle) shows the result of this computation. The system applies the physical simulation based on shape matching to obtain the final result (Figure 12, right). The depth of a primitive in a rope is given as the average of the associated layer depths.

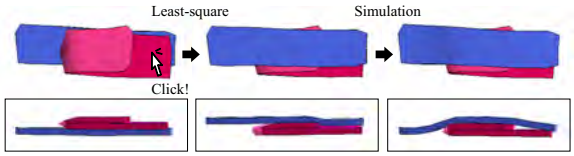

Figure 12: *Updating the geometry.*

4.2 Layer-aware dragging

In this subsection we describe the algorithm for computing the depth of the dragged vertex in a boundary drag (deformable drag). For an intact drag (non-deformable drag), the same algorithm is applied one by one to all vertices along the advancing front. The layer computation described here is based on the fraction of the objects. A fraction is a face in a cloth system and a spherical primitive in a rope system. The dragged fraction consists of multiple faces connected to the dragged vertex in a cloth system.

The system first computes the layer structure (depth-ordered list of fractions) at the current location of the dragged fraction (Figure 13 a). The system projects all fractions in the scene onto the screen, identifies fractions that intersect the dragged fraction in the screen space, and sorts fractions according to their depths. The system then similarly computes the layer structure at the target screen space location of the dragged fraction (Figure 13 b).

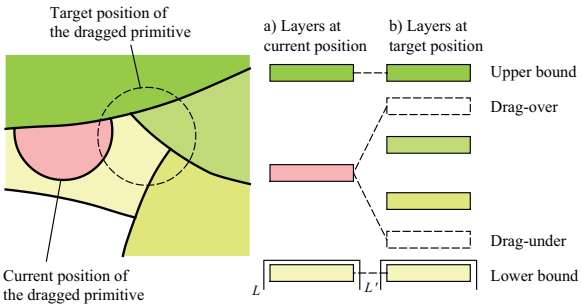

Figure 13: *Layer analysis for layer aware dragging.*

The system makes a list L by collecting fractions lower than the dragged fraction at the current position. The system then makes

a list L' by identifying fractions at the target position that are included in or connected to a fraction in L. The topmost fraction in L' is defined as a lower bound fraction. The system also identifies an upper bound fraction.

If the system is in drag-over mode, it identifies the highest fraction below the upper bound and sets it as a new lower bound. Similarly, if the system is in drag-under mode, the lowest fraction above the lower bound becomes a new upper bound.

Finally, the system determines the depth of the dragged fraction at the target location. The default value is the depth at the previous location. If the default depth is outside the range between the new lower and upper bounds, the system adjusts the depth so that it lies within those bounds.

Objects have a certain thickness, so the system offsets the lower and upper bounds with an appropriate thickness. If the depth difference between the upper and lower bound is less than twice the offset, the system sets the target depth to the middle of the two bounds. We expect that the simulation process subsequently pushes the primitives apart.

5 Results

Figure 14 shows textured examples. These examples represent the complexity of our target scenarios (mostly one or two layers, and no more than five or six layers). We also invited three test users to try our system. We provided them with textured mesh models, and after roughly 5 minutes of tutorial and practice, they were able to compose each scene in a few minutes. For the most part, they used layer-aware dragging in these examples, but they did occasionally use layer swap.

Figure 14: *Textured Examples.*

Figure 15 shows some relatively elaborate examples. These results indicate that our algorithm is reliable in fairly complicated cases. Table 1 displays the statistics for these examples as well as for those shown in Figure 3. The prototype system is implemented using Java on a laptop with 1.2 GHz CPU, 2GB RAM. All operations were completed within a few seconds, which is sufficiently fast for interactive operations. We also tested beam search with backtracking and found that exhaustive search is comparable to beam search except in the most complicated cases.

Table 1: *Statistics (Figure 3 and 15).*

			spiral	napkin	scarf	ribbon	tie
number of vertices			213	184	90	165	122
number of groups			4	12	25	33	45
number of layers			7	24	45	77	107
number of valid combinations			1	2	12	40	294
time (msec)	planar map & list graph		719	453	218	422	407
	search	exhaustive	1.3	34	10	76	1453
		beam	1.3	32	0.5	4.5	5.3

One issue with layer swap is that it sometimes performs complicated swaps to maintain consistency, which might confuse the user. We conducted another informal user study with 10 different test

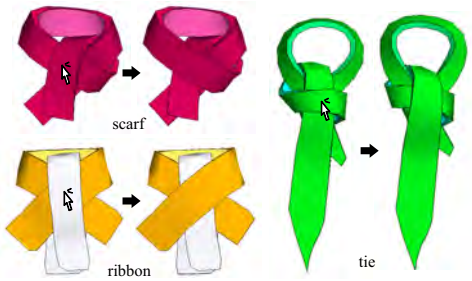

Figure 15: *Examples of layer swap operations.*

users to verify that this is not a problem in practice. We first presented a brief tutorial using the example shown in Figure 2g. Drag and camera control were disabled. We then provided the scene shown in Figure 16 (left; equivalent to Figure 8) and asked the users to create all 12 possible configurations using layer swap (Task 1) as a part of their training. Finally, we asked the users to perform the task shown in Figure 16 (center and right) using layer swap (Tasks 2 and 3). Table 2 displays the results. The tutorial took less than 2 minutes. Most of the users quickly grasped the fundamentals of layer swap and completed all tasks without much difficulty. A few test users found some of the swap results confusing but still successfully completed the tasks after some trial and error.

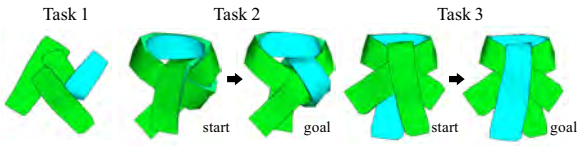

Figure 16: *Configurations used in the user study*

Table 2: *Results of the user study.*

	task1		task2		task3	
	average	stdev	average	stdev	average	stdev
time (sec)	121.3	41.3	47.5	22.0	39.2	12.5
# of clicks	24.2	8.7	10.0	7.2	9.4	4.6
# of undos	0.0	0.0	0.4	1.3	0.0	0.0

Figure 17 shows sample knots created using our knot design tool. Although similar results can be found in previous research (e.g., [Fabianowski and Dingliana 2008]), our contribution lies in our technique of making local changes to a given knot with simple click and drag operations. This enables quick exploration of alternative configurations, which is essential to the creative authoring process.

6 Limitations and Future Work

We only tested our algorithm with relatively small models and it can be too slow for more complicated models. The main bottleneck is the exhaustive enumeration part of the algorithm. Its computational complexity is $O(\Pi_{i \in regions}(N_i!))$, where N_i is the number of layers in the i-th region. This part depends on the number of regions and layers in each region. So, the resolution of the model (number of faces) itself is not critical. It only affects the construction of the planar map, whose complexity is linear in the number of faces. One possible solution to the scalability issue is to use a beam search with exhaustive culling. This might miss good solutions, so we need to carefully examine how serious the problem is. Another approach is to have the user limit the search space manually by deactivating irrelevant objects in the scene. Manual intervention is important for

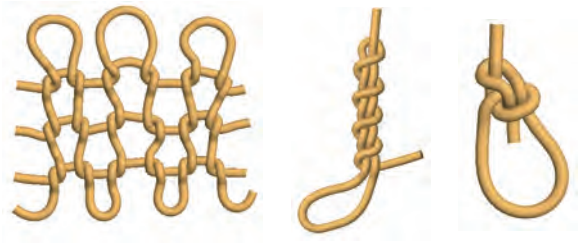

Figure 17: *Knots.*

the user to reliably obtain a desired result because even a computationally optimal solution can be counter-intuitive for the user if the search space is too large.

Another important future line of research will be to provide a more theoretical analysis of our layering problem. Our current swap interface allows the two topmost layers under the cursor to be swapped, but this does not guarantee that the user can obtain any arbitrary configuration. What, then, is the minimum set of operations? Most 2.5D systems support a limited number of operations, but it is not immediately clear that this would be sufficient in our case, because each swap can have unexpected side effects. Likewise, our current interface returns only one result at a time and requires the user to apply multiple swaps at different locations to obtain the desired end result. However, there is no guarantee that the user can reach an arbitrary target configuration.

We have focused on deformable objects in our current work because these objects can undergo local layer changes without it affecting distant locations. However, a small displacement at one end of a rigid object can cause a large displacement at the other end, resulting in unpredictable changes to the scene. The system must analyze the global structure to handle interlocked rigid objects and must also provide an interface that helps the user understand and control this global structure.

Our current system is designed for use with deformable objects resting on a flat surface, but the algorithm should work for general manifold surface as long as all layers in a given region are continuously and uniquely mapped to a topologically equivalent region on the base surface. For example, our algorithm can be applied to toroidal cloth on a toroid, but not spherical cloth on a toroid. We are especially interested in cloth and rope manipulation on body surfaces for application to garment design.

References

ASENTE, P., SCHUSTER, M., AND PETTIT, T. 2007. Dynamic planar map illustration. *ACM Trans. Graph. 26*, 3, 30.

BAUDELAIRE, P., AND GANGNET, M. 1989. Planar maps: an interaction paradigm for graphic design. In *Proceedings of CHI '89*, 313–318.

BERN, M., AND HAYES, B. 1996. The complexity of flat origami. In *Proceedings of SODA '96*, 175–183.

BROWN, J., LATOMBE, J.-C., AND MONTGOMERY, K. 2004. Real-time knot-tying simulation. *Vis. Comput. 20*, 2, 165–179.

BURGOON, R., GRINSPUN, E., AND WOOD, Z. 2006. Discrete Shells Origami. In *Proceedings of Computers And Their Applications*, 180–187.

CORDIER, F., AND SEO, H. 2007. Free-form sketching of self-occluding objects. *IEEE Comput. Graph. Appl. 27*, 1, 50–59.

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

EISEMANN, E., PARIS, S., AND DURAND, F. 2009. A visibility algorithm for converting 3D meshes into editable 2D vector graphics. *ACM Trans. Graph. 28* (July), 83:1–83:8.

FABIANOWSKI, B., AND DINGLIANA, J. 2008. Sketching complex generalized cylinder spines. In *Computer Graphics International 2008*, 270–276.

GLEICHER, M., AND WITKIN, A. 1992. Through-the-lens camera control. *Computer Graphics (Proceedings of SIGGRAPH 92). 26*, 331–340.

HERNDON, K. P., ZELEZNIK, R. C., ROBBINS, D. C., CONNER, D. B., SNIBBE, S. S., AND VAN DAM, A. 1992. Interactive shadows. In *Proceedings of UIST '92*, 1–6.

IGARASHI, T., AND HUGHES, J. F. 2001. A suggestive interface for 3D drawing. In *Proceedings of UIST '01*, 173–181.

IGARASHI, T., MOSCOVICH, T., AND HUGHES, J. F. 2005. As-rigid-as-possible shape manipulation. *ACM Trans. Graph. 24* (July), 1134–1141.

KARPENKO, O. A., AND HUGHES, J. F. 2006. Smoothsketch: 3D free-form shapes from complex sketches. *ACM Trans. Graph. 25*, 3 (July), 589–598.

MCCANN, J., AND POLLARD, N. 2009. Local layering. *ACM Trans. Graph. 28* (July), 84:1–84:7.

MIYAZAKI, S., YASUDA, T., YOKOI, S., AND TORIWAKI, J. 1996. An origami playing simulator in the virtual space. *The Journal of Visualization and Computer Animation 7*, 1, 25–42.

MÜLLER, M., HEIDELBERGER, B., TESCHNER, M., AND GROSS, M. 2005. Meshless deformations based on shape matching. *ACM Trans. Graph. 24* (July), 471–478.

REISMAN, J. L., DAVIDSON, P. L., AND HAN, J. Y. 2009. A screen-space formulation for 2D and 3D direct manipulation. In *Proceedings of UIST '09*, 69–78.

RIVERS, A. R., AND JAMES, D. L. 2007. FastLSM: fast lattice shape matching for robust real-time deformation. *ACM Trans. Graph. 26* (July).

SCHAREIN, R. G. 1998. *Interactive topological drawing*. PhD thesis. Adviser-Booth, K. S. and Adviser-Little, J. J.

SNYDER, J., AND LENGYEL, J. 1998. Visibility sorting and compositing without splitting for image layer decompositions. In *Proceedings of SIGGRAPH 1998*, 219–230.

WILEY, K., AND WILLIAMS, L. R. 2006. Representation of interwoven surfaces in 2 1/2 d drawing. In *Proceedings of CHI '06*, 65–74.

WILLIAMS, L. R. 1997. Topological reconstruction of a smooth manifold-solid from its occluding contour. *Int. J. Comput. Vision 23*, 1, 93–108.

ZELEZNIK, R. C., HERNDON, K. P., AND HUGHES, J. F. 1996. Sketch: an interface for sketching 3D scenes. In *Proceedings of SIGGRAPH 1996*, 163–170.

ZELEZNIK, R. C., FORSBERG, A. S., AND STRAUSS, P. S. 1997. Two pointer input for 3D interaction. In *Proceedings of SI3D '97*, 115–120.