Composite Rectilinear Deformation
for Stretch and Squish Navigation

James Slack and Tamara Munzner, *Member, IEEE*

Abstract—We present the first scalable algorithm that supports the composition of successive rectilinear deformations. Earlier systems that provided stretch and squish navigation could only handle small datasets. More recent work featuring rubber sheet navigation for large datasets has focused on rendering and on application-specific issues. However, no algorithm has yet been presented for carrying out such navigation methods; our paper addresses this problem. For maximum flexibility with large datasets, a stretch and squish navigation algorithm should allow for millions of potentially deformable regions. However, typical usage only changes the extents of a small subset k of these n regions at a time. The challenge is to avoid computations that are linear in n, because a single deformation can affect the absolute screen-space location of every deformable region. We provide an $O(k \log n)$ algorithm that supports any application that can lay out a dataset on a generic grid, and show an implementation that allows navigation of trees and gene sequences with millions of items in sub-millisecond time.

Index Terms—Focus+Context, information visualization, real time rendering, navigation.

◆

---

1 INTRODUCTION

Navigating through a visual representation of information by stretching or squishing regions of interest is an intuitive navigation metaphor that provides the experience of manipulating a flexible sheet, rather than changing the viewpoint of a rigid layout of items in a dataset. Within a stretchable region, such a metaphor preserves the relative spatial positions of items even though their absolute spatial positions change. One stretch and squish technique is called rubber sheet navigation [10], where the display acts as a metaphoric sheet with its borders nailed down, such that stretching one region requires shrinking in other regions of the sheet. Figures 1, 2, and 3 show examples of this navigation technique in action.

An important property of rubber sheet navigation techniques is that each successive deformation is a composition operation applied to the previous transformation state, which means that previous navigation actions continue to affect the current view. This composition provides a lightweight visual history, as shown in Figure 1. In this example, the effect of moving a horizontal boundary in Figure 1b is still visible after squishing a collection of rows in Figure 1c. Likewise, the horizontal column squishing shown in Figure 1d modifies, but does not erase, the results of the previous two vertical deformations.

Although the visual metaphor provided to the user is intuitive, the underlying infrastructure required to support this navigation is complex. The standard graphics pipeline supports a mapping from world to screen coordinates with a single projective transformation that encodes both the viewing and projection parameters of a camera. Stretch and squish navigation cannot be implemented with a single monolithic transformation that affects the entire scene. Instead, we discretize a scene into small, individually deformable regions.

In the standard graphics pipeline with unconstrained camera motion, many objects can be outside the view frustum. These objects are culled, so that only the visible objects in the scene are rendered; the number of objects drawn is normally far fewer than the total number of objects in the scene. Although rubber sheet navigation can be considered as a camera constraint guaranteeing that all objects are within the viewport, for sufficiently large datasets the number of visible ob-
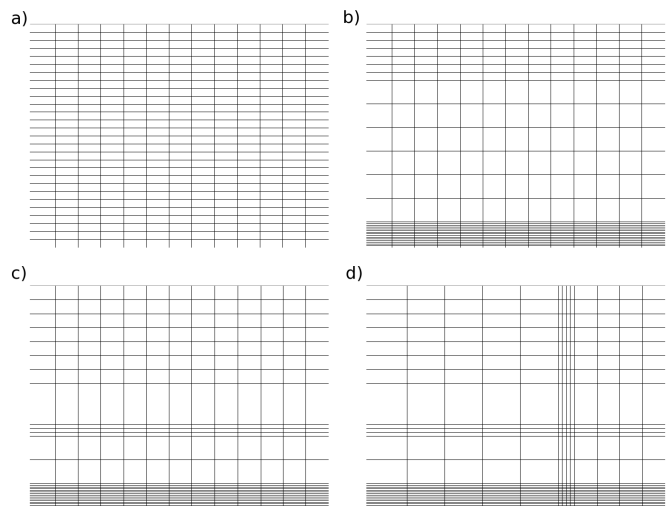

Fig. 1. Rubber sheet navigation actions performed on a rectilinear grid stretch regions of interest while shrinking the context. **a)** A uniform grid, in which dataset objects can be laid out, shows no screen-space priority to objects in any particular region. **b)** By stretching one region of objects vertically to focus on a subset of interest, regions not currently in focus are shrunk in the periphery as to not push any regions out of view. **c)** Previous navigations are not discarded when users change the focus with new deformations. The navigation history is composed of several deformations that result in a lightweight visual history. **d)** Horizontal navigations are independent of the vertical navigations.

jects is still far less than the total number in the scene. Although all objects are within the view frustum, most are of sub-pixel size. The PRISAD infrastructure [12] supports a sophisticated culling approach for scalable rendering that aggregates sub-pixel features to keep landmarks visible. PRISAD uses a hierarchical data structure that stores relative positions for region boundaries to quickly compute an absolute location for them. It calculates these absolute boundary locations only for the visible regions when rendering. The number of objects drawn is bounded by the display size, and is much smaller than the total number of objects in the scene.

Consequently, updating the navigation state should not require updating the position of every object in the scene, even if the deformation would change the absolute position of all objects, because only a sub-

● *James Slack is with the Department of Computer Science at the University of British Columbia, E-mail: jslack@cs.ubc.ca.*
● *Tamara Munzner is with the Department of Computer Science at the University of British Columbia, E-mail: tmm@cs.ubc.ca.*
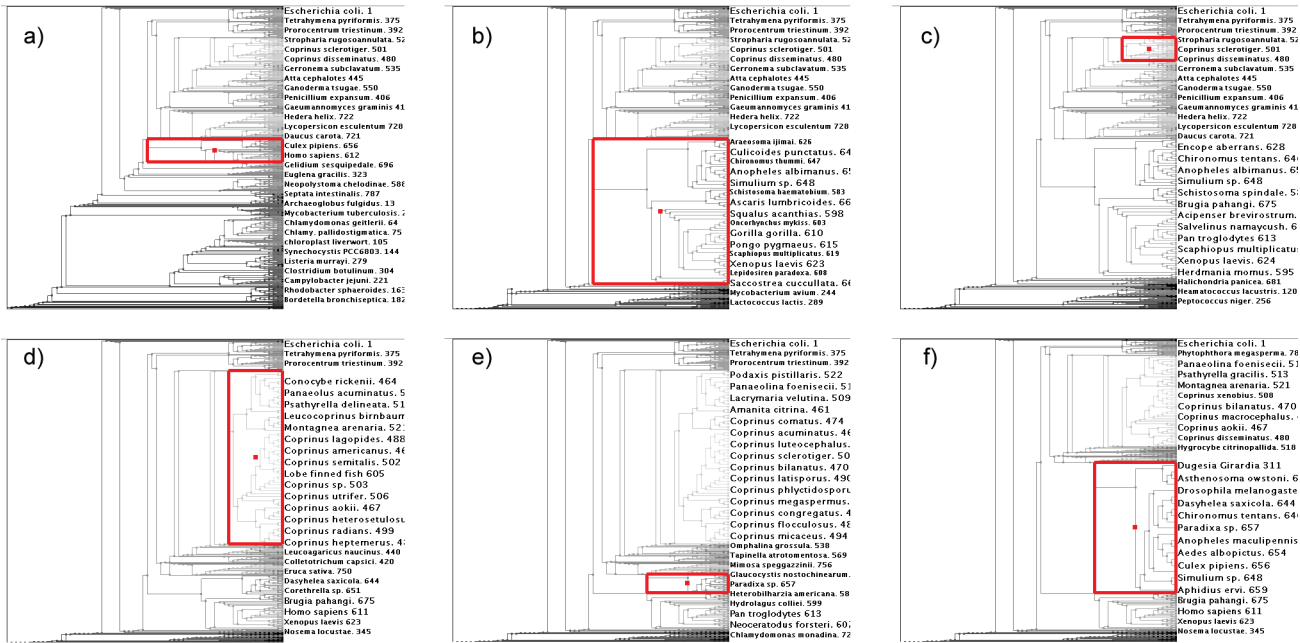
Fig. 2. User selected deformation regions, composed from several navigations, preserve the visual history of regions of interest. **a)** We select rectangular collective regions by dragging a cursor, which defines a current area of interest. **b)** We drag a selected region to reveal interesting dataset details, which squishes regions of context in the periphery. **c-f)** Our subsequent navigations do not undo previous navigations, since navigations persist to provide a visual history of our exploration.

set of them are drawn. The PRISAD paper [12] does not describe the navigation algorithm underlying the rendering infrastructure. The navigation solutions described in previous work [9] are linear in n, the total number of regions, and have only been demonstrated on an n in the hundreds.

We refer to the smallest deformable regions as *atomic* regions. *Collective* regions are collections of atomic regions. For maximum flexibility, the number of atomic regions should be large, on the order of the number of items being displayed. However, a typical navigation action would only directly change some small number of region boundaries at once, or else the visual complexity of the transition would be overwhelming. To be specific, we consider a single deformation to be the problem of explicitly resizing the boundaries of k collective regions, out of a much larger number n atomic regions, and these multiple boundaries can be moved simultaneously. For example, resizing the bounds of a single collective rectangular area would require $k = 2$ deformations in each direction.

Here we present the first scalable algorithm for stretch and squish navigation through the composition of successive rectilinear deformations. The input to our algorithm is new absolute positions for the boundaries of k collective regions to resize. The output of our algorithm is new positions for a limited number q of the n region boundaries, where q is bounded by $k \log n$. We have implemented our algorithm in the PRISAD infrastructure [12], which shows that it provides real-time control of scenes containing millions of potentially deformable regions.

We discuss usage scenarios in Section 2, and related work in Section 3. We describe the hierarchical data structures used in our approach in Section 4, and present the features of our stretch and squish navigation in Section 5. Details of our algorithm are in Section 6, with results in Section 7. We conclude in Section 8.

2 USAGE SCENARIOS

Typical usage patterns of stretch and squish navigation increases the amount of room devoted to some items of interest. When we increase the room for these items, we need to decrease the available space, and perhaps the level of detail, for other items. Our system supports selection directly in screen space, and indirectly through filtering dataset
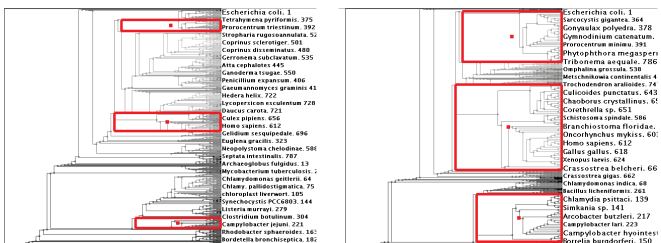

Fig. 3. **Left:** We can define several discontinuous regions of interest from search results, multiple region selection, or other grouping criteria. **Right:** We can stretch these multiple regions concurrently, to simultaneously inspect topologically distant features.

items of interest based on their attributes, such as search results or category membership. Figures 2a, 2c, and 2e show direct, user selected successive screen-space collective regions of interest formed through user actions of dragging out a rectangle with the cursor. Figures 2b, 2d, and 2f show how a user has directly resized their chosen regions of interest by dragging the selected collective region in upward or downward directions. In Figure 3 left, we show several collective regions selected, based on items that match a filtering criteria. Then, in Figure 3 right, we stretch all of these non-contiguous regions concurrently. Note that these regions are initially not the same size vertically, and our algorithm for concurrent region resizing can preserve the relative sizes of these regions. Furthermore, the sizes of regions between stretching regions shrink with identical rates to preserve navigation history throughout the dataset.

We provide a lightweight visual history through the composition of successive multiple deformations, as shown in Figure 2. Much of the previous work on showing history visually in visualization systems has been heavyweight, with a completely separate visual representation of history that is displayed in a second view. For example, Jankun-Kelly *et al.* [6] propose a graph of nodes representing previous system states, where each node shows a thumbnail image with a snapshot of the main view at that state. Switching from the main visualization dis-

play to the secondary history view, which shows a completely different scene, may impose a cognitive load. We argue that many tasks do not require seeing this full-fledged graphical history of the visualization process itself. Stretch and squish navigation provides lighter-weight visual cues showing navigation history integrated into the context of the main view. We conjecture that for many tasks, these implicit cues are sufficient in preserving user orientation without an explicit representation of past history.

3 RELATED WORK

Several early deformable-region navigation systems provide stretch and squish navigation for small datasets. For example, Document Lens [8] allows users to pick which single region should be stretched, with all other regions squished. In this simple navigation model, navigation history is not preserved visually.

Keahey *et al.* [7] and Carpendale *et al.* [3] propose navigation with deformable sheets using multiple foci or lenses. Nesting these lenses corresponds to composing deformations. However, these systems were designed for flexibility, supporting a variety of deformation types, rather than scalability. In contrast, we limit our system to rectilinear deformations, but our technique is highly scalable.

Rubber sheet navigation, which also supports composing successive deformations, was originally proposed by Sarkar *et al.* [9, 10]. Their navigation algorithm is linear in the number of reshapable regions, and requires each region to be at least one pixel wide. This technique is thus limited to hundreds of potentially deformable regions at best, whereas we handle millions. In addition to the rectilinear deformations that we support, they also allow polygonal stretching, where straight lines become piecewise-linear curves. Creating a scalable navigation algorithm to support the polygonal approach would be interesting future work.

Some visualization systems are designed for scalability, for example the MillionVis system [5] that handles one million items. However, their system does not support general navigation through deformation, only a specific set of transitions between layouts. Similarly, Tulip [1] handles large datasets, but does not support stretch and squish navigation.

Several recent publications deal with the problems of rendering with Accordion Drawing, but do not describe the underlying navigation algorithm [2, 12, 13]. Similarly, Slack *et al.* present SequenceJuxtaposer [14], a dataset-specific system that uses Accordion Drawing to compare gene sequences, but also do not describe a navigation algorithm [14]. In this work, we describe the features of our algorithm for efficient rubber sheet navigation. The split line data structure was originally introduced to accelerate rendering, whereas here we propose its use for navigation.

The paper on TreeJuxtaposer [4], which first introduces Accordion Drawing, does describe the need for a hierarchical data structure to support the deformation of regions, but provides only a hint of how to actually perform navigation tasks. The paper does not provide an algorithm, and we observed, through empirical investigation of the open-source implementation, that the implemented algorithm becomes numerically unstable with simultaneous deformations of multiple regions. Our new approach is generic rather than being tied to tree datasets, and supports robust deformations of multiple regions of interest. The navigation algorithm presented herein also forms part of Slack's unpublished Master's thesis [11].

4 SPLIT LINES

We base our navigation algorithm on deforming collective regions, as we illustrated in Section 2. We support deformation of rectilinear regions that are bounded by lines that stretch across the entire viewing area. Although the absolute positions of the lines change, deformations do not change the adjacency order of the lines. A rectangular deformable region is the intersection of four bounding lines: two horizontal and two vertical. The horizontal and vertical sets of lines are independently controllable, and we store these sets separately, as shown in Figure 4. If a user only moves a single line to a new position, then the absolute spatial position of every other line in its set could change

for a global effect. However, if we provide new positions for several lines as a simultaneous constraint, and set the new positions to match the old positions for some chosen anchor lines, we may constrain navigation to have a local effect.
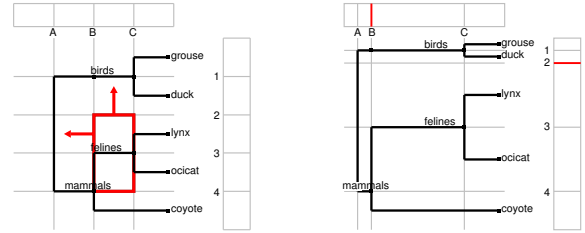

Fig. 4. The ordered list of horizontal lines is stored separately from the vertical lines. **Left:** A tree dataset with the number of deformable regions chosen such that each node can be resized independently. **Right:** Deformations change absolute spatial position of the lines, but not their adjacency ordering.

We define the *absolute position* of a split line to be a location between 0 and 1, the minimum and maximum bounds of the scene. A system that stores only absolute positions requires $O(n)$ updates to ensure each line is repositioned correctly for a proper visual history, since a single deformation action could change the absolute position of every line in the set. Since our navigation system supports architectures where only the p visible regions will need to be drawn, where $p << n$, our navigation system does not update and store the absolute positions of all n lines after each deformation. Our approach uses binary trees to manage the relative spatial ordering between regions, and we call the tree nodes *split lines*. They store the *split ratio*, a ratio between 0 and 1, of the size of the left child region and the size of the whole parent region.
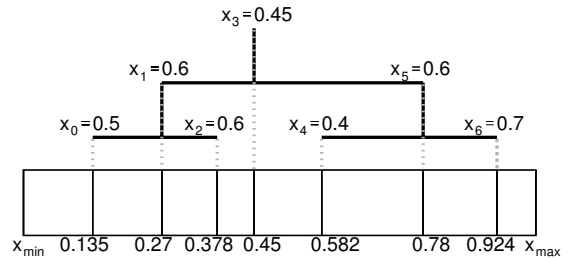

Fig. 5. The absolute positions of the sets of ordered split lines can be recovered from the split line hierarchies, which store split ratios between 0 and 1 at every node. The minimum and maximum split lines x_{min} and x_{max} cannot move, and are the boundaries of the visualization space.
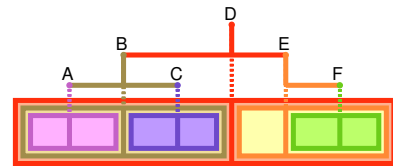

Fig. 6. In the split line hierarchy, each region is bounded by two ancestor split lines, and all descendants of a parent are spatially contained with its region. For example, the region defined as B is bounded by the minimum split line on the left and split line D, the parent region of B, on the right.

When two child regions are the same size, the split ratio stored by the parent is 0.5. We can compute the absolute position of any split line using these split ratios by traversing the path between the root and the line in the hierarchy. Figure 5 shows the relationship between these

ratios, such as $x_5 = 0.6$, and the absolute position of a split line, such as 0.78 for the position of x_5. To compute the absolute positions for x_4, for example, we use the split ratios and absolute positions of its ancestors in the hierarchy as follows:

$$
\begin{aligned}
x_4^A &= (x_5^A - x_3^A) \times x_4^R + x_3^A \\
&= (0.78 - 0.45) \times 0.4 + 0.45 \\
&= 0.582
\end{aligned}
$$

where an A superscript denotes an absolute position of a split line, and an R denotes a split ratio value for that split line. This computation is top-down, starting from the root, so the required absolute positions of the ancestors can be used in the calculations for their descendants. Thus, in a balanced split line hierarchy with n nodes, we can compute the absolute position of any split line in $O(log\ n)$ time.

The area affected by each split line is bounded by two other split lines, on the left and right[1], as shown in Figure 6. All descendants of a parent are spatially contained within its region. The left child node is bounded by its parent's left boundary on its left, and the parent itself on its right; the right child node is bounded by the parent on its left, and its parent's right boundary on its right.

Our infrastructure uses the split line hierarchies for storage and computation, but imposes very few constraints on end-user behavior. A user can select to stretch a collective region between any two split lines without regard to the underlying hierarchical structure, for example the region bounded by C and E in Figure 6. The regions bounded by the k lines that our navigation algorithm takes as input are collective regions. A single atomic region between two contiguous split lines is the smallest level of granularity that can be resized with the infrastructure. In typical applications, an on-screen rectangle dragged out by the user would be snapped to the enclosing split lines. In areas of the screen where the navigation state is zoomed out, this snapping distance would be smaller than one pixel and thus unnoticeable. In areas where the view is zoomed far in, the distance between the cursor and the nearest split line may cover many pixels, making the discretization of space directly visible to the user.

5 FEATURES

Our approach has three features that assist developers in designing appropriate applications for stretch and squish visualizations with visual history. We remove the need for users to explicitly place handles to control deformations, successive deformations compose with previous ones, and deformations can affect only local regions.

5.1 Removing explicit handle creation requirement

Early rubber sheet navigation systems required the user to explicitly manage *handles* to denote the regions that should be deformable [9]. Since the number of handles supported was very limited, in the dozens or hundreds, users wanting to explore large datasets would need to regularly delete previously used handles before being able to add more handles to achieve a new navigation state. Because our approach scales to millions of deformable regions, adding every handle is an automatic operation that occurs once at startup time. An application developer determines the desired granularity of control that applies to the family of datasets to deform, so the application automatically assigns enough split lines to allow immediate resizing of any potentially interesting dataset features. For example, TreeJuxtaposer allows the inspection of tree topology down to the level of individual nodes, and SequenceJuxtaposer has a nucleotide as the smallest resizeable unit.

5.2 Composing successive deformations

Our navigation strategy is to compose deformations rather than simply replace the old view of a scene with a new one. Fundamentally, this feature requires storing more information than would be needed with a replacement strategy. To achieve our desired scalability, we must minimize both our storage for navigation state, and our costs for updating

---

[1]We use left and right in our explanations for clarity, but our arguments apply symmetrically to top and bottom cases for the vertical hierarchy.

a deformed scene. Our architecture is general enough that it can also support the traditional approach of replacing an old view with a new one, as required for the common panning and zooming interfaces.

5.3 Supporting local-only deformation

Our algorithm supports moving multiple lines simultaneously, which means that our approach supports local-only, as well as global, deformations. For instance, users may wish to cause a deformation only in a limited region of the dataset, which preserves their previously selected regions of interest. Our algorithm allows some regions to be deformed while others remain unchanged by explicity defining the final absolute location of a split line to be equal to its initial absolute location: these split lines act as anchors. We may move split lines, subject to being constrained between these anchors, when any initial and final split line locations differ. Figure 7 categorizes three regions of interest in all deformation functions: regions that stretch to allocate more space for their objects; regions that squish to create space for other regions of interest; and regions that are invariant in size between deformation actions.
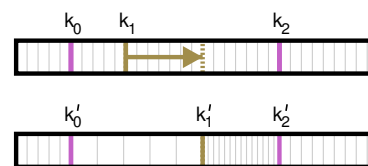

Fig. 7. Local vs. global deformation. For local-only deformation, the set of lines to move includes stationary lines that act as anchors. Here, we want to move three split lines at positions k_0, k_1, and k_2. The two split lines at k_0 and k_2 are stationary, to prevent deformations from modifying the regions to the left of k_0, and to the right of k_2. **Top:** In this example, before deformation, all lines are uniformly distributed through the space. **Bottom:** After deformation, the movement of k_1 to k_1' caused stretching in the region between k_0' and k_1', and shrinking in the region between k_1' and k_2'.

5.4 Example

Figure 8 shows an example of our approach with sequence data, where a user wishes to stretch one region but preserve some of their previous navigation actions. Previous approaches would have required explicitly placed anchors and handles for stretching, but our approach allows this local operation with implicitly placed anchors and handles. Applications that use our navigation approach give users a consistent model with sufficient flexibility to visualize single items in a large dataset, but do not require users to have extensive knowledge of the mechanics of their interactions. The final navigation state reflects previous navigation actions in addition to the latest one, preserving the relative sizes of Column2 and Column3. Finally, the size of Column4 is maintained, with the navigation action affecting only the local regions of Column1 through Column3.

6 ALGORITHMS

We break down our navigation algorithm into five functions, as shown in Figure 9. A setup function, **moveSplitLines**, calls the core recursive **resize** function, which in turn uses the three functions **partition**, **interpolate**, and **getRatio**. In this section we discuss these functions, then provide examples of the algorithm in action on small and large split line hierarchies.

6.1 MoveSplitLines

The **moveSplitLines** navigation algorithm takes as input K, an ordered list of split lines that have final absolute positions. The algorithm output is Q, a list of split lines with updated final split ratios. Both K and Q are subsets of N, the set of all split lines. The primary function of this algorithm is to initialize the first call to a recursive function, **resize**, which processes the hierarchy of split lines to construct Q. These

	Column1	Column2	Column3	Column4
Sloth	A	T	A	G
Hyrax	C	T	A	C
Rabbit	A	T	A	A

0.3	0.2	0.2	0.3

0.5	0.1	0.1	0.3

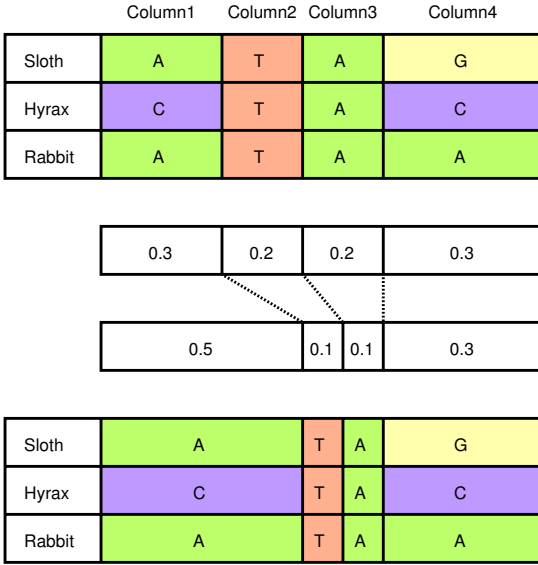|	Column1	Column2	Column3	Column4
Sloth	A	T	A	G
Hyrax	C	T	A	C
Rabbit	A	T	A	A

Fig. 8. Our navigation technique allows users to explore multiple localized regions of interest using deformations, without destroying previous navigations. For example, if a user wishes to stretch Column1, initially 30% of the display, to 50% but keep Column4 at 30%, they may do so by shrinking Column2 and Column3. The user achieves this effect by pinning the boundary line between Column3 and Column4, and moving the boundary line between Column1 and Column2 to the right. Our algorithm to perform this navigation keeps the sizes of Column2 and Column3 relative to each other, which is important for preserving the visual history of previous navigations.

$|Q|$ split ratios are used to deform the grid structure to properly align our initial $|K|$ split lines to their desired final locations. The lines in K are always a subset of the lines in Q, because moving a split line in K to a new absolute position requires updating the split ratios of it and all its ancestors in the split line hierarchy. The upper bound on $|Q|$ is $|K|\log|N|$, since there are at most this many ancestors for the $|K|$ initially selected split lines with final absolute locations.

6.2 Resize

The **resize** function is the main component of our deformation algorithm. This recursive function operates on a split line S, and the set of split lines, M, that are in the bounds of S that we wish to move. The **resize** function is first called by **moveSplitLines** with the root of the split line hierarchy, and the full list K of lines to move. Figure 10 shows the core structure: each step calls **partition** to split M into a left and right set, uses **interpolate** to find the new absolute position of one split line S using the relative distances between its neighbors in M, and uses **getRatio** to compute the new split ratio of S. Finally, it recurses on the left and right partitioned subsets.

The function terminates when M contains only the two bounds of S, which indicates there are no more lines to move in the subtree under S in the hierarchy. In this case we know no split line between these bounds must be processed, since these split lines must keep their initial split ratios. The maximum number of calls to resize is $|Q|+2|K|$, and the amount of work done in each recursive step is constant. The algorithm therefore has an overall time complexity of $O(|K|\log|N|)$.

6.3 Partition

The **partition** function finds the two lines in M which S falls between, which we call L (left) and R (right), and uses them to split M into two sets. The size of M decreases with recursive calls to **resize**. For clarity, we present the intuition behind partitioning in Figure 9, Algorithm 3, which is $O(|M|)$. In practice, we use an array indexing technique that performs a binary search for S in M. This search has a near-constant cost for each call to **resize** for the typical case of a small $|K|$.

Algorithm 1: **moveSplitLines**

Input: list K of split lines with absolute final positions
Output: list Q of split lines with final split ratios
$S \leftarrow$ **getHierarchyRoot**()
$M \leftarrow \{S.leftBound, K, S.rightBound\}$
$Q \leftarrow \emptyset$
resize(S, M, Q)
return Q

Algorithm 2: **resize**

Input: split line S, set of split lines M with absolute final positions, set of split lines Q with final split ratios
if $|M| = 2$ **then**
    return
end
$(L, R) \leftarrow$ **partition**(S, M)
if $S \notin M$ **then**
    $S.f \leftarrow$ **interpolate**(**secondLast**$(L), S, $**second**$(R))$
end
$S.fRatio \leftarrow$ **getRatio**(**first**$(M).f, S.f, $**last**$(M).f)$
$Q.add(S)$
resize$(S.$**leftChild**$, L, Q)$
resize$(S.$**rightChild**$, R, Q)$

Algorithm 3: **partition**

Input: split line S, list of split lines K
Output: two lists of split lines: L and R
$L \leftarrow \emptyset$
$R \leftarrow \emptyset$
$R.add(S)$
foreach $m \in K$ **do**
    if $m.$**leftOf**(S) **then**
        $L.add(m)$
    end
    if $m.$**rightOf**(S) **then**
        $R.add(m)$
    end
end
$L.add(S)$
return (L, R)

Algorithm 4: **interpolate**

Input: split line L, split line S, split line R
Output: final absolute position of S
$InitRel \leftarrow$ **getRatio**$(L.i, S.i, R.i)$
return $InitRel \times (R.f - L.f) + L.f$

Algorithm 5: **getRatio**

Input: absolute values $left$, $center$, and $right$
Output: split ratio of $center$ between $left$ and $right$
return $\frac{center - left}{right - left}$

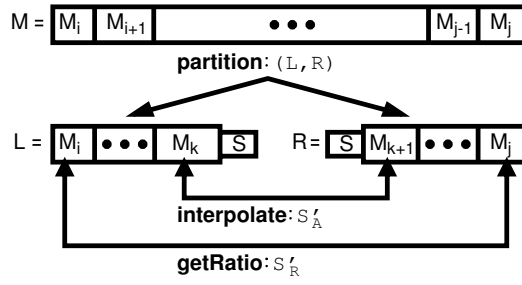Fig. 9. Pseudocode for the five functions used in navigation.

Fig. 10. The **resize** function first partitions the set M into two sets L and R at M_k and M_{k+1}, the lines in M which surround S. The **interpolate** function finds S'_A, the new absolute position of S, using M_k and M_{k+1}. The new split ratio S'_R is then computed with the **getRatio** function using M_i and M_j, the bounds of S.

6.4 Interpolate

The **interpolate** function finds the new absolute position of S given its adjacent split lines L and R in M. It uses the property that the relative position of S between L and R must match for the initial and final absolute positions, as shown in Figure 11. This property guarantees that the spacing in the collective region between these split lines is preserved. If S is already an element of M, then there is no need to interpolate because its final absolute position is already known.
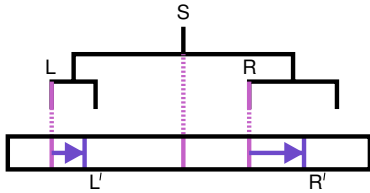

Fig. 11. The **interpolate** function computes the final position S' given the input of S, L, R, L', and R'. The ratio of the initial absolute positions $\frac{S-L}{R-L}$ matches that of the final absolute positions $\frac{S'-L'}{R'-L'}$ in order to preserve spacing in collective regions. The lines L and R are the neighbors, not the bounds, of S in the subset M of updated lines.

6.5 GetRatio

The **getRatio** function computes the ratio between three unique split lines given their absolute positions. When called from **interpolate**, it computes the ratio between S and its neighboring split lines L and R in M, an intermediate value used to compute the final position of S. The absolute positions of L and R are known, either because they were specified in K as input or were computed in a previous recursive call to **resize**. When called from **resize**, **getRatio** yields the split ratio for S between its bounds, and this value is stored in the split line hierarchy. The final positions of these bounds are always in M. At the beginning, we seed M with the final positions of the minimum and maximum bounds for the root of the entire split line hierarchy; that is, 0.0 and 1.0. Whenever a split line is added to M, its final position has just been calculated.

6.6 Examples

We provide two examples of our deformation algorithm. Figure 12 shows a small example with $k = 2$ and $n = 6$, where we show the four recursive calls to **resize** to compute the new absolute positions and split ratios of the $q = 4$ ancestors of A and E. Figure 13 shows a large example with $k = 2$ and $n = 2047$, where the difference between $|Q| <= |K| \log |N|$ and $|N|$ becomes substantial. Again, we consider only the horizontal case; the vertical case is analogous.
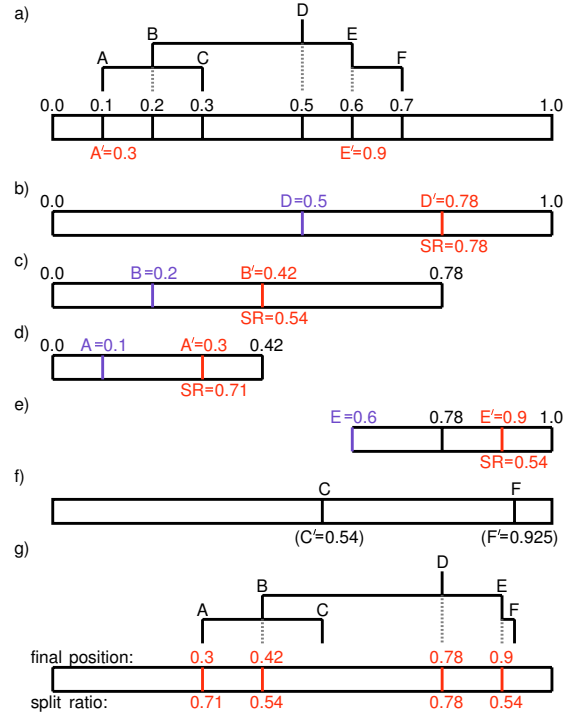

Fig. 12. Small recursive navigation example. Our deformation function moves a selected number of split lines to final locations, but does not require moving each split line individually. **a)** In this example ($k = 2$, $q = 4$, $n = 6$), we wish to move 2 split lines: A from 0.1 to 0.3, and E from 0.6 to 0.9. **b)** Starting with the hierarchy root D, we compute $D' = 0.78$ as its final absolute location based on A, A', D, E, and E'. D must be in the same relative position between two moving split lines to preserve relative distances in the deformation region $[A, E]$. **c)** Recursing on the left to B, we compute 0.42 as its final absolute location based on A and D, with similar constraints as before. **d)** A does not require an interpolation between the minimum boundary and B since we are given 0.3 as its final absolute position. **e)** Moving E to 0.9 does require interpolation, as before. **f)** The positions of C and F are not computed until required by the rendering algorithm, since they do not change their split ratios. **g)** From the final absolute positions of the $q = 4$ split lines, we can compute their split ratios. No split ratio updates are required for any other lines.
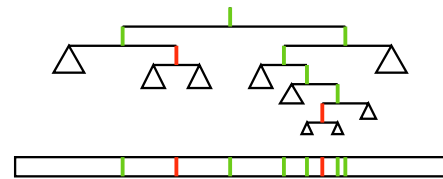

Fig. 13. Large recursive navigation example. Here ($k = 2, q = 8, n = 2047$), so the $q << n$ property holds: most nodes do not require split ratio updates. To move the $k = 2$ split lines marked in red to desired locations, only the $q = 8$ split lines marked in green that fall on the paths between the original k and the root need their split ratios updated. We represent subtrees that are not traversed, where no split ratio updates are needed, as triangles. Along the bottom, only the final absolute positions of the $q = 8$ lines are drawn, not those of the 2039 lines in the untraversed subtrees.

7 RESULTS

Our navigation algorithm as presented is generic in the sense of not making any assumptions about the characteristics of the underlying
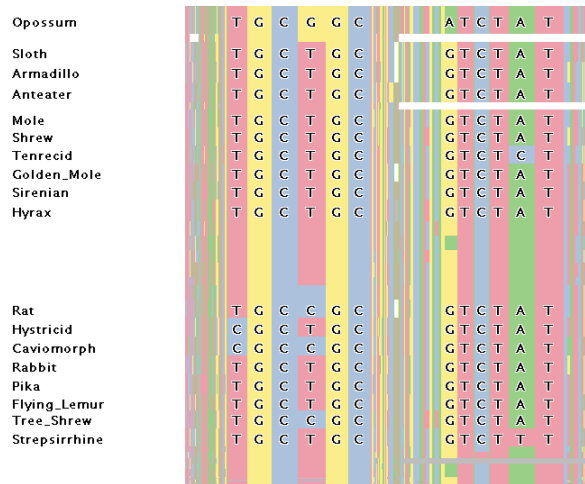
Fig. 14. After a series of deformations, a sequence dataset of $740,000$ objects has four stretched-out regions of interest. Navigations in this dataset are performed in under 1 millisecond, negligible when compared to the 30 milliseconds or more for rendering time, so navigation is not be the bottleneck in the interactive system.

dataset that is being stretched and squished. We have implemented a version of it in the PRISAD infrastructure [12], and have tested it with applications for tree and gene sequence visual comparison. Figures 2 and 3 show a series of navigations, both with direct manipulation and with indirect actions, on a tree dataset of several thousand nodes. Figure 14 shows a series of deformations on a sequence dataset of $740,000$ objects.

We tested our navigation algorithm on a tree of 4 million nodes, with 2 million split lines on the horizontal axis, and 21 split lines on the vertical axis. The navigation time was typically less than one millisecond. In contrast, the time to render, as described in [12], was approximately 40 milliseconds. Similarly, a gene sequence dataset of 40 million objects required 6400 by 6400 split lines. Again, the navigation time was negligible compared to the rendering time. Our navigation algorithm has fulfilled its design requirement: it is not the bottleneck of the interactive system.

8 CONCLUSIONS AND FUTURE WORK

We have presented the first scalable navigation algorithm that supports the composition of successive rectangular deformations, providing a lightweight visual history. When moving k collective regions out of n possible atomic regions, its complexity is $O(k \log n)$. Our approach handles the simultaneous resizing of multiple collective regions, and thus can support restricting a deformation to a local region simply by specifying anchor lines with the same initial and final positions. While previous work supported only a small number of deformable regions, and required users to explicitly add and delete handles to demarcate movable regions, our approach allows application developers to specify the desired granularity of control for all possible moveable regions as an automatic operation at startup time. Our implementation of the algorithm within the PRISAD framework supports applications for interactive exploration of tree and gene sequence datasets of millions of items, with the navigation calculations taking under a millisecond.

Our approach is highly scalable, but only supports rectilinear deformations. An interesting area of future study would be to handle fisheye and other radial effects that can be specified in a piecewise-linear fashion, as described by Carpendale *et al.* [3]. Localizing these non-rectilinear distortions between bounding anchor lines might allow us to capitalize on our current architecture.

REFERENCES

[1] D. Auber. Tulip - a huge graph visualization framework. In P. Mutzel and M. Jünger, editors, *Graph Drawing Software*, Mathematics and Visualization series, pages 105–126. Springer-Verlag, 2003.

[2] D. Beermann, T. Munzner, and G. Humphreys. Scalable, robust visualization of very large trees. In *Proc. Eurographics/IEEE Symposium on Visualization (EuroVis 05)*, pages 37–44, 2005.

[3] M. S. T. Carpendale, D. J. Cowperthwaite, and F. D. Fracchia. 3-dimensional pliable surfaces: for the effective presentation of visual information. In *Proc. ACM Symposium on User Interface and Software Technology (UIST 95)*, pages 217–226, 1995.

[4] T. M. *et al.* TreeJuxtaposer: Scalable tree comparison using Focus+Context with guaranteed visibility. *ACM Trans. on Graphics (Proc. SIGGRAPH 2003)*, 22(3):453–462, 2003.

[5] J.-D. Fekete and C. Plaisant. Interactive information visualization of a million items. In *Proc. IEEE Symposium on Information Visualization (InfoVis 02)*, pages 117–124, 2002.

[6] T. J. Jankun-Kelly, K.-L. Ma, and M. Gertz. A model for the visualization exploration process. In *Proc. IEEE Visualization (Vis 02)*, pages 323–330, 2002.

[7] T. A. Keahey and E. L. Robertson. Nonlinear magnification fields. In *Proc. IEEE Symposium on Information Visualization (InfoVis 97)*, pages 51–58, 1997.

[8] G. G. Robertson and J. D. Mackinlay. The document lens. In *Proc. ACM Symposium on User Interface Software and Technology (UIST 93)*, pages 101–108, 1993.

[9] M. Sarkar and S. P. Reiss. Manipulating screen space with StretchTools: Visualizing large structure on small screen. Technical Report CS-92-42, Department of Computer Science, Brown University, Sept. 1992.

[10] M. Sarkar, S. S. Snibbe, O. J. Tversky, and S. P. Reiss. Stretching the rubber sheet: a metaphor for viewing large layouts on small screens. In *Proc. ACM Symposium on User Interface Software and Technology (UIST 93)*, pages 81–91. ACM Press, 1993.

[11] J. Slack. A partitioned rendering infrastructure for stable accordion navigation. Master's thesis, University of British Columbia Department of Computer Science, 2005.

[12] J. Slack, K. Hildebrand, and T. Munzner. PRISAD: A partitioned rendering infrastructure for scalable accordion drawing. In *Proc. IEEE Symposium on Information Visualization (InfoVis 05)*, pages 41–48, 2005.

[13] J. Slack, K. Hildebrand, and T. Munzner. PRISAD: A partitioned rendering infrastructure for scalable accordion drawing (extended version). *Information Visualization*, 5(2):137–151, 2006.

[14] J. Slack, K. Hildebrand, T. Munzner, and K. S. John. SequenceJuxtaposer: Fluid navigation for large-scale sequence comparison in context. *Proc. German Conference on Bioinformatics*, pages 37–42, 2004.