Stretching the Rubber Sheet: A Metaphor for Viewing Large Layouts on Small Screens

Manojit Sarkar, Scott S. Snibbe, Oren J. Tversky, and Steven P. Reiss
Department of Computer Science
Brown University, Providence, RI 02912–1910 USA
{ms,sss,ojt,spr}@cs.brown.edu

Abstract

We propose the metaphor of rubber sheet stretching for viewing large and complex layouts within small display areas. Imagine the original 2D layout on a rubber sheet. Users can select and enlarge different areas of the sheet by holding and stretching it with a set of special tools called *handles*. As the user stretches an area, a greater level of detail is displayed there. The technique has some additional desirable features such as areas specified as arbitrary closed polygons, multiple regions of interest, and uniform scaling inside the stretched regions.

Keywords: Information Visualization, Graphical Visualization, Interface Metaphors, Interactive Systems

Introduction

Imagine working with a large and complex layout. Large layouts, diagrams, illustrations, and pictures occur in many areas of Computer Science, Engineering, Architecture, and Arts. Such a layout cannot be displayed in full detail on a typical computer display. Since the screen has a fixed resolution, scaling the entire layout *uniformly* to fit into the available space does not allow full detail to be shown. The most common solution provides a scrollable viewport. This shows full detail at the region currently visible through the viewport, but hides the rest of the layout. As a result, users often get lost and feel deprived of context. In certain situations, availability of adequate context may be crucial, or may improve the usability of the application significantly.

One common strategy to show detail and context in layouts uses two separate views. One view contains an overview of the entire layout. The details are usually too small to be seen in this scale-reduced version. Another view shows a *detailed* view of a selected region. The overview window may indicate the location of the selected region within the entire layout by a small box or a point. This simple technique is very effective in many situations. It allows one to view the entire structure as well as the detail at the region of interest. It is much easier to navigate in such a system. The technique, however, does not integrate detail and context in a single view.

An alternative to the above strategy is to distort the layout to display both detail and context in a single view. A number of techniques relying on this strategy have been invented in last few years. Generally these techniques allow users to specify some information item of current interest, show the specified item in detail, and provide context by displaying the remaining items in successively less detail. In the following section we review two such techniques.

Related Research

In 1986, Furnas proposed the concept of a Fisheye Lens [4]. In photography, a fisheye lens is a very wide-angle lens. It shows the nearby regions in great detail while showing surrounding regions in successively less detail. The software analog to a fisheye lens shows local detail and global context in one view. In 1992, Sarkar and Brown provided a concrete graphical interpretation to Fisheye Views by building a prototype browser for 2D layouts [7]. The browser allows a user to specify a focal point in a layout. It then expands the focal region, and correspondingly contracts the other regions. Users can browse a layout by clicking and dragging. The system keeps enlarging the regions near the foci in real time. It animates the display smoothly as the user changes focus.

A fisheye view allows visualization of a large layout in a single view. It smoothly integrates detail and context.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1993 ACM 0-89791-628-X/93/0011...$1.50

Apple Inc. v. Smith Interface Techs., LLC
IPR2024-01119, -01116, -01115 | Smith EX2011 | Page 1 of 11

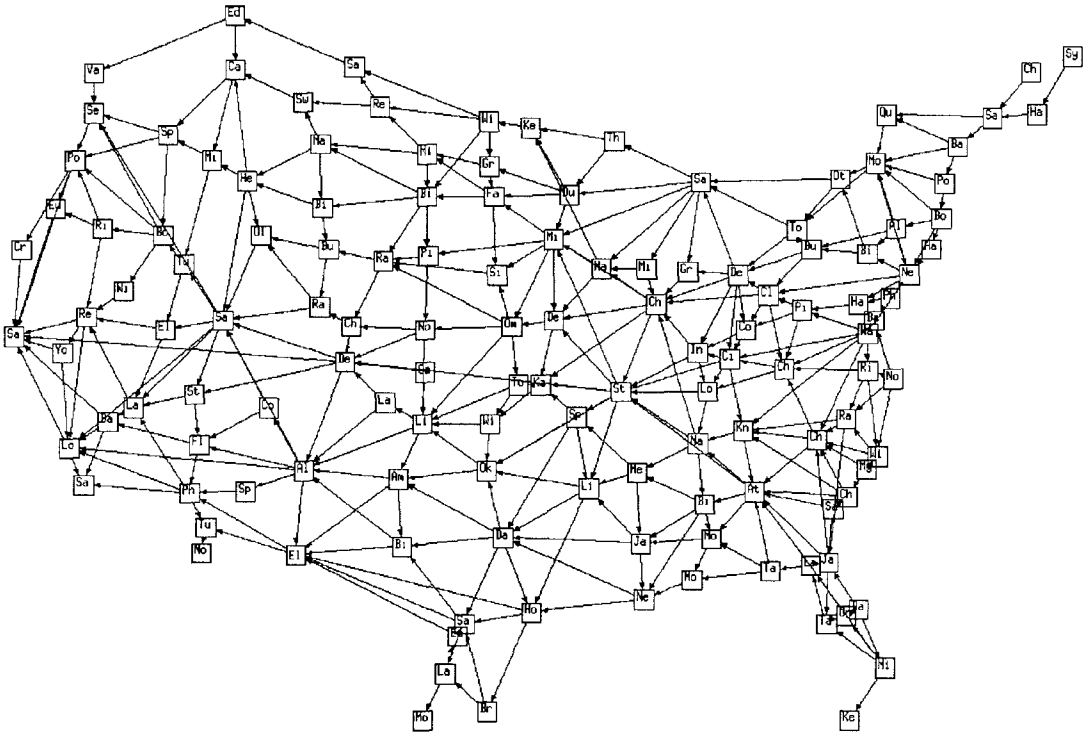

Figure 1: A graph layout with 134 nodes and 338 links. The nodes represent major cities in the United States and the links represent paths between neighboring cities.

Users are allowed to control the degree of distortion. The technique however has several drawbacks. A focus is always a point or a single information item; the system cannot treat an arbitrary region or an arbitrary set of information items as the focus. More importantly, users are not allowed to control the amount of space allocated to the focus. The system indirectly infers this from a specified distortion factor.

In 1991, Mackinlay, Robertson and Card developed a technique called the Perspective Wall for viewing large information bases ordered along a single dimension [6]. Typical examples of such information are project records ordered by chronology and directory entries ordered alphabetically. Because their lengths are much larger than their widths, such structures result in 2D layouts of wide aspect ratios. Their technique folds a 2D layout into a 3D wall. The wall has a panel in the center for viewing details, and two perspective panels on either side for viewing context. The specified item of interest is always moved to the center panel where the user can view it in detail.

Perspective Wall displays the entire layout in a single view. It shows detail and preserves relative distances at the focus and integrates the detail smoothly with the rest of the layout. It also solves the problem of inefficient utilization of screen space resulting from the wide aspect ratios. The technique, however, can deal only with information bases ordered along a single dimension. We have developed techniques similar in spirit to the Perspective Wall for dealing with arbitrary 2D layouts.

Desirable Features

After studying existing techniques and experimenting with several layouts, we feel the following features are very desirable for techniques that allow visualization of large layouts within limited display areas. Our stretching technique offers all of these features to a reasonable degree.

Allow exact specification of focus: Users like to be able to pick the exact items of interest. In the case of a graphical layout, we would like to specify the size, shape and location of focal regions. Ideally, the shape could be any arbitrary closed region, such as a country's border. Arbitrary convex polygonal regions provide reasonably good approximations.

Provide uniform scaling at focus: Uniform scaling preserves angles, proportionality in distances, and parallelism between lines. Certain types of information becomes useless if these properties are not preserved, for example road maps, and floor plans. For layouts in which connectivity is the only important property,

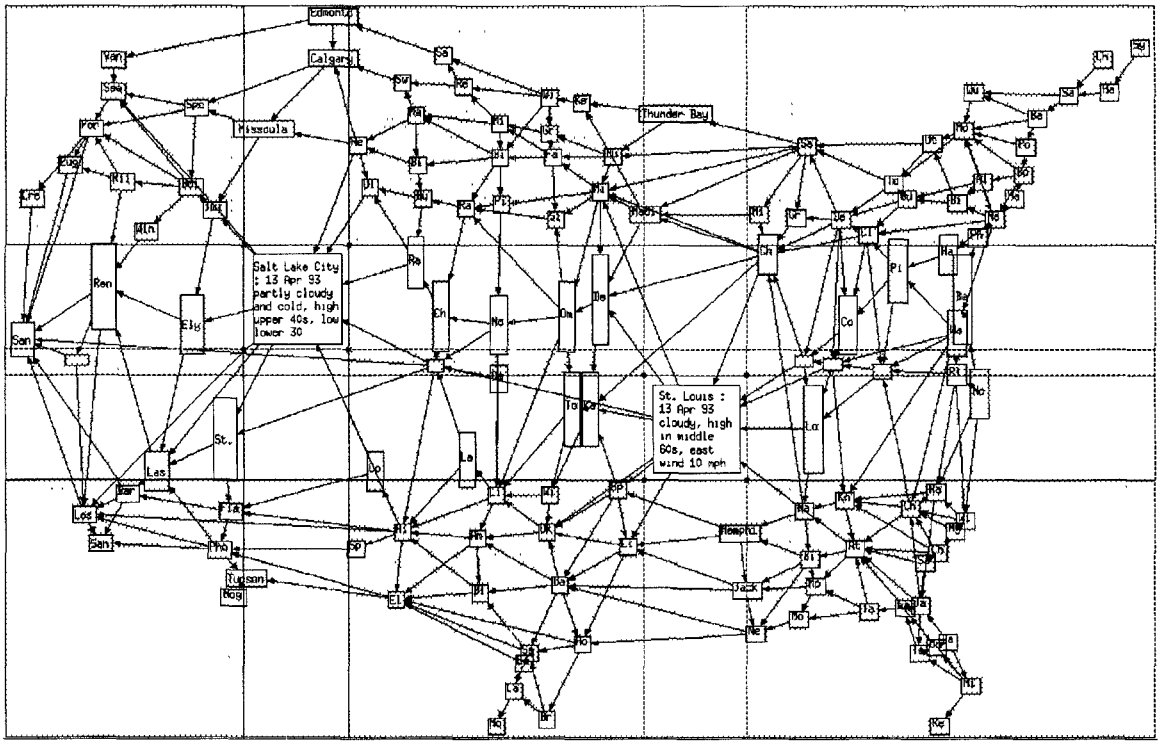

Figure 2: A view of the layout in Figure 1 after it has been stretched orthogonally to view details at St. Louis and Salt Lake City. The straight lines on the layout are the handles used to stretch the layout.

however, uniform scaling is not as important and may artificially constrain the user. A good solution should allow both uniform and non-uniform scaling of focus.

Show context: The display should preserve context around the focus. It is desirable that the user can always see the global structure of the layout.

Integrate focus and context: The level of detail should be a smooth function of distance from the focus. This provides a smooth integration of the focus and its context.

Allow precise space allocation: The user should be able to directly control the amount of space allocated to the focus. This allows the user to enlarge the focus until the required level of detail is achieved.

Multiple foci: The user may want to simultaneously view two or more distant regions of a layout, while preserving the information between and surrounding them. Therefore a complete visualization system should allow multiple foci.

Preserve overall shape: Maintaining the overall shape of the layout reduces the disorienting effect due to the introduced distortion. Eades, Lai, Misue and Sugiyama [3], mention three primary properties which should be preserved by transformations in order to preserve the user's "mental map" of the layout. The properties are *orthogonal ordering*, *clusters* and *topology*. Orthogonal ordering can be simply thought of as the "compass direction" between two points. Consider points p and q in a layout. If p is northeast of q in the undistorted view, the distorted view should preserve this relationship. Clustering requires that points which are *close together* in the normal view should also be close together in the distorted view. The topology requirement is that the transformation from undistorted to distorted view is a homeomorphism, in particular, the inside of each closed continuous curve is mapped to the inside of a closed continuous curve. These properties are more rigorously defined in the above paper.

Allow editing: It would be a significant advantage to be able to edit a region when it is enlarged. Many of the problems with large layouts involve losing context not only while viewing, but also while making changes to it. It may be enough to provide editing capability only at the focus, but it would be better to be able to edit the entire layout in its distorted view.

Stretching Techniques

We experimented with two techniques. Both techniques use the same metaphor of rubber sheet stretching, but employ different shapes of handles and different styles

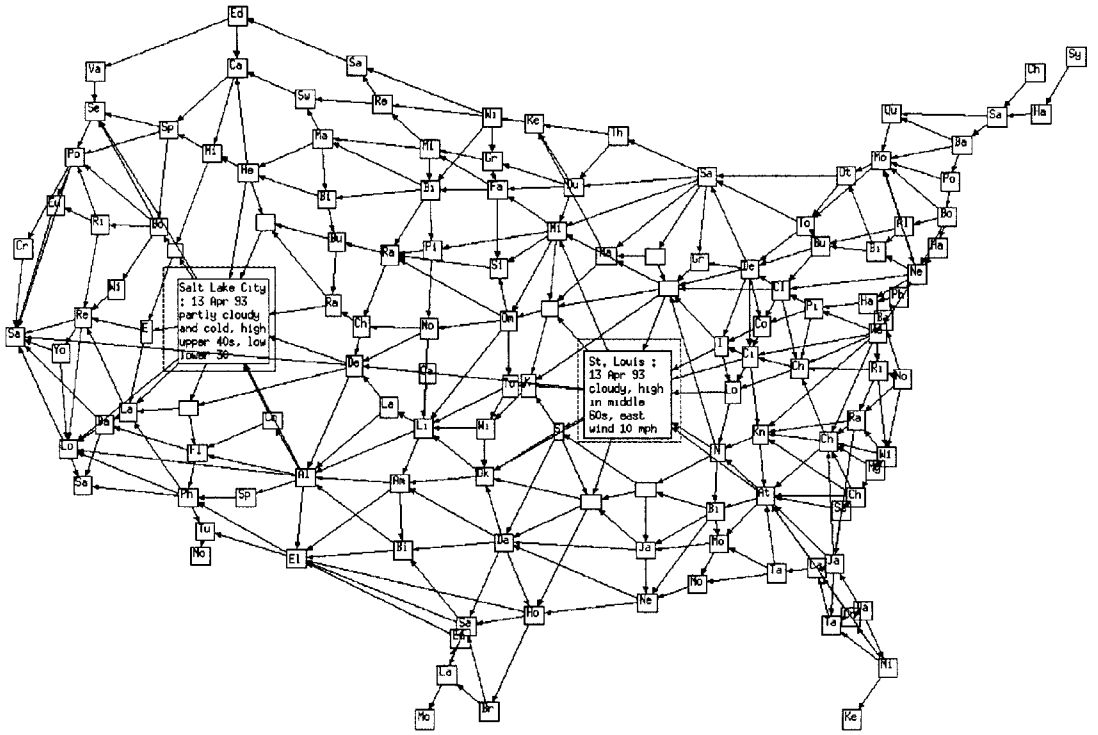

Figure 3: Another view of the layout in Figure 1 after it has been stretched with two rectangular handles shown around St. Louis and Salt Lake City. Compare this with Figure 2.

of stretching. The first technique allows only orthogonal stretching by long horizontal and vertical bar handles which span the entire layout. The second technique employs polygonal handles, and allows selection and stretching of polygonal regions. We have decided to describe both techniques, concentrating mostly on the later, because each has its own merits, and neither is subsumed by the other. Readers can find more detailed description of the former technique in [8].

Orthogonal Stretching

Figures 1 and 2 together illustrate the orthogonal stretching technique. Figure 1 shows a large graph layout with 134 nodes and 338 links. The nodes represent the major US cities, and the links represent the routes connecting the cities. Figure 2 shows a stretched version of this layout. It shows four horizontal and four vertical lines placed on top of the layout. These lines are the *handles* used to stretch the screen. These are called *bar* handles (to contrast them with the *polygonal* handles described in the following section). As a bar handle is placed on the screen and pulled, the screen expands on one side and contracts on the other. The system automatically scales up the layout uniformly in the expanded region, and scales it down uniformly in the contracted region. The regions are also shaded to reflect their stretch factors. In the resulting view the user can clearly read the information associated with St. Louis and Salt Lake City.

Each region of the screen is surrounded by four handles, two horizontal, and two vertical. Associated with each such region are a *horizontal stretch factor* and a *vertical stretch factor*. The horizontal stretch factor is the ratio of the stretched width to the original width of the region; vertical stretch factor is defined similarly. Each point in the region is mapped to a stretched point by scaling operations using the stretch factors as scale factors.

The technique provides *clamps* to preserve and view multiple enlarged regions. A horizontal handle and a vertical handle can be clamped together at their intersecting point. Conceptually, clamping can be thought of as driving a nail through the handles for the purpose of joining them. Figure 2 shows four clamps at the four handle-intersections around St. Louis. These clamps cordon off the region containing St. Louis preventing further expansion or contraction.

Orthogonal stretching allows users to pick a rectangular region and to allocate space to it interactively. It allows multiple foci, provides uniform scaling at foci, preserves orthogonal ordering of points, and allows editing.

Some undesirable features of orthogonal stretching are also apparent in Figure 2. For example, there is no way

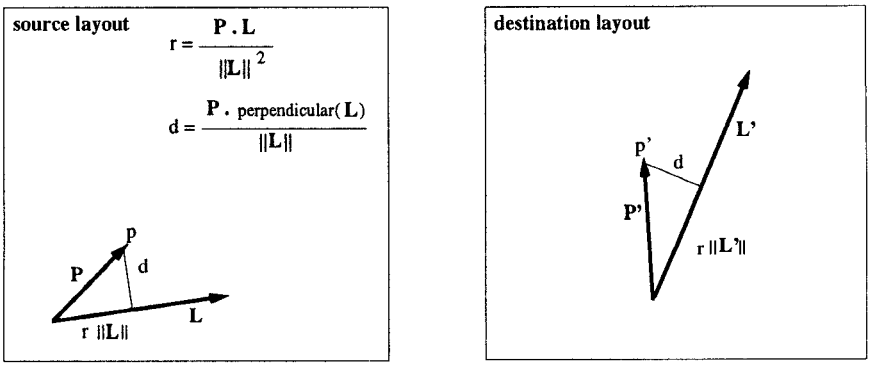

Figure 4: Mapping source point p to destination point p' using the single vector-pair $(\mathbf{L}, \mathbf{L}')$

to enlarge a single region without affecting an entire row or column of regions. In Figure 2, the node labeled Thunder Bay was also enlarged in the horizontal direction although we intended to enlarge only St. Louis. The technique also doesn't enlarge the regions near the focus automatically. The nearby regions are left just as small as the remote regions. This is because the technique scales all the regions outside the focus uniformly. It is possible to enlarge the nearby regions by placing additional handles. But it would be better to have it done automatically. A big disadvantage of orthogonal stretching is that there are discontinuities in the scale factor at the region boundaries. As a result it fails to integrate the individual regions smoothly into a coherent layout. We have developed the technique described in the following sections to overcome some of these problems.

Polygonal Stretching

This technique allows the user to specify a polygonal region as focus. User can specify the region either by explicitly drawing a polygon, or by specifying a set of information items. In the latter case the system can generate a rectangle by computing the bounding box or a convex polygon by computing the convex hull. The polygon acts as the *handle* for stretching. Enlarging the polygon enlarges the region inside it. The system scales the part of the layout inside the handle, and adjusts the rest of the layout to integrate it smoothly with the enlarged portion. Figure 3 shows the result. It shows a version of the graph layout in Figure 1 stretched with two rectangular handles. Compare this figure with Figure 2.

The algorithm for transforming a source layout to a stretched layout (henceforth called *destination layout*) is based on a technique used by Beier and Neely for image transformation [1]. A single pair of vectors, one corresponding to the source layout and another corresponding to the destination layout, defines a mapping between the two layouts. This can be extended to multiple pairs of vectors by weighting the pairs based on their *length* and *distance* from the source points. We generate vector pairs from the handles, and use the multiple-pair algorithm to transform our layouts. First we review the single-pair algorithm.

Mapping with Single Vector-Pair

Figure 4 illustrates the technique for mapping a source point p to a destination point p' using a single pair of vectors $(\mathbf{L}, \mathbf{L}')$. The box on the left represents the source layout, and the box on the right represents the destination. The vectors $\mathbf{L}$ and $\mathbf{L}'$ have been placed in their respective layouts. The vector $\mathbf{P}$ in the source is the vector from the tail of $\mathbf{L}$ to p. The vector $\mathbf{P}'$ is defined similarly. We derive $\mathbf{P}'$ using the following equation:

$$\mathbf{P}' = \frac{\mathbf{P} \cdot \mathbf{L}}{||\mathbf{L}||^2}\mathbf{L}' + \frac{\mathbf{P} \cdot \mathit{perpendicular}(\mathbf{L})}{||\mathbf{L}||} \frac{\mathit{perpendicular}(\mathbf{L}')}{||\mathbf{L}'||}$$

where the function $\mathit{perpendicular}(\mathbf{L})$ returns a vector which is of the same magnitude as $\mathbf{L}$, but perpendicular to $\mathbf{L}$. (There are two perpendicular vectors, at angles $+90^{\circ}$ and -90°. Either of them can be used as long as it is used consistently throughout.) The fraction $r = \frac{\mathbf{P} \cdot \mathbf{L}}{||\mathbf{L}||^2}$ is the length of the projection of $\mathbf{P}$ on $\mathbf{L}$ normalized by the length of $\mathbf{L}$. The fraction $d = \frac{\mathbf{P}\ \mathit{perpendicular}(\mathbf{L})}{||\mathbf{L}||}$ is the unnormalized length of the projection of $\mathbf{P}$ on $\mathit{perpendicular}(\mathbf{L})$.

It is possible to translate and rotate points using a single pair of vectors. All the transformations expressible by a single pair of vectors are affine, but not all affine transformations are expressible. Scaling is possible along the direction of the vectors, but it is not possible to specify uniform scaling and shearing. At least two pairs of vectors are required to specify uniform scaling. Mapping

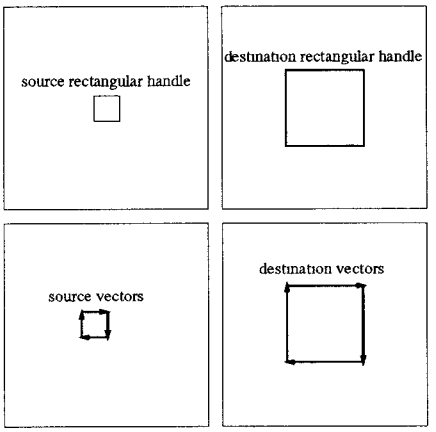

Figure 5: The source and destination of a rectangle handle, and its corresponding vector-pairs

with multiple pairs of vectors is discussed next.

Mapping with Multiple Vector-Pairs

Our system allows multiple handles to be used simultaneously, and each handle may contribute more than one pair of vectors. We therefore need the multiple-pair algorithm to transform our layout.

Suppose we want to map p to p' with n pairs of vectors $(\mathbf{L}_i, \mathbf{L}_i{}'), i = 1, 2, \ldots, n$. A pair $(\mathbf{L}_i, \mathbf{L}_i{}')$, for some i, contributes a displacement of $(\mathbf{P}_i{}' - \mathbf{P})$ to p. The combined displacement for p is computed by taking a weighted sum of these displacements, and dividing by the sum of the weights. The weight of the pair $(\mathbf{L}_i, \mathbf{L}_i{}')$ depends on the *length* of $\mathbf{L}_i$ and the *distance* of p from $\mathbf{L}_i$ according to the following function:

$$\mathbf{P}' = \mathbf{P} + \frac{\sum_{i=1}^{n} \frac{\|\mathbf{L}_i\|^a}{(distance(p, \mathbf{L}_i))^b} (\mathbf{P}_i{}' - \mathbf{P})}{\sum_{i=1}^{n} \frac{\|\mathbf{L}_i\|^a}{(distance(p, \mathbf{L}_i))^b}}$$

where $distance(p, \mathbf{L}_i)$ returns the distance from p to $\mathbf{L}_i$. The fraction $\frac{\|\mathbf{L}_i\|^a}{(distance(p, \mathbf{L}_i))^b}$ is the weight of the pair $(\mathbf{L}_i, \mathbf{L}_i{}')$, where a and b are two constants. When $distance(p, \mathbf{L}_i) = 0$ for some i, we use only the displacement contributed by $(\mathbf{L}_i, \mathbf{L}_i{}')$, and ignore the contributions of the others.

Implementing Polygonal Handles

A polygonal handle with k sides is implemented by k pairs of vectors. Each side of the initial polygon contributes a source vector, while each side of the stretched polygon contributes a destination vector. The length and the direction of the vector are derived from the *length* and the *slope* of the side respectively. Figure 5 shows how rectangular handles are implemented. A circular handle is approximated by a sixteen sided polygon.

Mapping Algorithms

We use a set four vectors to define a *border* for our layouts. Each side of the border contributes a pair of vectors. The source and the destination vector within each pair are identical. This implies that the destination border coincides with the source border. If the border vectors are given enough weight, no portion of the destination layout will go outside the border.

Mapping Points

The algorithm to map a source point p to a destination point is shown below as the procedure *mapPoint*(). If p falls in a region *enclosed* by a handle, it is mapped by the enclosing handle alone. This ensures that the part of the layout inside a handle are not affected by other handles. Each handle class is responsible for implementing its own *mapPoint*() method. The rectangular and the circular handles scale the region inside themselves, so their *mapPoint*() methods perform a linear scale operation.

The algorithm enhances the *weight* of the border by a factor approximately equal to the number of handles $||H||$ to prevent the destination layout from going out of the border. This however does not provide a guarantee. An extremely high degree of stretching may still cause the layout to go outside the border. This problem is solved by recomputing the world-to-window map as explained later in the algorithm for transforming the entire layout.

```
proc mapPoint(Point p, HandleSet border, HandleSet H,
  Float a, Float b)
begin
  foreach (h ∈ H) do
    if (h.encloses(p))
      return h.mapPoint(p, a, b);
    endif
  endfor
  dispSum = (0, 0);
  weightSum = 0;
  H.contribute(p, a, b, disp, weight);
  dispSum = dispSum + disp × weight;
  weightSum = weightSum + weight;
  border.contribute(p, a, b, disp, weight);
  dispSum = dispSum + disp × weight × (1 + ||H||);
  weightSum = weightSum + weight ×(1 + ||H||);
  return (p + dispSum/weightSum);
endproc
```

Mapping Handles

We allow the simultaneous presence of multiple handles. Stretching one handle doesn't alter the size of the other handles, but it may cause the other handles to move. Such incremental displacements of the other handles due to the stretching of the currently selected handle are computed by the procedure *mapHandle*() shown below. Each handle saves its last most recently used destination vectors. When a handle is stretched, its current destination vectors and last most recently used destination vectors are used to compute the incremental displacements. The method *contributeIncrement*() computes the contribution of the stretched handle towards the incremental displacements. The algorithm computes the displacement of the *center* of each of the other handles and translate the handles by the computed displacements. If a handle lacks a well defined center, we can compute the displacement of each of its vertices and use the average of these displacements as the displacement for the entire handle.

```
proc mapHandles (HandleSet border, HandleSet H,
  StretchedHandle s, Float a, Float b)
begin
foreach (h ∈ H) do
  if (h ≠ s)
    dispSum = (0,0);
    weightSum = 0;
    s.contributeIncrement(h.center, a, b, disp, weight);
    dispSum = dispSum + disp × weight;
    weightSum = weightSum + weight;
    border.contribute(h.center, a, b, disp, weight);
    dispSum = dispSum + disp × weight × (1 + ||H||);
    weightSum = weightSum + weight × (1 + ||H||);
    h.translate (dispSum / weightSum);
  endif
endfor
endproc
```

Mapping Entire Layout

The entire layout is mapped by transforming each of its nodes and links. Nodes and links compute their transformations by mapping the required number of source points by using the *mapPoint*() procedure. For example, in our current implementation a rectangular node maps itself by mapping points at its northwest and southeast corners.

A layout is specified in world coordinates, and we maintain the usual world-to-window map in a *canvas* object to render the layout on a window. The bounding box of the layout constitutes the world. The variable c of type Canvas in the procedure *mapLayout*() below maintains the world-to-window map. Straight forward application of the multiple-vector-pair algorithm does not guarantee that the bounding box of the entire layout will remain constant in size. In fact under a high degree of stretching the bounding box of the transformed layout may become significantly larger. It may also become smaller if handles are used to contract the layout. In procedure *mapLayout*(), the method *transform*() invokes the straightforward multiple-vector-pair mapping algorithm. The resulting bounding box of the layout becomes the new world for the canvas. Users however prefer that the size of the handles remain unaffected by the change in the world-to-window map. We therefore scale back the handles to their previous size. Scaling necessitates reapplication of the *transform*() method, and so on. This process is repeated till the size of bounding box is within acceptable range.

```
proc mapLayout(HandleSet border, HandleSet H,
  Float a, Float b, Layout l, Canvas c)
begin
  repeat
    l.transform(border, H, a, b);
    if (l.bbox > c.world or l.bbox ≪ c.world)
      sizeOkay := false;
      s1 := c.scale;
      c.setWorld(l.bbox);
      s2 := c.scale;
      foreach (h ∈ H) do
        h.scale(s1/s2);
      endfor
    else
      sizeOkay := true
    endif
  until(sizeOkay = true)
endProc
```

Inverse Mapping

Providing editing in the destination layouts necessitates mapping destination points to source points. Our transformation technique does not have a general inverse mapping. We implemented inverse mapping function for regions inside rectangular and circular handles. Since these handles map the source points inside them by scale operations, we can compute the corresponding source point for a given destination point by a scale operation. We use this technique to allow editing in the focus regions.

Placing new handles on the destination layout also requires us to infer its corresponding source positions to derive the source vectors. In this case, users pick a set of information items in the destination layout. Since both source and destination position and size of the information items are known, the system computes the bounding box (convex hull) of the selected items to generate a rectangle (polygon) handle.

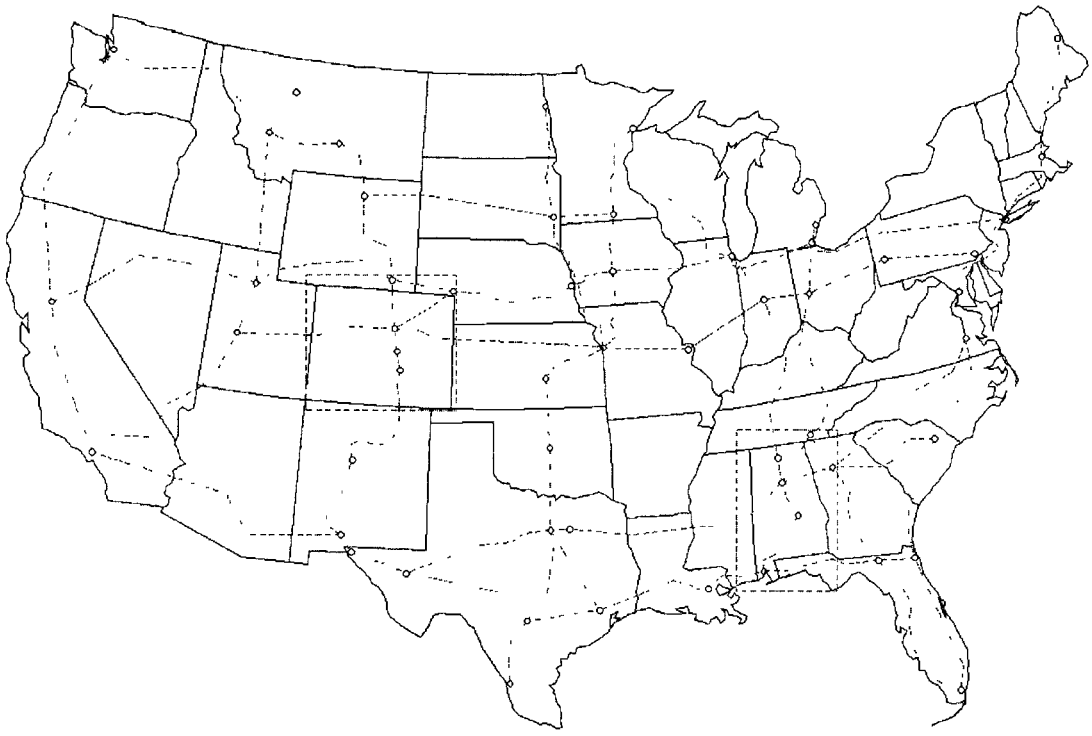

Figure 6: An outline of United States and some major highway routes. Small circles indicate cities.

Structured Layouts

Hierarchical abstraction is a useful technique for managing and navigating through large volumes of information [2]. It clusters related information nodes, and creates abstract higher level nodes to represent each cluster. These higher level nodes are again clustered together to create further higher level nodes.

We have integrated techniques of hierarchical decomposition and stretching for viewing structured layouts. A hierarchy can be represented by a tree. The *root* node of the tree contains the most abstract information, and *leaf* nodes contain the most concrete information. The children of a node constitute the cluster represented by that node.

The stretching allows the user to control the amount of space allocated to any information node. Each node computes the amount of detail displayed within itself. The amount of detail is a function of the available space. An internal node displays its children if there is sufficient space to do so; otherwise it displays itself. Figures 6 and 7 demonstrate the effectiveness of this technique. Figure 6 shows an outline of United States, and some major highways. Associated with each state, such as Colorado and Alabama, is an invisible bounding box node. The children nodes of these bounding box nodes are the city and highway information nodes. As the user stretches the states of Colorado and Alabama, more space gets allocated to the bounding boxes associated them. The city names and highway numbers can now be displayed. The highway routes have been drawn into the state outlines using this technique with our own prototype editor based on stretching.

We classify internal nodes into two categories, AND nodes and OR nodes. An AND node displays its children if *all* of its children can be displayed. An OR node displays its children if at least one of its children can be displayed. An OR node is useful when a user is interested in viewing a node in detail, but there is not enough space in the window to display all of its siblings.

The algorithm for computing detail starts at the root of each hierarchy. Each parent node decides if it is necessary for its children to compute their detail. If a parent node doesn't have enough space to be present in the view, it is automatically inferred that its children are not present in the view. This recursive top-down algorithm requires the minimum number of *mapPoint*() operations to transform a layout.

Speed of Transformations

Our focus in this paper is in viewing large and complex layouts. It is therefore particularly important to examine the speed of our algorithms in order to evaluate the potential for real time response and for allowing

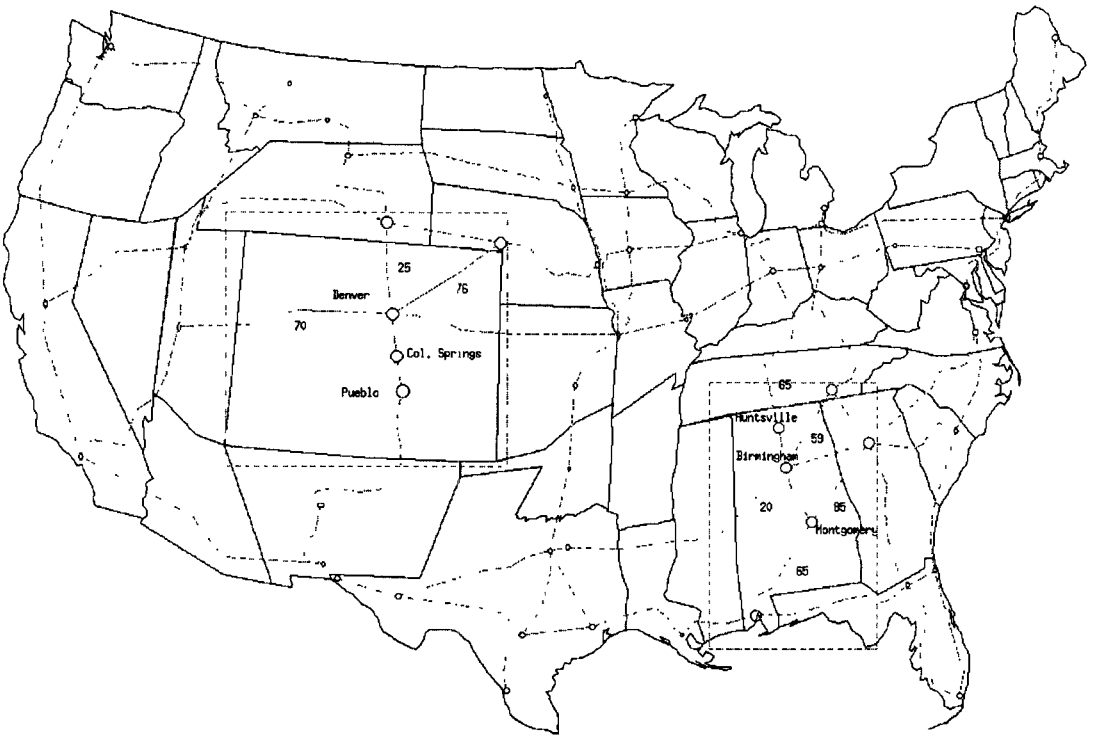

Figure 7: Stretched view of the outline of United States as in Figure 6. It now shows the city names and highway numbers in Colorado and Alabama.

animation between transitions.

The entire transformation process has two stages, handle mapping and layout mapping. Let m and n be the number of handles, and the number of points which need to be mapped in order to map the layout respectively. Almost always m is very small compared to n. Therefore mapping handles takes a negligible portion of the total time required for one complete transformation.

In case of orthogonal stretching, the complexity of layout mapping is $O(n)$ simple scale operations. But in the case of stretching with polygonal handles, layout mapping takes approximately $O(nmk)$ single-vector-pair mapping operations for each *iteration* in the *mapLayout*() procedure, where k is the average number of vector-pairs per handle. We have verified experimentally that it is rarely necessary to perform more than one iteration.

Our prototype viewer is implemented with **C++**, *X*, and *Motif*. It runs on SparcStation-10/41GX with 32Mb main memory and no special graphics hardware. During our informal study, we observed that our viewer provides real-time response for graph layouts of up to few hundred nodess and a similar number of links. As expected, viewing based orthogonal stretching is somewhat faster than that based on polygonal stretching.

Evaluation of Techniques

We believe stretching has several advantages over the existing techniques. It has an intuitive interface. It can respond in real-time which facilitates smooth animation during transitions between views. Stretching with polygonal handles provides focal regions of arbitrary polygonal shapes. It also provides very smooth integration of focus and context. On the other hand, orthogonal stretching preserves orthogonal ordering of points, a property not shared by polygonal stretching. Both techniques allow multiple regions of interest, and provide uniform-scaling at foci to preserve important spatial properties for viewing maps, floor plans and other spatially sensitive layouts. Precise editing within foci is also achieved by both.

Figure 8 provides a contrast between polygonal stretching and orthogonal stretching. Polygonal stretching relies on the strategy of combining contributions from many handles. This strategy produces good results when the inter-handle distances are relatively large and degrees of stretching relatively low. The middle view of Figure 8 shows what happens when the degree of stretching is moderately high, and the distances between the border handles and the rectangle handle are moderately low. The nodes 1,5 and 8,5 have become extremely small. Further stretching of the rectangle handle will reduce their sizes to zero, and may eventually cause

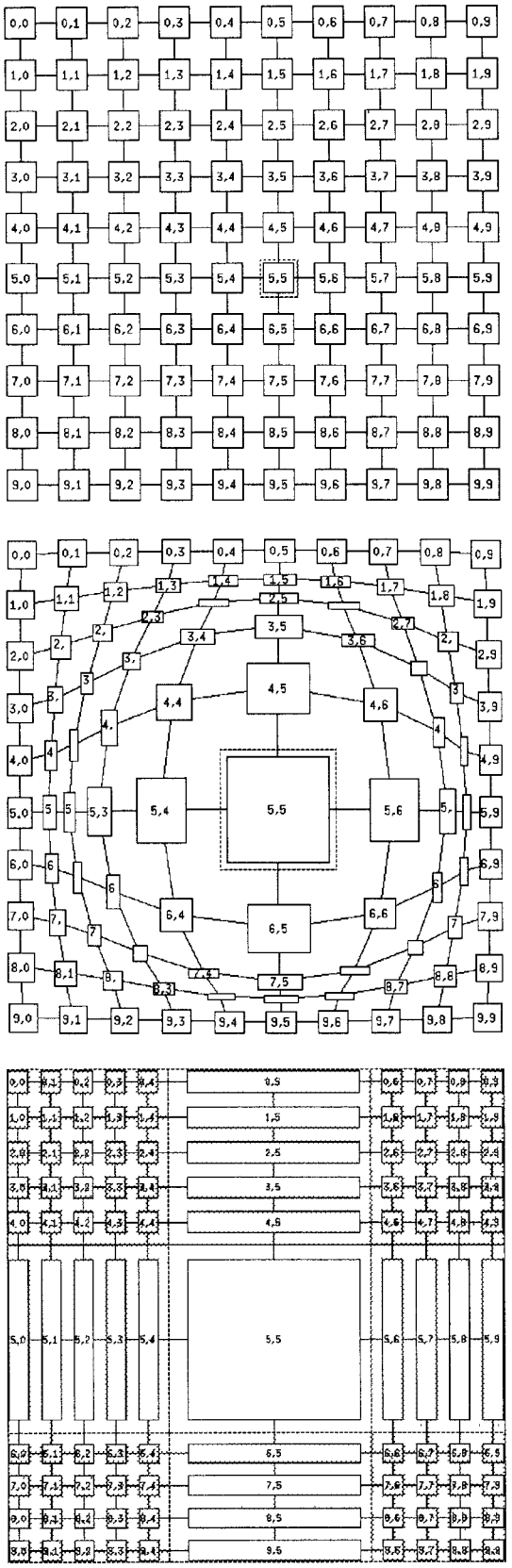

Figure 8: A symmetric layout, its view stretched with a rectangle handle and its view stretched with four bar handles. Orthogonal stretching can stretch to any degree without destroying order and symmetry.

their nodes to change relative orders along **X** and **Y** dimensions. Our prototype however does not allow the user to stretch handles to a degree which may cause any node size to become zero. Contrast this with the lower most view in the same Figure. It shows that orthogonal stretching does not have such anomalous behavior, the user can stretch node 5,5 to any degree without destroying the orthogonal ordering of the points.

Figure 8 also illustrates how orthogonal stretching can preserve symmetry in certain layouts, no matter how much stretching is applied. This suggests that orthogonal stretching may be very appropriate for layouts with nodes arranged in rows and columns such as in Spread Sheets, floor plans, and circuit layouts. Orthogonal stretching also has a general inverse mapping function which allows unrestricted editing capability. These properties make orthogonal stretching the more robust of the two techniques.

Future Work

Even though orthogonal stretching is more robust, polygonal stretching provides smoother integration of detail and context. It would be nice to be able to find a technique which is both robust, and provides smooth integration. Considering only one dimension, the problem can be stated as follows: find a shape-preserving higher order continuous curve passing through $4 + 2(n - 1)$ arbitrary points, and having arbitrary derivatives at $2 + 2(n - 1)$ arbitrary points, where $n \geq 1$. Here n is the number of polygonal handles, and we assume there is at least one handle on the screen. The values of the derivatives provide the scale factors at the handle boundaries. Figure 9 illustrates the problem for a single handle. Piece-wise curves such splines as discussed in [5] are not appropriate here because change in two control points and derivatives at two points of a spline (corresponding to a change in position and size of one handle) affects the spline only locally. We need a global change in the curve for global space adjustments and achieve smooth integration of the entire layout.

Orthogonal stretching achieves such global adjustment. It effectively uses piece-wise linear curves (line segments) with discontinuities at join points. Each region is therefore uniformly scaled with discontinuities at the region boundaries. This discontinuous curve is however monotone, and therefore preserves order of points.

As another important improvement to our technique, we would like to allow overlapping handles, so that users can enlarge a region which is itself inside another enlarged region. Currently, polygonal handles are not allowed to overlap with each other.

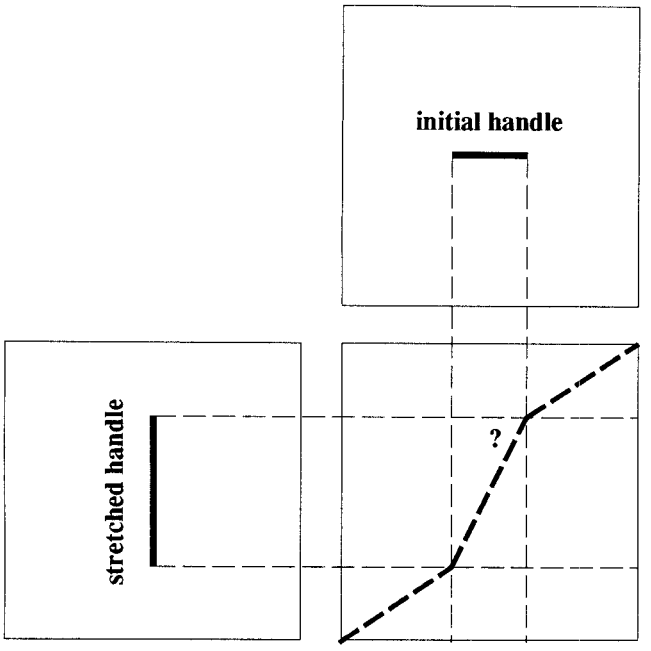

Figure 9: Mapping source layout to destination layout entails determination of smooth curve through given points and having given derivatives at given points.

Finally, we would like to perform user studies to determine if these techniques indeed result in improved user performance. The studies could be performed with a MacDraw like drawing program, giving the users specific goals for creating and editing drawings, as well as locating various features within the drawing.

Conclusions

We believe that stretching can find several possible real world applications. As mentioned earlier, this technique lends itself well to visualizing architectural plans and geographic maps. Figure 7, where we stretched Colorado and Alabama to look at the cities and highways, demonstrates one possible scenario. Further zooming might eventually lead one to one's own street and home. Information kiosks, navigation systems, VLSI circuit design tools may all find applications for this technique.

Acknowledgements

The authors would like to thank the UIST'93 reviewers for suggesting many improvements to the technical aspects as well as the presentation of this paper. Support for this research was provided by NSF grants CCR9111507 and CCR9113226, by ARPA order 8225 and by ONR grant N00014-91-J-4052.

References

[1] Thaddeus Beier, and Shawn Neely. Feature-based image metamorphosis. *Proc. ACM SIGGRAPH*, Published as *Computer Graphics, vol. 26, no. 2*, pp. 35—41, 1992.

[2] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The Information Visualizer, an information workspace. *Proc. ACM SIGCHI Conf. on Human Factors in Computing Systems*, pp. 189—194, 1991.

[3] Peter Eades, Wei Lai, Kazuo Misue, and Kozo Sugiyama. Preserving the mental map of a diagram. Research Report IIAS—RR—91—16E, *International Institute for Advanced Study of Social Information Science, Fujitsu Laboratories Limited*, 1991.

[4] George W. Furnas. Generalized fisheye views. *Proc. ACM SIGCHI Conf. on Human Factors in Computing Systems*, pp. 16—23, 1986.

[5] T. N. T. Goodman, and K. Unsworth. Shape-preserving interpolation by parametrically defined curves. *SIAM J. Numerical Analysis, vol. 25, no. 6*, pp. 1453—1465, December, 1988.

[6] Jock D. Mackinlay, George G. Robertson, and Stuart K. Card. The perspective wall: Detail and context smoothly integrated. *Proc. ACM SIGCHI Conf. on Human Factors in Computing Systems*, pp. 173—179, 1991.

[7] Manojit Sarkar, and Marc H. Brown. Graphical Fisheye Views of Graphs. *Proc. ACM SIGCHI Conf. on Human Factors in Computing Systems*, pp. 83—91, 1992.

[8] Manojit Sarkar, and Steven P. Reiss. Manipulating Screen Space with *StretchTools*: Visualizing Large Structure on Small Screen. Technical Report CS—92—42, *Department of Computer Science, Brown University*, Providence, RI 02912, USA, 1992.