Manipulating Screen Space with *StretchTools*: Visualizing Large Structure on Small Screen

Manojit Sarkar and Steven P. Reiss

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-42
September 1992

Manipulating Screen Space with *StretchTools*: Visualizing Large Structure on Small Screen*

Manojit Sarkar
Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI 02912-1910 USA

Abstract

StretchTools is a set of *tools* for manipulating the screen space. The user interface uses the metaphor of stretching a rubber sheet. The basic tool in *StretchTools* is the *handle*. A handle is an one-dimensional rigid object of a fixed orientation, either horizontal or vertical. As a handle is placed on the screen, engaged, and pulled, the screen expands on one side of the handle and contracts on the other side. Graphical objects on the screen transforms accordingly. The other type of basic tool is the *clamp*. Clamps are used to impose constraints on the movement of handles by joining handles with one another. In this paper, we show how *StretchTools* can be used to visualize large structures on small screens by selectively enlarging single or multiple regions of the screen at a time. Transformations achieved by *StretchTools* have a number of properties that alleviate the problem of user-disorientation due to the introduced distortion.

Keywords: Information Visualization, Graphical Visualization, Interface Metaphors, Fisheye Views

1 Introduction

There is not enough space on a typical computer screen to layout a large graphical structure in its full size and detail. Even so large graphical structures occur with increasing frequency in many application areas as the computer's processing units become increasingly more powerful. One of the challenges of human interface design is to overcome this spatial bounds of small, limited resolution screens.

There has been a number of innovative interface techniques designed in last few years for visualizing large information bases on small screens. The unifying theme of all the techniques is the same – allocate adequate space to the objects of current interest, sometimes called the *focus*, and maintain a sense of context with the rest of the space.

As early as in 1982, Spence and Apperley developed a technique called the Bifocal Display for visualizing office information [9]. Briefly, in their system the workspace is a set of information items positioned in a horizontal strip. The display is a combination of a detailed view of the strip and two distorted views, where items on either side of the detailed view are squashed horizontally into narrow vertical strips. In a CHI '86 paper, Furnas pioneered the Fisheye Lens [4] which shows the nearby regions of the focus in detail while still showing remote regions in successively less detail. Later at CHI '92, Sarkar and Brown generalized the concept and demonstrated a prototype Fisheye Lens browser for large graphs [8]. Earlier at CHI '91, Mackinlay, Robertson and Card developed a technique called the Perspective Wall which uses 3D interactive animation for visualizing linear information [5]. Perspective Wall has a center panel for detail and two side panels receding into the distance for context.

In this paper, we present a novel technique of directly manipulating the screen space. The user interface uses the metaphor of stretching a rubber sheet. The basic stretching tool is the *handle*. The graphical structures are laid out on the screen in usual manner. As a handle is placed on the screen, engaged and pulled, the screen expands on one side of the handle, and contracts on the other side. The layout on the screen transforms accordingly. *StretchTools* allows users to allocate space to different parts of the structure with great precision.

StretchTools has been designed and developed using object-oriented technology. The subsequent sections describe *StretchTools* using the object-oriented terminology. However readers unfamiliar with object-oriented terminology should also follow the description without much difficulty.

*Support for this research was provided by NSF grants CCR9111507 and CCR9113226, by ARPA order 8225 and by ONR grant N00014-91-J-4052

return value	operation	arguments
	Place	position
	Remove	
	Engage	
	Disengage	
	Fix	
	UnFix	
	Clamp	handle
	Unclamp	handle
	Move	amount
position	Position	
displacement	Displacement	
	Draw	window

Table 1: Messages accepted by *handle* object

return value	operation	arguments
handle	HorHandle	
handle	VerHandle	
	Draw	window

Table 2: Messages accepted by *clamp* object

2 StretchTools

StretchTools is a set of *tools* for direct manipulation of screen space. The user interface uses the metaphor of stretching a rubber sheet. Graphical structures are laid out on the screen in usual manner. For large structures the details are typically too small or too little for good visualization. Users can visualize details by enlarging parts of the layout by stretching the screen. The screen is stretched by first placing a *handle* on it, and then engaging and pulling the *handle*. The layout constantly recomputes and redisplays itself in real-time. Each part of the layout computes its details based on the space available for itself.

Handles

The basic tool in *StretchTools* is the *handle*. Conceptually, handles are rigid one-dimensional objects of infinite length. A handle can be either vertically or horizontally oriented, but it must always maintain its orientation.

Some handles and a graph layout are shown in Figure 1. The graph has 100 vertices arranged in 10 rows and 10 columns. It also has 180 edges. The label shown inside each vertex is its detail. Vertical handles are represented by vertical lines extending from the north edge to the south edge of the screen. Similarly horizontal handles are represented by horizontal lines extending from the east edge to the west edge of the screen. Each horizontal handle therefore intersects all the vertical handles and vice versa.

Table 1 shows the set of messages accepted by handle objects. Each handle has a *Position* attribute which describes its current position. No two handles of the same orientation can be at the same position, and there must be a minimum separation between them. For example, in our prototype system at least a single pixel must separate any two handles of the same orientation. The *Displacement* attribute specifies the displacement the handle has undergone since it has been engaged to the screen. It is also possible to *Fix* an handle, fixed handles do not move when pulled. In our prototype system, the four handles along the north, east, south and west edge of the screen are made fixed when the system is initialized.

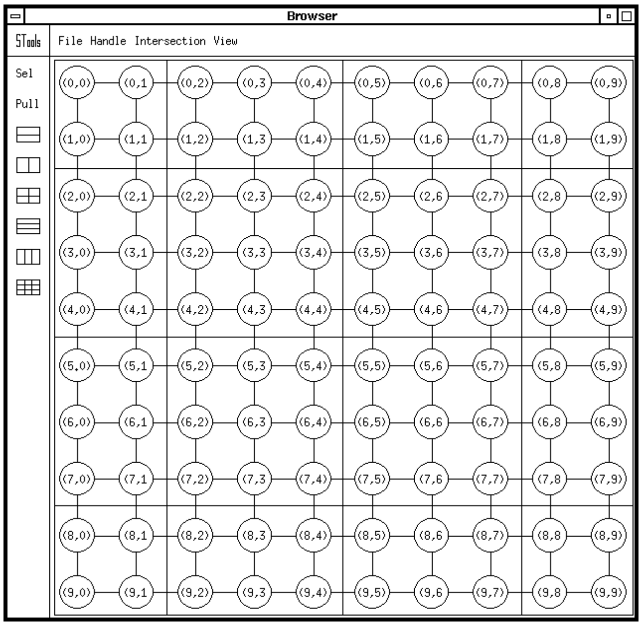

Figure 1: Initial screen

Pulling Handles

Handles can be pulled. Pulling sends a *Move* message to the handle. Each handle has a maximum allowable range for movement. Vertical handles may only move towards east or west, and horizontal handles may only move towards north or south. A *Move* message causes the handle to move by the smaller of the requested and the allowable amount. For example, a *Move* message has no effect on a fixed handle. Moving one handle may cause many other handles to move, either because they are directly or indirectly clamped to this handle or because they are engaged to the expanding or contracting part of the screen.

Figures 2 and 3 show of effect of pulling on handle on the layout and other handles in Figure 1. Handle movement algorithms are discussed in Section 6. Since a handle maintains its orientation, and extends from one edge to the other edge of the screen, the handle's movement uniformly affects the entire breadth or length of the screen depending on whether the handle is horizontal or vertical. This design choice was made to

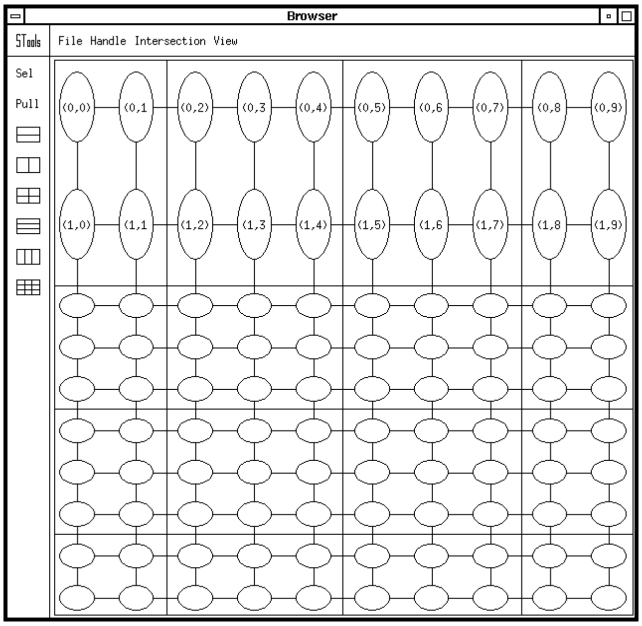

Figure 2: Vertical stretching

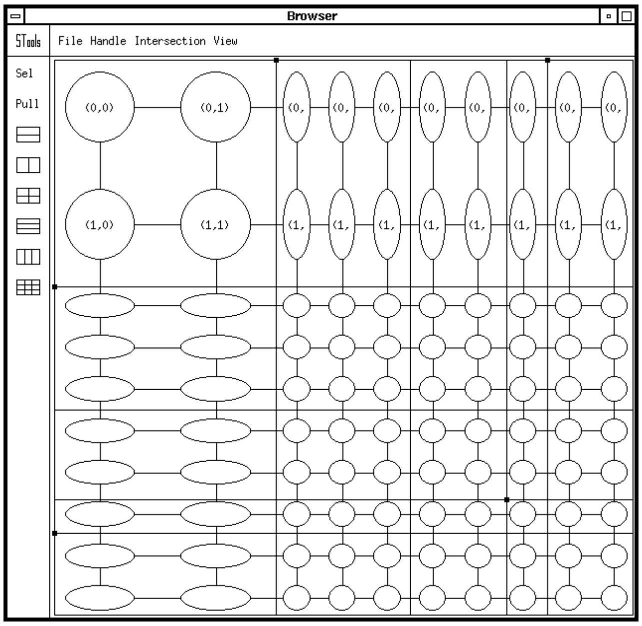

Figure 3: Horizontal stretching

preserve certain desirable properties of the layouts as they transform under manipulation. These properties are discussed in Section 5.

Clamps

A vertical handle can be clamped with a horizontal handle and vice versa. Handles are clamped at the point of their intersection. This is the function of the *Clamp* message. Conceptually clamping can be thought of as driving a nail through the handles. The messages accepted by clamps objects are shown in Table 2.

Using Clamps

Clamps can be used to cordon off parts of the screen from distortion. Figure 3 shows two clamps on the northmost handle and two more clamps on the westmost handle. Recall these handles are fixed as they are on the edge of the screen. The figure also has two new handles. Clamping separates the top-left region of the screen with four vertices labeled (0,0), (0,1), (1,0) and (1,1) from the rest of the layout. No further distortion in any other region can affect these four vertices now. This strategy can be used for viewing multiple regions at the same time as demonstrated in Figure 4. In Figure 3 vertices labeled (0,0), (0,1), (1,0) and (1,1) are shown in detail, in Figure 4 vertex labeled (7,7) can also be seen in detail.

3 Screen

Handles divide the screen into *regions*. A region is a contiguous area of the screen surrounded by four handles. Figure 1 shows the entire screen divided into sixteen regions by five horizontal and five vertical handles (recall that the screen has four fixed handles along its four edges). Region objects accept the messages shown in Table 3. The first four messages are self-explanatory. The *Map* message maps a point on the undistorted screen to a point on the stretched screen. A region can map only those points which lie inside that region.

Associated with each region are a *horizontal stretch factor*, s_x and a *vertical stretch factor*, s_y. The horizontal stretch factor is the ratio of the stretched width to the undistorted width of the region. Vertical stretch factor is defined similarly. Let W_p and W_d denote the current position and the displacement of the region's west handle. Similarly let S_p and S_d denote the position and the displacement of the region's south handle respectively. Any point $p(x, y)$ in the region is mapped to a stretched point $p'(x', y')$ by scaling operations using the stretch factors as scale factors as shown below:

$$x' = W_p + s_x(x - (W_p - W_d))$$
$$y' = S_p + s_y(y - (S_p - S_d))$$

Regions created by placement of handles form a two-dimensional array. This is called the *region-mesh*. Each time a handle is engaged to the screen, it causes an additional row or column of regions to be inserted into the mesh depending on whether the handle is horizontal or vertical respectively. These are the functions of the *InsertRow* and *InsertCol* messages to the region-mesh object shown in Table 4. The *Locate* message to a region-mesh returns a region containing the point specified in the argument. The returned region can be used to map the point to its corresponding stretched point. The layout object described in the next section uses the

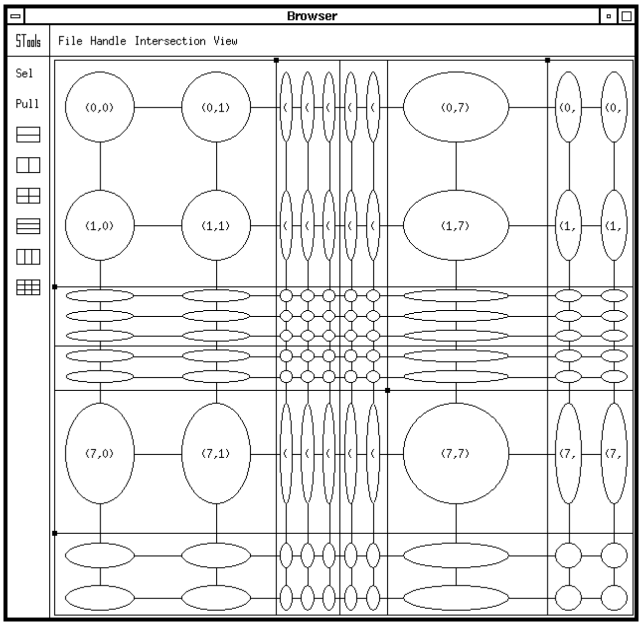

Figure 4: Using clamps to view multiple regions

Locate and the *Map* messages to compute its stretched view.

return value	operation	arguments
handle	NorthHandle	
handle	EastHandle	
handle	SouthHandle	
handle	WestHandle	
true/false	IsInside	point
point	Map	point

Table 3: Messages accepted by *region* object

return value	operation	arguments
	InsertRow	handle
	InsertCol	handle
	RemoveRow	handle
	RemoveCol	handle
region	Locate	point

Table 4: Messages accepted by *region-mesh* object

return value	operation	arguments
	Open	file
	Save	file
	Transform	region-mesh
	Draw	window

Table 5: Messages accepted by *layout* object

4 Layout

Layout objects hold the geometric specifications for the graphical structures and compute their own transformations. Layout objects accept the messages shown in Table 5. It reads and writes geometric specifications from files in response to *Open* and *Save* messages respectively. The *Transform* message causes the layout to compute its stretched view. Each handle movement causes the layout to recompute and redisplay its transformed view. Every part of the layout computes the detail to be displayed based on the screen space available to itself. Detail computation can be arbitrarily complex. For example, it can be as complex as *graph abstraction* as in [7] or as simple as font size determination.

As an example, in our prototype graph browser, a graph layout is specified in the following format:

```
begin graph
  begin vertex
    <vertex attributes>
    begin edge
      <edge attributes>
    end edge
    ...
  end vertex
  ...
end graph
```

Vertex attributes consist of shape, position, size and label. The label is an arbitrarily long text string, also called the *detail* of the vertex. Each vertex may have zero or more edges. An edge has a source vertex, and a destination vertex. It can be either a straight line, or a polyline with intermediate *bend points*.

5 Transformation

Transformation induced by *StretchTools* has a close physical analogue in the real world *i.e.* that of stretching a rubber sheet. This is the most important contribution of *StretchTools*. Additionally, a transformation must preserve sufficient similarity between the original layout and the transformed layout to minimize viewer-disorientation. In [2] Eades, Lai, Misue and Sugiyama list a number of properties of layouts – properties which should be preserved by transformations in order to preserve the "mental map" of the viewer. They include three properties, *orthogonal ordering*, *topology* and *clusters*. *StretchTools* is capable of preserving all of these three properties as explained below.

Orthogonal Ordering

Let p and q be any two points on the screen mapped to two points p' and q' by the transformation. *StretchTools* has the property that if q is on the north, or north-east,

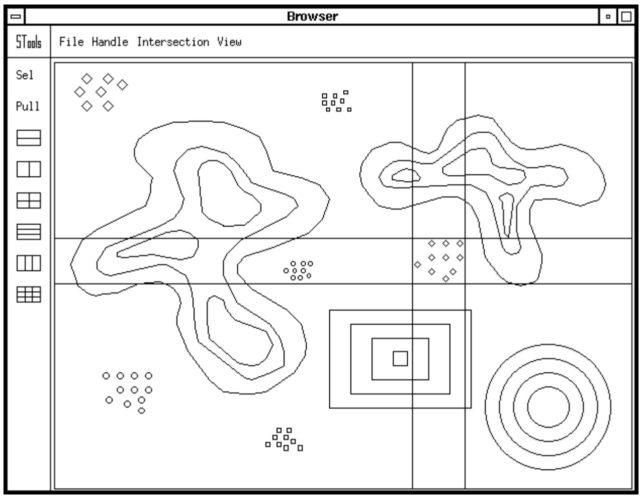

Figure 5: Initial layout

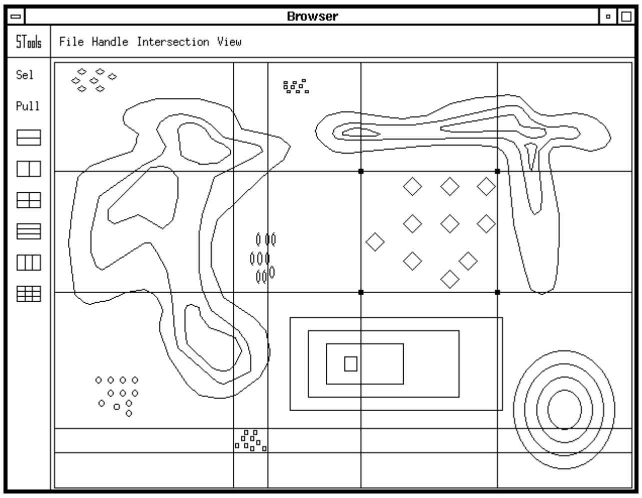

Figure 6: Enlarging a cluster

or east, or south-east, or south, or south-west, or west, or north-east direction of p, q' is also on the north, or north-east, or east, or south-east, or south, or south-west, or west, or north-east direction of p' respectively. In other words *StretchTools* preserves the up, down, left and right ordering of points.

Topology

Transformation induced by *StretchTools* is a *homeomorphism* because it preserves "open sets" in the topological sense uses (see [1]). An important consequence is that, the transformation maps the inside of a closed continuous curve to the inside of a closed continuous curve as demonstrated by Figures 5, 6, and 7. Put in another way, if there is no overlap in the original structure, there will be no overlap in the transformed structure.

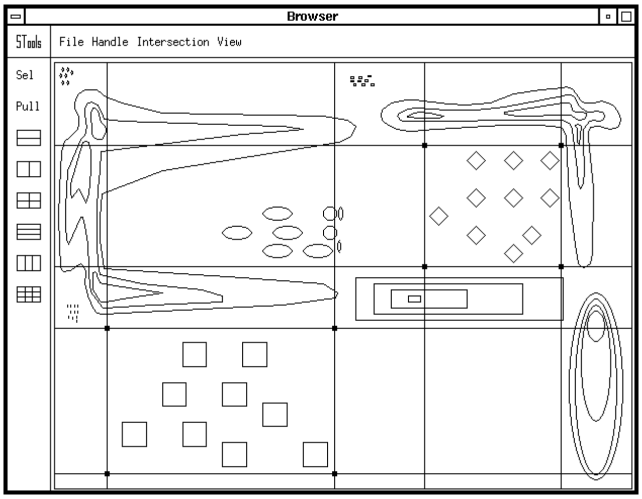

Figure 7: Enlarging another cluster

In addition, the transformation uniformly scales (although possibly with different horizontal and vertical scaling factors) points within a single region. If the vertical and horizontal stretch factors are equal for a region, the part of the diagram within that region is enlarged but otherwise completely undistorted.

Clusters

Informally, a set of objects form a cluster if they are close to each other. Figure 5 shows five clusters consisting tiny objects of various shapes. *StretchTools* allows the viewer to preserve clusters by controlling the placement of handles and by the degree of stretching. Any cluster of objects can be uniformly enlarged as shown in Figures 6 and 7.

6 Implementation

StretchTools has been implemented using *C++*, *X* and *Motif*. Currently it runs on standard SparcStation-1 and 2 at Brown University and can update the display in real-time for graphs of upto about two hundred vertices and few hundred edges.

Two major subtasks in implementing *StretchTools* are maintaining the connectivity information between handles, and computing the handle movements.

Handle Connectivity

Using graph theoretic terminology, a handle can be thought of as a vertex, and a clamp as an edge from one vertex to another. A *frame* is a *connected component* in this graph. Each time any two handles are clamped together or unclamped, this connectivity information changes.

Each handle is also assigned a *mobility number*. Conceptually, if two horizontal handles have the same mobility number, when one handle is moved, the other handle must move by the same amount. All fixed handles are assigned a single mobility number since none of them can move. Handles belonging to the same frame are assigned the same mobility number. Also all handles of any two overlapping and mobile frames are assigned the same mobility number. The algorithm for mobility number assignment is left out due to lack of space.

Handle Movement

The distance a handle is allowed to move in a certain direction is called its *mobility*. Vertical handles have *east mobility* and *west mobility*. Horizontal handles have *north mobility* and *south mobility*. When a handle is pulled, the system determines the handle's mobility in the appropriate direction. The outline of the algorithm for determining east mobility of a vertical handle follows:

```
EastMobility()
begin
  if (IsFixed())
    return(0);
  else
    m' = EastHandle().EastMobility();
    if (MobilityNo() = EastHandle().MobilityNo())
      return (m');
    else
      δx = EastHandle.Position() − Position();
      return (δx + m'− MinSeparation());
    endif
  endif
end
```

Mobility is computed by adding up amount of space available for movement less the space needed for maintaining the minimum separation between the handles. The algorithm is presented as a method of the handle object. The EastHandle() method returns the immediate next handle on the east of the handle under consideration. Essentially the same algorithm is used for determining mobility in other directions.

In the expanding part of the screen, the systems is not concerned about maintaining minimum separation between handles. Hence a different quantity called *spacing* is used instead of *mobility* for computing handle movements in this part. The algorithm for computing spacing is similar to the algorithm for computing mobility.

The algorithm for the *Move* method of an vertical handle object is shown below. Total mobility and spacing are computed for the handle using the algorithms described above. Amount of displacements are computed and propagated to the next immediate east and next immediate west handles using PropagateEast() and PropagateWest() messages.

```
Move (amount)
begin
  if (amount > 0)
    m = EastMobility();
    s = WestSpacing();
  else
    m = WestMobility();
    s = EastSpacing();
  endif
  d = Min (amount, m);
  if (MobilityNo() = EastHandle().MobilityNo())
    m' = m;
    s' = s;
    d' = d;
  else
    δx = EastHandle().Position() − Position();
    m' = m − δx + MinSeparation();
    s' = s − δx;
    if (d > 0)
      d' = d × m'/m;
    else
      d' = d × s'/s;
    endif
  endif
  PropagateWest (m, s, d);
  EastHandle().PropagateEast (m', s', d');
end
```

Below is algorithm for the PropagateEast() method:

```
PropagateEast (mobility, spacing, displacement)
begin
  if (displacement = 0)
    return;
  else
    if (MobilityNo() = EastHandle().MobilityNo())
      m' = mobility;
      s' = spacing;
      d' = displacement;
    else
      δx = EastHandle().Position() − Position();
      m' = mobility − δx + MinSeparation();
      s' = spacing − δx;
      if (displacement > 0)
        d' = displacement × m'/mobility ;
      else
        d' = displacement × s'/spacing ;
      endif
    endif
    EastHandle().PropagateEast (m', s', d');
  endif
end
```

The PropagateEast() message to a handle moves the handle by appropriate amount, computes the mobility, spacing, and displacement for the next east handle, and then sends a PropagateEast() message to the next immediate east handle. PropagateWest() method has a similar algorithm.

7 Evaluation

StretchTools has been used to visualize a number of graphs. Figures 8, 9, 10 and 11 demonstrate an instance. The graph Figure 8 has 134 vertices representing 134 cities in and around the United States and 338 edges. Figure 8 is the initial undistorted view of the graph – in this figure vertices do not have enough space to show their labels. Figure 9 shows the effect of expanding the region containing St. Louis. Next in Figure 10 Salt Lake City has been enlarged, now both St. Louis and Salt Lake City can be seen in detail. Finally a cluster of cities in the northeastern United States is shown in detail in Figure 11.

Like Fisheye Lens [8] and Perspective Wall [5], *StretchTools* allows viewers to visualize the entire structure in one window. It smoothly integrates context with detail. The advantage of *StretchTools* over other techniques is that it allows the user to precisely control the space allocation. The metaphor of stretching a rubber sheet is very intuitive. Multiple regions can be viewed at the same time in a very natural manner and the rubber sheet geometry naturally preserves orthogonal ordering, topology and clusters in layouts.

The main disadvantage of Fisheye Lens is that it does not allow viewers to control the space allocation. The Fisheye Lens however provides a very fast and convenient way to browse through large information bases. Perspective Wall has a very intuitive interface. But, Perspective Wall is designed to deal with only one dimensional information spaces. Also neither of these techniques have any natural way of viewing multiple objects at the same time.

The techniques such as *StretchTools*, Fisheye Lens and Perspective Wall manage size of the information bases by mainly distorting the layout. The other approach to managing size is *abstraction*. In [7] Paulisch and Tichy present two abstraction techniques called *graph abstraction* and *edge concentration* for dealing with large graphs. At CHI '92 Mills, Cohen and Wong described a practical tool for visualizing video data [6]. Their system also takes the approach of hierarchical abstraction for managing size. It may be worthwhile to combine the two approaches.

Our experience with the initial prototype *StretchTools* is encouraging, and we believe that this approach can provide an efficient and intuitive set of tools for viewing moderately large information bases. We are further investigating the possibility of providing composite stretching tools which are constructed from of basic handles and has additional power to make the viewing process more efficient.

References

[1] Nelson Dunford and Jacob T. Schwartz. *Linear Operators*. Interscience. 1957.

[2] 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, August, 1991.

[3] William A. Farrand. Information display in interactive design. Ph.D. Thesis, Department of Engineering, UCLA, Los Angeles, CA, 1973.

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

[5] 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.

[6] Michael Mills, Jonathan Cohen and Yin Yin Wong. A magnifier tool for video data. *Proc. ACM SIGCHI Conf. on Human Factors in Computing Systems*, pp. 93–98, 1992.

[7] Frances N. Paulisch and Walter F. Tichy. Edge: An extendible directed graph editor. *Software – Practice and Experience*, vol. 20, no. S1, pp. 63–88, 1990.

[8] 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.

[9] Robert Spence and Mark Apperley. Database navigation: An office environment for the professional. *Behavior and Information Technology*, vol. 1, no. 1, pp. 43–54, 1982.

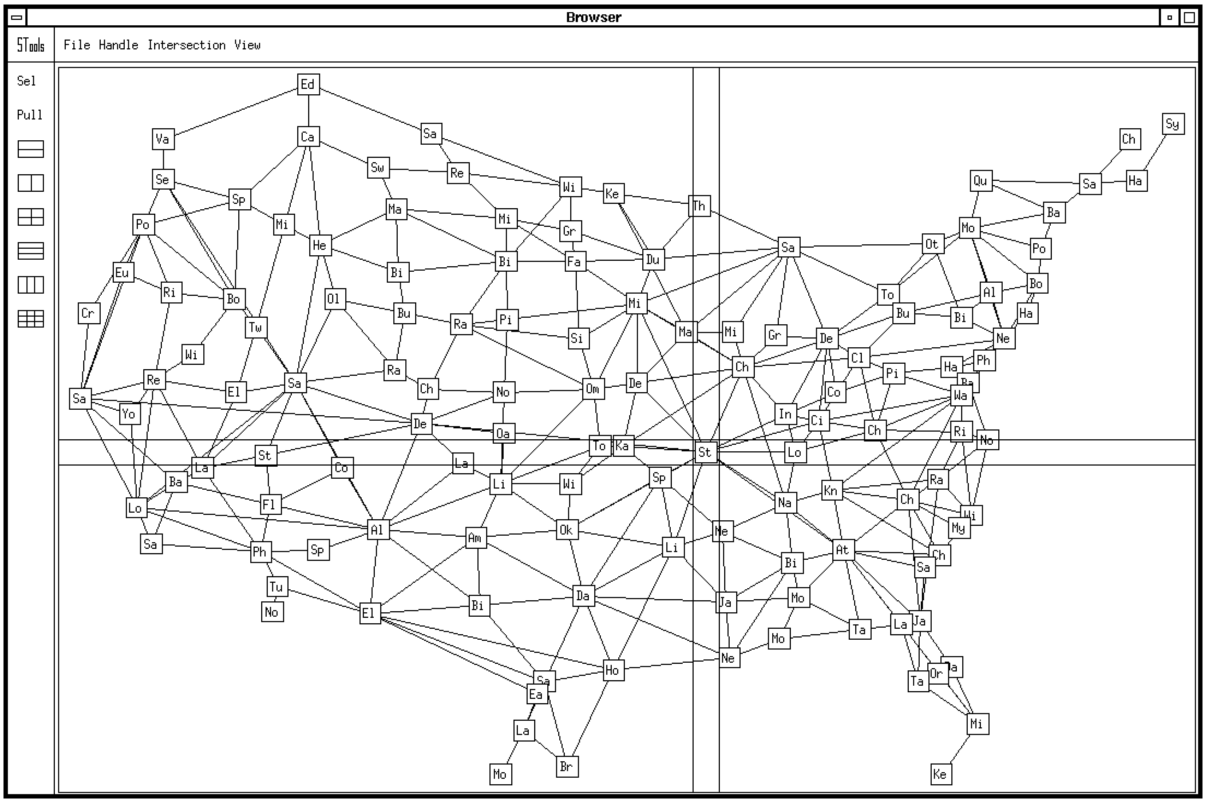

Figure 8: Initial graph with 134 cities in and around the United States as vertices

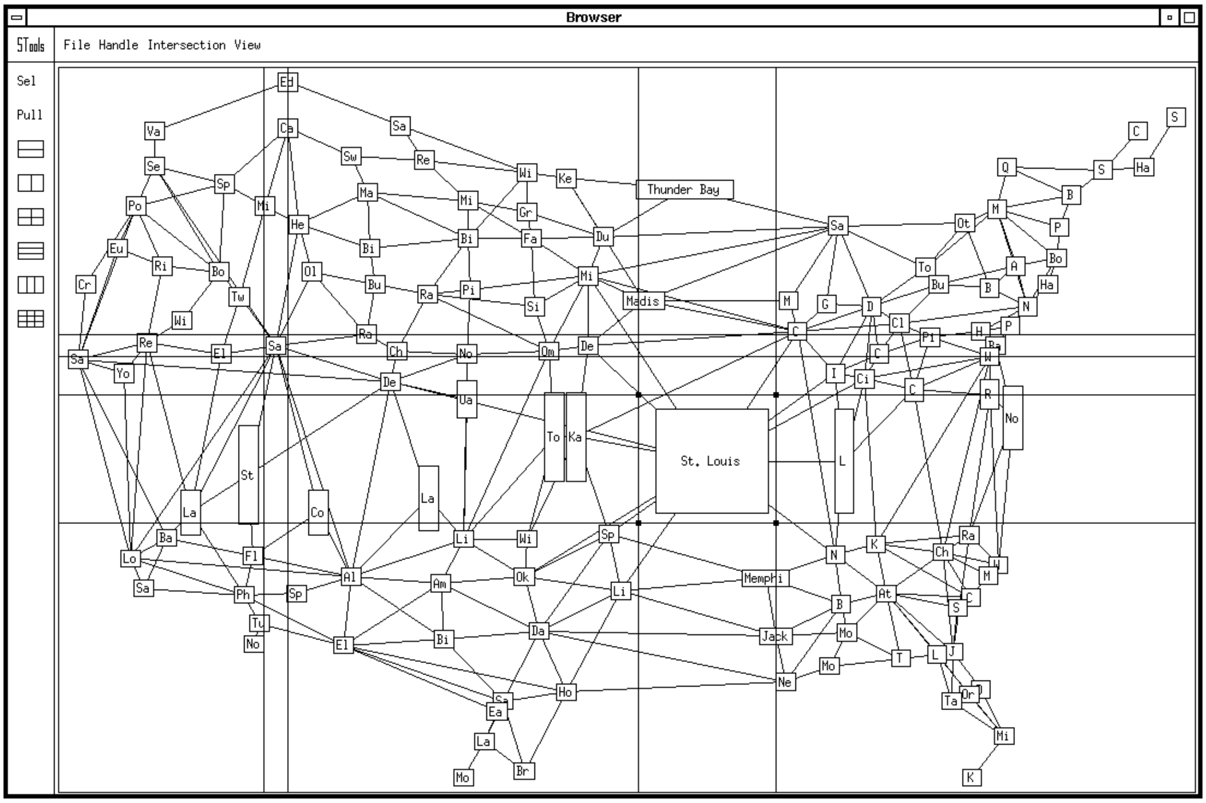

Figure 9: A view showing St. Louis in detail

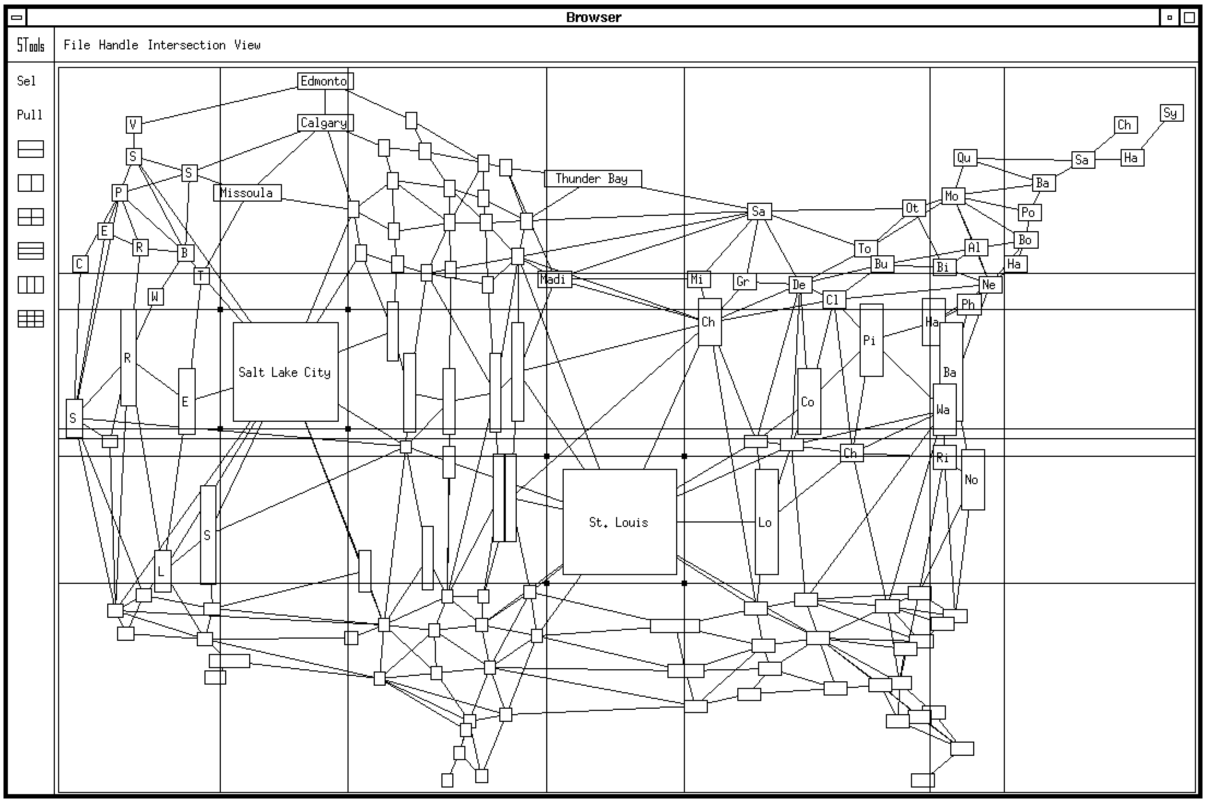

Figure 10: A view showing both St. Louis and Salt Lake City in detail

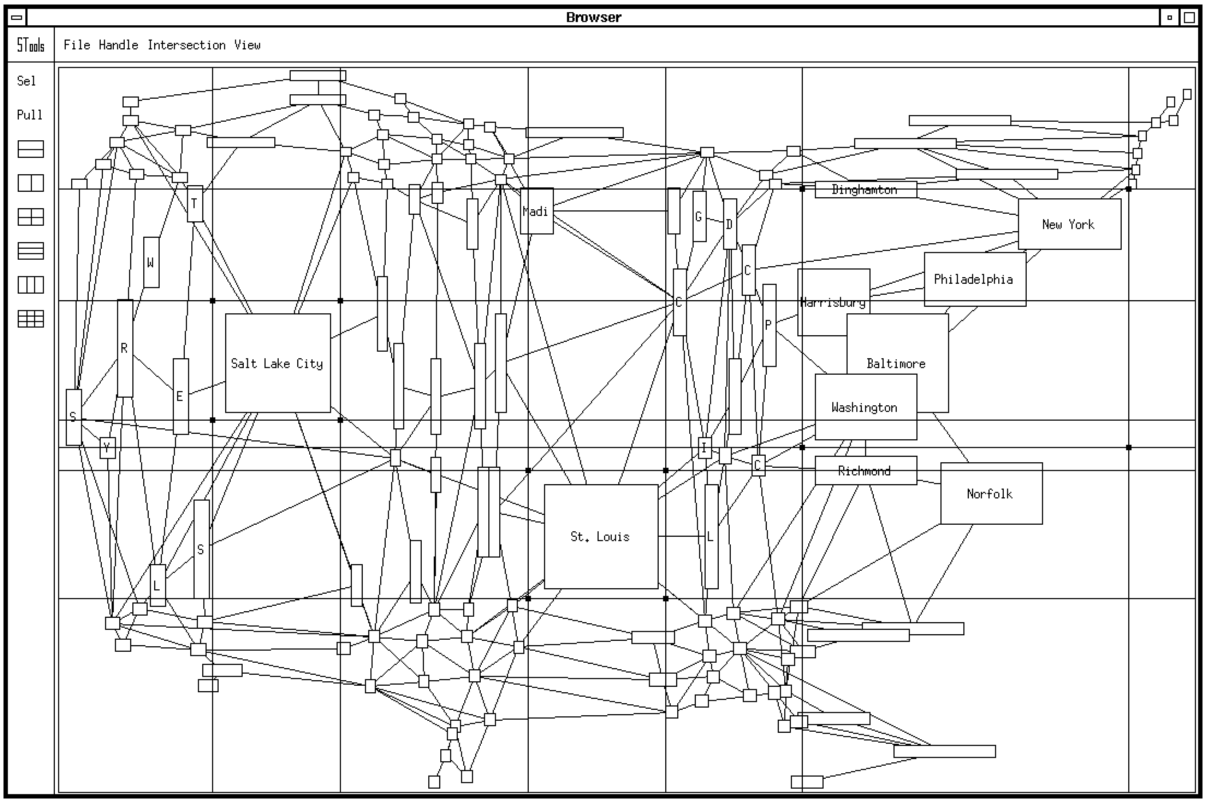

Figure 11: A view showing a cluster of cities in the northeastern United States