Graphical Fisheye Views of Graphs

Manojit Sarkar[1]
Marc H. Brown[2]

Technical Report No. CS-91-61

October 1991

[1]Brown University, Computer Science Department, Box 1910, Providence, RI 02912 USA
[2]Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301

GRAPHICAL FISHEYE VIEWS OF GRAPHS

Manojit Sarkar*
Computer Science Department
Brown University
Providence, RI 02912
(401)–863–7672
ms@cs.brown.edu

Marc H. Brown
Systems Research Center
Digital Equipment Corporation
Palo Alto, CA 94301
(415)–853–2152
mhb@src.dec.com

Abstract

A *fisheye* lens is a very wide angle lens that shows places nearby in detail while also showing remote regions in successively less detail. This paper describes a system for viewing and browsing planar graphs using a software analog of a fisheye lens. We first show how to implement such a view using solely geometric transformations. We then describe a more general transformation that allows hierarchical, structural information about the graph to modify the views. Our general transformation is a fundamental extension to the early efforts in fisheye views.

Keywords: Fisheye views, information visualization

1 Introduction

Graphs with hundreds or thousands of vertices and edges are common in many areas of computer science, such as network topology, VLSI circuits, and graph theory. There are literally hundreds of algorithms for positioning nodes to produce an aesthetic and informative display [1]. However, once a layout is chosen, what is an effective way to view and browse the graph on a workstation?

Displaying all the information associated with the vertices and edges (assuming it can even fit on a screen) shows the global structure of the graph, but has the drawback that details are typically too small to be seen. Alternatively, zooming into a part of the graph and panning to other parts does show local details but loses the overall structure of the graph. Researchers have found that browsing a large layout by scrolling and arc traversing tends to obscure the global structure of the graph [3]. Two (or more) views — one view of the entire graph and the other of a zoomed portion — has the advantage of seeing both the local detail and overall structure, but has the drawbacks of requiring extra screen space and of forcing the viewer to integrate the views. However, the multiple view approach still has the drawback that the parts of the graph adjacent to the enlarged area are not visible at all in the enlarged view.

This paper explores a *fisheye* lens approach to viewing and browsing graphs. A fisheye view of a graph shows an area of interest quite large and with detail, and shows the remainder of the graph successively smaller and in less detail. Thus, a fisheye lens seems to have all the advantages of the other approaches and without suffering any of the drawbacks.

A typical graph is displayed in Figure 1, and Figure 2 shows a fisheye view it. The small cross-mark on the vertex in Figure 2 indicates that it is the current point of interest to the viewer. We call this point the *focus*. In our prototype system, a viewer selects the focus by pointing with a mouse. As the mouse is moved with its button down, the focus changes and the display updates in real time. The size and detail of a vertex in the fisheye view depend on the vertex's distance from the focus and a preassigned importance associated with the vertex.

Our work extends Furnas's pioneering work on fisheye views [2] in two ways. First, we provide a graphical interpretation to fisheye views. Furnas's work was developed with an ascii text orientation. Second, we provide a general purpose framework for generating graphical fisheye views. Four functions are provided as framework to create a fisheye transformation. The framework allows information components to be present in the view at varying levels of detail. In Furnas's original formulation of the fisheye view, a component is either present in full detail or completely absent from the view.

The next section defines the terminology and conventions used in the remainder of the paper. In section 3 we describe our general framework. Section 4 presents the functions we used in our prototype system. The system itself is described in section 5. In the remaining sections, we review related efforts, and offer some thoughts on future directions.

*Supported by a research internship from Digital Equipment Corporation.

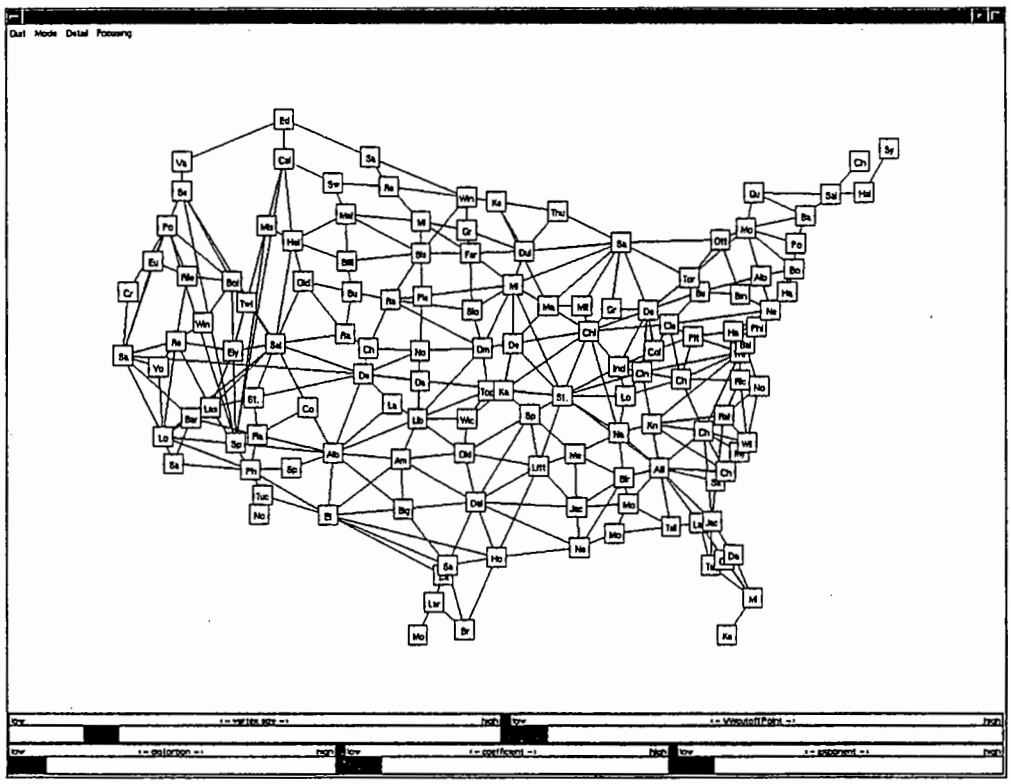

Figure 1: The initial layout of a graph with 134 vertices and 338 edges.

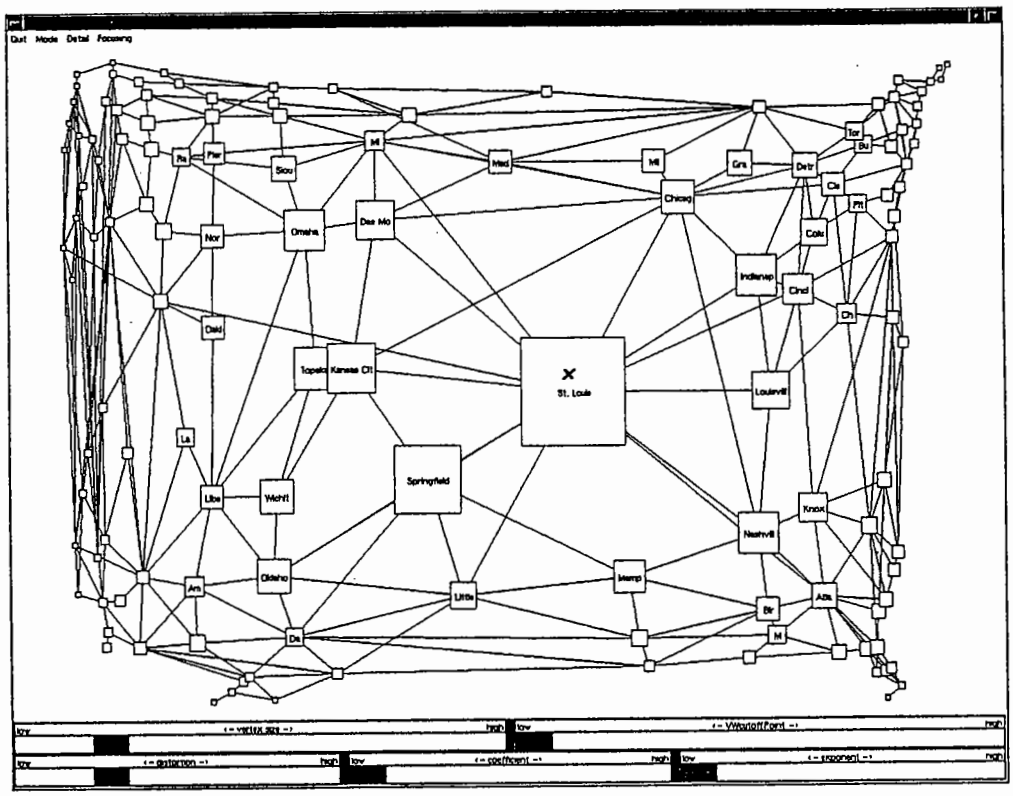

Figure 2: A fisheye view of the graph in figure 1, $d = 5.71, c = 0, e = 0$, $VWcutoffPoint = 0$

2 Terminology

A graph consists of *vertices* and *edges*. The initial layout of the graph is called the *normal view* of the graph, and its coordinates are called *normal coordinates*. Vertices are graphically represented by shapes whose bounding boxes are square. Each vertex has a *position*, specified by its normal coordinates, and a *size* which is the length of a side of the bounding box of the vertex. Each vertex is also assigned a number to represent its relative importance in the global structure. This number is called the *a priori importance* or the *API* of the vertex.

An edge is represented by either a straightline from one vertex to another, or by a set of straightline segments to simulate curved edges. Edges consisting of multiple straightline segments are specified by a set of intermediate *bend points*, the extreme points being the coordinates of its corresponding vertices.

The coordinates in the fisheye view are called the *fisheye coordinates*. The viewer's point of interest is called the *focus*. It is a point in the *normal coordinates*.

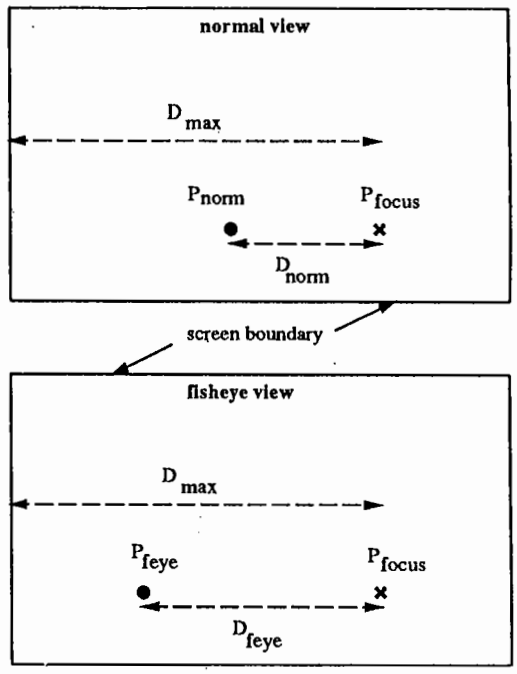

Figure 3: Mapping points

3 Generating Fisheye Views

Generating a fisheye view involves magnifying the vertices of greater interest and correspondingly demagnifying the vertices of lower interest. In addition, the positions of all vertices and bend points must also be recomputed in order to allocate more space for the magnified portion so that the entire view still occupies the same amount of screen space.

Intuitively, the size of a vertex in the fisheye view depends on its API and its distance from the focus. The amount of detail displayed in a vertex in turn depends on its size. The position of a vertex in the fisheye view depends on its position in the normal view and its distance from the focus. We now formalize these concepts.

The importance of a vertex v with respect to the current focus f is called its *visual worth*, VW. It is a function of the distance between v and f in normal coordinates and of v's API:

$$VW(v,f) = \mathcal{F}_1(D_{norm}(v,f), API(v)) \quad (1)$$

The position of v in the fisheye view $P_{feye}(v,f)$ is a function of its position in normal coordinates and distance from the focus in the normal coordinates:

$$P_{feye}(v,f) = \mathcal{F}_2(P_{norm}(v), D_{norm}(v,f)) \quad (2)$$

The size of v in the fisheye view $S_{feye}(v,f)$ is a function of its size in normal coordinates, its distance from the focus in normal coordinates, and its API:

$$S_{feye}(v,f) = \mathcal{F}_3(S_{norm}(v), D_{norm}(v,f), API(v)) \quad (3)$$

Finally the amount of detail to be shown for v depends on $S_{feye}(v,f)$, and its maximum detail:

$$DTL_{feye}(v,f) = \mathcal{F}_4(S_{feye}(v,f), DTL_{max}(v)) \quad (4)$$

One has to choose the functions $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_3$, $\mathcal{F}_4$ appropriately to generate useful fisheye views. Readers familiar with Furnas's work will note that our fundamental contributions are the existence of arbitrary functions $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_3$, $\mathcal{F}_4$. In the next section 4 we present the set of functions we used in our prototype system.

4 Fisheye Transformations

Generating fisheye views is a two step process. First we apply a geometric transformation to the normal view in order to reposition vertices and magnify and demagnify areas close to and far away from the focus, respectively. Second, we use the API of vertices to obtain their final size, detail, and visual worth. If the API of all the vertices are equal, the final size of all the vertices are equal and the second step is unnecessary.

4.1 Mapping Position

Transforming from normal coordinates to fisheye coordinates, using focus position P_{focus} requires us to implement the function $\mathcal{F}_2$ in Equation 2. The function we used was

$$P_{feye} = \mathcal{G}(P_{norm}) D_{max} + P_{focus}$$

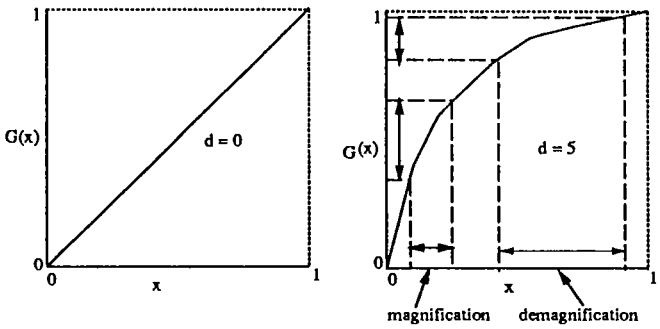

Figure 4: x versus $\mathcal{G}(x)$ plot for $d = 0$ and 5

where

$$\mathcal{G}(P_{norm}) = \frac{(1+d)\frac{D_{norm}}{D_{max}}}{d\frac{D_{norm}}{D_{max}}+1} = \frac{d+1}{d+\frac{D_{max}}{D_{norm}}}. \quad (5)$$

Note that the x and y dimensions are treated completely independently in the above mapping.

Figure 3 illustrates the mapping. D_{max} is the distance of the boundary of the screen from the focus. The function $\mathcal{G}(x)$ is monotonically increasing and continuous for $0 \leq x \leq 1$ with $\mathcal{G}(0) = 0$, and $\mathcal{G}(1) = 1$. The constant d in Equation 5 is called the *distortion factor*. The behavior of the function for $d = 0$ and $d = 5$ is shown in Figure 4. For large values of d the slope of the plot x versus $\mathcal{G}(x)$ near $x = 0$ is very high which results in high magnification. The plot has a very low slope near $x = 1$ which causes high demagnification. For $d = 0$ the normal and the fisheye coordinates of every point is the same.

4.2 Mapping Size

While computing size, the square shape of the bounding boxes of the vertices is preserved. Implementation of the size mapping function $\mathcal{F}_3$ in Equation 3 is described here in two separate steps. The first step uses the geometric transformation to compute the geometric size $S_{geom}(v, f)$ by ignoring v's API. This mapping has the special property that if no two vertices in the normal view overlapped, no two vertices in the transformed view overlap. The second step then uses $S_{geom}(v, f)$ and v's API to complete the implementation of $\mathcal{F}_3$. However, the vertices may overlap after the second step. We compute S_{geom} as follows:

$$S_{geom} = \sqrt{2}\ MIN(MapX(x_{norm} + \frac{S_{norm}}{2}) - x_{feye},$$
$$MapY(y_{norm} + \frac{S_{norm}}{2}) - y_{feye})$$

where $P_{norm} = (x_{norm}, y_{norm})$ and $P_{feye} = (x_{feye}, y_{feye})$. The functions $MapX$ and $MapY$ map the x and y coordinates of a point respectively using the function $\mathcal{F}_2$ in Equation 2.

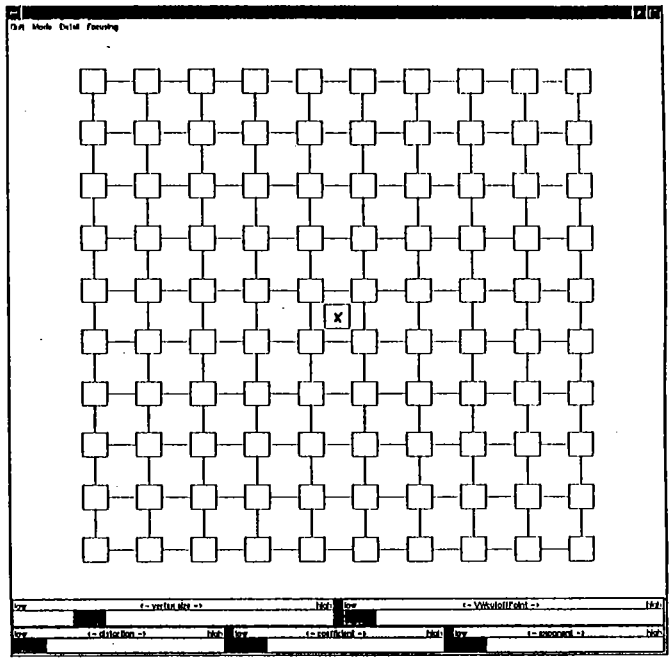

Figure 5: An undistorted symmetric graph, $d = 0$

Finally, the function $\mathcal{F}_3$ in Equation 3 is implemented by

$$S_{feye} = S_{geom}(cAPI)^e scale \quad (6)$$

where the coefficient c, exponent e, and scale factor *scale* are global constants.

4.3 Computing Detail and Visual Worth

The functions $\mathcal{F}_4$ and $\mathcal{F}_1$ are implemented by the Equation 7 and Equation 8 below. Both equations utilize S_{feye} computed by Equation 6.

$$DTL_{feye}(v, f) = MIN(DTL_{max}(v), \alpha S_{feye}(v, f)) \quad (7)$$

where α is a constant.

$$VW(v, f) = \beta S_{feye}(v, f) + \gamma \quad (8)$$

where β and γ are constants.

4.4 Mapping Edges

Straightline edges get mapped automatically when vertices at their end points get mapped. The edges with intermediate bend points can be mapped by mapping each bend point separately. Figures 5 and 6 demonstrate the effect of straightforward geometric transformation on a symmetric graph for two different values of the distortion factor d.

Unfortunately, this straightforward approach does not preserve parallelism between edges. This problem

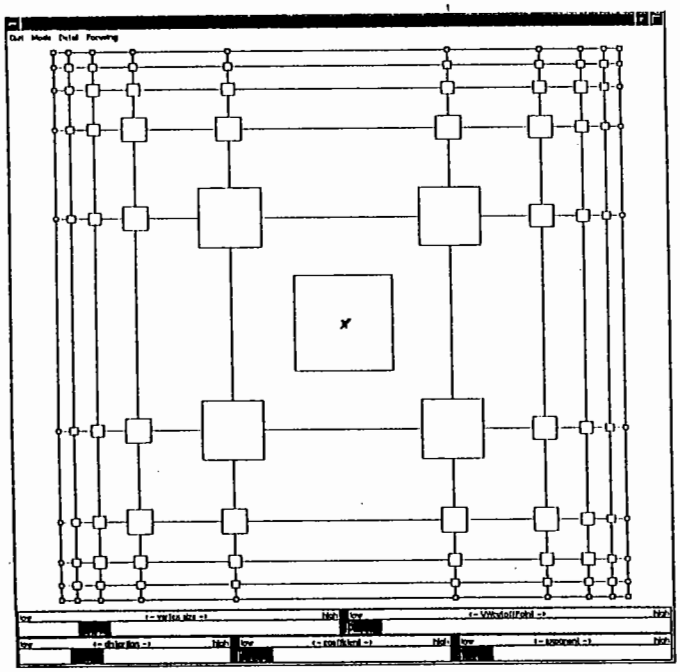

Figure 6: Transformation on the graph in Figure 6, $d = 5$

can be circumvented by mapping a very large number of intermediate points on each straightline segment individually. However, mapping too large a number of points may not be computationally feasible for real time response.

The mapping, however, has the property that all the vertical and horizontal lines remain vertical and horizontal after the transformation. Because of this property, our transformation is ideally suited for graphs with edges consisting of only horizontal and vertical linesegments, for example, VLSI circuits.

5 The Prototype

Our prototype browser is shown in Figure 7. The system displays a fisheye view of a graph, and updates the display in real time as the user moves the focus by dragging with the mouse. The user can specify a vertex textually that should become the focus. Sliders give the user control of the value of the distortion factor d in Equation 5, the coefficient c and the exponent e in Equation 6, the vertex scaling factor *scale* also in Equation 6, and a cutoff point at which vertices (and their incident edges) should no longer be displayed. The coefficient c and the exponent e in Equation 6 control the effect of the API of the vertices on the non-geometric part of the transformation while d affects the geometric part of the transformation. The combined effect of these parameters on the graph in Figure 1 are shown by the series of Figures 1, 2, 7 and 8.

By and large, as the focus changes, the new image is

```
loop
  f := GetFocus()
  if f ≠ f_old then
    for each v ∈ V
      eval P_feye(v,f), S_feye(v,f), DTL_feye(v,f)
    end
    for each e ∈ E
      if not straightLine(e) then
        for each bp ∈ bendPoints(e)
          mapPoint(bp, f)
        end
      end
    end
    for each v ∈ V
      eval VW(v,f)
    end
    sort V in order of VW
    for each (v1, v2) ∈ E
      if VW(v1) ≥ VWcutoffPoint and
         VW(v2) ≥ VWcutoffPoint then
        repaint edge between v1 and v2
      end
    end
    for each v ∈ V in nondecreasing order of VW
      if VW(v) ≥ VWcutoffPoint then
        repaint vertex v
      end
    end
    f_old = f
  end
```

Figure 9: Main loop of the prototype

similar enough to the previous image that the animation looks smooth. The system maintains the invariant that the location of the focus is the same in both the normal coordinates and in the fisheye coordinates. When the mouse is within the bounding box of a vertex, that vertex is displayed at its maximum size and the image is not updated while the mouse remains within the vertex, thereby breaking the invariant. When the mouse finally exits the vertex's bounding box, the invariant is again maintained. Because it would appear that the mouse moved quite a bit (the size of the bounding box), the change in the image upon exit is quite drastic, unless some special-casing is done to take this into account.

5.1 Implementation

The prototype is implemented in an event-driven style. The main loop is as in Figure 9 .

This code glosses over the special-casing that must be done when f remains within the bounding box of a vertex, and as soon as the mouse exits the bounding

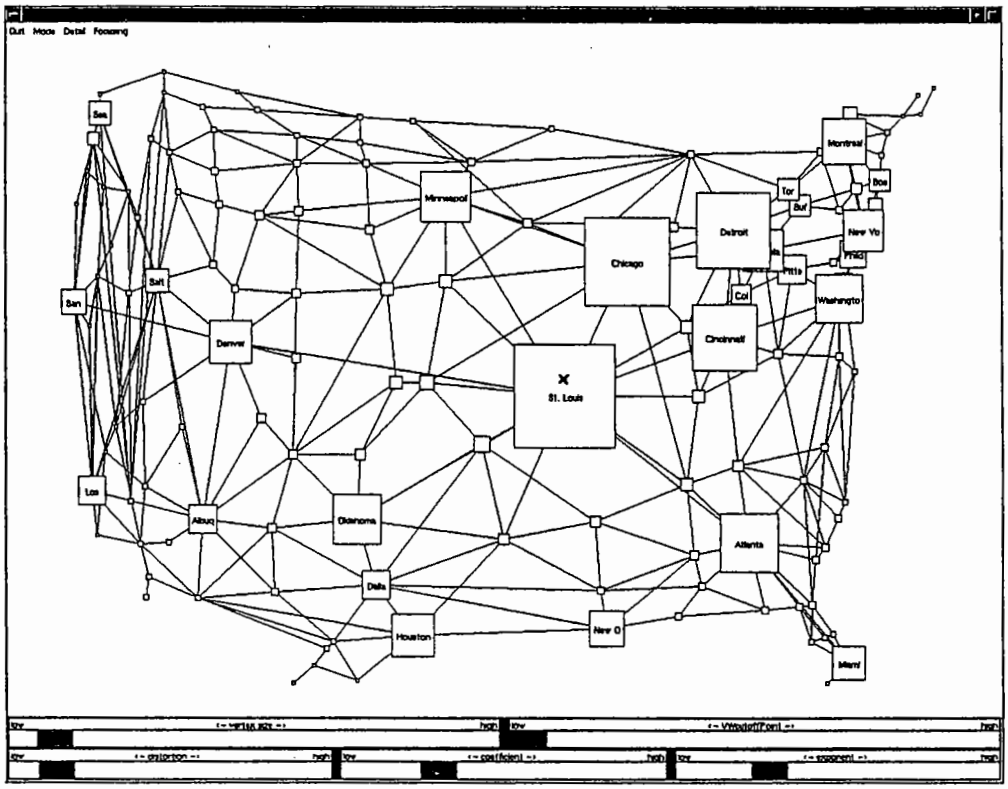

Figure 7: A fisheye view of the graph in Figure 1, $d = 2.38, c = 1.0, e = 1.14$, $VWcutoffPoint = 0$

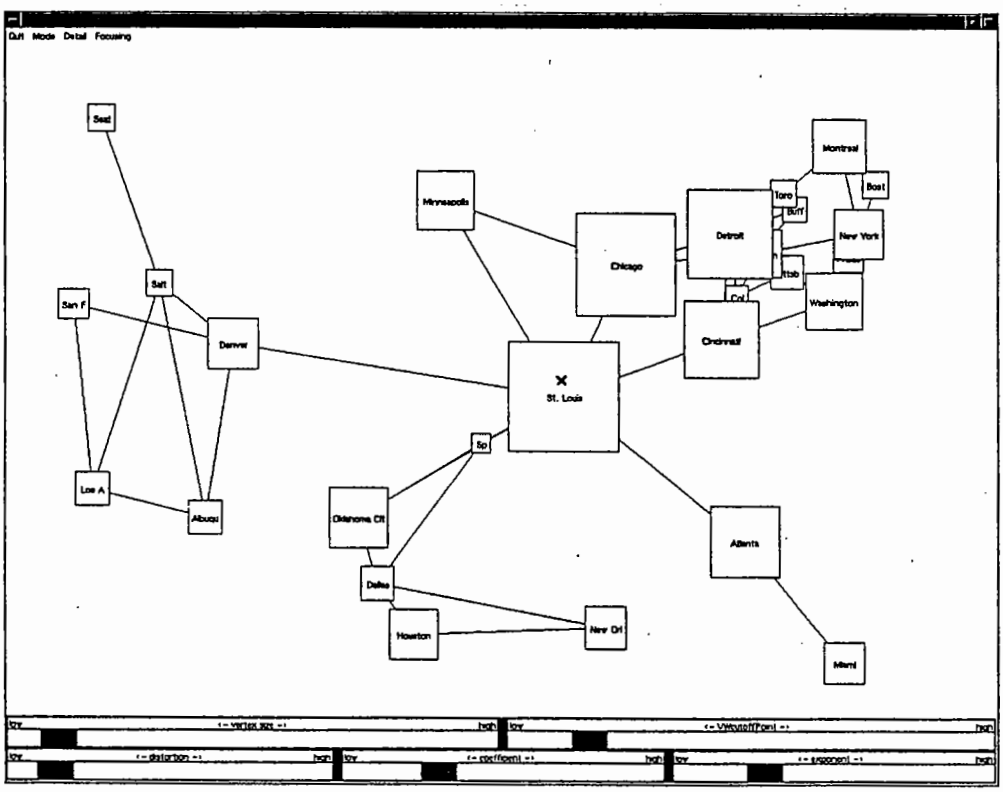

Figure 8: A fisheye view of the graph in Figure 1, $d = 2.38, c = 1.0, e = 1.14$, $VWcutoffPoint = 0.27$

box.

A standard bucket sort algorithm is used to sort the vertices in nonincreasing order of VW in linear time by normalizing VW of vertices to fall between 0 and 1. Sorting the vertices in order of their visual worth produces a useful order. The order can be used to determine the order of painting for two reasons: First, if the position of two vertices are in conflict, their VW can be used to resolve the conflict in favor of the vertex with higher VW. Second, to ensure that the screen is updated periodically, the code would paint the most important vertices first, and stop painting (the less important) vertices when a timer expired.

5.2 System Notes

The prototype was implemented on a DECstation 5000 using Modula-3 and Trestle, a portable X-toolkit [5]. This project was the first Trestle application to be written, beyond the handful of small examples that are part of the distribution package. Unfortunately, a number of features that we needed for real time animation (i.e., fast double buffering), and aesthetic drawings (curves and lines of arbitrary thickness) were not fully functional when the initial prototype was developed during the summer. We are currently working on upgrading to the latest release of Trestle.

6 Related Work

This work follows from the CHI '86 generalized fisheye paper by Furnas [2]. Furnas gave many compelling arguments and performed a number of experiments to validate the usefulness of fisheye views. His implementation was primarily oriented to ascii terminals. We have extended his work in two directions. First, we provide a fully general graphical interpretation to the notion of fisheye views involving a geometric transformation. Second, our transformations allow a graphical object to be shown at varying sizes and degrees of detail as functions of the object's distance from the focus and the object's preassigned importance in the global structure.

The geometric transformation described in section 4 is very general and can be applied to any arbitrary structure. Figures 10, 11, 12 show the geometric transformation on a outline of the United States.

Our approach can also be easily extended to multiple foci. In such situations, first one has to divide up the screen-space among all the foci using some criteria, and then apply the transformation individually on each portion of the screen.

In CHI '91, Mackinlay, Robertson and Card presented two views of information that have fisheye properties. The *perspective wall* [4] maps wide 2-dimensional layout into a 3-dimensional visualization with center panel for detail and two perspective panels for context. The *cone*

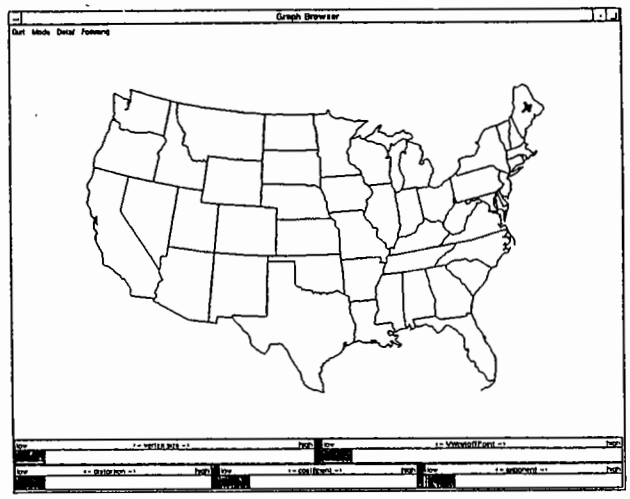

Figure 10: Outline of the United States

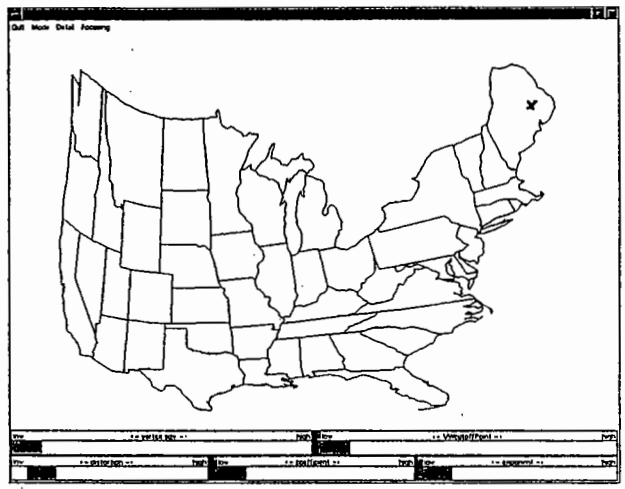

Figure 11: A distorted view of the outline in Figure 10

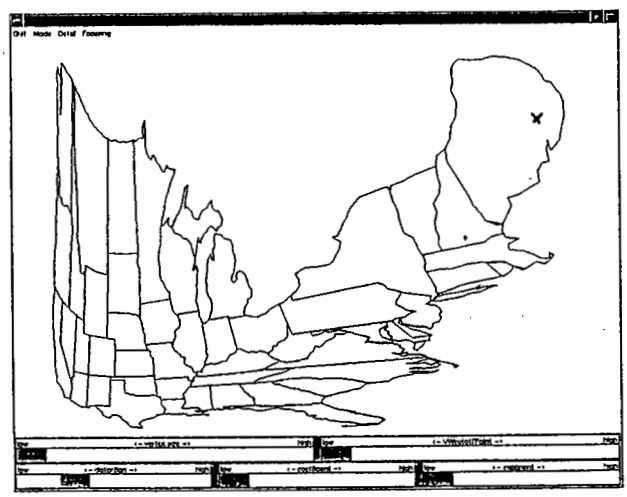

Figure 12: More distortion of the outline in Figure 10

tree [6] displays a tree onto 3-dimensions. Each node of the tree is the apex of a cone, with the children of the node positioned around the rim of the cone. The fact that the tree is in 3-dimensions has the basic fisheye property of showing local information in detail, because it is close to the viewer, while also showing the entire context.

Our approach could be applied to trees. In the fisheye view of a tree that Furnas shows, each node is either displayed or not. Our system, because of the extensions to Furnas's work, allows nodes to be of varying sizes and levels of detail. It is our intuition that our generalized graphical fisheye view would compare favorably with a cone tree as a visualization of hierarchical information.

7 Summary

The graphical fisheye view is a promising technique for view and browsing large graphs. Although we need to support its efficacy experimentally, experiences with our prototype browser gives us confidence that our approach is indeed powerful. However, we do not claim that a graphical fisheye view is *the* correct way to view and browse a graph. Rather, it is one of the many ways that are possible. Like other representations, it has strengths and weaknesses. Discovering and quantifying these are challenges for the future.

Acknowledgements

The work described in this paper was done at the Systems Research Center of Digital Equipment Corporation. Jorge Stolfi helped us with various ideas on geometric transformations. Greg Nelson, Eric Muller, Steve Glassman, Mark Manasse, and Bill Kalsow helped us solve various technical problems during the implementation.

References

[1] Eades, P., and Tamassia, R. Algorithms for drawing graphs: An annotated bibliography. Unpublished Technical Report, Brown University, Computer Science Department, October 1989.

[2] Furnas, G.W. Generalized fisheye views. In Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems (Boston, Mass.). ACM, New York, 1986, pp. 16-23.

[3] Henry, T.R., and Hudson, S.E. Interactive graph layout. In Proceedings of UIST '91, November, 1991.

[4] Mackinlay, J.D., Robertson, G.G., and Card, S.K. The perspective wall: Detail and context smoothly integrated. In Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems (New Orleans, Louisiana). ACM, New York, 1991, pp. 173-179.

[5] Nelson, G. Systems Programming with Modula-3. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[6] Robertson, G.G., Mackinlay, J.D., and Card, S.K. Cone Trees: Animated 3D visualizations of hierarchical information. In Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems. ACM, New York, 1991, pp. 189-194.