ok for ScM research project writeup
David Laidlaw

SciVL: A Descriptive Language for 2D Multivariate Scientific Visualization Synthesis

Jason S. Sobel
Masters Project
Brown University
May 14, 2003

Abstract

We present SciVL, the Scientific Visualization Modeling Language, a new way to describe scientific visualizations of multivariate, two-dimensional datasets. Our goal was to create a language with which a user could rapidly prototype complex and precise data-driven visualizations as well as facilitate their modification during the iterative design process. Our language can combine three types of basic visual elements: discrete icons, color planes, and streamlines in layers. A text file fully describes the resulting image and controls the different elements' layering, appearance, relative locations, and spacing, including the mapping assignments of any of these characteristics to data variables. In addition the language supports extensions to include other basic visual elements and to accommodate 3D data. An interdisciplinary scientific visualization course that synthesized the efforts of computer scientists as well as graphic design and illustration students inspired us to choose the three basic types of visual elements. A review of graphic design literature and feedback from expert visual designers provided us with an anecdotal evaluation of our design's adequacy for the visual language. We have applied our language to fluid flow and polarimetric radar data visualizations. These visualizations elucidated some issues in our framework, such as limited handling of different types of 2D data and the perceptual efficacy of our icon- and streamline-placing algorithm. In sum, the visual language we present, in conjunction with the simple text interface used for generating the visualizations, accomplishes our primary goal of providing a way to quickly describe and modify complex data-driven visualizations for two-dimensional, multivariate datasets.

1 Introduction

1.1 Motivation

One of the main challenges facing those in the field of scientific visualization is the length of the development process. Like a software engineering project there are design, implementation, and evaluation phases, but for visualizations this process is nested. First the programmer, the scientist, and the designer must work together to choose informative, visually pleasing, computationally feasible "visual elements." We use the term visual element to mean a graphic used in the visualization, such as "A semi-transparent gray triangle to show velocity, with area tied to speed and orientation to direction" or "A plane that is transparent where pressure is below a threshold and otherwise red, where saturation varies based on the pressure value."

Once these have been selected the programmer must go through the normal development cycle to produce a system that creates output images. Then all three get back together to review the results and, almost certainly, iterate on the whole process.

Rapid prototyping is a natural answer to this problem: if users can swiftly and easily create prototype visualizations without the long development process then they will converge on a solution much faster.

A second impetus for this work was our desire to "optimize" visualizations. We observed that, with the ability to represent a visualization as an abstract data object, we could use traditional optimization algorithms, like genetic algorithms. Optimization would allow us to answer questions like "What is the best way to show velocity, vorticity, and pressure in one image?" or "Is binding color to one variable and transparency to another an effective way to show two independent variables?"

1.2 Goals

Given these motivations we decided to create a language that would describe visualizations.

For rapid prototyping we determined that the language must be simple, expressive, and flexible. Simple so that users would not have to learn complicated syntax to design a visualization, expressive so a wide variety of visualizations are possible, and flexible so it is easy to make modifications to an existing file.

For optimization we wanted the language to be hierarchical and also break down in to small "genes." The hierarchy would allow easy "crossover" operations when running the genetic algorithm (swapping a portion of one visualization in to another). Breaking down each component of a visualization in to a gene is crucial to allow fine-grained optimization.

We also wanted the language to lead us towards a great understanding of how to do "good" multivariate visualizations. Ideally the language would allow us to determine which visual elements work well together to show different data variables.

1.3 Our Approach

Given these design goals we developed SciVL: The Scientific Visualization Modeling Language, which provides a way to abstract the representation of a visualization. With the ability to represent visualizations as simple text files, designers could rapidly prototype their solutions by creating a SciVL file that describes their desired visual elements. Then they would supply this file and a dataset to a rendering engine, producing a prototype image.

SciVL would allow optimization since its whole purpose is to be an abstract representation of a visualization.

Currently, SciVL is only used for 2D visualizations, but the language itself has the features necessary for three dimensions.

1.4 Related Work

A number of authors have investigated and classified the basic elements of a visualization in different ways. Bertin[1] and Tufte[2, 3] present, in similar ways, examples of successful and unsuccessful information visualization displays. They go further and analyze the reasons and, in the case of Bertin, present an initial set of visual elements and features that can be tied to data variables to represent information. Our classification is based, in part, on these authors' findings.

Cleveland et al.[4] include a classification of perceptual tasks that viewers of the visualizations need to perform to extract information from them. Indeed, the design process must be informed by the ultimate goal of the visualization. Our current implementation does not explicitly include design rules that trigger changes in the visualization depending upon the task at hand. The overall design of the language has been made so these kinds of rules could be included.

Our list of visual features for each type of element has been based, not only on feedback from expert designers and graphic design literature[5], but also on a compilation of the most common visual features researchers in computer graphics and visualization have used to represent visual information. Some of the most relevant work has been made by Interrante[6] and Taylor[7]. They defined textural features to represent data variables and explored ways to synthesize those textures based on their color, transparency or density. Our layers can be seen as individual overlapping textures representing information. The visual features try to capture the flexibility and expressiveness accomplished by Interrante and Taylor.

Ware[8] compiled visual perception and graphics to generate information visualization displays that optimally exploit the capabilities of the human visual system. SciVL tries to avoid a large "space" of possibilities by minimizing the number of styles and visual elements–without losing representation power.

One of the boons of collaborating with design and illustration experts was their feedback on any omissions in visual elements or any redundancy in the styles specified in our language.

Visualization systems have been created to allow scientists to design and test their own data visualizations. The basis of most of the systems is not perceptual optimality, but rather user flexibility to try different designs quickly. Interactive GUIs are used to help bind data variables to the available visual elements. Glyphmaker[9] is a tool that allows users to create their own complex icons with variable binding.. Our set of tools for the user is more limited but we believe it to be sufficient, until formal studies evaluate their full potential. As far as we could tell, however, streamlines and colorplanes do not appear in GlyphMaker.

AVS[10] goes even further by presenting the user with programmable modules that the information travels through. The modules form networks that ultimately represent the visualization design. Our text description of the visualization also contains complete information to generate the final image and requires no programming knowledge. Our current language and rendering system are not as sophisticated as AVS or similar packages, but their simplicity of use and expressive power will be shown through our initial results later in this paper.

1.5 Paper Organization

In Section 2 we will discuss in detail our methods, including the design of SCiVL and the rendering system. In Section 3 we will present our results and discuss how the system meets our goals and also provide anecdotal feedback from users. Then in Section 4 we will look at work in progress and future work that leverage the advances made by SciVL. We end in Section 5 with our conclusions.

2 Methods

We focus on how the language describes a visualization.

A SciVL file uses an arbitrary number of layers to build up a visualization. There are three different layer types: icon, colorplane, and streamline. Each layer defines some number of visual elements that will be used to visualize the dataset. During rendering, all the layers are composited together to produce the final image, as shown in Figure 1.

2.1 Values: Specifying "Numbers"

When reading about these layers it is important to realize that the properties we will mention are not given number values but instead one of three more abstract values: constant, random, or data-driven. A specific visual element will likely have many instances (appear many times) in the final visualization. The SciVL file determines, for each property of the element, if that property is constant, random, or data-driven across each instance. We now look at this idea in more detail.

A constant value is the simplest: a number is directly specified in the file and is used for each instance of the visual element. In the following example, the width of the border for the icon will always be two units:

```
BORDER WIDTH Constant 2.0
```

A random value specifies a range of possible values, to be sampled uniformly. In this example the border width will randomly vary between 1.0 and 3.5 units for each instance:

```
BORDER WIDTH Random 1.0 3.5
```

A data-driven value can be used in two different ways. The first is to sample the specified data variable at the icon's location and use the resultant number. In this example, the underlying data is a gradient

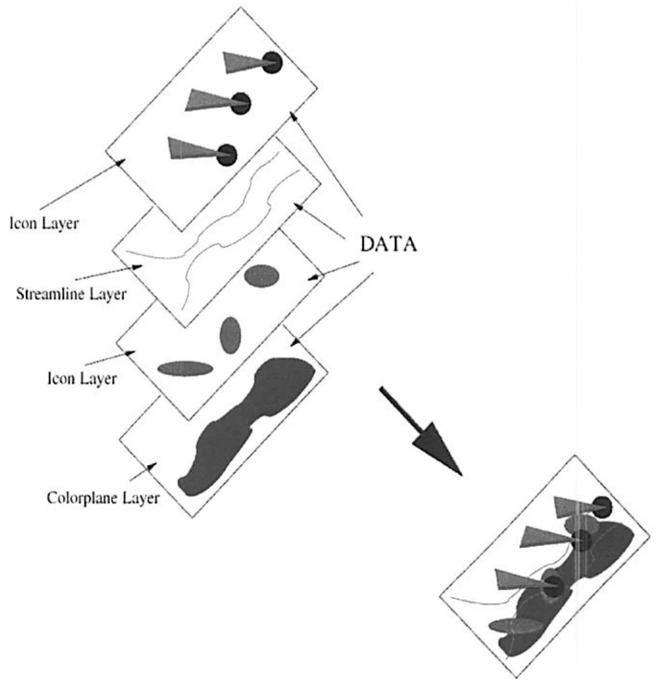

Figure 1 An example of the layering system. The SciVL file specifies the visual elements in each layer and whether the properties are constant, random, or data-driven. Then the rendering system composites the layers to produce the final image.

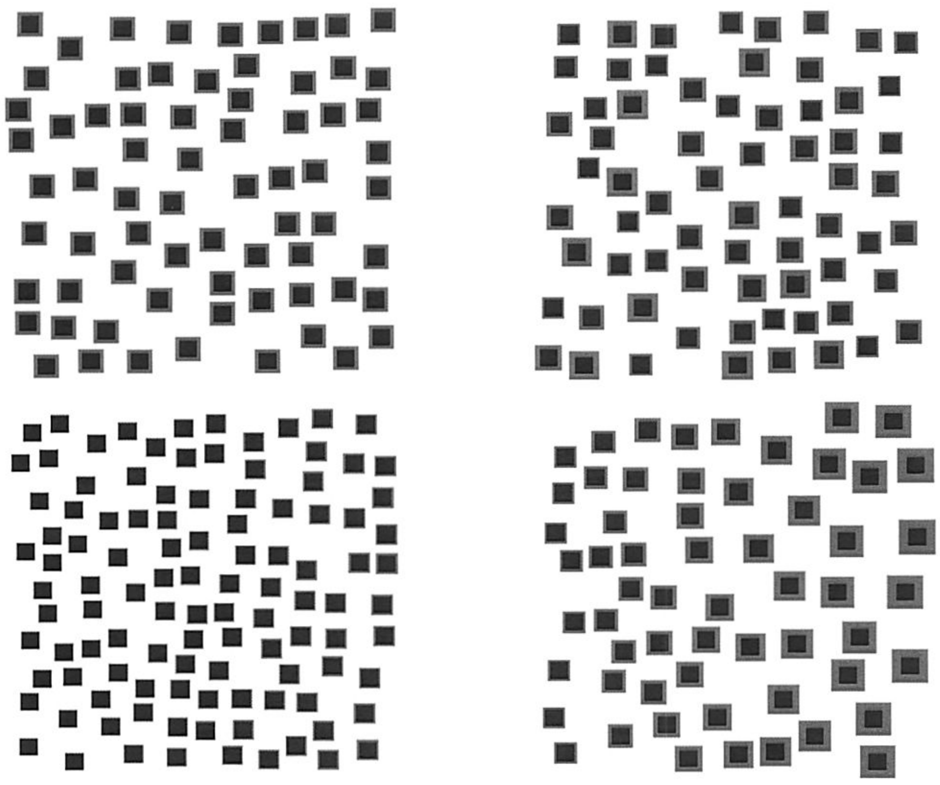

Figure 2 Border width realized using four different "values." The upper left image uses a constant, the upper right a random value, and the bottom two use a gradient variable. The dataset used is a gradient ranging from zero at the left to one on the right.

that ranges from 0.0 on the left to 1.0 on the right. Thus, the width will vary between 0.0 and 1.0 among the instances placed across the image.

```
BORDER WIDTH Variable gradient
```

The second syntax allows the data variable to be "scaled" in to a different range instead of directly using the number sampled from the data. This final example specifies that width should be dependent on the gradient but instead of using the sampled number it should be scaled in to the range [1.0, 5.0]:

```
BORDER WIDTH Variable gradient 1.0 5.0
```

The four resulting images are shown in Figure 2.

There is a minor subtlety to consider when dealing with values, however. There are two types of properties in a SciVL file, those that require a value between zero and one ("normalized") and those that accept any number. Transparency, for instance, must be normalized while border width need not. Therefore, when a value is specified for a normalized property it must be sanity checked by the engine. This simply means that a constant and the random/data-driven ranges must be within [0, 1].

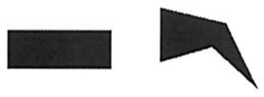

Figure 3 Icons created by the example files.

2.2 Icon Layer

An icon layer consists of an arbitrary number of icons and fully describes the "look and feel" of each one by providing values for

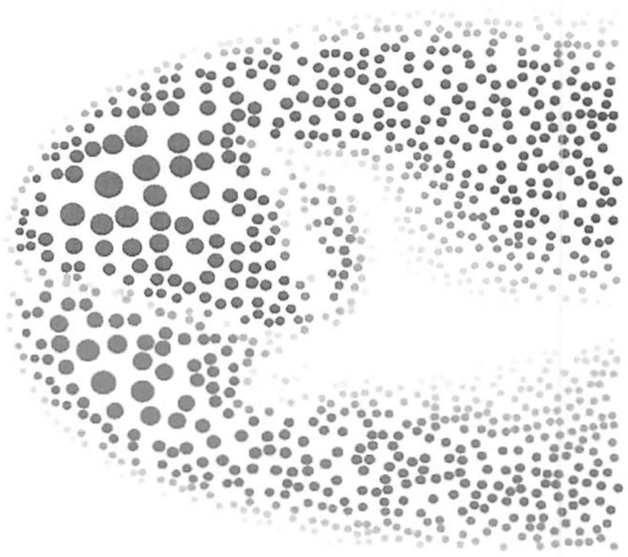

Figure 6 Vorticity visualized with red (clockwise) and green (counter-clockwise) circles. The size of the circle is proportional to the magnitude of the vorticity. Failures are used to prevent the icons from appearing when the magnitude is very low. Instead of having the icons simply disappear, creating a false threshold, transparency allow the icons to gradually fade out.

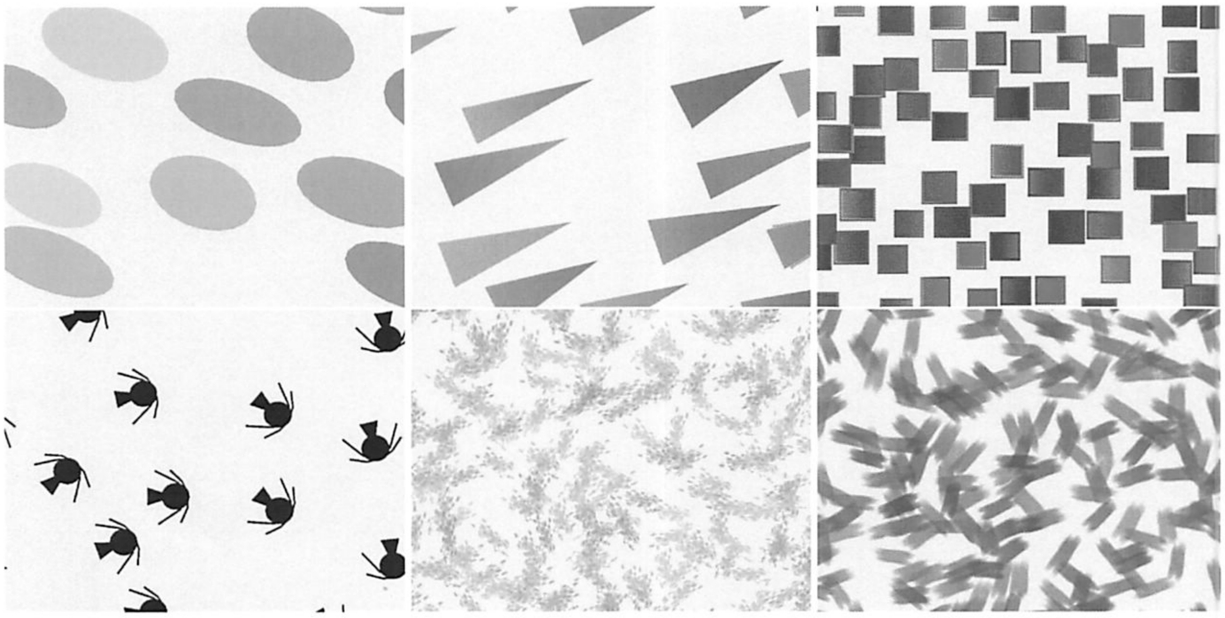
Figure 4 Different discrete icons.

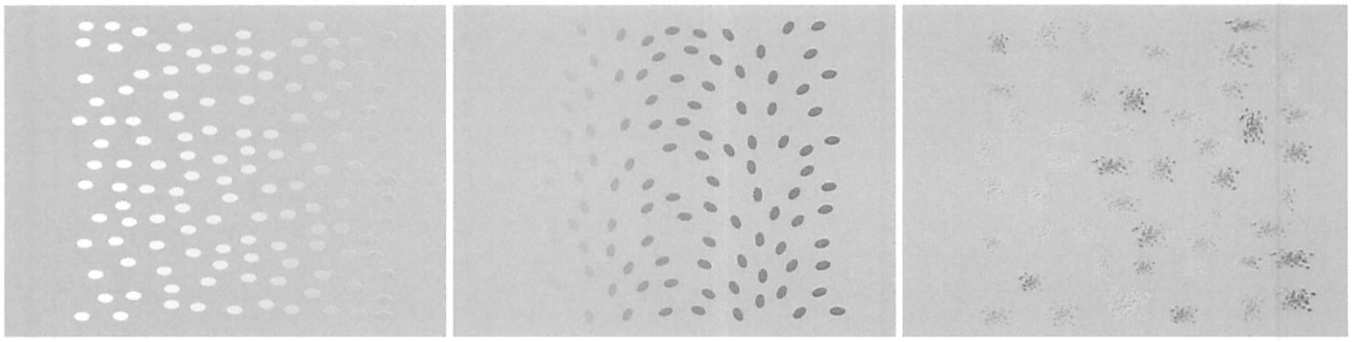
Figure 5 The basic properties of an icon controlled by SciVL. The left image shows color changing, specfically saturation. The middle image shows orientation and transparency, and the right image shows texture. Again we use a gradient dataset.

the many properties of an icon. The features of an icon that control its size and shape are: path, size, and shape. The path is simply a list of points that the icon will "follow" when it is rendered, forming the "spine" of the icon. This controls the size of the icon in one direction. The size value controls the size of the icon in the other direction. Shape is one of rectangle, triangle, ellipse. Consider the following examples (note this is not actual SciVL syntax):

```
PATH (0, 0) (5, 0)
SIZE 2
SHAPE rectangle

PATH (0, 0) (3, 0) (2, 3)
SIZE (3, 0)
SHAPE triangle
```

The resulting icons are shown in Figure 3. More complicated icons are seen in Figure 4.

Note that each point on the path is treated as an offset from the point preceding it. Also, a triangle has two sizes, one for the base width and a second for the tip width.

An icon has traditional visual attributes: color, transparency, texture, and orientation. Examples are shown in Figure 5.

There are two properties of an icon that relate to the rendering algorithm: Poisson area and failures. Poisson area controls how close the icon can get to another icon in the same layer. This property allows icons to range from a high degree of overlap to wide separation. See Figure 7 for examples.

Failures specify certain areas of the visualization that an icon should not appear in, causing icons to gradually become fully transparent as they approach areas in the image where they are no longer useful. There is no need, for instance, to show a turbulent charge icon when pressure is very small, as shown in Figure 6.

Finally, an icon can have a border which has three properties: width, color, and transparency. These are shown in Figure 8.

2.3 Colorplane Layer

A colorplane is a simple construct which allows continuous regions of color in a visualization. A colorplane is created by uniformly tessellating a plane with many vertices. The SciVL file is responsible for controlling the appearance of each vertex by specifying: failure, color, transparency, and texture. If a texture is activated then it is stretched so there is no repetition. Then the SciVL file is consulted to determine the color and transparency for each vertex. The failure condition acts the same as it did for an icon; when the condition

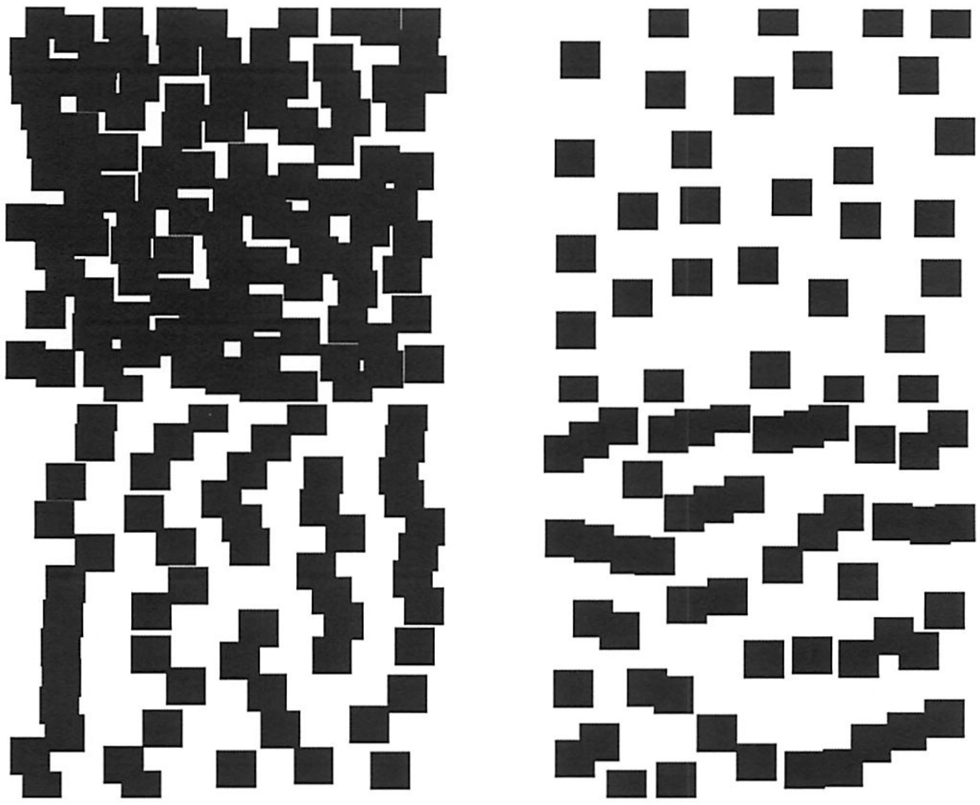

Figure 7 These images show different Poisson areas for an icon. In the upper left icons are uniformly close together and in the upper right they are uniformly far apart. In the bottom left they are spread apart in x but close in y and vice versa in the bottom right.

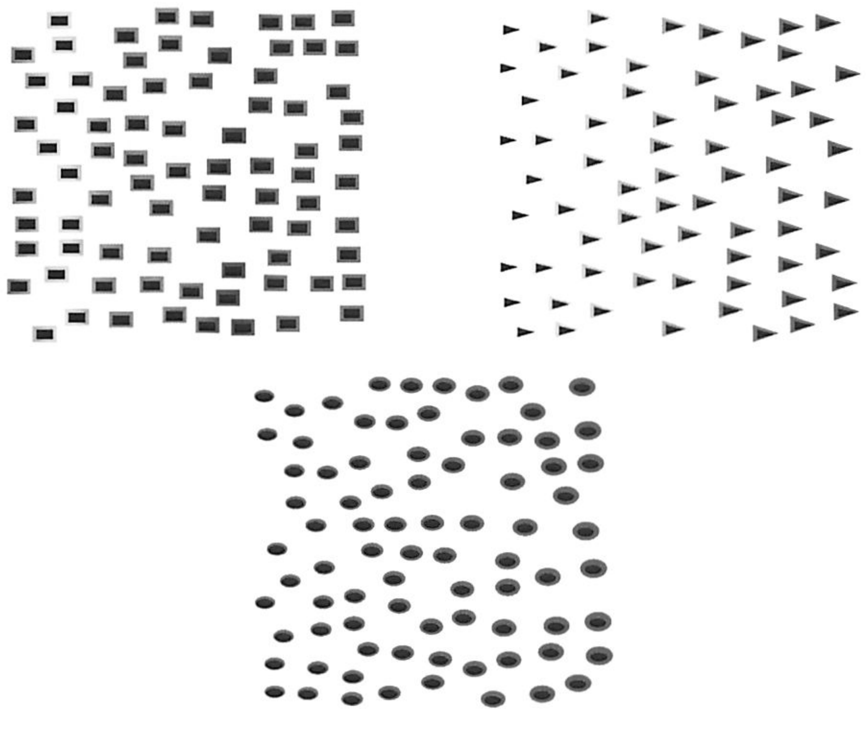

Figure 8 The three properties of an icon's border: color, transparency, and width.

is met the colorplane gradually becomes fully transparent. Colorplanes are showin in Figure 9.

2.4 Streamline Layer

A streamline allows long, continuous visual elements without having to specify a huge number of icon offsets. Controlling the place-

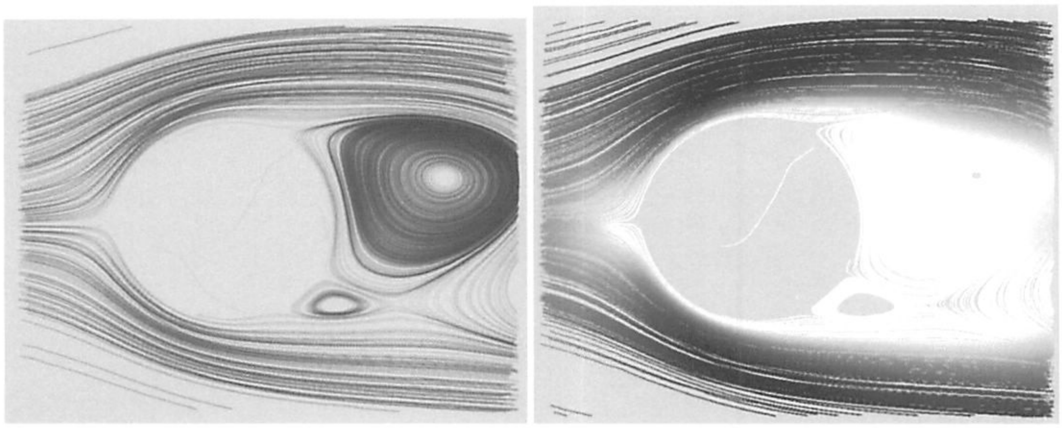

Figure 10 Streamlines running through the dataset of simulated fluid flow around a cylinder. The left streamline is textured and the right has saturation varying according to speed.

ment of streamlines are failures and density. Failures function differently here: they prevent a streamline from being seeded in a specific region and also end a streamline if it enters a failure region. Density controls how the rendering algorithm determines when the layer is "full" and will be discussed in Section 2.7.

There are three properties which control the path the streamline follows: vector, bidirectional, and survival. The vector determines which values are used to integrate the streamline in x and y. The bidirectional flag simply determines if the streamline should go in both directions or not. The survival condition controls if and when the streamline should stop being drawn. An example is:

```
SURVIVAL velocity_magnitude .8 1.2
```

The semantics for this statement are: "Sample the velocity magnitude at the seed point of the streamline. Then, as the streamline is integrated, sample the velocity magnitude at each point. If it goes below 80% or above 120% of the original value, end the streamline."

Finally, the SciVL file also controls the color, transparency, and width of the streamline at each point. A texture can also be applied to the line.

Streamline examples are shown in Figure 10.

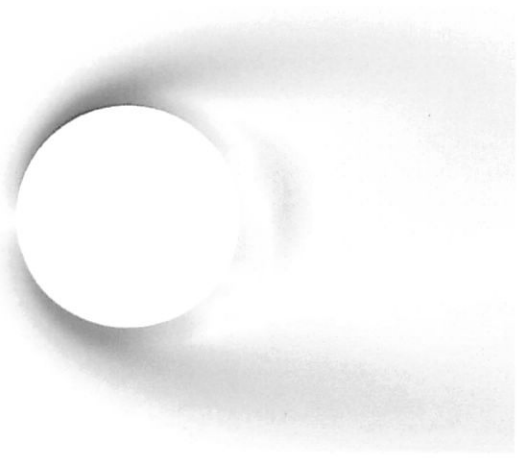

Figure 9 Two colorplanes are used to show clockwise (red) and counter-clockwise (green) vorticity. Opacity indicates the strength of the vorticity. The data is from a simulation of fluid flow around a cylinder.

Remember, for each of these layers, *every* property is tied to a value and can therefore be constant, random, or data-driven across each instance of the visual element.

2.5 Rendering System

Once a SciVL file is completed our rendering system uses it and a scientific dataset to produce an output visualization. We now discuss how this engine works.

At a very high level, the renderer simply loops over all layers in the file from back to front and "fills" them in with the appropriate visual elements. Layers are drawn back to front so occlusion and blending function properly.

The way a layer is filled varies by type. Filling a colorplane layer is simplest: just tessellate a plane, calculate the per vertex colors and transparencies, and draw the plane.

2.6 Rendering an Icon Layer

An icon layer is much more complicated to fill. The essence of the algorithm used is shown here:

```
list<BBox> box_list;

for (each icon in layer) {
  failed = 0;
  while (failed < FAIL_THRESHOLD) {
    P = random point in the visualization;

    if (all failure conditions met at P) {
      if (totally transparent) {
        ++failed;
        continue;
      }
      else save transparency for later use;
    }

    B = bounding box for icon if rendered at P;
    scale B by icon's poisson area;

    if (B overlaps a bbox in box_list) {
      ++failed;
      continue;
    }
    else {
      render icon at P;
      box_list += B;
```

```
        failed = 0;
      }
    }
}
```

This algorithm uses a Poisson distribution to ensure, with high probability, that the layer is as full as it can be. It tries to place icons down at random locations in the image and quits when it cannot find a space for the icon FAIL_THRESHOLD times in a row.

2.7 Rendering a Streamline Layer

The streamline layer uses a similar idea:

```
bool marked_squares[streamx][streamy];
set every element in marked_squares = 0;

while (marked_squares contains a 0) {
  (x,y) = a 0 element in marked_squares;
  marked_squares[x][y] = 1;
  P = (x,y) converted to a point;

  if (all failure conditions are met at P)
    continue;

  add P to streamline;
  do {
    N = next point on streamline;
    if (survival condition NOT met at N)
      break;

    (x,y) = N converted to array indices;
    marked_squares[x][y] = 1;

    if (all failure conditions are met at P)
      break;

    add N to streamline;
  } while (N in range of visualization);

  render streamline;
}
```

This algorithm ensures that there is a streamline running through every "square" in the visualization where not all the failure conditions are met. See Figure 11.

Increasing the streamline density values makes the squares smaller, thereby increasing their number, which means more streamlines are needed to fill them all. Decreasing the density values has the opposite effect. See Figure 12 for an example.

2.8 Realization

When rendering each of these layers, the engine has to convert SciVL values in to actual numbers. Transparency, for instance, might have been specified as:

```
TRANSPARENCY Variable pressure 0.5 1.0
```

When actually rendering the icon we need a number, however, and this happens through a simple process we call realization. Realizing a constant is trivial, simply return the number specified in the SciVL file. For a random value we generate a random number in the range specified in the file from a uniform distribution. A data-driven value requires looking up the appropriate variable in the dataset at the current location of the icon and scaling the number as necessary.

2.9 Jitter

The user can also inject "jitter" in to the visualization. When jitter is enabled, the user specifies a mean and a standard deviation which defines a normal distribution. Then, whenever a value is realized, we sample from that distribution to get a jitter value, which is added to the realized value. It does not make sense to jitter certain values when they are supposed to be continuous. Imagine a colorplane where color is tied to a data variable. Since the plane is highly tessellated, adding jitter at each vertex would cause a very undesirable, discontinuous effect. We are still considering how to control which values should not be jittered.

Jitter was added to the system because we felt the images often looked too precise. Slight variatons can be useful to give the visualizations a more "organic" feel, more like they were done by hand. An example is shown in Figure 13.

3 Results and Discussion

At this phase in the work, our results are entirely anecdotal. In the near future there are plans to conduct formal user studies to evaluate our success, but for now we only present the feedback we have received.

The first experience we had with the system was trying to approximate an image from Kirby et al. We used the same simulated fluid flow around a cylinder shown in previous examples. Once the datafiles were in place it took under 15 minutes to design a file with visual elements similar to those used by Kirby and under 5 minutes to render the output image. Naturally, we did not get the elements exactly right, but the tweaking process was extremely easy and in less than 30 minutes from when we started we had a loose approximation of their images. A comparison of the two images is shown in Figure 14. These images are visualizing velocity (gray triangles), vorticity (blue and yellow colorplanes), turbulent charge, and turbulent current (brown and gray rectangles).

The real power of the system was demonstrated when we wanted to create new visualizations of the same dataset. Using the SciVL file we already had it was easy to create new visual elements by "borrowing" features from old elements. We created four different visualizations in about five minutes each, as shown in Figure 15. Our initial goal of rapid prototyping was accomplished by using SciVL.

In the previous example we ourselves were the users and designers of the visualizations. In the future, ideally, scientists would use our system to visualize their data by designing the visualizations themselves. To test this process, the image shown here was created by a

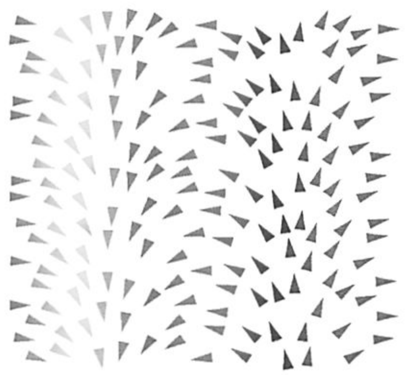

Figure 13 A jittered visualization. The mean is zero and the standard deviation is .0001.

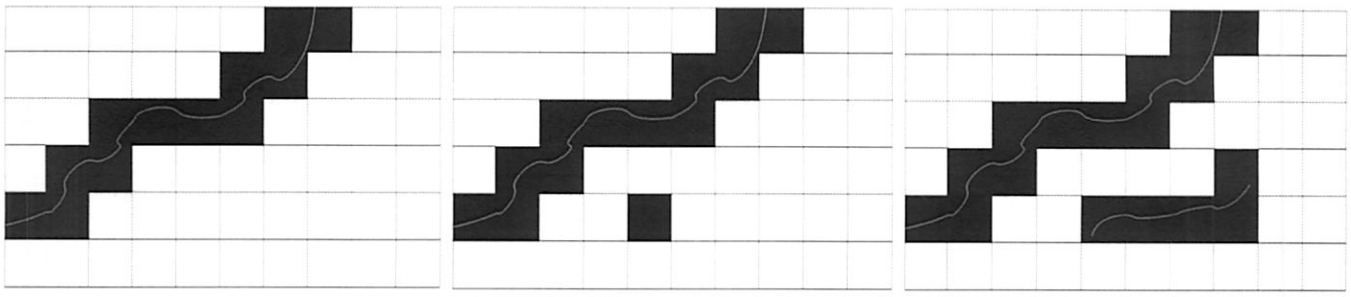

Figure 11 Our streamline seeding algorithm. In the left image one streamline has been drawn, "marking" many of the "squares" in the visualization. In the middle image we've chosen a random, unmarked square to start a new streamline and marked it. In the right image we've begun to draw the new streamline, marking squares as we go.

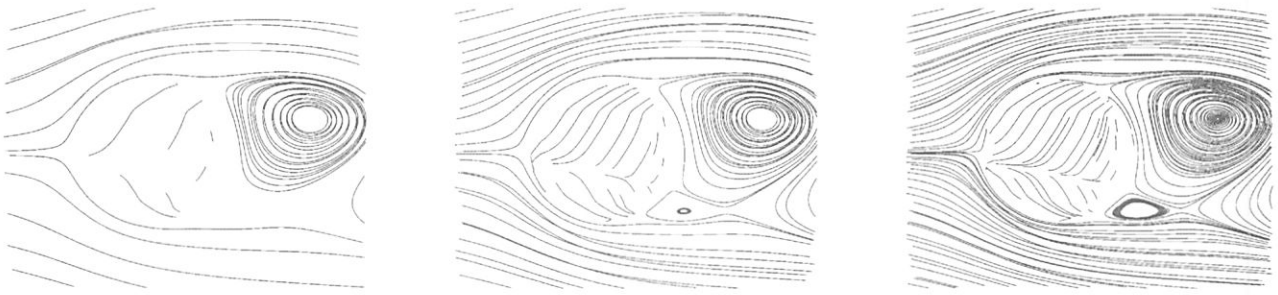

Figure 12 The effect of the density control on a streamline layer. The left image has density set to 10, the middle to 20, and the right to 30, for both x and y. Notice how increasing the densities creates more streamlines.

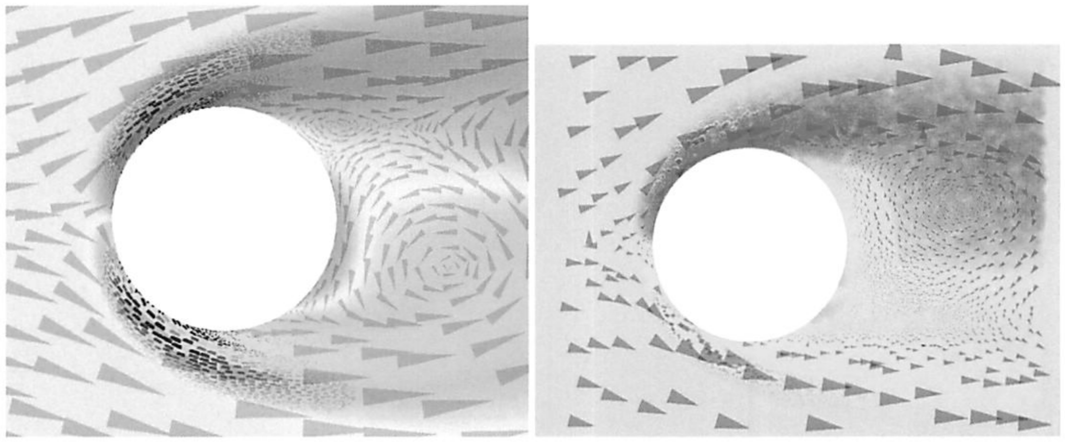

Figure 14 On the left is the image created by Kirby et al. to visualize simulated fluid flow around a cylinder. On the right is our first attempt to approximate this visualization using SciVL constructs.

polarimetry expert after a brief introductory session to SciVL and a few test examples using artificially created 2D scalar fields.

Polarimetric radar data presents the challenge of correlating a high number of scalar variables gathered over wide areas of terrain from an airborne set of antenna. The expert was able to overlay three different colorplanes so that their colors would combine to provide a meaningful explanation of the data. He was particularly impressed by the ease of use of our language, which allowed him to test many different variable bindings very quickly. A resulting image is shown in Figure 16.

The main issue users had in these tests was the limited handling of 2D data. The formatting of the datafiles has to be very specific so we must convert any other format into our own. At this point in the project we are concentrating on the visual language but we are working to improve the data handling capabilities. One of the first tasks is to allow algebraic operations on the variables within the SciVL file. Currently, a separate datafile must be created if a derived value is required, but it would be nice to do the following instead (for example):

```
ORIENTATION Variable atan / y x
```

As another evaluation step, we asked a local design professor to evaluate our visual variables. This collaborator has worked with us on previous scientific visualization projects for fluid flow and medical visualization. We presented him with a set of images showing each individual "axis" of our visualization space. He highlighted the expressiveness of SciVL and commented on the capability of our icon-placement and streamline-placement algorithms. He referred

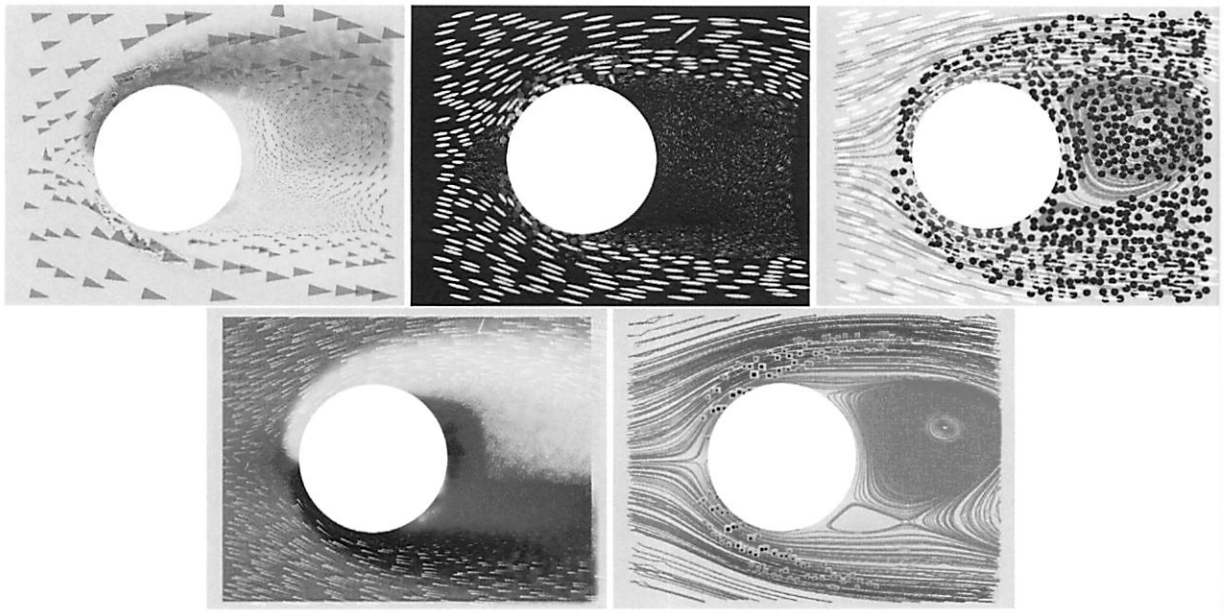

Figure 15 Our language allows us to quickly describe visually rich images for scientific visualization. These examples show six different visualizations of a 2D fluid flow simulation in which scientific data defines the appearance of the visual elements in the image. Velocity of the flow, vorticity, turbulent charge and turbulent current are all represented in each image. The layering of the elements and their mapping to data can be easily modified using our language. The visual elements that form each layer can be icons of different shapes, colored planes or streamlines.

to the Gestalt grouping laws and suggested ensuring that our Poisson distribution does not generate unwanted patterns.

He pointed out a color-banding artifact in our HSV to RGB conversion which has us considering switching to LAB color space. He also would like to specify custom color transitons instead of linearly progressing through whatever space we happen to be using. Instead the user could provide a number of colors and the system would generate a palette.

He also suggested that jitter be controllable per attribute. We are currently implementing this feature.

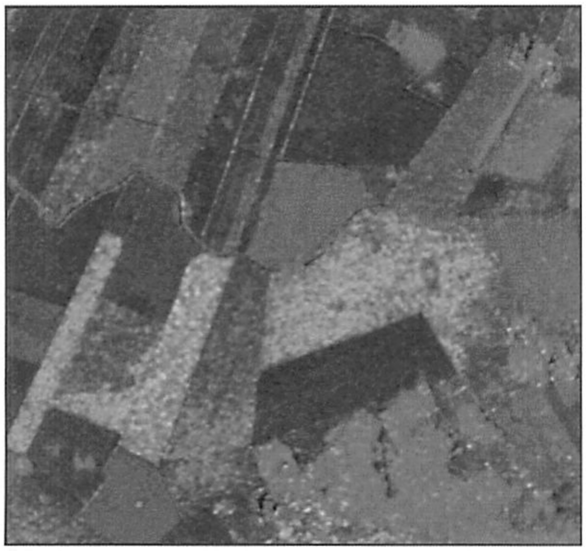

Figure 16 Example using radar polarimetry data. The image shows a colorplane combining the total power of the signal with a measure of the phase difference (alpha angle) between the signals from two colocated orthogonal antenna. Shades of brown yellow show bare fields, greens show vegetated areas and blue show dihedral interactions from crop stalks or buildings.

Our system has also been used by Vote et al.[11] to produce "previsualizations," which are then fed in to the image analogies[12] algorithm to create art-based visualizations. While there have been minor issues with this process it is far faster than designing a previsualization by hand.

4 Future Uses and Future Work

The language and rendering system we have developed are a step in our project to optimize multivariate data visualizations using genetic algorithms. We hope this system will allow us to quickly converge on good visualizations and discover some general rules for making good images. The addition of these design rules to our language will provide non-expert designers with a way of creating better visualizations without blindly iterating through many trials. Preliminary results with a simple genetic algorithm indicate that the basic concept is sound. Now that the language and rendering are finished the algorithm needs to be considerably revised.

Another parallel task is using the system to create a series of images, each one using two visual variables to visualize two data variables (a gradient in x and a gradient in y, for instance). Running a user study with these images will allow us to determine another set of "rules" about which visual variables work well in tandem when showing different data variables.

The main area of future work on SciVL itself is integrating it with a GUI, a la GlyphMaker. This will allow easier and more intuitive design of visual elements. It will also allow more complicated expressions; in a range, for instance, instead of specifying minimum and maximum values and linearly interpolating, control curves would allow the user more sophisticated control over their output.

Another use of the GUI would be to take an output visualization and provide a "gradient" to follow to improve the visualization. A user, by tweaking visual variables, would indicate a direction to travel in to improve the visualization by that user's standards.

5 Conclusions

We have presented the design of SciVL, a visual language for the description of scientific visualizations. Our initial results are from 2D scalar and vector-field datasets. Generating these examples has successfully evaluated, if only anecdotally, our goal of quickly creating complex and precise data-driven visualizations as well as facilitating their modification during the iterative design process. Our ongoing collaboration with design and art experts has provided us with reliable and accurate feedback on our language design and the classification of visual elements and styles.

These encouraging initial steps have shown promise for the use of our language for visualizing more complex datasets. Our main goal is still to create a perceptually optimal classification of visual elements and styles to be used in the design and synthesis of scientific visualizations.

6 Acknowledgements

I would first like to thank Daniel Acevedo who worked with me on the design of SciVL and also helped with the implementation. He also helped with the writing of this paper and I'm very grateful for all his support. Cullen Jackson and Eileen Vote were also critically involved throughout the entire process and they contributed greatly to the success of this project. I would also like to thank my advisor, Professor David Laidlaw, for the opportunity to work on this project. He also provided invaluable advice and guidance throughout the project. David Karelitz and Daniel Keefe worked with us during the design phase and also provided the foundation for our rendering engine; both of these contributions were tremendously valuable. Dean Turner and RISD Professor Fritz Drury provided the anecdotal feedback I used in this paper, so I'm very thankful for their feedback. I'd like to thank Professor Andries van Dam for being my second thesis reader and also inspiring me to work in computer graphics. Mark Ture also provided valuable feedback on this paper. Finally, I'd like to thank my family for so many things, especially their unwavering love and support over the years.

References

[1] J. Bertin. *Semiology of Graphics*. University of Wisconsin Press, 1983.

[2] Edward Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 1983.

[3] Edward Tufte. *Envisioning Information*. Graphics Press, 1990.

[4] William S. Clevel and and Robert McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79(387):531–554, September 1984.

[5] Charles Wallschlaeger and Cynthia Busic-Snyder. *Basic Visual Concepts and Principles for Artists, Architects and Designers*. McGraw Hill, 1992.

[6] Victoria Interrante. Harnessing natural textures for multivariate visualization. *IEEE Computer Graphics and Applications*, 20(6):6–11, November/December 2000.

[7] Russell M. Taylor II. Visualizing multiple scalar fields on the same surface. *IEEE Computer Graphics and Applications*, 22(2):6–10, March-April 2002.

[8] Colin Ware. *Information Visualization: Perception for Design*. Morgan Kaufmann Publishers, 2000.

[9] W. Ribarsky, E. Ayers, J. Eble, and S. Mukherjea. Glyphmaker: Creating customized visualizations of complex data. *IEEE Computer*, pages 57–64, July 1994.

[10] Craig Upson, Thomas Faulhaber Jr., David Kamins, David H. Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, and Andries van Dam. The application visualization system: A computational environment for scientific visualization. *Computer Graphics and Applications*, 9(4):30–42, July 1989.

[11] E. Vote, D. Acevedo, C. Jackson, D. Keefe, D. Karelitz, J. Sobel, and D. Laidlaw. Art inspired visualizations. Submitted for review for the Sketches and Applications Proceedings of SIGGRAPH 2003, 2003.

[12] A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin. Image analogies. In E. Flume, editor, *Computer Graphics (SIGGRAPH '01 Proceedings)*, pages 327–340, August 2001.