Design Choices when Architecting Visualizations

Diane Tang[*] Stanford University
Chris Stolte[†] Stanford University
Robert Bosch[‡] Stanford University

Abstract

In this paper, we focus on some of the key design decisions we faced during the process of architecting a visualization system and present some possible choices, with their associated advantages and disadvantages. We frame this discussion within the context of Rivet, our general visualization environment designed for rapidly prototyping interactive, exploratory visualization tools for analysis. As we designed increasingly sophisticated visualizations, we needed to refine Rivet in order to be able to create these richer displays for larger and more complex data sets.

The design decisions we discuss in this paper include: the internal data model, data access, semantic meta-data information the visualization can use to create effective visual encodings, the need for data transformations in a visualization tool, modular objects for flexibility, and the tradeoff between simplicity and expressiveness when providing methods for creating visualizations.

1 Introduction

Over the past five years, we have tackled many real-world analysis problems using visualizations built within the Rivet visualization environment [Bosch et al. 2000], including mobile network visualizations [Bosch et al. 2000], the Polaris user interface [Stolte et al. 2002b][Stolte et al. 2002c] and several multiscale visualizations [Stolte et al. 2002a]. Over this period, as we have gained experience and tackled successively more complex and difficult problems, we have needed to revisit certain design issues.

The design choices we faced as we refined Rivet ran the gamut from architecture refinements for handling more sophisticated data sets and visualizations to implementation issues. This paper discusses the following major design issues:

1. Data Model: a data model can simplify the user's task by providing an easily understood abstraction and flexibility in the analyses that can be performed on the data.
2. Data Access: the visualization tool should adapt to the user and make data access easy, rather than making the user adapt to the tool.
3. Data Transformations: because users want to be able to transform data during the process of analysis, visualizations need to provide transformation capabilities.
4. Sophisticated Meta-data: to create effective visualizations and support the analysis process, a visualization tool can take advantage of more semantic information than is usually provided.
5. Modularization of Data, Visualization, and Interaction Objects: finding the right granularity and API of objects allows us to combinatorially interchange and combine objects for maximal flexibility and extensibility.
6. Specification *and* Scripting: rather than providing only a programmatic or only a descriptive interface, we gain power and simplicity by using specification and scripting together.

[*]Now at Google Inc., e-mail:diane@google.com
[†]e-mail:cstolte@cs.stanford.edu
[‡]Now at VMware, Inc., e-mail:rbosch@vmware.com

In addressing these issues, we had to balance performance, flexibility and expressibility, and the amount of user expertise required. We discuss the tradeoffs involved throughout the paper.

These design issues fall within a larger context: the process of building a visualization. One way this process can proceed is via four basic steps:

1. Design: the designers must decide how to visually represent the data and which interactions are needed.
2. Build: given this design, they then need to actually build the tool.
3. Use: the tool is given to end users to analyze their data.
4. Iterate: with use, various refinements are usually needed, such as collecting different data, changing the visual representation, or adding additional user interaction capabilities.

While there are other ways of describing the process of building a visualization, we use these four steps to help frame the discussion in this paper. Specifically, given that the design decisions we discuss in this paper primarily deal with underlying infrastructure issues, such as data management, rather than the *visual* design, our primary focus is on steps (2) and (4) above. Specifically, all of the data choices (choosing a data model, importing data, supporting data transformations, and adding meta-data information) need to be addressed when building a visualization. In contrast, choosing the granularity of objects (data, visual, and interaction) and deciding between specification and scripting also occur in the build step but mainly impact how easy it is to iterate through the design process.

These steps apply in the design of both highly customized tools, such as comparing trees, as well as in the design of much more general tools. Thus, the same design issues are faced regardless of the type of tool being built. The main difference will be that different choices may apply depending on the specific tool being built. Our goal in this paper is primarily to explain these issues and some of the tradeoffs involved in possible design choices. We also explain our own decisions and discuss alternative choices, thus reducing the duplication of effort and the time needed to create visualizations.

The layout of the rest of this paper is as follows. We first present a review of the original architecture of Rivet [Bosch et al. 2000]. We then visit each design decision in turn, presenting the problems we faced, the possible choices and their ramifications, and the decision we made and the associated advantages and disadvantages. Related work will be presented throughout these sections, focusing on the aspects that are relevant to the issue being discussed; note that many of the systems being discussed are quite large and complex and also address many issues that are beyond the scope of this paper. We conclude and present directions for future work in Sections 9 and 10.

2 Background: Rivet

The original architecture for Rivet consisted of four basic portions: the data model, the visual objects, the mapping from data values to visual representations, and the user interaction components [Bosch et al. 2000][Bosch 2001]. The key design decisions in Rivet focused on modularity: choosing the right granularity for objects in order to maximize flexibility.

Originally, Rivet used only basic meta-data consisting of field names and types and a simple relational data model (i.e., tuples and

tables). Because of the focus on computer systems data, which often lives in log files, the only data import functionality was regular expression parsing of flat text files. Rivet also provided internal transformations, which, because they took tables as input and output, could be composed into an arbitrary transformation network, with every intermediate result visualizable.

Visually, Rivet used the notion of metaphors and primitives: Metaphors draw tables by laying out tuples spatially within the display, and primitives generate a visual mark for each individual tuple. Selectors identify data subsets and can be used to determine which tuples to highlight or elide.

Some of the issues we discuss in this paper describe how and why we modified the Rivet architecture. Other portions of Rivet remained stable. Rivet still uses encodings to map data values to visual representations, with metaphors using spatial encodings to determine the position and bounding box of tuples, and primitives using retinal encodings [Bertin 1983] to determine the visual properties, such as color and size, when rendering a particular tuple.

Rivet uses an event model, with objects raising events in response to user actions such as mouse clicks. To build visualizations, users write scripts that determine how data is imported, link data to visual objects, and bind actions to these events to create the effects of the user interaction. These actions can consist of an arbitrary sequence of operations, as created by the script-writer. For example, the script-writer chooses whether a mouse event that leads to a zoom corresponds to a semantic or optical zoom. Rivet also uses the listener model to propagate updates, so that if the user changes a filter, all associated data transformations are updated, which then updates the associated visual representations.

This architecture allowed us to create many interactive visualizations for exploratory data analysis. However, as our visualizations became more sophisticated, we needed to revisit our design decisions in order to provide increased functionality and generate more effective visualizations. The rest of the paper discusses these design decisions both in general and within the context of Rivet.

3 Data Model and Representation

The first design decision is what data model to use internally within the visualization system. This decision impacts almost every other design decision, in large part because the data model is the conceptual model of the data presented to the user. It also impacts other choices, such as:

- Data access: can this model accomodate different types of data and different types of data sources?
- Data transformations: what types of data transformations can be done within this data model?
- Meta-data: what meta-data information goes along with this type of data model?
- Moduralizing data, visual, and interaction objects: How easily can this data model be mapped to a visual representation? Different visual representations?

One common data model used in visualization systems, especially general visualization tools, is the relational model (essentially, a tuple- and table-based model). Before we delve into a detailed discussion of some nuances of the relational model, we discuss its overall advantages and disadvantages.

The relational model has several advantages. It provides an easy-to-understand conceptual model for the user. It is also easy to implement in its basic form. Another major advantage is that it provides access to many existing datasets, from computer systems logs to biological data to corporate data warehouses, that are already stored in relational databases without needing any additional translation. Another related advantage is that because the relational model is so prevalent, transferring data between tools also becomes easier. It is flexible and can be easily mapped to many different visual representations (bar charts, scatterplots, line charts, Gantt charts, etc.). Additionally, it provides a general abstraction layer so that all data sources look equivalent (see Section 4: once the data is in the visualization model, it does not matter to the user whether the data comes from a flat file or a data warehouse), and it provides a general interface to data transformations: the input and output of data transformations look equivalent, so that data transformations can be composed regardless of their actual implementation.

Conversely, one disadvantage to this model is that it assumes some homogeneity to the data. Some types of data, such as graphs or ragged trees, are difficult to represent using the relational model. Furthermore, even given such a representation, some data transformations, such as a prefix tree traversal, can be non-trivial. This limitation is especially problematic since this type of data is some of the richest and most interesting data to visualize. For these data structures, having a custom data model may be a solution that is both easier to design and likely to yield better performance than the more general relational model.

Another disadvantage that we discuss here is that certain data analyses are difficult to perform using the relational model. While some queries, such as spatial queries or dynamic queries (e.g., fast computation of filters), can be done using specialized indices or data structures, one type of query that is inherently difficult to perform in the relational model is data reshaping, such as transposing a relation representing a correlation matrix [Wilkinson 1999]. For example, consider a data set containing microarray data, with each microarray having an expression level for a set of genes. One way to structure the data is to have each tuple (row) represent a microarray and each column represent a gene, so that each tuple contains all the expression levels for the genes. This structuring of the data is compact, but limits the set of possible analyses; for example, while we can group and filter by the microarrays, we can only filter the genes. We cannot group by the genes since they are in the columns rather than the rows. Another way to structure the data is to have three columns in each tuple, with one column for the gene name, another column for the microarray data, and a third column for the expression level. This structuring is less compact but more flexible. This example illustrates that the data organization impacts which analyses can be performed, therefore impacting the flexibility of the visualization tool as well.

Given these advantages and disadvantages, we now discuss some nuances to the relational model, especially in comparing the needs of a visualization tool to the more general database implementations.

In the database community, the term "relational model" encompasses several concepts: the actual data structure (tuples and relations), the data organization (normalized versus de-normalized schemas[1]), and the relational algebra and query language for data manipulation. In this section, we focus on the first two aspects of the relational model and the associated advantages and disadvantages, as well as other possible data model choices. Data query and analysis are discussed further in Section 5.

In the database community, the standard relational model has a data structure consisting of unordered sets of tuples (e.g., facts or rows in a table), and tables consisting of homogeneous tuples. The unordered and homogeneous nature of tuples in a relational table is insufficient for visualizations because orderings are often needed both for analytical (discussed more in Section 5) and for drawing purposes: tuples are drawn in back-to-front order, and nominal domains, which are traditionally un-ordered, must sometimes be ordered for display, such as when drawing axis labels. Thus, in Rivet, we enhance the relational model with these orderings.

The organization of the data, i.e., the data schema, also impacts the effectiveness of the relational model for visualization. In addressing this issue, lessons from the database community are appli-

[1] A normalized schema divides the data into multiple tables to minimize duplication of data, both within a single table and between tables.

cable. Transaction processing (OLTP) was one of the first database applications. Because OLTP requires quick update capabilities, normalized data schemas are optimal: usually only one table needs to be updated. In contrast, analytical processing (OLAP) often consists of retrieving many tuples across many relations, which requires many expensive joins when using a normalized schema. Thus, using a de-normalized organization is often more optimal.

Alternative Data Models: The relational data model is not the only model commonly used in the database community. Other data models are the object-relational data model [obj n. d.] as well as a hierarchical XML data model [Bourret n. d.].

The object-relational data model essentially tries to have each table correspond to a table in a database (e.g., for an employee infrastructure, there might be an employee object and an address object, each with corresponding database tables). This type of model can be easily mapped to object-oriented code (e.g., Java, C++) using tools such as JDO (Java Data Objects). This type of model yields higher performance when traversing an object hierarchy is the common operation, and could therefore be used in some of the situations where the relational data model does not perform well, such as ragged hierarchies. While the object-relational model may handle complex cases where a great deal of flexibility is needed, in the general case, the object-relational model is most similar to the fully normalized OLTP model above and is designed more for quick updates and reads about specific items by traversing the hierarchy rather than large reads as is common in a lot of analytics.

The XML model may also be an alternative when the relational model is not flexible enough. In this model, XML is used to specify both the data and the meta-data, and is becoming the standard in some fields, such as biology. However, even XML proponents recognize that the XML as a database is still in its infancy. For example, the current XML query languages (XPath, XQuery) do not support basic operations such as sorting, joins, grouping, etc. that are needed for most analytical purposes.

Related Work: In existing general-purpose visualization tools, the most common data model choice is indeed the relational model. This model is used in Snap-Together Visualizations [North et al. 2002], DEVise [Livny et al. 1997][Livny et al. 1996], Tioga-2 / DataSplash [Aiken et al. 1996][Woodruff et al. 2001], Sage / Visage [Roth et al. 1997], and VisDB [Keim and Kriegel 1994]. These tools focus primarily on normalized relational data, since they focus significant effort on how to let the user easily perform joins (a non-trivial problem that Rivet does not tackle). However, with regards to the data model itself, there is no discussion of the pros and cons of the relational model, or other choices, in these papers. With regards to data models, Wilkinson [Wilkinson 1999] discusses many of the disadvantages of the OLAP model with regards to needing access to the raw data in order to properly compute common statistical measures.

4 Generalized Interface for Data Access

Data access is a key issue in visualization since users often want to simultaneously visualize data from many sources and the data from each source may change. The cycle of data exploration is one of hypothesis, experiment, and analysis, with each analysis either producing a result or suggesting a new hypothesis. With every iteration through the analysis cycle, new data may be collected, the format of the data being collected may change, and so on. Users do not care about tradeoffs or where their data or meta-data is stored; they only want to be able to import all of the data into the visualization system and then proceed with their exploration and analysis. Thus, making data access easy is important for making a visualization a commonly used analysis tool.

One might think that providing access to a database would be sufficient. Often, however, users need to combine data from multiple sources. Hierarchical relationships may be stored in a flat file, historical data may be stored in a SQL database, commonly used aggregates of the historical data may be in a datacube, while other data may be stored in a text log file.

One advantage of using a relational model is that we can decouple the actual data source from the visualization tool and provide a general, easy-to-use interface for importing data. Users can easily specify the different data sources, and once in the visualization tool all data sources should appear equivalent. To achieve this decoupling, Rivet uses three concepts:

- The "ordered" relational data model (Section 3): all data sources are represented using this abstraction, and thus appear equivalent to the user.
- Named data sources and a "data repository": data access is easy for users since they can use the name to query the repository to retrieve the data.
- An easy to read, and therefore easy to change, XML file specifying the provenance information about a data source. Thus, rather than writing a script to connect to a database or to parse a CSV file, the information can be quickly specified in this file. Note that more work, such as writing a script or writing a specialized translation module, is required to parse more complex text files using regular expressions or to transform other types of data to the relational model.

Because having easy data access is so important, we put the time into implementing a regular expression parser, CSV parser, and OLE DB drivers to connect to SQL databases and MDX datacubes to ensure that Rivet users could easily import their data. We initially provided only a regular expression parser for users; this functionality is quite flexible but requires a fair amount of expertise and work to import data correctly. In contrast, directly allowing relational data to be imported, whether from a CSV file or a database, is more restrictive but far easier, again showing a tradeoff between flexibility and ease of use.

Note that in order to achieve a full decoupling of the data sources from the visualization tool, users must also be able to perform analyses on the data without caring about the implementation of the transformations and whether they are done within the visualization tool itself or pushed to the data source (e.g., a SQL query performed on the database). In other words, the visualization tool must make all data sources appear equally expressive to users (see Section 5). One drawback to this approach is that it can be misleading to the user if the latency of the different data sources varies widely.

Related Work: Many existing visualization systems focus on the issues that arise after the data is already in the system and do not discuss data access extensively. Snap-Together Visualizations [North et al. 2002], and Tioga-2 / DataSplash [Aiken et al. 1996][Woodruff et al. 2001] mention their reliance on data stored in databases. DEVise [Livny et al. 1997][Livny et al. 1996] is one exception: they discuss how to support accessing large data sets that do not fit in memory and how to connect to both databases and alternative data sources. Specifically, they provide an API so that the user can program a converter / extractor for any data source. This approach is more flexible and extensible but requires more expertise. Note that this expertise is needed only once per additional type of data source, however.

5 Data Transformations

During the course of analysis, users will want to see different views and transformations of the data, such as aggregates, drill downs and roll ups, filters, and so on. In this section, we discuss several of the design decisions involved in providing this functionality within the context of a visualization tool, specifically:

1. Do visualizations need to provide transformation capabilities?
2. Which transformations are needed?
3. Where should these transformations be implemented?
4. How do users call these transformations?

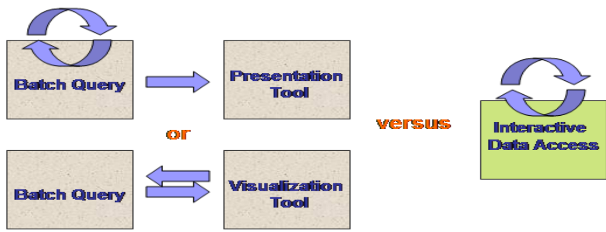

Figure 1: Data analysis tools are often separate from data visualization tools, thus requiring that users perform queries in one tool, and then visualize the results in a separate tool. To perform analysis, users need to context switch back and forth between two tools. We advocate a tighter integration to minimize the context-switch time for effective visualization and analysis.

5. What issues are involved in implementing these transformations?

Do visualizations need to provide transformation capabilities (independent of implementation)?

The cycle of analysis involves the user asking a question, choosing a view that will hopefully answer the question, creating the view, examining it, drawing conclusions, and then iterating.

Having as tight of a loop as possible between performing these analyses and seeing the resulting visualizations will encourage more analysts to use visualization (Figure 1) for analysis and not just presentation. There is a quite a spectrum of options in creating this loop for visualization designers. One extreme is to require users to leave the visualization environment to do data transformations, followed by either re-importing the data or creating a new visualization; this option creates a high barrier between analysis and visualization that hampers many users. A middle choice is to have a single tool with methods for analyzing the data and methods for displaying the data; Excel is an example of this type of tool. At the other end of the spectrum is a tool like Polaris [Stolte et al. 2002b], in which users create specifications (Section 8) that the system uses to generate both the requisite transformations and the visual display.

We believe that users want to focus on the data and its analysis, rather than on the details of data manipulation. Thus, to create as tight of a loop between analysis and visualization as possible, the user needs to be able to perform analyses within the context of the visualization tool.

Which transformations should a visualization provide?

Performing analyses within the context of the visualization tool means that users need to be able to execute data transformations within the tool. Note that this is still independent of where the transformations are actually implemented: they could be translated into database queries, internal transformations, some combination, etc.

Given this need, the next question is which data transformations are necessary? For general analyses, we have found several transformations that are commonly used in the analysis process:

- **Count and Aggregate:** In general, analysis is about counting or aggregating how many things are in various buckets. Aggregation enables the user to abstract large data sets into understandable and manageable sizes. For example, how many oranges come from the states California and Florida? What's the average profit per state? This high-level analysis idea translates into queries of the form group by and aggregate. Common statistical aggregation operations that are needed are count, average, sum, min, max, median, etc.
- **Sort:** Tuples need to be drawn in some order, so that "more important" tuples are drawn on top; this visual ordering of tuples is often very useful when analyzing data, since it can reveal patterns that would be hidden using a different ordering [Bertin 1983]. Sorting is one way to get this order. Sorting can also be used to order tuples when drawing non-point primitives such as lines or polygons. Also, sorting followed by filtering is one way (not necessarily the fastest) of computing ranking transformations, e.g., the top ten tuples according to a particular metric.
- **Filter:** Especially for large data sets, filtering to display only the "interesting" range of the data, e.g., only the Western states or only the top ten percent, is one of the most useful transformations for reducing clutter in the display.

This list contains the transformations we have needed consistently in the many visualizations we have built using Rivet. This list is not complete, because more specialized transformations are sometimes needed, either for performance reasons or for unique analyses, such as specialized normalization routines or the k-means clustering algorithm. Also note that if the visualization is a custom rather than general tool, custom transformations are often needed. Other transformations, such as taking the logarithm of a value, can either be a data transformation or merely a visual transformation (e.g., just drawing the data on a log scale).

Another transformation that is commonly needed, especially when dealing with databases, is the ability to *join* two (or more) tables together using a common key or set of keys. For example, users might want to join a dimension table to a fact table in order to aggregate by some property of a dimension, such as by the country of the data rather than by state. Joins, however, are not a transformation that is intrinsic to the data, but rather something that is necessary given the organization of the data into multiple tables. Joins are also very expensive to compute. Later in this section, we discuss whether it is necessary to expose this transformation explicitly to the user or whether we can infer which joins are needed using additional meta-data information.

How and where does the visualization perform the analysis?

Given the query specified by the user (directly or indirectly), the next question is how and where the visualization performs the query. Custom transformations likely need custom implementations. However, for more general transformations such as the ones listed above, the question is whether the visualization tool needs to implement its own transformations or whether it can use external implementations (such as in a database). For example, while databases incur a latency for transferring data to the visualization tool, they are optimized for performing such transformations and can provide the capability to scale to very large data sets. This latency can be overcome through various techniques, including having the visualization tool cache query results and implement prefetching or other visualization-specific performance optimizations. Another technique for mitigating latency issues is to do as much analysis as possible within the database to minimize the size of the data set that needs to be transferred.

Although pushing data transformations to the database can provide significant gains, even general visualizations often need their own internal data transformations for two reasons. First, not all data sources provide data transformation capabilities; flat files, for example, cannot transform their contents. Thus, internal data transformations are needed in order for users to see an equivalent abstraction for all data sources (e.g., the data model and transformation capabilities). By providing this abstraction, relational data transformations are compositional (their input and output are functionally equivalent). Thus, transformations can be composed to create more sophisticated visualizations regardless of their implementation.

Second, visualizations need internal transformations for drawing purposes. Polaris, for example, has multiple panes in its table-based visualizations. Rather than performing a query for every pane, we perform the minimal set of queries to minimize latency and then use an internal grouping and sorting network to send data to the appropriate panes for drawing in the needed sorted order.

How do users specify which analyses they want to see?

If a visualization provides transformation capabilities, the next question is: how do users specify which analyses they want to see? The basic tradeoff is between flexibility and ease of use, i.e., a programmatic, explicit interface versus an inferred, implicit interface. A highly flexible choice is to have the user learn a query language such as SQL or MDX. This option requires a great deal of user expertise and limits the user to the associated data sources. Another similar option is to create an abstraction layer and provide transformation primitives that the user can link together, or compose, to get the desired result; this can be done by visual programming (Tioga-2 [Aiken et al. 1996] or VQE [Derthick et al. 1997]), writing scripts (Rivet) or macros (Excel), or some other method.

In contrast, the inferred, implicit approach is to generate the needed transformations (or series of transformations) automatically from some description, as is done in Polaris [Stolte et al. 2002b]. However, in order for this approach to work, the visualization tool must provide abstractions and capabilities to make all data sources look equivalent and appear equally expressive. Further, there needs to be enough meta-data information to be able to generate the correct transformations. We discuss this meta-data more in Section 6 below.

Note that these options illustrate a tradeoff between expressibility and the level of user expertise needed: Of the systems mentioned above, Polaris is perhaps the most intuitive, but is also the most limited. While query languages provide the most general set of analysis operations, both query languages and transformation primitives require more sophistication than Polaris, since the user must determine not only how to express the analysis to perform but also how to specify the data display. We consider this latter question in more detail in Section 8.

What issues are involved when implementing data transformations?

Finally, if the choice is indeed to implement transformations internally within the visualization, some of the issues encountered are:

- Multiple quick passes versus a single pass: Within a transformation, there is the question of whether it is faster to have multiple quick passes over the data or a single, potentially much slower, pass through the data.
- Transformation and data granularity: Another question is whether it is faster to have each transformation perform a single operation or a combination of operations. For example, we have found that *group by and aggregate* are often performed in conjunction. One optimization is to combine these operations into a single transformation that the user can customize by choosing which fields to group by and then how to aggregate each group. In some sense, rather than looking at transformation granularity, we are looking at data granularity since the customization is being done at the tuple level rather than the table level. Some transformations, such as sorting and ranking, are more easily calculated at the table level.
- Flexibility: Once the decision to implement data transformations internally is made, the temptation is to make them as flexible as possible. There are several degrees of flexibility that can be achieved, some of which are both unnecessary and expensive, performance-wise. The first choice is whether to have composable transformations, where any transformation output can be the input to any other transformation, thus allowing a wide range of analyses to be performed. While composability has several benefits, designers often take the next step that, while independent, takes advantage of composability by providing the capability to form a general network of transformations, in which every intermediate result is kept and is visualizable. Rivet, like other systems such as Tioga-2 [Aiken et al. 1996] and VTK [Lucas et al. 1992], provides this capability. While we originally thought this flexibility would be a great advantage, not only is there a large performance cost, but we also never needed this level of generality and flexibility — we never visualized the intermediate results. This performance penalty can be mitigated by using customizable transformations such as the *group by and aggregate* example discussed above.

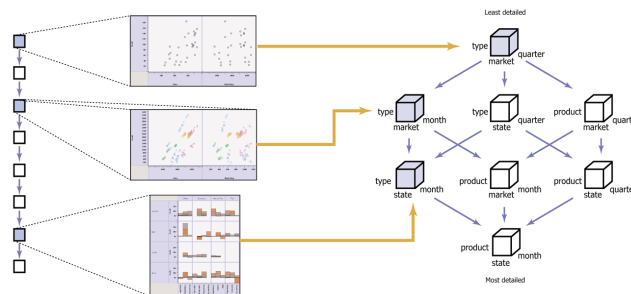

Figure 2: This figure shows a series of successive semantic zooms (on the left) that an analyst might see while exploring a data set (the semantic hierarchy is shown on the right). Each visualization corresponds to a drill down to a more detailed level in the data. This example illustrates one example of how meta-data information might be used in a visualization tool to determine the different aggregations needed when doing semantic zooming.

Related Work: Many existing visualization systems recognize the need for some data transformation capabilities, but they rarely discuss all the questions discussed above. Instead, they focus on some subset.

For example, both VQE [Derthick et al. 1997] and Snap-Together Visualizations [North et al. 2002] focus primarily on how the user can easily specify which transformations to perform, especially on the difficult problem of specifying which join(s) to perform. In terms of actually performing the transformations, they choose to push the actual transformations to the database.

In contrast, both Sage [Roth et al. 1997] and Tioga-2 / DataSplash [Aiken et al. 1996][Woodruff et al. 2001] focus more on how flexible the data transformations need to be, and, specifically, that data transformations need to be composable. This composability is needed for providing the maximal flexibility to the analyst, especially with the integration with visualization, so that the every intermediate result can be seen visually. Tioga-2 uses this functionality quite effectively for debugging purposes.

DEVise [Livny et al. 1997][Livny et al. 1996] focuses more on implementating queries, both in terms of specialized queries and performance optimizations. For example, like with the data access in Section 4, they provide an interface that allows expert users to implement data-source specific queries, such as queries for searching for particular company stock information. As before, this provides great flexibility and extensibility, but does require substantial user expertise to set up. They also discuss several interesting performance optimizations, particularly with regards to whether certain transformations, such as filters, should be performed on tuple (TData) or graphical primitives (GData).

6 Sophisticated Meta-Data

Meta-data information, the data about the data, is just as important as the data itself. This meta-data is needed for two main reasons: to expose semantically meaningful data transformations to the user

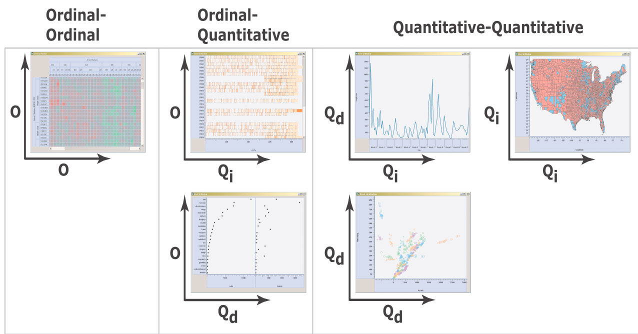

Figure 3: Knowing whether fields are ordinal or quantitative helps in determining which chart types will be most effective visually. For example, a simple table / matrix view is really the only possible view if both axes are ordinal (although the display within each cell can be anything from a simple text field showing the value of a particular metric to a color-encoded glyph). For ordinal-quantitative axes, some possible chart types are a Gantt chart or a dot-plot. In these cases, the ordering of the ordinal field can greatly vary the efficacy of the chart. Finally, for quantitative-quantitative charts, possible chart types include line charts, scatterplots, and map (latitude vs. longitude).

Figure 4: Knowing whether fields are ordinal or quantitative helps in determining which encodings are most effective visually. For each type of encoding, this table gives some examples of how the encoding might be used given the mark type (0-dim points, 1-dim lines, and 2-dim areas), and whether the data being encoded is nominal (unordered) or ordinal/quantitative (ordered). For example, there is no possible shape encoding given an ordered field since what is the canonical ordering of shapes?

and to drive design decisions when generating the visual representation.

Given a visualization tool, in order to expose semantically meaningful data transformations to the user, we need the appropriate meta-data information. For example, to know which aggregations are related for drilling down or rolling up data, we need to know the hierarchy describing the semantic levels of detail (Figure 2 shows a series of successive semantic zooms that an analyst might see while exploring a dataset; each visualization corresponds to a drill down to a more detailed level in the data). Another example is that we need to know the type of a field to determine which aggregation functions are appropriate: it does not make sense to apply an average funtion to nominal field, such as the country.

In addition to exposing meaningful data transformations to the user, additional meta-data information is useful when automatically inferring the needed transformations, especially when joins are needed. Joins may be the main reason that query languages are needed and are the main way to misuse data (e.g., doing inappropriate joins because of a misunderstanding about the keys). However, given the appropriate meta-data information and constraints on the data, joins can automatically be inferred. Specifically, we need information on which fields are in which tables, which fields are primary keys and foreign keys, and which fields are dimensions (independent variables) and measures (dependent variables). In addition to having this meta-data information, we also need to know which fields are unique or equivalent across tables (e.g., that a measure in multiple tables is equivalent), so that we can choose appropriate sources for output fields.

Having this meta-data information allows us to determine which transformations are legal (for example, applying an average to a categorical field makes no sense) as well as which joins are possible, and then to actually construct the right database queries.

In addition to needing meta-data information in order to infer data transformations, meta-data information is also needed to make design decisions when generating visualizations. For example, knowing whether fields are ordinal or quantitative (discrete or continuous) helps in determining which chart types are most effective, or even possible: a bar chart is one option given an ordinal x-axis and a quantitative y-axis; if both axes were quantitative, however, either a line chart or a histogram would be needed. Figure 3 shows some possible chart types given the field types. Knowing whether fields are nominal or ordinal/quantitative (unordered or ordered) also helps in determining which encodings are effective (see Figure 4). For example, using shape for ordered data does not make sense: what is the canonical ordering of shapes?

Meta-data information is needed for a whole host of other reasons, including providing context (e.g., axis labels) and determining the range of values to be displayed. The range is especially important since its choice (e.g., the spatial encoding on an axis) affects the user's perception of the data display. For percentages, not including zero can greatly distort the user's perception of the graph; similarly, reducing the range may emphasize the slope, leading the user to perceive a greater change than might actually exist. Another example is that knowing the absolute domain of a field is essential: knowing which values are missing is just as important as knowing which values have data.

Given all of these different ways of using meta-data information within a visualization tool, the next question is what specific meta-data information is needed? While one customary goal when designing systems is to generalize as much as possible to find the minimal set of information common to most data sets (typically the name and storage type of the field), visualizations can take advantage of additional details to create effective displays. To continue the example above, it is not enough to know that the data field is a quantitative field since not all quantitative fields should include zero on their axes. For example, geographical fields (e.g., latitude and

longitude) would be greatly distorted if zero were always included.

Given this insight, what meta-data information can a visualization use? First, visualizations can take advantage of *type* information for a field, consisting of several characteristics:

- The canonical "storage" type: real, integer, string, date / time, and so on. This information is used to determine type-specific transformations (e.g., Year for a time field), as well as in visualizations since there are clearly known and expected ways to draw, for example, time axes.
- Discrete versus continuous: real data is continuous, integer and nominal data are discrete. This type information can be used to determine chart type; for example, line charts should not be used with discrete data since interpolation does not make sense. This type information can also be used to determine which aggregation functions make sense.
- Ordered versus unordered: real and integer data are ordered, nominal data is unordered. When choosing retinal encodings, shape encodings are effective for unordered data while size encodings would be more effective for ordered data [Cleveland 1985].
- Scales [Stevens 1946]: continuous data may be intervals, ratios, geographic data, etc. Ratios should include zero when displayed spatially, while intervals and geographic fields may not. The aspect ratio for geographic data depends on the map projection.
- Units: most measures have units, such as currency, associated with them. Units are useful for determining which fields can share an axis. A good example is the Gantt chart: knowing that the beginning time and the event duration use the same units allows the visualization to use the duration to encode the length of the bar on an axis displaying time.

Exposing semantically meaningful data transformations and choosing effective visual encodings depends on having all of this type information.

Other meta-data information that is useful to know are which data fields are related. For example, knowing that the (city, state, country) fields are a semantic hierarchy help for exposing drill downs and roll ups. Knowing that the "Profit" field in one table is equivalent to the "Profit" field in another table is useful when inferring joins. When doing joins, we need to know more than just which data fields are related, we need all of the table meta-data as well: names of tables, which fields are in which tables, which field(s) are the keys, as well as which fields are equivalent.

Another useful type of meta-data is domain information. Having both the absolute domain (range of all possible values) as well as the actual domain (range of values actually present in a table) is useful for determining the range on an axis, for example.

Given the usefulness of meta-data, the final question is how to import and model the information within a visualization. Note that a database does not typically contain all of this meta-data information, so querying a database will not provide a complete solution. While there are many possible solutions to this problem, we briefly outline the approach we chose for Rivet.

To import the meta-data, we augment the same XML file used for specifying the provenance information for a data source (Section 4) with the additional meta-data information, such as the type information, absolute domains, hierarchies, derived field information, ad hoc groupings, and so on.

When this meta-data is brought into the visualization environment, it is treated as a first-class citizen rather than as an add-on to the data object. For example, not only are fields separate objects with type information, but domains and hierarchies are also separate objects, since some domains are static while other domains change depending on the data or user interaction (e.g., a domain containing a user-specified filter); domains may also be shared by different tables. Similarly, some hierarchies are known in advance,

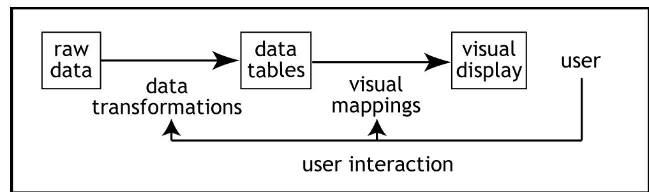

Figure 5: A pipeline showing the flow through a visualization.

while other hierarchies, such as ad hoc groupings, may change as the user interacts with the visualization.

One additional nuance is that meta-data is not always imported: if data transformations are supported within the context of a visualization tool, then there are now derived fields, e.g., Sum(Profit) is derived from the Profit field. The meta-data also needs to be derived. However, there are some subtle complexities: depending on the transformation, the type of the field may change. For example, COUNT(DISTINCT(Country)) is an ordered, integral field while the source field, Country, is an unordered, nominal field.

Related Work: Many existing systems, such as Snap-Together Visualizations [North et al. 2002], DEVise [Livny et al. 1997][Livny et al. 1996], and Tioga-2 / DataSplash [Aiken et al. 1996][Woodruff et al. 2001], use the meta-data information provided in the database schema, which typically consists of the field name, storage type, actual domains, and hierarchical relationships. Exbase [Lee and Grinstein 1995] discusses the need for additional meta-data information, but focuses on information that can be automatically inferred from the database or the query.

Both APT [Mackinlay 1986] and Sage [Roth and Mattis 1990] discuss how meta-data information can be used to generate appropriate visual encodings automatically, with the extended type information being the primary overlap. Sage discusses how to use additional meta-data describing the relationships between database tables in order to support joins.

7 Modularizing Data, Visualization, and Interaction Objects

The previous sections have focused on using the relational data model, generalized API for data access, and data transformations to abstract the raw data. We have also discussed how we can use meta-data information to choose which visual mappings might be most effective. In this section, we discuss the other portions of the visualization pipeline and some issues with modular architectures.

The high-level architecture of a visualization system is often drawn as a pipeline (Figure 5). One of the fundamental design decisions in Rivet was to modularize each box in the pipeline and then define the objects and API's for the arrows connecting the boxes. By performing this encapsulation, we gain flexibility, since we can combinatorially compose objects to create a wide variety of visualizations and analyses. We also gain extensibility, since new functionality can be incorporated by implementing additional box and arrow instances within the existing sytem.

This modularization also impacts how quickly one can change the design of a visualization tool: if all components are tightly tied together, then changing one requires changing everything. By choosing the right granularity, one can iterate through design refinements more quickly. However, there are many issues that arise in the modularization that we discuss in this section.

7.1 Visual Encodings

In Rivet, we initially created an architecture for the visual mappings and the visual display with four basic types of objects:

- Mappings that map values from one type to another, e.g., from quantitative values to size or from a nominal value to color.

- Encodings that use mappings to explicitly map a particular data field to a particular visual variable (e.g., Profit to the x-axis or Product Type to color).
- Metaphors that lay out tuples spatially (using spatial encodings).
- Primitives that use retinal encodings (e.g., shape, color) to render a tuple in the space allocated by the metaphor.

This architecture has several advantages, as we have discussed above: it maximizes code re-use, is easily extensible since only the new mapping or layout algorithm needs to be added, and is highly flexible since multiple visual representations can be applied to the same data set, and conversely, the same visual representation can be used by many different data sets.

However, there are also several disadvantages. One issue that we have mentioned before is that while increased generalization may lead to increased flexibility, we cannot hide too many of the details since visualization is about displaying data in context, and that context is needed even in the code. One example of where our generalization led to difficulties is in the separation of the visual into spatial layout and retinal encodings. Because we need to handle both line and polygon primitives, those primitives need some spatial knowledge in addition to needing the data to be sorted and drawn in a certain order (connected lines). Additionally, retinal encodings have slightly different meanings depending on the primitive type [Stolte 2003].

The other primary disadvantage is that this architecture is sometimes over-generalized, thus requiring additional work to generate the proper mappings and encodings. For example, to find the proper range in a stacked bar chart, we need to determine the maximum value for any particular stack. However, in order to determine this value, needed by the mapping, we need to know not only the numeric field being stacked (known by the encoding), but also which ordinal/nominal field is used for creating the stacks (known by the metaphor).

One area for future work is to examine whether our initial choices in determining the set of object primitives and interfaces led to these difficulties, as well as whether different modularizations solve these problems (and with what tradeoffs): a different choice may lead to a different set of issues.

Related Work: Many systems, including Sage [Roth et al. 1997], DEVise [Livny et al. 1997][Livny et al. 1996], and Tioga-2 / DataSplash [Aiken et al. 1996][Woodruff et al. 2001], use the idea of visual encodings to create an abstraction between the data and the corresponding visual representations.

7.2 Interaction

While we would like to encapsulate interaction much as we encapsulate the data to visual mappings in order to realize the same benefits, it is more difficult since there are at least three different axes along which to encapsulate interactions:

1. Outputs: The first axis to look at is the end effects of the interaction. For example, an interaction can affect the raw data, the data transformations, the visual mappings, or the view itself (e.g., rotation).
2. Inputs: The second axis to look at is the inputs. For example, some interactions require the mapping between data and visual so that when given a selected area in visual space, which data tuples correspond to the selected area can be determined, such as is needed for brushing or tooltips.
3. Triggers: The final axis to look at is how the interactions are triggered. For example, panning can be triggered by a trackball interface, a scrollbar, arrow keys, etc.

Looking at the visualization pipeline, what we chose to do is provide interfaces on the boxes (the data and visual objects), which correspond to the outputs axis. These interfaces correspond to high-level actions, such as pan and zoom. Note that with Rivet scripting (Section 2), we can build even higher-level actions in the script. Alternatively, the script-writer can choose whether an action (however triggered) corresponds to an optical zoom on the visual object or a semantic zoom on the data object.

To abstract the triggers axis, Rivet takes advantage of its event callback methodology to build interactions (Section 2): the objects themselves do not care who calls their interfaces or how. Like Java, the interfaces can be triggered by any mouse interaction, any key press, etc., all depending on how the script is built.

The inputs axis only needs to be abstracted when the user actions need additional translation to determine other inputs to the object interfaces. For example, in the simple panning case, the object just needs to know whether to move up or down (or left or right). A more complex interface would need to know how far to move — an additional input is needed, based on translating the actual triggers, such as amount of mouse movement or speed of key presses. To abstract the inputs axis, Rivet uses the notion of a transparent overlay to translate mouse events into higher-level interaction events. Different overlays implement different translations; one overlay translates mouse events according to a trackball interface for panning and rotation, while another overlay translates mouse events into input for selection (e.g., rubberbanding). Essentially, these overlays use the triggers to aid in translation when the interactions need to know about spatial encodings.

Because selection is such a prevalent and complex operation, we use two additional abstractions. The first abstraction takes advantage of our modular (and inheritance-based) model for visual objects. Specifically, we have a base metaphor class that implements techniques common to all metaphors, including whether tuples are being rendered or just laid out for selection (picking) purposes. Thus, individual metaphor classes only need to implement the actual layout. The second abstraction is to modularize the notion of selection itself. Selection may mean choosing a single range of values, choosing multiple ranges (e.g., selection on multiple axes), choosing distinct values, choosing distinct tuples, etc. Thus, we also have a general selection API, with individual selection classes implementing the specific type of selection desired.

Related Work: Snap-Together Visualizations [North et al. 2002] focus on how to layer user interaction on top of visualizations for coordination. While they focus on interaction rather than visualization (leaving that to the individual visualization tools snapped together using their interface), they have a very different model in which every visualization that is snapped together is done so via the equivalent of a database join. This model leads to easily achieving some powerful interaction capabilities such as brushing and linking.

Chi et al. [Chi and Riedl 1998] and Chuah et al. [Chuah and Roth 1996] present frameworks for organizing the different types of interactions within a visualization. They both organize the interactions by their end effects (e.g., whether the value (data), view (graphics), or some combination is affected). Chi discusses implementation only briefly, pointing out that where the operator would be optimally placed (within the visualization, the database, or in a specialized tool) depends on where in the visualization pipeline the interaction falls.

7.3 Modularity Issues

While we have mentioned the many advantages of modularity, there are several issues that arise with a modular architecture as well.

One issue with modularity is that if the objects and API's are not designed with all cases in mind from the beginning, making changes later on can be very difficult. We pointed out one example above with the point primitives, but meta-data information is another even more important example. We originally designed Rivet using the pipeline model and focused on modularizing data and its display. As we realized the importance of meta-data and its display and interaction, we created separate meta-data objects for use in

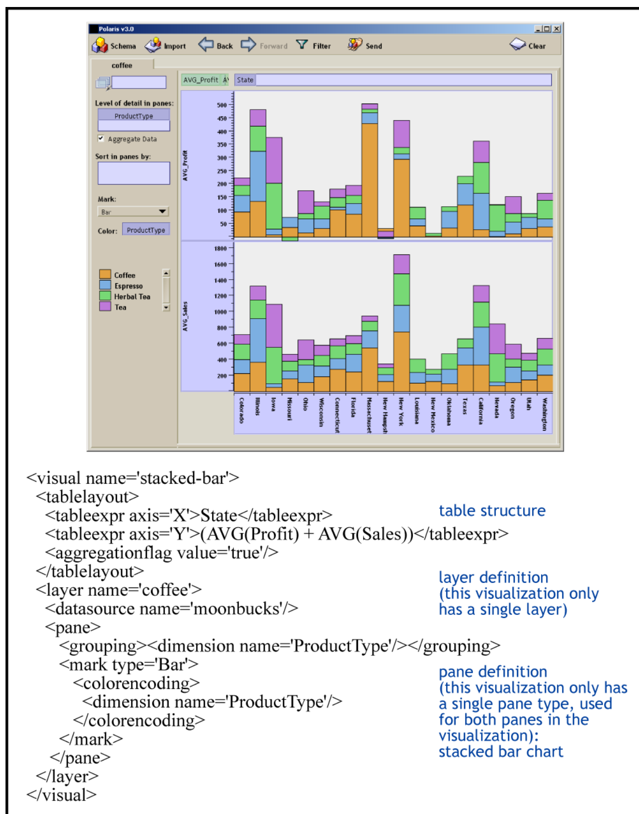

Figure 6: An XML specification and the generated visualization.

creating effective visualizations of data, as discussed in Section 6. However, rather than modularizing meta-data displays (e.g., axes, label, legends) and interaction separately, a more accurate pipeline would recognize that meta-data should be treated like data and be designed accordingly.

Another issue is the listener model commonly used in modular code. In this model, objects "listen" to one another, and when an object changes, it notifies all of its listeners so that they can update themselves. For example, if the user changes the color mapping, the mapping informs the encoding, which then notifies all of the objects that use that encoding for drawing, so that they then will redraw. While the listener model simplifies the implementation for the designer since the coder does not need to determine exactly what needs to be updated on every change, getting the implementation of the listener model correct is tricky, since the propagation of change events may easily lead to over-updating of objects, resulting in a performance loss. The key is that each event needs to say not just that something changed, but to say what changed so that only the minimal set of updates and propagations happen.

8 Specification and Scripting

The final design decision we discuss is how to create visualizations given the infrastructure discussed in the previous sections. The method chosen here impacts how quickly a visualization tool's design can iterate. There is a spectrum of choices here as well, again with a tradeoff between simplicity and expressibility.

One extreme is to use scripting (a procedural approach) to create the visualization. Scripting requires the most expertise and takes the most time, but has the most expressibility and flexibility: users can explicitly create the views of the data they want, the layout of those views, the exact interactions and results of those interactions, and so on.

The other is to use a tool like the Polaris user interface[2], where the user interactively drags-and-drops field names to create a visualization. While this interface is intuitive to use, the user is constrained to table-based visualizations and to the interactions and data analysis capabilities exposed by the user interface.

As we created more interactive exploratory visualization tools, such as the Polaris user interface and tools for exploring computer systems data, we found that table-based visualizations were predominant and highly useful. We created the Polaris formalism [Stolte et al. 2002b] describing this class of visualizations, which then formed the basis for our specification language. Writing a specification (a declarative approach) is like filling out a template containing the specific fields the user wants to visualize, and the specification can be used to generate both the data queries needed to retrieve the data being visualized and the visual encodings needed to create the display itself.

This specification language is a middle ground, in that the user can create static table-based visualizations simply by filling out a template (implemented using XML). In fact, the user can create more sophisticated visualizations using the specification language than is possible in the Polaris user interface[3], since creating intuitive affordances corresponding to some more advanced features, such as the nest operator, is quite difficult [Stolte 2003]. When comparing filling out specifications to writing scripts, specifications are easier but more restrictive. In this case, the user can only create table-based visualizations without interaction, while scripts require more expertise but are much more powerful.

Rather than choosing either specification or scripting, yet another option is to combine the two to gain expressibility without requiring too much more expertise. The specifications define both the visual and the data abstraction (i.e., the visual representation and the data transformations) [Stolte et al. 2002a]; a sample specification is shown in Figure 6. By pushing the repetitive definitions to the specifications, the scripts are simpler, since they only need to define the interaction and the transitions between specifications. The scripts are simplified even further since we have XML files for specifying the data (including the data import). We retain all the power of Rivet, since scripts can be used to create any visualizations beyond the scope of the specification language.

While specifications can be used to create static visualizations, one area of future work is to explore how to specify interactive visualizations. One approach is to research transitions and interactions further to see if we can find a parameterization or formal structure to use in a specification.

Related Work: Several systems address this debate between specification and scripting.

Snap-together Visualizations [North et al. 2002] falls on the specification side while addressing a very different problem of trying to define the interactions rather than the visualizations. However, they do make the point that scripting is too difficult for users.

Sage and Visage [Roth et al. 1997] provide several different interfaces. While they also use scripting for interaction, designing the visualization itself is done automatically using Sage, by visual specification using SageBrush, or by searching for similar examples using SageBook. Thus, while they recognize the usefulness of both scripting and specification, the two are not used in concert.

Tioga-2 and DataSplash [Aiken et al. 1996][Woodruff et al. 2001] use another approach, similar to visual programming, where the user can link together icons representing actions and both data and graphical transformations.

---

[2] The Polaris user interface is built using Rivet scripting and illustrates a third option, which is to bootstrap one extreme to reach the other.

[3] Creating visualizations using the Polaris user interface is equivalent to creating a specification visually. However, there are valid specifications that cannot be created using the Polaris user interface.

Wilkinson [Wilkinson 1999] also has a concise language for specifying visualizations that is quite powerful and descriptive. However, this language is used only for creating static displays.

9 Future Work

While we have pointed out various directions for future work within the various sections, graphics and database performance are a specific area of concern for visualization that bears further exploration.

9.1 Graphics Performance

Graphics hardware is getting faster at the pace suggested by Moore's law, yet most visualizations do not take advantage of the graphics hardware, even when displaying large data sets. One issue is that graphics hardware designers are optimizing for 3D graphics, partly because of the high performance demands from the lucrative gaming industry, but also because visualization designers have not pushed the limits of graphics hardware and come up with a list of needs. For example, while we initially chose OpenGL for its portability, it is designed for monolithic 3D graphics applications such as games. As a result, Rivet, with its modular architecture and 2D graphics, needs to do additional work for performance, such as maintaining its own state and using different contexts for different windows (different frames within the same window share the same context) to minimize context-switching, an expensive operation in OpenGL. Other interfaces such as DirectDraw may be a better fit for this type of visualization system.

Thus, one area for future work is to examine the limits of current graphics APIs and hardware for visualization. A few visualization applications do push the limits of graphics hardware, such as Fekete et al.'s work on large data sets [Fekete and Plaisant 2002] and Solomon's work on CAD layout [Solomon 2002].

9.2 Database Performance

Visualization can be seen as the intersection of user interface design, graphics, and databases, and therefore database performance is another area for optimizations.

While database researchers have done a lot of work on optimizing databases, these optimizations are for typical database usage: multiple users, each submitting unrelated queries simultaneously, with subsequent queries likely unrelated. In other words, there is little locality to exploit. Database researchers have also recognized the difference between OLAP and OLTP databases and created different database organizations and optimizations for each.

However, visualization systems require both more information and different optimizations than existing databases provide. One issue is to determine whether these optimizations belong in the database or in the visualization tool. For example, if the visualization wants to display sampled data or streamed data from a large database, that sampling or streaming functionality belongs in the database. On the other hand, prefetching belongs in the visualization, since the visualization tool has two advantages a general database does not. First, a single analysis session within a visualization is likely to have more locality. Second, visualizations can exploit this locality by providing interaction paths that make traversing the likely course of analysis easier, e.g., drill downs and roll ups, changing the filter, etc. While predicting the likely course of analysis is more difficult in tool like the Polaris user interface, it is significantly easier to predict the user's path in the multiscale visualizations [Stolte et al. 2002a] and prefetch accordingly. Other display-specific choices that affect the data abstraction, such as importance ordering, also belong in the visualization tool.

Finally, latency is the big issue with regards to integrating visualization tools with databases. Prefetching can help alleviate this problem, but the latency of bringing data into the visualization is still a fundamental bottleneck. Some other ways to address the latency issue include sampling the data set and estimating and bounding the uncertainty of the data [Hellerstein et al. 1999].

10 Conclusion

In this paper, we discuss the design decisions we needed to revisit as we refined Rivet to create more sophisticated interactive visualizations for exploration and analysis of large data sets. Specifically, for each design choice, we discuss its importance, implications, some possible approaches and the associated advantages and disadvantages, as well as some implementation challenges. The issues we discuss include:

- Data Model: the ordered relational model provides flexibility and a layer of abstraction that makes the generalized data interface, data transformations, and separation of data, visual, and interaction objects possible.
- Simplification of Data Access: users will not use visualization tools regularly unless it is easy to import their data.
- Data Transformations: a tight loop between analysis and visualization allows the user to perform analyses more quickly and easily.
- Sophisticated Meta-data: with additional meta-data, visualizations can choose encodings more effectively.
- Modularization of Data, Visual, and Interaction Objects: while modularization gains flexibility and extensibility, there can be dangers to over-generalization.
- Specification *and* Scripting: choosing how to create visualizations involves a tradeoff between expertise and expressibility.

These issues arise whether the designer is building a custom visualization or a more general tool; the main differences are the choices made. This paper has presented a discussion of the different trade-offs and options so that designers need not go through the same discovery process that we went through and can thus architect visualizations quickly and effectively.

11 Acknowledgments

This work was supported by the US Department of Energy through the ASCI Level 1 Alliance with Stanford University.

References

AIKEN, A., CHEN, J., LIN, M., SPALDING, M., STONEBRAKER, M., AND WOODRUFF, A. 1996. The Tioga-2 database visualization environment. *Lecture Notes in Computer Science: Database Issues for Data Visualization 1183*.

BERTIN, J. 1983. *Semiology of Graphics: Diagrams, Networks, Maps*. Univ. of Wisconsin Press.

BOSCH, R., STOLTE, C., TANG, D., GERTH, J., ROSENBLUM, M., AND HANRAHAN, P. 2000. Rivet: A flexible environment for computer systems visualization. *Computer Graphics 34*, 1.

BOSCH, R. 2001. *Using Visualization to Understand the Behavior of Computer Systems*. PhD thesis, Stanford University.

BOURRET, R. Xml and databases (http://www.rpbourret.com/xml/xmlanddatabases.htm).

CARD, S., MACKINLAY, J., AND SHNEIDERMAN, B. 1999. *Readings in Information Visualization*. Morgan Kaufmann.

CHI, E., AND RIEDL, J. 1998. An operator interaction framework for visualization systems. In *Proc. of IEEE Symposium on Information Visualization*.

CHUAH, M., AND ROTH, S. 1996. On the semantics of interactive visualizations. In *Proc. of IEEE Symposium on Information Visualization*.

CLEVELAND, W. 1985. *Elements of Graphing Data*. Wadsworth Advanced Books and Software.

DERTHICK, M., KOLOJEJCHICK, J., AND ROTH, S. 1997. An interactive visual query environment for exploring data. In *Proc. of ACM SIGKDD*, 2–9.

FEKETE, J., AND PLAISANT, C. 2002. Interactive information visualization of a million items. In *Proc. of IEEE Symposium on Information Visualization*.

HELLERSTEIN, J., AVNUR, R., CHOU, A., OLSTON, C., RAMAN, V., ROTH, T., HIDBER, C., AND HAAS, P. 1999. Interactive data analysis: The control project. *IEEE Computer*.

HUFF, D. 1954. *How to Lie with Statistics*. WW Norton.

KEIM, D., AND KRIEGEL, H. 1994. VisDB: Database exploration using multidimensional visualization. *IEEE Computer Graphics and Applications 14*, 5, 40–49.

LEE, J., AND GRINSTEIN, G. 1995. An architecture for retaining and analyzing visual explorations of databases. In *Proc. of IEEE Visualization*, 101–108.

LIVNY, M., RAMAKRISHNAN, R., AND MYLLYMAKI, J. 1996. Visual exploration of large data sets. In *Proc. of SPIE*, vol. 2657.

LIVNY, M., RAMAKRISHNAN, R., BEYER, K., CHEN, G., DONJERKOVIC, D., LAWANDE, S., MYLLYMAKI, J., AND WENGER, K. 1997. DEVise: Integrated querying and visual exploration of large datasets. In *Proc. of ACM SIGMOD*.

LUCAS, B., ABRAM, G., COLLINS, N., EPSTEIN, D., GREESH, D., AND MCAULIFFE, K. 1992. An architecture for a scientific visualization system. In *Proc. of IEEE Symposium on Information Visualization*.

MACKINLAY, J. 1986. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 110–141.

NORTH, C., CONKLIN, N., INDUKURI, K., AND SAINI, V. 2002. Visualization schemas and a web-based architecture for custom multiple-view visualization of multiple-table databases. *Information Visualization*.

Web services and service-oriented architectures (http://www.service-architecture.com).

ROTH, S., AND MATTIS, J. 1990. Data characterization for intelligent graphics presentation. In *Proc. of SIGCHI*.

ROTH, S., CHUAH, M., KERPEDIJIEV, S., KOLOJEJCHICK, J., AND LUCAS, P. 1997. Towards an information visualization workspace: Combining multiple means of expression. *Human-Computer Interaction Journal 12*, 1 and 2, 131–185.

SOLOMON, J. 2002. *The ChipMap: Visualizing Large VLSI Physical Design Datasets*. PhD thesis, Stanford University.

STEVENS, S. S. 1946. On the theory of scales of measurement. *Science 103*, 2684, 677–680.

STOLTE, C., TANG, D., AND HANRAHAN, P. 2002. Multiscale visualization using data cubes. In *Proc. of IEEE Symposium on Information Visualization*.

STOLTE, C., TANG, D., AND HANRAHAN, P. 2002. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics 8*, 1, 52–65.

STOLTE, C., TANG, D., AND HANRAHAN, P. 2002. Query, analysis, and visualization of hierarchically structured data using Polaris. In *Proc. of ACM SIGKDD*.

STOLTE, C. 2003. *Query, Analysis, and Visualization of Multidimensional Databases*. PhD thesis, Stanford University.

THOMSEN, E. 1997. *OLAP Solutions: Building Multidimensional Information Systems*. Wiley Computer Publishing.

WILKINSON, L. 1999. *The Grammar of Graphics*. Springer.

WOODRUFF, A., OLSTON, C., AIKEN, A., CHU, M., ERCEGOVAC, V., LIN, M., SPALDING, M., AND STONEBRAKER, M. 2001. Datasplash: A direct manipulation environment for programming semantic zoom visualizations of tabular data. *Journal of Visual Languages and Computing 12*, 5, 551–571.