Tuple-Space Mapper: Design, Challenges and Goals

Joseph A. Cottam*
Computer Science Department, Indiana University

Andrew Lumsdaine†
Computer Science Department, Indiana University

ABSTRACT

Library-based and pre-compiled visualization tools incur many penalties that hinder the adoption of visualization as a technique for many applications. Libraries necessitate familiarity with the data structures and control flows that are incumbent in traditional programming, but not central to visualization. Task-specific visualization applications alleviate these needs, but induce users to move data between applications as their needs change. We propose a generative programming approach to visualization tool construction based on domain specific languages. We believe this approach will simplify the process of visualization creation, simplify integration with many tools (reducing error-prone conversion steps) and ease the modification of existing tools (reducing the pressure to introduce new tools as needs change). In this paper, we present an architecture for a such a tool, outline the technical requirements and challenges and argue for its relevance to improving the adoption of visualization.

Keywords: Visualization, Domain Specific Language, Declarative Language, Generative Programming.

Index Terms: I.3.4 [Computer Graphics:]: Graphics Utilities—Software support I.2.2 [Computing Methodologies:]: Automatic Programming—Program synthesis D.3.2 [Programming Languages:]: Language Classifications—Specialized application languages H.1.2 [Information Systems:]: User/Machine Systems—Human information processing

1 INTRODUCTION

Information Visualization has encountered many roadblocks to adoption beyond academic research or very isolated applications. Common visualization practice follows one of two routes: 1) Employ or train an expert to handle the visualization task; 2) Purchase off-the-shelf software. These two methods incur penalties that limit their utility beyond research applications or very specific problems as they both are expensive to employ and costly to maintain. In short, there is no way for visualization problems to be 'played with' before committing substantial resources or limiting the solution scope in early stages of development (the importance of play in visualization is discussed in [13]).

We believe that a declarative programming language that holds close to the core concepts of information visualization will reduce the cost of adoption and, combined with a generative programming style, ease maintenance difficulties. The need for flexible, extensible visualization has been recognized before in [22], but our system goes a step further by allowing the visualizations themselves to be specified, not just the coordination between them. The need to improve the programming of visualizations was recognized by [8], but the Processing system retains a traditional language structure while simplifying other parts of the process. To expand beyond this prior work, we describe the embodiment of our concepts in an abstract Tuple-Space Mapper (TSM). In this paper, we explore the goals and motivation of the TSM, then explore a possible architecture and related works. Finally, we give a brief example in a prototype language and its interpretation.

*e-mail: jcottam@cs.indiana.edu
†e-mail: lums@cs.indiana.edu

2 GOALS

Much of information visualization is (conceptually) mapping abstract attribute sets (tuples) to graphic representations (glyphs) where particular glyph features are related to the values present in the tuple. In the TSM system, we seek to use a declarative language built on simplifying this mapping process employing on a stream processing metaphor (ala YACC/Lex [18]) . Specifications written in the TSM language will be used to generate components hosted in larger applications. We believe such a system will enable:

1. Rapid prototyping of visualizations using real data
2. Synthesis of multiple data sources in a conceptually elegant and practical fashion
3. 'Data conversations' by easing customization of visualization applications
4. Decoupling the visualization specification from its implementation allowing greater flexibility through a visualization-focused upgrade path

3 MOTIVATION

3.1 Software Visualization

Standard visualization do not serve the expert programmers who most often request visualization support. This is a major challenge to the adoption of visualization in software development. Visualization tools to support expert programmers encounter several obstacles that disqualify many standard tools. First, expert programmers tend to work 'between' mental models, borrowing conceptual tools from several at once. It is *ad-hoc* the expert programmers that develop new models for others to use. As such, the assumptions encoded in standard representations are more limiting to them than to other groups. Expert programmers are *ad-hoc* more likely to have one-off needs for visualizations, where a temporary task would be benefited by a visual representation [25]. Existing applications for this ever-shifting target typically only provide the ability to switch between predefined visualizations (a step in the right direction, but just one step).

Another major obstacle to expert programmers adoption of visualization is the quantity of data encountered. The ability to automatically instrument and monitor large software systems yields a nearly limitless source of data. Inefficiencies that result as artifacts of the process of applying a general tool to a specific problem effectively disqualify many general purpose visualization tools from the field of software visualization [17].

A third problem encountered by expert programmers employing general-purpose tools is the inability to easily integrate visualization tools with other special purpose tools that have been developed. This problem appears in two forms. First, visualization tools often accept only a short list of file formats. These formats are conducive to the production of images, but are often specific to the application being used. On the output side, most visualization programs

provide only limited support to non-visual outputs. If a visualization is used as the front-end to an interactive analysis, the results of that analysis are often difficult to separate from the resulting image. This insularity and vertically integrated design style of visualization tools is a barrier to exploration and adoption of visualization in programming, a field that prizes customizability and integration [25].

A final challenge for effective software visualization tools is the exploratory nature of many of the software analysis tasks. The levels of abstraction employed in programming make drawing relationships between observed program behavior and machine state very difficult. Problem solving involves many hypothesis testing iterations (checking if abstraction changes influence machine state in expected ways), each iteration may require distinct (but related) visualizations. Furthermore, the questions are often less about the exact state of the machine, and more about the transitions that state is going through. These two facts make the techniques of Exploratory Data Analysis (especially available in real-time) of growing in importance [16].

Each of these problems (i.e. mental model incongruence, data overload, tool-chain integration and information contextualization) is encountered in other areas of visualization. However, their confluence in software visualization presents a unique opportunity to apply and evaluate the visualization solution space. Existing tools address some, but not all, of these issues. Developing effective visualization tools for programmers *ad-hoc* supports the second motivation, increasing visualization adoption, by putting visualization tools in the hands of programmers thereby showing them what is possible.

3.2 Visualization Adoption

A common complaint in the visualization community is the lack of adoption of their techniques by other fields. Many years of research have gone into exploring techniques to facilitate the communication of information via visual means, but only a few disciplines have adopted these techniques into their standard toolbox (and the most notable, scientific visualization, is another research community in its own right). A commonly suggested solution is to have visualization researchers work on highly contextualized projects (solving the issues of group X by visualization) to increase the number of fields manually [28]. This is a 'top-down' solution since visualization adoption comes from and is driven by the few visualization practitioners into the population at large. Though effective in producing high-quality visualizations, this limits the available groups to those with funding or current experience.

The database research community paralleled this frustrating situation of low adoption and high barrier to entry. Early research in databases lead to a diversity of competing concepts in how to facilitate data management and retrieval. Each database system had its own particular interface. When a database was desired for a project, the first question always had to be "Which one?" Futher, changing that decision was very difficult. The next step was to hire/train a new professional practitioner in the particular system. Much of these problems eased with the introduction of the relational database model and its accompanying Structured Query Language (SQL). SQL provides an abstraction layer between the database and the people/programs using it. It is a language focused on specifying the data to be retrieved, rather than the process of its retrieval. This allows projects to maintain a loose commitment to the decision of which database to use and the exact data organization. Using standard SQL, most databases can be configured and interacted with. Granted, not all databases made by SQL novices are efficient, but they get a project going. When it becomes apparent that a project needs more database muscle, a professional database technician can be called in and more complicated questions (like exactly which database project would be 'best' and how to optimize the program for that project by using its special features) can be addressed [10].

The visualization community would benefit from a similar underlying metaphor and structured, focused language. Many projects probably do not use visualization because existing visualization frameworks *require* an expert to use. Worse yet, they require a visualization practitioner to even decide which visualization framework to use. These frameworks differ to such a degree that moving between them is akin to a complete system redesign. Having a well-structured abstraction between to the visualization framework and the application (akin to the relationship of SQL with databases) would allow projects to begin to use visualization at their own discretion and then call on the (expensive) professional practitioner when the benefits clearly out-way the incumbent costs.

Facilitating this 'bottom-up' style of adoption of information visualization is a major motivation for the Tuple Space Mapper architecture and its accompanying language.

4 Tuple-Space Mapper

The Tuple-Space Mapper aims to reduce the problems associated with adoption of visualization by moving issues central to visualization into the foreground while retaining access to the power of existing algorithms. We employ a Domain-Specific Language (DSL) with facilities to access functions written in a general purpose language. This gives strength in three areas. First, it explicitly exposes the visualization problem as a mapping from source values to visual values. Second, the DSL compiler relieves the programmer of explicit control flow and data representation concerns. This relief allows for rapid prototyping of visualization applications using real data and eases modification of existing tools when updates are required. Finally, it reduces the complexity of the visualization tool chain by presenting a simpler mental model during compile time and more easily extended applications at execution time. The remainder of this section describes the concepts of the TSM Language and the related system architecture.

4.1 Core Concepts of the TSM System

There are five concepts in the TSM that provide a foundation for the included features.

1. Most visualization is mapping entity features to visual attributes. The TSM makes that explicit in the programming process, reducing the the distance between concept and implementation. To do this effectively language concepts, metadata facilities and integration facilities will all be related back to this central idea.

2. Domain specific languages (DSL) permit strong conceptual links to their target problem domain. In contrast, general-purpose languages force all problem domains to be reduced to issues of control flow and data structures. Employing a DSL with facilities to interact with general purpose languages blends to two by allowing the DSL to capture the core concepts without sacrificing the flexibility of the general purpose language.

3. Extensibility of existing applications permits incremental changes, allowing gradual adaptation to an evolving environment. This helps check the rapid expansion of the analysis tool chain by allowing existing applications to address new problems, rather than requiring the introduction of new applications (and the incumbent coordination overhead, see Figures 2 and 3).

4. Integration is the ability of an application to work with other applications. It is related to extension, but integration focuses on working with existing tools where extension is focused on

modifying those tools to fit a new need. The ability to 'play well with others' is important as it permits the use of existing tools or new tools that fit differing mental models. By including integration as a core concept, we admit that our tool will not solve all problems, but we hope to minimize the required conceptual and technical overhead of working with other tools when solving such out-of-scope problems (see Figure 4). The necessities of integration inform our selection of I/O abstractions and (to a lesser degree) internal data models.

5. Rapid prototyping permits changes to be evaluated quickly, reducing the overall cost of exploring a solution space. This is especially key in visualization as there is no underlying formal theory predicting successes [30]. This concept informs the support tools that will be created and the composition/extenstion mechanisms that will be provided.

4.2 Challenges

There are several major challenges to the success of the TSM tool at both the theoretical/technical and the use levels. These challenges include:

1. Building a conceptual abstraction that can be applied in a diversity of graphical and visualization toolkits. Each toolkit has its own conceptual abstractions, so broader abstractions will have to be used in the TSM.

2. Deriving the proper data and control structures from sparse declarations. This has implications for program correctness, performance and the interaction of generated components with their host application.

3. No underlying theoretical framework to guide construction of the language (like relational algebra for SQL queries). There are notions of the equivalence of certain conceptualizations, but no proof that any of them are complete [4].

4. The best practices of information visualization are scattered across many domain-specific papers. Some concentrations exist (e.g. info vis conferences) but to fully understand the implementation of these domain-embed visualizations requires knowledge of the embedding domain [30].

5. Effectiveness measures involve subjective components (e.g. ease and attractiveness) combined with traditional objective measures (e.g. speed and correctness) [3].

6. Maintaining the TSM as a lightweight tool. If the TSM system becomes another complex tool in the main data analysis tool chain, the goal of reducing tool-chain complexity will have been missed. As such, the TSM needs to be able to generate flexible components so they don't need to be updated with each new analysis task.

4.3 Central Features of the TSM Language

To support the concepts given above, the DSL of the TSM has the following features.

Glyph: A visual representation of data. This is a circle or a dot or text, etc. The goal is to turn abstract data spaces into glyphs.

Layer: A layer serves two purposes. First, it is a logical grouping of elements. This group acts as both a namespace and a way to apply defaults (such as color or visibility). The unique identifier of a glyph is its layer and its ID. Second, layers serve as a visual organizing principle. The order of layers determines which aspects of the visualization occlude others (this is the classic 2.5-dimensional metaphor).

Tuple: A compound data element where the order of the elements matters. Classic examples are coordinates ((X,Y,Z), notice X is always first, Y second and Z third) and database recordset rows. All of the parts of a tuple relate to the same base entity (often named as a part of the tuple).

Stream: All information comes into the framework as part of a tuple stream. A stream is a forward-only reading information source. This is like a forward-only cursor on a database recordset. Items seen before cannot be recalled unless they have been explicitly stored.

Legend: Contextual mapping function. A legend takes data values and produces representative visual attributes. Classic examples include color or shape legends, but the same conceptual abstraction applies to the X- and Y-axis. To be 'complete' a legend must be able to produce a visual representation of itself (e.g. an axial legend should be able to produce an axis with labels, a color legend should produce a mapping chart). The context of a legend may be just the data of the visualization, or it may involve outside influences as well.

Types: The data types significant to visualization are not the same ones significant to a programming language. Visualization data types are more akin to statistical ones: Nominal, Ordinal, Interval and Ratio [21]. The stored data types will need a classical programmatic representation as well (e.g. int or string), but from a visualization standpoint, that is often not as important as the statistical type.

Program Integration: The TSM is intended to be used in the context of a larger program. As such, it needs to export an application programming interface (API). To make this useful, it must be a two-way API (so the TSM can talk to the program as well). This may be supported with features similar to those found in compiler compilers (like YACC [18] or ANTLR [24]). Arbitrary method-call and direct host-language imbedding functionality allows the visualization to talk to the host program. Having standard API's (like stream communication, zoom and pan and visualization event listening) is a start for having the program talk to the visualization. Further work could be done with 'externalized' variables that can be modified by either the visualization or the host program (similar to volatile variables [15]).

4.4 System Architecture

The system architecture is illustrated in Figure 1 and outlined here.

Visualization Specification: A 'front-end' language designed to be easily used by humans.

XML Specification: Language with corresponding semantics to the front-end language, but with XML syntax. This exists to allow external tools to more readily create visualization specifications. In some ways the 'Front-End' language is a specification for the XML that should be generated.

TSM Generator: A compiler to translate the specification into a component in the target language. In reality, this will probably *ad-hoc* include the compiler between the front-end language and the XML specification.

Generated Component: Component created by the TSM Generator. This is the focus of the tool chain. In many circumstances, this will be the only part of the TSM system explicitly included in the host application.

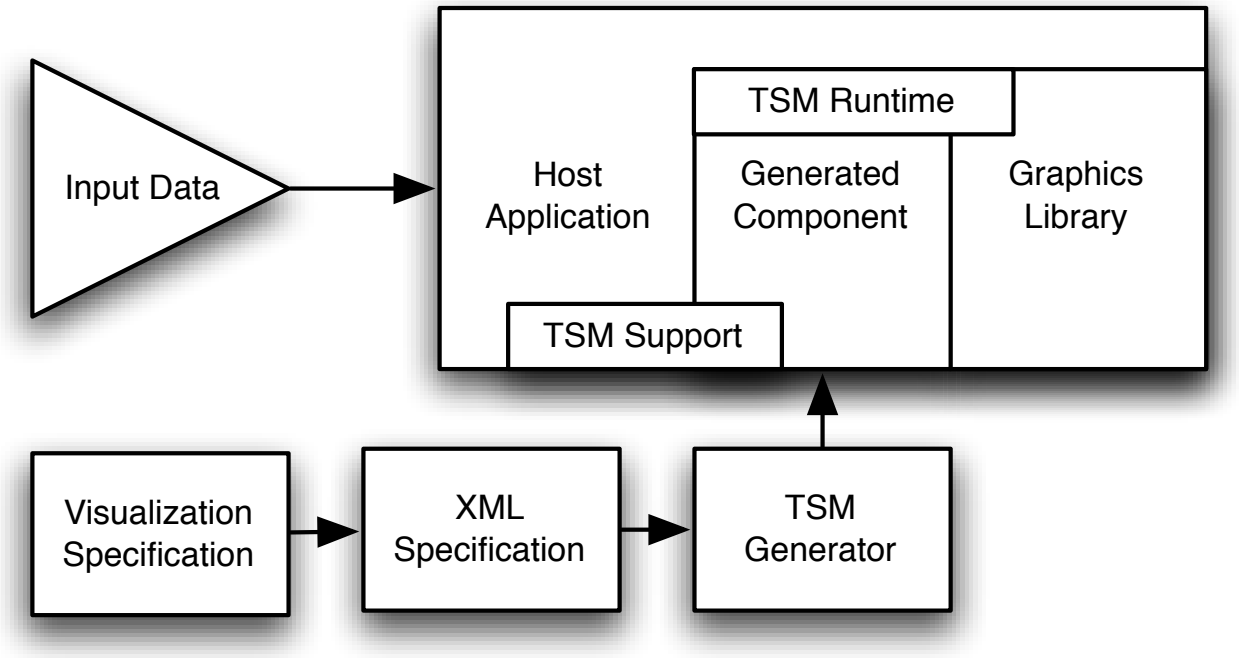

Figure 1: General architecture of a TSM application.

Host Application: A program to host a component generated by the TSM generator. The TSM generator is not intended to create all aspects of a program, just the visualization component and interfaces sufficient to have it interact with a larger program. The host application will be responsible for mediating inputs (user and data), initialization, startup and non-graphical outputs.

TSM Runtime: Support functions defining basic entities (Glyph, Layer, Stream) and providing the guaranteed functionality. This is constructed with a graphics library in the target language and provides services extending those functions to the needs of the Tuple Space Mapper language semantics.

Graphics Library: Basic library in the target language. Examples include Piccolo, OpenGL, the Python Imaging Library (PIL), wxObjects, Swing or an accordion drawing library. The generated component from the TSM pipeline does not replace the graphics library, it sits between the host application and the library. In most cases, the visualization application will only see the generated component. It can, therefore, be agnostic to the underlying library and the TSM compiler system. These only come into concern during visualization specification, not during normal data analysis.

TSM Support: Support code includes various ancillary functions to facilitate integration between the generated component and the host application. This could include class definitions for converting a comma-separated values file to tuple streams and main method to test a visualization on its own (with tuples streams from data files)

Note: In some environments, the TSM Generator may be a part of the TSM runtime. This would allow new rules to be defined while a program is executing, enhancing the interactive capabilities of the system. This makes the most sense in interpreted languages, though any environment that supports runtime loading of modules could support this option.

4.5 Extending and Integrating

The described architecture could be used to generate application components, but the TSM would represent little progress if those generated components cannot integrate with other tools or if they are not extensible after their initial creation. Data exploration is often described as a dialog between an analyst and their data. The dual need for extension of a TSM-derived tool after its initial creation and the integration of a TSM-derived tool with other tools takes on three forms. (Figures 2 and 3 illustrate the problems associated with integration of current applications.)

First, exploratory data analysis takes the form of a series of questions (posed by the analyst) and responses (given by the computer). In visualization, this dialog often follows the mantra "Overview, Zoom and Focus, Details on Demand". Providing a dynamic data environment that facilitates this data dialog is essential if the Tuple-Space Mapper is to have high utility. The ability to ask novel questions is difficult to support, but represents a critical capability.

A second issue is the need to integrate existing visualizations (such as Google maps or a spring force embedding) into a TSM definition. There has been a lot of work done to make visualization techniques useful. Being able to leverage this prior work would be of high utility. An inability to integrate with other tools would be a severe handicap.

A third (closely related) issue is that visualization is usually in a larger context. The ability to integrate the visualization in with another application (freeing the user from saving the data and loading it into another program) would expand system applicability.

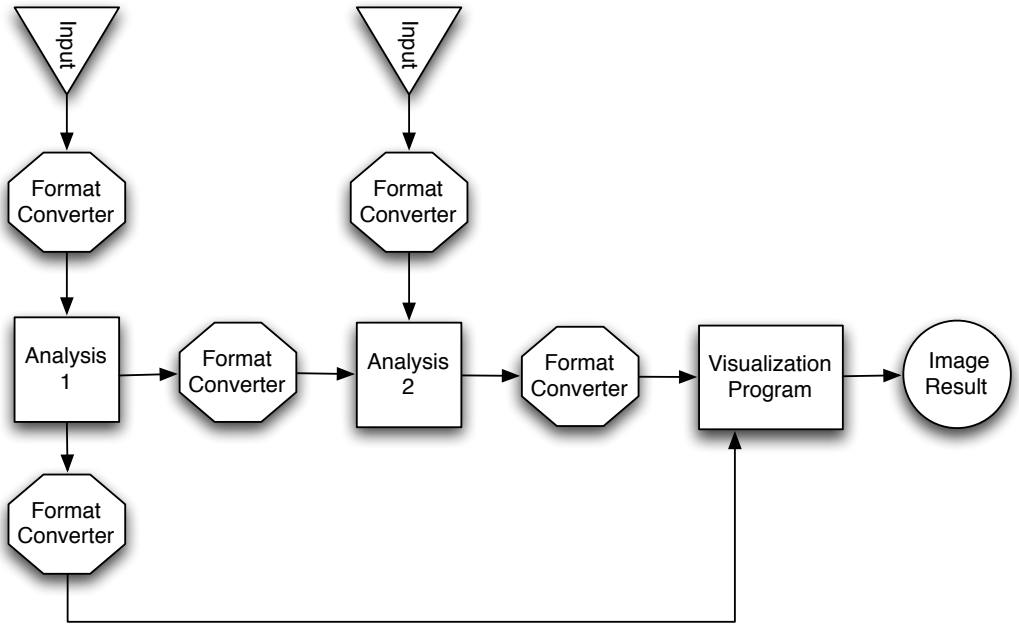

Figure 2: Example tool-chain with two input sources and two analysis steps. Compare this to the top image in figure 4. As each analysis program or data source is introduced, accompanying coordination and conversion steps must *ad-hoc* be included.

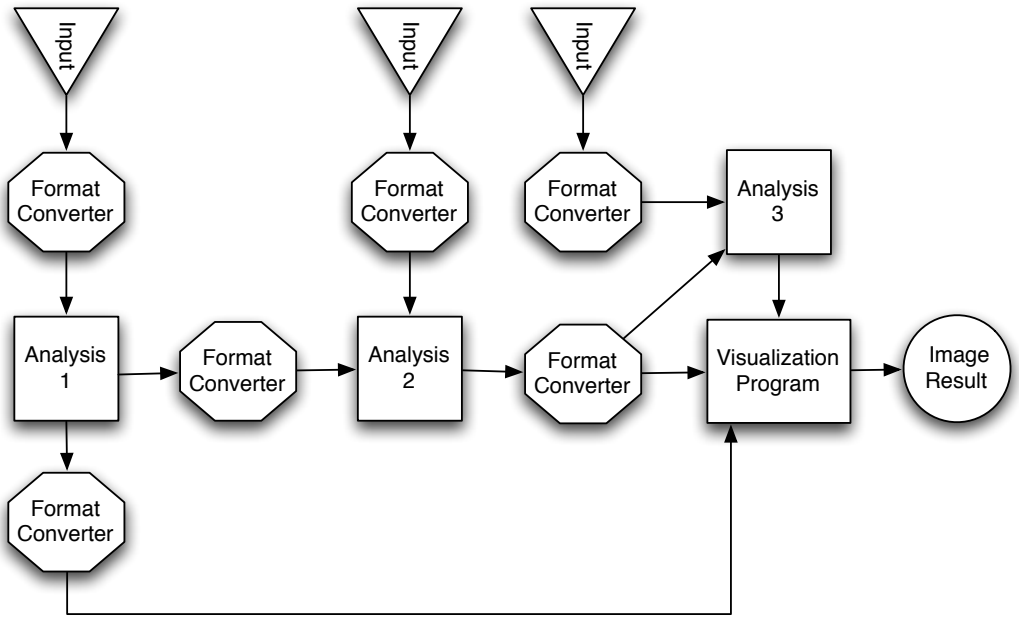

Figure 3: The additional of a third input and analysis step incurs additional conversions, especially as analysis steps inter-related. Compare this with figure the transition between the top and bottom images in 4.

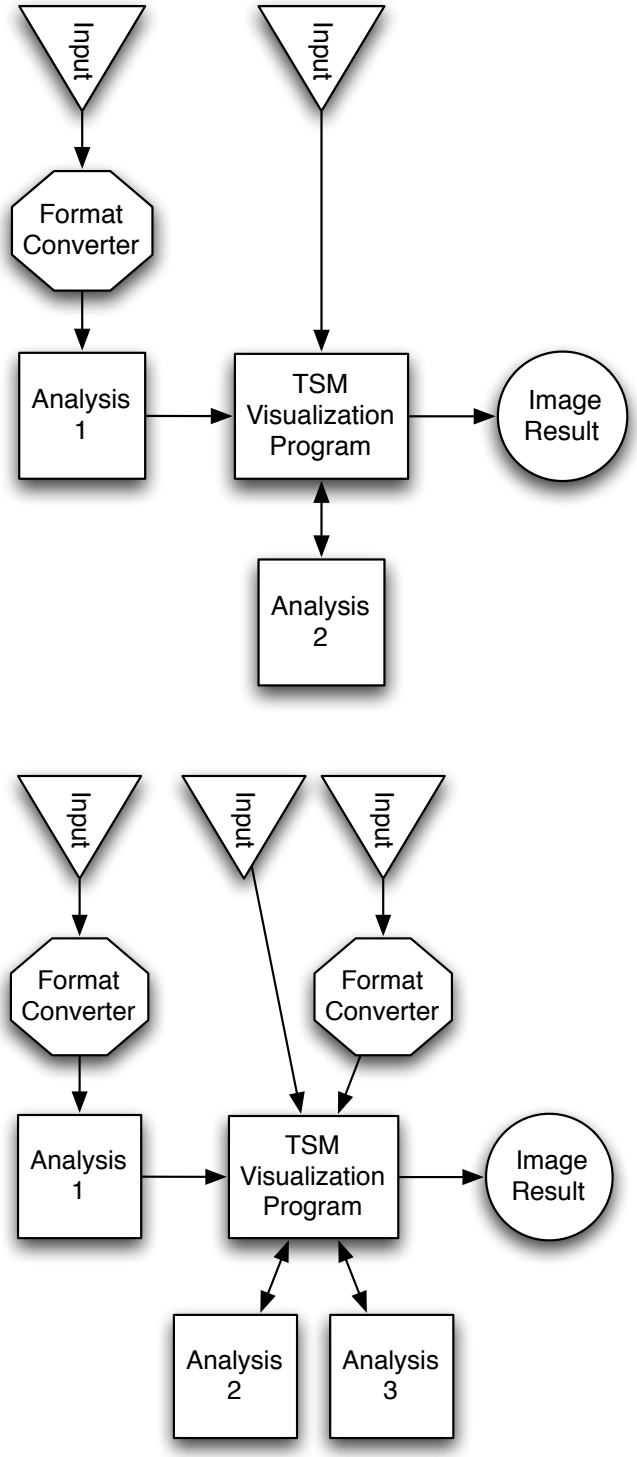

Figure 4: Example tool-chains with a TSM derived application. The two-input process (top image) is simplified (see figure 2). As more inputs and analysis steps are added, this savings compounds (compare with 3). Much of this simplification is enabled by the tuple abstraction.

To facilitate these goals, the following features are planned:

1. **Self contained widget:** Having the visualization wrapped up into a single graphical object (such as a JPanel) allows it to be plugged into an application in a way similar to a button.

2. **Ad-hoc functions:** Being able to define functions on the fly and call them from visualization rules would allow many "I have an idea" quick-checks to be performed. Implementing the TSM in a dynamic language (like Python) or imbedding a language interpreter (like Jython) makes this easier. This option *ad-hoc* makes sense when creating a compiled rule-set as it puts all of the visualization specification in one place.

3. **Tuple input streams:** Using a minimal definition of data input (must be representable a sequence of tuples) maximizes the data that can be loaded. This *ad-hoc* easily maps onto common data formats (relational databases, list of pairs graphs, delimited files, etc). By having a general input format and generated internal representation, the number of conversion steps required in a tool chain is reduced. Combined with ad-hoc functions, this should simplify tool-chains (see Figure 4).

4. **Custom legends:** A legend is a mapping function. Though there are several standard legend functions (like heat-maps or alphabetical axes), the ability to share legends across visual panes and augment standard legends by locking certain mappings would help coordinate across visualizations (e.g. highlighting particular values in all panes). *ad-hoc*, providing complete custom mappings would allow for novel visualization techniques to be explored at low overhead. Custom legends *ad-hoc* introduce a 'tweakable' aspect. Good graphics kits allow adjustments to the exact color, shape, scale or position of elements [9]. These could be encoded as custom legends. This may be very helpful for the presentation aspects of a visualization.

5. **Custom layer definitions:** A layer is a namespace for data. Differing layers do not need to use the same underlying graphics engine. Allowing layer 'types' provides an integration point between existing frameworks with the TSM. A tuple IO abstraction and a coordinate alignment protocol would be required on the layer's side and a composition engine would be required in the component. Such division of labor would allow (for example) data overlays on top of map engines.

6. **Mutable definitions:** The ability to add additional rules to a definition (including introducing new data streams) allows an analyst to incorporate new information after visualization has already begun. This may be the result of external analysis (see Output Streams) or of information becoming appropriate as a visualization progresses.

7. **Output streams:** Allowing the layers to act as streams supports migrating between definitions (see Mutable Definitions above), it *ad-hoc* enables external processing. Taking the visualization data (including X/Y/Color information and the original values used to compute them) and making it easy to output for further processing (such as statistical analysis) is difficult in many frameworks. Having convenient output streams would help tie the visualization and the analytical spaces.

5 Related works

The following sub-sections describe related work. Each sub-section describes the applicable parts of the relevant technology, followed by descriptions of how the TSM system extends or substantially differs.

5.1 GUESS: The Graph Exploration System

GUESS is an exploratory data analysis and visualization tool for graphs and networks. The system contains a domain-specific embedded language called Gython (an extension of Jython) which supports operators and syntactic sugar for working on graph structures in a straightforward manner. An interactive interpreter binds the commands entered to objects being visualized for more useful integration. GUESS *ad-hoc* offers a visualization front end that supports the export of static images or movies [1].

Guess only works with graph data represented as a node-link image. A lot of information visualization works with graphs, but not all of it. The TSM tool seeks to generalize some of the concepts shown in Guess to a broader range of data models.

A common problem with interpreter driven systems is that the actual command sequence used is not the same as the significant command sequence. The actual command sequence involves many forward and backward steps while the significant sequence is only those commands that impacted the end product. Discerning the significant sequence from the actual sequence is difficult (often involving a trial and error process similar to the original one). Having a specification that is interpreted as a whole, as the TSM does, means the ending command sequence is always the significant sequence. This presents challenges such as providing an undo function (no greater than those in the interpreter) and the ability to detect the changed commands and only execute those (this will be much harder).

5.2 Prefuse

Prefuse is a visualization framework with eye towards generality and runtime configurability. This is done with with very general data structures, a strong event framework and small expression language. The expression language allows filters and very simple property mapping to be done at runtime [14].

The threading structure of Prefuse implies the majority of data is present when the visualization phase begins. Many of its activities involve iterating through the whole data set (and if more data is added, the data whole set is iterated again). This is often unnecessary, and may be impossible with realtime data. The TSM focuses on data streams, so full data re-iteration is not assumed to be possible. This implies different data structures and visualization oportunities.

Integrating Prefuse with non-standard methods (e.g. not supplied by the Prefuse developers) requires the developer to conform to their event model and complex interfaces. The TSM concept has a simple interface standard (essentially just specifying the return type) allowing broad integration.

Prefuse requires the host application to copy the data into their data structures explicitly. This requires the developer to understand the visualizations data structures. The TSM concept would allow the developer to specify data structures, but not require it. An iterator would be required to expose the data to TSM as a stream, but TSM would handle copying to internal structures itself.

Prefuse is tied to java. TSM would allow for the same framework to be used in many languages. Only the non-standard functions would need to be recoded if a language change was made (similar to ANTLR's host-language integration).

5.3 Processing

Processing provides a Java library and development environment targeted to visualization tasks. It provides full access to the Java programming language for analysis, but shortens the write-compile-execute-debug cycle by integrating the compile and execute steps in the programming environment. It *ad-hoc* provides library-based tools for many common visualization tasks [8].

The TSM seeks to improve the write-compile-execute-debug cycle in a similar way to Processing, but takes it one step further by providing a DSL. The Processing library and tool still require the programmer to maintain the general-purpose programming language mindset (e.g. data structure and control flow). Using a DSL simplifies this mindset, making the write-compile-execute-debug cycle tighter by having the write and execute cycles conceptually more similar. To avoid loss of the expressive power of a general-purpose language, the DSL maintains bridges to the target language allowing arbitrary code execution, but this is not the main mode of interaction.

5.4 Snap-Together

The Snap-Together system employs a declarative language to control coordination between multiple pre-coded views displayed in separate windows. It provides a way to modify the relationships between views at run-time, including adding new views to a system [22]. Where Snap-Together permits only on coordination, the TSM system tries to expand this flexibility to creation of views as well .

5.5 Chizu

Chizu is a Perl module for GIS annotation. Chizu uses a declarative language to describe annotations to GIS maps in a conceptually simple way. Chizu interprets an annotation file and combines it with the GIS data. It supports ad-hoc location naming to simplify the descriptive process [20].

Chizu only supports GIS data, and only in a very limited way. However, it shows that ad-hoc naming and declarative annotations are an effective way to aid in constructing graphics for presentation from structured data. The TSM is intended to handle many different data types.

5.6 VTK: The Visualization Toolkit

VTK provides a collection of standard algorithms for data analysis and visualization built into an object-oriented framework. Programs written with VTK build a network of visualization components in a traditional language (C++, Python, Tcl and Java are supported). The start of the network is a data source, its terminal is a display object [27].

The standard methodology in VTK is to create a static network in the language of choice. This provides a convenient way to define a visualization system, but it encumbers the definition with the non-visualization targeted components of the language as well.

Ad-hoc processing in VTK is supported, provided the programmer supplies a component that compiles to one of their interfaces. This places responsibility on the implementer to understand the semantics of all of the standard methods and implement them accordingly. The proposed mapping system of the TSM would support the ad-hoc processing in a more straightforward way.

VTK specializes in 3D scientific visualization, the Tuple Space Mapper would be targeted at 2.5D information visualization applications. Additionally, the TSM system would be targeted at lower-level integration of predefined components, more in line with philosophy of VTK but not relying on a traditional language to define the integration (rather generating the traditional language integration from a declarative description).

5.7 Picture Grammars

Picture grammars are an extension of string grammars (such as those used in compiler compilers) to 2-dimensions. They can be employed to describe fractal structures. Pictures grammars are of note because they are usually declarative but integrate conditionals that are derived from the external environment [7]. The formalized interaction with the environment is very limited, but shows a way for declarative image-genarating languages to take into account external conditions (unlike ContextFree, see below).

5.8 ContextFree

ContextFree is an example of a picture-grammar based language and supporting runtime for programmatically generating graphics. It is a graphic design tool/toy that allows graphics to be generated from grammar-like descriptions. Rules are applied recursively until a graphic primitive is reached, then primitive is displayed [5].

Context Free does not consume input. It simply iterates. Terminal conditions are stack depth and graphic primitives. When all iteration paths have reached a terminal, the program stops. Context Free shows that a declarative rules-driven system can directly specify a visualization. The TSM system will extend it by having the rules consume external inputs (not just each other).

5.9 Database Visualization

The field of database visualization is broad, [11] proposes an architecture for rapidly preparing database tables for a visual representation. It was illustrated that a declarative, rules-based language could efficiently handle mapping data to suitable representation for visualization. The efficiency applied both to program execution time as well as human interaction efficiency (it was easier for volunteers to grasp than prior methods based on correct application in pencil/paper testing scenarios). The visualizations handled by the tool described were of only a few types, but the concepts embodied in the query mapping language (called MQL), Plot and Filter modules are of particular interest.

The TSM generalizes this functionality to any stream-based environment. Traditional database query record sets can be thought of as streams of records. This extends simply to the 'stream queries' such as those used in the Calder/dQUOB system [26]. It simplifies the use synthesizing multiple data sources (a needed work for Visual Analytics). It eliminates the need to have a database per-se by abstracting a database to a stream source.

The TSM language could be viewed as an extension of MQL. MQL is a domain-specific language used to transform data into a form amenable for use with pre-defined visualization modules. In particular the mappings "...provide order and scale for input data..." (e.g. convert categorical data to numerical data or binning) [11]. The transformed data is passed on to a Plot module. The Plot module is coded in a traditional language (in this case Java) and used to produce 'standard' visualizations. The TSM system is targeted at the Plot and Filter modules, providing a declarative language for constructing these modules instead of forcing visualization programmers to use standard ones.

5.10 Stratego

Stratego is a language processing tool with arbitrary tree manipulations. It is intended for compiler creation (and is thus a compiler-compiler) but Stratego unique in its field for several reasons. It codifies concepts of tree traversal in unusual ways, but thereby provides flexibility and expressive power in a functional fashion that is often only available by employing ad-hoc tools in other compiler compilers. Stratego takes a number of elementary tree traversal operations and a builds a powerful tree manipulation language [2]. It has arcane error messages, but otherwise it a strong tool that generalizes many concepts and allows them to be used via a declarative language. Stratego generates C code that then must be compiled, provides an API for using generated components and a data structures library for simplifying integration. The TSM differs in terms of target appication domain, but maintains many philosophical links to Stratego.

5.11 ANTLR: Another Tool for Language Recognition

Given a language specification in a declarative language, ANTLR generates code in any number of languages (C, C++, C#, Java, Python, Tcl). The code generated consumes character streams and converts them into token streams then abstract syntax trees. The abstract syntax trees can be further manipulated by an ANTLR 'tree grammar' described in a similar declarative language. AST and Tree Grammars *ad-hoc* support arbitrary ad-hoc actions specified in the target language. The generated code plus the ANTLR runtime can be used as a component to any project capable of using the target language. This illustrates that a stream-processing program can be specified effectively in a declarative language and shows precedence for multi-language targets (with accompanying runtime)[24].

The TSM tool would employ a similar architecture: declarative specification handed to a compiler-like tool that generates code in the target language; runtime interface to expose the functionality as a component in a larger system; ad-hoc rules to be employed where the declarative language's features are insufficient or cumbersome. However, it would be targeted at visualization instead of tree manipulation. The multi-step process ANTLR encourages through its tree grammars (use a separate grammar for each transformation, then chain them together) is analogous to a visual layering approach, but with different mechanics.

5.12 Generative Programming

The process of using programs to create other programs is broadly referred to as Generative Programming. Since the TSM is using a specification to generate a program component, it falls under this umbrela. It could be thought of as either a compile-time generator, or when the TSM generator is included in a host application, as 'dynamic compilation' [6].

5.13 Vizz3D

Vizz3D is the visualization component of a reverse engineering framework described in [23]. It provides runtime definable visualization module. However, this is limited to configuration of existing modules that have been coded in a traditional language (such as Java). In many ways this is similar to [22], but differs in that it focuses on mapping values more than coordinating views. Configuration is a relatively high-level (depending heavily on reflection for runtime inspection of argument types and interface definitions) and possibly suffers from a high abstraction penalty. The TSM would be targeted at providing compile-time definition of visualizations to increase the flexibility in the types of displays that can be used, rather than runtime-time selection of mappings to pre-defined visualization views.

Note: It may be desirable for certain components to work in a runtime-configurable mode. This could be especially useful in a filter and focus styles of interaction. Therefore, having a subset of the language that operates more like a configuration language on pre-defined components may be desirable. This may lead to parallel components (one only compile time configurable, the other runtime configurable).

6 Conclusions

The challenges faced by the Tuple-Space Mapper are varied and rich for research. It is clear that many of these problems have been encountered before in other contexts or as isolated issues in visualization. We believe that the solutions can be unified to create an extremely powerful tool, but such integration is non-trivial. We believe the design presented here for the Tuple-Space Mapper, e.g. employing a Domain Specific Language to generate visualization components, has the potential to address these issues. As such tools are developed, the current limits of information visualization adoption will recede and knowledge of effective visualization practices will expand..

7 APPENDIX: EXAMPLES

7.1 Concepts and Syntax

This is not intended as an exhaustive list; just enough to give a concept of the language.

Glyph: A display element like a circle or a label. Explicit reference to GLYPH terminates a chain of transformations.

LAYER <*name*>*: <*stream*>: Logical group of Glyphs and a namespace. The ID of every glyph on a layer must be unique, but the IDs on different layers may be the same for different glyphs. Since layers are used as namespaces, they provide a searching function 'Find' that can be used to coordinate across layers. Layers are defined in their stacking order with the first node being the bottom.

LEGEND <*name*>: <*rule*>* A legend is a mapping function declaration that *ad-hoc* has standard visual representation. A legend may be referenced in a mapping function by the name given.

STREAM <*name*> (<*name*>*): <*rule*>* Declares that a tuple stream exists. The format gives a name to each tuple element for future reference. By default, all names appearing in the associated rules the stream declaration refer to the stream values.

Simple Rule: <*name*> –> <*value*>: The named element is used to determine the value via the default function. The default function will depend on the value specified and (potentially) runtime characteristics of the data value. Items on the left-hand side of the arrow are matched in order to the items on the right-hand side of the arrow.

Complex Rule: <*name*> –> <*function*> –> <*value*>: The function is used with the value of the tuple supplied to the determine the value. Layers used as functions use the ID as a map to retrieve an item that share the ID or create one if it does not already exist. There may be as many steps in the chain as are required. As in the simple rule, values on the left-hand side of an arrow are used as arguments to function on the right-hand side of the arrow. The return value of each function (which must be a Tuple) is used as arguments to the next step.

DEFAULT: Defaults may occur in many contexts. This allows explicit reference. In a Legend, the default is the value returned if no other item matches. If a default GLYPH is defined, it sets the property defaults for the current visualizations.

{...}: Code in curly braces will be literally included in a 'sensible' context when the component is generated. It should be written in the language that is being targeted by the generator. Examples are given assume Java is the target language.

7.2 Example Specification: Node/Edge Plot

Graph layout algorithm are often run off-line, and re-integrated with the data at rendering time. This example follows this pattern. A list-of-pairs file is combined with the output of a graph layout algorithm. Each is stored separately, and combined using the layers concept. Additional information is placed on a 3rd layer that comes from a separate data source (e.g. a database with True/False values derived from a query). An example program of this type can be seen in Figure 5.

```
{import java.awt.Color;}

LEGEND Coloring:
  DEFAULT -> {new Color(200, 20,20);}
  True -> BLUE
  False -> GREY

LAYER Nodes:
  STREAM Layout (name, x, y):
    name --> GLYPH(ID)        # (1)
    CIRCLE --> GLYPH(Type)    # (2)
    name --> GLYPH(Label)
    (x,y) --> GLYPH(X,Y)

  STREAM Attributes (name, value):
    name  --> Nodes.Find --> GLYPH      # (3)
    value --> Coloring --> GLYPH(Fill) # (4)

Layer Connectsion:
  STREAM Conns (start, end):
    AUTO_ID --> GLYPH.ID
    LINE --> GLYPH(Type)                # (2)
    start --> Nodes --> GLYPH(X1, Y1)  # (5)
    end --> Nodes --> GLYPH(X2, Y2)
    ARROW_HEAD -> GLYPH(EndShape)

# (1) Glyph ID is used in lookups.  When
      assigning in the default form, it
      will either create a new item or
      replace an existing one with the
      same ID.
# (2) The Glyph type determines the visual
      properties that can be set.  Lines
      have start and end, but simple shapes
      (like circles) only have a single X
      and Y.
# (3) Using a 'Find' on layer indicates
      that the operation should only be
      performed if an item of the given
      name can be found.
# (4) The Coloring function is the ad-hoc
      function defined at the start of
      this example.
# (5) Since this rule is on a separate
      layer, it is inferred that the find
      function will be used.
```

Figure 5: Example TSM Specification (with annotations).

ACKNOWLEDGEMENTS

The concepts of this document have passed through many hands in editing. Thank you to the members of the Open Systems Laboratory (Indiana University, Department of Computer Science) and the InfoVis Lab (Indiana University, School of Library and Information Science) who have given feedback in the process. This work was supported in part by a grant from the Lilly Endowment.

REFERENCES

[1] E. Adar. Guess: a language and interface for graph exploration. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 791–800, New York, NY, USA, 2006. ACM Press.

[2] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.16. a language and toolset for program transformation. *Science of Computer Programming*, 2007. (To appear).

[3] C. Chen. Top 10 unsolved information visualization problems. *IEEE Comput. Graph. Appl.*, 25(4):12–16, 2005.

[4] E. H. Chi. Expressiveness of the data flow and data state models in visualization systems. In *Advanced Visual Interfaces*, Trento, Italy, May 2002.

[5] C. Coyne, M. Lentczner, and J. Horigan. *Context Free Art*. http://www.contextfreeart.org/index.html.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. ACM Press/Addison-Wesley Publishing Co, New York, NY, 2000.

[7] F. Drewes. *Gramatical Picture Generation: A Tree-Based Approach*. Texts in Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1998.

[8] B. Fry. *Computational Information Design*. PhD thesis, Massachusetts Institute of Technology, 2005.

[9] B. J. Fry. Organic information design. Master's thesis, MIT Media Lab Aesthetics & Computation Group, 2000.

[10] J. Gray. Database management: Past, present, and future. *IEEE Computer*, 29(10):38–46, 1996.

[11] D. P. Groth. *Architectural Support for Database Visualization*. PhD thesis, Indiana University, 2002.

[12] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, Cambridge, MA, USA, 1992.

[13] J. Heer. Socializing visualization. In *CHI 2006 Workshop on Social Visualization*, 2006.

[14] J. Heer, S. K. Card, and J. A. Landay. Prefuse: a toolkit for interactive information visualization. In *CHI '05: Proceeding of the SIGCHI conference on Human factors in computing systems*, pages 421–430, New York, NY, USA, 2005. ACM Press.

[15] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1978.

[16] D. Kimelman, B. Rosenburg, and T. Roth. Visualization of dynamics in real-world software systems. In *Software Visualization: Programming as a Multimedia Experience*, pages 293–315. MIT Press, Cambridge, Massachusetts, 1998.

[17] E. Kraemer. Visualizing concurrent programs. In B. Price, J. Stasko, J. Domingue, and H. Brown, editors, *Software Visualization: Programming as a Multimedia Experience*, pages 237–256. MIT Press, Cambridge, Massachusetts, 1998.

[18] J. Levine, T. Mason, and D. Brown. *Lex & YACC*. Nutshell Handbooks. O'Reilly & Associates, Inc., Cambridge, MA, USA, 1992.

[19] F. T. Marchese, editor. *Understanding Images: Finding Meaning in Digital Imagery*. Springer-Verlag, 1995.

[20] M. Meiss and H. Roinestad. *Chizu Tutorial*. http://iv.slis.indiana.edu/lm/lm-chizu.html, 2004.

[21] G. J. Myatt. *Making Sense of Data: A Practical Guide to Exploratory Data Analysis and Data Mining*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2007.

[22] C. L. North. *A User Interface for Coordinating Visualizations Based on Relational Schemata: Snap-Together Visualization*. PhD thesis, University of Maryland, College Park, May 2000. Chair-Ben Shneiderman.

[23] T. Panas, R. Lincke, and W. Löwe. Online-configuration of software visualizations with vizz3d. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 173–182, New York, NY, USA, 2005. ACM Press.

[24] T. J. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, 2007. ISBN 0-9787392-5-6.

[25] M. Peter, A. Blackwell, and T. Green. Cognitive questions in software visualization. In J. Stasko, J. Domingue, H. Brown, and B. Price, editors, *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1998.

[26] B. Plale and K. Schwan. Dynamic querying of streaming data with the dquob system. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):422–432, 2003.

[27] W. Schroder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics (3rd Edition)*. Kitware, Inc., USA, 2002.

[28] B. Shneiderman. Information visualization: The path from innovation to adoption. In *IV '01: Proceedings of the Fifth International Conference on Information Visualisation*, page 3, Washington, DC, USA, 2001. IEEE Computer Society.

[29] P. ten Hagen, T. Hagen, P. Klint, H. Noot, H. Sint, and A. Veen. *ILP: Intermediate Language for Pictures*. Number 130 in Mathematics Center Tracts. Amsterdam Mathematics Center, Amsterdam, Netherlands, 1980.

[30] J. J. Thomas and K. A. Cook, editors. *Illuminating the Path*, chapter Chapter 6. IEEE Press, 2005.