Library and application-based information visualization tools incur significant penalties that hinder the adoption of visualization as a technique in a variety of situations. Traditional library use necessitates familiarity with the data structures and control flows that are integral to traditional programming, but not central to visualization. Additionally, many library based visualization tools do not explicitly address interaction issues, forcing programmers to fall back on language provided interaction metaphors instead. Task-specific visualization applications alleviate these issues, but limit users to pre-supplied visual arrangements and induce users to move data between applications as their needs change.

We propose a declarative, domain-specific programming language approach to visualization tool construction. We believe this approach will simplify the process of custom visualization program creation, simplify integration with many tools and ease the modification of existing tools. Further, the conceptual model underlying this custom language can be explicitly constructed to handle both data and interaction components in a consistent manner. This proposal outlines the architecture for such a tool, exploring the technical requirements and challenges. Constructing this tool and model will expand the circumstances under which visualization techniques can be readily applied and extend the theoretical understanding of those techniques.

1 Introduction

Information Visualization has encountered many roadblocks to adoption beyond academic research or 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. Each of these methods are expensive to employ and costly to maintain. These costs create barriers to the use of visualization as a tool in many potential applications. As such, visualization has seen limited growth relative to its attention in the research community. In essence, 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 by Heer [30]).

Many of the issues of adoption are a direct result of the distance between the visualization concepts and the implementation models that are used when working with traditional programming
languages. We believe that a model that holds close to the core concepts of information visual-
ization will be required to reduce the cost of adoption. Moreover, if such a model were combined
with a generative system, maintenance difficulties would also correspondingly decrease. We be-
lieve that such a model can be realized with a declarative domain specific language to provide
a basis for its use (both directly and through other tools). The need for flexible, extensible visual-
alization has been recognized many times before (including in [5, 27, 38, 44]), but we propose a
novel approach using a new conceptual model for integration with existing systems. The need to
improve the programming of visualizations was recognized by Fry [26], but the Processing lan-
guage simplifies a limited set of operations while retaining a traditional language structure (and
thus a focus on traditional programing issues). To expand beyond this prior work, we describe
the embodiment of our concepts in a declarative language called Stencil. In this paper, we explore
the motivations and goals of Stencil. This proceeds as a discussion of the problems encountered
in visualization adoption, a preliminary model that approaches some of those issues, exploration
of language features to support that model and provide integration, description of a practical
architecture for implementation and a discussion of related works.

2 Motivation

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 commu-
nication 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 com-
community in its own right). A commonly suggested solution is to have visualization researchers work
on context-sensitive projects (solving the issues of group X by visualization) to increase the num-
ber of fields manually [66]. This is a top-down solution where 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 adoption of visualization tools to those
with funding to hire (or time to train) such a practitioner.

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 inter-
face. When a database was desired for a project, the first question always had to be “Which one?”
This early decision influenced many issues of software structure and personnel training, making it
hard to change in the future. These issues (forced early decision and change difficulty) eased with
the introduction of the relational database model and its accompanying Structured Query Lan-
guage (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 distinction between process and product allows projects to maintain a loose
commitment to the decision of which database to use and the exact data organization. To config-
ure and interaction with most databases only standard SQL need be employed. 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 prowess, 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 [28].
The visualization community would benefit from a similar conceptual model and structured, focused language. Since existing visualization frameworks require an expert to use, this limits the projects that can employ visualization. Worse yet, visualization frameworks 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 conceptual abstraction for visualization (like the sets and relational structures of relational algebra) and a flexible, integratable tool capturing those concepts (like SQL) would enable people to begin to use visualization at their own discretion and then call on the (expensive) professional practitioner when the benefits clearly out-weigh the incumbent costs.

The potential for this bottom-up style of adoption has recently been recognized in the visualization community, but few formal methods of addressing the related issues have been proposed [32, 40, 81]. Facilitating this style of adoption of information visualization is a major motivation for the Stencil language, architecture and the accompanying model discussed in detail in the remainder of this document.

3 Visualization Models

Information visualization systems are typically constructed according one of two theoretical architectures, either the data flow model [65] or the data state model [16]. These two architectures address differing concerns for speed, composition and process visibility. The following overviews describe each and their inter-relationship.

3.1 Existing Models

The data-flow model of visualization grew out of the practice of the scientific visualization community. The central idea is to build data processing pathways. Each phase of the pathway is an encapsulation of a particular operation. Operations may be ‘sources’ that produce data to be consumed by other operation, ‘sinks’ that receive information or both (the most common type). Data objects are passed between each operation. In this way, data ‘flows’ through a network of operations (see Figure 1). The final step is a sink that performs final rendering. This model is strongly rooted in the pragmatics of control flow and data storage for efficient handling of large volumes of data. VTK is based on these principles [65].

The data-state model is described by Chi [16]. Notable implementations based on this model are the ‘visualization spreadsheet’ application described by Chi and the Prefuse visualization framework [31]. The core concept of the data state model is that each distinct stage of a visualization process is a state of the data. Operators perform transformations on the data and move it between states. An example of this is converting raw-text to stemmed text. The original data state is free-form paragraphs. The operator is a stemmer. The new state is a collection of canonically formed words. Conceptually, the raw text still exists and so instead of stemming, the next analyst may decide to perform stopping, histogram the counts, invert the stemming (to get just the prefix/suffix usage pattern in stead of the root word pattern), etc. States are divided into four (sequential) categories: Raw Data, Analytical Abstractions, Visualization Abstractions and Views. Transformation operators are categorized according to the characteristics of the target state. The three operator categories are Data, Visualization and Mapping. In many ways the data-state model (nodes are data states and edges are transformation) is a conceptual dual to the data flow model.
Figure 1: Illustration of series of transformations in the data flow model. See Figure 2 for the same operations in the data flow model.
Figure 2: Illustration of series of transformations in the data state model. See Figure 1 for the same operations in the data flow model.
Figure 3: Major decompositions of the visualization process, rough aligned to their division of tasks. Stages/tasks and alignments were derived from [15, 26, 31, 65].

Figure 4: Example visualization software aligned to the Processing pipeline stages. Though VTK and Prefuse include interaction, it is not derived from their model (and thus not include above). Further, the VTK and Prefuse implementations focus on representation, not refinement. Altering a representation often takes as much effort as the original definition. ELog ‘Creative Discovery’ provides some tools for interaction and refinement but these tools are commensurate with the smaller scope it provides in other stages of the process.

(where nodes are transformations and edges are data traces). It is argued that the data-state model represents a more analyst-centered view of visualization, as the ‘locations’ are more tangible entities and the ‘movements’ are operations (see Figure 2). Though this represents progress towards a user-focused representation, it still focuses programming on control flow and data-storage (especially in the representational stages).

3.2 Model Shortcomings

The data-state and data-flow have been shown to be expressively equivalent [15]. However, no attempt has been made to show that they cover all of the visualization problem space. Each of the two models provides useful a useful view on building a visualization program but unfortunately
each also has substantial shortcomings that make them difficult to apply and applications based on them difficult to extend.

The data-flow model is entirely concerned with data flow and storage. Library-provided components from data-flow libraries provide discrete processing steps; however, managing the associated control flow and many aspects of data storage are left up to the programmer. VTK [65] reduces some of the issues by allowing the programmer to ‘wire together’ the components and executing the resulting network in a separate thread. However, the control flow must be entirely specified in this manner before any results can be viewed. Additionally, the operators provided are often quite abstract. As a conceptual framework for structuring control flows, it is straightforward. But as a conceptual framework for building visualizations, it is quite distant from the core issues of visual representation and transformation.

The data-state model suffers from a similar problem of abstraction. The primary downfall of the data-state model for creating visual representations is that it is too general. It explicitly accounts for a wide variety of phenomena (from data collection through analysis and finally to visualization). This leaves the actual visual representation and interaction processes compressed into a single transition. Visualization creation is a rich area and would benefit from more detailed treatment. To be effective, such details must be integrated with the analysis and collection processes, but treating it as “more of the same” is misleading and potentially damaging.

Both models also share two very important domain mismatch issues. First, both of them are quite separated from standard graphic design practice. They provide very few tools to support the level of control or the types of abstractions that are supplied in other tools for visual communication. The separation from graphic design practice provides a high barrier to entry for those most skilled in visual communication. This shortfall has been noted before by Fry [26], but his solution, the Processing language, provides only minimal steps towards bridging this gap (see Section 6.7.3 for a more complete critique). The second issue is that neither model includes interaction as key component. Information visualization has been described as representation + interaction but the interaction is often treated as a secondary consideration [88]. The before discussed conceptual models reinforce this secondary position by not accounting for this important behavior. Toolkits that do provide interaction support typically use a metaphor derived from the host language Examples of such metaphor borrowing include the use of event-listiners in the Prefuse and Piccolo frameworks, or the button constructs in VTK and OpenGL [8](see Figure 4). The Processing framework is a counter-example to this trend, allowing access to mouse and keyboard state through special variables instead of forcing the use of event-listeners required by its host language. The general reliance of the language-supplied interaction models requires additional consideration by visualization application programmers to aspects removed from the issues of visual representation.

4 Declarative Visualization

The Stencil system aims to reduce the problems associated with adoption of visualization by (1) moving issues central to visualization in to the foreground while (2) retaining access to the power of existing algorithms. The conceptual model necessary to concurrently achieve these goals is the focus of the next section. Following that, we explore the pragmatics of exposing that model through a Domain-Specific Language (DSL). This includes a discussion of additional concepts
Figure 5: The Stencil model aligned to the Processing pipeline stages. (See Figures 3 and 4 for corresponding treatment of other models). Stencil does not directly provide assistance in acquire, parsing or mining, but provides links to other software that does. Furthermore, the model provides differing amount of focus on filtering, representation, refinement and interaction. This variable focus and partial coverage aides in simplifying the model. Further, the links to external code allows this simplification without a corresponding loss of expressive power.

for the Stencil language and how the system architecture supports improving the accessibility of visualization.

4.1 Model & System 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 Stencil system, we seek to use a declarative language built on simplifying this mapping process, employing a stream processing metaphor (as per YACC/Lex [36]). Specifications written in the Stencil language will be used to generate components, that are then hosted in larger applications. We believe such a system will enable:

1. Rapid visualization prototyping using real data [71]
2. Practical and conceptually elegant synthesis of multiple data sources [72]
3. ‘Data conversations’ by easing customization of visualization applications [73]
4. Decoupling the visualization specification from its implementation, thereby allowing greater flexibility through a visualization-focused upgrade path

4.2 Model Concepts

The following concepts are considered the starting point for a model to reify as the Stencil language. These elements are believed to capture many of the core concepts of a visualization that can be directly represented in the Stencil language. They are based on the experience of a preliminary declarative visualization system implemented in the Summer of 2007 (the system was called ‘ThisStar’ and the underlying concepts were referred to as a ‘Tuple Space Mapper’; see [19,20] for details). This is likely not a complete list of concepts required for the model (and some of these ideas may be cut or combined), but this represents a departure point.
Core data concepts identified in the ThisStar implementation will probably be little changed as they have proven effective in capturing the required ideas. The two major data abstractions are:

**Tuples** are the basic data unit. A tuple is a compound data element where the order of the constituent data points is significant. Classic examples are coordinates ((X,Y,Z), notice X is always first, Y second and Z third) and database record-set rows. All of the parts of a tuple relate to the same base entity (often identified as an element in the tuple). Tuples are the smallest unit of data passed around the system, and are thus analogous to the ‘information packet’ found in flow-based programming [43].

**Streams** represent groups of tuples. A stream is a forward-only reading information source, items once seen cannot be recalled unless they are explicitly stored. Streams are the data interface between a Stencil derived component and the rest of a software system. All information entering or leaving the component can be represented as a stream (though there may be other means of accessing the information as well).

Logical graphical elements comprise the items that will eventually be rendered. Upon rendering, the logical elements actually represented are referred to as ‘marks’. The following logical elements are anticipated in the model:

**Glyphs** are visual representations of data. This is a circle or a dot or text, etc. The goal is to turn abstract data spaces into glyphs. Glyphs are Tuples with a minimum subset of attributes. These minimal attributes are ID, coordinate-system appropriate position. Additional attributes specialize glyph types and may include size, coloring, mark styles, and labeling.

**Glyph Groups** create groups of related glyphs. Glyph groups are glyphs themselves, having the minimal attributes described above. However, they are compound elements of finite size. Simple glyphs can be thought of as tuples, glyph groups then are small tables or as tuples with several repeated fields. The ID of a glyph group is a namespace identifier for the glyphs it contains. Since groups are glyphs themselves, they can be further grouped. The globally unique identifier of any glyph is its path from the root through all of the enclosing groups. Glyphs in a group must have unique IDs, but between groups the names may be shared. Glyph Groups in Stencil are **not** glyph prototypes that may be instantiated, but rather collections of glyphs that share properties, like width or coordinate system (including polar vs. cartesian etc. and origin). When groups use differing coordinate systems, they must adhere to a coordinate alignment protocol to permit coordination of glyphs between groups in an absolute sense.

**Layers** form the root of the group hierarchy. Layers are like glyph groups, in that they contain glyphs but have a few restrictions and additional properties. In terms of restrictions, layers (and only layers) may appear as children of the root of the glyph hierarchy. As a major grouping unit, they provide a means for setting defaults on glyphs to create a common visual style for all elements in a given layer. Layers also need to provide a searching function for all contained glyphs (regardless of depth in groups) so their properties can be referred by other layers or modified by subsequent data mapping operations. Layers also perform the function of the namesake and provide the major unit of depth ordering.
4.2.1 Display

To represent the physical display of information on the limited display spaces actually available, the following grouping elements are considered for the Stencil model:

Canvas defines the rendering space for the eventual marks. All layers cover the entire canvas (regardless of their coordinate systems) but may have different origins. The canvas is a logical space, the view is the actual rendering of a section of the canvas. The canvas sets properties for the entire visualization.

Views are a slice of the canvas as it is rendered onto a device. This is a page from a printer, the bounds of a file or a window on a screen. Views form the basis of the visual interaction and represent the only implicitly non-stream output of a stencil component. Since views are a data destination, but not a data source, this special status is not believed to interfere with the streaming model employed in the rest of the system.

Manipulating data from streams is achieved via a mapping operator. Data tuples enter a mapping function that produces new tuples on exit. This is analogous to the operators found in data-flow languages. In a data-flow language such an operation is the only relationship possible between processing entities. However, in the Stencil model the operators only act within a limited scope, while larger relationships are captured as the composite of mappings contained with a layer. Mapping operators may be chained together to create complex results. Two special classes of operators exist: the legend and the foreign operator. Legends are operators that also have a visual representation, like a legend on any chart or map. Foreign operators will be discussed later.

The preceding elements already provide a few benefits over existing models that should be preserved as changes are made. First, the introduction layers provides an organizational structure that mimics that found in many graphic-design oriented products [83]. This moves the conceptual level of the Stencil model (and language) closer to that used in other visual communication environments. Second, the streams and tuples are interface-level descriptions of a data model. As such, the underlying storage model is left unspecified. This gives freedom for implementers to choose the most appropriate model for the situation (internal table, graph, automatic generator, SQL database lookup, etc.). Using streams allows for unbounded information sources as well. Finally, since mouse and keyboard interaction can be modeled as a stream of tuples, the the data abstractions, coupled with the explicit inclusion of the display, allow interaction to be treated on equal footing with representation concerns. This makes accesses to many common single-user interaction techniques straightforward and allows more exotic techniques, including multi-user inputs and collaborative techniques [84]), approachable.

4.3 Pragmatic Stencil Language Features

As mentioned above, we seek to employ a declarative DSL that includes links to traditional programming constructs. We feel this arrangement has a number of both technical and practical advantages. First, such a language can explicitly expose the visualization described. Where general purpose languages usually burry the visual representation tasks, the DSL can explicitly expose representation and integration issues while employing appropriate visual communication concepts. Second, a DSL compiler relieves the programmer of explicit control flow and data representation concerns [22]. This relief allows for rapid prototyping of visualization applications.
using real data and eases modification of existing tools when updates are required [79]. Finally, it reduces the complexity of the visualization tool chain by presenting a simple integration model that permits modification and extension of a visualization by applications at execution time.

The model described above does not include tools to integrate a visualization with a larger system beyond the basic Stream (which is only described as input above). It also does not acknowledge elements (such as data type) than can be used to make more efficient programs. In order to be useful, the Stencil language must include additional facilities to respect physical constraints, aid in optimization and, permit communicate with other programs (or program components). A number of these are described below. Many of them are specializations of the concepts listed above, but benefit from separate identification and further discussion.

4.3.1 Optimization

Opportunities for optimization are exposed through providing additional information about the types of data being processed and the sequence of arrival of that data. Optimizations based on both type and sequence impact both the memory footprint and the processing speed of the application. The core Stencil model does not explicitly address either of these opportunities. Annotations on tuples could provide data typing facilities. Type annotations will likely be treated as optional, maintaining a straightforward language that is easy to play with (like many type-less interpreted languages) and will likely derive from statistical (rather than traditional computer science) data types. Sequencing operations is a more complicated task than data types. At a minimum, giving tools for prioritizing and coordinating consumption from multiple streams are anticipated.

4.3.2 Integration

Integrating with a surrounding context can be divided into two parts. First is the interaction of a Stencil component with its host program (discussed here) and second the interaction of a Stencil-based program with a toolchain. The basis of interaction with a host program is the introduction of ‘foreign’ entities (so called as they are not ‘native’ to the stencil system). The core foreign concepts are: Layer, Operator and Variables. The first two are extensions to concepts described above. The foreign variables represent a new concept. These three provide the basis of Stencil’s integration with existing systems and user interaction.

Layers imported from other systems provide a means to use the Stencil language to interface with existing graphics. Such foreign layers need only provide the minimal interface (name-space, coordinate alignment and drawing on a canvas) to interact with other layers. As foreign layers, they need not expose additional functionality (add/remove glyphs, set defaults, etc.). This provides opportunities to, for example, use Stencil with GIS systems to produce thematic maps. The GIS system can handling the mapping and perspective issues and stencil can focus on the data overlay. This nearly trivial, as the heart of it is an image import, but the required search/find features and coordinate alignments make such foreign layers more than mere images.

Operators may be imported from any source language. The only restriction is that they return tuples. This provides ad-hoc functionality to the Stencil language, and any computation
process can be instantiated in the foreign operator. This mechanism is similar to the ad-hoc syntax-direct transformation found in compiler-compilers [36,49] and may be implemented as an embedded fragment of target-language code similar to that found in the ANTLR system.

Furthermore, having operators as composeable, abstract functions allows for many interesting visualization techniques to be explored. Sharing operators (esp. legends) across visual panes provides a simple coordination mechanism. Augmenting Legends to highlight certain mappings is a way to provide brushing effects. Legends providing complete custom mappings would allow for novel visualization techniques to be explored at low overhead. Custom legends introduce a ‘tweak-able’ aspect. Good graphics kits allow arbitrary adjustments to the exact color, shape, scale or position of elements [27]. These could be encoded as custom legends. This may be very helpful for the presentation aspects of a visualization.

**Variables** represent the ability to directly manipulate control parameters on the visualization. Their target is the dynamic query or similar controls provided in many visualization toolkits [4,31,65].

The foreign operator and variable entities described above can be implemented in terms of the core Stencil model features (i.e. streams of tuples). However, they represent common cases that benefit from special representation in the Stencil language (and perhaps a distinct implementation). The foreign layer can be thought of as a pre-rendered canvas, that provides location names beyond raw coordinate values.

In addition to the above described Stencil language features, the model tuple and stream abstraction aid in integration as they are simple representations for moving data into and out of the visualization component. Working with the name-space feature of the Layer helps request particular data points.

The proposed system architecture also aids the integration of Stencil elements with their surrounding context. Benefits of this architecture are discussed in Section 4.5.

### 4.3.3 Declarative languages

Intuitively, declarative programming is “…programming by defining the what without explaining the how…” [1, 64]. It has alternatively been refereed to as a ‘denotive’ style [35] as a contrast to an ‘imperative’ style programming. In this way, programs written in a declarative language become specifications for programs that must be generated by the language interpreter.

We believe that a declarative language is the correct level of abstraction for visualization description since declarative languages generally relieve the programmer of low-level programming details, allowing high-level concepts be more directly expressed. In truth, all programming languages lie on a spectrum between declarative and imperative [64], as the distinction between how and what depends largely on the goal of the observer. In Stencil we seek to allow programmers to specify a visualization’s general structure without having to additionally specify the loops and branches or data containers to achieve that structure. In this way, the core language will be declarative. However, to efficiently state some operations or to integrate with other components we will allow bridges to non-declarative components. These bridges are encapsulated in ways to keep them supplemental to the core Stencil concepts but easily integrated when desired. The core concepts for these bridges and how they enable integration are described in the next section.
4.4 System Architecture

The overall application architecture is little changed by including a Stencil-based component (see Figure 6). From the application point of view, the Stencil generated component takes the place of the graphics library. The majority of the interaction occurs between the Stencil component and the application itself (and thus is mostly in terms of streams of tuples). The Stencil runtime and graphics library are still accessible to the host application, but interaction in many cases will not be required. These relationships are captured in the upper half of Figure 6.

An example workflow for creating an application using Stencil is pictured in the lower half of Figure 6. A Stencil specification is passed to a stencil generator. The generator creates source code for the visualization component. The generated source is presented to the compiler in the same way that standard program source code is. This overall structure follows closely on theoretical work described by [80] with algebraic language descriptions. In particular, it presents the DSL components of the system as a pre-processor in a separate compiler, this is often called an ‘external’
Figure 7: Application architecture when employing a Stencil interpreter. Grayed items are stencil specific.

DSL (contrasted with an ‘internal’ or ‘embedded’ DSL [1]). Furthermore, such systems have been successfully implemented in the JavaFX Script [68] and Adobe Source Libraries Adam and Eve projects [48] (these systems are discussed further in Section 6.9).

By constructing self-contained software components the system architecture makes modifying or changing Stencil-employing visualization applications simple when the visualization iterations have the same expected inputs (number of streams and tuple descriptions) and provide the same outputs. Changes to the underlying visualization framework, the processing performed in the Stencil-derived component, etc. are isolated from the rest of the program.

The relationship between the Stencil component, the Stencil runtime and the underlying graphics library differs substantially from the relationships presented in a typical compiler-compiler, and is more similar to the techniques used in SQL. The generated component wraps uses the functionality of the graphics library, but provides a significantly different set of semantics. The majority of the Stencil generated code is to ensure the Stencil semantics are correctly represented in the graphics library. The visualization model described above and its relationship to the model used in the graphics library will guide this generation process. The runtime component eases this
transition by providing certain invariants. Additionally, if features of the Stencil language are desired in the application (e.g. the ability to create new layers on the fly), a Stencil interpreter may be included in the runtime (an example of a pure interpreted architecture is given in figure 7). Interpreting and compiling Stencil descriptions are not mutually exclusive.

Many domain specific languages have benefited from meta-tools that allow the DSL source code to be generated indirectly (and only edited by experts in that DSL). Many query forms are used to indirectly generate SQL, early versions of PageMaker were a thin layer over the PostScript [54] and similar tools exist for HTML as well. Since the syntax of the Stencil language will be partially driven by representational concerns that are not shared when the description is generated through a tool, an alternative syntax may be constructed for tools. If constructed (as may be required for evaluation purposes [54,67]), this will likely be similar to the abstract syntax tree structure employed by the Stencil generator.

4.5 Toolchain Integration

Visualization applications must work with other applications in a larger toolchain to be effective [70]. Just as databases alone solve no problems, visualization without a surrounding context is nothing more than pretty pictures. Integrating with that context (not just other libraries in an application, but other applications) is a key component to the potential that Stencil holds. Several elements of the Stencil system make this process more tenable than many existing systems (still a manual process, but a more approachable one). These include the simple data interface, declarative definitions, a generative life cycle, and the facilities to integrate with existing code at an API level.

The data definition (described in Section 4.2) has two parts. First, Tuples are a simple format. As has been observed, many formats are either already tuples or can be converted to them in standard ways (e.g. XSLT to flatten XML trees). Streams release the interface from many timing-related constraints.

Employing a declarative, domain specific language that generates a self-contained widget is hoped to provide the ability to modify existing Stencil visualizations in a straightforward manner; essentially making Stencil components mutable (possibly even during runtime). The self-contained nature of generated components allows elements to be interchanged. The declarative language, as mentioned, will be a straightforward definition of high-level concepts. Making new representations available by changes to the stencil definition helps keep toolchains simple by allowing them to be mutated instead of extended. If changes can be made at runtime (where the stencil compiler is part of the host application), then such changes can be made interactively.

The foreign functions interface and the self-contained widget methodology form the basis for Stencil-derived components to use existing visualization code when desired, but keep it disentangled on a conceptual level. This is the basis of integrating a Stencil component with existing visualization code at the program level. Foreign layers provide similar functionality, but break the self-contained widget abstraction. These concessions are hoped to provided sufficient integration to keep toolchains from unnecessarily extending.

4.6 ThisStar Revisted

Precursors to many of the concepts described above can be seen in the ThisStar system [19,20]. However, changes to the underlying model and implementation lead to significant differences.
The data representation of ThisStar is nearly identical, except that in ThisStar, tuples were not strictly required. Instead, hash-tables with string keys could be used. This was for convenience of implementation, and will likely be dropped. Several changes are proposed for the graphical representation. The glyph group, explicit canvas and view are new. Grouping in ThisStar was only possible at the layer level. Canvas and view were implicit in the framework and could not be addressed. Including the canvas (hopefully) allows for navigation paradigms to be defined in a declarative style. Explicit views allow for multiple views on a single data-set. The Glyph concept has been refined, removing the explicit need for sizing and expanding the range of custom attributes. ThisStar only handled point implantations, so a single ‘size’ value was required to supplied the necessary scaling information. Removing the requirement of sizing allows glyphs of various proportions to be included (such as lines or ares) but also glyphs with no inherent concept of size (such as text or groups). Foreign operators were partially supported by ThisStar, but only through a Jython interpreter. Generalizing this concept is important to supporting integration with a variety of non-Stencil systems. Foreign layers and variables were not included in ThisStar but were often desired.

5 Research Plan

Our study of visualization specification is designed to identify a set of concepts that can be used to effectively define visualizations for a spectrum of data types and in a variety of circumstances. This concept identification will form the basis of a final language, and be driven by iterations on the language. The result will be a set of concepts that can be used define a broad set of visualizations and a concrete language that embodies those concepts. To further describe the visualization space, we will use the concepts to define a logical model for visualization. That model will be analyzed to describe the range of visualizations it captures, expanding the theoretical basis of visualization practice.

In the following sections, we provide an overview of the applications we will study to drive the development of Stencil and the accompanying model. In each section we will examine the importance of the application itself, how a Stencil-like tool would be well suited to that application and the unique demands it places on visualization systems that help encourage breadth in the eventually developed model.

5.1 Applications

Our study will focus on on a few visualization applications of interest. We believe the selected applications highlight important (and often problematic) parts of visualization practice. Furthermore, we feel that results achieved in these limited domains can safely be extended to visualization in general.

5.1.1 Interactive Data Exploration

One of the key modalities for our visualization system will be the ability to rapidly iterate over visualization designs. This is best captured in the idea of interactive data exploration. As such, we will develop an interpreter systems to support these behaviors, shortening the write → compile → run → modify cycle (modifications in this sense may not be bugs, but instead reflect changing
desires as the data is better understood). To investigate this modality, work will be done with individuals interested in general-purpose visualization applications.

5.1.2 Program 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 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 more likely to have ‘one-off’ needs for visualizations, where a temporary task would be benefited by a visual representation [51]. 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 [38]. 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 [34].

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 [42]. 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 insular design style of visualization tools is a barrier to exploration and adoption of visualization in programming, a field that prizes customizability and integration [51].

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 [33].

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 supports the second motivation, increasing visualization adoption, by putting visualization tools in the hands of programmers thereby showing them what is possible.

Beyond the technical challenges posed, software visualization provides a variety of opportunities in a well-defined domain. There are opportunities for network diagrams, timelines and
document visualizations. Entity relationships cover technical and social realms. There is a strong community of software visualization researchers that can be used to supply a variety of visualization ideas and source material (see the ACM Symposium of Software Visualization).

We believe that the Stencil system can effectively meet the challenges posed by software visualization. The minimal mental model and simple data model help reduce the cognitive load. Stencil’s mutability and ability to interact with existing software help the problem of integration. The fast-iteration style enabled by the declarative basis allow for information to be kept in context. Furthermore, Stencil fulfills the majority of requirements laid out by Meyer et. al. [42] for software visualization: Domain independent visualization engine; Composeable from smaller parts; Fine grain control and; Declarative descriptions. Stencil does not necessarily have low object-creation overhead, as this is an implementation that may vary by implementer. However, it does not implicitly have high object-creation overhead either and many of integration hooks (foreign layers and operators) are included, in part, to reduce the visualization’s impact on the overall system.

5.2 Theoretical Model Development

The preceding applications will drive concept discovery. Extending this to a model will be a central task and a major contribution. Existing theoretical models represent a low-level view on the pragmatics of visualization programming, focusing on data storage and control-flow issues (see the Models section for a discussion of existing visualization models). In this way existing models are similar to the physical models of database systems. They provide low-level guidance on how to store data and how to process incoming requests efficiently. This is a necessary component to understanding how to effectively provide visualization software, but presents a relatively poor model from several important standpoints. First, they are closely tied to particular visualization libraries. As such, moving between libraries often requires a complete model switch (and the incumbent data storage and control flow rewrites). This ties individuals tightly to their library selection, regardless of its current ability to meet their needs. Second, such libraries focus you on the issues of data storage and control flow instead of the actual goal of visual representation. In many ways they are too concerned with how to do display instead of what to display. The model we propose to develop needs to address these two issues. By focusing on the issues of what to display, we hope to make the insights gained from graphical semiology directly accessible. This focus on what issues should also allow us to abstract over a variety of pragmatics focused models, enabling more simple migration across frameworks. In effect, we want to develop a logical model for visualization to be used by application developers that is independent of (and agnostic towards) the physical models employed by the actual visualization frameworks.

5.3 Compile-time Optimization

One interesting opportunity that the proposed Stencil system offers is the ability to optimize a visualization component based on the functionality actually being used [3]. Traditional visualization libraries must provide full functionality in each component, even if that functionality is not used in a particular application. This has consequences for both the component’s responsiveness (as computation resources are spent on features unused by the application) and often its memory footprint (as auxiliary data structures accumulate). Compile-time systems, such as templates, are often used to ameliorate this problem. A compile-time solution presents itself in the proposed Stencil system as well. Using the context provided by the component definition, context sensitive
optimizations can be performed on the visualization component generated beyond those practical in a standard library system or in an interpreter environment. These optimizations likely to be identifiable based on the theoretical model and may provide insight into the unique role that domain-specific languages may be able to play in the software development environment. Furthermore, this may also illustrate the differences between the three types of library optimizations mentioned here: runtime, template-driven compile time and DSL supporting generative.

5.4 Evaluation

Evaluation of the Stencil System will be divided into two parts. First, an evaluation of the theoretical model will be undertaken to formally construct a correspondence between the Stencil model and existing visualization models. This evaluation will focus on just the model, not the pragmatic language features (like foreign operators or foreign layers), so it will not capture the full extent of the Stencil system. It is valuable to evaluate the theoretical properties of the conceptual model alone for two reasons. Primarily, it will illustrate where concessions were required in the language to integrate with other systems. Additionally, such an evaluation will highlight the visualization techniques that fall outside of the given metaphors, and thus help classify visualization techniques in a useful manner. Existing theoretical models [14,65] are driven by the pragmatics of data representation and control flow. They are analogous to the physical data models used by databases. To represent a higher level of abstraction, the theoretical model we seek to develop must encompass these models but be markedly different in abstraction level (just as the logical SQL model is to the underlying physical database models). Such a construction is non-trivial and the full scope of such a logical model is not currently known. Even a partial logical model will be an important step, but understanding the bounds of such model will be important for understanding future work.

The second evaluation component will look at the Stencil model as a whole. The over-arching goal of the system (which all of the goals listed in Section 4.1 support) is to improve the time-to-solution for visualizations. To evaluate the impact of the Stencil system on this front, we will pursue case-studies in visualization. We believe that software visualization is a fertile ground for such studies. Both the application-focused and compiler-focused discussions given above are in support of this evaluation path. The applications discussed are probable sources for data sets and visualization techniques to be used in evaluation. As noted, many software visualization projects are short-term needs, and thus will stress the prototyping support aspects of the Stencil system (rapid iteration, high-level conceptual model, etc.). The optimizations we believe will be possible with a compilation-based approach address the transition between prototyping activities and a final visualization implementation. Examining elements such as lines of code, memory footprint, execution speed and visualization modification difficulty are likely elements to measure in this second evaluation phase.

6 Related Work

6.1 Production and Interpretation Graphics Theory

Graphically represented data is interpreted according to principles of psychology rooted in evolution and culture. Understanding the relationship between data and effective representation is a task that has been undertaken by various psychologists and statisticians. Seminal works include
those by Bertin [6,7]. Recent works have also tried to blend visual and psychological concerns with production pragmatics [16,85]. In a more production/consumption focused vein are the works of Tufte [74–77] and William Cleveland [17,18]. The major points of these works are discussed below.

### 6.1.1 Semiology of Graphics

Bertin’s seminal works [6,7] take a de-constructive and taxonomic approach to statistical graphics. They ask the questions “What are the properties of data that are represented in graphics?” and “What are the properties of graphics that foster data representation?”

The first of Bertin’s questions lead to the categorization of data to be represented. This is significant as the pre-work of categorization helps ensure that the graphics produced do not misrepresent data. By identifying the significant characteristics of data, the significant characteristics of graphics are more easily identified. The basic character of data identified are:

**Number of components** shows how many values there are per observation. This dictates the number of visual variable that will be required to represent all of the data or how many n-variable graphics will be required if representing all of the data in one graphic is impractical. It is assumed by Bertin that only values of significance to the task at hand will be considered at this stage, so prior analysis may reduce the number of observed characteristics to a significant set, however many exploratory operations do not enjoy this luxury.

**Length** of a component describes how many valid values there are to a particular observed component. This may be two (binary variable), any specific finite value or infinite (as is common for real-valued measurements of continuous phenomena).

**Level** indicates the organization of values that a component may take. In its most basic form, levels are a superset statistical categories of data. The levels specifically addressed by Bertin are Qualitative/Nominal, Ordered (including rankings, [7] asserts that true ordered components must have equidistant values) and Quantitative (including ratios, quantitative components foster variable distance and may include units). Levels form a hierarchy, so representation techniques for qualitative components may be used with quantitative components, but not *vice-versa*. Some representation techniques are level agnostic.

The second core Bertin question, “What are the properties of graphics that allow data representation,” leads to a practical graphic vocabulary that is used to describe how data is being represented. This vocabulary is then used in an assessment of techniques to address the second question. He develops three categories of graphic variables: Imposition (the type of coordinates used), Implantation (the type of marks used) and Visual Variables (the properties of marks recorded by the eye).

The first concept is the Imposition, or the spacial arrangement style used. The basic impositions are:

**Arrangement** provides elements placed on a plane with no significance to their position or relationship. This is the random layout encountered in graph layouts or groupings where proximity matters but displacement from a reference point does not.

**Rectilinear** places items along a single linear dimension. A single stacked bar chart employs such an imposition.
Circular imposition uses a radial layout instead of a linear layout. Circular layouts typically place significance on the radial dimension (but they may use the distance from center instead). The variation in the non-significant dimension remains an arrangement. Pie charts employ a circular imposition with significance in the radial dimension. The displacement from center of a slice often used only to emphasize particular points.

Orthogonal imposition uses two (or three, but Bertin generally assumes only two dimensions will be used [6]) dimensions placed at right angles. This forms the basis for a cartesian coordinate system if both dimensions are continuous, but generalized systems may employ discrete dimensions (such as months of the year).

Polar represents a two-dimension imposition with a relationship to circular analogous to the relationship between rectilinear and orthogonal.

The graphic vocabulary of Bertin continues with the a classification of marks. As each mark represents a data point in a space, the mark type is referred to as the implantation in the selected imposition. The three implantation classes are point, line and area (though area becomes volume if a 3D space is used, again Bertin generally assumes a 2D graphical representation). Point implantations use glyphs to represent a location. Though the glyph technically has an area, the extent of the area over the plain is not intended to express an extent to the data. Examples include a scatter plots and the locations of cities on a world map. The extent of a point does not try to convey the physical extent of the city, the point only tries to communicate location and must have some area to succeed. Generally speaking, if the map were zoomed, the points would not proportionally increase in size. Line implantations are used to describe a connection between two points. The connection may or may not include the span between them. Roads are an example of a line implantation that includes the space between points in the relationship but flight schedule maps do not. Variations in size, texture, etc. follow similar rules to point implantation. Area implantation is intended to convey an extent in two dimensions. Stacked line charts are a statical example and country maps are a geographic example. When zooming in on an area implantation, the shape changes proportionally.

Visual variables describe how a mark actually appears. There are two sub-categories, location and retinal variables. Location is the physical position of the mark on the represented logical space. Imposition dictates the relationship between position and meaning, but any mark will have a location that can be expressed relative to the page boundaries as well. Location is significant for two reasons. First, it is the only visual variable that be treated as continuous. All other visual variables have a limited number of values that can be expressed, forcing binning of data. Second, it is homogenous, so no values are universally preferred.

The second category of visual variables is retinal variables, so called as they are recorded by the retina without respect to the physical location. Retinal variables are shape, orientation, color, texture, value and size. Each of these variables has various properties depending up on the implantation being used. These properties (associativity, selectivity, ordering and quantization) correspond to data properties and guide proper representation of data based on the task at hand. For example, the size of a glyph allows inequalities to be identified quickly (selectivity), allows ‘greater than’ and ‘less than’ comparison (ordered), and estimates of the ratios between values (quantization). It does not effectively create groups across variation (associativity), but shape variations do.

The properties of data described (number, length and level) are used to guide the graphical properties (imposition, implantation and retinal variables) used in a specific representation. For
example, if a continuous, quantitative variable is present, it is preferable to use an axis to represent it. By contrast a nominal variable of secondary importance would be more likely represented as an associative retinal variable to allow more significant graphical features to dominate. The categorizations presented above are not sufficient to automate graphical representation in all cases but they do construct a framework for discussion and foundations for objective evaluation of any particular representation. The utility of various visual variables for displaying various types of data has been re-examined on several occasions [39, 50] but these are extensions and refinements of Bertin’s work, not departures from it.


Constructing an effective software library requires constructing an implicit taxonomy of the problem space. Data structures and methods embody a problem representation and an organization space for solutions. Leland Wilkinson extended his work on statistical graphics software into a language for describing the construction of a graphic from base tables. His work is presented in [85]. Wilkinson’s work presents a more pragmatic view than those of Bertin, as the grammar presented is tied directly to a particular object-oriented graphics library. However, the issues addressed in that library are shared by all statistical graphics systems. Understanding the components of the over-arching grammar illustrates a distinct view on the problem of understanding graphics that extends the work of Bertin.

The primary contribution of [85] as an analysis of the graphical elements common to all data graphics, with an eye towards generality and the relationships between them. This produces a list of graphical elements and their respective properties. Wilkinson identifies the following:

Scales captures the basic ideas of data Level from Bertin, but expands to address various pragmatics of internal representation and composition. Wilkinson’s grammar-based description begins to address issues such as the results of values calculated from variables of differing Levels, transformations of variables (and their implications) and presents a unified way to handle unit carrying and unit-less values. The theory of scales introduces two additional Bertin-like levels Order and Measure. Handling of arbitrary units (and their composition) is deeply explored.

Coordinate spaces are also expanded on. Bertin makes the point that many circular and rectangular graphs are simply transformations on the same linear graphs. Wilkinson extends this to arbitrary transformations. Affine, projective and conformal transformations are all discussed. Their relation to each other and to the representation of various types of data (abstract, geographic and illustrative) are all examined.

Aesthetic functions move the abstract data to the physical senses. They are the step from an internal representation to a visible representation. These functions need to account for not only the data characteristics, but also the psychological (such as customary representation or cultural meanings) and physical (such as color blindness and the unequal intensity of different colors) characteristics of the observer. This type of function is the major focus of the psychology based visualization research including [82] and to a lesser degree [52].

Facets are the formalization of multiple views on the same data. The central concept is that high-dimensional data may be better served by multiple 2-dimension displays than an attempt
to capture all of the data in a single image. This problem is the origin of brushing and linking [11] and the focus of many papers in its own right [38, 44, 45]. This also captures the essence of Tufte’s small-multiples [75], though Tufte expands the use beyond viewing multiple dimensions.

**Guides** require care in placement and in their data density. Poorly placed guides can be confusing at best and misleading at worst. Overly dense guides become illegible. The interaction between the data, message and the guides is complex and requires a separate, but related, consideration from the data itself.

Since Wilkinson is developing a grammar for graphics construction, he also includes notes about generalized algebras. Though Wilkinson focuses on statistical graphics of static data, he also includes chapters targeting time and uncertainty. These are particular types of data that have special graphical considerations, and represent potential extensions to Bertin’s notion of level.

The majority of language description found in [85] is targeted at the graphics library described above. The concepts included in the library have been embodied in several iterations of domain-specific languages. These languages include ViZml, Graphics Processing Language [86] and nViZn [87]. These additional languages are closely related to the core grammar described by Wilkinson and are tied to their particular package. As language specification, they are instructive as they cover an analogous space to that of Stencil (but the model for Stencil extends beyond a particular library’s implementation). In addition to the items described by the core grammar, these implemented grammar-driven systems must more precisely describe the incoming data format, default values employed and the interfaces between the data, processing and aesthetic concerns. These pragmatic issues are of importance in any data-driven graphics system, and Wilkinson’s work lays a foundation for generalization. It is important to note that several of the languages described have found use in the internal structures of the SPSS statistical package, validating the practicality of a language-based approach for tool creation.

### 6.1.3 William Cleveland

William Cleveland’s work on statistical graphics follows up on that of John Tukey [78], exploring graphical representations of data and examining what specific methods offer in various situations to enhance the process of exploratory data analysis [17, 18]. Cleveland goes farther than Tukey in examining diagnostic plots as well as exploratory plots. Cleveland also makes explicit use of computation power, where Tukey focused on pen-and-paper techniques.

A fundamental aspect of Cleveland’s treatment of data graphics is the essential contrast between *graphing* and *fitting*. Graphing is displaying data directly or through summarization techniques. By contrast, fitting is displaying data generated by a mathematical model. Graphing is closely tied to the actual observations, the underlying data is the standard by which any graphing is to be judged. However, fitting approaches the capture of an underlying phenomenon, giving a characterization of data that can be compared to other data sets (via their respective models). These two aspects combine in effective data graphics, but require separate supports. The Stencil system directly addresses the needs of graphing through the mapping operators. The integration tools provided are, in part, to allow links to the processing required to assess fitting.

Cleveland’s further descriptions for data graphics are tied to task and data characteristics. His data characteristics closely follow those outlined by Bertin, but Cleveland explicitly discusses variables that can be interpreted at more than one *level* in Bertin’s hierarchy.
6.1.4 Edward Tufte

Where Bertin worked on a descriptive system, Tufte’s works [74–77] focus more on proscription and prescription by analysis of particularly good (or bad) examples. Bertin’s work is informed by psychology, where Tufte operates from a graphic design background. As such, they address much different issues. Bertin looks at a need to build a formal system of decomposition. Tufte’s instead looks to the questions of “What are graphic solutions that have worked?” and “What graphical properties have contributed the success or failure of a system?” Since Tufte’s work is not intended to be a formal whole, the descriptions are all written by analogy. Major themes include:

1. Eliminate non-data ‘ink’. If a mark exists on the page, it should be for a communication reason. This is the essence of “Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away” [23] applied to data graphics. Reduce the amount presented to just what is required to communicate.

2. Proximity manipulation as techniques for organizing data. This discussion extends the idea of imposition to relationships between coordinate spaces. Layering superimposes coordinate spaces, re-enforcing relationships that can be easily co-located. Separation is its perceptual inverse, distancing coordinate spaces that were previously spatially collocated, but not necessarily its cognitive inverse as such separations may help highlight relationships hidden in a superimposition. The semantics of these operations are modifications of those found in impositions, but represent significant extensions to the more uniform spaces presented by Bertin.

3. Don’t lie. Graphical representation provide a powerful tool to building both a general and detailed understanding of data. However, they also provide many opportunities to misrepresent. Much of Bertin’s work is targeted at how to avoid common forms of misrepresentation, and Tufte provides detailed analysis such lies that have been ‘told’ in particular situations.

4. Provide reference points. All statistics need context, provided by a comparison to other things. Graphical representations need to explicitly provide those comparison points. The types of references also guide the comprehension of the imposition, implantation and visual variables employed. Bertin refers to these roles, but Tufte develops guidelines for implementation of things like legends and axial labeling. His treatment is less formal than Wilkinson’s, but more broadly applicable.

A recurring theme of Tufte is enabling comparison. Graphics intended for communication (as opposed to decoration) have the necessity of context both internally and externally (also noted by Bertin). Internal context is provided by the data scales and labeling of a single graphic. This provides a means for identifying the degree and direction of variations within a set. Providing such context is the root of the argument for the introduction of text as a first-class graphical element and the representation changes prescribed for common items such as scatter-plot scales [74]. Providing external contrasts is primarily achieved by layout, it is the core of the ‘small multiple’ layout [75]. Positioning graphics in a way to compare them is a technique required of powerful exploratory tools.

Tufte’s work provides a gallery of good and bad examples of graphic communication. They include detailed discussion of the elements that help and hinder respectively. As a series, they
provide a good basis for a gestalt understanding of graphic communication. They also provide a set of goals that data graphics should strive to achieve, and that a data graphics production tool should therefore strive to enable. These include integration of text and graphic, composition of multiple graphics, graphic simplification, layering, and designer discretion to balance issues a per-instance basis.

6.1.5 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 with conditionals that are reference an external environment [24]. This formalized system for interaction with an environment shows that declarative, image-generating languages can successfully take into account external conditions. However, this interaction is limited in two ways. First it is strictly one-way, data comes in from the environment but the only communication back to the environment is the generated image. From a programatic stand-point the picture grammar is a write-only data structure. Second, the data read from the environment is usually assumed to be constant during a run (e.g. a single assignment model). This fits the model of command-line arguments, but is not sufficient for dynamic environments.

ContextFree is an example of a picture-grammar system. Its comprised of a 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. Each graphic grammar is created from a series of recursively-related rules. Rules may contain terminals or non-terminals arbitrarily. Each rule may also be annotated with a weight indicating a selection preference between rules with the same name. Rules are applied recursively until a graphic primitive is reached, then that primitive is displayed [21]. Context Free does not consume input; it simply iterates. A ContextFree program terminates when all rules paths have been fully expanded to terminals or a maximal stack-depth has been reached. ContextFree illustrates some of the pragmatics of a declarative, rule-driven system for directly specifying a visualization.

6.2 Visual Description Languages

Research into languages for describing pictures has been pursued previously, but with different goals from Stencil. Mackinlay [39] developed ‘A Presentation Tool’ (APT) based on a language that closely followed Bertin’s data classification system. APT was designed to take relational information and an APT description as input. It would then prepare a graphic according to Mackinlay’s interpretation of Bertin’s rules. This required Mackinlay to prioritize different presentation styles over others, beyond the prescription of Bertin. APT focused on automatic graphic creation, removing many of the graphical decisions from the end user’s hands. Stencil, by contrast, focuses on exposing the mapping mechanisms for manipulation by the user. However, Stencil does not completely reverse the approach of APT, as Bertin’s rules can form the basis of default processes when required. Mackinlay also required relational data; his work is a philosophical precursor to the database visualization work.

Roth et.al. [63] followed Mackinlay’s approach of automatic data mapping, but extended the data descriptors of Bertin. In particular, they introduced special categories for coordinates vs. quantities, domain membership, algebraic dependencies and made relational structures more explicit. Their work allowed for conventional representations to be identified and applied (particu-
larly through domain identifiers, provided the domain vocabulary is known by the user). Their work illustrated that Bertin’s classification system could be meaningfully extended, especially with respect to data decomposition. However, this system was still largely concerned with automatically creating charts based on rules, keeping the mapping operators separated from the user. This work was instantiated as the SAGE system. SAGE was later extended as Visage [62] where coordinating multiple views as also considered. In Visage, data classification was extended to explicitly handle certain types of meta-data separately. The focus remained on automatically creating visualizations based on data descriptions.

Casner [12] departed from Mackinlay’s approach of data descriptions and developed languages to prepare graphics based on tasks rather than data characteristics. Casner described the graphical production as a sequence of abstract actions. The key insight is that each graphical operation (zoom, highlight, rotate, etc.) if performed by a user corresponds to a step towards a solution. As such Casner’s descriptive language abstracts graphical operations to goal-seeking tasks. This work has been recently revisited by Yi [88]. This style of decomposition illustrates the types of compound operations people perform. As such, it is a rich source for abstractions for Stencil. Casner focused on interacting with a particular system while Yi presents a taxonomy for discussion purposes and is not intended as an implementable software system.

In a departure from prior strictly “classification” based methods of visualization description, Reichenberger et.al. [57] use classification only in the first step. The remaining phases are encoding of entities and layout of resulting glyphs. Similar to Casner’s concepts, data relations are matches with logical equivalents in the encoding step. The sum of the relations encountered is fed as input to a procedural layout engine. A language for describing data as a series of binary relations is used to describe data, with special relations to guide encoding and layout. This makes the resulting AVE system more specification-like than prior systems, but the emphasis is still on data description and algorithmic determination of the eventual representation.

6.3 Visualization Theory

The data flow and data state visualization models were discussed earlier (see Section 3). These two are the main conceptual models employed in visualization library development, but are not the only ones. We will examine an additional model for visualization libraries and some general graphics models here.

A partial model for graphics creation is provided in the ILog Creative Discovery application. This provides a means of generating graphics for any single data table that can be constructed in polynomial time relative to the number of elements in the data set. The details given in [4] are insufficient to explore the full breadth of this system, but it shows a unique conceptualization of the graphics creation problem. It also shows that comprehensiveness may not always be the only effective target for a model. Its grammatical basis is also of interest as many of the ideas presented can be mapped directly to Bertin’s descriptive framework. Their model is insufficient to capture the full extent of the proposed Stencil system as it requires all data to be stored in tables of known length prior to visualization.

Graphics systems involving multiple inter-related elements have typically followed two organizational patterns: entity tables and scene graphs. Entity tables are similar in structure to the Stencil Layers discussed earlier, representing visual objects as arrays of attributes. Entity tables group all similar object together into a single table to achieve higher processing speeds. This is a
strictly pragmatic organization scheme. Stencil Layers are similar to entity tables, but also provide external semantic meaning.

In contrast to the pragmatic entity tables, scene-graph libraries provide a highly conceptual view of the object-management problem in visualization. Entities are arranged in trees. These trees are typically relatively deep. The components of a tree include concrete items that are displayed, but also abstract concepts such as cameras, layers and property pages (which set defaults for all of their children). The scene graph describes is traversed at render time to create the visualization. Stencil captures many of the ideas of the scene graph, but provides less abstract construction and grouping concepts.

6.4 Visualization for the Masses

At the IEEE Conference for Information Visualization 2007 (InfoVis 2007), a number of papers were presented that approach the idea of making visualization available to non-visualization experts. This took the form of the keynote address, panels and a session entitled ‘InfoVis for the Masses’. Most prominent was the Many Eyes project at IBM [81]. Many Eyes is a web-site that provides standard visualizations for single table and paragraph-style data. The major innovation is the collaboration (Blog-style) support, encouraging people to converse about their findings. It is in the same vein as [32]. This project is designed as a platform for novices to discuss their data, mediated by easily defined visualizations. The visualizations are pre-defined and limited in number, but have enticed many people into discussions about actual data.

Tableau software takes a different approach to the problem of helping non-experts create visualizations [40]. It represents a continuation of the work found in [39]. The focus is on automatically constructing the ‘best’ visualizations for real-world data based on minimal meta-data. Very much in line with the prior APT tool, there is an underlying grammar for representing data and matching it to visualizations. However, an unique extension is the introduction of grammatical permutations. Since ‘best’ is subjective, and often a choice between several good options, the Tableau system provides a way to explore other options outside of the defaults. The analyst composes data views in a drag-and-drop fashion. Defaults for display are selected by the system, conforming to APT-like rules. In fact, APT-like descriptions are constructed behind the scenes. Alternative representations can be easily viewed as rule-respect permutations to the grammatical structures are generated by Tableau, but following competing rule-sets. This is one approach to balancing the competing desires of users: Power and Ease. Intelligent defaults are programmed in, but alternatives are easily accessed.

These two projects represent the current direction of the field to include a more novice-centered approach to adoption. This will likely continue in parallel with the expert-driven adoption strategies, but they indicate an opportunity for tools to bridge the gap is being recognized.

6.5 Generative Programming

The process of using programs to create other programs is broadly referred to as Generative Programming. Since Stencil is using a specification to generate a program component, it falls under this umbrella. If Stencil is used as development tool to create program components, it is a compile time generator similar to compiler-compilers. If the Stencil generator is included in a host application, it behaves more like a ‘dynamic compilation’ or ‘Just-In-Time’ system [22].
6.5.1 Compiler-Compilers

Compiler creation tools (known as Compiler-Compilers) come in many varieties, but share some common features. All take a specification written in a high-level language, usually a variant of a context-free grammar. All produce a program that consumes a text stream and produces trees. These trees may be an intermediate stage, or the final output of the result program. Individual tools provide different context-free grammar extensions, tree manipulations and analysis tools. Compiler creation tools of interest to the Stencil system include:

**YACC and Lex** constitute the basic compiler-compiler architecture for UNIX systems. YACC grammars are used to generate parsers that generate trees. YACC programs are typically coupled with Lex programs, where Lex programs generate token streams from textual inputs. YACC and Lex each take as input high-level descriptions of the desired behavior and generate programs that implement that behavior according to mathematical models. This typically leads to faster components than hand-made counterparts, since the generated components tend to use programming models that are untenable for direct hand-made programming. This illustrates one of the major benefits of a DSL and compiler solution: the ability to use more complex efficient implementations in the general population. Of course, the YACC and Lex programs themselves and the program models they employ had to be understood by their programmers, so the price for the complexity is not zero, but it is only paid once (and typically somebody else).

**Another Tool for Language Recognition** (ANTLR) provides three tools in one. First, it provides a unified replacement for YACC and Lex, allowing the token structure and tree structure to be described in the same language. Second, it provides a tree-traversal system, controlled by a specification language similar to that used to describe the tokens and initial tree. ANTLR further generates the source code for program components in a number of languages (though ANTLR itself is written in Java). Supported languages in v2.7 included C, C++, C#, Java, Python, Tcl. AST and Tree Grammars 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 components written in 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) [49]. The Stencil 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 Stencil 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.

**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 builds a powerful tree manipulation language [9]. 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. Stencil differs in terms of target application domain, but maintains many philosophical links to Stratego.

6.5.2 Just-In-Time Compilers

The above systems imply that all processing work on Stencil components is done at compile time. What about run-time specification of visualization components? Such a system is useful in two circumstances. First, for annotating existing graphics in an ad-hoc, data-dependent way. Such an operation is often performed when a new phenomenon is encountered and patterns are sought in old data. Second, for exploring representations of new data sets. In this case, rapid exploration of visualization techniques is desirable. In both cases, having to restart the program to change the visualization is disruptive to the ideal workflow. Just-In-Time compilers present a solution.

A Just-In-Time compiler can be viewed as a compiler that executes during another program’s runtime. Depending on the usage of the program, compilation may or may not occur. Further, the compiler may re-compile parts of a program as the program usage pattern changes. In this case, embedding a Stencil compiler into a host application would allow new visualizations to be specified, compiled and loaded while the host application is running. The requirements of Just-In-Time compilation are different from those of pre-compilation, but many of the concepts are the same.

6.6 Visualization Systems

Many visualization frameworks and libraries have been developed over the past several years. The library-level concepts and implementation details provide a window into the practical aspects of visualization programming. They can show how others have conceived of the visualization process and how those concepts mapped to actual data structures and control flows.

6.7 DEVise

Conceptually, the DEVise framework holds the most in common with the Stencil language and model. DEVise is a C++ library and accompanying runtime environment [38]. The runtime provides support for customizing the visualizations created in the framework. The framework itself is designed to handle streaming data and employs a tuple-mapping metaphor. DEVise divides all data into two types of tuples, either text or graphic. Data are received from a stream as text tuples and stored in a database. Processing on the text tuples generates graphic tuples that are also stored in the database. DEVise programs specify the mapping between the text and graphic data [13]. Graphic data is associated with a DEVise view and rendered to the screen. Multiple views can be linked together via the ‘cursors’ and/or ‘links’ mechanisms [38].

Stencil shares the streams, tuples and mapping metaphors, but extends them farther than DEVise does. Particularly, the Stencil model uses the streams and tuples metaphors to handle user input and view coordination where DEVise relies on special mechanisms for each. Another significant issue is DEVise does not allow views to be composed from smaller parts in a straightforward
manner. In DEVise, the mapping between text and graphic tuples is essentially handled as a pre-
processing step. Database queries define the views. As such, there are no direct mechanisms for
building layer-like abstractions without explicitly including meta-data in the graphic data schema.
Stencil explicitly exposes this powerful organization metaphor without requiring the stencil au-
thor to design the meta-data mechanism.

The visual template system [38] shows the power of stateful, transportable, interactive objects.
Stencil definitions can be used in a similar fashion, but do not necessarily capture the underlying
data. This allows a view to be created on one data set, and then quickly applied to another. If an
interactive view needs to be shared with a collaborator, simply sharing the data and the stencil
definition will be sufficient. This type of asynchronous collaboration has been little studied, but
may provide some benefits [84].

A DEVise-like system, where visual and source data are stored in a database, is a model that
Stencil may use to implement layers. This model was adopted in DEVise because it was explicitly
designed to handle data sets that cannot fit in main memory (and may not fit in virtual memory).
It is a promising direction for handling very large data sets, but these techniques are not always
required. Using the declarative/generative model of Stencil, deciding to employ such a storage
model can be delayed until it is needed, and then implemented (provided such a Stencil layer
exists) by changing a configuration and re-generating the Stencil component. Such a change may
be very important to the success of a visualization application, but is far removed from the aspects
of visual representation. The ability to make such changes to underlying representations in a
straightforward manner is a major benefit of the proposed Stencil systems.

6.7.1 GUESS: The Graph Exploration System

GUESS is an exploratory data analysis and visualization tool for graphs and networks. The sys-
tem 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 man-
ner. An interactive interpreter binds the commands entered to objects being visualized for more
useful integration. GUESS offers a visualization front end that supports the export of static images
or movies [2].

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. Furthermore, node-link diagrams are not always
the most effective means of displaying network data. The Stencil 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 Stencil 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).
6.7.2 Prefuse

Prefuse is a visualization framework with eye towards generality and runtime configurability. This is done 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 [31].

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 constantly updating data. Stencil focuses on data streams, so full data re-iteration is not assumed to be possible. This implies different data structures and visualization opportunities.

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 Stencil 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 Stencil concept would allow the developer to specify data structures, but not require it. An iterator would be required to expose the data to a Stencil defined component as a stream, but the component would handle copying to internal structures itself.

Prefuse is tied to java. Stencil 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).

6.7.3 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 $write \rightarrow compile \rightarrow run \rightarrow modify$ cycle by integrating the compile and execute steps in the programming environment. It provides library-based tools for many common visualization tasks [26]. Furthermore, Processing presents a simplified version of the Java language, abstracting out many tasks not directly related to the visualization process (such as window management, threading and class definition). The net result of these abstractions is that the Processing compiler presents a procedural version of Java, rather than an object-oriented one. Visualization code is still specified in terms of for loops and classically-typed variables, but ancillary issues (like the class hierarchy) can be ignored if the programmer so desires.

Stencil seeks to improve the $write \rightarrow compile \rightarrow run \rightarrow modify$ cycle in a similar way to Processing, but takes the language modification 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 $write \rightarrow compile \rightarrow run \rightarrow modify$ 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 Stencil DSL maintains bridges to the target language, allowing arbitrary code execution, but this is not the main mode of interaction.

A second point of interest in Processing is its handling of mouse and keyboard input. There are two methods of handling input devices: events and special variables [56]. The events methods is
the same as standard Java. The special variables method automatically loads globally visible variables with the mouse position, button states and key press information in each animation frame. A similar set of variables holds the information for the prior frame. These special variables are similar to the ‘foreign’ variables discussed earlier and could be thought of as a stream of states that runs in lock-step with the animation frames. The special variables representation differs significantly from the stream model proposed since the special variables still represent a special case while the stream model for interaction makes the user input handling identical to any other input handling. Furthermore, the special variables of processing only handle standard input devices (one keyboard, one mouse) while the stream model of Stencil allows arbitrary input devices (multiple keyboards or unusual inputs) without modifying the abstraction. Such non-standard inputs would be unavailable in the Processing special variables model.

6.7.4 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 [44]. Where Snap-Together permits only on coordination, Stencil system tries to expand this flexibility to creation of the views as well.

6.7.5 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 [65].

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 Stencil 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 Stencil 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).

6.7.6 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 [41].
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. Stencil is intended to handle many different data types.

## 6.8 Domain-Specific Tools

### 6.8.1 Databases

Structured Query Language (SQL) is the default interaction mode for many database systems. Many database tools either generate or consume SQL queries to perform their work. Using a single data model (tables) and a few operators (filter, join and grouping), the power of a relational database is exposed in a straightforward, platform neutral way. Platform-specific functionality is often exposed as an extension to SQL. This provided three benefits to the database community. First, it abstracted database control from the data consumer. Database administrators or developers could focus on issues of physical representation and control follow, making changes if necessary, without impacting the consumer of the data stored. This reduced the cognitive load for people using a database to solve a problem. They only needed to understand the logical structure. Second, it provided a standard way to introduce non-standard functionality. Enhancements and changes appear as incremental updates to the end user. This natural migration path eases adoption of new (possible better) ideas. Finally, the introduction of SQL allowed the focus of database related conversations to focus on the data being stored instead of how the database was going to retain it. This focus put problem and solution statements into a user centered vocabulary instead of database-centered vocabulary.

SQL has not been a panacea in relational databases, but it has been a beneficial step. As the community of database users has expanded, the implicit technical prowess has decreased and new problems are being approached with databases. These new problems may or may not be well served by SQL, but it is unlikely that they would even be approached if SQL had not provided a first broadening step.

The success of SQL can be largely attributed to its ability in two factors. First, it handled a broad set of known problems. This claim is justified by the backing of SQL in relational algebra. Second, the clear extension paths for many known classes of solutions (stored procedures for summary and grouping, new selection keywords for data transformations, etc.) allowed it grow with the databases it supports. This extension includes some visualization research.

The field of database visualization focuses on establishing mappings between data stored in a database and a visual display. Groth proposes an architecture for rapidly preparing database tables for a visual representation [29]. 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 itself (called MQL) and the Plot and Filter modules from the support framework are of particular interest.

Stencil 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 [53]. It simplifies synthesizing multiple
data sources (a needed work for Visual Analytics [72]). It eliminates the need to have a database *per-se* by abstracting a database to a stream source.

The Stencil 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) [29]. 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 Stencil 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.

6.8.2 Software Visualization

Prior work has been done in programs for software visualization [10, 25, 37, 55, 59, 60]. Further, a few frameworks for software visualization in particular have been developed. Vizz3D is the visualization component of a reverse engineering framework described in [46]. 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 [44], 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. Stencil targets compile-time definition of visualizations to increase the flexibility in the types of displays that can be used, rather than runtime selection of mappings to pre-defined visualization views (see Just-In-Time compilation for a description of runtime vs. compile time generated Stencil components).

The coupling of a visualization module to a software analysis package (as the Vizz3D is to its analysis framework [46]) is also discussed in other software visualization systems. The Jove system [60] has a lower level of coupling between the software analysis and the visualization system as the Vizz3D system, but the provided visualization system architecture is configured in a similar manner.

The Pavane system for software visualization is similar to the Stencil system as it provides a declarative language for specifying the visualization [61]. However, a major part of the specification language is tied to the description of the program states that the programmer wants visualized. The specification of the visual representation is inherently tied to this model, and thus unsuitable for generalized visualization. Further, the Pavane does not provide bridges to host language, nor is it expected to integrate with a host application (the Pavane system is the application). The first issue limits the visualizations that can be produced to only those captured in the system (no efforts were taken to make the visualization language ‘complete’). The second issue makes Pavane an isolated tool, incurring all of the tool-chain issues described earlier.

More recently, the Mondrian system [42] has been developed to support the needs of software visualization in a declarative manner. Mondrian tries to address two obstacles identified by Reiss [58] that limit software visualization: (1) Difficult data formats and; (2) Special purpose tools. To address the second issue, Mondrain presents a general-purpose, declarative programming language based on Smalltalk. This permits arbitrary calculations with simple conceptual model. This model, however, is specific to the software visualization problem. Visualization specific issues are handled by a library of objects. In regards to the first problem (difficult data formats), Mondrain takes an unusual approach. Instead of specify a data model, it assumes the model already exists
and provides interfaces for interacting with arbitrary models. Since the full range of Smalltalk is available, any model iteration strategy can be implemented. In this way, any data model that can be navigated may be used. Mondrain represents a uniquely structured approach to software visualization strongly informed by information visualization research.

6.9 User Interface Design

Information Visualization inherently has a large user interface element. Both representation and interaction are user-focused tasks. Recent work on declarative DSLs in User Interface Design show that the concepts underlying the Stencil system are viable in general. Two major projects are examined here: Adobe Source Libraries (ASL) and JavaFX Script from Sun.

6.9.1 Adobe Source Libraries

The Adobe Source Libraries project is comprised to two distinct but related sub-projects: Adam and Eve. These two projects carry a philosophy and style similar to Stencil, but with different focus. Combined, Adam and Eve provide the ability to specify the visual and logical structure of user interface components in a declarative fashion. Adam focuses on logic, providing a spreadsheet model with named cells and relationships declared between cells. Eve focuses on layout aspects, with means to tie into abstract data models conforming to certain interfaces (which Adam’s spreadsheet complies with). Adam and Eve each include an independent declarative domain-specific language (like Stencil) and a ‘solver’ that acts like an interpreter. The primary goals of the combined project is to decrease the effort and increase the reliability of the user interface creation problem. The secondary goals are to provide consistency across platforms and shifting some of the implementation tasks to the interface designer. The shared abstract data model enables them to be independently defined, but work together to implement a user interface [48]. This shared data model also provides a basis for reasoning about the capabilities and correctness of the items described in the respective DSLs. Anecdotal evidence suggests that Adam and Eve are being beneficially employed in commercially available Adobe products [47].

The goals of the Adam and Eve projects are similar to those of the Stencil project. Principally, to decreasing required effort and increasing reliability are almost exactly the same. The secondary goal of enabling a new group (designers for Adam and Eve, analysts for Stencil) to develop the actual software is also closely aligned. The architecture used in the Adam and Eve system is also related to that proposed for Stencil. Both use a declarative DSL to describe a component of software, but rely on a host program to provide an execution context [48]. Stencil has a more strict interface for the resulting component (streams of tuples) but it is designed to internally handle larger data loads than Adam and Eve (which primarily function as interfaces to other libraries).

The process of converting from an Adam or Eve description to an executable program is similar, but distinct to that of Stencil. Adam and Eve currently both execute in an interpreted manner on top of a state machine. This is similar to the interpreted option mentioned for Stencil. Integration with an existing code-base is mediated through a support code layer (similar to the Stencil runtime). It has been noted in the Adam and Eve literature that such an architecture permits visual editing tools and opportunities for multiple, concurrent syntaxes [48]. This is promising for making Stencil-based tools accessible to non-programmer audiences.
6.9.2 JavaFX Script

JavaFX is a family of products built on the Java Runtime Environment (JRE) but not necessarily leveraging the traditional Java development tools. JavaFX Script is a declarative, statically typed, scripting language targeted at user interface definition. JavaFX Script syntax and semantics are closely related to those of Java, but not identical. In particular, rules for evaluation order and the interpretation of encapsulation have been modified to match the mental model employed for UI creation and update [69]. In this way, JavaFX Script can be considered a DSL for interface definition. JavaFX Script retains structured means to interact with existing Java classes. From the Java point of view, the JavaFX Script object is just another Java object or class. From the JavaFX Script side, special syntax with the ‘operation’ statement is used to invoke a method on a standard Java class [68].

The Stencil system represents a more radical departure in terms of form, mental model and interaction model than the JavaFX Script system. However, the goals and system architecture are similar.

7 Schedule and Risk Assessment

The primary work for this thesis will be performed between February 2008 and December 2008, with the final thesis presentation targeted for March 2009. The general research schedule is shown in table 7. This is an appropriate schedule for the proposed work, but may need adjustment on an ongoing basis to ensure we maintain a high-level of quality in our study while providing results suitable for a PhD level dissertation.

<table>
<thead>
<tr>
<th>Project</th>
<th>Planned Dates</th>
<th>Risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThisStar Concept Application</td>
<td>June-August 2007</td>
<td>Complete</td>
</tr>
<tr>
<td>Initial Operator Definition</td>
<td>August-December 2007</td>
<td>Complete</td>
</tr>
<tr>
<td>Interpreter</td>
<td>Sept 2007-July 2008</td>
<td>Low</td>
</tr>
<tr>
<td>Software Visualization</td>
<td>Feb-Dec 2008</td>
<td>High</td>
</tr>
<tr>
<td>Theoretical Model Relationships</td>
<td>Jan 2008-March 2009</td>
<td>Medium</td>
</tr>
<tr>
<td>Compiler</td>
<td>Oct 2008-February 2009</td>
<td>High</td>
</tr>
<tr>
<td>Final Results</td>
<td>April 2009</td>
<td>Med</td>
</tr>
</tbody>
</table>

Table 1: Research Schedule and Risk Assessment

The initial interpreter is of lower risk than the compiler as it presents a simpler model than the compiler. The interpreter needs to abstract over the selected graphics libraries, acting as a demonstration of the generality of the system. The compiler, on the other hand, will need to encompass the ideas of a compile-time configured library (similar to a template system). This requires tighter integration with the graphics framework selected. Also, to investigate the performance potential, the compiler analysis will need to be more in-depth.

The correspondence of the Stencil model to other visualization models is central to the evaluation of the Stencil system (see Section 5.4 for details). Model relationships will be explored concurrently with the system development. Since the compiler and interpreter will likely be implemented with the help of some existing systems, some of the relationships will follow naturally.
from the implementation. However, a complete characterization of the relationship between the Stencil model and any existing visualization model will likely require more direct attention after the system is fully implemented. Such a formalization always requires both creativity and careful thought, and is thus estimated at a medium level of risk.

In support of the evaluation plans listed in Section 5.4, several options software visualization options are in development. The preferred method is to work in collaboration with software teams inside and outside of IU. Contacts have been made interested parties within the Open Systems Lab, at Lawrence Livermore National Labs and at Intel Corporation. Since this will be a real-world application, many surprises are expected. In working with outside entities, a MILC-like approach [67] will likely be employed. If such studies cannot be accomplished in a reasonably beneficial way for all parties, a large collection of open-source software has been acquired from Sourceforge.net. Analysis and visualization of this information will be pursued in place of (or in addition to) such collaboration. This is rated as a high risk as any collaboration with external parties incurs many unknowns.

To ensure the final results are available on schedule, incremental results will be continually documented in research papers.
References


