2013-01-01

A Declarative Domain Independent Approach for Querying and Generating Visualizations

Nicholas Ricky Del Rio
University of Texas at El Paso, ndel2@utep.edu

A DECLARATIVE DOMAIN INDEPENDENT APPROACH FOR QUERYING AND GENERATING VISUALIZATIONS

NICHOLAS DEL RIO

Department of Computer Science

APPROVED:

Paulo Pinheiro da Silva, Chair, Ph.D.

Vladik Kreinovich, Ph.D.

Rodrigo Romero, Ph.D.

Aaron Velasco, Ph.D.

Benjamin C. Flores, Ph.D.
Dean of the Graduate School

to my wife

Jerien E. Rausch

with love

A DECLARATIVE DOMAIN INDEPENDENT APPROACH FOR QUERYING AND GENERATING VISUALIZATIONS

by

NICHOLAS DEL RIO

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

May 2013

Preface

The term *visualization* can refer to (1) the process of generating graphical representations from non-graphical data or information, or (2) the graphical representations themselves. This dissertation opts for definition (2), and so all representations of data or information such as *isosurfaces* [63], *volumes* [21], and *Treemaps* [72] will be referred to as visualizations. To distinguish between definition (1), this dissertation will explicitly refer to the process of generating visualizations as *visualization generation* or the *visualization process*.

Abstract

Constructing visualizations using modular visualization environments (MVEs) requires knowledge about visualization transformation theory as well as implementation specific details about how to programmatically chain together relevant sets of modules into pipe and filter-like architectures known as *pipelines*. Constructing visualization pipelines introduces a number of challenges including: understanding how to transform raw data into forms that can be ingested by MVEs, identifying target modules that map data into graphical data, and understanding how to further transform resulting graphical data into forms that can be presented. Users typically must immerse themselves in a deluge of documentation and usage examples before becoming proficient in the aforementioned tasks and therefore able to generate suitable visualizations required for analysis.

This dissertation presents a visualization query language that was designed with the goal of hiding the associated complexities of visualization pipeline construction by abstracting away implementation specific details. By posing visualization queries, users can declaratively specify *what* visualizations they want generated from their data and quickly explore a variety of results that may only be generated using complex combinations of modules possibly supported across different MVEs. This dissertation explores the query answering facilities associated with visualization queries as well as presenting a pilot study which supports the claim that users both prefer and are more proficient at using queries than writing traditional pipeline-based code.

Table of Contents

Appendix

List of Tables

List of Figures

Chapter 1

Introduction

Visualization generation is a topic that encompasses a number of computer science related sub-fields including graphics, performance (e.g., parallel processing) and perhaps the most crucial field, human computer interaction. Visualizations convey *messages* that are designed for human consumption and so understanding human cognition, perception and semiology (i.e., the study of symbols) is fundamental when generating visualizations. Although a large number of visualization techniques have been proposed, studied and implemented in a large number of systems, approachable methods for specifying how data or information should be transformed into these visualizations remains an active area of study.

Chapter 1 presents the *visualization toolkit deluge* and explains why a variety of different visualizations may be needed for comprehensive data analysis and the cost associated with supporting such diversity. The chapter follows by introducing *visualization queries*, a novel approach for declaratively specifying the generation of visualizations in a manner that minimizes cognitive loads when working within the context of multiple toolkits. Posing queries requires less visualization-specific expertise than existing specification efforts supported by many toolkits, which are described in Chapter 2. Chapter 3 presents a survey of related visualization models that aim at structuring the conceptual space of visualization generation into forms more comprehensible for users. Chapter 4 presents a new visualization model that can be leveraged to answer visualization queries by way of a query answering system described in Chapter 5. Chapter 6 presents a user study that aims to compare the readability and writability of visualization queries with the popular paradigm of specifying visualization generation through pipeline orchestration. Finally, the dissertation concludes with some future work and closing remarks in Chapter 7.

1.1 Problem: The Toolkit Deluge

There are numerous software packages, or *toolkits* that support the generation of visualizations. This section aims to provide motivation for why there exists such a large number of toolkits and also presents a usage scenario that highlights the benefit of having a wide varying selection.

1.1.1 Deluge Source

The visualization field has historically been divided into two complementary sub-fields: information visualization and scientific visualization. The distinguishing characteristics between these sub-fields is wide ranging. Some authors have proposed defining characteristics that support the dichotomy, including:

- whether the spatialization of the data is inherent (scientific visualization) or chosen by users (information visualization)
- whether the visualization is used as a component in the scientific method (scientific visualization) or non-scientific purposes (information visualization) [10]
- whether the underlying data is measurement based (scientific visualization) or abstract, such as stock exchange rates (information visualization) [10]

Other authors have argued that this categorization is not in-fact a true dichotomy because many visualizations cannot be easily classified within the binary space. For example, Tory and Möller argue that air traffic control data is always accompanied by a spatial component, however the resulting visualizations are not used for scientific analysis and thus can be considered information visualization [76]. Additionally, GIS data is always associated with geographical coordinates, and as such, the corresponding map visualizations can be considered scientific, although their intended use can fall outside the realm of science, for example: finding directions to the hospital. To compensate, Tory and Tory and Möller

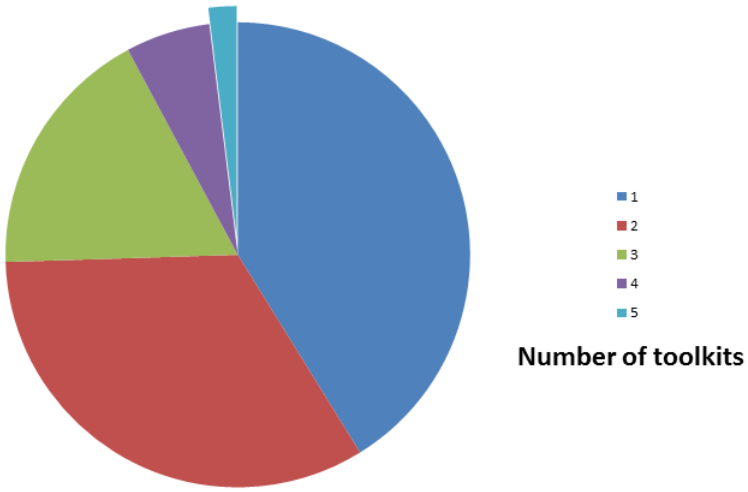


Figure 1.1: Percentage of Users who Employ more than a Single Toolkit

have proposed alternative categorizations for visualizations that are based on *perspectives*, for example:

- the user perception of the data
- the perception of the data from an algorithmic standpoint

Regardless of what categorization is most accepted, for twenty five years toolkit developers have published visualization software that adheres to the sci-vis and info-vis dichotomy or provides some hybrid capabilities of both [76]. The result is a large set of visualization toolkits that can be challenging and time consuming to filter through when identifying a single toolkit for use. Additionally, it is common that a single toolkit may not provide the full set of visualization generation capabilities needed for analysis and so users are left with the laborious task of researching different toolkits and identifying the right subset of tools that serve their specific needs. For example, during an American Geophysical (AGU) meeting, a small survey was conducted in which the variance of employed visualization toolkits was noted during a poster session. The findings are presented in Figure 1.1 and show that although the majority of users employ only a single toolkit, many users employ more than one.

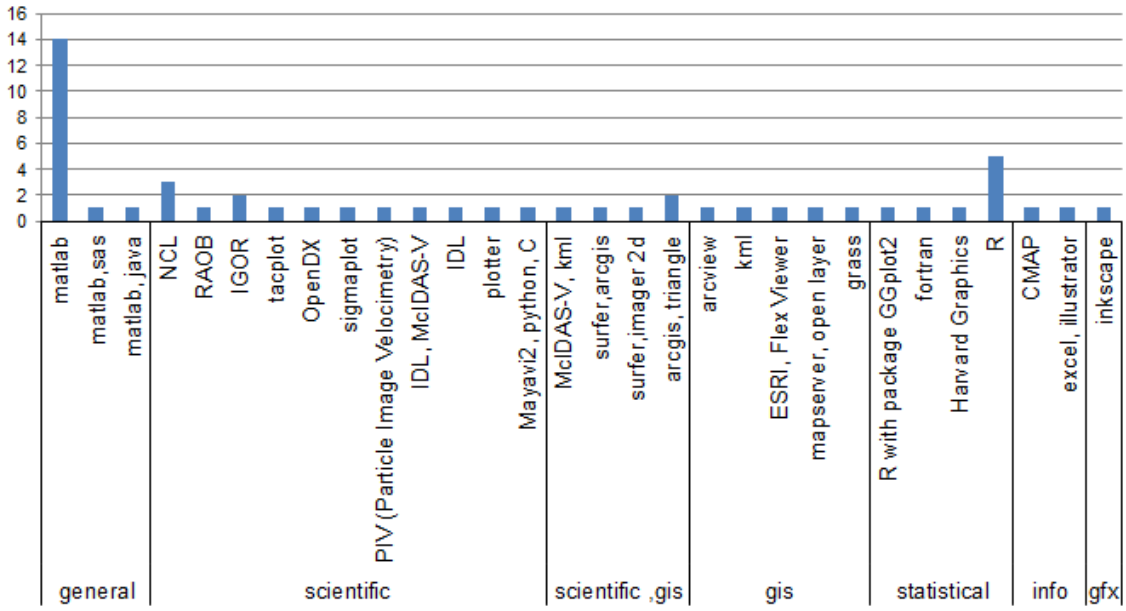


Figure 1.2: Variety of Employed Toolkits (gpx = graphics)

1.1.2 Deluge Benefits

In addition to noting the number of different toolkits each poster presenter employed, our study also examined the diversity of employed toolkits. Figure 1.2 presents the different toolkits used and the number of users who employed them. Matlab [32] and R [40] in particular were the most widely used, although the most compelling aspect of the graph is the number of different toolkits used considering our relatively small sample space. Additionally, the kinds of toolkits employed ranged from scientific systems to general graphics software. A possible reason for this diversity is that each visualization approach provides a different insight or perspective into the underlying data driving the visuals. Another reason could be based on the programming language familiarity of the user regarding a specific toolkit. In the latter case, users may be compromising higher quality visuals for ease of use. To illustrate the diversity of toolkit visualizations, consider Figure 1.3 that shows a single gravity data being visualized using four different techniques:

- Generic Mapping Tools (GMT) [80] raster map based on a nearneighbor algorithm

- GMT raster map based on a surface algorithm
- Visualization Toolkit (VTK) [70] isosurfaces
- VTK three-dimensional point plot

While using GMT, the different gridding algorithms employed (e.g., nearneighbor or surface) each yielded different two-dimensional visualizations. The nearneighbor version resulted in missing pixel data within the visualization, which may have resulted from incomplete data, a software glitch, or perhaps an inappropriate configuration of a gridding parameter with respect to the density distribution of the underlying data. From this visualization alone, the source of the missing pixels is not completely evident without trial-and-error reconfiguration of parameters and gridding techniques. The visualizations generated from VTK, on the other hand, represent three-dimensional spaces and provide a different perspective of the data than GMT, for example, a three-dimensional point distribution plot. Users would be interested in noting that the VTK point distribution plot also exhibits a sub-region of missing imagery, eluding to the greater possibility that the underlying gravity data is missing a substantial number of data points. This cross-analysis was directly supported by the visualization diversity resulting from the use of multiple toolkits.

1.1.3 Deluge Challenges

The benefit of using multiple visualizations supported by multiple toolkits comes with a price. The price is the overhead necessary to become proficient using each toolkit, which each can vary dramatically in terms of:

- interfacing language
- supported functionality of modules
- supported visualizations
- supported data model (e.g., two or three-dimensional)
- performance

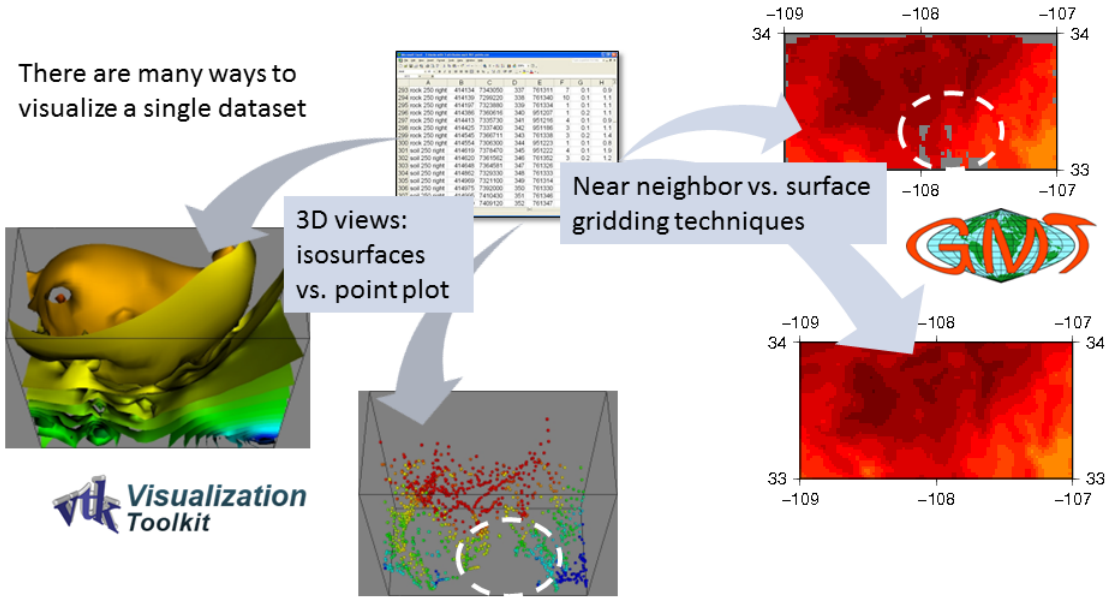


Figure 1.3: Analyzing Data Using Two Different Toolkits

- portability
- ease-of-use (e.g., graphical user interface support or textual programming based)

In order to support scenarios where multiple toolkits are required, users may have to understand each toolkit in terms of the aforementioned properties, which may be challenging if users are not visualization experts or well versed in programming. Furthermore, it may be required that users *intermix* different functionalities supported across different toolkits and form hybrid applications. For example, some toolkit A may provide a high-performance gridding routine while a different toolkit B supports a high quality volume rendering technique that is superior when compared to the volume rendering supported by toolkit A. In these cases, users may want to construct visualizations that leverage routines from the different toolkits A and B, possibly leading to visualization application that mitigates disparities between specifications and data models.

Chapter 2 highlights the above disparities in the context of generating two-dimensional contour maps using GMT and VTK. Chapter 3 notes that despite the differences between

the two toolkits, both share a conceptual structure that is capitalized upon and used to drive the proposed visualization model described in Chapter 4. The query answering facilities associated with visualization queries are based on our visualization model and aim to provide users with an abstract view of the different toolkits that is not confounded with the scientific/information visualization dichotomy. In fact, visualization queries are agnostic to any single type of visualization and can address a high variety, depending upon the comprehensiveness of the answering system described in Chapter 5. Due to the high level nature of queries, users only have to know about certain properties regarding their data and possibly information about the visual form they want their data projected in, although not necessary.

1.2 Querying for Visualizations

Users generally have two options when deciding upon how to visualize data or information: find existing *turn-key* applications that can generate the required visualization or construct custom visualization applications that suit their requirements. Although there are many visualization applications available for download or usage on the Web, most are configured to generate specific kinds of visualizations from specific kinds of data. Users must be able to identify an application that both generates the required visualizations *and* ingests their data in the form it resides. If both conditions cannot be met, then users can develop their own custom visualization applications that satisfy both the visualization and data requirements. Our work with visualization queries, described further below, is aimed at addressing the issues associated with the second option: developing custom applications.

Fortunately, users that choose to develop their own applications may not have to become experts in computer graphics, performance, and semiotics. Users can develop visualization applications from *visualization toolkits*, which provide sets of modules that abstract the aforementioned visualization concerns. Modular visualization environments (MVE) [8] are one such type of toolkit that support the development of visualization applications, by

allowing users to assemble modules into pipe and filter [58] based architectures known as *pipelines*. Although visualization applications can reside in forms other than pipelines described in Chapter 2, this work is primary focused on the MVE variety. The modules that are used to construct pipelines are known as *operators* and toolkits generally only provide sets of operators related to some specific domain, for example scientific or information based visualizations.

Despite most toolkits' ability to abstract away low level graphical and performance based details, the burden of both selecting appropriate toolkits and constructing pipelines that generate required visualizations falls upon users. Although users may be able to read documentation and make well-informed assessments regarding the relevancy of a given toolkit, provided their needs, constructing pipelines requires in-depth knowledge about format conversions, data type transformations, filtering, visualization mapping and viewing [16, 33]. The overhead of becoming well versed with these concerns can drive users to seek help from visualization experts or resort to adapting existing pipeline code found on the Web to suit their specific needs, which often results in time consuming trail-and-error programming [79].

To mitigate the challenges associated with constructing visualization pipelines, this dissertation proposes an additional semantic layer that envelopes operators and abstracts away the pipeline construction process from users as shown in Figure 1.4. Rather than specifying pipeline sequences, users can interact with toolkits through the semantic layer by issuing visualization queries and delegate the task of pipeline of construction (i.e., program synthesis [19]), to the semantic layer. Figure 1.4, also presents the lower level layers of the visualization application stack, which include parallel libraries such as Message Passing Interface (MPI) for distributed architectures and OpenMP for shared memory systems. Graphics libraries such as OpenGL [81] and data streaming facilities can interact with the operating system through the invocation of performance based libraries. Visualization Toolkits, described further in Chapter 2, abstract the graphics and data streaming layers though a set of operators from which visualization pipelines can be composed.

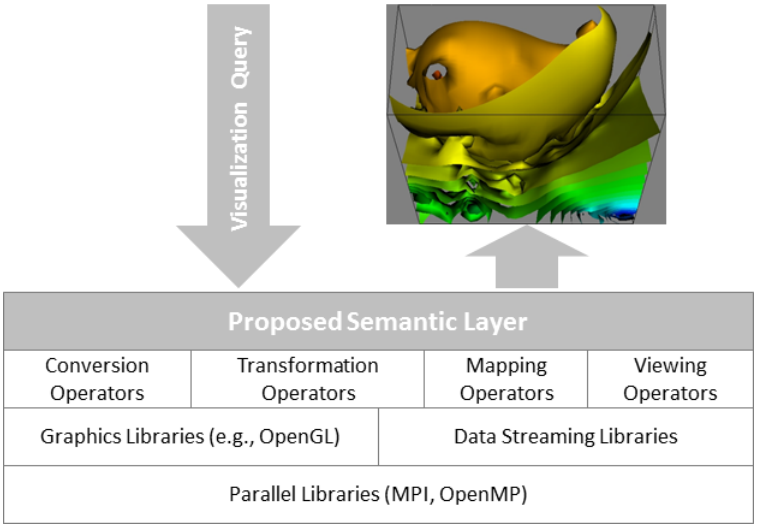


Figure 1.4: Performance, graphics, and toolkit layers are all abstracted away by the proposed semantic layer

Since the aim with visualization queries is to minimize the cognitive load associated with generating visualizations, queries are structured in such a way that users need only to be aware of:

1. **Source**: type and format encoding of their data to be visualized
2. **Target**: requested visualization abstraction and viewer that should present the abstraction

Therefore, writing visualization queries requires that users need only know a *source* (i.e., the input data characteristics) and a *target* (i.e., the required visualization and the viewer responsible for presenting the visual). The work with queries is motivated by observations that many popular visualization toolkits require that users know both source and target information *plus* sequences of operators that implement the required transformation from source to target. Identifying these relevant operator transformation sequences requires in depth knowledge about some toolkit, or set of toolkits in the case when a single toolkit does not support all required transformations. As more toolkits are ported to the Web, such as ParaViewWeb [45] and Data Driven Documents (D3) [3], users will have an even

QUERY	→	{PREFIX} SOURCE VISUAL DATA {PARAM}	Overall Query Structure
PREFIX	→	"PREFIX" {alphanumeric-character} namespace	Prefix Declarations
SOURCE	→	"VISUALIZE" url	Input Dataset
VISUAL	→	"AS" (visualization-abstraction | *) "IN" viewer	Requested Visualization and ViewerSet
DATA	→	"WHERE TYPE =" type "AND FORMAT =" format	Format and Type
PARAM	→	"AND" param "=" value	Parameter Bindings

Figure 1.5: Query Grammar: Extended Backus-Naur Form

greater opportunity to re-purpose published operators into their own visualizations scenarios, in which case delegating the operator sequencing task to machines will become even more desired due to incomplete or inconsistent documentation practices among toolkits. Our visualization queries aim to compensate for the steep learning curve imposed by such scenarios.

In comparison to visualization queries, relational database users have long interacted with data stores by issuing queries against data management systems that carry our lower level relational algebraic operations. For example, relational database users request for data, filtering and transformations by posing queries in Structured Query Language (SQL), a declarative language that is translated by data management systems into equivalent relational algebraic operations that compute the results [41]. Therefore, our proposed semantic layer can be viewed as a kind of management system for visualization where instead of composing relational algebraic query plans, the semantic layer composes visualization pipelines. To contrast, our notion of query and answering is less akin to the querying facilities in information retrieval that retrieve rather than compute or transform answers.

1.3 Visualization Query Grammar

The structure of visualization queries is defined by the following Extended BackusNaur Form (EBNF) specified in Figure 1.5:

1.3.1 SOURCE Rule

The SOURCE components of visualization queries are constructed from both the SOURCE and DATA rules. The SOURCE rule produces a clause that specifies the location (i.e. url) of the input data to be visualized. The location of the data can be specified using the (file://) or (http://) protocols to support scenarios where the data is either locally available or remotely stored on the Web. The DATA rule generates clauses that characterize the input data in terms of `Type` and `Format`. `Type` refers to specific programming language data types, for example `vtkPolyData`, and represents the structure of the data independent of how that structure is encoded in a file. For example, tagging some data as type `vtkPolyData` indicates that the data contains: `field data`, `vertices`, `lines`, and `strips`.

Format declarations specify how a particular type is encoded or serialized into a file, for example `vtkPolyData` can reside in the legacy serial text format or the more flexible eXtensible Markup Langauge (XML) [5] format [70]. In general, formats can enforce very specialized rules for encoding only a single kind of structure, such as ESRI Grid [83] that can only encode 2D Grids. Other more flexible formats allow encoding of arbitrary structures, such as the JavaScript Object Notation [20] and XML formats, which can encode arrays, trees, and graphs. In the case when the format is flexible, we couple the `Format` and `Type` declaration in order to disambiguate the data structure being captured in a file. These couplings are denoted as **F**ormat[Type] in the remainder of this dissertation. Our **T**ype[Format] couplings extend the approach offered by Multipurpose internet mail extensions (MIME) [30] by allowing for an extensible type system that supports an arbitrary depth type hierarchy encoded in OWL [38], as opposed to the rigid two-level hierachy enforced by MIME.

1.3.2 TARGET Rule

The query target is captured by the the VISUAL rule, which produces a clause specifying the `Visualization Abstraction` the source data should be transformed into, as well the

`Viewer` that should present the resulting abstraction. This information is used by our management system, described in Section Chapter 5, to identify the end-condition for our query-to-pipeline translation algorithm that comprises the semantic layer.

The concepts `Visualization Abstraction` and `Viewer` are elaborated on in Chapter 4, but for the purposes of this chapter it suffices to know that `Visualization Abstractions` can be considered labels that refer to specific visualizations, such as isosurfaces and volume rendering. `Viewers` refer to programs that present the abstractions onto the screen, such as visualization specific software like ParaView [11] or more generic applications like a Web browser that can display images in standard formats. Using labels to refer to visualization techniques has some negative implications regarding universal interpretation among members of across communities, but these limitations may be addressed by appending more rigorous descriptions of the visualization technique, for example functional specifications. This is described further in Chapter 4.

1.3.3 PARAM Rule

The PARAM production rule generates parameter binding declarations that serve to bind a value to a visualization service parameter. Parameters can control many aspects of visualization generation, from filtering of input data to controlling retinal properties [9], such as color, size, and orientation. Further elaboration on the role of parameters in visualization generation processes is described in Chapter 2. In our query language, parameter bindings are closely related to Cascading Style Sheets (CSS) [48] for describing the presentation semantics (i.e., the look and formatting) of documents and consist of a list of property (i.e., parameter) and value pairs. The kinds of properties that can be controlled using parameter bindings depends upon the kind of `Visualization Abstraction` specified in the VISUAL clause. For example, volume rendering typically has an opacity (i.e., inverse of transparency) and color functions parameters while isolines and isosurfaces have an associated contour interval and perhaps an annotation interval for labeling purposes. Some parameters are generic, for example spatial orientation or scaling (i.e., size) control. The

relationships between a particular `Visualization Abstraction` and associated parameters is captured in the model described in Chapter 4. We describe how these relationships can be leveraged in order to narrow the parameter space users must contend with, similar to how faceted searches restrict selection options [35] in Chapter 5. Additionally, a set of hard coded parameter vales can be associated with a data type, alleviating users from explicitly setting parameter bindings in the query.

1.3.4 PREFIX Rule

The resources `Visualization Abstraction`, `Viewer`, `Type`, and `Format` are specified using Uniform Resource Identifiers (URIs) [56]. These URIs can become long and make visualization query reading and writing cumbersome. We therefore have included the capability of specifying prefixes that provide a concise notation for referring to query resources. A prefix declaration is constructed by associating a shorthand label with a base URI similarly to constructing prefix declarations in SPARQL [64].

1.4 Visualization Query Examples

The following is a set of visualization queries, presented along side with the resultant visualizations. The examples range from scientific visualizations of seismic data to information based visualizations of graphs (i.e., networks).

1.4.1 Velocity Models

The following visualization queries are requesting visualizations from seismic velocity data. Velocity data results from measuring the travel time required for a sound wave to propagate through a subterranean region of the Earth. From the travel times, users can calculate the density of the subsurface and use the density values to infer the material comprising the region.

PREFIX	formats	http://somedomain.org/formats.owl#	
PREFIX	types	http://somedomain.org/types.owl#	
PREFIX	abstractions	http://somedomain.org/visualizations.owl#	
PREFIX	viewersets	http://somedomain.org/viewersets.owl#	
PREFIX	renderParams	http://somedomain.org/isoParams.owl#	
PREFIX	renderParams	http://somedomain.org/renderParams.owl#	
VISUALIZE http://somedomain.org/velocity.3d			
AS abstractions:Isosurfaces IN viewersets:WebBrowser			
WHERE	TYPE	= types:3DVelocityGrid	AND
	FORMAT	= formats:binFloatSequence	AND
	isoParams:contourInterval	= 1000	AND
	renderParams:backgroundColor	= 1/1/1	AND
	renderParams:xRotation	= 5	AND
	renderParams:yRotation	= 0	AND
	renderParams:zRotation	= 0	

Figure 1.6: Velocity Model Isosurface Query

The query in Figure 1.6 requests for `types:3DVelocityDataset` encoded in `formats:binFloatSequence` and located at `http://somedomain.org/velocity.3d`, to be transformed into a set of `abstractions:Isosurfaces`. Isosurfaces consist of a set of surfaces, where each surface represents points of a constant value (e.g. $f(x, y, z) = w$, where w is a constant) within a volume of space. Since the the surfaces are drawn at constant intervals, areas where surfaces are in closer in proximity indicate regions with a higher rate of change. This particular visualization query requests that surfaces be drawn at regular intervals spaced by 1000 units, as indicated by `isoParams:contourInterval = 1000`. Additionally, the query specifies parameter arguments that set the background color to white (i.e., 1/1/1 for Red/Green/Blue) and rotate the volume 5 ° about the x-axis. All other axes are not rotated but left in their default orientation designated by the underlying toolkit.

The query also specifies that the resultant isosurfaces should be viewed using `viewersets:WebBrowser`. Since most web browsers can display standard image formats, including GIF, JPEG, and PNG, the resultant isosurfaces visualizations can reside in any one of the aforementioned formats and still adhere to the query specifications.

Item (a) in Figure 1.8 presents a visualization resulting from execution of the query in Figure 1.6. Note the slight angle at which the image is oriented due to the 5 ° rotation of the X axis. To emphasize the effects of parameters on the final visualization, consider the

WHERE	TYPE	= types:3DGrid	AND
	FOMAT	= formats:binFloatSequence	AND
	isoParams:contourInterval	= 500	AND
	renderParams:backgroundColor	= 1/1/1	AND
	renderParams:xRotation	= 30	AND
	renderParams:yRotation	= 0	AND
	renderParams:zRotation	= 0	

Figure 1.7: Velocity Model Query Snippet

Table 1.1: Opacity Piecewise Function

X Component (Velocity Value)	**Y Component (Opacity Value)**
1000	0.2
2000	0.3
4000	0.4
8000	0.5

query fragment in Figure 1.7.

These parameter bindings specify that the resulting visualization should have twice the amount of surfaces as the query in Figure 1.6 and should be rotated at 30 ° about the X axis rather than 5 °. The resultant visualization is presented in item (b) in Figure 1.8 and the effects of the parameter setting can be seen when compared to item (a).

Item (c) presents a visualization from the same source data as the queries in Figure 1.6, except that the specified abstraction is `abstractions:VolumeRendering`. Volume rendering refers to projecting three dimensional, discretely sampled points as a two-dimension continuous objects on the screen. In order to allow for some glimpse into the internal structures of volumes, volume rendering algorithms typically provide parameters to control the opacity of the object. Opacity refers to the condition of lacking transparency or translucence and therefore is equivalent to the inverse of transparency. For example, associating the exterior surface of volume with a high opacity value would hide structures within the volume. Many toolkits such as VTK specify opacity using piece-wise functions that are defined by a set of points. In this visualization, the particular opacity function used is defined by Table 1.1.

This particular function is monotonically increasing with respect to increasing velocity

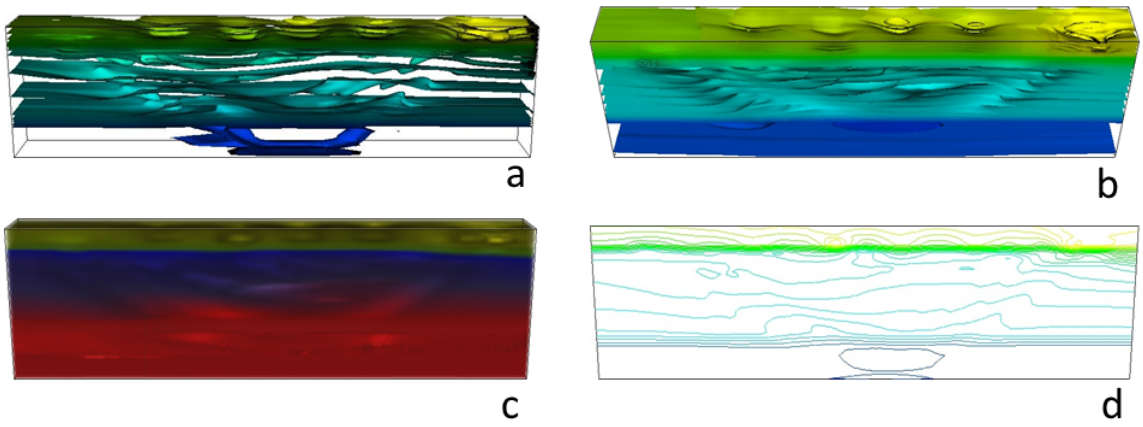


Figure 1.8: Velocity Model Visualizations: (a) isosurfaces-1, (b) isosurfaces-2 (c) volume rendering, (d) contour lines

values and therefore features in the dataset associated with higher velocity values will be more prominent in the visualization. In terms of a visualization query, we would encode this function in the form: `volumeParams:opacity = 1000,0.2/2000,0.3/4000,0.4/8000,0.5`. In these cases when complex parameters represent objects other than scalars, a serialization scheme to represent the objects in the query was devised. A complete serialization system that can encode arbitrary objects is outside the scope of this research however.

Finally, item (d) was generated by executing a query specifying for `abstractions:ContourLines`, a 2D analogue of isosurfaces. In this case, the resulting visualization was constructed from slicing a two dimensional plane from the three dimension input grid of velocity values. All four of these velocity model visualizations were generated from services supported by the Visualization Toolkit (VTK) [70] (i.e,. the semantic layer translated the queries into VTK pipeline equivalents).

1.4.2 Solar Data

The query in Figure 1.9 is requesting the generation of a `abstractions:Sphere` from solar image data encoded in the Flexible Image Transport (FITS) [34]. The type of this data is specified as `owl:Thing`, which is the most generic data type; every data type can be

PREFIX	formats	http://somedomain.org/formats.owl#	
PREFIX	owl	http://www.w3.org/2002/07/owl#	
PREFIX	abstractions	http://somedomain.org/visualizations.owl#	
PREFIX	viewersets	http://somedomain.org/viewersets.owl#	
VISUALIZE http://somedomain.org/solor.fits			
AS abstractions:Sphere IN viewersets:WebBrowser			
WHERE	TYPE	= owl:Thing	AND
	FORMAT	= formats:FITS	

Figure 1.9: Solar Spherize Query

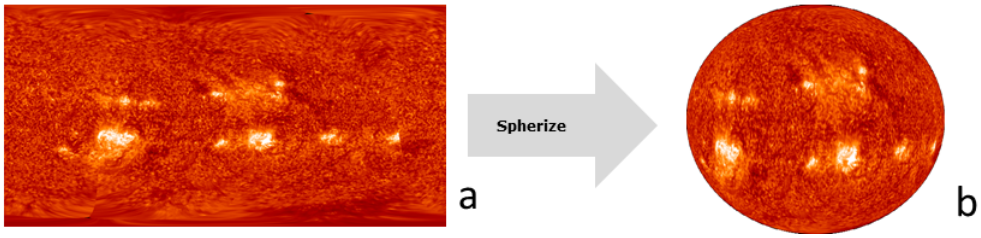


Figure 1.10: Solar Visualizations

classified as `owl:Thing`, similarly to how all Java classes share a subclass relationship with the generic type Object. Figure 1.10 presents the input data to the query in item (a) and the result of the spherize query in item (b). The transformation in this example was supported by ImageMagick [49]. Note that this query is devoid of any parameter bindings meaning that that the underlying ImageMagick services were devoid of parameters or were *hard-coded* and made unavailable for configuration.

1.4.3 Graphs and Networks

In this example, a Resource Document Framework (RDF) [53] graph that represents a portion of the visualization model described in Chapter 4 is visualized. The graph data describes the possible transformations between different data and are visualized as nodes labeled with data **T**ype[Format] couplings. The directed arcs indicate that some data X can be transformed into some other data Y. The query in Figure 1.11 requests for the RDF graph data to be transformed into `abstractions:DataTransformationsGraph` as shown in Figure 1.12. Since the input RDF graph data is actually used by our semantic layer to

PREFIX	formats	http://somedomain.org/formats.owl#	
PREFIX	types	http://somedomain.org/types.owl#	
PREFIX	abstractions	http://somedomain.org/visualizations.owl#	
PREFIX	viewersets	http://somedomain.org/viewersets.owl#	
VISUALIZE http://somedomain.org/services.rdf			
AS abstractions:DataTransformationsGraph IN viewersets:WebBrowser			
WHERE	TYPE	= types:VisualizationKnowledgeBase	AND
	FORMAT	= formats:RDFXML	

Figure 1.11: Data Transformation Paths Query

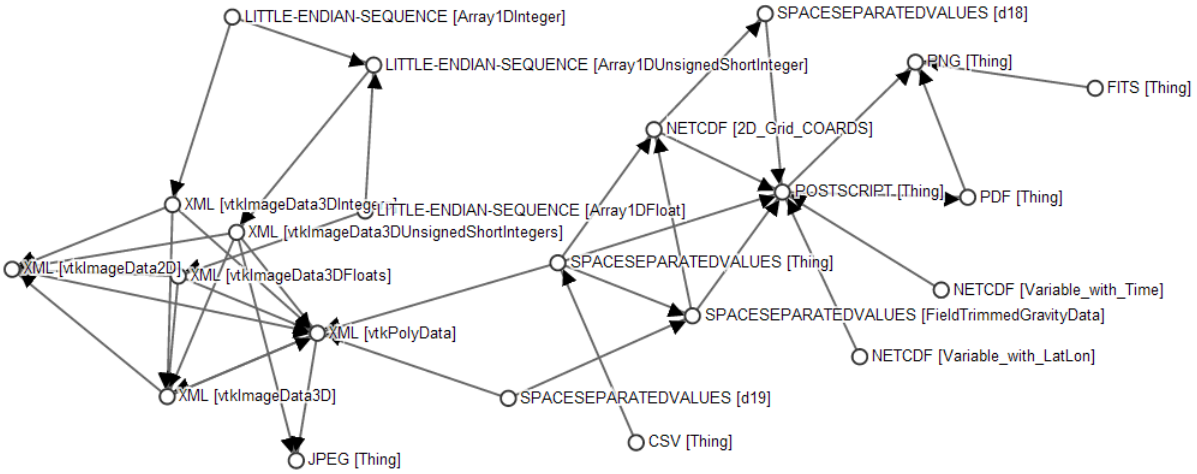


Figure 1.12: Data Transformations Visualization (Note the labels refer to Format[Type] rather than Type[Format]

translate visualization queries into pipelines, the visualization is actually a representation of the layer's translation capabilities.

1.4.4 Gravity Data

The final query presented exercises the usage of the wildcard (∗) symbol for specifying the `Visualization Abstraction`. The semantics of the wildcard symbol are similar to the that of the UNIX star, and *match-up* with any abstraction the system can generate. The wildcard query is useful for exploration and benefits users who do not have any preconceptions about how their input data should be visualized. In these cases, the system will return every possible visualization it can generate for the data (shown in Figure 1.14 which

PREFIX	formats	http://somedomain.org/formats.owl#	
PREFIX	types	http://somedomain.org/types.owl#	
PREFIX	abstractions	http://somedomain.org/visualizations.owl#	
PREFIX	viewersets	http://somedomain.org/viewersets.owl#	
VISUALIZE http://somedomain.org/gravity.txt			
AS ∗ IN viewersets:WebBrowser			
WHERE	TYPE	= types:gravity-data	AND
	FORMAT	= formats:CSV	

Figure 1.13: Gravity Data Query

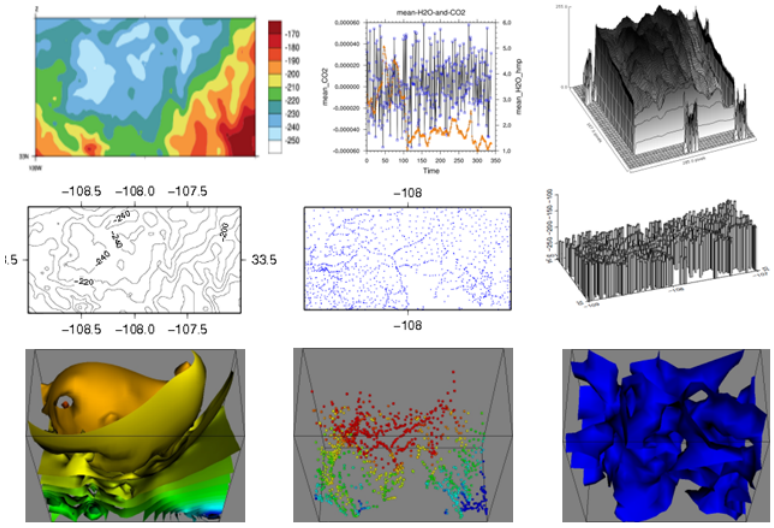


Figure 1.14: Gravity Visualizations

can be overwhelming depending upon the comprehensiveness of the answering system (i.e., number of known toolkits).

When using wildcards, the semantic layer still uses the information about the query source and target to compose pipelines, but does not restrict the relevant pipeline set based on the visualizations they generate; the pipelines can generate any visualization as long as they operate on the source data and generate visualizations that can be presented by the specified `Viewer`. The query in Figure 1.13 specifies for the visualization of gravity data. Since the wildcard abstraction was employed, the system returned many different visualization results.

1.5 Approach Overview

This dissertation describes not only the components of the semantic layer, but how past research has paved its design. This section provides a high level description of the components that make up the semantic layer:

1. a visualization query language (this chapter)
2. a *knowledge base* of visualization toolkit operators (Chapter 4)
3. a query answering system that is composed of a query-to-pipeline translator and a pipeline executor (Chatper 5)

Visualization queries can only specify a data source and visualization target and therefore it was necessary to develop automated methods for supplementing the information specified in queries since they are devoid of any sequencing information required for constructing pipelines. The supplemental sequencing information is structured and organized by a novel visualization model described in Chapter 4. To design the model, concepts from past visualization modeling efforts (Chapter 3) were borrowed and expanded upon. Using the resulting model, visualization toolkit operators can be described in a formal machine-readable manner that can be leveraged by our semantic answering system. Operator descriptions usually reside in natural language (e.g., English) embedded within scientific publications or users manuals and therefore cannot be reasoned with by machines. Our approach relies heavily on users' ability to map these natural language descriptions into formal model-based operator descriptions (by way of an API), thereby populating a *knowledge base* that serves as the foundation for our work. The knowledge base contains formal descriptions about visualization toolkit operators, how they can be composed, and what task they implement (e.g., transforming, mapping, filtering). The knowledge base is used by our query-to-pipeline translator (Chapter 5) to construct pipelines equivalent to input queries.

The knowledge base resulting from our formal descriptions can be conceptualized as a graph, where nodes represent toolkit operators and arcs represent the ability of data to

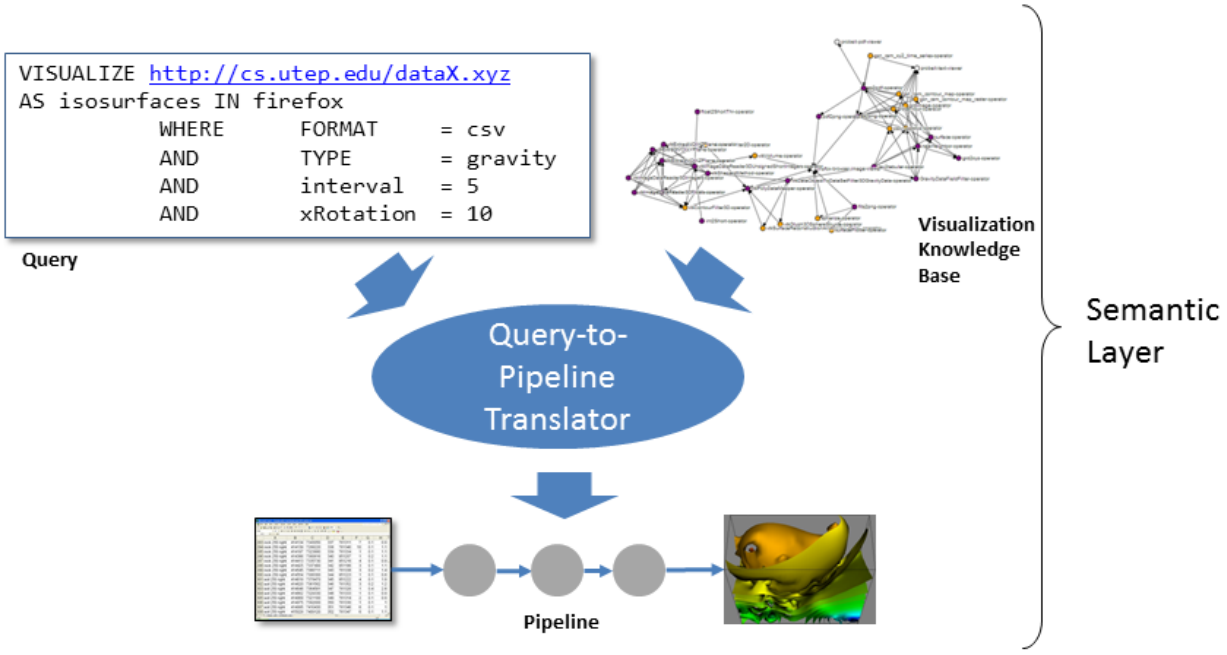


Figure 1.15: Components of the semantic layer

flow between operators. Therefore, a linear chain of connected nodes represents a pipeline in our system and can be executed in a similar manner as UNIX pipes. Although, more complex visualization applications may reside in a wider variety of pipeline forms, such as trees or graphs, our work is limited to the automatic translation and execution of linear style pipelines (more on the structures of pipelines can be found in Chapter 4). Figure 1.15, presents the components of our semantic layer: visualization queries, the knowledge base, and the resulting pipelines translated from the query. The pipeline executor is not shown in this figure however.

1.6 Value of Queries

Visualization queries were conceived with the intent of reducing the cost associated with generating visualizations. Using a functional model, *cost* and *profit* can be used to identify stages in visualization processes that should be optimized [79]. At a high level, the profit P

of a given visualization method can be calculated by the expression, $P = V - C$, where V is the value provided to users and C is the cost of developing and generating the visualization method. Value can be expressed in terms of the insight gained from the visualization (i.e., visual analytics not yet considered with this work), while cost can be expressed in terms of resources (e.g., time and effort) expended through the visualization development process. Considering this visualization economic model, queries may increase profit P, by reducing the amount of time (and therefore cost) associated with generating visualizations. Our user study in Chapter 6 demonstrates that users were more accurate using queries than using pipelines. This entails that in the wild, visualization query users should spend less time in the typical trail-and-error cycle noted in [79], because they were able to express their visualization needs more accurately.

In the scheme of the visualization process as a whole, queries also aim at reducing the amount of effort users expend conveying their visualization requirements to systems and thus possibly enable non-visualization experts to more quickly generate visualizations. Cycle A in Figure 1.16 presents the visualization process that begins with non-visualization experts conveying visualization requirements to visualization experts (human to human) [22]. The experts in turn are capable of translating the non-formal visualization requirements into software specifications that can be used to drive the development of the visualization applications (human to system). In many popular systems, the software specifications are describing pipelines composed of visualization modules. The resultant application (i.e., pipeline) may rely on a number of third party transformational modules scattered across a network and therefore inter-module communication may be required (system to system). Finally, the visual artifact generated by the application must be presented back to the domain user for interpretation and analysis, which may lead to an alteration in visualization requirements and thus trigger another round of the cycle.

Another aim of visualization queries is to enable non-visualization experts to easily specify visualization requirements, where before these users may have needed to consult with experts. The role of the proposed semantic layer is to replace the human expert

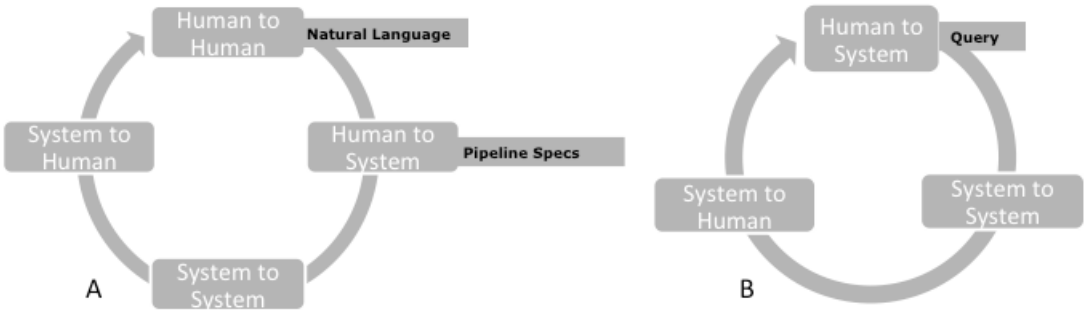


Figure 1.16: Visualization Cycle

and thus reduce the number of visualization phases as shown in Cycle B. In this case, non-visualization experts no longer need to consult with visualization experts to generate visualizations. This is also beneficial because visualization experts do not have to interpret natural language (e.g., English) visualization specifications, but can now convey their needs directly to systems using a formal query language.

Chapter 2

Visualization Toolkits

As introduced in Chapter 1, one class of visualization toolkits are known as *modular visualization environments* (MVEs) and contain sets of operators that are used to compose visualization pipelines. This chapter presents a deeper look into MVE based toolkits as well as other toolkits, some of which rely on declarative based specifications for visualization generation. Table 2.1 below presents some of the toolkits that fall into one of the two classes: MVE and declarative language based toolkits. This chapter highlights each toolkit class by presenting exemplar visualization specifications for each. The examples in this chapter will demonstrate the breadth of knowledge required by users to configure and leverage these kinds of toolkits to generate visualizations, thereby providing additional motivation for the use of queries.

Table 2.1: Visualization toolkits grouped according to class

Toolkit Class		**Toolkit**
MVE	Textual Programming Based	VISAGE [71], Visualization Toolkit (VTK) [70], Generic Mapping Tools (GMT) [80], NCAR Command Language (NCL) [28], Titan [82], Prefuse [36]
	Visual Programming Based	A Visualization System (AVS) [24], ParaView [11], ParaViewWeb [45], IBM Explorer (now OpenDX) [50, 4], IRIS Explorer [29], Sieve [42]
Declarative	Query Based	A Presentation Tool (APT) [51], Tableaux [75, 52], SAGE [66], VisTrails [69]
	Domain Specific Language Based	Protovis [37], Data Driven documents (D3) [3]

2.1 MVE Based Toolkits

Modular visualization environment (MVE) based toolkits provide *building blocks* known as operators, from which uses can sequence to form custom visualization pipelines. These pipelines typically share a structure in which each operator provides only a piece of the functionality necessary for transforming data and information into visualizations, although it is possible for a single operator to perform the necessary transformations. Constructing pipelines is similar to and faces the same challenges associated with general software development, except in a restricted design space. Users are required to formulate conceptual sequences of tasks that can transform their data into some visualization (i.e., visualization theory). During the implementation phase, users must understand how to map the conceptual sequence of tasks to executable operators sequences. In both phases, the user is required to manually define the transformation sequences.

2.1.1 Textual Programming Based Toolkits

Depending upon the MVE, some toolkits require that humans use some textual programming language to specify pipeline sequences. These kinds of toolkits will be elaborated on the most since they drive the motivation for our work and serve as the foundation of our model in Chapter 4. In these kinds of toolkits, the programming language used to specify the pipeline is largely dictated by the communication protocol supported by the operators, for example through command line invocations or application programming interface (API) function calls. On one hand, Generic Mapping Tools (GMT) is available as a set of executables and so can be invoked through scripts or directly through the command line. On the other hand, Visualization Toolkit (VTK) operators are implemented in C++ and therefore any C++ application can interface with these modules, although the VTK library set also includes a set of operator wrappers implemented in Java, Python, and TCL. In other cases, toolkits introduce a unique language, such as NCAR Command Language (NCL), in which operators can only be accessed though the NCL programming

language. In this case, writing NCL-based pipelines requires knowledge not only about the set of supported operators, but also about the NCL programming language itself. The issue of communication interfaces and protocols may not be an issue when using a single toolkit, however development of hybrid applications spanning across the usage of multiple toolkits may become cumbersome in cases when the employed toolkits do not share common interfaces. The visualization model described in Chapter 4 addresses the variance in communication protocols by introducing a service layer that implements a common communication protocol.

To exemplify the use of manually programmed MVE-based toolkits, consider the following high level task description of a scenario for visualizing gravity data. Gravitational anomalies can be visually observed when data is represented as a raster or contour map, because users can quickly identify where spikes in the data exist by finding regions on the map containing a high diversity of colors or contour lines close in proximity; contour or isolines are drawn at regular intervals so tightly drawn lines indicate a rapid increase. At a high level of abstraction, the sequence of tasks specified in Figure 2.1 can generate a contour map of 3D points (i.e., scalars associated with an X, Y, and Z spatial coordinate).

Task	*Description*
1. Grid:	Create a uniformly distributed point set by applying a gridding algorithm. Typically, in order to generate contour lines, the underlying data must be uniformly distributed
2. Contour:	Extract isolines geometry from the uniformly distributed dataset and possibly fill in areas with color associated with a color map
3. Render:	Project the geometry into 2D pixel data that can be presented on the screen or printed

Figure 2.1: Task description on how to build a gravity contour map

In our contour map task description, we assume our input data is sourced from the Pan American Center for Earth and Environmental Science (PACES) data store [62], which provides access to gravity data as a set of XYZG records. Note that XYZG data can be broken down into two components, the 3D spatial component (i.e., XYZ or longitude, latitude, and elevation) and the associated scalars (i.e., G) and is thus regarded as three-dimensional

data. Using the visualization query notation, PACES gravity data can be described as **P**ACES-Gravity[Tabular-ASCII]. In addition to the data points, gravity datasets also have additional fields for specifying alternative coordinate systems and meta-data indicating the station in which the data was collected. The following is a snippet of a gravity dataset:

```
lonnad83 latnad83 cbanom267 elevngvd88m sourcecode author contributingagency
-108.9992812 33.2830637 -199.39 1896.38 4686_ngs UNKNOWN U._S._GEOLOGICAL_SURVEY_(USGS)
-108.9964498 33.1484000 -193.98 1766.27 6335WIL8 R._A._MARTIN,_J._C._WYNN,_G._A._ABRAMS U.
    _S._GEOLOGICAL_SURVEY_(USGS)
-108.9947900 33.9340538 -235.33 2656.47 3638_183 UNKNOWN DMAH/TC
-108.9927849 33.6000539 -212.15 2127.53 4686_312 UNKNOWN U._S._GEOLOGICAL_SURVEY_(USGS)
-108.9919492 33.0878989 -195.33 1777.17 6335MU10 R._A._MARTIN,_J._C._WYNN,_G._A._ABRAMS U.
    _S._GEOLOGICAL_SURVEY_(USGS)
```

The dataset fragment is structured as a table with a set of fields, where not all fields are pertinent to the desired contour map visualization. Independent of any toolkit idiosyncrasies, only three fields are required to generate a two dimensional contour map: `lonnad83` that provides an X coordinate, `latnad83` that provides a Y coordinate, and `cbanom267`, which is the gravity reading associated with the two dimensional point specified by `lonnad83` and `latnad83`.

Generic Mapping Tools

Generic Mapping Tools (GMT) provides a set of over fifty operators in the form of executables from which users manually construct pipelines together by writing shell scripts, which provide a data-flow-like processing of the input data. GMT is an open source alternative for generating Geo-graphical Information System (GIS) based visualizations which are commonly developed using sophisticated commercial software such as ESRI ArcGIS [83]. GIS systems provide extensive support for projecting three dimensional surfaces associated with the curvature of the Earth onto a variety of two dimensional planes, which each provide a different distortion. Projection is a relevant aspect of our gravity map scenario because the coordinates are specified using the geospatial coordinates latitude and longitude. Additionally, GMT provides parameters to specify what map projections should be

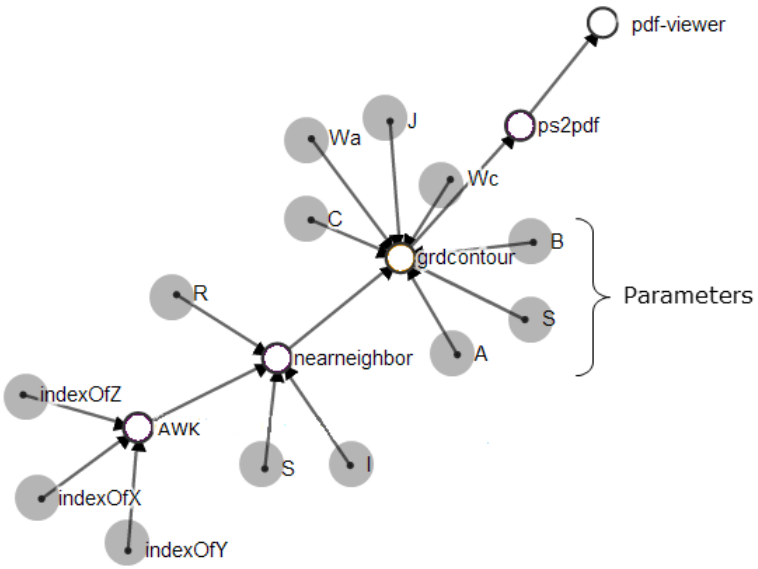


Figure 2.2: GMT Contour Map Pipeline

used to best represent the data.

Due to the shell-based environment facilitating GMT execution, UNIX based tools (such as AWK) which process field data (i.e., tabular data) can be easily incorporated into GMT pipelines and used to help mitigate the disparities between the structure of the raw input data and the data requirements of GMT executables. Figure 2.2 presents a diagram of the GMT pipeline that processes the input data and generates the requested contour map. The figure presents the pipeline from an operator-centric perspective, where the sequence of operators are represented by the white nodes whereas the flow is represented by the arcs. In an actual GMT pipeline script, the pipeline property is supported by the fact that output files of each operator serve as an input file to the next operator in the sequence. The GMT pipeline figure also presents the parameters to each operator which are highlighted by the greyed out nodes.

In our GMT pipeline, **A**WK ingests the input data (i.e., data.xyz) and generates a new filtered dataset that is devoid of the header and contains only the first three fields of data: `lonnad83`, `latnad83`, and `cbanom267`, effectively transforming our three-dimensional

point data into two-dimensional points. The **n**earneighbor operator transforms the two-dimensional data into a two-dimensional grid encoded in the scientific format Network Common Data Format (netCDF) [77]. The **n**earneighbor operator is one specific kind of gridding technique provided by GMT and can be replaced by other techniques, such as **s**urface, that may offer better coverage in terms of interpolating missing values. In these cases where more than a single operator can satisfy a conceptual task, users are responsible for understanding the underlying algorithm (provided by the different gridding operators such as **n**earneighbor and **s**urface) and for manually altering the pipeline to test the effect of the alternatives on the final contour map. The third operator **g**rdcontour is the most critical in terms of the pipeline goal because it maps the gridded netCDF data to the isoline geometry composing the requested contour map. Finally, **p**s2pdf converts the contour map residing in PostScript format to the Portable Document Format (PDF) that can be viewed using the openly available Adobe PDF viewer.

In addition to specifying the pipeline sequence of operators, users must also contend with setting the associated parameters, which fine-tune the behavior of operators and are the highlighted nodes in Figure 2.2. Improperly setting parameters can cause pipelines to crash during executions, yield empty visualizations (i.e., blank displays), or worse, misrepresent data by hiding features. For example, specifying the C parameter in Figure 2.2 with too large a value for the density distribution of the grid being contoured may generate insufficient number of isolines; an insufficient number of isolines may not expose areas where rapid change is occurring. Therefore these kinds of parameters are dependent on the structural properties of the specific data being visualized. Other parameters are more independent of the underlying structure of the data, for example setting retinal properties [9] including background color, size, and contour line thickness (e.g., Wc). These kinds of parameters can be set based on personal preference or community conventions, for example hydrology data should most likely be painted using a bluish hue. This distinction between parameters was first introduced by the Data State Model [15].

Setting parameters is challenging and usually requires trial-and-error processes, where

users learn about the sensitivity of some parameter in regards to its effect on a visualization and use this knowledge to provide a new set of parameters for the next cycle of visualization generation [43, 12]. Some efforts have delegated the trial-and-error process to automated systems that enumerate the possible combinations of parameter values and then generate visualizations for each possible combination [54]. This computationally expensive approach would then present all the possible visualization outcomes to users in a gallery-like fashion, allowing users to browse both visualization results as well as the corresponding parameter sets.

Visualization Toolkit

The Visualization Toolkit (VTK) [70] provides a set of operators implemented in C++, which can be assembled into visualization pipelines similarly to GMT. VTK provides rich support (over 200 operators) for generating three-dimensional scientific visualizations, but can still be leveraged to generate two dimensional visualizations such as isolines. Figure 2.3 presents the VTK version of the gravity contour map pipeline that follows the same representational conventions as the GMT figure.

Operator **v**tkDataObjectToDataSetFilter3DGravityData is responsible for transforming the XYZG text file into a three-dimensional point set, where the G scalars are associated to each point. Note that in VTK, points must be defined using three coordinates, and so the operator binds elevation to the Z coordinate of the points. Operator **v**tkShepardMethod is a three-dimensional version of **n**earneighbor and therefore implements the gridding task by transforming the non-uniform three-dimensional point set to a uniform three-dimensional grid. However, the requested contour map visualization is a two-dimensional object and therefore the dimensionality of the grid must be reduced. Operator **v**tkExtractVOI can be configured to extract two-dimensional slices from volumes or grids, thus implementing the required reduction and also contributing an extra pipeline step when compared to the GMT equivalent. In this case, an XY plane was extracted (i.e., sliced) from some elevation Z. The operator **v**tkContourFilter processes the extracted grid slice in order to generate

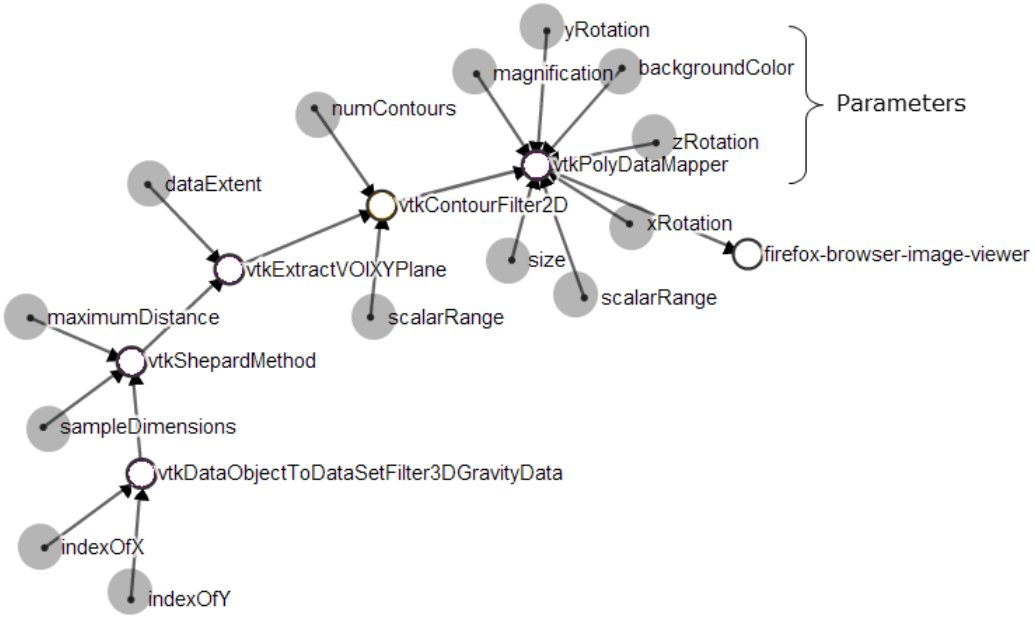


Figure 2.3: VTK Contour Map Pipeline

the isolines geometry that represent the required contour map. **v**tkPolyDataMapper converts the contour line geometry into a standard image format that can be viewed by a web browser such as FireFox. Similary to GMT, VTK operators are associated with parameters from which users are expected to manually specify using their intimate knowledge about the structure of the input data as well as that of the requested visualization.

In these pipeline examples, VTK contains some parameters that are semantically similar to GMT, for example **v**tkContourFilter relies on the parameter **n**umContours to control the number of contour lines generated, which can also be controlled in GMT. However, VTK also contains parameters not common to GMT, for example controlling the spatial orientation of visualizations by setting **x**Rotation, **y**Rotation and **z**Rotation accordingly. This added ability results from VTK's scene-graph space, which is defined in three dimensions. Therefore, all visualizations can be oriented in three dimensions regardless of dimensionality.

Output Comparisons

Figure 2.4 presents the output from the GMT and VTK contour map pipelines. Contour map A was generated from GMT and contains a nice labeling, with latitude and longitude bound to the X and Y axes respectively. This map was generated by two-dimensional points devoid of the elevation component and so all the point data resides in a single plane. As a result of this misrepresentation of reality, many line features are present in this map that are absent in map B, which was generated by VTK. In contrast, the VTK pipeline populated three dimensional points using latitude, longitude and elevation and thus distributed the point set along a volume rather than a single plane. However in VTK, an XY slice was extracted from this volume and depending upon where on the Z axis the slice was taken, the map may contain minimal data that reduces the number of line features. If a majority of the points resided on the XY plane located on elevation 1000, and a slice was taken at elevation 0, then the resulting slice would not be very representative of the input data and would only be an approximation generated by the **v**tkShepardMethod gridding algorithm. This approximation of data is evident in the VTK contour map, in which the visualization has considerably less features than the GMT version.

Additionally, the contour map generated by VTK is slightly rotated about on the X-axis and this is due to the VTK scene-graph space that is defined in three-dimensions. Therefore, even though the contour map itself exists as a two-dimensional plane, it resides in a there-dimensional space allowing for XYZ orientation. The visualization model in Chapter 4 captures the dimensionality of scene-graph spaces, thereby providing precise descriptions of visualizations.

2.1.2 Visual Programming Based Toolkits

Many MVE toolkits allow for construction of pipelines using the drag-and-drop paradigm rather than writing the pipeline code in text. These visual programming based toolkits provide users with a set of operators represented as some icon which can be dragged on a

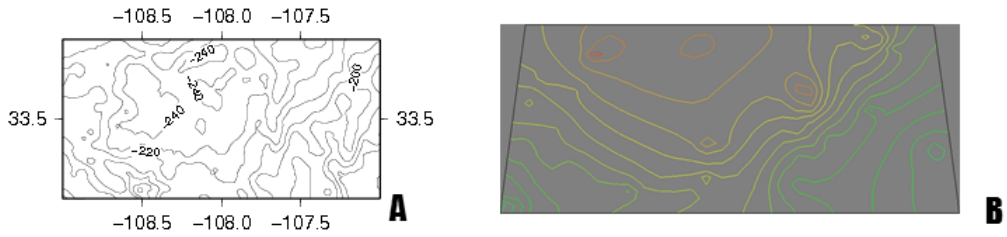


Figure 2.4: Contour Maps: map A was generated from the GMT pipeline. Map B was generated from the VTK pipeline.

canvas and connected to form pipelines, for example IBM explorer [50] and later OpenDX [4]. In these environments users construct graphs, where the operators are represented as nodes and arcs between nodes denote the flow of data between operators as shown in Figure 2.5. OpenDX provides a categorization and grouping of operators based on similar functionality, for example *rendering*. Despite these kinds of data-flow like interfaces, users are still responsible for identifying relevant operator sequences for generating requested visualizations.

2.1.3 Search Space Reduction

Although pipelines must be manually configured using MVEs, some toolkits guide the pipeline construction process by enforcing certain pipeline structural rules enforced by different visualization models described in Chapter 3. For example, the toolkits [50, 29, 70, 78] are constrained by the Data Flow model while the toolkits [16, 36] are constrained by the Data State model. These systems enforce structural rules that restrict the number of possible legal operator combinations and therefore reduce the search space; for example, renderers cannot precede mappers in a pipeline. In objected-oriented visualization systems such as [36, 70], these pipeline composition rules are enforced by the type system, for example a renderer will not produce a data type that can be ingested by a mapper and so a configuration that specified this sequence would never compile. In fact, the type system is so compelling that it is leveraged in the visualization model described in Chapter 4.

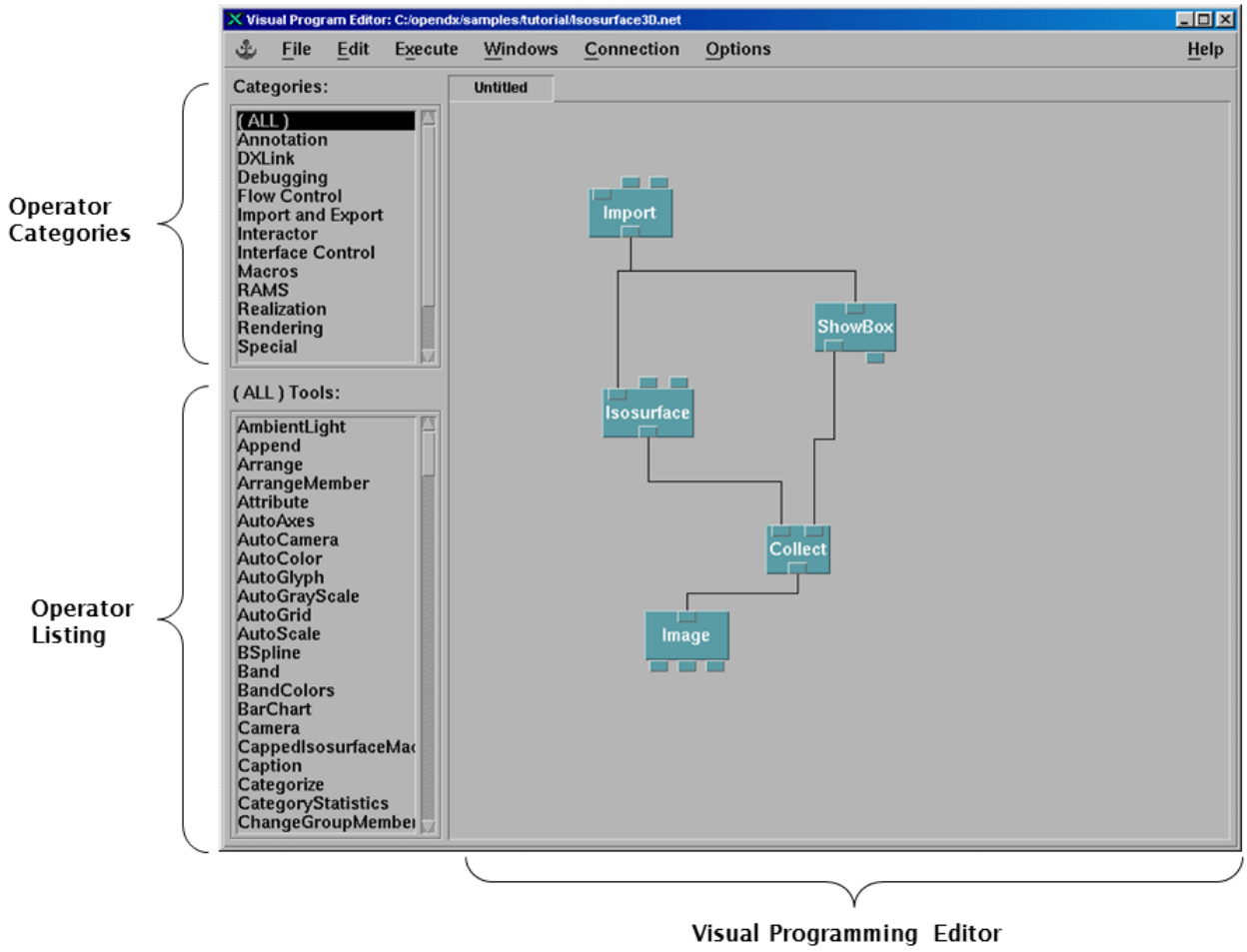


Figure 2.5: OpenDX Visualization Pipeline

2.1.4 Comparison with Visualization Queries

Using visualization queries presented in Chapter 1, users only need to specify a visualization in terms of source and target, whereas when constructing MVE based pipelines, users must concern themselves with the orchestration of operators that support the transformation of the source into the target. In general, there are a number of properties that distinguish our query language from visualization pipeline code:

- Declarative specifications − users only specify what visualization they want (e.g., iso-surfaces visualization, with black background) rather than specifying how to generate the visualization using a visualization pipeline
- Size − the query language is tailored for the single purpose of specifying visualizations, which keeps it considerably smaller in terms of language constructs when compared to general programming languages such as C++ or even domain specific languages described below.

The abstract nature in which users specify visualizations through queries greatly reduces the amount of information the user needs to understand, reducing cognitive load. In Table 2.2, different kinds of information associated with visualization generation using MVEs is listed. Through this table it is evident that writing visualization queries requires less knowledge than writing visualization pipelines. Specifically, query users do not need to understand visualization generation in terms of operators and additionally do not have to understand the language used to interface with the toolkits. Because they do not need to know about interfacing languages, they are also relieved of the integration challenges associated with using operators spanning across different toolkits.

The table indicates that MVE users require more information to use visualization toolkits than users of visualization queries, as expected considering visualization queries are a fourth generation declarative language. This provides some evidence that users may be able to generate visualizations more efficiently through specifying queries, but this claim is formally investigated in Chapter 6. The next section delves into other declarative based

Table 2.2: Queries versus Pipelines

Concern	Knowledge Type	Queries	Pipelines
Data	Type	X	X
	Format	X	X
	Location (URL)	X	X
Abstraction	Visualization	X	X
	Properties	X	X
Toolkit Expertise	Inter-toolkits		X
	Operators		X
	Language		X
Pipeline Expertise	Transformers		X
	Mappers		X
	Viewers	X	X
	Sequencing		X
	Parameters	X	X

approaches for specifying visualizations and so is regarded as related work. Through the following toolkit descriptions it will become evident that these kinds of declarative languages serve to specify visualizations belonging to a given domain, where the visualization queries in this work are domain agnostic.

2.2 Related Work

The challenges associated with manually constructing visualization applications has been well documented and in some cases quantified [79]. To remedy, previous efforts have focused on abstracting away the low level transformational and graphical mapping algorithms by using declarative languages. The abstract nature of the specification languages allows users to focus on what visualizations best suit their data rather than focusing on the required transformations. To this end, the efforts are broken into two broad categories: query languages and domain specific languages [59].

2.2.1 Queries

An early effort allowing users to issue query-like visualization requests was supported by a Presentation Tool (APT), which generated abstract *graphical designs* from relational data

[51]. Users would only need to specify the data that should be visualized and the presentation tool would leverage the characteristics of the specified data (e.g., nominal, ordinal, quantitative, functionally dependent) to generate two dimensional graphical designs that could be rendered into into two dimensional images such as scatter plots and bar charts. Other work has also documented similar characteristics that can be used to drive visualizations of relational data [67] and are elaborated on in Chapter 3. Using these kinds of systems, data specifications would be dictated using a controlled vocabulary such as:

> Present the Price and Mileage relations.
> The details about the set of Cars can be omitted.

This query requests that Price and Mileage data be represented in a graphical form which emphasizes the structural relationships between the two data. Given that Price and Mileage are quantitative fields (e.g., continuous data) and functionally dependent upon a type of Car (e.g., $Price(Car)Mileage(Car)$, an appropriate representation would be a scatter plot with Price and Mileage bound to the axes. This association of data with some tic-mark is specified using the *Encodes* predicate shown below. Going back to the query, note the clause which requests that "The details about the set of Cars be omitted", thus specifying that the generated visualization consist only of Price and Mileage data. Any other information about Cars contained in the relation, such as example weight and condition, should not be painted.

ATP houses a library of visualizations, such as *ScatterPlot*, each described in terms of a *composition algebra* that serves as a language for which to describe visualizations. Each ATP visualization has a formal definition described in terms of the composition algebra that describes how input data should be mapped to the graphical constructs comprising the visualization. These graphical constructs serve as graphemes in the language and are inspired by Bertin's work on semiology [1]. Given an ATP query, a set of relational data, and the set of visualizations described using the composition algebra, the presentation

system knows how to map the relational data to the visualization by way of the *Encodes* predicate:

Encodes (VertAzis, [3500, 13000], ScatterPlot)
Encodes(HorzAris, [10, 40], ScatterPlot)
Encodes(Points, Cars, ScatterPlot)
Encodes(Position(Points, VertAxis), Price(Cars), ScatterPlot)
Encodes(Position(Points, HorzAxis), Mileage(Curs), ScatterPlot)

The *Encodes* predicate specifies the mapping between the input data and the visualization, in this case a *ScatterPlot*. In this example we see that the functional range of Price (i.e., [3500, 13000]) and Mileage (i.e., [10, 40]) are used to populate the ticks on the Y and X axes respectively. Additionally, the Car instances should represent the points of the ScatterPlot and be positioned according to the X coordinate (i.e., Mileage) and Y coordinate (i.e, Price). These predicates, along with the algebraic definition of scatter plot, compose the *graphical designs* that ATP can then render into images.

APT has since evolved into a commercial tool Tableaux [75, 52], which is founded on the same principles as APT; the data characteristics of relational data are still used to automatically drive the generation of two dimensional visualizations. Using Tableaux, input data can be specified by a novel user interface where users drag relational table fields onto *shelves* and based on the order and position on the shelves, Tableaux can determine whether the data should be treated as functionally dependent and what axes the data should reside on. Based on data characteristics, the system can best select the appropriate visualization that *best* represents the data given criteria outlined by Bertin. For example, Table 2.6 below maps different combinations of ordinal (O) and quantitative data (Q) to different visualizations.

The query language this thesis proposes is more generic than previous efforts and can work with a multitude of different data structures and formats as well as generate wider

Table 2.3: Tableaux Visualization Taxonomy: O = Ordinal Data, Q = Quantitative Data, i = Independent Variable, d = dependent Variable

Data Characteristics X Y		**Visualization Type**
O O		Text Table
O Q	O Q_i	Gantt Chart
	O Q_d	Dot Plot
Q Q	Q_d Q_i	Line Graph
	Q_d Q_d	Scatter Plot
	Q_i Q_i	Geographic Map

range of visualizations, in contrast to systems such as Tableaux that only work with multivariate relational data and two dimensional statistical-like visualizations. The trade-off with visualization queries is that the required data characterization (i.e., **Type**[**Format**]) treats data holistically and is generally weaker than the rich characterizations used by Stolte and Mackinlay. Therefore, the visualizations resulting from visualization queries may not be as *expressive* or *effective* [51] as the displays generated by Tableaux. Using visualization queries, it may be possible to set parameters that control the association of relation data fields with Bertin's marks and axes, however this is contingent upon the set of available parameters the operators expose.

Other graphical user interface (GUI) based approaches to specifying visualization queries include systems which allow users to specify visualization pipelines using a data-flow like graphical notation, where nodes represent operators and arcs denote the directional flow of data [69]. In VisTrails, these pipelines are treated as search patterns, which can be matched up with existing similar pipelines that had been previously created and executed. In this sense, user could focus on querying for visualizations from a process-centric standpoint rather than a data characterization standpoint as supported by Stolte and Mackinlay, although VisTrails can only provide answers for visualizations that had been previously generated.

Similarly to VisTrials, a query language was developed that could be used specifically for the retrieval of existing statistical based plots, such as time series and scatter plots [27]. The same language that was used to query the plots is used to describe and annotate the

components of the plots and includes constructs for describing if the plot is composed of points or lines. Additionally, the language allows for the description of the data that was used to derive the plots, including the semantic type of the data as well as specific data point values. The query language could be used to retrieve whole plot instances or specific data points contained in a specific plot instance.

Finally, systems have also been devised that consider visualization specification from both a bottom-up (i.e., data characteristics) and top-down (i.e., visualization properties) approach. The SAGE system [66] allows users to request for visualizations by sketching out *visualization templates* using a palette of marks, similar to those proposed by Bertin, including points, bars, and lines. SAGE users would drag different marks onto a canvas to specify the template and then drag relational data fields over the marks on the canvas, which would specify a mapping between that data and the mark on the canvas. In essence, SAGE provides a GUI based approach for specifying the *Encodes* relationship of ATP.

2.2.2 Domain Specific Languages

Data Driven Documents (D3) [3] allows users to associate array based data with elements of the Domain Object Model (DOM) using a declarative domain specific language built on JavaScript. Two features associated with D3 that are relevant for this work are the: lack of an abstraction that encapsulates the underlying graphics primitives and the declarative domain specific nature of the language.

Rather than introducing a set of mark definitions that abstract away the lower level graphical constructs of Scalable Vector Graphics (SVG) [26], as is the case with Protovis [2], D3 allows users to bind data directly to the SVG canvas. In contrast, the visualization queries in this dissertation require that users specify `Visualization Abstractions`, such as isosurfaces, that abstract the low level graphical primitives. Visualization query users are not burdened with the geometric primitives such as points, lines and polygons supported by graphics libraries such as OpenGL [81]. Therefore, visualization queries allow users to focus on specifying visualizations at a more holistic level of detail rather than binding data

directly to points and lines.

The more direct approach offered by D3 can attribute some of its success to the large set of interoperable web-based technologies that many developers are already familiar with. Rather than hide the technologies developers are already familiar with (e.g., HTML and SVG), D3 empowers their usage by providing mechanisms for easily binding data to them. So even though D3 users need to know the display capabilities of web related technologies (the premise is that they already do), they do not have to understand how to bind data to these technologies as this is the function provided by D3. Our premise with visualization queries is that a large part of our user base may not have experience using the DOM or graphics libraries such as OpenGL, in which case our queries serve to abstract away the lower level primitives, and perhaps consequently, reduce the level of control offered.

D3 also provides a declarative manner for which to bind data to the DOM. For example, the following code specifies that an array should be bound to the SVG layer (Lines 1-2) and that the array data should be passed, element by element, to some user defined functions that define a curve (Lines 4-6) and the area under the curve (Lines 8-10).

```
1  var svg = d3.select("body").append("svg:svg")
2    .data([[1, 1.2, 1.7, 1.5, .7, .5, .2]]);
3
4  svg.append("svg:path")
5  .attr("class", "line")
6  .attr("d", d3.svg.line().x(x).y(y));
7
8  svg.append("svg:path")
9    .attr("class", "area")
10   .attr("d", d3.svg.area().x(x).y0(160).y1(y));
```

Note the brevity of the code that hides the complexity of iterating over the data array and binding coordinates to svg:path. Also highlighted in this code snippet is the capability of D3 to leverage the myriad of web-based technologies, such as CCS, to declaratively specify presentation details such as line color and thickness.

```
1  .line {fill: none; stroke: black; stroke-width: 1.5px;}
2  .area {fill: lightblue;}
```

Chapter 3

Visualization Models

Chapter 2 presented different toolkit-based approaches for generating visualizations. In particular, MVE based toolkits require that users generate visualizations by specifying and then executing pipelines, which describe *how* the requested visualizations are generated in terms of operator execution sequences. Declarative language based toolkits provide abstractions from which users specify *what* visualizations are requested using queries and domain specific languages. The visualization queries proposed in Chapter 1 follow the declarative toolkit paradigm and allow users to generate visualizations by specifying only the source (**T**ype[Format]) descriptions of input data and target (i.e., `Visualization Abstraction` and `Viewer`). These queries are then processed by a semantic layer that automatically translates queries into equivalent pipelines composed of MVE operators and executes the pipelines to generate query results.

In order for the semantic layer to be able to automatically construct pipelines using operators across different MVEs, it must be equipped with a uniform view or *model* of different operators supported by a diversity of toolkits. This chapter introduces previous work in modeling visualization processes from which are borrowed and extended to define a new model that is generic enough to describe many operators but specific enough to drive automatic pipeline orchestration (described in Chapter 4). Table 3.1 presents a grouping of existing visualization models according to perspective: transform centric, data centric, user centric, or hybrid, in which case a single model is based on a combination of perspectives. Note that the listing below is not comprehensive but targets specific models that contain relevant perspectives and concepts for the model proposed in the next chapter.

Table 3.1: Visualization Models

Model Focus	**Model Name**
Transform Centric	Data Flow [33]
	Data State [15]
	Direct Mapping [52]
Data Centric	Relational Data Characterizations [67, 84]
User Centric (taxonomies)	Data State [13]
	Design Model [76] Domino Notation [7]
	Task Based Model [73]
Knowledge Driven Models	Knowledge Cycle Model [12]
Hybrid Models	Visualization Ontology [23]
	Our MVE centric model

3.1 Transform-Centric Models

This first class of models aims to provide an abstraction of visualization processes in terms of functions that map data and information into visualizations. The functional mapping can be composed of a sequence, for example $g(f(e(x)))$ where x is the input data and e, f, and g are MVE based toolkits operators. This functional sequence has been modeled from a Data Flow perspective [33] presented in Figure 3.1. The Data Flow model conceptualizes three basic transformational stages as boxes while arrows indicate the flow of data between the stages, for instance the output of the data gathering stage feeds into the mapping stage. The data enrichment and gathering stage is responsible for collecting and transforming the input data into forms that can be ingested by a mapper. The mapping stage transforms or *maps* the gathered data into a geometric form (e.g., points, lines, and polygons) that is fed into the rendering phase, which transforms the geometry into pixel data that can be presented on the screen. The Data Flow model was meant to provide a high level architecture for visualization applications, thereby providing structure for future visualization applications. In practice, the data gathering stage can consist of a complex network of operators that can aggregate, filter, convert and transform input data before being served as input to a mapper. Additionally, there may be multiple format conversions and data type transformations of the geometry output by a mapper before that data can be ingested by a renderer. For

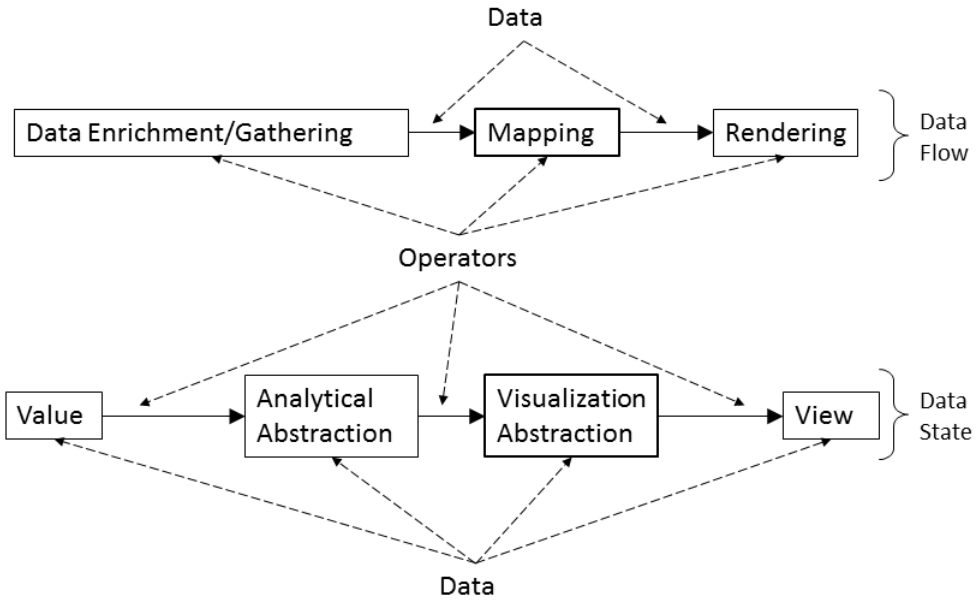


Figure 3.1: Data Flow versus Data State Model

example, a PostScript file containing some visualization may be converted into Portable Document Format (PDF) before being presented by a PDF reader to a scientist. These concerns are addressed in the model presented in the next chapter.

The Data Flow model places emphasis on the processing steps associated with visualization pipelines and so provides users with an understanding of visualization processes from a procedural perspective. Because there is a correspondence between the data flow processing steps and specific toolkit operators, most MVE based toolkits allow users to specify visualization pipelines from a Data Flow perspective [50, 70, 42, 29, 82]. Therefore, if users have a strong understanding of the high level data flow model semantics, they are more likely to have faster turn around times developing visualization applications using the aforementioned toolkits because they only have to map the high level flow to executable modules.

The Data State model, also presented in Figure 3.1, serves to provide users with an understanding of how data is transformed from its value (i.e., raw form) to its view (i.e., graphical representation) and so the emphasis is placed on the different states of data as it

flows though a pipeline. Note in Figure 3.1 that the boxes in the Data Flow model refer to operation stages while the boxes in the Data State diagram refer to the different states of data as it evolves through a visualization pipelines: value, data abstraction, visualization abstraction, and view. The visualization abstraction can be compared to the geometry that is generated by the mapping stage in the Data State model. In the model described in the next chapter, the `Visualization Abstraction` concept is reused to describe what kinds of geometries mapper operators can generate. Additionally, the concept is reused in queries as a mechanism to specify what visualizations users are requesting.

The underlying premise of the data state model is that users can describe different visualization techniques using the model [13] which serves as a sort of interlingua that fosters comparison among different visualization techniques. However, it may be challenging for users to map these data-state transition diagrams to operator implementations, due to the lack of a more algorithmic perspective; generally, authors of visualization techniques describe their methods from an algorithmic viewpoint rather than a stage transitional perspective. In the diagram, arrows refer to some transformation step and so the most that can be inferred about a given operator (i.e., arrow) is what states it plays a role in transitioning. In practice, most visualization toolkits are implemented using the Data Flow Model, however there are two systems that actually implement the data state model, Visualization Spreadsheet [16] and Prefuse [36].

The Data Flow and Data State model differ only in perspective; they each describe the same fundamental process of data transformations from data from raw to a viewable states. In fact, the two models have been shown to be duals of each other [14] and provide equivalent expressiveness. Note that expressiveness in this context is different from the *expressiveness* defined by Mackinlay in [51]. Mackinlay's definition of expressiveness refers to how well a visual representation captures the relationships or properties of the data of interest while in this context expressiveness can only be considered when comparing visualization models. For example, it can be stated that some visualization model is as-expressive-as another model if for the same inputs, they both generate the same visualization [14].

In contrast to the Data Flow and Data State models, the functional transformation of data to visualization can be atomic from the perspective of the user, rather than a sequence of operators. The Tableaux toolkit presented in Chapter 2 was founded on this kind of atomic model and supports a set of rules that can directly map a specific kind of data to some visualization (e.g., Gantt Chart, line graphs, and scatter plots). The mapping rules are based on the characteristics of the data, and these characteristics are further elaborated on in the next section.

3.2 Data-Centric Models

This class of models places emphasis on characterizing data and information in terms of its semantics and structure rather than the visualization transformation process. Some of the earliest work in this area was associated with A Presentation Tool (APT) (later Tableaux), presented in Chapter 2, that characterized data in such as a way that it could be automatically mapped to effective two dimensional visualizations. In general, most of the work on data centric models are applied to relational data or information. For example, Roth [67] proposed a set of information characteristics for relational data that:

- describe the *expressiveness* of different graphics
- order graphical techniques based on *effectiveness*
- determine how information can be integrated within displays

In this characterization, the type of the data, which includes *quantitative*, *ordinal*, and *nominal*, can be used to configure visualizations that express the specific nature of the data. Knowing that a data-set is quantitative tells a system that the elements can be conveyed effectively by a graphical technique with a quantitatively varying visual dimensions, such as position along an axis or angles in a pie chart. Another dimension considered by Roth is whether data is *coordinate* or *amount* based, where coordinates are used to position *amounts* on the screen and thus provide a frame-of-reference that is an effective communicator.

Additionally, these models also rely on the the *relational structure* of the data (e.g., functional dependence). For example, consider a relation describing the cost of specific models of cars at some dealership. The *cost* relation is functionally dependent because every car model has a single cost. Knowing that some data is functionally dependent can aid in choosing an effective visualization, for example a bar chart. Alternatively, a bar chart would not be an effective visualization for data that is not functionally dependent because of the challenges associated with painting multiple bars for a single domain value.

Zhou also presents multiple characteristics for describing information that overlap, generalize, and expand on the characterizations from Roth. In terms of added characteristics, Zhou considers Roth's characteristics as intrinsic to the data whereas she includes more non-intrinsic task driven and user-driven characteristics, such as *data role* and *data sense*. The data role dimension is used to characterize the *functional role* each piece of information plays in a visual presentation context. For example, data roles can support visual tasks including: *categorization*, *clustering*, *identification*, and *distinguishing*. Data sense on the other hand is used to characterize data in terms of the visual preferences of the user, for example specifying that some kind of data should be displayed as a label, list, plot, symbol, or portrait. This user-centric aspect of the data characterization is key in distinguishing Zhou's work from Roth's.

3.3 User-Centric Models (Taxonomies)

Whereas transform and data-centric models have generally been used to structure and design visualization toolkits, taxonomies can be used to help users compare among different techniques and understand when to use a particular technique, although there are many cases when taxonomies serve as the foundation of visualization systems as well [16, 52, 36].

The Data State model, in addition to providing structure to the visualization execution space, has also been used to taxonomize a wide range of visualization techniques [13]. The four states defined in the Data State model can serve as a canonical form for representing

different visualization techniques; a particular technique is described in terms of the fixed set of four data states. Using this approach, users can identify where visualization techniques overlap and differ by comparing the different data states associated with the techniques.

Other work by Tory [76] proposes taxonomies based on *design models* or the set of assumptions about data that are encoded into visualization techniques. For example, a visualization operator designer might assume that its input data is ordinal, that data can be interpolated, or that triplets of consecutive numbers represent 3D spatial directions. This information is often not directly represented by data values but is *interpreted* by the algorithm as such. In the next chapter, the proposed model is shown to rely heavily on **T**ype[Format] to explicitly convey the data assumptions an operator has about its inputs. Tory also proposes a *user model*, that provides a means to characterize data but at a higher level and abstract level of detail than Roth and Zhou. A user model is an abstraction based on the users understanding or perception of the input data. For example, consider Computed Tomography (CT) scans that take discrete image slices of a subject. Although the data is discrete, radiologists perceive this data as continuous and therefore opt to view the data using algorithms supporting continuous visualizations such as volume renderings. Tory uses the design models of algorithms to taxonomize different visualization techniques, which is shown to be more consistent than the traditional classification scheme of *information* and *scientific* based visualizations.

The final taxonomy that serves to classify techniques is the Domino scheme proposed by Brodlie [7]. The domino notation provides another reference model from which to compare visualizations, which is specified in a concise *domino-like* form where the top portion of the domino specifies the dependent variable (or value in terms of Zhou) and the bottom portion of the domino specifies the independent variable (or coordinate in terms of Zhou). Similarly to Roth and Zhou, the domino notation provides a set of data types for the variables, including: ordinal, nominal, and quantitative in which *quantitative* is further specialized by the set of real numbers and integers. In essence, the domino notation allows users to specify some of the data characterizations identified by Roth and Zhou in a more

concise form using a set of symbols.

A taxonomy designed by Shneiderman [73] was conceived with a specific purpose of classifying visualization techniques based on the data it visualizes as well as the visualization *task* the visualization supports. Thus this taxonomy not only classifies different visualizations, but also guides users towards a visualization based on their analysis needs. In terms of data characterization, Shneiderman's classification is rather broad in comparison to Roth's and Zhou's and only considers certain data characteristics such as dimensionality or whether the data is temporal, hierarchical (tree based), or network (node and link) based. The set of tasks on the other hand are much more well defined and include: filtering, zooming, details-on-demand, relating, history, and extraction. Shneiderman's taxonomy is structured as a two-dimensional matrix, where the different types of data lie on one axis and the different task lie the other axis. The specific visualization techniques populate the matrix grid cells. Given a visualization scenario, described in terms of the data being visualized and a task to be performed on the data, users can easily identify a visualization that supports their analysis scenario.

3.4 Knowledge Driven Models

Perhaps because of advances in the semantic Web, a renewed interest in understanding how knowledge plays into the visualization process has arisen. In particular, Min et al. [12] have described visualization models that explicitly highlight how users' explicit and tacit knowledge affect or steer the visualization generation process in terms of parameter settings. In this context, users' knowledge pertains to choosing effective orientations/positions of the data as well as setting color transfer functions, which can have dire effects on the final visual artifact. To compensate for incomplete knowledge, the paper describes a *knowledge-assisted visualization* model where automated reasoning capabilities may help to alleviate users' naivety about configuring visualization processes. This assisted knowledge may come explicitly from other users based on their epiphanies associated with trial-and-

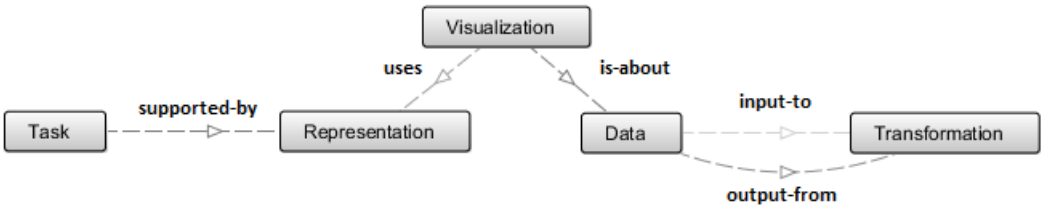


Figure 3.2: Duke and Brodlie Visualization Ontology

error visualization cycles. Recall that in Chapter 1, visualization generation is an iterative process, where during each new cycle users refine their requirements and adjust how the next visualization is generated. The knowledge driven models propose to save the settings and configurations that users learn though each cycle and make this available for other users with similar visualization needs.

3.5 Hybrid Models

Some models have been devised to provide a unified perspective of the visualization generation process that includes both the users' tasks as well as the transformational processing the occurs. Duke and Brodlie [23] proposed a visualization ontology that was initially sketched in a workshop report [6] presented in Figure 3.2. In this work, they model visualization generation from the high-level concepts: `tasks`, `representation`, `data`, and `transformation`. The concept $\widehat{\text{V}}$isualization is an abstract concept that is `about` data and is manifested-as or `uses` a representation. Additionally, the work proposes how such an ontology might be segmented according to different concerns: World of Representation, World of Users, World of Data, and World of Techniques. The goal of Duke and Brodlie is to provide a uniform perspective on visualization that can be used to foster interoperability among different scientific disciplines. The proposed model presented in the next chapter is most closely related to the approach by Duke and Brodlie, except that the proposed model is more detailed in terms of processing and does not include the notion of task.

Table 3.2: Model Granularity: 0 = no modeling, 1 = coarse, 2 = fine

Model	**Operators**	**Params**	**Data**	**Tasks**	**Vis.**
Data State	1	0	1	0	0
Data Flow	2	0	1	0	0
Lattice	0	0	2	0	2
Mackinlay	0	2	2	0	2
Zhou	0	2	2	0	1
Task-by-Data	0	0	1	2	1
Domino	0	0	1	0	1
Duke Ontology	1	0	1	1	1

3.6 Comparison

Table 3.2 summarizes the above models in terms of how much detail each designates for the following visualization apsects: operators, params, data, tasks, visualization (Vis). A score of 0 indicates that this perspective of visualization is not modeled, 1 indicates a course level of modeling, and 2 indicates that the model provides a finer level of detail. For example, the Data Flow model has the finest characterization of different visualization operators and so it scores a 2 for this aspect. However, Data Flow does not capture the intent of user or tasks and so it scores a 0 in *Tasks*.

Chapter 4

An MVE-Centric Visualization Model

Chapter 3 presented different approaches to modeling visualization generation from different perspectives: transformations, data characterizations, and user tasks. This chapter presents a new model, known as the VisKo model, which is specifically focused on describing MVE based operators in terms that can be used by our proposed semantic layer to answer visualization queries. In order for the layer to automatically compose pipelines equivalent to queries, it must be equipped with the same kinds of knowledge users employ when manually constructing pipelines. We demonstrate through examples that the VisKo model captures all these concerns in a form that can be processed by reasoning and search algorithms presented in Chapter 5. The following is the list of concepts composing the VisKo model that are elaborated on in later sections.

- format converters, data type transformers, and mappers
- visualization abstractions and viewers
- formats and types
- operator parameter sets

4.1 Model

Chapter 3 included a description of the visualization transformational spaces defined by the Data Flow and Data State models. In particular, the Data Flow model defined a

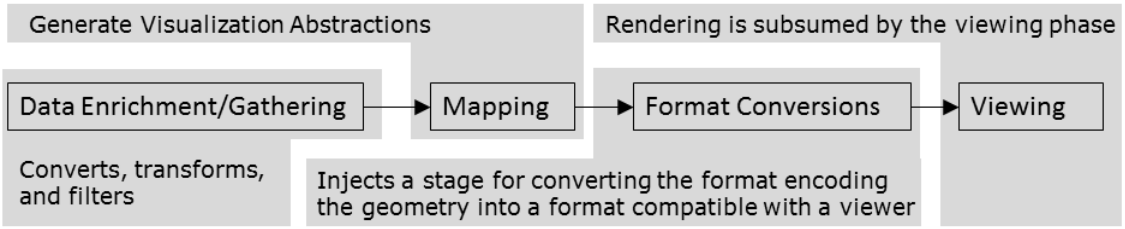


Figure 4.1: VisKo Centric Model

pipeline space decomposed into three major stages: data gathering/enhancement, mapping, and rendering. The Data State model defined four distinct states of data as they flow through visualization pipelines, including a *Visualization Abstraction*. The VisKo model is an extension of Data Flow and infuses the concept of `Visualization Abstraction` from Data State. Figure 4.1 presents the high level concepts of the VisKo model: data gathering/enrichment, mapping, converting, and viewing. Data enrichment is borrowed from Data Flow and includes filtering, converting, and transforming. Additionally, the mapping stage is also borrowed from Data Flow, except that the VisKo model specifies that mappers generate `Visualization Abstractions`

The VisKo model also diverges from Data Flow after the mapping stage, where another round of format conversions is introduced. These conversions may be necessary before the `Visualization Abstraction` geometry can be consumed by a `Viewer`. Although this conversion stage may be implicit in Data Flow, the VisKo model explicitly defines this conversion stage. In contrast to data type transformations, format conversions preserve the type of data and only change the file encoding (i.e. format). This preservation is required because the type of the geometry output from a mapper must be maintained. Without this restriction, the geometry could potentially be transformed into another geometric form, in which case the function served by the mapper would be lost. Imagine a pipeline that maps to a set of isolines but then the isolines are transformed to a point set, negating the functionality provided by the isoline mapper. These kinds of sequences are prevented in the VisKo model.

Additionally, the notion of rendering is subsumed by the more general activity of viewing. As presented in Chapter 2, `Viewers` display visualizations onto the screen, so rendering (i.e., transforming data into pixels) is an essential component of the viewing process, although viewers may support a number of additional operations to manipulate visual properties such as color, size, magnification, and orientation. The VisKo model, however, does not capture these viewer-enabled operations.

Finally, although the pipelines defined by Data Flow and Data State can take on a variety of structural forms, including network flows that generate *mash-ups*, the VisKo model only considers linear pipeline sequences that ingest single data sources and generate single visualizations, such as Pipeline A in Figure 4.2. A consequence of our linear model is that *aggregation* operators are not considered in the VisKo model, in contrast to the other models. Pipeline B includes data aggregation that mashes-up two different datasets during the pre-mapping phase, for example creating a data set containing fatal car accidents and intoxicated drivers that can then be presented holistically as a scatter plot. In contrast, Pipeline C aggregates two different visualizations (i.e., post mapping geometries), each composed from distinct datasets, into a single composite visualization. For example, overlaying a partially transparent precipitation raster map over a digital elevation map in order to see if there exists a relationship between precipitation and elevation. Finally, Pipeline D presents a structure where multiple distinct visualizations from the same data source are aggregated into a single composite view. For example, an isosurfaces rendering embedded within a partially transparent volume rendering of the same dataset, allowing for a simultaneous view of both the discrete and continuous nature of the data.

4.2 Operator Types

As was described in Chapter 2, operators serve as building blocks from which pipelines are constructed in MVE based toolkits. Operators, although wide-ranging in functionality, can usually be classified in terms of the role they play in the visualization generation process.

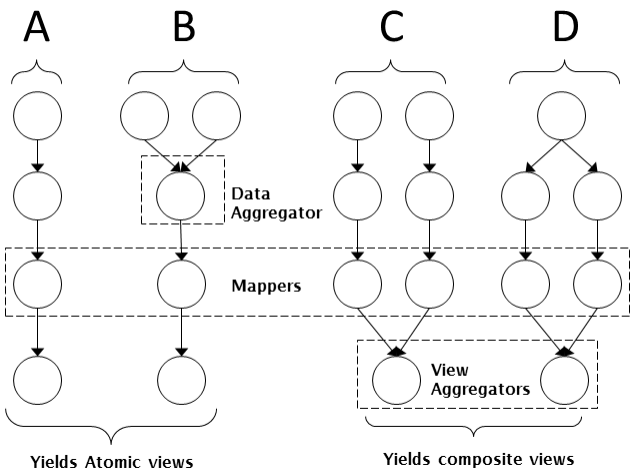


Figure 4.2: Visualization Pipeline Types

These categorizations are important because they are used to define a pipeline composition grammar described later in this chapter. Past work, however, has either provided very fine grained taxonomies of interactive operations [18] or very high level classifications of operators that may not directly map to executable operator supported by MVEs. For example, the granularity supported by the Data Flow model stages may be inadequate to describe the complex format and data type transformations that occur within open environments such as the Web, where data resides in many different types and formats. In Data Flow, filtering, converting, and data transformations have been abstracted into a single *data gathering/enriching* stage and so mapping this conglomerate stage to a specific operator would be challenging.

The VisKo model therefore expands the abstract Data Flow stages with more specific operators that can be found in most common visualization toolkits. Figure 4.3 provides a classification scheme that is closer to the level of MVE-based operators. In the figure, the Data Flow model stages were renamed to reflect a particular type of operator, rather than a stage, and are specialized by more specific operators. The arrows in the figure refer to the `is-specialization-of relationship`. For example, `Operator` represents an abstract classification from which more specific operators specialize. Additionally, `Converters`,

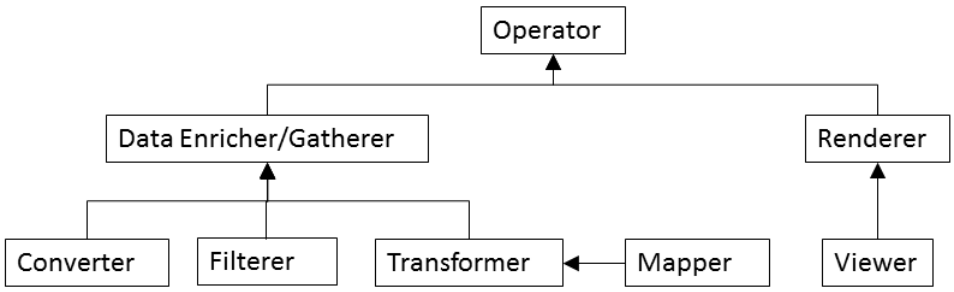


Figure 4.3: VisKo Operator Classification

`Transformers` and `Filters` specialize the general notion of gatherers and enrichers. Note that the VisKo model excludes the notion of an aggregator due to the linear pipeline restriction.

4.2.1 Data Enhancers/Enrichers

Format conversion is commonplace in visualization and scientific data processing in general, especially when data is being shared between different communities that each have a preferred storage format. The Postscript to PDF operator introduced in Chapter 2 belongs to the `Converter` class. In addition to format conversion, data type transformations, supported by `Transformer` operators, are also necessary when the underlying structure of data must be modified. Examples of data type transformations include:

- creating a uniform grid from unstructured point data
- reducing the dimensionality of data, for example transforming a three dimensional grid into a two dimensional grid as demonstrated in the VTK based examples of Chapter 2
- transforming records containing spatial fields into geometric points
- transforming a set of geometric points into a wire frame

Data filtering may also be required in cases when a user wants only a subset (i.e., sample) of some data to be visualized, perhaps for performance reasons or when only considering

data that falls within some *acceptable* range of values. Operators that filter datasets are referred to as `Filters`.

4.2.2 Mappers

Mapping refers to the process extracting geometric and presentation-based information from data. In the Data Flow model, the mapping stage is the most critical component in the pipeline because it is responsible for translating the data into the graphical language (e.g., geometry) that composes the visualization. In essence, `Mappers` are the only operators exclusive to visualization, where `Converters`, `Transformers` and `Filters` can exist in both visualization and non-visualization specific processing. The processes of mapping is so key to visualization, that we can classify pipeline operators according to whether they operate in the pre-mapping, mapping, or post-mappering. We show later in this chapter that the mapping concept was at an adequate level of abstraction in the Data Flow model to support automatic pipeline generation, and so it is not elaborated.

4.2.3 Viewers

`Viewers` are software modules that display visualizations on the screen and therefore serve as the final operator of pipelines. `Viewers` provide an interface between visualizations and humans who are interpreting the image. The generic functionality of *rendering*, defined in the Data Flow model, is specialized by the concept of `Viewers`, which can include visualization specific software such as ParaView [11] or more generic software such as Adobe Portable Document Viewer (PDF). Because `Viewers` present visualizations onto the screen, they must inherently support rendering, but also may include other interactive operations, such as magnification, rotation, color-settings, and filtering (e.g., cropping). Currently, the VisKo model does not include these kinds of interactive operations and so `Viewers` only display.

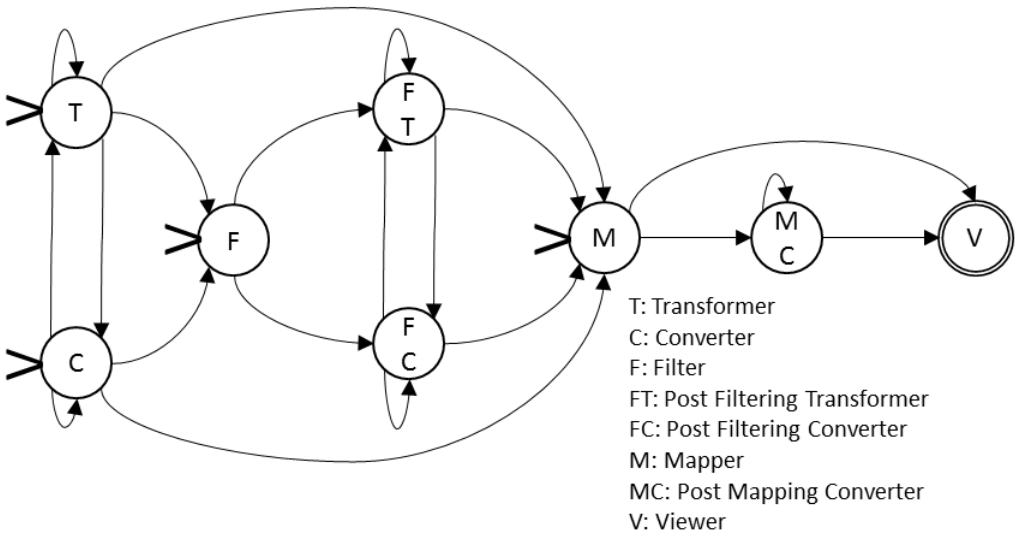


Figure 4.4: Visualization Pipeline Types

4.3 Grammar

A pipeline specification can be considered a *sentence* that describes some visualization in terms of operator executions, and therefore has a set of rules for identifying valid and non-valid sentences (i.e., valid and non valid pipelines). In this pipeline language, operators types serve as an alphabet, therefore pipeline sentences can only consist of: `Converters`, `Transformers`, `Mappers`, `Filters`, and `Viewers`. The rules that define valid pipeline sentences are collectively referred to as the VisKo pipeline grammar. Rather than enumerating the VisKo grammar rules in natural language, they are presented as an automata shown in Figure 4.4.

The simplest pipeline consists of a two-operator sequence containing only a `Mapper` followed by a `Viewer`; all pipelines must contain at least these two operators. In these cases, the input data was already in a form ingestable by a `Mapper` and the resultant visualization geometry could be directly consumed by a `Viewer`. More complicated forms of pipelines arise when the data cannot be directly processed by a `Mapper`, and must undergo rounds of conversions, transformations, and filtering. Similarly, on the post-mapping side of processes, the output geometries may undergo arbitrary rounds of format conversions before

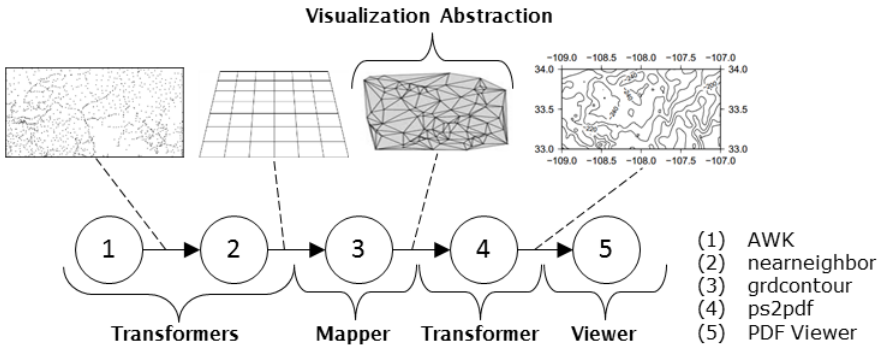


Figure 4.5: GMT Contouring Pipeline outlined using the VisKo Model

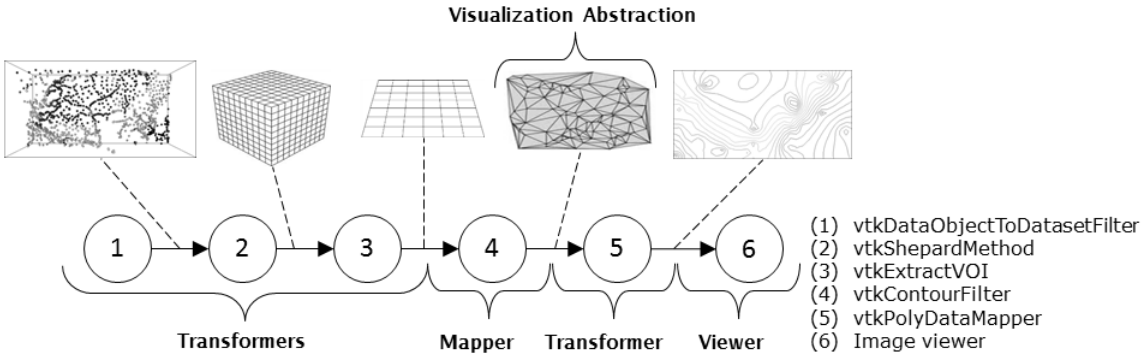


Figure 4.6: VTK Contouring Pipeline outlined using the VisKo Model

landing in a form that can be ingested by a `Viewer`. Note that all pipelines must conclude with a `Viewer`, which supports the interface between humans and the visualizations. Another rule specific to the VisKo language specifies that only one filtering operator may be injected into a pipeline. This restriction greatly reduces the pipeline search space presented in the next chapter and avoids pipelines consisting of unnecessarily long chains of redundant filtering. Figures 4.5 and 4.6 use the VisKo model to diagram our GMT and VTK contour map pipelines presented in Chapter 2. Emphasis is placed on the classification of the operators as well as on the intermediate data that flows through the pipelines.

4.4 Ontology

The operator classes described in Section 4.2 are described in OWL2 [38] and serve to as a formal language from which to describe (i.e., classify) executable toolkit operators. The resultant ontology, known as VisKo (to be distinguished from the VisKo model), is decomposed into three sub-ontologies: (1) *VisKo-View*, (2) *VisKo-Operator*, and (3) *VisKo-Service*, which specialize in defining:

(1) the kinds of `Visualizations Abstractions` that can be generated

(2) the `Operators` that provide tranformational, conversion-based, filtering, mapping, and viewing facilities

(3) the services that implement the `Operators`

The separation of VisKo-View from VisKo-Operator is similar to the approach described by Duke and Brodlie (see Chapter 3), which proposes to segment an ontology according to a: "World of Representations", "World of Users", "World of Data", and "World of Techniques". The VisKo ontology excludes "World of Users", but "World of Representation" can be compared with Visko-View while "World of Data and Techniques" can be compared with VisKo-Operator. The *VisKo* prefix is derived from *visualization knowledge* and serves as the name for the work presented in this dissertation (e.g., the model, ontology, and query answering approach).

4.4.1 VisKo-View

Visualization toolkits are typically specialized for specific domains, including scientific or information based visualizations. Therefore, the kinds of visualizations supported is a key consideration when choosing an toolkit to serve as a platform for a visualization application. Scientists will want to know if the toolkit supports scientific visualizations such as volume renderings [21], isolines, isosurfaces [63], and glyphs or information-based visualizations such as bar charts, networks, and tree maps [72]. In addition to the kinds of visualization

abstractions supported, knowing whether the abstractions are projected in two or three-dimensional space is also important, so that users may also understand the kinds of spatial orientations supported. For example, being able to state that a two-dimensional contour map is projected in three-dimensional space may help users understand that the contour map can be oriented about X, Y and Z axes. VisKo-View therefore was conceived to allow for the expression of different kinds of `Visualization Abstractions`, considering

(1) whether the abstractions are scientific or information based
(2) which geometry (or data structure in the case of information visualization) the abstractions are constructed from (which includes dimensionality)
(3) the dimensional space into which the abstractions are projected

To address consideration (1), VisKo-View includes the concept `Visualization Abstraction`, presented in Figure 4.7. The directed arcs within the greyed-out regions indicate a superclass relationship between two node concepts. In this figure, `Visualization Abstraction` is specialized (i.e., the inverse of the superclass relationship) by `Visualization Abstraction_Geometry` and `Visualization Abstraction_DataStructure`, to represent scientific and information based visualizations respectively. These two different abstractions are further refined according to what dimensional space the visualizations are projected in, which in turn addresses consideration (3).

Therefore, in isolation, VisKo-View can only express whether a visualization is scientific or information based as well as the dimensional space into which the visualizations are projected. To further describe the structure of the visualizations (i.e., underlying geometric or data structure), VisKo-View imports an ontology of geometries provided by the Federation of Earth Science Information Partners (ESIP) [25], thus addressing concern (3). A fragment of this ESIP ontology is presented in the figure and highlights some of the three dimensional geometries supported. In the ESIP ontology, the dimensionality of the geometry is not explicitly stated (e.g., volumes are three-dimensional) because the dimensionality of these objects is inherent within the mathematical specification of the geometry. In con-

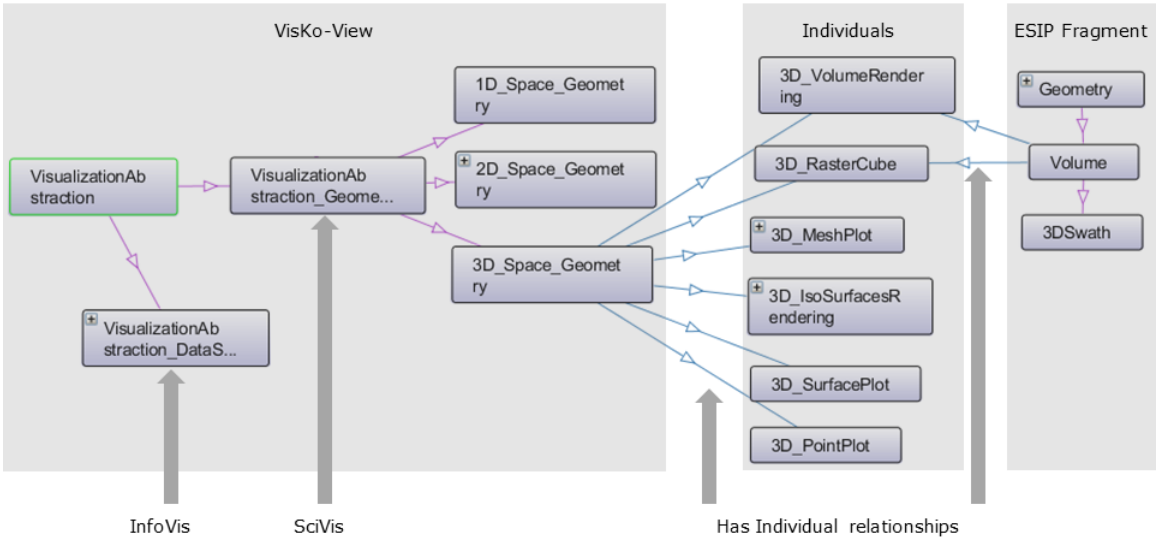


Figure 4.7: VisKo-View and ESIP Ontologies used in conjunction to describe a set of three dimensional visualization abstractions

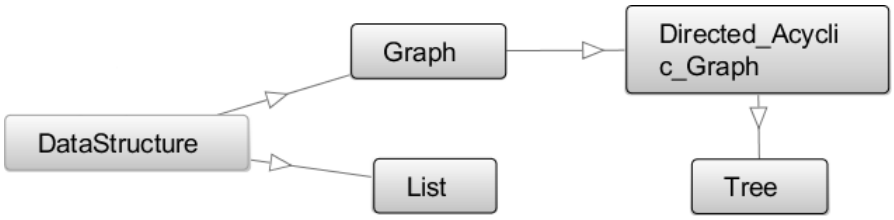


Figure 4.8: VisKo-View Data Structures fragment

junction, the VisKo-View and ESIP ontologies provide the expressive power necessary to fully describe visualizations, such as the three dimensional scientific visualizations tagged as `Individuals` in the figure. To support information based visualizations (not shown in the figure), VisKo-View also imports a small preliminary and extendable data structure ontology presented in Figure 4.8.

4.4.2 VisKo-Operators

The VisKo ontology defines different operator classes presented in the VisKo model above: format `Converters`, data type `Transformers`, `Filters`, `Mappers`, and `Viewers`. Figure

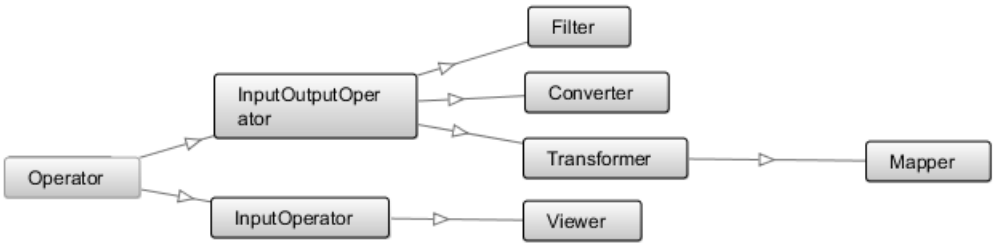


Figure 4.9: VisKo-Operator

4.9 presents a set of OWL classes capturing these operators, which are grouped into two fundamental categories, those operators that have inputs and outputs and those operators that have only inputs. In the current VisKo ontology, `Viewers` are the only operator that do not output data or information and so are classified as an input only operator; `Viewers` are always the final operator specified in pipelines. Although `Viewers` do output visualizations, they are consumed by a human and further mental processing is carried out to analyze and interpret the presented visual, however, this mental processing is not captured in the VisKo model.

The input and output interfaces of operators are described in terms of **T**ype[Format] couples. Therefore, instances of InputOutput operator must have four properties asserted: (1) `hasInputFormat`, (2) `hasOutputFormat`, (3) `hasInputType`, and (4) `hasOutputType`. In OWL2, these restrictions can be specified by equating the `InputOutputOperator` to some anonymous class, where all instances contained in the anonymous class must have the four properties asserted. This OWL equivalency is specified using the Manchester Syntax [39] below. Additionally, the range of the `hasInputFormat` and `hasOutputFormat` is restricted to only Proof Markup Language (PML) `Formats` [57].

```
1 Class: InputOutputOperator
2     EquivalentTo:
3         hasOutputFormat some Format,
4         hasInputFormat some Format,
5         hasOutputType some Thing,
6         hasInputType some Thing
7     SubClassOf:
8         Operator
```

Table 4.1: Operator Classification Rules

Operator Type	**Type Condition**	**Format Condition**
is `Filter`	if input.type == output.type	and if input.format == output.format
is `Converter`	if input.type == output.type	and if input.format != output.format
is `Transformer`	if input.type != output.type	inconsequential

When describing a toolkit operator, there are a set of rules that guide the classification process. Table 4.1 presents the rules for deciding how toolkit operators should be classified based on the characterization of their input and output data. `Filters` are those operators that preserve both the input type and format in their processing. In essence, `Filters` remove or *subset* data while preserving both the type and format. Examples would be a sampling method or a subsetter that remove data points that are within a given range. `Converters` change the format of some data while preserving the type or structure. An example is converting between a two-dimensional grid in ESRI grid format to netCDF or HDF. In this case, the grid must be maintained in the resulting netCDF file to be considered a format conversion. `Transformers` change the type or structure of the data, which may or may not also include converting the underlying format. For example, gridding algorithms, such as minimum curvature or nearneighbor, are considered `Transformers` because they convert their irregularly distributed two-dimensional point data into two-dimensional grids.

`Mappers` are a special case of `Transformers` and specify what `Visualization Abstraction` they are responsible for generating, as shown in the following `Mapper` class definition.

```
1 Class: Mapper
2     EquivalentTo:
3         mapsTo some VisualizationAbstraction
4     SubClassOf:
5         Transformer
```

`Viewers`, as described above, are responsible for presenting a generated abstraction and possibly rendering the abstraction onto the screen.

4.4.3 Formats and Types

The VisKo ontology allows for arbitrary extensions to PML `Formats` and `Types` and places no restrictions on their pairings. To prime the ontology with a well known set of `Formats`, the published Internet Assigned Numbers Authority (IANA) [60] MIME sub types were scraped from the Web and encoded as instances of `PML2:Format`, ignoring the MIME super types they were associated with. Although IANA contains a large set of over one-thousand formats, users are still permitted to annotate the input and output operator descriptions using their own custom `Formats`. The set of `Types` is also extendable and users are free to import their own ontology of `Types` and pair them with any `Format`. Due to this freedom, users are able to define their own unique data characterizations (**T**ype[Format]) from which to describe their operator interfaces.

Because the VisKo ontology is encoded in OWL, users have the freedom of constructing hierarchies of types, for example specifying that `Time-Series-Data` is a subclass of `Array`. Similarly to object oriented languages, these types exhibit polymorphic behavior and therefore any subclass can be substituted for a super class where appropriate. Figure 4.10 highlights the use of polymorphism when reasoning about the set of types that can legally serve as inputs to **O**perator 1, abbreviated as **O**p 1. In this case, a user asserted that **O**p 1 has an input of type `Array`. However, since `Time-Series-Data` is a subclass of `Array`, owl description logic reasoners can infer that any data of Type `Time-Series-Data` can also be considered as a valid input to **O**p 1.

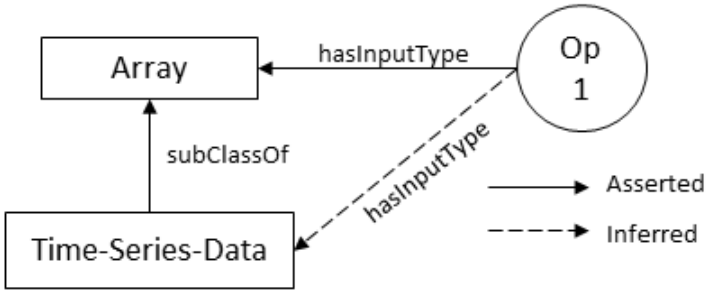


Figure 4.10: Polymorphic Behavior of Types

`Type` and `Format` specifications serve a unique role in the VisKo ontology. They ensure that an assembled pipelines are executable, meaning that the output data requirements of some operator matches, or is subsumed, by the data requirements of the next sequential operator in the pipe. Past visualization systems [50, 29, 80] have aimed at mitigating the challenges associated with data type and format match-up by supporting a common data model, which serves as a sort of *inter-format* that can be processed by large sets of operators within some toolkit. Users of these systems can focus less on mitigating data requirements between different operators and focus more the key operators such as the `Mapper`. However, progress can fall back when the toolkit does not support the right *data reader* that can transform the input source data into the toolkit's common form, leaving users to manually program custom data readers.

The controlled environment required to support common data formats is not feasible in more open environments such as the Web, where users may want to construct hybrid pipelines whose web-based operators span across many toolkits, each with their own set of supported types and formats. In practice there are over thousand different data format and type combinations registered with the Internet Assigned Numbers Authority (IANA) MIME website. Although only small subsets of of these data types and formats need to be understood to support visualization in some domain, learning the right subset of needed formats can be challenging for non-experts. Visualization on the Web provides the challenges that the proposed visualization query answering system addresses, which is to limit the amount of information users need to specify visualizations.

4.4.4 VisKo-Services

Only input and output data requirements can be described using the VisKo-Operator ontology. However, the Visko-Service ontology represented in Figure 4.11 augments these operator descriptions with information about *where* and *how* to invoke services that implement these interfaces. The execution details are described in terms of the Web Ontology Language Service (OWL-S) ontology [55], which defines a comprehensive set of classes and

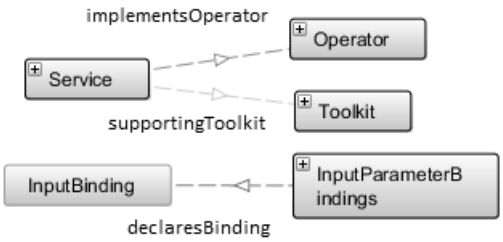


Figure 4.11: VisKo-Service

properties needed for describing executable Web services. Therefore, all `operators` must be exposed as Web Service Description Language (WSDL) services [17] in order to be annotated by the OWL-S vocabulary and thus appended to an `Operator` description. The `implementsOperator` property serves as the relationship that associates OWL-S service descriptions with VisKo `Operators`.

There are benefits to keeping invocation details about operators separate from the abstract behavioral descriptions. Consider an `Operator` named `gridder`, which ingests **2**D-XYZ-Point-Data[ASCII] and generates **2**D-Grid[netCDF]. In GMT, there are two implementations of this gridding functionality, **g**rdContour.exe and **s**urface.exe, which each are implemented using a different algorithm. Using VisKo-Service, both these gridding implementations can be associated `gridder` via the `implementsOperator` relationship, thus indicating that both **g**rdContour.exe and **s**urface.exe can fulfill the abstract operator requirements.

The `InputParameterBindings` class represents a list of parameter bindings (i.e., a parameter and associated input value). Typically, operators composing visualization pipelines are heavily parameterized and the `InputParameterBindings` allows us to specify a set of hard-coded values that can be fed into the operators to be used as defaults. These hard-coded bindings are associated with a data type, and so by specifying a particular `Type` in a visualization query, the answering system can retrieve these bindings and set the parameters of resulting pipelines. Other concepts defined in Figure 4.11 include `Toolkit`, which represents the different visualization toolkits such as GMT, NCL, and VTK. `Services`

can be associated with a `Toolkit` and this information is useful if scientists want to filter pipelines based on their supporting toolkit.

4.5 Ontology in Use

The VisKo ontology provides a vocabulary from which to describe the functions provided by MVE based operators, such as those operators comprising the GMT and VKT pipeline in Chapter 2. In terms of the ontology, users will describe the behavior of operators using the Resource Document Framework (RDF) [53]. RDF descriptions are composed of a set of statements structured as *triples*, or subject, predicate, object (see Figure 4.12 for an example of a description of the VTK pipeline in Chapter 2). The legal values that comprise the subject, predicates, and objects are constrained by the VisKo OWL ontology. The resulting set of operator triples comprises a *visualization knowledge base* that is used by the querying answering system described in the next chapter.

4.5.1 Asserted Knowledge Base

Figure 4.12, contains a set of RDF triples that describe the operators of the VTK contour map pipeline in Chapter 2. The figure presents how the subject (i.e, `Operator`) is described in terms of the data it processes. Additionally, each `Operator` is connected to an executable service that implements the specified behavior.

The knowledge base presented in Figure 4.12 can be conceptualized as a graph. In such a graph, nodes represent **T**ype[Format] couples and edges indicate that the couple `canBeTranformed` to another couple. In order for a `canBeTransformed` edge to exist between nodes A and B, there must exist an operator that has **i**n-Type[Format] A and **o**ut-Type[Format] B. Thus the graph contains all possible combinations of data transformations, where only a subset of the paths reflect valid pipelines in terms of the VisKo grammar. This data-centric graph is just one way in which a visualization knowledge base can be perceived, as shown in Figure 4.13. In this figure, blue nodes represent data that

Subject	**Predicate**	**Object**
vtkDataObjectToDataSet	is a	Transformer
vtkDataObjectToDataSet	in-Type[Format]	XYZData[XML]
vtkDataObjectToDataSet	out-Type[Format]	PolyData[XML]
vtkDataObjectToDataSet	implementedBy	vtkDataObjectToDataSet-Service
vtkShepardMethod	is a	Transformer
vtkShepardMethod	in-Type[Format]	PolyData[XML]
vtkShepardMethod	out-Type[Format]	ImageData3D[XML]
vtkShepartMethod	implementedBy	vtkShepardMethod-Service
vtkExtractVOIXY	is a	Transformer
vtkExtractVOIXY	in-Type[Format]	ImageData3D[XML]
vtkExtractVOIXY	out-Type[Format]	ImageData2D[XML]
vtkExtractVOIXY	implementedBy	vtkExtractVOI-Service
vtkContourFilter3D	is a	Mapper
vtkContourFilter3D	in-Type[Format]	ImageData3D[XML]
vtkContourFilter3D	out-Type[Format]	PolyData[XML]
vtkContourFilter3D	implementedBy	vtkContourFilter-Service
vtkContourFilter3D	mapsTo	2DContourMap
vtkPolyDataMapper	is a	Transformer
vtkPolyDataMapper	in-Type[Format]	PolyData[XML]
vtkPolyDataMapper	out-Type[Format]	owl:Thing[JPEG]
vtkPolyDataMapper	implementedBy	vtkPolyDataMapper-Service
WebBrowser	is a	Viewer
WebBrowser	in-Type[Format]	owl:Thing[JPEG]
WebBrowser	in-Type[Format]	owl:Thing[PNG]
WebBrowser	in-Type[Format]	owl:Thing[GIF]
WebBrowser	in-Type[Format]	owl:Thing[PDF]

Figure 4.12: Knowledge Base Fragment

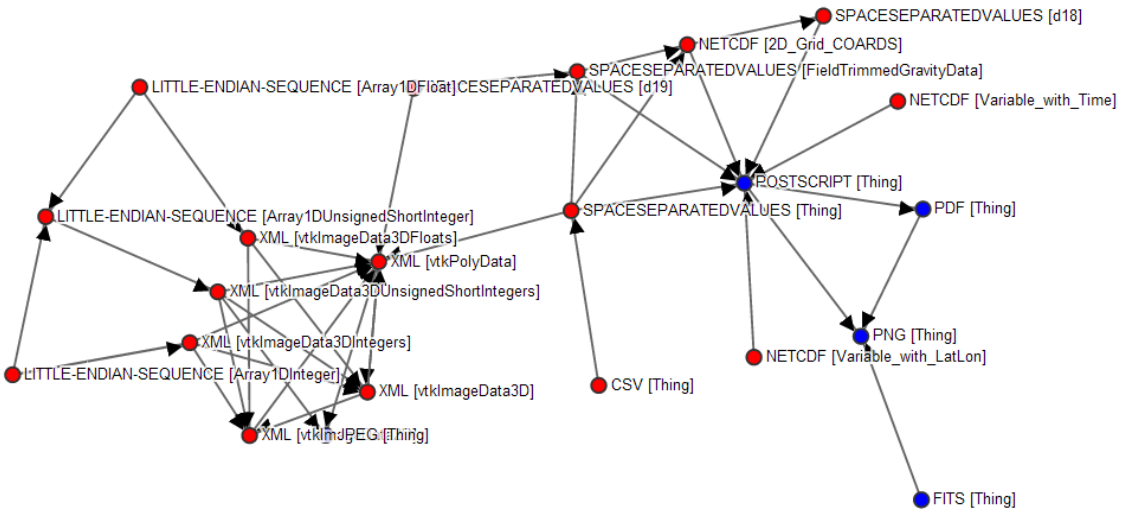


Figure 4.13: Data transformation paths (note the labels refer to Format[Type] rather than Type[Format]

can be directly ingested by some `Viewer`, and therefore cannot be transformed further. Red nodes cannot be directly ingested by some viewer and so represent data in one of the intermediate states between the value and view [15].

Similarly to how Data State model has a Data Flow dual, the data transformation graph has an operator-centric dual presented in Figure 4.14. Figure 4.14 presents the set of possible pipelines. In this figure, nodes represent different operators classified according to their function-which is specified by node color; purple nodes represent `Converters`, `Transformers`, and `Filters`, gold nodes represent `Mappers`, and white nodes represent `Viewers`. Arcs represent that the output data of some operator can be ingested by the operator with the in-going arc edge. According to the VisKo pipeline grammar, all legal pipelines end with a `Viewer` and so there are no arcs leaving any white nodes.

In addition to describing operators in terms of data requirements, the knowledge base also contains a set of the known `Visualization Abstractions`. In Figure 4.15, there are a few abstractions described and they are used to adorn the `Mappers` with information about what visualization they generate; `Mappers` reference these abstractions through the

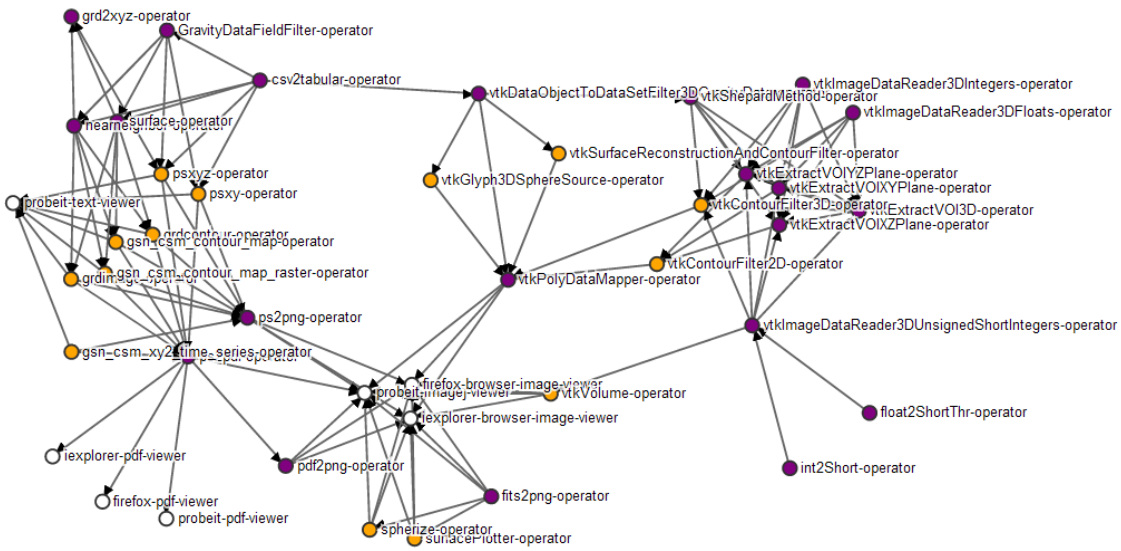


Figure 4.14: Operator paths (i.e., pipelines)

mapsTo property as shown in Figure 4.12.

4.5.2 Asserted Knowledge Base Population

In practice, users will populate the visualization knowledge base by programming against an Application Programming Interface (API), rather than composing RDF statements directly. In the future, inductive wrappers [46] will be employed that are automatically configured to translate published toolkit user manuals into VisKo RDF descriptions. However, the current process of adding VisKo RDF to a knowledge base is a manual task known as *registration*. Registration includes both the publishing of operators as web services and generation of RDF descriptions of said operators. To register an operator, users must construct a *module*, which is a component that encapsulates both (1) operator service code and (2) annotation code that generates RDF. For example, **v**tkContourFilter ingests **v**tkImageData[XML] and generates isosurfaces encoded in **v**tkPolyData[XML]. The actual service code that wraps the VTK functionality can be found in Appendix A, as well as the annotation code that describes the VTK operator in terms of the ontology. The annotation

Subject	*Predicate*	*Object*
2DContourMap	is a	VisualizationAbstraction
Isosurfaces	is a	VisualizationAbstraction
2DPointPlot	is a	VisualizationAbstraction
3DPointPlot	is a	VisualizationAbstraction
2DRasterMap	is a	VisualizationAbstraction
VolumeRendering	is a	VisualizationAbstraction
2DBarChart	is a	VisualizationAbstraction
3DBarChart	is a	VisualizationAbstraction
2DGraph	is a	VisualizationAbstraction
3DSurfacePlot	is a	VisualizationAbstraction

Figure 4.15: Partial Set of Visualization Abstractions

code, in addition to describing an operator service, also allows users to set parameter bindings. Users will use the API to associate a set of quads, in the form of (`Type`, `Service`, `Parameter`, `Value`). This quad of information effectively associates a semantic type with a service parameter binding (i.e., $ServiceParameter = Value$). Chapter 5 describes how these bindings can be used as a set of default parameter values when executing pipelines.

Assuming the user has created the module code (i.e., service and registration code), an installation infrastructure will:

1. Publish the service code into a Java Enterprise Edition (EE) servlet container [44].
2. Execute the annotation code and dump the resulting RDF into a knowledge base, perhaps made accessible through a triple store [65]

Figure 4.16 presents a data flow perspective of the registration process. In this figure, a user provides both the operator service code as well as the annotation code that describes the service. The service code is transformed into a Java EE compliant Web Application Archive (WAR) file and dumped into a Java EE servlet container-effectively publishing the code as a service. Additionally, a WSDL document describing the service endpoint, as well as the annotation code, is passed to the VisKo RDF generator. The VisKo RDF generator executes the annotation code thus generating the RDF describing the service. Additionally, the VisKo RDF generator produces OWL-S RDF based on the WSDL document and

associates the OWL-S RDF with the VisKo RDF using the `implementsOperator` property. The resulting set of RDF is stored in an knowledge base, which may be accessible by a triple store.

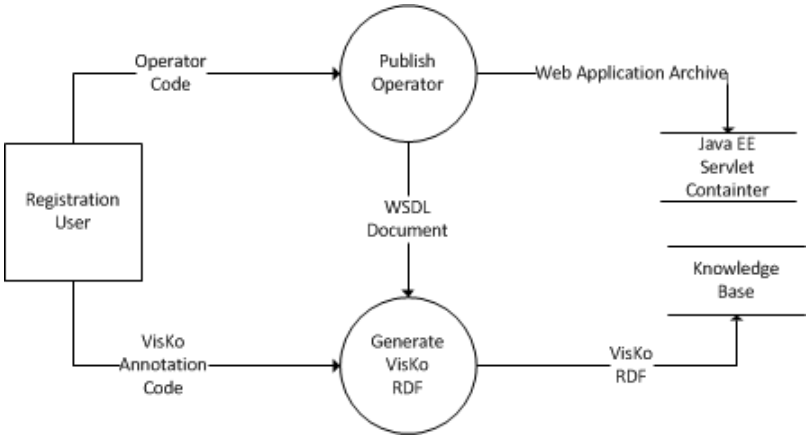


Figure 4.16: Operator Registration Data Flow

4.5.3 Inferred Knowledge Base

Given the data transformation graph, one natural task is to check whether some data X can be transformed into another data Z. Consider the velocity model query in Chapter 1, which requests that **3**DVelocityGrid[binaryFloatSequence] be visualized as a set of isosurfaces that can be viewed in a web browser. Searching through the data transformation graph, the query answering system must verify if a path exists from **3**DVelocityGrid[binaryFloatSequence] to any standard image data, such as owl:Thing[JPEG] or owl:Thing[GIF]. Any standard OWL DL reasoner, such as Pellet [74], can infer transitive OWL2 property chains and make all connections directly explicit. For example, the property chains will infer a `canBeTransformed` relationship directly between **3**DVelocityGrid[binaryFloatSequence] and **o**wl:Thing[GIF], since there is a sequence of data transformations that connect the two. Upon completion of the property chain reasoning, simple SPARQL [64] queries can be posed to check if the `canBeTransformed` relationship holds between any two kinds of data.

The Horn rules in Figure 4.17 represent these transitive OWL2 property chains in a more human friendly format than the RDF/XML format they naturally reside in. Rather than referring to the properties, `inputFormat/inputType` and `outputFormat/outputType` as they are specified in the ontology, these properties have been collapsed into: inputData and outputData. Additinally, `data` encompasses the **T**ype[Format] characterization scheme.

dataInputTo(?data1, ?A) :-	inputData(A, ?data1)
canBeTransformedTo(?data1, ?data2) :-	dataInputTo(?data1, A), outputData(A, ?data2).
canBeTransformedTo(?data1, ?data2)	is a transitive rule

Figure 4.17: Data Transformation Rules

The first rule, `dataInputTo` is the inverse property of `inputData`. `dataInputTo` serves as an antecedent to the `canBeTransformedTo` rule, which is transitive and states **d**ata1 can be transformed to **d**ata2 if there is an operator that has an input of **d**ata1 and an output of **d**ata2. Since this rule is transitive, a reasoner can infer that **d**ata1 `canBeTransformed` to another **d**ata3 though the intermediate data **d**ata2. Figure 4.18 presents a set of inferred statements after applying the rules to the asserted knowledge base in Figure 4.12.

4.6 Multiple Operator Perspectives

Often times, the behavior of a toolkit operator varies according to its input data such as **v**tkContourFilter. Feeding **v**tkContourFilter `2D-vtkImageData` results in the generation of isolines (i.e., contour lines), whereas three-dimensional versions of the image data result in the generation of isosurfaces.

To describe both of these behaviors, users can describe the same operator, [**v**tkContourFilter, from two different perspectives: the two and three-dimensional versions; there can exist a 1-to-many relationship between an operator service and its VisKo RDF descriptions. Figure 4.19 presents two different RDF descriptions for a single **v**tkContourFilter exposed as a web service named **v**tkContourFilter-service.

Subject	Predicate	Object
CSV[XYZData]	canBeTransformedTo	PolyData[XML]
CSV[XYZData]	canBeTransformedTo	ImageData3D[XML]
CSV[XYZData]	canBeTransformedTo	ImageData2D[XML]
CSV[XYZData]	canBeTransformedTo	owl:Thing[JPEG]
PolyData[XML]	canBeTransformedTo	ImageData3D[XML]
PolyData[XML]	canBeTransformedTo	ImageData2D[XML]
PolyData[XML]	canBeTransformedTo	owl:Thing[JPEG]
ImageData3D[XML]	canBeTransformedTo	ImageData2D[XML]
ImageData3D[XML]	canBeTransformedTo	owl:Thing[JPEG]
ImageData2D[XML]	canBeTransformedTo	owl:Thing[JPEG]

Figure 4.18: Inferred Knowledge Base Fragment

Subject	Predicate	Object	Perspective
vtkContourFilter2D	is a	Mapper	2D
vtkContourFilter2D	in-Type[Format]	ImageData2D[XML]	
vtkContourFilter2D	out-Type[Format]	PolyData[XML]	
vtkContourFilter2D	implementedBy	Service4	
vtkContourFilter2D	mapsTo	Isosurfaces	
vtkContourFilter3D	is a	Mapper	3D
vtkContourFilter3D	in-Type[Format]	ImageData3D[XML]	
vtkContourFilter3D	out-Type[Format]	PolyData[XML]	
vtkContourFilter3D	implementedBy	Service4	
vtkContourFilter3D	mapsTo	2DContourMap	

Figure 4.19: Two different RDF descriptions of vtkContourFilter

Chapter 5

Answering Visualization Queries

Chapter 1 introduced the notion of a semantic layer that is responsible for:

1. translating queries into equivalent toolkit pipelines
2. executing the resulting pipelines to generate visualization query results

To support the first task, the notion of *equivalence* had to first be defined in order to ensure that the automatically derived pipelines would generate the precise visualizations specified by queries. Since the structures of queries and pipelines are not directly comparable (i.e., declarative specification versus imperative), the *expressiveness* of each structure is compared. Expressiveness had been previously defined by Mackinlay [51] to provide a measure for how well a visual representation *captures* the relationships or properties of the underlying data depicted by a visualization. In the context of queries and pipelines, the required *expressiveness* metric is more akin to Chi's [14], which defines expressiveness in terms of the visualization *capabilities* of some technique. For example, some query A is as expressive as some pipeline B if both A and B specify the same the kind visualization from the same kind of data. Because pipelines are considered sentences composed of an operator alphabet described in Chapter 4, it is possible to compare the expressiveness of both specification forms: queries and pipelines. In this dissertation, if some query A is as expressive as some pipeline B, then they are regarded as being equivalent.

Provided both a definition of equivalence and a knowledge base of toolkit operators (Chapter 4), a query-to-pipeline translator is equipped with all the components necessary to derive equivalent pipelines from visualization queries. Queries provide both the source and target of a pipeline. The query translation problem is then based on searching for

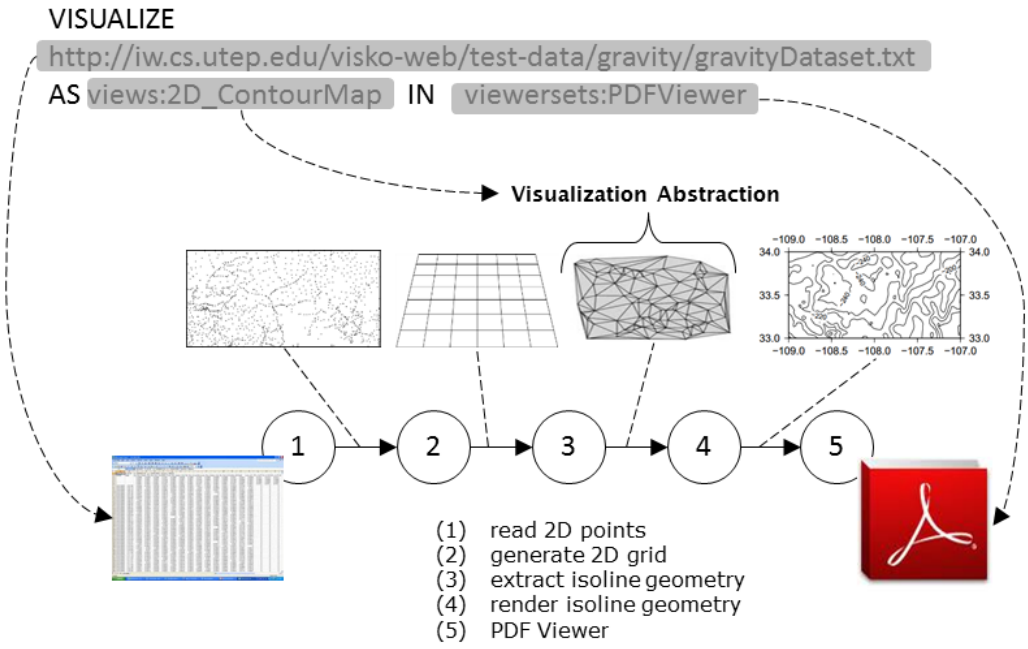


Figure 5.1: Queries contain information about pipeline sources and targets

paths of operators in the knowledge base that can transform the data from its source form into the requested visualization target specified by queries-while adhering to the pipeline structural rules described in Chapter 4.

Figure 5.1 shows a query juxtaposed with an equivalent pipeline. The figure highlights how queries provide the source and target information used by graph search algorithms to identify paths (i.e., pipelines). This chapter presents such a search algorithm and includes a brief analysis on its complexity. The chapter also describes how these derived pipelines are executed, including a description of the different approaches for setting pipeline parameters.

5.1 Query Answering Data Flow

Answering visualization queries is a three step process:

1. **Translate** a query to a set of pipeline specifications
2. **Select** a pipeline to be executed among the set of returned pipeline specifications

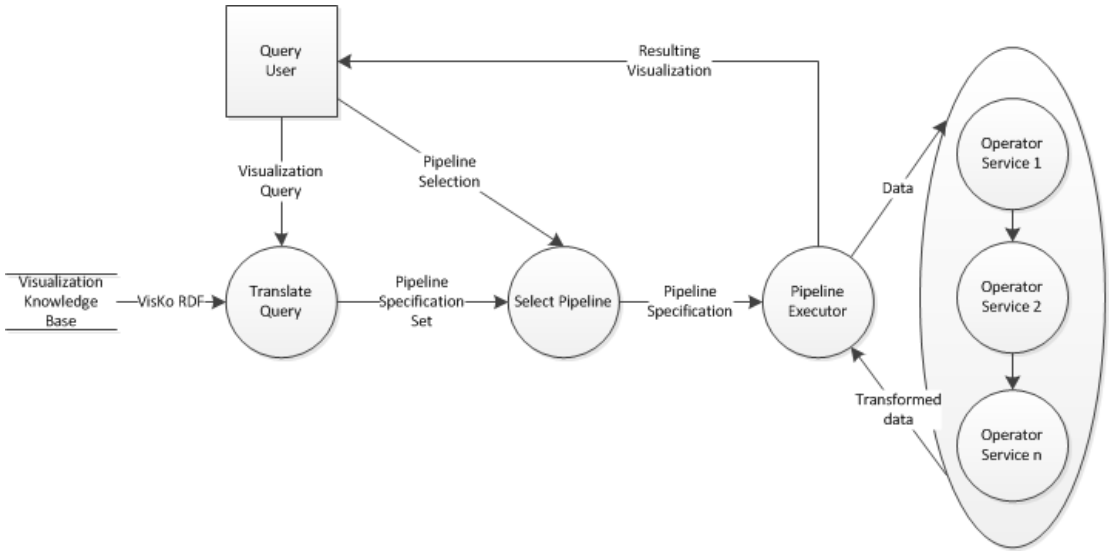


Figure 5.2: Query Answering Data Flow

3. **Execute** the chosen pipeline to generate a resultant visualization

Figure 5.2 presents a data flow perspective of the query answering process. In the figure, a visualization query is transformed into an equivalent pipeline specification set through a query translation process that relies on a knowledge base of VisKo RDF. In addition to providing a query, users are also responsible for selecting a pipeline to be executed, which is then forwarded to a pipeline executor. The executor invokes a sequence of services in the order specified by the pipeline. The final service invocation results in the visualization that satisfies a query.

5.2 Query-to-Pipeline Search Algorithm

The query-to-pipeline search algorithm is a modified depth-first search that is tailored for only traversing paths that maintain the pipeline structural rules defined by the VisKo model. Although the operator space is a graph (presented in Chapter 4), when provided a query target, the pipeline search space is viewed as a forest [68]. In this forest, each root node is some `Operator` that can ingest the input data and the leaves are the `Viewers`

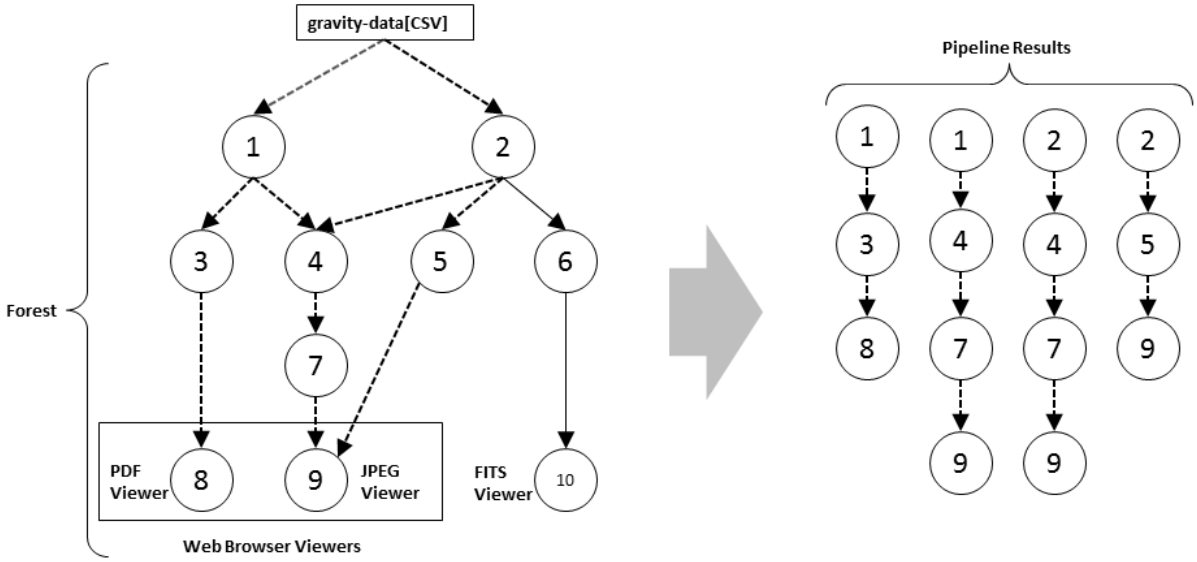


Figure 5.3: Forest Search Space to Pipeline Results

that can ingest some transformed representation of a resulting visualization. Figure 5.3 presents a possible forest search space. In this case, the input query requested the visualization of **gravity-data[CSV]** that could be viewed using a web browser. Web browsers can present standard image formats such as PNG and JPEG, as well as PDF documents, so any path that transforms the **g**ravity[CSV] into an owl:Thing[PNG], owl:Thing[JPEG], or owl:Thing[PDF], while maintaining the pipeline grammar, can be used to generate a visualization specified by the query. Note that a path that terminates with a FITS viewer is not considered an equivalent pipeline because the query in the figure requested a visualization that can be presented using a web browser, and web browsers cannot view FITS images by default. The figure also illustrates the output of the algorithm, which is a set of pipelines equivalent to the query. Note that the algorithm *pulls* apart the pipelines from the forest using the *clone* function described below.

5.2.1 Psuedo Code

The search algorithm begins by first identifying the applicable root nodes of a forest that can ingest the input data specified by the query. The procedure *root-node-search(G,data)*, shown below, identifies the set of root nodes provided the knowledge base graph of operators and the input data specified by the query. The procedure traverses through all nodes (i.e., candidate operators) and and checks whether any can ingest (i.e., `hasInputFormat` and `hasInputType`) the data specified in the query. If so, the candidate nodes are added to a list of applicable root nodes.

```
procedure root-node-search(G,data):
  roots <- empty
  for all vertices v in G.listVerticies(v) do
    if v ingests data
      roots.add(v)
```

Once the root nodes are identified, the forest search algorithm *pipeline-search-forest(G,roots)*, shown below, can be initiated. For each root node, a specialized depth-first search (i.e., *pipeline-search-tree(G,v,pipeline)*) is called by passing: (1) the knowledge base graph, (2) a root node v, and (3) an incomplete pipeline with v as the first operator in the sequence.

```
procedure pipeline-search-forest(G,roots):
  pipelines <- []
  for all roots v
    pipeline-search-tree(G,v,[v])
  return pipelines

procedure pipeline-search-tree(G,v,pipeline):
  label v as explored
  for all edges e in G.adjacentEdges(v) do
    clonedPipeline <- pipeline.clone()
    if edge e is unexplored then
      w <- G.adjacentVertex(v,e)
      clonedPipeline.add(w)
      if vertex w is unexplored and isLegal(clonedPipeline) and isTargetViewer(w) then
        pipelines.add(clonedPipeline)
      else if vertix w is unexplored and isLegal(clonedPipeline) then
        label e as a discovery edge
        recursively call DFS(G,w,clonedPipeline)
```

```
19        else
20          label e as a back edge
```

The depth first search algorithm *pipeline-search-tree* strays from the typical depth first search in a number of ways outlined below:

- Line 10: for every edge connected to the pipeline, the algorithm *clones* a copy of the pipeline, essentially enumerating every distinct path in the tree as shown in Figure 5.3.
- Line 13: adds new vertices (i.e., new discovered operators) to a cloned pipeline
- Line 14: checks if the updated pipeline generated from line 13 is legal in terms of the grammar described in Chapter 4, and if it meets a terminating condition (i.e., the current node is a target viewer). If the conditions are met, then the pipeline is added to the set of pipeline results
- Line 16: if the pipeline is legal but has not yet reached the target viewer, then recur

5.2.2 Complexity

The *pipeline-search-forest* algorithm will return all legal paths, which entails exploring the entire forest. So on the average case, the algorithm will run in $O(|E - R| \times |R|)$, where R is the set of root nodes *plucked out* by the procedure *root-node-search*. Note that this is a very loose approximation of the algorithm and does not include:

- the search time for finding R
- the processing time to check for the legality of the pipeline
- the processing time to clone pipelines

5.3 Hybrid Pipelines

From the perspective of the algorithm, an operator is just a node that has input and output data, and so any combination that satisfies the query specifications, as well as the pipeline grammar, is considered a valid result. This presents an opportunity for the algorithm to

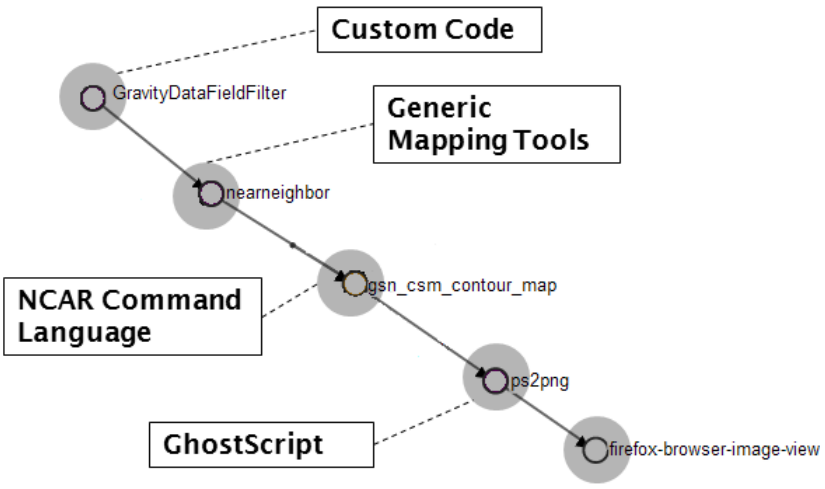


Figure 5.4: Hybrid Pipeline Generating a Contour Map

return pipelines that consist of operators which are supported across different toolkits. We refer to these kinds of pipelines as *hybrid*, and without both a canonical description language provided by the VisKo ontology and the agnostic perspective of the search algorithm, users may not be easily afforded with hybrid pipelines. Without such an infrastructure, users would have to research different toolkits, understand the data requirements of the contained operators, and understand how to mitigate the disparity between invocation protocols of each toolkit.

The notion of hybrid pipelines will be exemplified using the gravity map visualization scenario presented in Chapter 2, where **P**ACES-Gravity[Tabular-ASCII] is transformed into a two dimensional contour map. Considering the GMT pipeline, some of the employed operators could be swapped out with NCAR Command Language (NCL) [28] operators for implementing the contouring task. In particular, the GMT operator `nearneighbor` outputs netCDF grids, which can then be fed to NCL and processed by `gsn-csm-contour-map` to generate a contour map. Such a hybrid pipeline is presented in Figure 5.4. The advantages of hybrid pipelines may have to do with performance or perhaps a preference. For example, a user may prefer the quality of some toolkit's contouring geometry, but require a different toolkit's interpolation routine that is superior in performance.

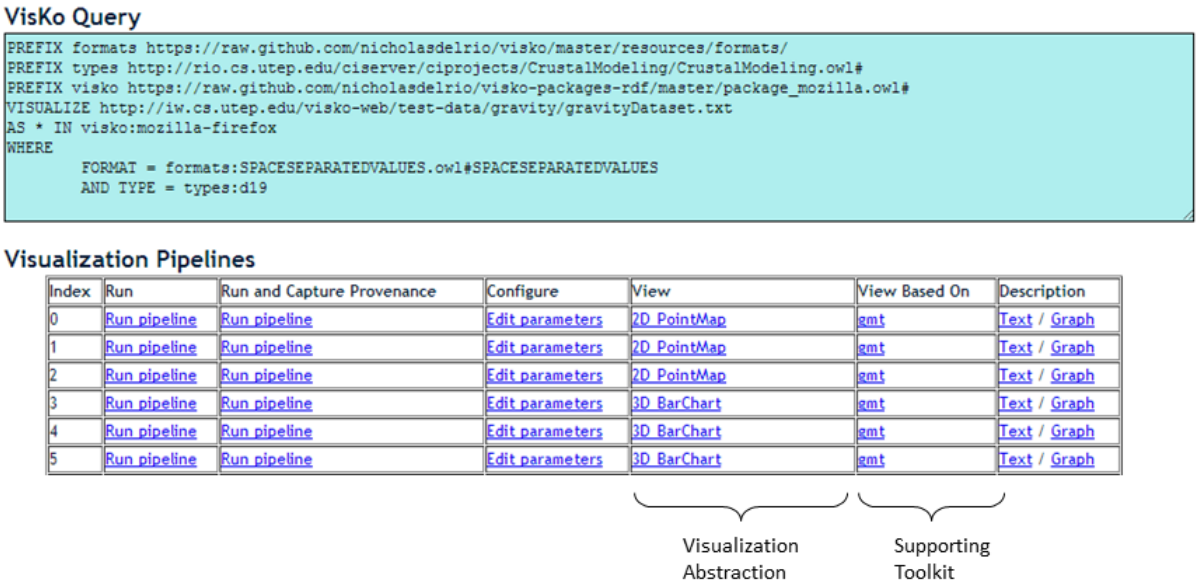


Figure 5.5: Pipeline Query Results

5.4 Pipeline Execution

Upon submitting a query, users are provided with a set of equivalent pipelines that can generate the visualization specified by the query, as shown in Figure 5.5. Users can only obtain a visualization by first selecting a resultant pipeline to execute. This is in contrast to a system that would execute all identified pipelines in batch, gather the resulting visualizations, and present the visuals to the user as a collection. The batch approach however is costly in terms of time and resources and would therefore delay the response time between submitting a query and obtaining a result. This is especially true for cases when the wildcard (*) symbol is used in lieu of specifying a `Visualization Abstraction`.

This dissertation's approach of returning the pipelines to the user first, rather than the visualizations generated by the pipelines, does introduce a gap between the users expectations and the actual system response. For example, relational database users are provided with results of executing an SQL query rather than the relational algebraic operations that compute the results. To compensate for this gulf of execution [61], each returned pipeline is annotated with information about the `Visualization Abstraction` it generates (use-

ful if wildcard query was specified), the toolkit responsible for generating the abstraction, and both a textual and graphical description of the pipeline. By providing this meta-information, users may inspect the transformation process offered by the pipeline and determine its appropriateness. For example, users may not want any visualizations generated by Generic Mapping Tools.

Upon selecting to run a given pipeline, the answering system will execute the following algorithm expressed as pseudo code shown below:

```
1  procedure execute-pipeline(P,source-data,global-param-bindings, type-URI):
2    input-data <- source-data
3    for all Operators o in P.getOperatorSequence() do
4      owlsService <- o.getServiceImplementation
5      bindings <- global-param-bindings.getBindingsForService(owlService, type-URI)
6      owlsService.setParamBindings(bindings)
7      output-data <- owlsService.execute(source-data)
8      input-data <- output-data
9
10   return input-data
```

The procedure **execute-pipeline** iterates through the pipeline `Operator` sequence (line 3) and queries for the OWL-S service implementation attached to each `Operator` (line 4). Before the service is executed, its parameter set must be bound with values and these bindings are gathered (line 5) from the bindings hash table: **global-param-bindings**. Once the relevant set of bindings have been associated with the service (line 6), the service can be executed and generate the **output-data** from the **input-data**. The **output-data** is then set as the **input-data** for the next round of execution (line 8). Upon iterating through the entire sequence, the final **output-data** can be returned as the result of the pipeline execution.

One approach in this dissertation for gathering parameter bindings is to rely on the default values specified by an `InputParameterBindings` construct described in Chapter 4. Based on the semantic type of the dataset being visualized, the pipeline executor will look-up the corresponding `InputParameterBindings` associated with the type and attach the relevant bindings to each OWL-S service. The service can only be executed once all

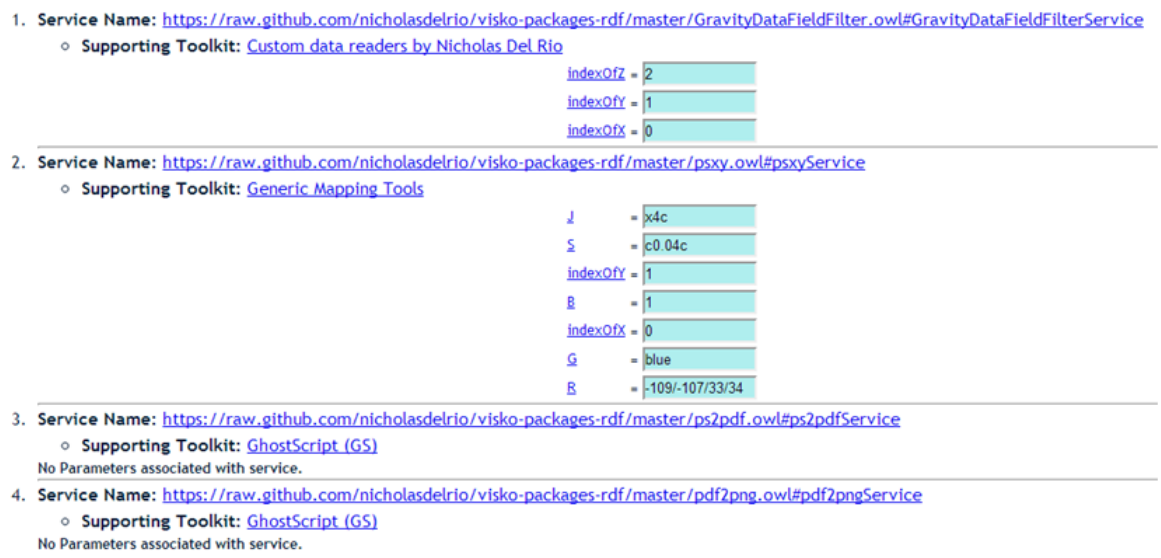


Figure 5.6: Manually Setting Parameters

arguments have been bound and will throw and exception otherwise.

Another approach to gather parameters is to allow the users to explicitly specify these parameter bindings by prompting them to fill out a form as shown in Figure 5.6. In this form, the parameters are grouped by the service they correspond to and users can specify the parameter arguments (i.e., input values) by entering text into the boxes labeled by parameter name. Additionally, parameter bindings can be specified in the query as presented in Chapter 1. In these cases, the **g**lobal-param-bindings hash table is populated from bindings specified in either the form or the query, rather than relying on the `InputParameterBindings`.

Chapter 6

Evaluation

Readability and writability are metrics that are used to assess programming language design and therefore are appropriate metrics for evaluating visualization query and pipeline specification languages. This dissertation aims at determining if users are more proficient at describing visualizations through declarative queries than through specifying sequences of operations. This chapter describes an experiment to verify if the visualization queries (Condition 1) can be *read* and *written* more accurately than pipeline specifications (Condition 2). In the case of readability (Use 1), the accuracy of participants was measured when asked to select the correct visualization described by a query or a pipeline. Conversely, in the case of writability (Use 2), the accuracy of users was measured when composing visualization queries and pipelines that describe some given visualization.

Although reading and writing queries using the visualization query language requires less knowledge than writing procedural pipeline code (Chapter 2), it is uncertain whether the foreign nature of the query language would be a hindrance to experienced visualization toolkit users. The possibility existed that participants would perform the visualization reading and writing tasks with greater speed and precision using traditional pipeline approaches, because they were already familiar with this paradigm of visualization generation. This uncertainly inspired the experiment described in this chapter.

6.1 Pre-existing Knowledge

In order to better isolate the variables being tested (i.e., readability and writability), only participants who are proficient in one of the toolkits listed in the next section were tested.

If experienced users were not targeted, the experiment would have to include a documentation phase on how to use a particular toolkit and therefore introduce a bias with respect to quality of the documentation. A background pre-questionnaire helped to provide some diagnostic information regarding the participants expertise in one of the experiment toolkits. The pre-questionairre can be bound in Appendix B.

6.2 Toolkits Used in Experiment

There are a number of popular MVE-based visualization toolkits that provide users with the modules necessary to construct visualization pipelines. In this experiment, only the following toolkits that support scientific visualizations were considered:

- The Visualization Toolkit (VTK) [70]: http://www.vtk.org/
- Generic Mapping Tools (GMT) [80]: http://gmt.soest.hawaii.edu/
- NCAR Command Language (NCL) [28]: http://www.ncl.ucar.edu/

Because all visualization toolkits in this list embody the procedural pipeline paradigm, they are appropriate for use in the experiment since the aim was to compare the effectiveness of visualization queries against the paradigm of generating visualizations by specifying pipelines.

6.3 Language Factor

To facilitate the broader comparison of our visualization queries against different visualization toolkits in a true unbiased nature, different visualization toolkit idiosyncrasies such as the language used to interface with the toolkit must be factored out. With the exception of the Visualization Toolkit (VTK), most toolkits have a programming language exclusive to interfacing with the toolkits application program interface (API). For example, Generic Mapping Tools (GMT) pipelines are composed using a variety of shell scripting languages like C-Shell, while the NCL toolkit is exclusive to the NCL visualization programming

language. VTK, on the other hand, can interface with Java, TCL, C++, and Python languages. This experiment aimed at assessing only the readibility and writeability of visualization queries and pipelines conceptually, without confounding the results with the readability and writability of the different programming languages themselves. To remove this language variability from the experiment, the pipelines provided and collected from users were specified in a neutral and abstract form that is not executable. The pipelines used by participants are simply a sequence of operator names, for example:

1. operator 1
2. operator 2
3. operator 3
4. ...
5. operator n

In this case, data or parameters need not be specified. The flow of data is inferred from the pipeline structure; **o**perator 1 outputs data that is fed into **o**perator 2. A benefit of this uniform abstraction is that pipelines can be compared among different toolkits. Additionally, users of visual programming toolkits, such as ParaView, can participate in the experiment. For example, ParaView users interact with VTK libraries through the ParaView drag-and-drop interface, and so have some understanding about how to sequence VTK based pipelines devoid of any programming knowledge. Therefore, ParaView users qualified for participation in the experiment, since from their perspective, the pipelines they will be required to read and compose are specified at a conceptual level that they are familiar with.

6.4 Parameter Factor

Specifying visualizations using both procedural pipeline languages and visualization queries requires users to specify parameters that control visual properties such as color, orientation,

and lighting [9]. Properly setting the values of these parameters has been historically challenging for users, and the visualization query language prescribes no method for making this task any easier. Although the approach for answering queries allows three approaches for specifying parameters (e.g., specified through queries, though hard-coded parameter bindings, or through the pipeline execution forms - Chapter 5), it is devoid of any facility for helping users choose appropriate values. From a conceptual point of view, setting parameters in visualization queries and setting parameters in visualization pipelines are equivalent: users are binding values to parameters that control some property of the resulting visualization. Due to this equivalence, we have excluded parameter-setting tasks from our experimental trials and therefore users will not have to read or write parameters in either the pipeline or query based tasks in our evaluation.

6.5 Tutorials

Before users could engage in an experimental trial, they were briefly introduced to visualization queries as well as the notion of conceptual pipelines, as described above. The query tutorial presented the grammar of the visualization query language as well as an example of specific query. Users were given 10 minutes to review the query tutorial and were allowed to ask clarification questions during this period.

Additionally, users were also provided with a pipeline tutorial. Although only users experienced with pipeline programming were targeted for the experiment, the notion of the conceptual pipeline they would be reading and writing needed to be discussed. Similarly to the query tutorial, users we allotted 10 minutes to review the tutorial as well as to ask clarification questions regarding the context of the material. An example of both the query and pipeline tutorials can be found in Appendix C.

Table 6.1: Trial Types

Reading (Use 1)			Writing (Use 2)			
Given	Task	Requested	Given	Task	Requested	
Pipeline Visualization Options Data Description	Identify	Option	Data Description Operator List Visualization	Compose	Pipeline	Pipeline (Condition 2)
Query Visualization Options Data Description	Identify	Option	Data Description KB Concepts Visualization	Compose	Query	Query (Condition 1)

6.6 Trial Types

Participants were required to read and write a visualization pipeline as well as read and write a visualization query. Thus the test space consisted of the different combinations of uses (i.e., readability and writability) against the different conditions (visualization pipeline specification or query language) resulting in four unique trial types, each of which participants were required to complete. Table 6.1 summarizes this enumeration for comparison purposes.

6.6.1 Readabililty

The reading comprehension (Use 1) potion of the experiment required two tests: a pipeline (Condition 1) reading test and a visualization query (Condition 2) reading test. In these reading tests, participants were scored on their accuracy when selecting a visualization from a set of candidate visualizations that was best described by the query or pipeline. The two reading tests were identical in form, where the only difference between the two was the specification languages that the participants were required to read. Appendix D contains an example of both a pipeline and query reading comprehension test.

6.6.2 Writabililty

The writing (Use 2) portion of the experiment also required two tests: a pipeline (Condition 1) writing test and a query (Condition 2) writing test. In these tests, participants were scored on their accuracy when writing either a pipeline specification or query that best described a given visualization. The two writing tests were identical in form, where the only difference between the two was the language used to write the specification. Appendix E contains an example of both a pipeline and query writing ability test.

6.6.3 Supplemental Material

In order to compose visualization queries, users needed to know about a set of resources (i.e., URIs) representing the different components of a visualization query (Chapter 1):

- `Formats`: the file encoding of the input data
- `Types`: the data type of the input data
- `Visualization Abstractions`: the kind of visualization request (e.g., isosurfaces)
- `Viewers`: the module responsible for presenting the generated visualization

Participants were provided with a list of these kinds of resource URIs during the query portions of the experiment. An example of this listing is found in Appendix F.

6.7 Results

Participants were scored a value of 0 for incorrect completion of a task and a value of 1 for correct completion of a task. Future work will include development of a the web-based version of the tasks that can be configured to record participant response times and augment the accuracy scores based on this time. For example, the accuracy score can be defined as: $s = a(1/t)$, where s denotes the computed score, a denotes accuracy, and t denotes response time. Note the inverse relationship between score and time, indicating

Table 6.2: Average Scores for Tasks

	Pipeline (Cond. 2)	Query (Cond. 1)	$P(T <= t)$
Read Avg. Score	0.80	0.73	0.36
Write Avg. Score	0.13	0.54	0.01

that the faster users respond, the higher their score. For this experiment however, users were scored using a binary scoring system: 0 = incorrect and 1 = correct.

The aim in this experiment was to verify if participants had a statistically significantly higher score when completing tasks using queries versus completing tasks using pipelines. Significance was verified using a paired single-tailed t-test. The paired version was employed because both the pipeline and query tasks were completed by the same group of participants. Additionally, the single-tailed version of the test was used due to the consideration of "scoring higher using queries" as more interesting than "scoring higher using pipelines".

Table 6.2 presents the subjects average scores when completing the readability and writability tasks with the testing conditions: pipelines and queries. The data suggests that users are more accurate with the reading and interpretation of pipelines than visualization queries, where the average score for reading pipelines is 0.8, while the average score for reading queries is 0.73 (see Figure 6.2). However, the claim that queries are easier for participants to read can be disputed because these reading scores failed to pass the t-test ($P(T <= t) = 0.36$).

On the other hand, participants were more successful writing correct queries than writing correct pipeline code, and this interpretation is justified by the t-test.

6.8 Participant Feedback

Users were also provided the opportunity to express their measured opinions with regard to what was *easier* for them to read and write. Users were asked to complete a set of

Table 6.3: Average Scores for Questionnaire

	Pipeline (Cond. 2)	Query (Cond. 1)	$P(T <= t)$
Read Avg. Score	3.1	4	0.02
Write Avg. Score	2.5	3.9	0.002

follow-up questions, presented below, which rated on a scale from (1) to (5), where a value of (1) means extremely difficult and a value of (5) means extremely easy. The following is a listing of some of the feedback-related questions.

- How challenging did you find reading and understanding visualization queries?
- How challenging did you find writing visualization queries?
- How challenging did you find reading and understanding visualization pipelines?
- How challenging did you find writing visualization pipelines?

Table 6.3 presents the results of the questionnaire. In both readability and writability cases, users believed that using queries was easier as indicated by the higher means. In the case of the questionnaire, both scoring means were able to to reject the null hypothesis (i.e., query users think queries are equally as challenging as pipelines) under the scrutiny of the same t-test used for verifying the significance of the task-based scores.

6.9 Error Analysis

This experiment also provided information about the kinds of errors users made when writing visualizations. By analyzing the frequency of mistakes, it is possible to identify what portion of the query or pipelines gave users most problems. From this data, it may be possible to enhance the query language to accommodate for its possible shortcomings.

Firstly, both the query and pipelines must be partitioned into sections so that user errors can be grouped according what section of the query or pipeline the error is associated with.

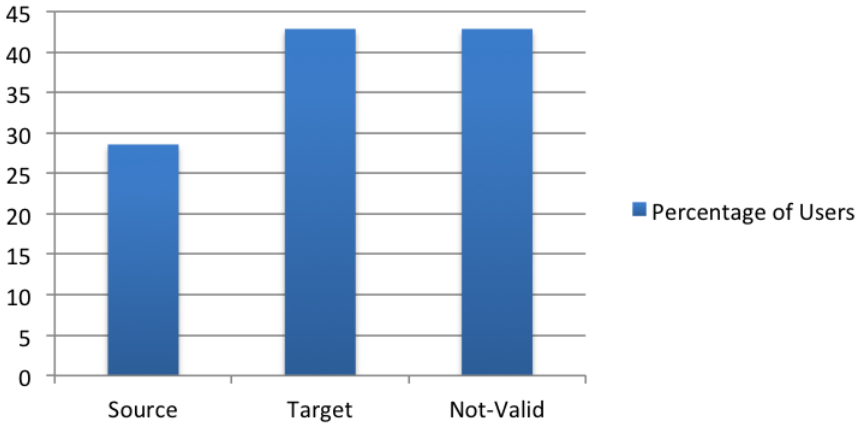


Figure 6.1: Distribution of Query Errors

For queries, we can use the grammar described in Chapter 1 to diagram out the query structure. For this analysis, the query will be decomposed into the follow sections:

1. source: input data URL as well as the `Type` and `Format` clauses
2. target: `Visualization Abstraction` and `Viewer`

The results in Figure 6.1 presents the percentages of users who incorrectly specified some segment of the query they were required to write. This percentage was calculated from the total number of users who scored a 0 for the query writing score (i.e., was not not calculated from the total sample size). Figure 6.1 shows that the majority of query errors resulted from users incorrectly specifying the query target (i.e., the `Visualization Abstraction`,
tt Viewer, or both). This may be due to users not reading the label attached to the visualization they were supposed to write a query for, or perhaps users are just not comfortable with the concept of specifying *what* visualizations they want generated. The *Not-Valid* column refers to instances when users either failed to write a query at all or misunderstood the instructions and wrote a pipeline instead. Later iterations of this experiment will place greater emphasis on the instructions as to prevent users from writing pipelines when they should have written queries.

To partition pipelines, consider the proposed visualization presented in Chapter 4 that defines a pipeline as three major stages:

1. pre-mapping: all type transformations, format conversions, and filtering required before data can be ingested into a `Mapper`
2. mapping: the process of extracting the visualization geometry (i.e., `Visualization Abstraction` from data
3. post-mapping: format conversions required for the geometry to be ingested by a `Viewer`

Using this partitioning scheme, Figure 6.2 presents the percentages of users who incorrectly specified some segment of the pipeline they were required to write. Similarly to Figure 6.1, this percentage was calculated from the total number of users who scored a 0 for the pipeline writing score. The figure shows that the majority of pipeline errors resulted from incorrectly specifying both the mapping and post-mapping stages. Users were better at selecting toolkit operators to read in the raw input data, but did poorly when figuring out what mapping operator would generate the visualization. Additionally, users did not fair well when figuring out how to further transform the `Visualization Abstractions` into a format suitable for `Viewers`. This is expected as often times there is a dependency between the structure of the data output from a `Mapper` and the successive `Transformers` that can operate on that data. If a user choose an incorrect `Mapper`, then it was likely that the successive sequence of operators that follow the mapper would not match-up to the *ideal* pipeline.

6.10 Past Evaluations of Toolkits

Evaluation in the field of visualization can range from performance based studies of some new method or toolkit to user-based studies that aim to evaluate how effective some visualization techniques are. Studies can also be conducted to evaluate the ease-of-use with

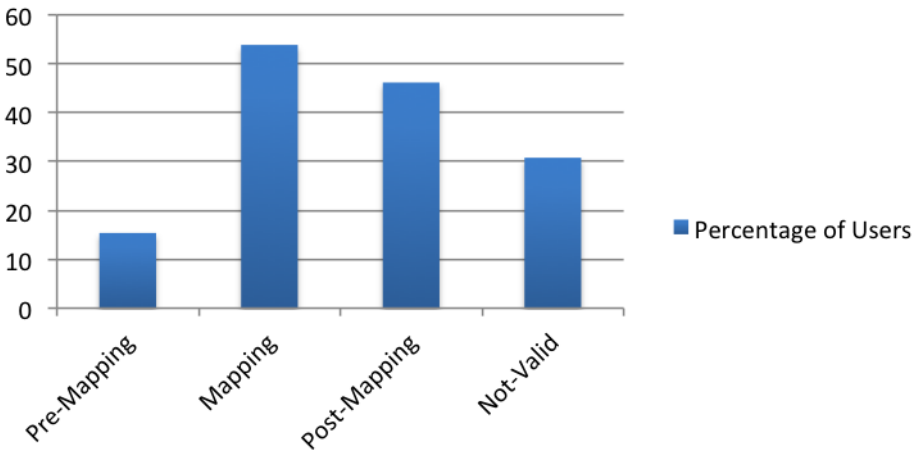


Figure 6.2: Distribution of pipeline Errors

regards to writing visualization applications using some toolkit; the study proposed in this section falls into this category. This section will take a look at some of the past evaluation efforts in each of the aforementioned groups: performance, effectiveness of techniques, and ease of use.

6.10.1 Performance

Data Driven Documents (D3), presented in Chapter 2, provides a novel approach for binding data directly to DOM components without the need for an intermediary layer that abstracts the DOM components as a set of *marks*. Besides benefiting web developers by not blanketing the DOM-based technologies they are already familiar with, there are also performance benefits for not having to translate specifications supported by some intermediate layer. D3 supports this claim by comparing the initialization and frame rates of D3 with Protovis and Flash, which are two popular web-based alternatives for generating visualizations. In terms of initialization, D3 outperformed Protovis and Flash although it is hypothesized that the loading time of the Flash plug-in resulting in a considerable performance hit. In terms of frame rate, Flash outperformed both Protivis and D3 in all cases; when the number of points on the screen was small (500) to large (20,000).

The experiment presented in this chapter is not concerned with performance at this time. Since the visualization query answering approach rests on third-party toolkits, such as GMT and VTK, the pipeline execution performance is partly determined by the performance of the underlying toolkit supporting the services and partly determined by the overhead associated with passing data between the services. It is anticipated that the overhead of passing data back and forth between services will soon become a bottleneck, in which case it will become necessary to localize services with the data. In this case, the pipeline search algorithm would only consider operators which are published local to the data being visualized. This entails a more sophisticated model that captures the notion of locality.

6.10.2 Effectiveness of Techniques

Other work in evaluation has tested the effectiveness of different `Visualization Abstractions` in the context of a set of completion tasks [47]. In this work, authors identified what vector field visualizations best allowed users to:

- choose the number and location of all critical points in an image
- identify the type of a critical point at a specified location
- predict where a particle starting at a specified point will advect

Note that these tasks are very domain specific (e.g., predict where a particle will advect) when compared to Shneiderman's information visualization tasks [73].

In a similar study, the Tableaux authors aimed to assess the usefulness of their visualizations in the context of identifying a performance bottleneck. The case study followed the developers or Argus (parallel graphics lib) in using Tableaux to try to figure out why the performance of the library plateaued before the theoretical limit. The study revealed that Argus debuggers viewed system profile data in different *linked* configurations that allowed them to identify a bug on the OS code and thus figure out the loss of performance. The debuggers formed hypothesis (e.g., too many remote memory accesses) and tried to test hypothesis by analyzing visualizations of profiled memory hits/misses. They would then

reformulate the hypothesis and eventually drill down to the root cause: lock contentions. In this case, the Tableaux system supports the visualization cycle, where visualization requirements are reformulated based on past visualizations and the insight gained.

6.10.3 Ease of Use

Some evaluation has been conducted in assessing how easy it is to generate visualization by writing procedural pipeline specifications based on visualization toolkit APIs. The authors of the Prefuse system aimed to assess the learnability of their visualization API in light of a set of visualization tasks that were completed by eight participants [36]. The tasks included using the API to program an interactive visualization system of social network data in which the results provided evidence that the Prefuse API was very learnable and easy-to-use. Using Prefuse, all but one participant completed every task posed by the user study. While the Prefuse team aimed at evaluating the learnability of their toolkit through non-comparative qualitative means, the study proposed in this chapter specifically aimed to assess both the readability and writability of a visualization query language by comparing the performance of participants using each approach (our query language vs. pipeline specifications). More notably, the experiment presented in this chapter was not just comparing the readability and writability of the query language with some specific toolkit. The experiment is comparing the query language against the overwhelmingly popular paradigm of writing visualization toolkit pipelines.

In contrast to user-centric evaluation approaches, Protovis [37] authors evaluated their domain specific JavaScript language against a set of heuristics, similar to the *Golden Rules* for user interface design proposed by Shneiderman. These heuristics were based on the cognitive dimensions of notation (CDN) [31], which includes:

- closeness of mapping (closeness of representation to domain),
- hidden dependencies (important links between entities are not visible)
- role-expressiveness (the purpose of a component is readily inferred)

- consistency (similar semantics are expressed in similar syntactic forms)
- viscosity (resistance to change)
- hard mental operations (high demand on cognitive resources)
- diffuseness (verbosity of language)
- abstraction (types and availability of abstraction mechanism)

In the future, the visualization language might be evaluated using these kinds of metrics. For the work described in this dissertation however, it seemed more compelling to include users in the loop, thereby gaining feedback from the very patrons of the system. Additionally, the task based evaluation approach provides a direct quantification of the benefit provided by queries (can users read and write queries better than reading and writing pipelines), while most heuristics are approximations and are biased in terms of some usage scenario.

In another study by the authors of Tableaux, the researchers analyzed Tableaux user log files to identify which functions were being used the most among skilled and novice users, as well as to measure the *correctness* of the automatic presentation capabilities provided by the *Show Me* function. Firstly, the authors found that the *Automatic Marks* default is generally correct with 8,248 shelf changes but only 560 mark changes for a 6.8% error rate. Secondly, the *Add to Sheet* command is not being used by skilled users (43 automatic adds for 3,921 direct adds, i.e., 1.1%). Thirdly, the *Show Me* and *Show Me Alternatives* are being used modestly by skilled users (220 commands for 3,921 shelf adds, i.e., 5.6%). This evaluation provided insight about the accuracy of the automatic presentation facilities of Tableaux, in terms of how many *changes* were applied to the default visualization generated. Less mark changes were inferred by authors to indicate that a useful visualization had been generated, while more mark changes indicate that users needed to *tweak* the visualization. In these cases, users were not completely happy with the generated visualization in its default state.

Chapter 7

Conclusion

The premise of this dissertation is that generating visualizations using pipeline specifications is challenging. This premise is based on an observation that generating visualizations using imperative pipeline specifications requires both complex theoretical and implementation specific visualization knowledge. If pipeline specifications are defined as a language, with a corresponding alphabet and grammar, then the problem can be phrased as: describing visualizations using pipelines specification languages can be challenging for both experienced and novice users. To compensate for the complexities of pipeline languages, this dissertation introduced a small declarative query language that can also be used to describe visualizations (Chapter 1).

Another driving force for the advocacy of queries is driven by the fact that different visualization techniques each provide different insights or perspectives into underlying data. In order to support thorough analyses and thereby draw accurate conclusions, it is sometimes necessary to employ multiple visualizations of the same data using various techniques and toolkits. Requiring users to manually program pipelines that generate a diversity of visualizations entails that they understand multiple different toolkits, which may take considerable time before becoming well versed.

In either case when only a single or diverse set of visualizations must be generated using traditional pipelines, users must understand software development factors (Chapter 2) including:

- interfacing languages of the toolkit operators
- supported operators of the toolkit suite

- supported visualizations that can be generated by the operators
- supported data models (e.g., two or three-dimensional)
- performance limitations
- portability limitations

In addition to the more theoretical concerns, users must also understand concepts of visualization theory (Chapter 3) including:

- purpose and goal of pipelines stages
- data characterization (data types and formats)
- different kinds of visualization techniques (e.g., isosurfaces and volumes)
- type transformations
- format conversions
- filtering

The declarative query language and query answering system proposed in this dissertation serves to abstract away the aforementioned software development and visualization theory-based concerns. In the presence of visualization query answering infrastructures, users can move away from the integration concerns associated with using multiple toolkits and move towards environments where these concerns are mitigated by automated mechanisms. In these environments, knowledge about how to generate visualizations is delegated to machines; machines translate queries into equivalent visualization pipelines that are executed to generate requested visualizations. This dissertation defined mechanisms for both populating machines with the required visualization knowledge (Chapter 4) and algorithms for leveraging the knowledge to automatically derive equivalent pipelines that generate visualizations specified by queries (Chapter 5). The knowledge is structured according to a model, known as the VisKo model, which was conceived for the specific purpose of describing toolkit operators in a fashion that could be leveraged by the query answering algorithm. The VisKo model borrows from a number of previous conceptualizations in visualization

theory, where historically most of these models were not designed for automating visualization processes but rather for taxonomization purposes. We therefore augmented and fused concepts from existing models until we arrived at a conceptualization that provided the right level of granularity in each of the aforementioned software development and theoretical concerns.

The question this dissertation ultimately addresses is whether the declarative nature of the query language has any affect on the accuracy of users when describing visualizations. To answer, this dissertation describes a user experiment that, in some cases, verified that users both prefer and are more proficient at using queries to specify visualizations rather than using procedural-style pipelines. Proficiency was defined in terms of *readability* and *writability*, and therefore sets of evaluation tasks were designed from which readability and writeability could be measured. The results indicate that users had some difficulty reading and interpreting queries when compared to interpreting pipelines. This could be due to the fact that the demographic was based on users who were already considered experts using pipelines. In terms of writability, users performed this task with higher accuracy using queries to specify visualizations than pipelines. This is compelling because despite the participants' pre-existing expertise in writing procedural-style pipeline code, they still performed with greater accuracy using queries to which they had only just been exposed.

In addition to accuracy, the experiment revealed the possible inadequacies of visualization queries. Specifically, experimental participants had trouble specifying the the kind of visualizations they wanted (i.e., query target), although they were able to describe the characteristics of the data (i.e., query source) accurately. To compensate, systems such as VisKo, which implement the querying answering facilities described in Chapter 5, can employ form based inputs and allow users to compose visualization queries by selecting values from combo-boxes rather than inputting cumbersome URIs. Additionally, the forms can present users with small thumbnails of different visualizations (i.e., `Visualization Abstractions`) the system can generate, rather than having users mentally map their perception of visualization types with representative URIs. In contrast to queries, users had

much greater difficulty writing pipelines, partly because the structure of a pipeline is more sophisticated than the structure of a query. In the experiment, users had problems with specifying nearly all stages of the pipeline including the pre-mapping, mapping, and post-mapping. Both the task data and error analysis provide evidence supporting the claim that users are more proficient at generating visualizations using the query language than writing traditional pipelines. This preliminary study will hopefully catalyze a paradigm shift regarding how users specify visualizations.

References

[1] Jacques Bertin. Semiology of graphics: diagrams, networks, maps. 1983.

[2] Michael Bostock and Jeffrey Heer. Protovis: A graphical toolkit for visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 15(6):1121–1128, 2009.

[3] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2301–2309, 2011.

[4] Jeff Braun, Ray Ford, and David Thompson. *OpenDX: Paths to Visualization: Material Used for Learning OpenDX-the Open Derivate of IBM's Visualization Data Explorer*. Visualization an imagery solutions, 2001.

[5] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Journal*, 2(4):27–66, 1997.

[6] Ken Brodlie. Visualization Ontologies. http://www.nesc.ac.uk/talks/393/vis_ontology_report.pdf.

[7] Ken Brodlie and Nurul Mohd Noor. Visualization notations, models and taxonomies. *EG UK Theory and Practice of Computer Graphics*, pages 207–212, 2007.

[8] Gordon Cameron. Modular visualization environments: Past, present, and future. *ACM SIGGRAPH Computer Graphics*, 29(2):3–4, 1995.

[9] Stuart K Card and Jock Mackinlay. The structure of the information visualization design space. In *Information Visualization, 1997. Proceedings., IEEE Symposium on*, pages 92–99. IEEE, 1997.

[10] Stuart K Card, Jock D Mackinlay, and Ben Shneiderman. *Readings in information visualization: using vision to think*. Morgan Kaufmann Pub, 1999.

[11] Andy Cedilnik, Berk Geveci, Kenneth Moreland, James Ahrens, and Jean Favre. Remote large data visualization in the paraview framework, 2006.

[12] Min Chen, David Ebert, Hans Hagen, Robert S. Laramee, Robert van Liere, Kwan-Liu Ma, William Ribarsky, Gerik Scheuermann, and Deborah Silver. Data, information, and knowledge in visualization. *IEEE Comput. Graph. Appl.*, 29(1):12–19, 2009.

[13] Ed Huai-hsin Chi. A taxonomy of visualization techniques using the data state reference model. In *INFOVIS '00: Proceedings of the IEEE Symposium on Information Vizualization 2000*, page 69, Washington, DC, USA, 2000. IEEE Computer Society.

[14] Ed Huai-hsin Chi. Expressiveness of the data flow and data state models in visualization systems. In *AVI '02: Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 375–378, New York, NY, USA, 2002. ACM.

[15] Ed Huai-hsin Chi and John Riedl. An operator interaction framework for visualization systems. In *INFOVIS '98: Proceedings of the 1998 IEEE Symposium on Information Visualization*, pages 63–70, Washington, DC, USA, 1998. IEEE Computer Society.

[16] Ed Huai-hsin Chi, John Riedl, Phillip Barry, and Joseph Konstan. Principles for information visualization spreadsheets. *IEEE Comput. Graph. Appl.*, 18(4):30–38, 1998.

[17] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (wsdl) 1.1, 2001.

[18] Mei C Chuah and Steven F Roth. On the semantics of interactive visualizations. In *Information Visualization'96, Proceedings IEEE Symposium on*, pages 29–36. IEEE, 1996.

[19] Daniel E Cooke, Vladik Kreinovich, and Scott A Starks. Alps: A logic for program synthesis (motivated by fuzzy logic). In *Fuzzy Systems Proceedings, 1998. IEEE World*

Congress on Computational Intelligence., The 1998 IEEE International Conference on, volume 1, pages 779–784. IEEE, 1998.

[20] Douglas Crockford. The application/json media type for javascript object notation (json). 2006.

[21] Robert A Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In *ACM Siggraph Computer Graphics*, volume 22, pages 65–74. ACM, 1988.

[22] David J Duke, Ken W Brodlie, David A Duce, and Ivan Herman. Do you see what i mean?[data visualization]. *Computer Graphics and Applications, IEEE*, 25(3):6–9, 2005.

[23] David J Duke, KW Brodlie, and DA Duce. Building an ontology of visualization. In *Visualization, 2004. IEEE*, pages 7p–7p. IEEE, 2004.

[24] D Scott Dyer. A dataflow toolkit for visualization. *Computer Graphics and Applications, IEEE*, 10(4):60–69, 1990.

[25] ESIP. Earth Science Information Partners ESIP Federation Datatype Ontology. http://wiki.esipfed.org/index.php/Data-Service-Ontologies, 2007.

[26] Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. Scalable vector graphics (svg) 1.1 specification. *World Wide Web Consortium (W3C). URL http://www. w3. org/TR/SVG11*, 2003.

[27] Leo Ferres, Michel Dumontier, and Natalia Villanueva-Rosales. Semantic query answering with time-series graphs. In *EDOC Conference Workshop, 2007. EDOC'07. Eleventh International IEEE*, pages 117–124. IEEE, 2007.

[28] NCAR National Center for Atomospheric Research. NCAR command language reference manual. http://www.ncl.ucar.edu/Document/Manuals/Ref_Manual/, 2012.

[29] David Foulser. Iris explorer: a framework for investigation. *SIGGRAPH Comput. Graph.*, 29(2):13–16, 1995.

[30] Ned Freed and Nathaniel Borenstein. Multipurpose internet mail extensions (mime) part one: Format of internet message bodies, 1996.

[31] Thomas RG Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.

[32] MATLAB Users Guide. The mathworks. *Inc., Natick, MA*, 5, 1998.

[33] Robert B Haber and David A McNabb. Visualization idioms: A conceptual model for scientific visualization systems. volume 74, pages 74–93. 1990.

[34] Robert J Hanisch, Allen Farris, Eric W Greisen, William D Pence, Barry M Schlesinger, Peter J Teuben, Randall W Thompson, and Archibald Warnock III. Definition of the flexible image transport system (fits). *Astronomy and Astrophysics*, 376(1):359–380, 2001.

[35] Marti Hearst. Design recommendations for hierarchical faceted search interfaces. In *ACM SIGIR workshop on faceted search*, pages 1–5, 2006.

[36] Jeffrey Heer. Prefuse: a toolkit for interactive information visualization. In *In CHI 2005: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 421–430. ACM Press, 2005.

[37] Jeffrey Heer and Michael Bostock. Declarative language design for interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16:1149–1156, 2010.

[38] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F Patel-Schneider, and Sebastian Rudolph. *OWL 2 web ontology language primer*, volume 27. 2009.

[39] Matthew Horridge, Nick Drummond, John Goodwin, Alan Rector, Robert Stevens, and Hai H Wang. The manchester owl syntax. *OWL: Experiences and Directions*, pages 10–11, 2006.

[40] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.

[41] International Organization for Standardization (ISO). *SQL Part 2: Foundation (SQL/Foundation)*, 2008.

[42] Philip L. Isenhour, James Bo Begole, Winfield S. Heagy, and Clifford A. Shaffer. Sieve: A collaborative interactive modular visualization environment, 1995.

[43] TJ Jankun-Kelly, Kwan-Liu Ma, and Michael Gertz. A model and framework for visualization exploration. *Visualization and Computer Graphics, IEEE Transactions on*, 13(2):357–369, 2007.

[44] Eric Jendrock. *The Java EE 5 Tutorial: For Sun Java System Application Server Platform Edition 9*. Addison-Wesley Professional, 2006.

[45] Sebastien Jourdain, Utkarsh Ayachit, and Berk Geveci. Paraviewweb, a web framework for 3d visualization and data processing. In *IADIS International Conference on Web Virtual Reality and Three-Dimensional Worlds*, volume 7, 2010.

[46] Craig A Knoblock, Steven Minton, Jose Luis Ambite, Naveen Ashish, Pragnesh Jay Modi, Ion Muslea, Andrew G Philpot, Sheila Tejada, et al. Modeling web sources for information integration. In *Proceedings of the National Conference on Artificial Intelligence*, pages 211–218. JOHN WILEY & SONS LTD, 1998.

[47] David H Laidlaw, Robert M Kirby, Cullen D Jackson, J Scott Davidson, Timothy S Miller, Marco Da Silva, William H Warren, and Michael J Tarr. Comparing 2d vector field visualization methods: A user study. *Visualization and Computer Graphics, IEEE Transactions on*, 11(1):59–70, 2005.

[48] Håkon Wium Lie and Bert Bos. *Cascading style sheets*. Addison Wesley Longman, 1997.

[49] I.S. LLC. Imagemagick. *URL, http://www. imagemagick. org/script/index. php*, 2009.

[50] Bruce Lucas, Gregory D. Abram, Nancy S. Collins, David A. Epstein, Donna L. Gresh, and Kevin P. McAuliffe. An architecture for a scientific visualization system. In *VIS 92: Proceedings of the 3rd conference on Visualization 92*, pages 107–114, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[51] Jock D. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5(2):110–141, 1986.

[52] Jock D. Mackinlay, Pat Hanrahan, and Chris Stolte. Show me: Automatic presentation for visual analysis. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1137–1144, 2007.

[53] Frank Manola, Eric Miller, and Brian McBride. *RDF primer*, volume 10. 2004.

[54] Joe Marks, Brad Andalman, Paul A Beardsley, William Freeman, Sarah Gibson, Jessica Hodgins, Thomas Kang, Brian Mirtich, Hanspeter Pfister, Wheeler Ruml, et al. Design galleries: A general approach to setting parameters for computer graphics and animation. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 389–400. ACM Press/Addison-Wesley Publishing Co., 1997.

[55] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, et al. Owl-s: Semantic markup for web services, 2004.

[56] Larry Masinter, Tim Berners-Lee, and Roy T Fielding. Uniform resource identifier (uri): Generic syntax. 2005.

[57] Deborah McGuinness, Li Ding, Paulo Pinheiro da Silva, and Cynthia Chang. PML2: A Modular Explanation Interlingua. In *Proceedings of the AAAI 2007 Workshop on Explanation-aware Computing*, Vancouver, British Columbia, Canada, July 22-23 2007.

[58] M Douglas McIlroy, JM Buxton, Peter Naur, and Brian Randell. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98. sn, 1968.

[59] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[60] Thomas Narten. Guidelines for writing an iana considerations section in rfcs. 2008.

[61] Donald A Norman. *The design of everyday things*. Basic Books (AZ), 2002.

[62] PACES. Pan American Center for Earth and Environmental Studies. http://research.utep.edu/Default.aspx?alias=research.utep.edu/paces, 2002.

[63] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and P-P Sloan. Interactive ray tracing for isosurface rendering. In *Visualization'98. Proceedings*, pages 233–238. IEEE, 1998.

[64] Eric PrudHommeaux, Andy Seaborne, et al. Sparql query language for rdf, 2008.

[65] Kurt Rohloff, Mike Dean, Ian Emmons, Dorene Ryder, and John Sumner. An evaluation of triple-store technologies for large data stores. In *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, pages 1105–1114. Springer, 2007.

[66] Steven F. Roth, John Kolojejchick, Joe Mattis, and Jade Goldstein. Interactive graphic design using automatic presentation knowledge. In *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 112–117, New York, NY, USA, 1994. ACM.

[67] Steven F Roth and Joe Mattis. Data characterization for intelligent graphics presentation. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people*, pages 193–200. ACM, 1990.

[68] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[69] Carlos E Scheidegger, Huy T Vo, David Koop, Juliana Freire, and Cláudio T Silva. Querying and creating visualizations by analogy. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1560–1567, 2007.

[70] Will Schroeder, Kenneth M. Martin, and William E. Lorensen. *The visualization toolkit (2nd ed.): an object-oriented approach to 3D graphics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[71] William J. Schroeder, William E. Lorensen, GD Montanaro, and Christopher R. Volpe. Visage: an object-oriented scientific visualization system. In *Proceedings of the 3rd conference on Visualization'92*, pages 219–226. IEEE Computer Society Press, 1992.

[72] Ben Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on graphics (TOG)*, 11(1):92–99, 1992.

[73] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 336, Washington, DC, USA, 1996. IEEE Computer Society.

[74] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics*, 5:51–53, 2007.

[75] Chris Stolte and Pat Hanrahan. Polaris: A system for query, analysis and visualization of multi-dimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8:52–65, 2002.

[76] Melanie Tory and Torsten Moller. Rethinking visualization: A high-level taxonomy. In *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*, pages 151–158. IEEE, 2004.

[77] UCAR University Corporation of Atmospheric Research. Network common data form netcdf. http://www.unidata.ucar.edu/software/netcdf/, 2012.

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

[79] Jarke J Van Wijk. The value of visualization. In *Visualization, 2005. VIS 05. IEEE*, pages 79–86. IEEE, 2005.

[80] Paul Wessel and Walter HF Smith. New, improved version of generic mapping tools released. *Eos, Transactions American Geophysical Union*, 79(47):579–579, 1998.

[81] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[82] Brian Wylie and Jeffrey Baumes. A unified toolkit for information and scientific visualization. In *VDA*, page 72430, 2009.

[83] Michel Zeiler. *Modeling our world: the ESRI guide to geodatabase design*, volume 40. Esri Pr, 1999.

[84] Michelle. X. Zhou and Steven. K. Feiner. Data characterization for automatically visualizing heterogeneous information. In *Proceedings of the 1996 IEEE Symposium on Information Visualization (INFOVIS '96)*, INFOVIS '96, pages 13–22, Washington, DC, USA, 1996. IEEE Computer Society.

Appendix A

Example Module Code

The following Java code frangments compose a module for registering the Visualization Toolkit (VTK) operator vtkContourFilter. The behavior of this operator is dependent upon the input data type. Feeding vtkContourFilter two-dimensional `vtkImageData` results in the generation of isolines (i.e., contour lines), whereas three-dimensional versions result in the generation of isosurfaces.

To describe both of these behaviors, users can describe the same operator, vtkContourFilter, from two different perspecives: the two-dimensional and three-dimensional versions. Therefore, there can exist a 1-to-many relationship between an operator service and its descriptions. The module annotation code below describes vtkContourFilter from the three-dimensional perspective.

A.1 Module Service Code

```
1  package org.openvisko.module;
2
3  import javax.jws.WebParam;
4
5  public class ModuleService {
6    //these annotations ensure the paramter names in the WSDL
7    // are preserved
8    public String vtkContourFilter(
9        @WebParam(name="url") String url,
10       @WebParam(name="numContours") String numContours,
11       @WebParam(name="scalarRange") String scalarRange)
12   {
13     // VTKContourFilter wraps the vtkContourFilter java object
14     VTKContourFilter transformer = new VTKContourDataFilter(url);
```

```
    return transformer.transform(numContours, scalarRange);
  }
}
```

A.2 Module Annotation Code

```
package org.openvisko.module;

import java.net.URL;

import org.openvisko.module.registration.AbstractModuleRDFRegistration;
import org.openvisko.module.registration.ModuleInputParameterBindings;
import org.openvisko.module.registration.ModuleOperatorService;
import org.openvisko.module.registration.ModuleWriter;

import com.hp.hpl.jena.ontology.OntResource;
import edu.utep.trustlab.visko.ontology.pmlp.Format;
import edu.utep.trustlab.visko.ontology.viskoService.Toolkit;
import edu.utep.trustlab.visko.ontology.viskoView.VisualizationAbstraction;
import edu.utep.trustlab.visko.ontology.vocabulary.ViskoV;

/*
 * ModuleRDFRegistration contains code for annotating an visualization toolkit operator
    that is exposed as a WSDL
 * Service. It contains three methods which must be overriden:
 * - populateServices: contains code to annotate an operator exposed as a service
 * - populateToolkit: contains code to describe a toolkit and associate the toolkit with
    the operator service
 * - populateParameterBindings: contains code to associate a set of parameter values with
    a semantic type
 */
public class ModuleRDFRegistration extends AbstractModuleRDFRegistration {

  /*
   * The nested Resources class contains a set of URIs that refer to:
   * - Types
   * - Formats
   * - Visualization Abstractions
   */
  static final class Resources {

    // Formats
```

```
34    static final Format xml = ModuleWriter.getFormat("http://openvisko.org/rdf/pml2/
          formats/XML.owl#XML");
35
36    // Data Types
37    static final OntResource vtkImageData3D = ModuleWriter.getDataType("http://www.vtk.org
          /vtk-data.owl#vtkImageData3D");
38    static final OntResource vtkPolyData = ModuleWriter.getDataType("http://www.vtk.org/
          vtk-data.owl#vtkPolyData");
39
40    // Visualization Abstractions
41    static final VisualizationAbstraction isosurfaces3D = ModuleWriter.getView(ViskoV.
          INDIVIDUAL_URI_3D_IsoSurfaceRendering);
42
43    // Semantic Types
44    static final OntResource velocityDataURI_1 = ModuleWriter.getDataType("http://rio.cs.
          utep.edu/ciserver/ciprojects/HolesCode/HolesCodeSAW3.owl#d14-0");
45
46    static String vtkContourFilter3D = "vtkContourFilter3D";
47  }
48
49  @Override
50  public void populateServices() {
51
52    // ModuleOperatorService annotates a WSDL service.
53    ModuleOperatorService service1 = packageWriter.createNewOperatorService(null,
          Resources.vtkContourFilter3D);
54    service1.setWSDLURL(wsdlURL);                    // specifies service to annotate
55    service1.setInputFormat(Resources.xml);               // specifies input Format
56    service1.setOutputFormat(Resources.xml);              // specifies output Format
57    service1.setLabel(Resources.vtkContourFilter3D);          // sets a natural language
          label
58    service1.setComment("Generates isosurfaces from 3D vtkImageData");  // sets a natural
          langauge description
59    service1.setInputDataType(Resources.vtkImageData3D);        // specifies input type
60    service1.setOutputDataType(Resources.vtkPolyData);          // specifies outoput type
61    service1.setView(Resources.isosurfaces3D);              // specifies Visualization
          Abstraction being generated
62                                    // can be null if operator is a Transformer, not a
                                          Mapper
63  }
64
65  @Override
```

```
66    public void populateToolkit() {
67
68      // The toolkit which supports the vtkContourFilter3D operator being registered
69      Toolkit toolkit = getModuleWriter().createNewToolkit("vtk");
70      toolkit.setComment("Visualization Toolkit (VTK) Library");
71      toolkit.setLabel("Visualization Toolkit (VTK)");
72
73      String stringURL = "http://www.vtk.org/";
74      try
75      {
76        URL toolkitURL = new URL(stringURL);
77        toolkit.setDocumentationURL(toolkitURL);
78      }
79      catch(Exception e){e.printStackTrace();}
80    }
81
82
83    @Override
84    public void populateParameterBindings() {
85      // Create bindings for Semantic Type of VelocityData
86      ModuleInputParameterBindings bindings = getModuleWriter().
            createNewInputParameterBindings();
87      bindings.addSemanticType(Resources.velocityDataURI_1);
88
89      // for vtkContourFilter3D
90      bindings.addInputBinding(Resources.vtkContourFilter3D, "numContours", "35");
91      bindings.addInputBinding(Resources.vtkContourFilter3D, "scalarRange", "0.0/9000.0");
92    }
93  }
```

Appendix B

Pre-Questionnaire

Please try your best to answer the following questions as accurately as possible.

B.1 Background Questions

1. What is the highest academic degree you have earned?
2. What is your primary field of study (e.g., computer science, chemistry)?
3. How many years have you worked in your field of study?
4. Do you have any experience programming?
5. If so, what is your favorite programming language?

B.2 Visualization Questions

1. Have you ever needed to use a visualization to analyze data? (yes/no)
2. What kinds of visualizations have you generated (e.g., contour maps, scatter plots, volume rendering)?
3. Do you think visualization is a useful technique for assisting with data analysis? (yes/no)
4. Do you work more with scientific visualizations (e.g., surfaces and volumes) or information visualizations (graphs and networks)?

B.3 Visualization Toolkit Questions

1. Which visualization toolkit are you most proficient in? (VTK, GMT, NCL, or other please specify)
2. How many months did it take you to become proficient using this toolkit?
3. Did you find there is adequate documentation describing how to use the toolkit you are proficient in? (yes/no)
4. Could you write an application using the toolkit you are proficient in to generate a simple contour map of tabular data? (yes/no)
5. Could you write an application using the toolkit you are proficient in to generate a 2D plot of tabular data? (yes/no)
6. Could you write an application using the toolkit you are proficient in to generate raster map of tabular data? (yes/no)
7. Could you write an application using the toolkit you are proficient in to generate a time series plot of netCDF data? (yes/no)
8. Could you write an application using the toolkit you are proficient in to generate a set of isosurfaces from a three dimensional grid of data? (yes/no)
9. Could you write an application using the toolkit you are proficient in to generate a volume rending from a three dimensional grid of data? (yes/no)

Appendix C

Query and Pipeline Tutorials

This section presents the tutorial material presented to users.

VisKo Query Language (VisKo-QL) allows users to request for visualizations declaratively and without being fully aware of a wide range of visualization toolkit implementation details. To write visualization queries, users will need to know how their data is characterized in terms of ***type*** and ***format*** as well as what ***view*** their data should be visualized as.

The following segment defines our query grammar, which has been simplified for this experiment:

QUERY	-> INPUT VIEW VIEWER TYPE FORMAT	
INPUT	-> **"VISUALIZE"** url	input data to visualize
VIEW	-> **"AS"** view	view to be generated from input
VIEWER	-> **"IN"** viewer	viewer used to present view
TYPE	-> **"WHERE TYPE =** " type	type of input data
FORMAT	-> **"AND FORMAT =** " format	format of input data

A listing of the supported views, viewers, formats, and types can be found on the query resources handout.

Example: Generate a contour map (i.e., iso-lines) that can be viewed in a web browser, from 2D unstructured points encoded in plain ASCII text. Data is stored at http://trust.utep.edu/dataset.txt.

Query:

```
VISUALIZE http://trust.utep.edu/dataset.txt
AS http://rio.cs.utep.edu/views.owl#CONTOUR-LINES
IN http://rio.cs.utep.edu/viewer-sets.owl#WebBrowser
WHERE TYPE = http://rio.cs.utep.edu/types.owl#UNSTRUCTURED-POINTS-2D
AND FORMAT = http://rio.cs.utep.edu/formats.owl#PLAINTEXT
```

Result:

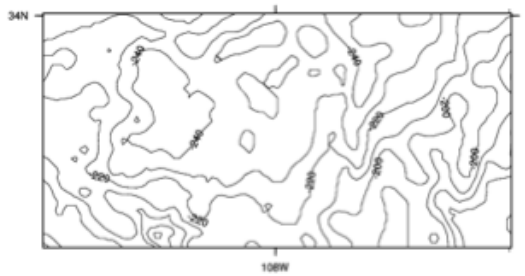


Figure C.1: Query Tutorial

Pipelines are sequences of visualization toolkit modules (e.g., functions, methods, executables) in which the output of each module is fed as input to the next module in the sequence. In this user study, we will define a pipeline as a sequence of module names supported by some visualization toolkit (e.g., VTK and GMT) without any parameter specifications. For example, consider the following pipeline:

Pipeline
1. Routine1 **(custom code that performs function X)**
2. Operator2
3. Operator3
4. Operator4

Operator2, *Operator3*, and *Operator4* could be either GMT or VTK operators that are piped together to form an application that can generate some visualization. Note that *Routine1* is not a standard VTK or GMT operator, but provides functionality that was needed to support some transformation. ***We allow pipelines to consist of both standard toolkit and custom modules, but when using custom modules, please provide a brief description of its function or capabilities.***

Figure C.2: Pipeline Tutorial

Appendix D

Readability Tests

The following are example readability tests using pipeline and query languages.

Trail Type 1 (VTK-based)

Instructions: Using the input data and pipeline described below, ***choose the visualization*** that would most likely be generated by circling it.

NOTE: Please refrain from leveraging any source outside of the evaluation material presented to you. This includes toolkit manuals of any kind (e.g., versions published on the Web).

Input Data Description:

- **Data Format:** Binary Float Array
- **Data Type:** Gridded Scalars
- **Data Dimensionality:** 2

Pipeline:

```
1. vtkStructuredGridReader
2. vtkContourFilter
3. vtkPolyDataMapper
4. vtkActor
```

Possible Visualization Outputs (**circle the most likely output and justify your selection on the back**):

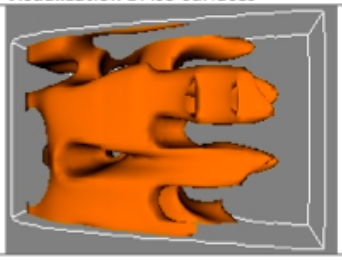
Visualization A: Iso-lines	**Visualization B:** Iso-surfaces	**Visualization C:** null
		

Figure D.1: Pipeline Example

Example: Trial Type 3 (VTK-based)

Instructions: Using the input data and query described below, ***choose the visualization*** that would most likely be generated by circling it.

NOTE: Please refrain from leveraging any source outside of the evaluation material presented to you. This includes toolkit manuals of any kind (e.g., versions published on the Web).

Input Data Description:

- **Data Format:** Binary Float Array
- **Data Type:** Gridded Vectors
- **Data Dimensionality:** 3

Query:

```
VISUALIZE http://density-data.bin
AS http://rio.cs.utep.edu/views.owl#STREAM-LINES
IN http://rio.cs.utep.edu/viewer-sets.owl#WebBrowser
WHERE TYPE = http://rio.cs.utep.edu/types.owl#GRIDDED-VECTORS-3D
AND FORMAT = http://rio.cs.utep.edu/formats.owl#BINARYFLOATARRAY
```

Possible Visualization Outputs (**circle the most likely output and justify your selection on the back**):

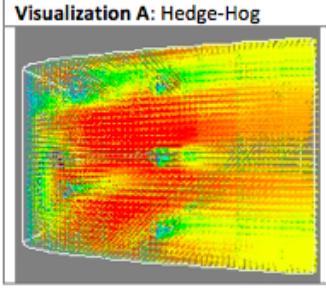
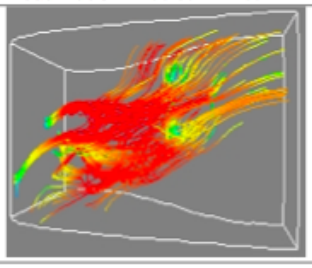
Visualization A: Hedge-Hog	**Visualization B**: Stream-lines	**Visualization C:** null
		

Figure D.2: Query Example

Appendix E

Writability Tests

The following are example writabililty tests using pipeline and query languages.

Example: Trial Type 2 (VTK-based)

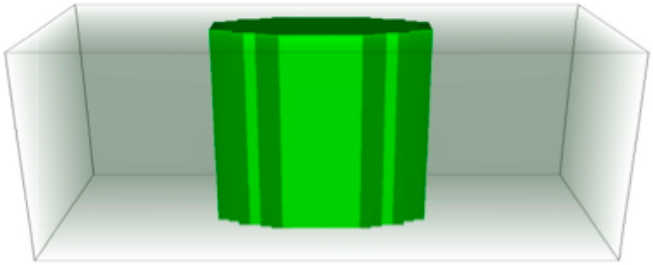
Instructions: Using the input data, list of pipeline operators, and visualization shown below, ***write the visualization pipeline*** that would most likely generate the visualization.

NOTE: Please refrain from leveraging any source outside of the evaluation material presented to you. This includes toolkit manuals of any kind (e.g., versions published on the Web).

Input Data Description:

- **Data Format:** Binary Float Array
- **Data Type:** Gridded Scalars
- **Data Dimensionality:** 3

Visualization: Volume



Visualization Pipeline (please write down the pipeline that could generate the visualization):

Figure E.1: Pipeline Example

Example: Trial Type 4 (VTK-based)

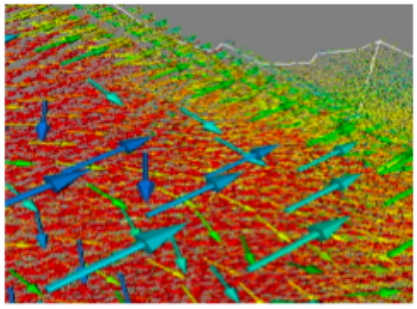
Instructions: Using the input data, list of visualization resources, and visualization shown below, ***write the visualization query*** that would most likely generate the visualization.

NOTE: Please refrain from leveraging any source outside of the evaluation material presented to you. This includes toolkit manuals of any kind (e.g., versions published on the Web).

Input Data Description:

- **Data Format:** Binary Float Array
- **Data Type:** Gridded Vectors
- **Data Dimensionality:** 2
- **Data Location:** http://trust.utep.edu/data.bin

Visualization: Glyphs to be viewed in a web browser



Visualization Query (please write down the query that could generate the visualization):

Figure E.2: Query Example

Appendix F

Supplemental Material

This section provides a listing of the URI resources that were provided to participants during the query-testing portion of the experiment.

http://rio.cs.utep.edu/formats.owl#GIF
http://rio.cs.utep.edu/formats.owl#JPEG
http://rio.cs.utep.edu/formats.owl#PNG
http://rio.cs.utep.edu/formats.owl#VTKPOLYDATA
http://rio.cs.utep.edu/formats.owl#PDF
http://rio.cs.utep.edu/formats.owl#NETCDF
http://rio.cs.utep.edu/formats.owl#SPACEDELIMITEDTABULARASCII
http://rio.cs.utep.edu/formats.owl#ESRIGRID
http://rio.cs.utep.edu/formats.owl#BINARYINTARRAY
http://rio.cs.utep.edu/formats.owl#BINARYSHORTINTARRAY
http://rio.cs.utep.edu/formats.owl#VNDWTSTF
http://rio.cs.utep.edu/formats.owl#VTKIMAGEDATA
http://rio.cs.utep.edu/formats.owl#PLAIN
http://rio.cs.utep.edu/formats.owl#PLAINTEXT
http://rio.cs.utep.edu/formats.owl#VNDLATEXZ
http://rio.cs.utep.edu/formats.owl#FITS
http://rio.cs.utep.edu/formats.owl#TIFF
http://rio.cs.utep.edu/formats.owl#DICOM
http://rio.cs.utep.edu/formats.owl#RAW
http://rio.cs.utep.edu/formats.owl#BINARYFLOATARRAY
http://rio.cs.utep.edu/formats.owl#POSTSCRIPT
http://rio.cs.utep.edu/formats.owl#HTML

Figure F.1: Formats

http://rio.cs.utep.edu/types.owl#UNSTRUCTURED-POINTS-2D
http://rio.cs.utep.edu/types.owl#UNSTRUCTURED-POINTS-3D
http://rio.cs.utep.edu/types.owl#STRUCTURED-POINTS-2D
http://rio.cs.utep.edu/types.owl#STRUCTURED-POINTS-3D
http://rio.cs.utep.edu/types.owl#GRIDDED-SCALARS-2D
http://rio.cs.utep.edu/types.owl#GRIDDED-SCALARS-3D
http://rio.cs.utep.edu/types.owl#GRIDDED-VECTORS-2D
http://rio.cs.utep.edu/types.owl#GRIDDED-VECTORS-3D
http://rio.cs.utep.edu/types.owl#TIME-SERIES

Figure F.2: Types

http://rio.cs.utep.edu/views.owl#VOLUME
http://rio.cs.utep.edu/views.owl#RASTER
http://rio.cs.utep.edu/views.owl#CONTOUR-LINES
http://rio.cs.utep.edu/views.owl#ISO-SURFACES
http://rio.cs.utep.edu/views.owl#HEDGE-HOG
http://rio.cs.utep.edu/views.owl#XY-PLOT
http://rio.cs.utep.edu/views.owl#GLYPHS
http://rio.cs.utep.edu/views.owl#STREAM-LINES

Figure F.3: Visualization Abstractions

http://rio.cs.utep.edu/viewer-sets.owl#PROBE-IT
http://rio.cs.utep.edu/viewer-sets.owl#INTERNET-EXPLORER
http://rio.cs.utep.edu/viewer-sets.owl#WebBrowser

Figure F.4: Viewers

Curriculum Vitae

Nicholas Del Rio earned his Bachelor of Engineering degree in Computer Science from the University of Texas at El Paso in 2004. He received his Master of Science degree also in Computer Science during 2007, upon which time he joined the doctoral program. During his tenure as a doctoral student, Nicholas was employed as a Research Associate funded by the Cyber-ShARE Center. He also served as a graduate intern at NASA Goddard, where he was responsible for integrating his dissertation work with an existing analysis system for remote sensed data. Nicholas was a recipient of the Cotton Memorial Scholarship as well as the Outstanding Graduate Student in Computer Science award.

Nicholas Del Rio has presented his dissertation work to a number of organizations including: the Center for Extreme Data Management Analysis and Visualization (CEDMAV), Federation of Earth Science Information Partners (ESIP), NASA Goddard Space Flight Center, Pacific Northwest National Laboratory (PNNL), and the Rome Air Force Research Lab. His graduate work has been published at a number of venues including the International Semantic Web Conference (ISWC) and Association for the Advancement of Artificial Intelligence (AAAI) Fall Symposium. Nicholass dissertation entitled, A Declarative Domain Independent Approach for Querying and Generating Visualizations, was supervised by Dr. Paulo Pinheiro.

Permanent address: 513 Queretaro Dr.
El Paso, Texas 79912

This dissertation was typed by the author.