Visualization Queries

Nicholas Del Rio and Paulo Pinheiro da Silva

The University of Texas at El Paso, Computer Science,
500 W. University Ave. El Paso TX 79968 USA

Abstract. This paper introduces the notion of a *query* that describes a visualization request in terms of a universe of concepts, which includes but is not limited to *Views*, *Operators*, and *Parameters*. In the presence of specifications based on these concepts, users may be able to request for *what* properties they want to see in their visualizations (i.e., the graphical analogues of some dataset) without having to specify *how* to generate them, and more importantly, without being fully aware of a wide range of toolkit specific implementation details. The result of a query is a visualization that satisfies declarative user-defined criteria for creating visualizations. This paper explores the requirements for visualization queries and exemplifies how such queries could be used to drive the generation of gravity contour maps using two popular visualization toolkits. Additionally, the paper highlights the infrastructure requirements that could support visualization queries.

1 Introduction

Past experiences with databases have shown that users can effectively request for data using declaratively languages. For example, relational database users have long relied on structuring their queries in Structured Query Language (SQL), a declarative language that is translated into equivalent relational algebraic operations that actually compute the results [9]. In this paper we define our notion of a *query* as a request for some visualization (i.e., the graphical analogue of some dataset) specified as a conjunction of first-order logic clauses. The clauses specify what visualization to generate (e.g., volumes, contours, surfaces) as well as associated display attributes (e.g., color and opacity); the structure of visualization queries is further discussed in 4. Our notion of a visualization query mirrors that of the relational database querying approach; declarative descriptions of requested visualizations are translated into a pipeline of visualization operations that can be used by appropriate visualization infrastructure to generate the requested visualization. The resultant visualization is then sent back to the user. To contrast, our notion of query is less akin to the querying facilities in information retrieval that retrieve rather than compute answers.

Our work is motivated by our observations that many popular visualization toolkits require that users manually design and configure visualization pipelines that transform their raw datasets into graphical form [13, 6, 18]. A visualization pipeline is a sequence of steps that generates an image or video from stored data. Unfortunately, manually developing an executable visualization pipeline

(i.e., a program) that generates an effective visualization can be challenging for two reasons:

Challenge 1. Understanding visualization processes at the conceptual level. Users need to understand how logical sequences of steps will yield their requested visualization.

Challenge 2. Understanding visualization processes at the implementation level. Users need to be able to translate the sequence of steps into an executable application.

If we understand the logical chaining of concepts (Challenge 1) and how to map these concepts to implementations (Challenge 2), then we can use this knowledge to support visualization queries. In this case, we can pose queries at the conceptual level and machines would know how to map the concepts defined in the query to executable code that can compute the result. To support this environment, we need to understand how visualization toolkits work with data and how users work with toolkits. We must extend well-known visualization models such as "Data State" [4] to include implementation level details about visualization toolkits discussed later in 3. The Data State model, which is further defined in Section 2, defines visualization processes in terms of sequences of *operators* that transform data in a raw state (i.e., value) to its graphical state (i.e., view).

This paper first introduces previous work in modeling visualization processes in Section 2. It follows with a visualization scenario on how users currently generate 2D contour maps in Section 3 and shows how queries can be alternatively used to generate visualizations in Section 4. In addition to defining visualization queries, Section 4 explores requirements for an infrastructure that can support them. The paper closes with a discussion in Section 6 and conclusions in Section 7.

2 Background

Past efforts in defining visualization models have focused on defining concepts necessary for users to better understand visualization processes at a high level. Currently, the two widely accepted visualization models are the data flow model [18] and the data-state model [4]. Although the two models have been shown to be equivalent in terms of expressibility [3], each model has its strengths and weaknesses in terms or providing users with an understanding of a particular visualization technique. Note that expressiveness in this context is different from the *expressiveness* defined by Mackinlay in [14]. 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, we can say a visualization model is "as-expressive-as" another model if for all inputs the two models give exactly the same output [3].

The data flow model emphasizes the processing steps associated with a visualization technique and so provides users with an understanding of a process from an executable standpoint. In fact, many popular visualization toolkits allow users to specify a visualization pipeline from a dataflow perspective [13, 18, 10, 6, 23]. The dataflow technique does not define any standard algorithms or operators, and because of this it may be challenging for users to compare among different visualization techniques described by the dataflow model. For example, users might have to understand that "gridding" and "surface reconstruction" algorithms are conceptually providing the same functionality, although they are named differently. Because the dataflow model does not provide a baseline for comparison, true understanding of a visualization technique may be prohibited for novice users.

The two models however do not define the concept of parameter and so do not provide support for modeling implementation level processes. This is because almost every module in toolkits like GMT and VTK are configurable via parameters. In fact, some GMT modules can take as many as ten different parameters shifting the problem of sequencing the right operators to specifying the correct parameters. Parameters can have drastic effects on the behavior of operators and so it is crucial to understand the role that they play in the visualization process. Our work aims at defining those roles.

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. [1] have described visualization models that explicitly highlight how a particular user's knowledge affects a visualization process. In this context, the users' knowledge about how to choose effective orientations/positions of the data as well as color transfer functions can have dire effects on the final artifact. To compensate for incomplete knowledge, the paper describes a "knowledge-assisted visualization" model where reasoning capabilities may help to alleviate a users' naivety about the visualization process. This assisted knowledge may come from other users explicitly adding the knowledge or by the system automatically inferring the knowledge by using past histories of other users. The infrastructure requirements for supporting our visualization queries, described in Section 4, will need knowledge about different toolkit operators and the associated parameter sets. This kind of visualization knowledge is similar to the knowledge described in [1].

3 Gravity Map Visualization Scenario

2D contour map visualizations generated from gravity data readings serve as models from which geophysicists can identify subterranean features of the Earth. In particular, geophysicists are often concerned with data anomalies, e.g., spikes and dips, because these are indicative of the presence of some subterranean resource such as a water table or an oil reserve. These maps may be used for explorative purposes and thus support the browsing task, which is defined as developing an understanding of unexpected patterns within a collection as noted

by [19, 15].

In terms of visualization, these anomalies or features are best highlighted when the data is presented as a contour map because users can quickly identify where spikes exists by finding a region on the map where the isolines are in close proximity as in Figures 1 and 2. To generate these maps, scientists begin by providing a footprint defined by latitude and longitude coordinates, which indicates what region of gravity data is to be mapped. A sequence execution of tasks specified in Figure 3 describes how a contour map would be generated as a visualization of gravity data readings.

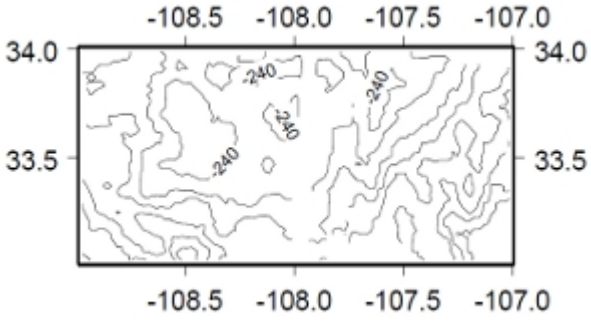

Fig. 1. Generic Mapping Tools (GMT) contour map

Fig. 2. Visualization Toolkit (VTK) contour map

The tasks specified in Figure 3 can be supported by many toolkit operations and this paper discusses two implementations, one using the Visualization Toolkit (VTK) [18] and one using Generic Mapping Tools (GMT) [22]. In the rest of this section, we compare the capabilities of these two toolkits and describe

Fig. 3. Task description on how to build a gravity contour map

Task	*Description*
1. Gather:	Gather the raw gravity dataset readings for the specified region of interest. The data provided by Pan American Center for Earth and Environmental Sciences (PACES) [17] gravity database is tabular and formatted in ASCII.
2. Filter:	Filter the raw gravity dataset readings by removing unlikely and duplicate point values. Unlikely values are those readings that fall outside of some range specified by the scientists.
3. Grid:	Create a uniformly distributed dataset by applying a gridding algorithm. Typically, in order to generate contour lines, the underlying data must be uniformly distributed (i.e., a surface).
4. Contour:	Create a contoured rendering of the uniformly distributed dataset.

how to build the map with each toolkit.

3.1 On the Use of Visualization Toolkits

Visualization toolkits typically provide modular environments from which scientists chain together operators to form executable pipelines that generate visualizations for their data [13, 6]. A goal of these toolkits is to provide users with a common data model from which different operators can be uniquely combined to create novel visualizations. Because users do not have to regard format in these environments once they have translated their data into the tool's data model, they essentially have a great deal of freedom when combining sets of operations. This is true in VTK and evident in almost every VTK program through the required use of "Readers". Readers are responsible for transforming stored data into some VTK specific dataset type, which can then be processed by VTK modules.

Similarly, GMT adopts NetCDF's gridded datasets [16] as its common data model. Many of the GMT visualization modules are configured to work with only binary NetCDF grids. Thus if data is not already in NetCDF, users are responsible for figuring out how to read their data into a NetCDF dataset before it can be processed by GMT modules.

Toolkits have radical differences in terms of implementation, scope, and, performance and thus their differences are not restricted to data formats. On one hand, VTK is available as a set of C++ libraries with support for very high resolution scientific visualizations that can be generated on a cluster. GMT, on the other hand, is tailored to generating geographic maps and available as a set of standalone programs without support for distributed environments. Despite these differences, we observe a significant overlap of visualization capabilities between these two toolkits. A discussion of the differences in generating gravity contour maps in Section 6 will exemplify the differences and similarities.

3.2 Building the Gravity Contour Map in GMT

In GMT, there is great deal of support for GIS-based operations and so the above gravity map task description in Figure 3 can almost be mapped entirely one-to-one with GMT operators. By GIS data, we mean data that is referenced in 2D or 3D space, but is visualized in a 2D projected space (e.g., Mercator) to account for the curvature of the Earth. GMT provides extensive support for working with field data (i.e., tabular data) such finding the min/max and filtering values of these kinds of datasets. In our example, we assume our gravity data is sourced from the Pan American Center for Earth and Environmental Science (PACES) data store [17], which provides access to the data as a set of XYZ records. Note that XYZ data can be broken down into two components, the 2D spatial component (i.e., XY or longitude and latitude) and the associated scalars (i.e., Z) and thusly is regarded as 2D data. Additionally, GMT provides a module to grid XYZ data and output the gridded result as standard GIS formats such as "ESRII grid", which can then be imported by a number of GIS applications.

To build a gravity contour map in GMT, a pipeline sequence of GMT executables would need to be piped together as a sequence of operator invocations, such as in Figure 4 below. The resultant visualization from the GMT pipeline is seen in Figure 1.

Fig. 4. GMT Pipeline that visualizes gravity data as a 2D contour map

Operator	*Associated Parameters*	*Description*
1. xyzFilter.exe	columns to filter	filters the XYZ colums to be mapped
2. xyz2grd.exe	grid spacing, search radius	generates a uniform 2D grid of data from XYZ
3. grdContour.exe	contour interval	generates isolines from 2D gridded data

The final output from *grdContour* operator is a contour map projected in Mercator with a labeling of the x and y axis in terms of latitude in longitude. The added bonus of getting the axis labeled comes from the fact that GMT is very GIS oriented and does such labeling by default, although GMT can be used to map arbitrary data as well. In this example, a user would be responsible for writing a script or workflow to coordinate the execution of the above pipeline. This is not possible if the user does not understand how these executable operators map to the conceptual tasks described at the beginning of Section 3.

3.3 Building the Gravity Contour Map in VTK

The visualization toolkit provides rich support for generating 3D scientific visualizations, but can still be leveraged to generate 2D visualizations. One caveat when using VTK is when rendering 2D data registered in some coordinate system. For example, operators that can translate from 3D Cartesian coordinates to 2D projected coordinates in "Mercator" are not supported in VTK standard

Fig. 5. GMT Pipeline that visualizes gravity data as a 2D contour map

Operator	*Associated Parameters*	*Description*
1. DataObjectToDataSet:	X-col, Y-col, Z-col, Scalar-col	creates 3D points from XYZ and associates a scalar with each point
2. vtkShepardMethod:	search radius	generates 3D grid from 3D points
3. vtkExtractVOI:	Volume of Interest (VOI)	extracts a subvolume from a 3D grid (can also extract a 2D slice)
4. vtkContourFilter:	number of lines, interval	generates isolines from a 2D/2D grid (in our case the grid is 2D)
5. vtkPolyDataMapper:	NONE	renders the isolines using OpenGL
6. vtkJPEGWriter:	magnification, quality	writes the rendered data to a JPEG image

distribution. Figure 5 below is a pipeline that generates a gravity contour map in VTK as shown in Figure 2.

The VTK pipeline in Figure 5 is more complex than the GMT version in Figure 4 because VTK is a more general-purpose visualization toolkit and thus must be manually tailored to work with 2D data. The first operator in the sequence in Figure 4, *, DataObjectToDataSet*, is responsible for parsing the XYZ text file and generating VTK 3D point data and associating the Z scalars to each point. The parameters to this operator are very important because they associate the XYZ fields (i.e., lon lat, and scalars) to point coordinates (X,Y,Z) and associated scalars. However, VTK points must be specified in 3D although our data is in 2D. Thus we must meet the operator requirement by inputting an array filled with zeros for the Z component of the VTK points.

The second operator in the VTK pipeline sequence, "vtkShepardMethod", implements the "gridding" task of our conceptual gravity map pipeline in Figure 3 and creates a uniform 3D grid of data from the unstructured points. At this point, however, we have a 3D grid although we want to generate isolines (2D) not isosurfaces (3D). Thus we need to obtain a 2D grid from the 3D grid through use of the operator "vtkExtractVOI", which can extract 2D slices from 3D grids. At this point, the contouring filter "vtkContourFilter" can be applied to the 2D grid computed from "vtkExtractVOI" to generate the isolines which comprise our map view.

4 Visualization Queries

We define a visualization query as a conjunction of first order logic statements that contains the following clauses:

1. a single *(hasView -X ?V)* clause that describes the relation between the requested visualization X and some view it encodes *V*. The views that are bound to V represent graphical abstractions that are rendered in the visualization.
2. a set of optional clauses describing the attributes of the view

3. a set of clauses declaring the semantic type and format of the data to be visualized

Visualization queries borrow the notion of query from logic based languages such as Prolog [11] and more specifically DataLog. The "-" and "?" variable modes are also borrowed from the Prolog language definition and denote whether a variable should be bound or unbound at the time the clause is evaluated. The symbol "-" specifies that the variable should be unbound whereas "?" specifies that the variable can be both bound or unbound, in which case it functions as a wild card (i.e., can be bound to more than one value). Our *hasView* predicate requires that the visualization X variable be left unbound. The goal of a querying answering system is then to generate a visualization that can be bound to X. At an implementation level, the object bound to X is a reference to some visualization such as a URL, rather than the visualization itself.

In order to specify meaningful visualization queries, users must know about the different kinds of views, their associated attributes, and the format and type of dataset being visualized. This is similar to how users must know the names of tables and corresponding field names when composing a query in SQL. We assume that the user has access to this meta-information. For example, we expect that users may already know the set of attributes associated with a particular view. Our notion of *attribute* is equivalent to the definition of "display attributes" in [20] that control a view's spatialization, timing, color, and transparency.

Considering the gravity contours map tasks in Figure 3, the contour map, visualization X, might be requested by the Knowledge Interchange Format (KIF) [12] statement as shown in Figure 6.

Fig. 6. Example Query for an *gravity map* visualization.

(and (hasView X isolines) (hasAttribute isolines interval) (hasAssignedValue interval 10) (hasProjection isolines mercator)	*View and Attibute Predicates*
(visualizationOf X gravityData.2d) (hasType bouguerAnomaly) (hasFormat ascii-tabular))	*Format, Source and Type Predicates*

The query in Figure 6 is requesting the generation of *visualization X*, which supports an "isolines" view with an annotation interval of 10 units. In this example, the user has a desired view (i.e., isolines), but the *hasView* predicate can be evaluated when V is also left unbound. When left unbound, any system capable of reasoning with visualization knowledge will try to bind V to any view that can satisfy the other criteria. This has the potential to generate visualizations that are meaningless and ineffective to the user. For example, the variable V could be bound to a *2d-delaunay-triangulation* view in which case which case the gravity Z scalars would be disregarded in the rendering.

The *hasProjection* clause specifies that the isolines be projected onto a plane using the Mercator projection. This is not an intuitive specification since it is a GIS-specific statement that might not be easily satisfiable by all visualization toolkits, in particular VTK. Additionally, the query specifies that the data being visualized is available in the file *gravityData.2d*, is of type *bouguerAnomaly*, and is encoded in a *asciiTabular* format. The view and attribute predicates serve as goal whereas the format, source, and type predicates serve as starting point for any system answering the query. The task then becomes searching for a sequence of steps that can translate the data in its given form (i.e, asciiTabular) into the requested view (i.e., set of isolines). In GMT, we need to figure out how to transform the gravity data into a "binary grid" that can be processed by the contour.exe operator as shown in Figure 4. In VTK, we must figure out how to transform the gravity dataset into a 2D "vtkGrid" which can then be processed by the operator "vtkContourFilter" as shown in Figure 5. The following sections explore the requirements of an environment suitable for answering visualization queries, such as the gravity map query.

5 An Infrastructure that can support Visualization Queries

Now that we understand how to build gravity contour maps in both VTK and GMT, we can explore how a visualization query might be able to drive the generation of these pipelines automatically in terms of toolkit operations that can actually be used to generate visualization results. We can also begin to identify the components needed to support this visualization synthesis. We will characterize this environment by considering how to support the gravity map visualization query defined in Section 4.

5.1 A General Purpose Visualization Model

We will begin the characterization of our visualization query environment by noting the objects composing our universe (i.e, objects of our first order logic clauses), which correspond to visualization concepts including *View*, *Format*, and *Type*. In Figure 6, those concepts correspond to "isolines", "ascii-tabular", and "bouguerAnomaly" respectively. In order to be able to construct these queries, users will be responsible for knowing what view they want their data represented in and both the semantic type and format of their datasets similarly to how users must know certain properties of a database (e.g., tables and fields) in order to construct SQL queries.

In addition to the information specified in visualization queries, we need to understand sequences of operators, supported by some visualization toolkit, which can generate the requested view. In order to build this sequence we need to know the computational constraints imposed by the different toolkit operators. Thus we perceive the need to define a general purpose visualization model that is built on visualization concepts such as *Operations*, *Operators*, and *Parameters* and their interrelationships.

Operations, Operators, and Parameters We can leverage concepts from existing models such as Chi's "Data State" model [4, 2, 3] that introduces an operator driven visualization model that is suitable for understanding visualization processes at a conceptual level, although to the best of our knowledge, only one toolkit actually implements the model [7]. We can build upon Chi's operators, which are defined as "any user interaction whether based on direct manipulation of other interactions", by defining an *operation*, which is a high level concept that represents the use *parameters* to control some aspect of an *operator's* [4] behavior. Operations are what compose visualization pipelines such as the gravity map pipelines of VKT and GMT shown in Figures 4 and 5. Our notion of parameter is based on the definition of parameter defined in the Unified Modeling Language (UML) meta-model found in [21].

Going into further detail, an operator can be a *transformer* that is responsible for encoding a dataset in some format into different format. In VTK, the "DataObjectToDataset" operator is a kind of transformer, because it transforms 2D points data encoded in *ascii-tabular* into *vtkPoints*. Transformers are often necessary because most toolkits are not "format agnostic" [8], and the user is responsible for "corralling their data into a toolkit specific data representation". Thus typically, an essential part of most visualization pipelines is a sequence of transformers that move a source dataset from one form a format into a different format than can be consumed by operators of some toolkit. Knowing the input and output format of operators is therefore necessary for our general purpose visualization model.

Alternatively, an operator can be a *renderer* that generates some view, whether the view is geometric such as volumes and contours or graph-like such as trees and networks (i.e., scientific views vs information views). A view can support display attributes [20] such as color, contour intervals, opacity, lighting, position. In many toolkits including VTK and GMT, these display attributes are controlled by sets of parameters that are input to some operator. For example, in VTK the volume generator filter accepts an opacity parameter. In VTK, the opacity parameter is specified by a piecewise function that controls the mapping of scalars to opacity level.

Semantic Type The notion of "semantic type" is often neglected in both conceptual and executable visualization models and toolkits. Our proposed visualization model relies on type to ensure that requested visualizations satisfy the "effectiveness" criteria described by Mackinlay[14], in cases when the user doesn't specify the attribute clauses in the query. For example, the *contour interval* parameter corresponding to the *contouring* operator should be set to "10", when contouring data of type "gravityData". Typically, gravity values will not range in the order of pow(10,2) within any given region, so a value of "10" will generate contours lines that are not too cluttered but still provide high resolution of the data being visualized. Without type knowledge, the correct value of some parameters may not be set, leading to ineffective visualizations. These semantic types can be defined by some domain ontology (e.g., ESIP data-type ontology

[5]) and integrated into our "open" visualization model

5.2 Building Visualization Pipelines From Visualization Queries

Visualization queries are mapped to sequences of operations, similarly to how SQL queries are translated to sequences of relational algebra operations. In our visualization scenario, the sequence of operations forms a pipeline, meaning that the output of one operation feeds into the input of the next operation in the sequence. Thus we need to rely on the input and output format/type information associated with operators to assure that we construct pipelines that can actually execute.

Provided that there was a reasoning system that could work with all the knowledge about our general purpose model we could synthesize visualization pipelines provided the information provided in a visualization query. For example, given the visualization query requesting the gravity contour map in Figure 6, the system could deduce that the final operator of the resultant pipeline would contain the VTK operator "vtkContourFilter" because this operator is responsible for generating isolines. Additionally, we can conclude that the parameters to operator will include a "interval" parameter set to a value of "10".

However, because most toolkits are not format agnostic, there still exists the problem of selecting a sequence of transformers that copy the input data into a form that can be operated on by "vtkContourFilter". Provided that our data is of type *gravityData* and encoded in *ascii-tabular* format, we can automatically find a sequence of transformers that will convert the input data into a VTK "grid", which can then by transformed into the isolines view requested in Figure 6. We can chain together transformers with the conditions that:

1. the first transformer of the sequence operates on *gravityData* encoded in *ascii-tabular*
2. the final transformer of the sequence outputs *vtkpolydata*.
3. the output of a transformer in the sequence must match next transformers input format/type requirements

6 Discussion

Referring back to the VTK pipeline in Figure 5, the resultant visualization does not satisfy the clause "hasProjection(X, Mercator)" of the query specified in Figure 6. The output visualization from the VTK pipeline will not be as effective as the GMT visualization because of a number of additional reasons including:

1. VTK doesn't automatically label axes
2. VTK renders the geometries using exact Cartesian coordinates but does not come with a standard operator to project onto other coordinate systems
3. more effort is required to visualize 2D data in VTK because most of the operators are tailored for 3D data (i.e., scalars associated with 3D points)

Despite the additional effort required to build 2D contour maps of field data in VTK, the exercise helped us verify that visualization queries can be answered by pipelines of operators supported by different toolkits. The ability to provide answers supported by different visualization toolkits sets our work apart from similar visualization efforts such as Protovis [8]. Protovis is a Java based visualization toolkit that distinguishes between visualization specifications and visualization execution, allowing users to request for visualizations declaratively as well. Protovis execution, however, is supported by only a single engine (i.e., the Protovis system). Our goal with the use of visualization query is to target a wide variety of different visualization toolkits with a single visualization request.

Additionally, Protovis takes a graphical approach to data visualization, composing custom views of data with simple graphical primitives like bars and dots. Our proposed visualization model, composed of *Views*, *Operators*, and *Parameters*, does not support graphics at this low a level and thus may not be able to provide users with the same level on control as provided by Protovis.

7 Conclusions

We have defined a notion of visualization query. The notion leverages the following concepts related to both information visualization and scientific visualization: operations, operators, parameters, and views. Through the use of an example in the geophysics domain, the paper demonstrated how a conceptual query was used to specify user's requirements for the visualization of data from a gravity reading database. Our conceptual query is declarative in nature because the user was not required to specify how toolkit operations were used to generate visualizations. The paper further demonstrated how parameters specified at the conceptual level were used in the running example to actual implement the query by two well-known visualization toolkits: GMT and VTK.

Finally, the paper discussed some design strategies of an infrastructure for the execution of visualization queries through the translation of the queries into multi-toolkit-based executable visualization pipelines. For example, we consider the use of the following: a general purpose visualization model; and mechanisms for assigning richer semantic annotations to data enabling visualization applications to better understand data and to thus better tailor visualization capabilities to data.

Acknowledgements

We would like to acknowledge the support for this work granted by the CyberShare Center for Excellence and Department of Homeland Security.

References

1. Min Chen, David Ebert, Hans Hagen, Robert S. Laramee, Robert van Liere, Kwan-Liu Ma, William Ribarsky, Gerik Scheuermann, and Deborah Silver. Data, infor-

mation, and knowledge in visualization. *IEEE Comput. Graph. Appl.*, 29(1):12–19, 2009.

2. Ed H. 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.
3. Ed H. 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.
4. 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.
5. Earth science information partners (esip) federation datatype ontology. http://wiki.esipfed.org/index.php/Data-Service-Ontologies.
6. David Foulser. Iris explorer: a framework for investigation. *SIGGRAPH Comput. Graph.*, 29(2):13–16, 1995.
7. Jeffrey Heer. Prefuse: a toolkit for interactive information visualization. In *In CHI 05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 421–430. ACM Press, 2005.
8. Jeffrey Heer and Michael Bostock. Declarative language design for interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16:1149–1156, 2010.
9. International Organization for Standardization (ISO). *SQL Part 2: Foundation (SQL/Foundation)*, 2008.
10. Philip L. Isenhour, James Bo Begole, Winfield S. Heagy, and Clifford A. Shaffer. Sieve: A collaborative interactive modular visualization environment, 1995.
11. ISO. *ISO IEC 13211-1 Prolog Part 1.* ISO, 1995.
12. Knowledge interchange format draft proposal. http://logic.stanford.edu/kif/dpans.html.
13. 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.
14. Jock D. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5(2):110–141, 1986.
15. Gary Marchionini. *Information Seeking in Electronic Environments.* Cambridge University Press, 1995.
16. Network common data form netcdf. http://www.unidata.ucar.edu/software/netcdf/.
17. Pan merican center for earth and environmental studies. http://research.utep.edu/Default.aspx?alias=research.utep.edu/paces.
18. 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.
19. 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.
20. M. Tory and T. Moller. Rethinking visualization: A high-level taxonomy. pages 151 –158, 2004.

21. Unified modeling language (uml) infrastructure specification 2.3. http://www.omg.org/spec/UML/2.3/.
22. P. Wessel and W. H. F. Smith. New, improved version of generic mapping tools released. *EOS Transactions*, 79:579–579, 1998.
23. Brian Wylie and Jeffrey Baumes. A unified toolkit for information and scientific visualization. In *VDA*, page 72430, 2009.

This article was processed using the LaTeX macro package with LLNCS style