

The Tulip 3 Framework: A Scalable Software Library for Information Visualization Applications Based on Relational Data

David Auber, Daniel Archambault, Romain Bourqui, Antoine Lambert, Morgan Mathiaut, Patrick Mary, Maylis Delest, Jonathan Dubois, Guy Melançon

▶ **To cite this version:**

David Auber, Daniel Archambault, Romain Bourqui, Antoine Lambert, Morgan Mathiaut, et al.. The Tulip 3 Framework: A Scalable Software Library for Information Visualization Applications Based on Relational Data. [Research Report] RR-7860, INRIA. 2012, pp.31. hal-00659880

HAL Id: hal-00659880
https://inria.hal.science/hal-00659880

Submitted on 13 Jan 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

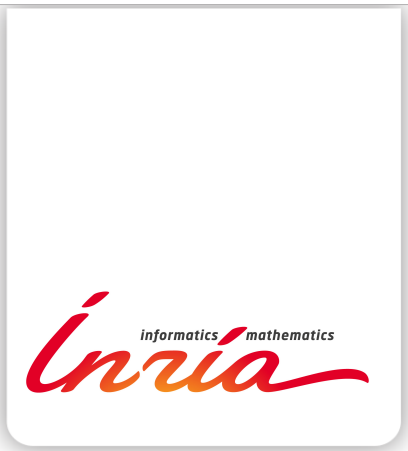


The Tulip 3 Framework: A Scalable Software Library for Information Visualization Applications Based on Relational Data

D. Auber, D. Archambault, R. Bourqui, A. Lambert, M. Mathiaut, P. Mary, M. Delest, J. Dubois and G. Melançon

RESEARCH REPORT
N° 7860
January 2012
Project-Team GRAVITE

ISSN 0249-6399 ISRN INRIA/RR--7860--FR+ENG



The Tulip 3 Framework: A Scalable Software Library for Information Visualization Applications Based on Relational Data

D. Auber *†, D. Archambault, R. Bourqui, A. Lambert, M. Mathiaut, P. Mary, M. Delest, J. Dubois and G. Melançon †

Project-Team GRAVITE

Research Report n° 7860 — January 2012 — 28 pages

Abstract: Tulip is an information visualization framework dedicated to the analysis and visualization of relational data. Based on a decade of research and development of this framework, we present the architecture, consisting of a suite of tools and techniques, that can be used to address a large variety of domain-specific problems. With Tulip, we aim to provide the developer with a complete library, supporting the design of interactive information visualization applications for relational data that can be tailored to the problems he or she is addressing. The current framework enables the development of algorithms, visual encodings, interaction techniques, data models, and domain-specific visualizations. The software model facilitates the reuse of components and allows the developers to focus on programming their application. This development pipeline makes the framework efficient for research prototyping as well as the development of end-user applications.

Key-words: Graph visualization framework, scalable graph visualization, information visualization, interactive graph visualization, graph drawing, graph hierarchy interactive visualization

* Contact author: David.Auber@labri.fr

† D. Auber, R. Bourqui, A. Lambert, M. Mathiaut, P. Mary, M. Delest, J. Dubois and G. Melançon are with CNRS UMR 5800 LaBRI and GRAVITE INRIA Bordeaux - Sud-Ouest, E-mail: {auber, bourqui, lamabert, mathiaut, mary, maylis, melancon}@labri.fr. Daniel Archambault is with UCD Dublin, Clique Strategic Research Cluster, E-mail: daniel.archambault@ucd.ie.

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

351, Cours de la Libération
Bâtiment A 29
33405 Talence Cedex

The Tulip 3 Framework: A Scalable Software Library for Information Visualization Applications Based on Relational Data

Résumé : Tulip is an information visualization framework dedicated to the analysis and visualization of relational data. Based on a decade of research and development of this framework, we present the architecture, consisting of a suite of tools and techniques, that can be used to address a large variety of domain-specific problems. With Tulip, we aim to provide the developer with a complete library, supporting the design of interactive information visualization applications for relational data that can be tailored to the problems he or she is addressing. The current framework enables the development of algorithms, visual encodings, interaction techniques, data models, and domain-specific visualizations. The software model facilitates the reuse of components and allows the developers to focus on programming their application. This development pipeline makes the framework efficient for research prototyping as well as the development of end-user applications.

Mots-clés : Graph visualization framework, scalable graph visualization, information visualization, interactive graph visualization, graph drawing, graph hierarchy interactive visualization

1 Introduction

Although this paper presents a system and discusses its design, its content goes much further. In a sense, this paper is a position paper following ten years of lessons learned working in graph visualization, developing new visualization techniques, and building systems for users. The strategy we have adopted is to develop, maintain, and improve the Tulip framework[1] aiming for an architecture with optimal data structure management from which target applications can be easily derived. The benefits of our strategy have paid off on several fronts. We have used the framework to demonstrate the *Reproducibility* of work published by others, allowing us to experiment with and validate our work. The architecture has promoted *Extensibility* and *Reusability* of our results and those of other researchers as discussed in detail in forthcoming sections. Tulip has facilitated *scientific collaboration* and *technology adoption*. The framework serves as a tool to demonstrate our expertise and know-how when interacting with scientific collaborators or end-users. As we shall argue, the evolution path of our framework brings it into full coherence with Munzner's nested model [37], and serves all facets of InfoVis guiding the creation and analysis of visualization systems.

Tulip is one of the very few systems that offer the possibility to *efficiently* define and navigate graph hierarchies or cluster trees (nested subgraphs). This technique has been a central visual paradigm in our group, as it often provides answers to data analysts. The reason is quite simple: large graphs must be clustered to reduce visual complexity, turning the data exploration process into one involving a hierarchy built by a clustering algorithm. Hence, Tulip's low level data structure was designed since its birth to support the creation of nested and/or overlapping subgraphs, integrating at the heart of the system a property heritage mechanism that both provides coherence and optimal space usage.

Tulip started after David Auber decided to enter the huge graph visualization arena [9, 10, 12]. The library was designed to deal with graphs (relational data), focusing on graph topology as the main ingredient for visual encodings and mainly exploiting node-link diagrams (points and straight lines) as a central visual metaphor. The framework was primarily designed to challenge *scalability*; its core architecture and low level data structures were optimized to reach ambitious goals in terms of graph size (nodes and edges) that could be handled and visualized. After these initial efforts, Tulip found a place within our research group and soon became an everyday experimental tool. Because data analysis and combinatorial mathematics are companion fields to graphs visualization, Tulip included a rather exhaustive list of node and edge metrics that could then be mapped to color or size. Obviously, Tulip initially served as an experimental framework from which the design of drawing algorithms and visualization techniques were developed, tested, and validated. From this point of view, Tulip can certainly claim to be part of the champion's club of state-of-the-art graph visualization libraries and software.

The growth of our community helped us gained visibility, and we were soon asked to cooperate with end users to build visualization applications: navigating protein interaction networks [32], producing automated drawing for secondary RNA structures [14], visualizing software reverse engineering graphs [25], social

[1] See `http://www.tulip-software.org`

networks [13] or air passenger traffic [42]. The graph hierarchy paradigm residing deep within Tulip was later fully exploited by the work of Archambault, Munzner and Auber [4, 6, 5, 7, 8]. We also became aware of the use of Tulip by others (see [40, 21] for instance[2]).

Graph Visualization is often a possible avenue for data analysis as seen from these numerous collaborations. However, once the graph has been established, the visualization process often needs to be supported by techniques in graphical statistics or visual data mining. Tulip has been extended to offer visual encodings for relational as well as non-relational data. The libraries have matured from their algorithmic-centered viewpoint towards a data analysis dashboard combining different visualization techniques and support for visual analytics.

The Tulip architecture has been designed to promote extensibility and reusability of results. As such, from a software engineering perspective, it heavily relies on object composition rather than inheritance. Even if object composition is often more complex for the programmer, it considerably reduces code duplication and dependencies between modules. We are constantly improving and refactoring our library to minimize the code duplication and re-implementation, to ease the addition of future research results, and to preserve architecture scalability.

Tulip offers a software library is in coherence with Munzner's nested model and has software support for validation at any level of this model. Our paper is thus structured to illustrate this property. Section 2 describes previous and related software systems that inspired the design of many parts of Tulip. Section 3 describes the architecture of the Tulip libraries and software. In this section, we describe elements that support each level of validation in Munzner's nested model: algorithm plug-ins (section 3.1.2) provide support for validating *algorithm design*, views and interactors (section 3.3.1 and section 3.3.2) provide support for validating *encoding/interaction technique design*, and perspectives (section 3.3.3) provide support for validating *data/operation abstraction design* and *domain problem characterization*. Section 4 presents applications, two to support information visualization group needs and one to support a domain-specific application, where Tulip was found to be helpful. Finally, section 5 presents some conclusions and future work.

2 Background and Related Work

Developing a framework over an extended period often means being compared to or challenged by competitor systems and libraries. This section presents a representative subset of the libraries that are closest in spirit to our work. We briefly discuss the philosophy or underlying principles of each, contrasting them to Tulip. Many of these competitors[3] have been benchmarked against Tulip in terms of scalability, one of Tulip's strong points.

[2] We should also mention that Tulip is actually distributed in several Linux releases.

[3] Our group together with collaborators maintains a comparative list of existing graph visualization software. See the URL `http://gvsr.polytech.univ-nantes.fr/GVSR/`

2.1 Libraries

LEDA/AGD/OGDF [35, 39, 24] : The LEDA/AGD/OGDF series of graph drawing libraries were built to provide a collection of efficient graph drawing algorithms. These libraries include some of the most powerful, sophisticated, and complex algorithms to produce graph drawings. However, the aim of these libraries is to draw graphs – that is, to decide the positions of nodes in the plane. This library tends not to focus on a fully integrated information visualization library. Furthermore, these libraries tend to focus on graph connectivity. Extra information linked to the nodes and edges of the graph are difficult to integrate into the visualization process. That said, LEDA/AGD have inspired our work (see section 3.1.2).

GraphVis [26]: This library is similar to OGDF and has support extrinsic data in its graph drawing algorithms (for instance, labels, size, orientation of graph elements are all supported). GraphVis has been successful from both an end-user and InfoVis community member perspective. It offers one of the best solutions for drawing hierarchical (directed acyclic) graphs which is state-of-the-art in hierarchical graph drawing. However, the library does not focus on fully integrating its algorithms into a fully functional information visualization system.

VTK/Titan [43, 47]: VTK is the standard library for producing applications supporting scientific visualization techniques. Recent developments of this library extend its scope to information visualization. With the integration of VTK and Boost [4], the latest versions support many information visualization techniques, even though VTK was not originally designed to support the visualization of abstract (non-geometric) data. The original strength of the library was its efficient rendering of meshes in three dimensions and optimizations can be made under the assumption that most information visualization techniques are focused on rendering information in two dimensions. However, information visualization often focuses on user interaction and visual data manipulation requiring efficient methods for tracking changes to the data needs to be supported, and this library does not appear to directly support this functionality. We compare the performance of the library to the Tulip one in section 3.2.

2.2 Toolkits

Toolkits offer users an environment for the development of InfoVis applications. They offer an off-the-shelf data import/storage solution and often include a variety of widely used graph layouts and node/edge metrics. The two toolkits we comment on here primarily support the design, development, and validation of new interactive visualization techniques, rather than offering sophisticated support for graph drawing algorithms.

Prefuse [31]: This framework provides a comprehensive set of interactive information visualization techniques. Its clever design and management of interaction make this toolkit one of the most widely used for information visualization applications. On the other hand, the toolkit supports only a few graph drawing algorithms and node/edge metrics. The latest "pure" Prefuse release goes back to 2007, but recently Prefuse/Flare targeted the toolkit towards web-based InfoVis. In term of scalability, efforts have been made by the authors to

[4] www.boost.org

provide an efficient JAVA based implementation. However, Tulip can handle larger data sets. For instance, a graph of 300,000 nodes and 600,000 edges take 1.2Gb in Prefuse when it takes only 170Mb in Tulip. Furthermore, interaction with such a graph is almost impossible in Prefuse where it still reasonable with Tulip.

InfoVis Toolkit [28]: The Infovis Toolkit shares similarities with Prefuse and offers a comprehensive set of information visualization techniques. For instance, node link diagrams, tree maps or matrix views. As such, it has many of the advantages and disadvantages of Prefuse. The toolkit supports few but relevant graph drawing algorithms and metrics. The last release of this toolkit was in 2006. The concept of multi-views implemented in this framework have inspired a similar design in Tulip.

2.3 Software

ASK-GraphView/CGV [1, 46]: This software system shares an important feature with Tulip as it relies on the computation of subgraph hierarchies and implements multi-scale graph drawing techniques to explore large data sets that do not necessarily fit into main memory. ASK-Graph view is part of the few scalable graph visualization frameworks. However, it essentially offers a single visualization technique relying on multi-scale graph drawing as a central visual paradigm.

GUESS [2]: GUESS uses a scripting language to perform basic tasks (searching and filtering, etc.). This scripting language is very useful and powerful users with programming experience in Python. However, direct manipulation of the data through interactive techniques may be preferable for some users, which is the focus of Tulip. Through the plug-in architecture of Tulip, it would be possible to implement a scripting language such as this one, but as of yet, we have not implemented such a feature. Also of concern is the scalability of an interpreted scripting language on very large data set sizes.

Pajek [19]: The Pajek software focuses on the analysis of large graphs, providing several powerful tools such as k-core computation, eccentricity and others. In earlier versions, Tulip shared many similar ideas with this software. However, few visualization techniques outside graph are supported. Also, the software is not open source, making it difficult to use for information visualization research.

Cytoscape [44]: Cytoscape is dedicated software for visualization of networks in Biology. In many ways, it shares many ideas with the Tulip Framework. However, it is primarily focused on biological networks and can have scalability problems. For instance, loading and displaying a grid graph having 10000 nodes and 20000 edges requires 1.5 Gb in Cytoscape where it only requires 98Mb with Tulip.

2.4 Software Engineering Background

Designing a comprehensive set of information visualization techniques for relational data that is always evolving requires a robust software engineering methodology. In the case of Tulip, we use the **agile software method** [34]. The first principle of this method is to provide the continuous delivery of valuable software to end user. In order to achieve this goal, the client and developer

must work closely together. In our case, the first client is our information visualization research team and our collaborators.

A pair of principles taken from the extreme programming method [20] is simplicity and the courage. **Simplicity** mandates that complex behavior should only be implemented when it is needed, and **courage** requires one to rewrite code from scratch, if necessary. Of course, complete re-implementation is an extreme case, and by correctly applying design patterns limits the application of necessary changes, in most cases, to code refactoring [29].

Tulip has been developed and maintained with these principles in mind. Over time, we have integrated all necessary tools to support all the research and projects completed to date. At the end of this development process, we have software of a reasonable code length, using almost all of the well known design patterns.

Design patterns can be viewed as building blocks for software projects. Different design patterns are used for different purposes, and a good software architecture must use good patterns when needed needed. Design patterns can be classified into three main groups. **Creational patterns** abstract away the instantiation of objects. We frequently use this design pattern in the Tulip meta-model and for the plug-in mechanism. **Structural patterns** model the relationship between entities. We use this design pattern for efficient data storage, to implement the flexible rendering engine. **Behavioral patterns** model the communication between entities. We use them for the design of interaction, synchronization, and to model specific user tasks. Throughout the paper, we use design pattern terminology to describe the Tulip architecture with a short summary of the pattern to provide intuition about how it is used. A reader who is interested further could look at one of the textbooks on the subject [30] for further details.

3 Architecture overview

The Tulip framework consists of four packages. The first package, the core of the Tulip library, provides an efficient data structure designed for abstract data visualization. The second package is a complete OpenGl [5] rendering engine tailored for information visualization techniques. The third package is a library of GUI components created using the Qt-Nokia [6] library. Finally, Tulip software is an application where one can embed their algorithm, visualization technique, or complete information visualization system. Figure 1 summarized the connections between these different libraries. In the following we detail the first three packages of our architecture.

3.1 Tulip Core

The Tulip Core library was created for the purpose of visualizing data sets consisting of entities and the relationships between them. It enables to store into memory in an efficient way these entities/relations as well as attributes attached to them. Furthermore, it provides the necessary functions to access to these data and standard useful algorithms. For instance, it includes function to

[5] www.opengl.org
[6] qt.nokia.com

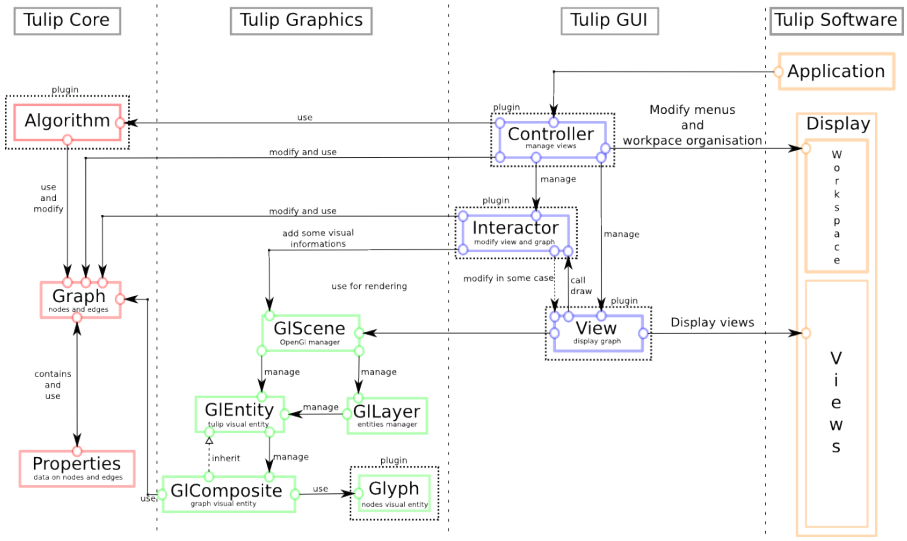


Figure 1: Tulip architecture overview. The Tulip framework consists of four packages. **Tulip core** provides an efficient data structures for relational data. **Tulip graphics** is a complete OpenGl rendering engine. **Tulip-GUI** is a collection of widgets built on top of Qt-Nokia library for the purpose of information visualization. Finally, **Tulip software** is an application for embedding algorithms, visualization techniques/interaction, and complete information visualization systems. All theses packages can be dynamically extended through the plug-in architecture of Tulip.

test whether or not a graph is planar or to compute a uniform quantification of a set of values.

The Tulip core library also integrates a generic plug-in mechanism [17]. It is used many times in our library to enable easy extensions of our framework. The principle of that plug-in mechanism is to enable each plug-in to specify their input/output requirements as well as their dependency with other plug-ins. Similar to what is done with Java Beans, we are able to call these plug in directly in a program or to use them directly through an automatically generated user interface. Furthermore, since plug-ins are dynamically loaded, dependency mechanism enables us to check the coherence of a set of plug-in.

In the following we describe the Tulip meta model that is from our point of view, the part that differentiate the most Tulip from all other Information Visualization system or libraries. For more details on the basic data structure or functions (matrix, convex hulls etc...) provided by Tulip the reader could have a look to the developer manual.

3.1.1 Meta-model III

Based of the previous Tulip version [12], the Tulip meta-model III focuses on minimizing the amount of memory used while providing efficient operations on the data set. The original idea behind the data structure was to manage high-level operations used in the visual analysis process in the data structure.

Integration of all these operations provides a global optimization during the interactive exploration of abstract data.

As shown in figure 2, the Tulip meta-model user only has access the class called **Graph**. In terms of design pattern terminology, the class is a **facade**. This facade provides simplified and centralized access to a set of complexly interacting classes. The programmer does not need to understand the behavior of the objects they manipulate through the facade. Furthermore, it eases the implementation of data storage optimizations to the library as external modules are not directly accessed. One should note that this facade can be used even when working on non-relational data. This property is due to the fact that a graph data structure with an unbounded set of attributes is extremely versatile and allows to store a wide variety of data (relational, multi-dimensional, geospatial, etc...). In the following section, we present some of the operations provided by this facade.

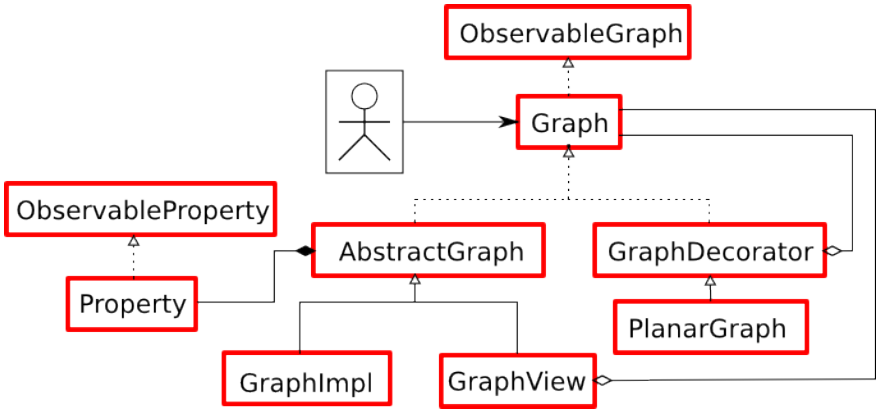


Figure 2: Overview of the meta-model class diagram. Instead of providing a complex set of classes to programmers to use, The Tulip philosophy is to provide centralized access to the data structure through the **Graph** interface. This approach the implementation of an optimized and extensible data structure.

Subgraph hierarchy: One of the first requirements was to provide efficient managements of subgraphs. As a subgraph generalizes the notion of a sub set to relational data, it is often used in graph visualization systems that follow the "overview first, zoom and filter, detail on demand" *Shneiderman mantra* [45]. In figure 2, we see that the facade currently uses two classes. **GraphImpl** is responsible of storing the entities and relations while **GraphView** is responsible of storing subgraphs by using a filtering mechanism on a **Graph**. This approach is efficient in terms of memory, because, in most cases, storage needed for entities and relations in a filter can be done in a single bit (worst cases appear when fragmentation of these indexes are maximal). Furthermore, when a subgraph structure is implemented with filtering, entities and relations used are exactly the same. Thus, no overhead is required for correspondence between entities and relations and their subgraphs. To guarantee the coherence in the subgraph hierarchy, all modification operations on a subgraph apply recursively to sub-subgraphs or its super graphs when necessary. Using this implementation allows the tulip framework to a large number of subgraphs. Using the current imple-

mentation, a graph having 1,000,000 nodes and 5,000,000 edges with 200,000 subgraphs requires 825 Mb on a 64-bit architecture. If one is only interested in graph partitions, where elements must be strictly contained in a subgraph and all its ancestors to the root, this data structure can be optimized. Tulip does not support this optimization as it would limit visualization techniques for overlapping sub-graphs and clusters. HGV [41] does implement this efficient data structure, and an interested reader could get more details there.

Property sharing: Our second requirement was to support storing an unbounded number of properties, or attributes, on graph elements. In the case of properties, the philosophy of Tulip is to not store them inside the entities and relations, but to have a single object for each property. Even if this data structure is slightly less intuitive for a programmer, this choice is necessary to enable global optimization and increase cache hits during iteration of entities (especially during rendering). This idea is also used in the IVTK [28] framework. In order to enable sharing of properties between subgraphs, we provide an inheritance mechanism for properties. As shown in figure 3, each subgraph inherits its super graph properties and can also redefine or create is own properties, similar to the inheritance mechanism in object oriented languages. Finally, the model integrates a widget similar to the virtual tables function to optimize access to properties when dealing with a deep hierarchy of subgraphs. In all the visualization technique and system we have developed, this property sharing mechanism has been key in providing overview+detail implementations and for synchronization.

Aggregation: The third key feature is to enable hierarchical aggregation [27] of entities/relations, and the Tulip meta-model III has been extended and optimized for this purpose. As presented in [16], the subgraph hierarchy presented above can support the efficient aggregation of subgraphs. However, after applying this technique to several multi-scale problems [8, 7, 22], we have integrated into the facade accessors to meta-information graph elements that are stored in the memory (GraphImpl). This solution memory overhead when compared to [16] but enables independence from the meta-graph construction order and helps support de-aggregation operations. We also introduce aggregation functions in order to be able to modify the way aggregated values are computed.

Observable data structure: Interactive visualization often requires the modification of graph topology (graph structure), decomposition (subgraph or aggregation), and attributes (properties). To prevent static links between the Tulip data structure and the external algorithm or system, we provide an observer mechanism that listens for all modifications and applies them to the data structure.

State management: The most substantial improvement in the new meta model is to add to the facade the ability to save the current state of the data structure. Like the OpenGl matrix stack, we provide two functions: **push** and **pop**. These two functions can save or restore the current state of the data structure through a stack. A naive implementation of this feature would be sub-optimal when dealing with a large number of graph elements and their properties. In Tulip, this mechanism has been designed with the proxy design pattern. This pattern allows objects to behave like other objects, hiding direct manipulation of the data structure from the user and allowing data sharing to be globally optimized. Using that stack of state, we were able to implement ef-

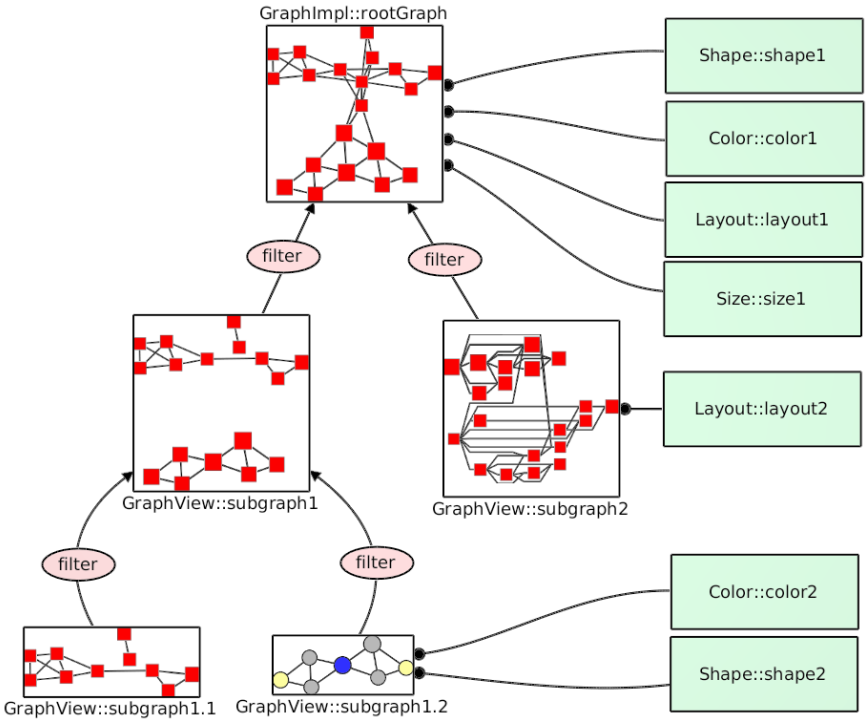


Figure 3: Graph Hierarchy: Tulip provides management of a hierarchy of subgraphs through an efficient filtering mechanism of graphs. For example, a graph with 1,000,000 nodes and 5,000,000 edges and 200,000 subgraphs requires 825 Mb on a 64-bit architecture. Furthermore, through an inheritance mechanism of properties of graph through that hierarchy, it maximizes the number of properties shared between subgraphs. For instance, the subgraph1.2 inherits the layout of the root graph. The inheritance mechanism is also able to redefine properties in subgraphs like one would do in an object oriented programming language. The subgraph2, for example, has redefined its layout but inherits the colors/sizes and shapes of its parent.

ficient implementation of the **command design pattern** and thus, we provide efficient undo/redo operations on large data sets. For instance, a graph with 40000 nodes and 80000 edges under the following modifications: "change all the size", "change the layout", "change all the colors", requires less than 115Mb (including Tulip-GUI, 3D rendering engine, and plug-ins memory usage), enabling immediate undo/redo on a 64bit Intel Q9300 processor.

3.1.2 Algorithms

Several kinds of algorithms are used in information visualization systems but can be clearly separated from the technique. In Tulip, based on our plug-in mechanism, we provide a way to add such new feature. To be independent from visualization techniques, these plug-ins are only authorized to modify the meta-model described above. Furthermore, we do provide a call back mechanism inside our algorithm allowing for interactive use in visualization techniques or

information visualization systems.

For all the algorithms, we do not limit the input parameters, and thus, by using our dynamic parameters declaration mechanism, a programmer can write a large variety of algorithms. However, in order to categorize major classes of algorithms and ease automatic connection with the user interface, we provide interfaces for algorithms that modify a single Tulip property. For instance, standard **graph drawing algorithms** only need to modify the positions of nodes and the positions and number of bends in an edge which can be store in a Layout Property. Based on this idea, we provide plug-ins for hierarchical graph drawing, radial trees, force-directed approaches, spectral methods, planar graph drawing, space filling curves, edge bundling, and bin packing. The **measure algorithm** is based on this same idea and produces real values on entities/relations. It provides, algorithms, such as the computation of k-cores, eccentricity, betweenness centrality, page rank, (bi/tri)connected component, strength metric or Strahler number. Furthermore, we also provide a general algorithm type that can modify any element of the data structure if necessary. We use it for **clustering algorithms**, and it enable us to provide implementations for many approaches including: agglomerative clustering methods, divisive clustering methods and metric-based approaches. We also provide a adapter (ie. wrapper) to directly use the algorithms provided in the open graph drawing framework OGDF library.

3.1.3 Data Import and Export

The efficient import and export of a variety of data formats are key for building a generic information visualization libraries. However, supporting these formats in a generalizable way is not obvious. A basic version of Tulip is able to import CSV (comma separated value) files, GML, and dot formats for graphs and their attributes. We also invented our own format (tlp), that allows meta-information to be saved to disk and for custom configuration of graph appearance. Import algorithms are also available for randomly generating graphs, importing web graphs, or importing a file system.

An important feature of the import/export architecture present in Tulip is that it also forms part of the plug-in architecture. Therefore, programmers can extend the import and export capabilities of Tulip by designing their own plug-ins for custom file formats.

3.2 Tulip Graphics

Efficient rendering of large amounts of geometric information is a bottleneck in most information visualization systems. In the Tulip Graphics library, we provide an OpenGl-based, multi-layer rendering engine that includes the necessary functions for implementing information visualization techniques.

In our multi-layer rendering engine, three dimensional information can be displayed on different layers. For instance, using layers and transparency enables the graphics library to render: textured quads behind the scene, transparent convex hulls on top of graph elements, or displaying legends (2D rendering on top of the scene) for visualizations. Through the OpenGL stencil buffer, we are able to force the visibility of elements on layers. This functionality implements guaranteed visibility [38] for rendered elements. For example, in

our visualization techniques, we use this capability to guarantee that selected elements are always visible.

To ease the implementations of new techniques, we provide functions to: manipulate the camera, select elements, render aggregated elements, render basic geometric entities, and facilitate the use of vertex/pixel/geometric shaders. Special attention has been paid to render these operations usable on huge data sets. For instance, computing and rendering curves, such as Bézier, Splines, and B-Splines, is done on the GPU, allowing Tulip to render more than 10,000 with more than 100 bends in real time without storing any precomputed geometry. In this example, we save the storage and transfer of 2,000,000 triangles required to render this set of curves.

When implementing a new visual metaphor (see section 3.3.1), this graphics library. Using standard C++ inheritance, the programmer can extend this library. However, to be able to extend existing visual metaphors without modification, we provide a plug-in mechanism to add new visual objects. These geometric plug-ins can be used to create **Glyphs**. For example, a programmer can create new plug-ins for rendering pie charts according to specific attribute values. After installing the plug-in, all views (node link diagram, scatter plots etc...) can render graph elements using the new representation.

Using an external rendering engine could have been possible. Two main reasons required that we design our own rendering engine. First, external rendering engines can generate memory overhead unable to handle graph of over 500,000 elements in less than 256Mb of memory. Secondly, when the Tulip project began in 2000, 3D rendering engines was not readily available. However, designing an OpenGl **citation** rendering engine for the purpose of abstract data visualization allows us to optimize and tune the rendering engine according to the visualization techniques we have implemented. As an example, in earlier versions of Tulip, the skeletons of graphs were computed using the Strahler numbers to incrementally render graph nodes and edges [11].

In software engineering terminology, a composite design pattern is used to model the hierarchy of visual objects to be rendered. A naive implementation of this pattern requires the instantiation of a large number of objects, and therefore does not scale to large data sets because of memory constraints. To solve this problem, the Tulip Graphics library accesses this composite using a visitor pattern. First, the visitor pattern adds new functionality to the composite without any modification to its data. For instance, the visitor can compute bounding boxes needed for level-of-detail used during rendering. Secondly, the visitor pattern can simulate a hierarchy of objects without building it. For example, when using a GraphComposite, the visitor traverses a dynamically created hierarchy of objects instead of creating this hierarchy beforehand. Objects are generated and reused on the fly in a way that is similar to the flyweight design pattern during rendering. This pattern avoids data duplication in the data model and graphics library, allowing the system to scale to larger data sets and synchronize rendering with the model.

The philosophy behind the Tulip graphics library is the efficient, direct rendering of data stored in the Tulip data structure without duplication. However, as the amount of available memory has increased significantly, we have integrated into the last version of Tulip optimizations that are more memory intensive. For example, we use octrees to optimize selecting elements or computing level-of-detail, and we use texture based rendering to accelerate the rendering

of aggregated elements during zoom and pan.

Comparing the performance of Tulip to that of VTK/Titan in terms of speed and memory efficiency, we found that loading and rendering a grid of 1,000,000 nodes 2,000,000 edges from scratch takes 20s and 320MB in Tulip and 50s and 1.3GB with VTK/Titan. After this initial rendering, VTK/Titan is 5 times faster than Tulip for subsequent renderings under simple zoom and pan navigation, without modification of graph structure or selection of elements. However, if the selection is modified, selection of elements on this grid is immediate with Tulip and it takes more than 60 seconds with VTK/Titan. These results illustrate the trade-offs Tulip has made between rendering performance and memory usage for the implementation of information visualization techniques.

3.3 Tulip GUI

According to Munzner [37], deciding on the proper visual encoding to use should be decided after problems from real-world users have been characterized. Now, it's not that each problem each time calls for a unique and completely new visualization techniques. The problem often turns into selecting the proper techniques to assemble and implement together with the proper operations. Some techniques now have been used and studied long enough so that their usability perimeter has more or less been established. Because Tulip aims at to be used for implementing end user visualization system, it has to implement a wide palette of existing techniques. Thus, a choice has been made to implement pairs of visual encoding and operations based on their usefulness and scope as assessed by the InfoVis community.

Tulip progressively started adding new features that allowed users to go back and forth between a node-link diagram where metrics were mapped as color or size and histograms that helped understand how a metric was able to capture a key property in the data. These data analysis features have grown and now include a set of well established data visualization techniques (see section 3.3.1). Tulip has evolved from essentially offering a unique visual encoding (node-link diagrams) to a variety of data analysis techniques that can moreover be astutely combined and synchronized. All these new features were carefully and coherently integrated into the framework using agile development methodology (see section 2.4).

We obtained an architecture based on the **Model-View-Controller (MVC)** architectural pattern. The model view controller approach is a well known approach for designing interactive systems. The pattern splits the software architecture in three independent components. The **model** component has the responsibility to store the information, the **view** component gives a representation for the information, and the **controller** manages communication between one or more views and the model. This architecture disassociates the data structure (Model) from the representation (View) and the system behavior (Controller). In the following we describe three main components of the Tulip GUI library.

3.3.1 Views

Views can be defined as visual representations of data. Node link diagrams, parallel coordinates, and scatter plots are are just a few examples of views that

can be used to gain insight into a data set. Tulip uses the above described meta-model to create multiple views of the same data set. The idea is to use the same data independent of the current view. For example, nodes in the node-link view of a graph may have several attributes, and these attributes could be placed in a 2D scatter plot. Having all views share the same data model helps maintain system coherency and enables working with several views simultaneously. Structuring data manipulation in this way allows the information in one view to be easily analyzed in all other views, hopefully providing more insight. Figure 4, shows three different views, in each view one can see that shapes, colors and relative size are preserved. This makes a fundamental, although simple, user interaction quite powerful. As an example, when selecting nodes in a histogram, to focus on high value nodes for instance, the user instantly see where these nodes spread in the node-link view.

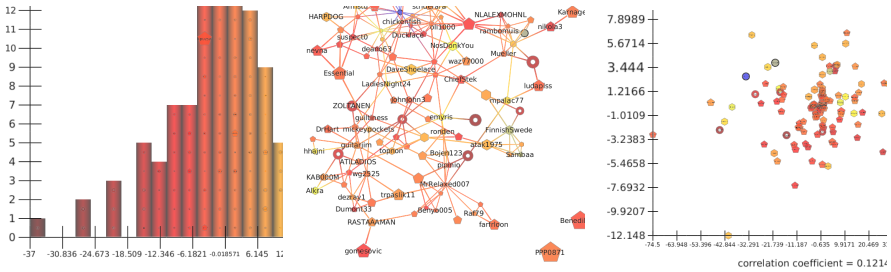


Figure 4: A centralized meta model maintains coherence between views. (**Left**) Histogram view. (**Middle**) node link diagram. (**Right**) Scatter plot views. All three views share the same visual attributes enabling the user to switch between views easily.

For optimization purposes or in order to implement specific types of views, the programmer occasionally needs a custom data structure. For these cases, views can observe any change to the meta-model (see section 3.1.1 for details), synchronizing all views to it. As an example, consider the scatter plot matrix view (see figure 5) implemented in Tulip. This view generates a buffer of textures for efficient navigation through the matrix. The data model, in this case, is used to generate the scatter plot representation for each pair of dimensions and the view stores these results as images. During interactive navigation, the rendering engine displays only the textured quads. If data set is modified by other views or interactors, the set of textures needs to be rendered again. The observer mechanism of Tulip notifies the appropriate views and modifies the data only when necessary.

Views are implemented as Tulip plug-ins. Currently, all views are implemented using the Tulip rendering engine, but programmers are not limited to this engine. Integrating rendering engines such as VTK, other engines, or even multiple engines simultaneously inside a single view can be supported. However, the programmer would need to synchronize all views manually. An example of a foreign rendering engine used in conjunction with the Tulip rendering engine inside a single view is the Google Map mash-up, where Google Map API renders a map in one layer while the Tulip rendering engine renders the remaining layers on top of this map. Figures 5, 6, 7, and 8 present an overview of the major views implemented in the current Tulip release.

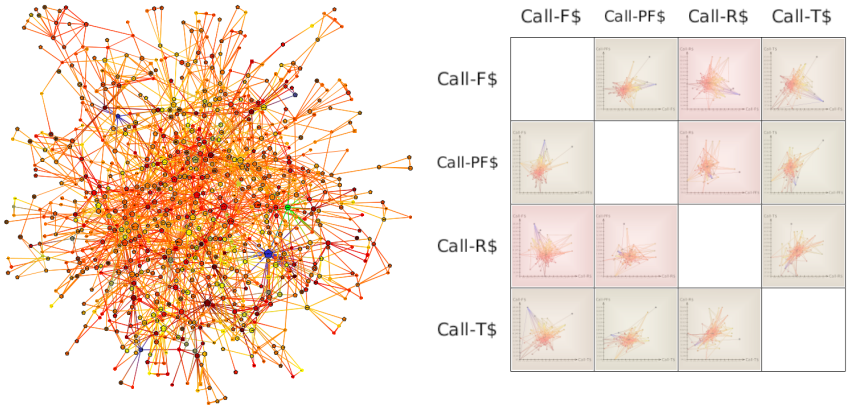


Figure 5: **(Left)** The node-link diagram view renders glyphs for nodes and curves for edges. The view provides navigation such as zoom and pan, bring and go [36], fish eyes views, and a magnifying glass. Direct editing of the graph elements and data, such as adding or removing nodes and edges or translating rotating or scaling elements, are also supported. Other operations on this view include graph splatting, meta-node/graph hierarchy exploration, and texture-based animation. **(Right)** The Scatter plot 2D view renders attribute values to depict possible correlations between properties and the matrix allows efficient navigation between dimensions. The view provides similar interaction to the node link view and implements an interactor to search for correlation in an interactively defined subsets of elements. Splatting is also available in this view.

3.3.2 Interactors

Interaction is essential for most information visualization techniques. However, generalizing interaction in an extendable way raises a significant challenge as a wide range of methods require support. Some selections require transparent rectangles to be drawn on top of selected elements. Opening a metanode requires a single click, a small amount of zooming and panning, and modifying graph structure locally at the metanode. The bring-and-go technique [36] changes the layout of the graph and requires both zoom and pan of the camera along a well defined trajectory. Furthermore, programmers should be able to combine all these interactive techniques in the final visualization. As an example configuration, the mouse wheel could handle both zoom and pan, a left click could modify element selection, and a right click could display a context menu.

To support a range of interaction methods, we implemented the chain of responsibility design pattern. This pattern models the transmission of a message through a chain of linked objects. During the transmission, the message can stop or continue along its path according to the object it passes through. In Tulip, we call an **Interactor** an entire chain and an **InteractorComponent** an object in the chain.

An **InteractorComponent** implements an interaction method and can: handle all GUI events on a view, modify the Tulip data structure, modify the view, and to render objects on top of the view. In the model view controller

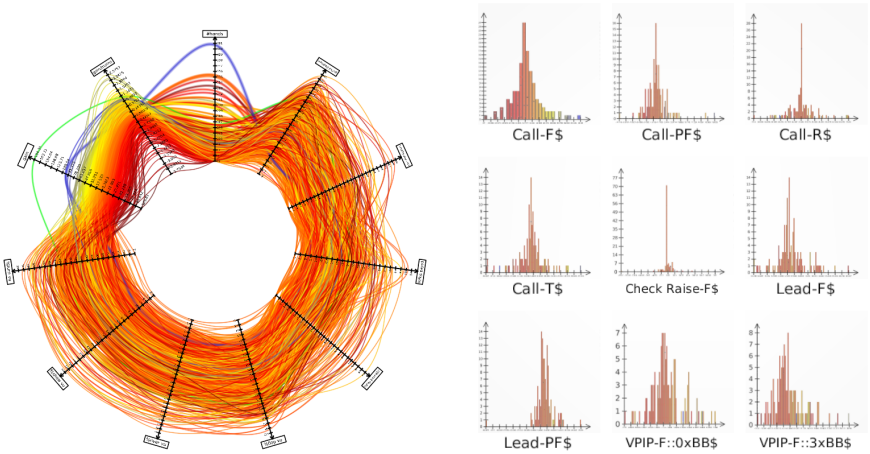


Figure 6: **(Left)** The Parallel Coordinates view depicts multivariate data, using the the traditional parallel coordinates representation as well as a circular representation. In both views, lines can be rendered with smooth Bézier curves. Interaction with the view is supported through zoom and pan, axis edition/permutation/shifting, and multi-criteria/statistical selection. **(Right)** The Histogram view provides a view of element frequency. A matrix of histograms allows for the visual comparison of several statistical properties of a set of dimensions. This view has a standard set of navigation and statistical interactors. Additionally, an interactor enables the user to build non-linear mapping functions to any of the graph attributes such as size, colors, glyphs, etc..

paradigm, this component can be seen as a micro controller. To encourage reuse, InteractorComponent are programmed to be as small as possible. For instance, the zoom & pan, fish-eye lens, magnifying glass, zoom box, and box selection interactors are often reused and implemented in five individual interactors.

An **Interactor** is an ordered set of InteractorComponents. The interactor receives all events from the view and implements the chain of responsibility which asks each interactor component whether or not it can handle an event. The Qt-Nokia library is used as much as possible achieve these operations. The interactor is also responsible for providing configuration widgets, documentation, and an icon for display in toolbars. Furthermore, interactors report the views with which they are compatible. In order to reuse the interactor without modification of source code, the set of views that an interactor supports can be dynamically extended.

Interactors also implement the plug-in interface. Thus, programmers can create their own interactors by combining interactor components or developing new ones. As a result, interactors can be reused across views and the programmer can extend the different types of interactions supported by Tulip. For example, GPU-based graph splatting can be implemented as an interactor.

3.3.3 Perspectives

As each application requires considerable programming effort which we hope to reuse, Tulip recently added domain-specific or user-centered perspectives.

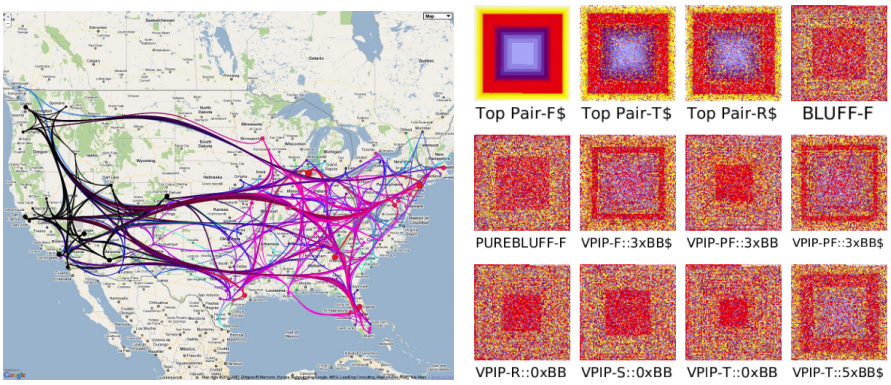


Figure 7: (**Left**) The Google Map view implements a mash-up of the Google map API. With this API, geospatial positions for the layout of graph elements can be specified. When working with data in geography, graphs can be displayed on top of the map. This view supports standard zoom and pan as well as the selection of elements. (**Right**) The Pixel Oriented view uses space filling curves to display large number of entities and relations on a screen. This view supports Hilbert curves, Z-order curves, and spiral curves. The Pixel Oriented view is based on our previous data cube [18] visualization and supports zoom and pan/selection interaction as well as focus+context techniques.

Following Munzner [37], real-world problems should be first characterized and abstracted into good operations and data types. There are good reasons to believe that Tulip contains several of the basic ingredients needed to properly combine and/or develop these operations and data types using Tulip's plug-in based architecture.

After applying the Tulip framework in a variety of domains, including biology, social network analysis, and geography, we realized that many aspects of a visualization system can not be generalized and must be left to the developer to specify. However, in order to reduce re-implementation, we tried to contain all domain-specific elements inside **perspective**[7] plug-ins, allowing general system components and interaction to be re-used across applications.

A Tulip perspective specifies the visualization techniques (algorithms, views, and interactors) to assemble and how to load them. These plug-ins can use domain-specific widgets, menus, and libraries. Perspectives are very different from the generic perspective that come with the open source release 4.1. They are designed through user interviews and problem characterizations and are customized using Tulip libraries and plug-ins.

As our meta-model is generic, we hypothesized that one could keep the same data representation and switch between user interfaces depending on task. The development of the Tulip perspective was inspired by this requirement. In the MVC model, controllers are responsible for managing connections between models and views. Thus, by changing the controller, also known as a mediator pattern in the design pattern terminology, one can change system behavior.

We have had some experience using Tulip in such a context. In order to properly assess the effectiveness of Tulip in visual analytics solutions, more work is

[7] We borrow this terminology from Eclipse project

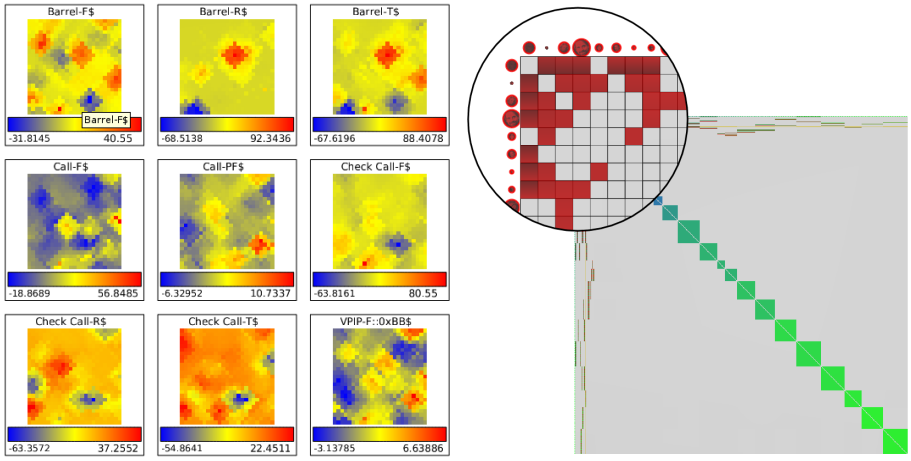


Figure 8: (**Left**) The Self Organizing View implements Kohonen self-organizing maps [33]. Several kinds topology/connectivity for the generated maps are supported as well as navigation and selection interactors. (**Right**) The Matrix view implements a matrix view of the graph. This view has been built to support graphs with a large number of nodes and edges. Zooming and selection interactors are available for this view.

needed. However, what is clear is the gain we experience, as visualization designers and experts. Tulip is a toolbox we use when demonstrating the potential use of visual encodings to define paths to follow with end-users.

3.4 Tulip run time environment

As described above, the philosophy of the Tulip framework is to facilitate the re-use of plug-ins over many contexts. The advantage of this approach is that it allows easier framework extension. However, a disadvantage of this approach is programming an application that exploits a collection of plug-ins is more difficult to implement. This added complexity is, more generally, a disadvantage of plug-in based systems. Tulip Software aims at providing this needed organization to these plug-ins so that they can be more easily used. Section 3.3.3 shows that it is not the Tulip Software that creates a visualization system, but a perspective plug-in launched by the Tulip software.

The design of Tulip Software was inspired by all the stand alone applications that we have implemented with the Tulip libraries [13, 15, 14, 32, 23]. Using agile method, refactorization aims to place all duplicated code inside this software. The primary difficulty of designing Tulip Software is to determine the maximum set common functions between perspectives. In our experience, we found these functions either necessary or general enough to be used by all designed systems:

Model Management: All the perspectives store data inside the Tulip data model. Thus, Tulip Software supports this model. The software provides import, export, open, close, and checks the data structure for modifications. As the model can be analyzed with different perspectives, Tulip Software is also responsible for changing/choosing the perspective used and managing the multi-document interface with tab widgets.

Plug-in Management: Since perspectives are plug-ins, they cannot be used until they are loaded. Thus, the software initializes all the libraries and plug-ins. The software automatically checks for plug-ins dependencies and can update or download plug-ins using the Tulip plug-in web service. When creating a desktop application, as opposed to a web application, this functionality is necessary to involve the end user in the development. Frequent installation of new releases is one of the most important problems for end users.

Cross platform support: Supporting multiple platforms is very time consuming when designing new applications. In Tulip, we aim to provide a platform independent execution environment. The programmer can thus focus on the implementation of their. Tulip is available for Linux, Windows, and Mac OS operating systems. Through the plug-in web service, access to plug-ins compiled for all three platforms.

4 Case Studies

The Tulip framework consists of a set of libraries and an application for managing plug-ins using these libraries. In a way, without plug-ins Tulip is not able to visualize data. However, it provides the necessary functions and data structures to build a system tailored to the task of the user. In this section, we describe some visualization systems we have built using Tulip.

4.1 Generic Tulip Perspective

The Tool Box system (known as the Tulip Graph Visualization Software) provides a generic software interface for the purpose of information visualization research. We identified the following tasks as the most important for our research.

Reproducibility: The most important task is the *reproducibility* of results published in our community. Such a system should be able to integrate many different types of techniques and algorithms. In early versions of Tulip, the focus was on graph visualization and therefore, our requirements consisted of graph metrics, graph drawing algorithms, and graph clustering algorithms. In later version, we furthered this idea to include visualization techniques and user interaction approaches.

Rapid Prototyping: we would like to quickly prototype new algorithms or visualization techniques and analyze them in a general visualization context. For example, we could see how a new clustering algorithm or graph drawing algorithm eases understanding of a data set.

Pipeline Exploration: We would like to interactively combine existing algorithms, techniques, and interaction methods easily to construct domain-specific visualizations. This feature is helpful for interviews with end users as a combination of existing features can often be used as a starting point for user feedback, delaying implementation of custom visualization methods to later stages in the project. For instance, when working with biologist, we prototyped the analysis pipeline using the generic perspective (see section 4.2) before further implementation.

We have implemented this generic Tulip perspective that supports: editing graph element properties, exploration of the subgraph hierarchy, and access

to built-in functions of the Tulip Core libraries. Sample operations that are available include: undo/redo, aggregation, subgraph creation, planarity testing, and cut/paste. Moreover, this system automatically constructs menu items and tool bars, allowing access to all installed algorithms, view, and interactors.

The connection statistics for our plug-in web service, a service that checks for updates to perspectives when they are launched, indicates that the perspective is frequently used for direct data analysis. Every day more than 100 people use this perspective and and the number of hits to its web site [8] is about 8000 in March of 2010.

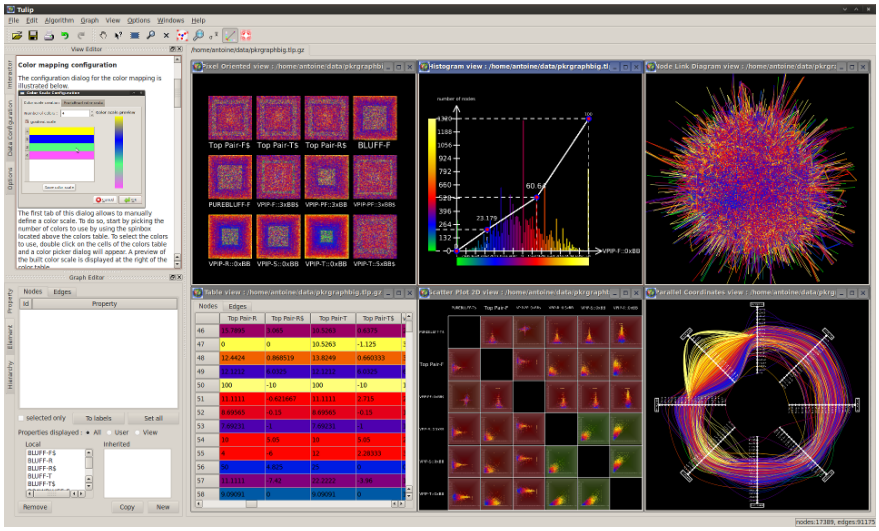


Figure 9: The Tulip generic perspective provides an automatically generated user interface depending on available plug-ins. It also provides tools for manual configuration of both views and interactors.

4.2 Systryp Perspective

The Systryp perspective was constructed in order to help biologists understand the metabolism of the tsetse fly parasite that causes sleeping sickness. During initial user interviews, we found they seem to follow an analytic process which involves getting an overview of the data first and then focusing on a few relevant sub-networks. Using the generic perspective 4.1, we tested various interaction methods via manual selection of elements and the sub-graph hierarchy. In a sense, this stage experimented with many different visualization pipelines for exploratory analysis with little implementation.

After this initial stage, we implemented biology-specific algorithms to extract these subgraphs using a Tulip clustering plug-in. A custom import plug-in allowed Tulip to directly load their dedicated data format. By using this generic perspective, we were able to run a second round of interviews to determine if we were on the right track. Figure 10 shows two pipelines identified to be useful for their tasks.

[8] www.tulip-software.org

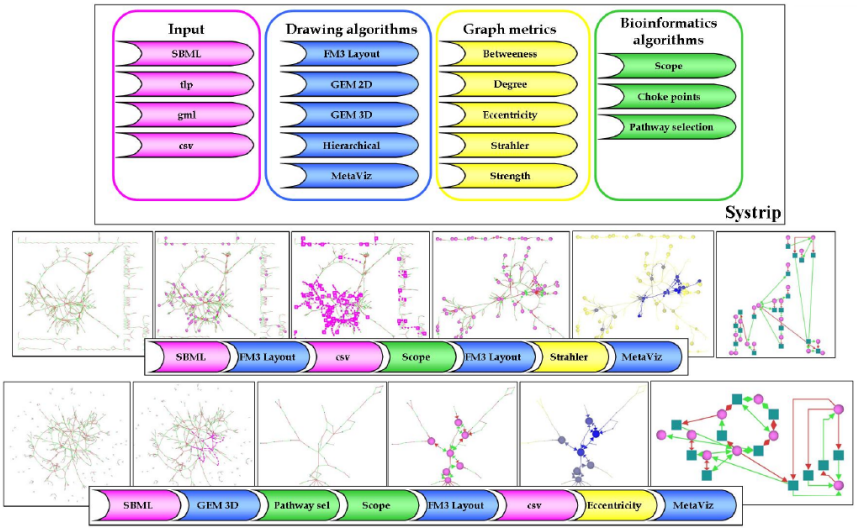


Figure 10: The Systrip pipeline was created to help biologists understand the metabolism of the tsetse fly parasite that causes sleeping sickness. First, through the generic perspective of Tulip and then through a custom Systrip perspective, we began to understand the task requirements, providing a visualization pipeline customized to the task of the biologist.

After the preliminary prototypes, we implemented a custom perspective (see figure **??**) that integrates these two pipelines. This perspective limits access to only the Tulip functionality that relevant for their task and and uses domain-specific terminology in the user interface. As an example, graph terminology is ineffective with this audience and the terms network and sub network need to be used.

With this prototype, the user community experimented with the perspective without our assistance, allowing them to suggest improvements. For instance, data is generated over time during experiments and the user users required animation capabilities showing the changes induced by biological events. We were able to manually simulate this behaviour using the generic perspective for feedback. Subsequently, the functionality was implemented as an interactor for the node-link diagram view. The final perspective integrates other domain-specific capabilities such as connections appropriate databases and the three-dimensional rendering of molecules.

4.3 Grouse, GrouseFlocks, and TugGraph Perspective

Sometimes, when the number of nodes and edges in a graph becomes large, rendering all of them directly can be an obstacle to graph readability. Also, computing a full drawing of the graph can be expensive in terms of running time. As a new approach for dealing with these problems, members of our team applied Tulip to research new techniques for graph visualization. The perspective for this system was originally an application that used the Tulip and QT libraries. Subsequently, the application was converted into Tulip perspective

when support became available in version 3.

In this approach, the contents of metanodes, either derived from topological structures or attribute information, were constructed and/or drawn on demand as the user explores the data. Grouse [5] took a large graph and hierarchy as input and was able to draw parts of it on demand as users opened metanodes. Appropriate graph drawing algorithms were used to draw the subgraphs based on their topological structure. For example, if the node contains a tree, a tree drawing algorithm will be used. GrouseFlocks [7] was created to construct graph hierarchies based on attribute data and progressively draw them. Search strings selected or categorized nodes and computed induced subgraphs based on attribute values that were placed inside connected metanodes. These metanodes could be drawn on demand with Grouse. However, often parts of a graph are near certain nodes and metanodes are of interest and certain metanodes can be too large to draw on demand. TugGraph [8] was created for these situations when topology near a node or metanode is interesting. Also, it can summarize specific sets of paths in the graph.

Efficient implementation of these three software techniques benefited greatly from Tulip's metanode/metagraph structure, its ability to handle the large numbers of subgraphs generated by the systems, and its animation functions to animate graph elements on the screen. One of the biggest advantages of using Tulip for developing this software was the number of graph drawing algorithms it supports. In fact, all three systems were made to be configurable so that new graph drawing algorithms could inserted into the system at compile time at Tulip made plug-ins for new graph drawing algorithms became available. Images of TugGraph and GrouseFlocks are shown in Figure 11.

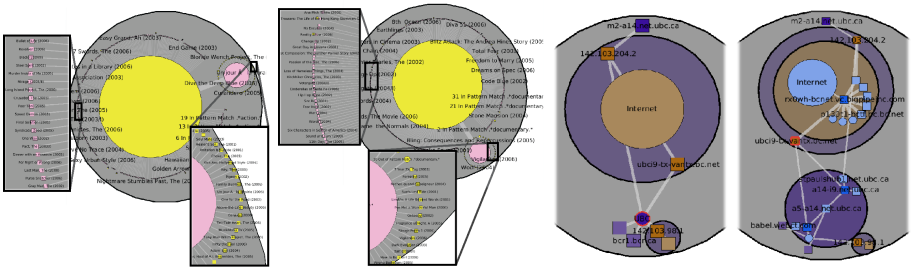


Figure 11: Images produced by GrouseFlocks and TugGraph. **(a)** Two perspectives of a movie graph, nodes are movies and edges link movies that share an actor, indicating genre lock. In both the action and documentary genres we get a large metanode of non-genre (yellow) movies and a large metanode of in genre movies (pink). **(b)** TugGraph explores the structure of the Internet around UBC. In this case, *ubci9* is tugged on the left image revealing its direction connections in saturated blue on the right. In many cases, these direct adjacencies fragment the graph into multiple connected components shown in light blue.

5 Conclusion and Future Work

We have presented the Tulip 3 framework which is based on ten years of of our research. We have explained the architecture choices we have made to

create a stable and maintainable platform for information visualization research. The framework allows us to test all levels of the Munzner nested model: from the algorithm, to technique/interaction, to encoding, and finally to validating a complete system taking end users into account. Through technical details and a few experiments, we have demonstrated that our framework can scale to large data sets. Furthermore, we provide this framework to the information visualization community for reproducibility of our research under the LGPL license. Tulip is available under Windows, Linux, and Mac OS.

A future challenge for Tulip will include integrating our initial experiences working with dynamic graphs [3] into this model and optimizing data storage for dynamic data. Furthermore, integrating this concept directly into our facade will provide a unified set of visualization techniques using relational data as a basis.

References

[1] J. Abello, F. van Ham, and N. Krishnan. ASK-graphview: A large scale graph visualization system. *IEEE Trans. on Visualization and Computer Graphics*, 12(5):669–676, 2006.

[2] E. Adar. GUESS: A language and interface for graph exploration. In *In CHI '06: Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 791–800, 2006. `http://graphexploration.cond.org/`.

[3] D. Archambault. Structural differences between two graphs through hierarchies. In *Proc. of Graphics Interface*, pages 87–94, 2009.

[4] D. Archambault, T. Munzner, and D. Auber. Smashing peacocks further: Drawing quasi-trees from biconnected components. *IEEE Trans. on Visualization and Computer Graphics (Proc. Vis/InfoVis 2006)*, 12(5):813–820, Sept.-Oct. 2006.

[5] D. Archambault, T. Munzner, and D. Auber. Grouse: Feature-based, steerable graph hierarchy exploration. In *Proc. of Eurographics/IEEE VGTC Symp. on Visualization (EuroVis '07)*, pages 67–74, 2007.

[6] D. Archambault, T. Munzner, and D. Auber. TopoLayout: Multilevel graph layout by topological features. *IEEE Trans. on Visualization and Computer Graphics*, 13(2):305–317, March/April 2007.

[7] D. Archambault, T. Munzner, and D. Auber. GrouseFlocks: Steerable exploration of graph hierarchy space. *IEEE Trans. on Visualization and Computer Graphics*, 14(4):900–913, 2008.

[8] D. Archambault, T. Munzner, and D. Auber. TugGraph: Path-preserving hierarchies for browsing proximity and paths in graphs. In *Proc. of the 2nd IEEE Pacific Visualization Symposium*, pages 113–121, 2009.

[9] D. Auber. Tulip. In P. Mutzel, M. Jnger, and S. Leipert, editors, *9th International Symposium on Graph Drawing, GD 2001*, volume 2265 of *Lecture Notes in Computer Science*, pages 335–337, Vienna, Austria, 2001. Springer-Verlag.

[10] D. Auber. *Outils de visualisation de larges structures de donnes.* Phd, University Bordeaux I, 2002.

[11] D. Auber. Using Strahler numbers for real time visual exploration of huge graphs. In *Using Strahler numbers for real time visual exploration of huge graphs International Conference on Computer Vision and Graphics*, volume 1-3, pages 56–69, 2002.

[12] D. Auber. Tulip : A huge graph visualization framework. In P. Mutzel and M. Jünger, editors, *Graph Drawing Software*, Mathematics and Visualization, pages 105–126. Springer-Verlag, 2003.

[13] D. Auber, Y. Chiricota, F. Jourdan, and G. Melanon. Multiscale navigation of small world networks. In *IEEE Symposium on Information Visualisation*, pages 75–81, Seattle, GA, USA, 2003. IEEE Computer Science Press.

[14] D. Auber, M. Delest, J.-P. Domenger, and S. Dulucq. Efficient drawing of rna secondary structure. *Journal of Graph Algorithms and Applications*, 10(2):329–351, 2006.

[15] D. Auber, M. Delest, J.-P. Domenger, P. Ferraro, and R. Strandh. EVAT: Environment for Visualization and Analysis of Trees. In *EVAT: Environment for Visualization and Analysis of Trees Proceedings of the IEEE Symposium on Information Visualization*, pages 124–125, 10 2003.

[16] D. Auber and F. Jourdan. Interactive refinement of multi-scale network clusterings. In *IV '05: Proceedings of the Ninth International Conference on Information Visualisation*, pages 703–709, Washington, DC, USA, 2005. IEEE Computer Society.

[17] D. Auber and P. Mary. Mise en place dun mécanisme de plugins en c++. *Programmation sous Linux*, 1(5):74–79, 2006.

[18] D. Auber, N. Novelli, and G. Melançon. Visually mining the datacube using a pixel-oriented technique. In *IV*, pages 3–10, 2007.

[19] V. Batagelj and A. Mrvar. Pajek - analysis and visualization of large networks. In *Graph Drawing Software*, volume 2265, pages 77–103, 2003.

[20] K. Beck and C. Andres. *Extreme Programming Explained : Embrace Change (2nd Edition).* Addison-Wesley Professional, November 2004.

[21] R. Boulet, B. Jouve, F. Rossi, and N. Villa. Batch kernel som and related laplacian methods for social network analysis. *Neurocomputing Special Issue on Progress in Modeling, Theory, and Application of Computational Intelligenc - 15th European Symposium on Artificial Neural Networks 2007*, 71(7-9):12571273, 2008. See also http://www.nature.com/news/2008/080519/full/news.2008.839.html.

[22] R. Bourqui and D. Auber. Large quasi-tree drawing: A neighborhood based approach. In *IV '09: Proceedings of the 13 International Conference on Information Visualisation (IV'09)*, pages –, Washington, DC, USA, 2009. IEEE Computer Society.

[23] R. Bourqui, V. Lacroix, L. Cottret, D. Auber, P. Mary, M.-F. Sagot, and F. Jourdan. Metabolic network visualization eliminating node redundance and preserving metabolic pathways. *BMC Systems Biology*, 1(29), 2007.

[24] M. Chimani, C. Gutwenger, M. Jünger, K. Klein, P. Mutzel, and M. Schulz. The open graph drawing framework. In *Posters of the 15th International Symp. on Graph Drawing (GD'07)*, 2007. `http://www.ogdf.net/ogdf.php/ogdf:publications` (visited 18/03/2010).

[25] Y. Chiricota, F. Jourdan, and G. Melanon. Software components capture using graph clustering. In *11th IEEE International Workshop on Program Comprehension*, pages 217–226, Portland, Oregon, 2003. IEEE / ACM.

[26] J. Ellson, E. R. Gansner, E. Koutsofios, S. North, and G. Woodhull. Graphviz - open source graph drawing tools. In *The 9th International Symp. on Graph Drawing (GD'01)*, volume 2265 of *LNCS*, pages 483–484, 2002.

[27] N. Elmqvist and J.-D. Fekete. Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines. *IEEE Transactions on Visualization and Computer Graphics*, 16:439–454, 2010.

[28] J.-D. Fekete. The infovis toolkit. In *The 10th IEEE Symp. on Information Visualization (InfoVis '04.)*, pages 167–174, 2004. `http://ivtk.sourceforge.net/`.

[29] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1 edition, July 1999.

[30] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, illustrated edition edition, November 1994.

[31] J. Heer, S. K. Card, and J. A. Landay. 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, 2005. `http://prefuse.org/`.

[32] F. Iragne, M. Nikolski, B. Mathieu, D. Auber, and D. J. Sherman. Proviz: protein interaction visualization and exploration. *Bioinformatics*, 21(2):272–274, 2005.

[33] T. Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43:59–69, 1982.

[34] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 1st edition, October 2002.

[35] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *Comm. of the ACM*, 38(1):96–102, 1995.

[36] T. Moscovich, F. Chevalier, N. Henry, E. Pietriga, and J. D. Fekete. Topology-aware navigation in large networks. In *SIGCHI Conference on Human Factors in Computing Systems (2009)*, pages 2319–2328, 2009.

[37] T. Munzner. A nested process model for visualization design and validation. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):921–928, 2009.

[38] T. Munzner, F. Guimbretiére, S. Tasiran, L. Zhang, and Y. Zhou. TreeJuxtaposer: Scalable tree comparison using focus+context with guaranteed visibility. *Proc. SIGGRAPH 2003, ACM Transactions on Graphics*, 22(3):453–462, 2003.

[39] P. Mutzel, C. Gutwenger, R. Brockenauer, S. Fialko, G. Klau, M. Krüger, T. Ziegler, S. Näher, D. Alberts, D. Ambras, G. Koch, M. Jünger, C. Buchheim, and S. Leipert. A library of algorithms for graph drawing. In *The 6th International Symp. on Graph Drawing (GD'98)*, volume 1547 of *LNCS*, pages 456–457, 1998.

[40] U. A. Perego. The power of dna: Discovering lost and hidden relationships. how dna analysis techniques are assisting in the great search for our ancestors. In *World Library and Information Congress: 71th IFLA General Conference and Council*, pages 1–19, Oslo, Norway, 2005.

[41] M. Raitner. Hgv: A library for hierarchies, graphs, and views. In *10th International Symposium on Graph Drawing, GD 2002*, pages 236–243, 2002.

[42] C. Rozenblat, G. Melanon, M. Amiel, D. Auber, C. Discazeaux, A. LHostis, P. Langlois, and S. Larribe. Worldwide multi-level networks of cities emerging from air traffic (2000). In *International Geographical Union IGU 2006 Cities of Tomorrow*, Santiago de Compostela, Spain, 2006.

[43] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit An Object-Oriented Approach To 3D Graphics*. Kitware, Inc., 4 edition, 2006.

[44] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research*, 13(11):2498–504, 2003.

[45] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL'96: Proc. of the 1996 IEEE Symp. on Visual Languages*, pages 336–344, 1996.

[46] C. Tominskia, J. Abello, and H. Schumann. CGV – an interactive graph visualization system. *Computers & Graphics*, 33(6):660–678, 2009.

[47] B. Wylie and J. Baumes. A unified toolkit for information and scientific visualization. *Visualization and Data Analysis 2009*, 7243(1):72430H, 2009. `http://titan.sandia.gov/` (visited 18/03/2010).

Contents

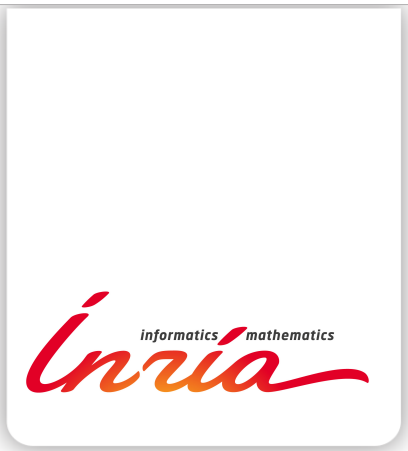
1 Introduction **3**

2 Background and Related Work **4**
2.1 Libraries 5
2.2 Toolkits 5
2.3 Software 6
2.4 Software Engineering Background 6

3 Architecture overview **7**
3.1 Tulip Core 7
3.1.1 Meta-model III 8
3.1.2 Algorithms 11
3.1.3 Data Import and Export 12
3.2 Tulip Graphics 12
3.3 Tulip GUI 14
3.3.1 Views 14
3.3.2 Interactors 16
3.3.3 Perspectives 17
3.4 Tulip run time environment 19

4 Case Studies **20**
4.1 Generic Tulip Perspective 20
4.2 Systryp Perspective 21
4.3 Grouse, GrouseFlocks, and TugGraph Perspective 22

5 Conclusion and Future Work **23**



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

351, Cours de la Libération
Bâtiment A 29
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399