

A Unified Toolkit for Information and Scientific Visualization

Brian Wylie and Jeffrey Baumes

Abstract— We present an expansion of the popular open source Visualization Toolkit (VTK) to support the ingestion, processing, and display of informatics data. The result is a flexible, component-based pipeline framework for the integration and deployment of algorithms in the scientific and informatics fields. This project, code named “Titan”, is one of the first efforts to address the unification of information and scientific visualization in a systematic fashion. The result includes a wide range of informatics-oriented functionality: database access, graph algorithms, graph layouts, views, charts, UI components and more. Further, the data distribution, parallel processing and client/server capabilities of VTK provide an excellent platform for scalable analysis.

Index Terms—Visualization toolkit, scientific visualization, information visualization, pipeline model.

1 INTRODUCTION

There are many existing open source toolkits that provide functionality around the traditional domains of scientific and information visualization. However, the algorithms, techniques, and data structures present in these toolkits are specialized for either the scientific visualization or information visualization domain. The visualization community has for many years discussed the unification of these domains in panels [24] [27] and publications [18]. In this paper we describe Titan, an extension to the open-source toolkit VTK that embodies this unification.

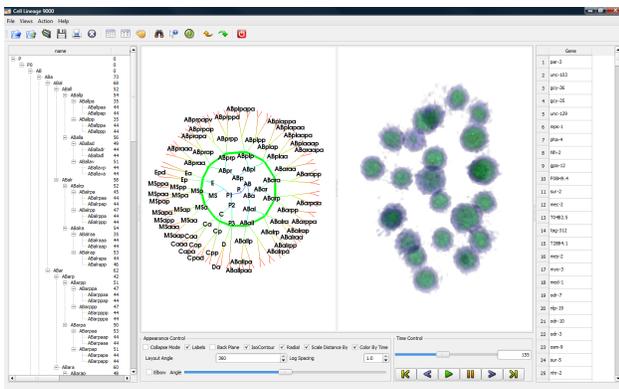


Fig. 1. An application using the VTK/Titan toolkit to combine information and scientific visualization [7]. The tree represents the development of a *C. elegans* embryo from a single cell and is linked to a volume rendering of the embryo at that time. Cells selected in the tree view are also linked to gene expression data.

The expansion of the Visualization Toolkit (VTK) [19] to support informatics in a meaningful and effective way represents a substantial amount of current and ongoing work. Sandia National Laboratories is spearheading this effort under the project name of “Titan”. In close collaboration with Kitware, Inc. All of Titan is open source and will be made available through the public VTK repository. Much of the work is already available in the repository, with additional releases planned quarterly.

- Brian Wylie is with Sandia National Laboratories, E-mail: bnwylie@sandia.gov.
- Jeffrey Baumes is with Kitware, Inc., E-mail: jeff.baumes@kitware.com.

Manuscript received 31 March 2007; accepted 1 August 2007; posted online 27 October 2007.

For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

1.1 Our Goal

The goal of this project is to unify scientific and information visualization under a single flexible, extensible, and scalable architecture. The component pipeline design in VTK, with its executive/algorithm model [21], enables developers to use and combine components in a dynamic way. Extension of the toolkit is straightforward with well defined APIs and excellent documentation; laboratories and academic institutions around the world have already contributed new components to the toolkit. Los Alamos, Lawrence Livermore and Sandia National Laboratories have all contributed to the client/server, distributed memory and parallel rendering features provided by VTK.

As part of the project goals we wanted to use third-party software wherever possible within the Titan project. The framework is well suited to support optional dependencies, allowing us to build upon the strengths of excellent packages such as the following:

- Qt, for interface and application development
- BGL (Boost Graph Library), for graph algorithms
- PBGL (Parallel BGL) for parallel graph algorithms
- MTGL (Multithreaded Graph Library) algorithms for hyper-threaded hardware
- Trilinos: Scalable, distributed memory linear algebra library

1.2 Why VTK?

Building Titan within VTK allows us to benefit from VTK’s wide exposure and community support. Contributions are welcome, encouraged, and supported under the existing VTK contribution model. Moreover, since VTK is part of the curriculum in many visualization courses, we have the opportunity to provide powerful new tools in a framework that is already familiar to a wide audience.

We felt that VTK represented the best technology platform for a unified toolkit. In the scientific visualization domain its usage is common, diverse and widespread. The component-based, demand-driven pipeline provides a flexible integration platform for the coupling of third party software packages. The toolkit supports a distributed memory usage model, and as the basis for the ParaView product it has demonstrated world class scalability [4] [8] [2].

1.3 Main Contributions

The major contributions described in this paper are as follows:

- An open-source toolkit embodying a unified approach to scientific visualization and information visualization
- A mechanism to use databases and database queries as the input to an on-demand, streaming data pipeline
- The set of “Data-less” adapters to leverage the strengths of other libraries and toolkits with minimal overhead

2 RELATED WORK

There is a large, impressive body of existing work in informatics toolkits. While an exhaustive list of existing toolkits, frameworks, and feature sets is beyond the scope of this paper, we highlight below a few of the most popular projects and their key features. We hope to learn from their successes and collaborate with them in the future.

- Prefuse Visualization Toolkit (www.prefuse.org): A Java-based toolkit for building interactive information visualization applications [15]. Prefuse provides an effective interaction and animation infrastructure for building many different types of views.
- Tulip Toolkit (www.labri.fr/perso/auber/projects/tulip): A C++ toolkit for building interactive information visualization applications with both 2D and 3D display capability. Tulip implements some very fast algorithms for graph layout such as HDE [20]. Tulip also has a plugin architecture for building and testing graph and tree layout algorithms.
- GraphViz (www.graphviz.org): A set of libraries and executables, written in C, specifically for the visualization of many different types of graphs [13]. GraphViz mainly has functionality around static, non-interactive diagram generation. It has high quality graph layout algorithms and yields especially good results for directed acyclic graphs.
- InfoVis Toolkit (ivtk.sourceforge.net): A Java-based toolkit to ease the development of Information Visualization applications and components [11]. IVTK employs a unified data structure (a table of columns) for all of its graph, table and tree data in order to reduce memory footprint and reduce the complexity of filtering, selection and interaction algorithms. The InfoVis Toolkit is quite popular and allows the integration of several different types of views.
- InfoVis Cyberinfrastructure (iv.slis.indiana.edu/sw): Like Titan, this project provides an integration framework for other software packages. While the IVC framework itself is written in Java, it aims to allow contributors to integrate algorithms written in many different languages. The InfoVis Cyberinfrastructure uses the Eclipse Rich Client Platform for its application framework.
- Piccolo Toolkit (www.cs.umd.edu/heil/piccolo): Piccolo is a layer built on top of optimized language-dependent graphics APIs. Currently supports Java and C# (Piccolo.Java, Piccolo.NET) [5]. Piccolo provides a built-in zoomable user interface (ZUI) which targets building applications where the user can transition smoothly from overview to fine detail.
- GeoVista Studio (www.geovistastudio.psu.edu/jsp): GeoVISTA Studio [26] is an open software development environment designed for geospatial data. It provides a visual programming environment to allow application development without writing code.
- Improvise (www.personal.psu.edu/faculty/c/e/cew15/improvise): Improvise is a Java-based infovis toolkit especially strong in linking and coordination between different views. It is capable of visualizing its own structure and allows users to specify queries and filters using a declarative, visual language.

Most existing information visualization toolkits use Java for its ease of use, its embeddability within Web pages, and its cross-platform compatibility. While these are major strengths, there are also many reasons to provide information visualization capabilities in other languages such as C++ such as the ability to integrate mature C++ libraries for information processing (e.g. the Boost Graph Library) and GUI building (e.g. Qt). We discuss this further in Section 3.

Heer and Agrawala describe software patterns in information visualization, which the developers of Titan reviewed while building the toolkit [14]. Since the Data Column, Renderer and Camera patterns are already part of VTK, they are naturally a part of Titan. The functions of the Operator pattern are already provided by VTK's data-flow pipeline.

The Boost Graph Library, while not a visualization library *per se*, contains flexible data structures and C++ generic algorithms for graphs [25]. Because of its success, we chose to integrate Titan with BGL for graph algorithms, data structures, and parallel processing.

3 OVERVIEW

The Titan informatics components share the same C++ software toolkit model as VTK. Moreover, by using the CMake build system [22] Titan runs on Macintosh, Windows and various Unix platforms without modification.

The Titan/VTK libraries contain data processing units called *filters*. Each filter implements a single algorithm. A filter specifies its inputs and outputs and may be hooked together with other filters into data processing pipelines. Figure 2 gives an overview of the stages of the toolkit pipelines. At the beginning of a pipeline are data sources such as databases, XML or flat text files. These data sources are fed into the pipeline either through database queries or traditional file readers. The output from data sources is processed by filters that implement algorithms and analysis techniques. The output of a series of filters is connected to a Titan view for visualization. An application is typically centered around one or more views. The user may interact with these views and through the UI may change data sources, configure filter parameters, or change view settings. The pipeline executive in the toolkit detects changes in pipeline components and updates the appropriate parts of the pipeline accordingly.

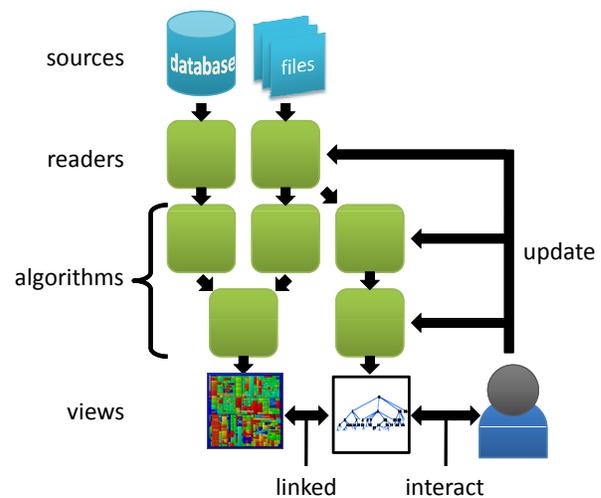


Fig. 2. Overview of the data flow model used by the Titan informatics toolkit. Readers load data from files or databases. The data are transformed by sequences of filters and then displayed in linked views. By interacting with the application and its views, the user can change parameters on filters and renderers and see the results immediately.

The well defined, well documented APIs associated with the pipeline and components make extending the toolkit straightforward. Developers often inspect an existing filter that does something similar to what they want to do, and use that code as a starting-point for their development.

Since Titan is a part of VTK, wrappers for Python, Java and Tcl are automatically generated for every class. This allows the full functionality of Titan to be used in applications written in any of these languages. In Figures 3 and 4 we illustrate a “Hello, World” example using Titan in both C++ and Python. The Java and Tcl wrappings are similar. The code generates a random graph, runs a Boost Graph Library breadth first search algorithm on the graph and displays it in a graph layout view.

These examples can be expanded with just a few more lines of code to provide visualizations like that shown in Figure 6 which demonstrates both the use of dynamic labeling [6] and hierarchical graph bundles [17]. These techniques, developed by other members of the visualization community, were added to Titan in just days. The pipeline architecture allows new techniques to be rapidly deployed within the

```

#include "vtkBoostBreadthFirstSearch.h"
#include "vtkGraphLayoutView.h"
#include "vtkRandomGraphSource.h"
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"

int main(int argc, char* argv[])
{
    // Create a random graph
    vtkRandomGraphSource* source =
        vtkRandomGraphSource::New();

    // Create BGL algorithm and put it in the pipeline
    vtkBoostBreadthFirstSearch* bfs =
        vtkBoostBreadthFirstSearch::New();
    bfs->SetInputConnection(source->GetOutputPort());

    // Create a view and add the BFS output
    vtkGraphLayoutView* view = vtkGraphLayoutView::New();
    view->AddRepresentationFromInputConnection(
        bfs->GetOutputPort());

    // Color vertices based on BFS search
    view->SetVertexColorArrayName("BFS");
    view->ColorVerticesOn();
    view->SetVertexLabelArrayName("BFS");
    view->VertexLabelVisibilityOn();

    vtkRenderWindow* window = vtkRenderWindow::New();
    view->SetupRenderWindow(window);
    window->GetInteractor()->Start();

    source->Delete();
    bfs->Delete();
    view->Delete();
    window->Delete();
    return 0;
}

```

Fig. 3. Simple Titan example in C++.

```

from vtk import *

# Create a random graph
source = vtkRandomGraphSource()

# Create BGL algorithm and put it in the pipeline
bfs = vtkBoostBreadthFirstSearch()
bfs.SetInputConnection(source.GetOutputPort());

# Create a view and add the BFS output
view = vtkGraphLayoutView()
view.AddRepresentationFromInputConnection(
    bfs.GetOutputPort());

# Color vertices based on BFS search
view.SetVertexColorArrayName("BFS");
view.ColorVerticesOn();
view.SetVertexLabelArrayName("BFS");
view.VertexLabelVisibilityOn();

window = vtkRenderWindow();
view.SetupRenderWindow(window);
window.GetInteractor().Start();

```

Fig. 4. Simple Titan example using Python bindings.

toolkit.

4 DATA STRUCTURES

New data structures tailored to informatics algorithms are one of the fundamental changes introduced as part of Titan. Currently, we support two main new data structures, `vtkTable` and `vtkGraph` (along with associated subclasses for directed, undirected, DAGs, and trees). We are testing a third set of structures for supporting N -dimensional ar-

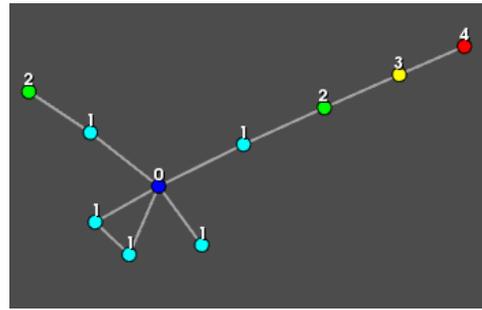


Fig. 5. Example graph colored and labeled by the results of a Boost Graph Library breadth-first search algorithm starting from the vertex labeled 0.

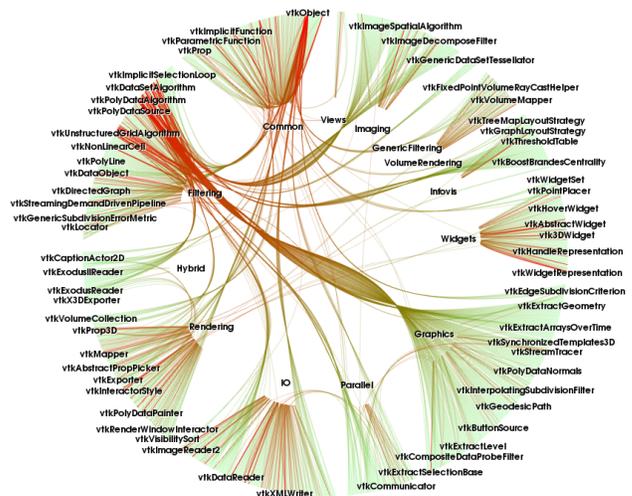


Fig. 6. Titan view demonstrating hierarchical graph bundles and dynamic labeling. Edges link each class in VTK to its superclass, and are grouped by library. Classes are labeled without overlap and priority is given to classes with many subclasses (see `VTK\Infovis\Testing\Cxx\TestGraphHierarchicalBundle.cxx`).

rays (vector/matrix/tensor). Linear algebra and statistics support will be released in a later version of Titan.

4.1 Table

`vtkTable` is the simplest of the Titan data structures. It is simply a collection of columns stored in arrays. Each column is accessed by name, and columns may be added, altered, or removed by algorithms. As part of the support for `vtkTable` the toolkit now includes discriminated-union (`vtkVariant`, `vtkVariantArray`) and string (`vtkStdString` and `vtkStringArray`) types in addition to the numeric types already in VTK.

4.2 Graph Classes

The hierarchy for graph classes is shown in Figure 7. The blue classes are the main data structure classes that flow through a VTK pipeline. At the top level, we distinguish between graphs whose edges have inherent order from source to target (directed graphs) and graphs whose edges do not indicate direction (undirected graphs). The directed graph subclasses are naturally structured by specialization. A directed acyclic graph (i.e. a graph with no paths that lead back to the same place) is a subset of the class of all directed graphs. A `vtkTree` further restricts this by enforcing a hierarchy: every vertex but the root must have exactly one parent (incoming edge). The green classes are the mutable classes used to create or modify graphs. The structure of a graph may be copied into any other graph instance, if it first passes a

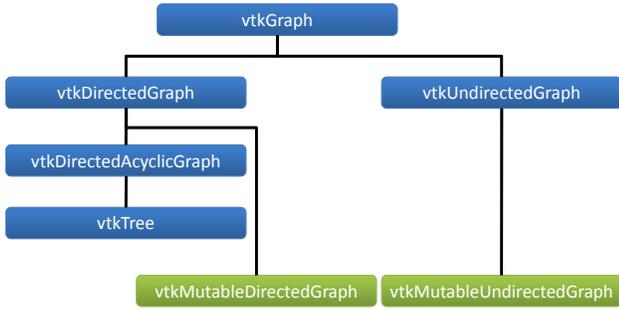


Fig. 7. Graph class hierarchy.

compatibility test.

This hierarchy gives us several advantages. Filters that operate on `vtkGraph` will work for all graph types. So one can send a `vtkTree` or a `vtkDAG` into a filter that takes the more general `vtkGraph` as an input type (such as `vtkVertexDegree`). The subclasses also enable filters to be specific about input type: if a filter requires a tree, that can be specified as the input type.

The separate mutable classes allow the enforcement of the following invariant: *All instances of graph data structures are valid at all times*. If `vtkGraph` itself was mutable, it would have methods for adding edges and vertices. Due to inheritance, this method would exist in all subclasses, including `vtkTree`. The result would be that `vtkTree` could at times hold a structure that is not a valid tree. It would be prohibitively expensive in time and complexity to check for a valid tree after every edge or vertex addition. Prefuse resolves this issue by not enforcing proper structure at all times, instead providing a method to check whether a tree is valid. This solves the problem, but relies on the caller to check whether the tree is valid at appropriate times.

In Titan, the user generates a tree by first creating an instance of `vtkMutableDirectedGraph`. After adding the appropriate vertices and edges to create the tree, the user calls `CheckedShallowCopy()` on an instance of `vtkTree`, passing the mutable graph as an argument. This method will do one of two things. If the tree is valid, it will set the structure via a shallow copy and return true. If it is not a valid tree, it will return false.



Fig. 8. Copy-on-write of graph structure (a) after a shallow copy, and (b) after the mutable graph is modified.

An additional feature of the graph data structures is copy-on-write sharing. Since all graphs share the same internal representation, multiple objects, even those of different types, may share the same structure. A deep copy of the structure is only made if the user modifies a graph whose structure is shared with other graphs (see Figure 8). To the caller, the graph instances behave as though they were independent of other graphs, while internally memory usage is optimized.

The graph data structures were designed for maximum efficiency. Figure 9 compares the time to construct and analyze `vtkGraph` with a comparable graph from the Boost Graph Library (BGL). The BGL provides a family of graph types with selectable storage structures. For this comparison, we chose the BGL graph type which stores the graph adjacency list as a vector of vectors, which is similar to the `vtkGraph` implementation. This structure provides memory efficiency and fast traversal, trading-away efficient modification and deletion. This trade-off is appropriate for VTK’s pipeline architecture since most filters either append to an existing data structure or create new structures from

scratch, making in-place deletion nearly nonexistent.

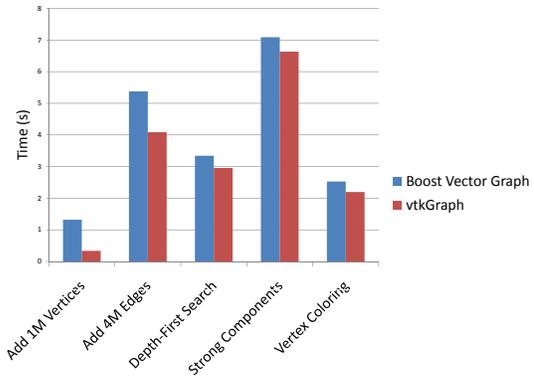


Fig. 9. Runtime of `vtkGraph` as compared to a boost graph. All algorithms were run on a graph with 1 million vertices and 4 million edges.

As in other toolkits, the user may assign an arbitrary number of arrays to the vertices and edges of the graph. These attributes are may be used in various ways to alter the visualization of the graph, such using them to color or label the graph.

5 CODE COUPLING

Applications targeted at real world analysis problems have to provide end-to-end functionality. To ease application development, a unified toolkit should provide a broad range of functionality from database connections and data ingestion to filtering, algorithms, analysis, presentation and interaction. To provide an expanded range of functionality the Titan toolkit uses, and collaborates with, a number of popular open source projects.

5.1 Boost Graph Library

Graph algorithms are an essential part of many informatics applications and a unified toolkit needs to provide that functionality. The Boost Libraries [1] are a popular and powerful set of libraries for C++. One of these libraries is the Boost Graph Library (BGL) [25]. The BGL provides many efficient, generic graph algorithms which operate on any graph type that implements a set of concepts. The Titan team decided to use these BGL algorithms instead of reimplementing them from scratch. To use a `vtkGraph` with a BGL algorithm, callers simply include `vtkBoostGraphAdapter.h`. This “data-less” adapter implements the required BGL concepts for `vtkDirectedGraph` and `vtkUndirectedGraph`, thus allowing BGL algorithms to process Titan graphs directly.

The interface between Titan and BGL follows the VTK pipeline model. Referencing the “hello world” code example in Figure 3 we see that the header files for the adapter and for the BGL algorithm are included. We create a BGL algorithm, put the algorithm in the pipeline, and then color the nodes of the graph by the results of the algorithm. From the developers perspective the BGL functionality is a pipeline component. In practice all the BGL algorithms can be run in similar fashion. We often string up combinations of various BGL algorithms in pipelines. In fact, one of the layout algorithms in the Titan toolkit (G-Space [29]) runs three BGL algorithms to compute geodesic distances.

5.2 Parallel Boost Graph Library

As a distributed memory toolkit, VTK currently provides a myriad of functionality around parallel scientific processing and visualization. The Parallel Boost Graph Library (PBGL) is a generic C++ library for high-performance parallel and distributed graph algorithms. The `vtkGraph` data structure, along with some distributed helper classes, enables the PBGL functionality to work in the same way as the BGL classes. The integration of PBGL functionality into Titan is currently in the early stages, but since we are collaborating directly with the Indiana University developers, the integration should provide high-performance and memory efficient access to PBGL algorithms.

5.3 Multithreaded Graph Library

The MultiThreaded Graph Library (MTGL) targets shared memory platforms such as the massively multi-threaded Cray MTA/XMT [12], and when used in tandem with the Qthreads library [28], chip multi-processors such as the Sun Niagara [3] and multi-core workstations. MTGL is based on the serial Boost Graph Library, as data distribution is not an issue on the platforms in question. Shared memory programming is a challenge, and algorithm objects in the MTGL can encapsulate much, but not all, of this challenge. As in the BGL, the visitor pattern enables users to apply methods at key points of algorithm execution. MTGL users write adapters over their graph data structures as BGL users do, but there is no assumption that Boost and STL are available. MTGL codes for connected components on unstructured graphs, a difficult problem for distributed memory architectures, have scaled almost perfectly on the Cray MTA-2.

5.4 Qt Libraries

Qt comprises a set of dual-licensed (open source and commercial) libraries for robust application development. The libraries include an extensive cross-platform GUI toolkit, unified database access, and SVG/XML processing. Integrating with Qt provides some immediate benefits for Titan. Although Titan now supports many native database connections, Qt's stellar database support jump-started our efforts to connect to many types of databases.

Information Visualization applications require not only powerful view types, but also tightly integrated GUI widgets. Qt satisfied this requirement for Titan applications. We were easily able to adapt vtkTable, vtkTree and vtkSelection into Qt's model/view architecture. Figure 10 depicts how the Titan's Qt tree view accepts a vtkTree and uses a Qt adapter to use it as the underlying model for a QTreeView widget. The view is also able to translate Qt selections into VTK selections in order to synchronize the selection with other VTK views.

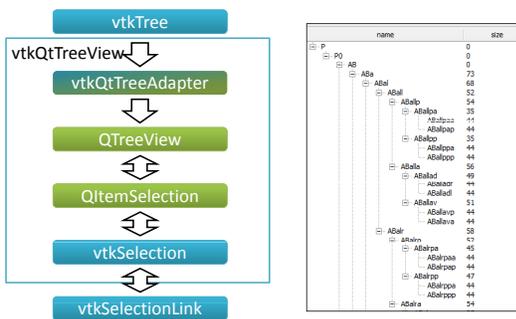


Fig. 10. How vtkQtTreeView encapsulates a Qt tree widget.

6 DATA INGESTION

In order for a toolkit to be useful (and used) in practice, it must be as simple as possible for users to load their data. In this section we describe Titan's mechanisms for ingesting data from files and databases as well as the construction process to generate graphs and trees from tabular data. The general model is to load unprocessed information from a file or database; apply filters to convert that information into a graph, tree or table; and finally to pass the finished data structures to the rest of the pipeline for further processing and analysis.

6.1 Loading From Files

For Titan's initial release we have implemented readers for some of the data formats most commonly used in informatics applications. These include delimited and fixed-width text readers for tabular data, graph readers that support the Tulip, Chaco and DIMACS file formats, and an XML reader that can convert well-formed XML to a vtkTree. Adding readers for new file formats is straightforward. If a file format explicitly defines its contents as a graph or a tree, the reader can immediately output a vtkGraph or vtkTree suitable for further processing. In

cases where a file contains tabular data, however, we choose to treat the reader's output as if it had been read from a database. In the next section we describe how this works.

6.2 Loading From Databases

While loading data from files is a useful capability, direct loading of data from an unprocessed database is one of the most powerful and most widely used features in Titan. Our work with customers has almost always focused on ingestion of data from arbitrary database queries. Large existing databases have thousands or millions of implicit graphs embedded within them. In addition to VTK readers for existing file formats, one of Titan's major contributions is an SQL database interface that allows a database to be used directly as a data source for the processing pipeline.

6.2.1 Database Support

Like file formats, there are many databases in everyday use, each with its own interface semantics. Titan already includes drivers for SQLite [23], MySQL [10] and PostgreSQL [9]. When built with Qt, Titan can also use Qt's support for many additional database types. An upcoming release will add ODBC support, which allows us to talk to a still-wider set of databases without having to write native drivers using each database's particular and often proprietary API.

6.2.2 Loading Data

Ingesting data from an SQL database is unlike loading from files in that we must first specify what data we want. This is exactly what SQL is designed for. Rather than retrieving the database schema at connect time and attempting to deduce a reasonable set of data to load, we allow the user to supply an arbitrary SQL query that identifies a data set of interest. This query can be as broad or as narrow as desired: the same mechanism that allows a brief preview of a large database can use the full power of SQL to extract data from multiple interacting tables. In both cases the output from the SQL query is a vtkTable. In Section 6.3 we describe how Titan converts this row-based output into trees, graphs and tables for further processing by the pipeline.

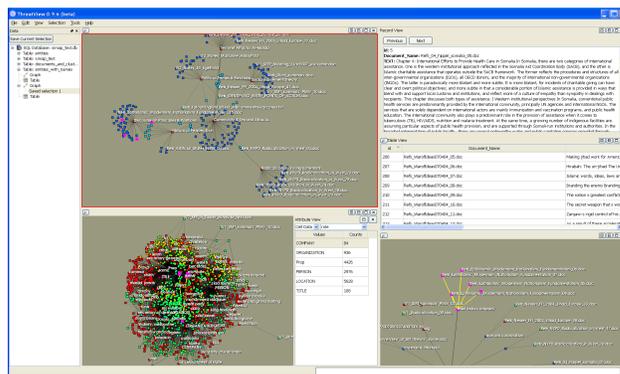


Fig. 11. A VTK/Titan toolkit based application incorporating multiple views of a data set.

6.3 Creating Graphs and Trees from Tables

In a typical analysis, data organized in tables only forms part of the end product. Analysts are commonly interested not only in the entities in a table but also in the relationships and links between them. When these links are stored as tabular data (which is often the case) we must provide a way to reconstitute them into a graph or a tree as appropriate. Moreover, since a nearly unlimited number of relationship graphs can be derived from any large database depending on the questions being asked, this mechanism must be flexible enough to adapt to the items and link structures of interest. In Titan we implement this using the vtkTableToGraph filter.

The vtkTableToGraph filter takes two inputs: an edge attribute table and an optional vertex attribute table. Each column in these tables

specifies a different property of the vertices or edges. Vertex attribute arrays are naturally assigned to the vertices of the output graph and commonly contain properties such as vertex name, size, or spatial position (where appropriate). Edge attributes are likewise assigned to the edges of the output graph but also encode the graph connectivity. The `vtkTableToGraph` filter creates one or more edges for each row in the edge table by following a template called the *link graph*. The link graph is a graph which has one vertex for every column in the edge table. The edges in the link graph specify how to create edges from each table row.

Figure 12 demonstrates this translation. On the left is the input edge table, in the center is the link graph, and on the right is the resulting graph. The link graph may connect an arbitrary number of columns in arbitrary patterns such as full connectivity (cliques), paths, or stars.

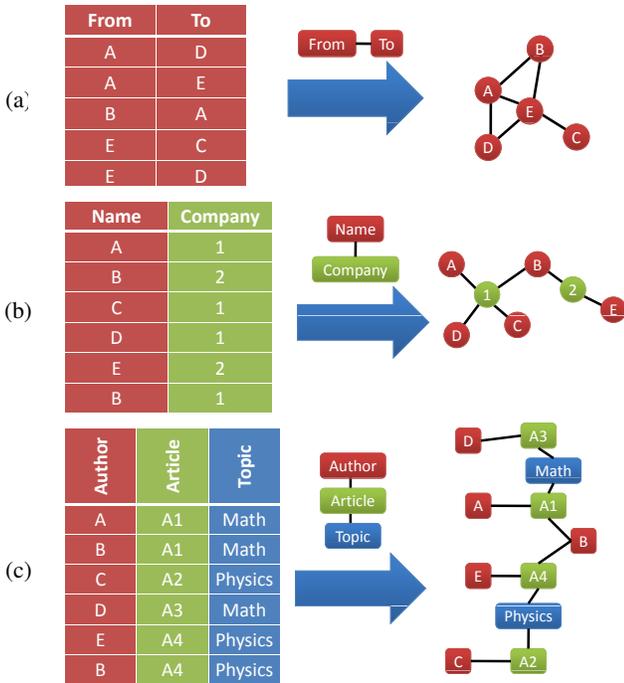


Fig. 12. How `vtkTableToGraph` translates an edge attribute table (left) into graph edges (right) based on a link graph (center).

Although `vtkTableToGraph` can create graphs of nearly arbitrary complexity, we do not often need its power when creating trees. Instead, we use two filters, `vtkTableToTree` and `vtkGroupLeafNodes`, that in combination produce arbitrary hierarchies. First, `vtkTableToGraph` takes a table as input and creates a simple two-level tree as output. A single vertex is created to be the root of the tree. Next, we create one new child of the root vertex for each row in the input table. Once all the data exist in this trivial tree structure we use `vtkGroupLeafNodes` to organize it.

We use a series of `vtkGroupLeafNodes` filters to organize a tree into multiple levels. Each instance of `vtkGroupLeafNodes` adds one level to a tree, just above the leaf nodes, that groups the leaves based on the values of some vertex attribute. The order in which these groupings are applied is reflected in the order of the `vtkGroupLeafNodes` instances within the pipeline. This is illustrated in Figure 13.

7 PIPELINE MODEL

Since Titan is integrated into the VTK, it inherits many of the powerful qualities of that toolkit. One of the distinguishing characteristics of this toolkit is a sophisticated pipeline model. Visualizations are created from processing units called algorithms or filters. These building blocks are reusable components that perform a well-defined operation on one or more input datasets in order to produce one or more outputs.

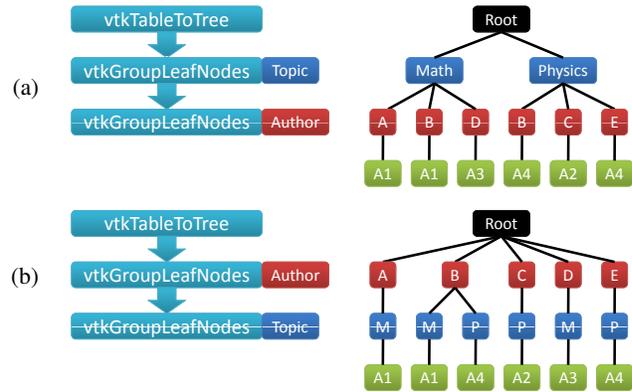


Fig. 13. Here we illustrate the use of `vtkTableToTree` and `vtkGroupLeafNodes` to convert the table in Figure 12 into a tree. At top we divide the leaf nodes first by topic, then by author. At bottom we divide first by author and then by topic. Any number of these groupings may be arranged to handle complex data.

A specific benefit which the Visualization Toolkit provides is what is called a “demand-driven pipeline”, which incorporates logic guaranteeing that algorithms will not be executed unless necessary. This results in optimally efficient component execution with no effort on the developer’s part. In a demand-driven pipeline, the user may modify parameters in algorithms at any place in the pipeline at any time. When the user requests an output of a specific algorithm, a request travels up the pipeline, asking algorithms along the way if their inputs or parameters have been modified since the last time the pipeline was updated. Then, the pipeline executes the minimal number of algorithms in order to bring the requested output up to date.

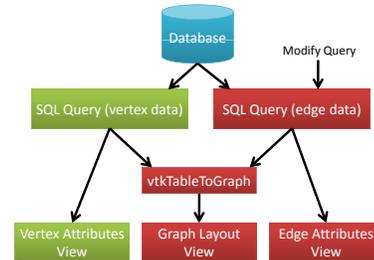


Fig. 14. The result of a demand-driven pipeline model. When the edge query is changed, only the pipeline components in red are updated when the views are redrawn.

This is best described by a practical example in information visualization. Suppose that we wish to query a database in order to construct a graph and layout the graph in a view. Additionally, we want to show the edge and vertex attributes of the graph in tabular views. A possible pipeline for this application is shown in Figure 14. If the edge query changes, initially nothing will update. When the application decides to update all views, only the parts of the pipeline in red will execute. While this may seem trivial in such a small example, in practical applications there are many more algorithms involved. Without this advanced pipeline model, the bookkeeping necessary to optimally reexecute algorithms quickly becomes prohibitive and unneeded processing will likely result. VTK’s demand-driven pipeline provides all of this complex bookkeeping for free, so algorithms are never executed until necessary.

8 PACKAGING PIPELINES WITH VIEWS

The Titan filters for generating geometry from data structures, glyphing, coloring, etc. may be used by themselves, but can take some

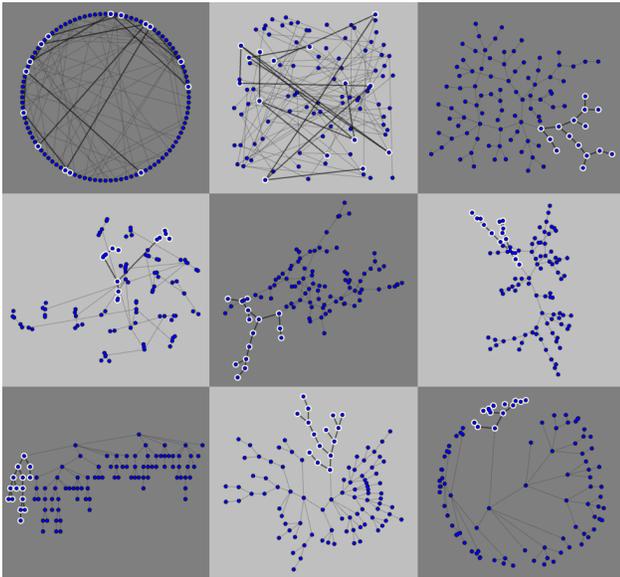


Fig. 17. A random tree shown in nine views with different layout strategies. The same subtree is selected through linked selection among the views. The layout strategies are circular (upper left), random (upper center), fast 2D (upper right), clustering (center left), force directed (center), simple 2D (center right), standard tree (lower left), radial tree (lower center), and radial tree with logarithmic scaling (lower right).

10 FUTURE WORK

At the time of this writing most of the informatics capabilities within Titan are serial and are not using the parallel processing and client/server capabilities of VTK. Our team (Sandia, Kitware, and Indiana University) is finalizing the parallel graph data structures (based on PBGL design) and have some working “hello world” examples. Once the parallel graph data structures are completed, the parallel development effort should proceed quite smoothly; `vtkTable` is trivially distributed, `vtkTree`, and `vtkDirectedAcyclicGraph` inherit from `vtkGraph` so are taken care of by the distribution and functionality in the superclass. The entire informatics pipeline will have to work efficiently in a distributed memory context. While the developers do not want to underestimate the complexity of this task, we have extensive experience in distributed memory software development; using the existing capabilities within VTK and packages like PBGL should greatly accelerate our progress.

For real world analysis, we will have to expand the functionality of Titan to include not only graph algorithms, but also parallel statistics, and linear (multi-linear) algebra libraries. The data structures that are still in our sandbox repository (`vtkDenseArray`, `vtkSparseArray`, `vtkFactoredArray`) will support N -dimensional (vector/matrix/tensor) storage and operations. The data structures are being designed so that we can provide “data-less adapters” between Titan and the popular linear algebra packages and statistics libraries (like Trilinos [16], ScaLAPACK and “R”).

11 CONCLUSION

We present an expansion of the popular open source Visualization Toolkit to support the ingestion, processing, and display of informatics data. The informatics capabilities are being released as open source into the VTK repository and leverage the same flexible, component based, demand-driven pipeline architecture. The unified toolkit should provide an excellent platform for the development of applications needing database access, scalable analysis, and views that combine scientific and information visualization techniques. The unification of information and scientific visualization is long overdue and we believe that VTK/Titan represents a good first step in that direction. We also recognize that a toolkit does not succeed without community

support and contribution, so we are actively seeking members of the community to collaborate with us on Titan 2.0.

ACKNOWLEDGEMENTS

The authors would like to thank the extended “Titan Family”: Patricia Crossno, Berk Geveci, Doug Gregor, John Greenfield, Randy Heiland, William McLendon, Kenneth Moreland, Thomas Otahal, Dave Partika, Philippe Pebay, David Rogers, Timothy Shead, Eric Stanton, David Thompson and Andy Wilson.

This work was conducted as part of the Titan project sponsored by Sandia National Laboratories in collaboration with Kitware Inc. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy’s National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

- [1] Boost C++ libraries. <http://www.boost.org>.
- [2] Press release: Sandia national labs visualizes some of the world’s largest simulations using nvidia technology. <http://www.nvidia.com/object/IO27539.html>.
- [3] Sun niagara 2 processor. <http://www.sun.com/processors/niagara>.
- [4] J. Ahrens, K. Brislaw, K. Martin, B. Geveci, C. C. Law, and M. E. Papka. Large-scale data visualization using parallel data streaming. *IEEE Computer Graphics and Applications*, 21(4):34–41, July/August 2001.
- [5] B. B. Bederson, J. Grosjean, and J. Meyer. Toolkit design for interactive structured graphics. In *IEEE Transactions on Software Engineering*, volume 30, pages 535–546, 2004.
- [6] K. Been and C. Yap. Dynamic map labeling. In *IEEE Transactions on Visualization and Computer Graphics*, volume 12, pages 773–780, 2006.
- [7] A. Cedilnik, J. Baumes, L. Ibanez, S. Megason, and B. Wylie. Integration of information and volume visualization for analysis of cell lineage and gene expression during embryogenesis. In *IS&T/SPIE Electronic Imaging, Visual Data Analytics (VDA)*, Jan. 2008.
- [8] A. Cedilnik, B. Geveci, K. Moreland, J. Ahrens, and J. Favre. Remote large data visualization in the ParaView framework. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, May 2006.
- [9] K. Douglas. *PostgreSQL*. Sams Developer’s Library, 2 edition, 2005. ISBN 06272327562.
- [10] P. DuBois. *MySQL*. Sams Developer’s Library, 3 edition, 2005. ISBN 0-672-32673-6.
- [11] J.-D. Fekete. The Infovis Toolkit. In *10th IEEE Symposium on Information Visualization (InfoVis’04)*, pages 167–174. IEEE Press, 2004.
- [12] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *CF ’05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, New York, NY, USA, 2005. ACM.
- [13] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.
- [14] J. Heer and M. Agrawala. Software design patterns for information visualization. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 12(5), Sep/Oct 2006.
- [15] J. Heer, S. K. Card, and J. A. Landay. Prefuse: A toolkit for interactive information visualization. In *ACM Human Factors in Computing Systems (CHI)*, pages 421–430, 2005.
- [16] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.
- [17] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. In *IEEE Transactions on Visualization and Computer Graphics*, volume 12, pages 741–748, 2006.
- [18] A. Kerren, J. T. Stasko, J.-D. Fekete, and C. North. Workshop report: information visualization: human-centered issues in visual representation, interaction, and evaluation. *IEEE Information Visualization*, 21(6):189–196, Oct 2007.
- [19] Kitware, Inc. *The Visualization Toolkit User’s Guide*, 2006.
- [20] Y. Koren and D. Harel. Graph drawing by high-dimensional embedding. In *LNCIS Graph Drawing (GD’02)*, volume 2528, pages 207–219, 2002.

- [21] C. Law, A. Henderson, and J. Ahrens. An application architecture for large data visualization: A case study. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, Oct. 2001.
- [22] K. Martin and B. Hoffman. *Mastering CMake*. Kitware, Inc., 4 edition, 2008. ISBN 1930934203.
- [23] M. Owens. *The Definitive Guide to SQLite*. Apress, 1 edition, 2006. ISBN 1-590-59673-0.
- [24] T.-M. Rhyne, M. Tory, T. Munzner, M. Ward, C. Johnson, and D. H. Laidlaw. Information and scientific visualization: Separate but equal or happy together at last. In *Proceedings of the 14th IEEE Visualization Conference*, 2003.
- [25] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*, 2002.
- [26] M. Takatsuka and M. Gahegan. Geovista studio: A codeless visual programming environment for geoscientific data analysis and visualization. *Computers & Geosciences*, 28(10), Dec 2002.
- [27] D. Weiskopf, K.-L. Ma, J. J. vanWijk, and R. Kosara. Scivis, infovis: bridging the community divide. In *Proceedings of the IEEE Visualization Conference*, 2006.
- [28] K. B. Wheeler, R. Murphy, and D. Thain. Qthreads: An api for programming with millions of lightweight threads. In *IEEE Workshop on MultiThreaded Architectures and Applications (MTAAP)*, 2008.
- [29] B. Wylie, J. Baumes, and T. M. Shead. Gspace: A linear time graph layout. In *IS&T/SPIE Electronic Imaging, Visual Data Analytics (VDA)*, Jan. 2008.