Rapid Visualization Development based on Visual Programming
Developing a Visualization Prototyping Language

Benedikt Stehno*
Supervised by: Eduard Gröller, Martin Haidacher

Institute of Computer Graphics and Algorithms
Vienna University of Technology
Vienna / Austria

Abstract

In this paper we introduce a dataflow visual programming language (DFVPL) and a visual editor for the rapid development of visualizations. It enables users with only little programming experience to develop custom visualizations. With this programming language, called OpenInsightExplorer, users can develop visualizations by connecting graphical representations of modules rather than writing source code. Each module represents a part of a processing step in the visualization pipeline. Modules are designed to function as an independent black box and they start to operate as soon as data is sent to them. This black box design and execution model allows to reuse modules more frequently and simplifies their development.

The usability of the programming language was evaluated by implementing two example visualizations with it. Each example originates from different areas of visualization (scientific and information visualization), therefore demanding different data types, data transformation tasks and rendering.

Keywords: visual programming, rapid prototyping, dataflow programming

1 Introduction

Visualization is a scientific research field which deals with developing computer aided techniques for visually representing large quantities of data. These techniques transform data into meaningful visual images, that should allow people to gain insight and help to interpret the data. Some of these techniques are interactive and allow to analyze the data sets in an interactive manner. Every visualization follows the concept of the visualization pipeline (see Figure 1) [3, 4, 10]. To develop a custom visualization a user needs to implement the stages of the visualization pipeline. The visualization pipeline consists of the following successive processing steps:

*benedikt.stehno@gmail.com

Data acquisition: In the first step the user defines the data source from which the data should be loaded. The data may get read out of special formatted files, databases or various other sources like simulations or real time measurements. Additionally this step often contains a *data analysis* process. The dataset is prepared for visualization in this step, e.g. by interpolating missing values, applying a smoothing filter or correcting erroneous measurements.

Filtering: Filtering is a user centered step. The user selects the portions of data he/she wants to be visualized. For example, a user selects data out of a certain time range, which should be visualized.

Mapping: The focused data gets mapped to geometric primitives (e.g. points and sprites) and their attributes (e.g. color, position). The focused data gets transformed to geometric data in this process stage.

Rendering: The final step of the pipeline transforms the geometric data into the resulting image, providing the *visualization output*.

Users who want to rapidly develop a visualization for certain data can use visualizations packages. Most of them are specialized to a certain field of visualization making it rather complicated to extend them with custom needed functionality. Such extensions can actually only be developed by users with significant programming knowledge. Moreover developers need to be familiar with the programming language the visualization application is written in.

In contrast OpenInsightExplorer supports visual programming which even non programmers can learn in a short timespan. The framework contains modules which can be combined in a graphical editor to a custom visualization pipeline. Only missing functionality needs to be implemented by developing new modules and adding them to the framework. Since modules in OpenInsightExplorer are designed to work as independent black boxes this can be easily achieved.

The next section deals with the state-of-the-art of dataflow visual programming languages and provides an

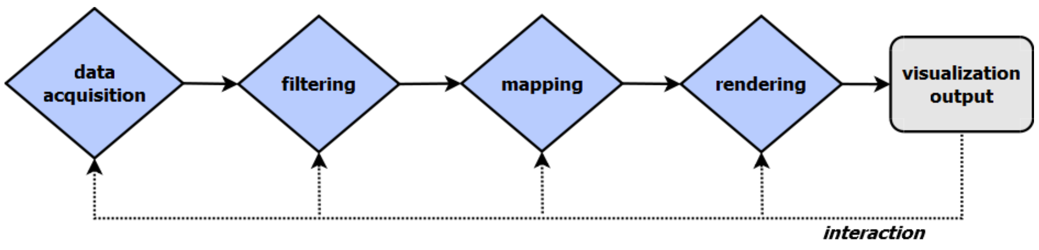


Figure 1: Visualization pipeline: It describes all possible processing steps from the data acquisition to the final visualization output. Since interaction plays a crucial role in visualization a user may interact with all processing steps of the pipeline in order to generate the desired visualization output.

introduction to their underlaying paradigms, visual programming and dataflow programming. In Section 3 an overview of the OpenInsightExplorer framework is given and its features are described in detail. The implementation of the framework is discussed in Section 4. The results that could be achieved by implementing two example visualizations with the framework are described in Section 5, followed by a conclusion including a discussion about future work in Section 6.

2 State of the Art

OpenInsightExplorer is based on two paradigms, the visual programming paradigm and the dataflow programming paradigm. These two paradigms were merged together, resulting in the family of *dataflow visual programming languages (DFVPL)*, to which OpenInsightExplorer belongs to.

Visual programming languages (VPL) allow users to program by manipulating or arranging graphical elements rather than writing textual source code. Users arrange or combine graphical symbols, following the specific syntax rules of a language.

Visual programming languages can be designed to work on a higher abstraction level than their textual counterparts using graphical metaphors [8]. This gives users the ability to work with them in a more intuitive way. Often they reach such an abstraction level that no prior programming experience or knowledge is required to express or design programs. Hence they often are used for *End User Development*, where users can create, modify or extend parts of a software without any significant knowledge about programming.

All of the programming languages presented in this section follow the concept of *boxes and arrows* [1, 5, 6]. Boxes represent independent modules which are connected by arrows in a graphical editor. The modules exchange data over these connections. They perform calculations or tasks as soon as they have received all necessary data for the execution. This principle is called the *dataflow execution model* [7, 13, 15]. In this model modules can be executed in parallel when they have received all necessary data for an execution.

Programming becomes in dataflow visual programming languages the task of connecting modules to a graph or network. This concept was implemented for example by the following state-of-the-art dataflow visual programming languages:

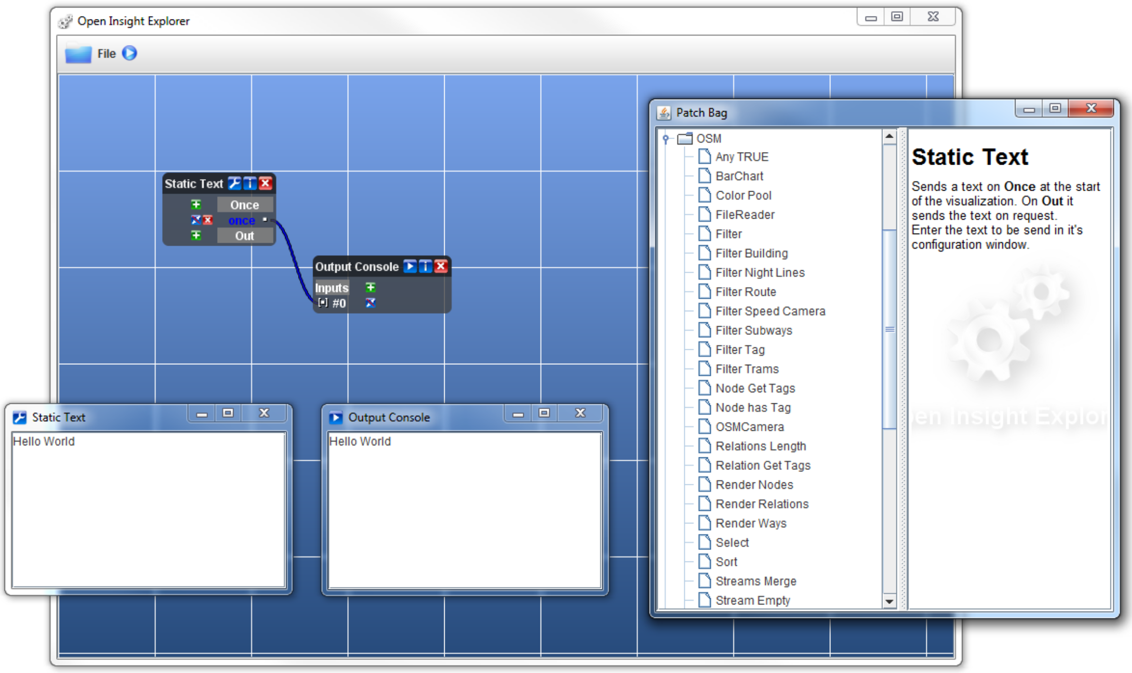
One of the first commercial DFVPLs was LabVIEW [9, 12]. LabVIEW is still in development and several versions of the platform have been released. With this software users can build virtual instruments by connecting different function nodes within a block diagram by drawing wires. It has been shown that large projects can be developed faster with a visual programming language in comparison to traditional text based programming languages [2]. LabVIEW became an industrial success and its benefits made it popular among researchers.

The DFVPL concept was also adopted for visualization purposes. OpenDX (Open Data Explorer) [16] is a cross-platform scientific data visualization software. It can deal with different kinds of data such as scalar, vector or tensor fields. A noteworthy feature of OpenDX is that it supplies GUI modules for interaction. With them the user is able to manipulate various aspects of the visualization with graphical user elements. Some of these, so called interactors, were developed to be smart and data driven. For example, sliders can automatically determine the minimum and maximum value(s) of the dataset, setting its boundaries appropriately.

Another example of a DFVPL for visualization purposes is MeVisLab [14, 19]. It is a very specialized visualization package for medical imaging and processing. It integrates VTK (Visualization Toolkit) [22, 20] modules in addition to its own ones to provide a wide range of specialized visualization modules.

But DFVPLs can also be found in the field of rendering and graphics processing. Quartz Composer [18] developed by Apple allows users to develop graphical rendering applications, e.g. for music visualization or as system screen saver. This framework can only handle built-in data types.

The presented state-of-the-art DFVPLs are highly optimized for their specific purposes. None of them was designed to serve as a language for scientific and information visualizations programming. E.g. Quartz Composer does not allow users to introduce custom data types, which proved to be a key feature for information visualization



Hello World

Hello World

Sends a text on **Once** at the start of the visualization. On **Out** it sends the text on request.
Enter the text to be send in it's configuration window.

Figure 2: A screenshot of the visual editor of OpenInsightExplorer. The editor's main window is depicted in the center and on the right side the patch repository (*Patch Bag*) is shown.

(see Section 5.2). Also OpenDX, which is a scientific visualization language, is incapable of supporting information visualization features. Additionally extending OpenDX with new functionality seems to be a more complex task in comparison to OpenInsightExplorer, which was specially designed for rapid prototyping and to be easy extendable with new modules (see Section 4).

The following section provides an introduction to OpenInsightExplorer and describes the features it provides to fit the needs as a general purpose visualization DFVPL.

3 OpenInsightExplorer

OpenInsightExplorer allows users to program their custom visualizations visually. Users simply connect graphical representations of modules in a visual editor rather than writing source code. Each module represents a part of a certain stage of the visualization pipeline (Figure 1). Figure 2 depicts a screenshot of the visual editor of OpenInsightExplorer.

There are modules that cover the step of *data acquisition*, for example a module that loads data from a file. Other modules may transform this data to geometric primitives. This occurs in the *mapping* stage of the pipeline. Connecting multiple individual modules with certain functionality together results in building a custom visualization pipeline. The user-defined connections express paths on which the data flows from one module to the next.

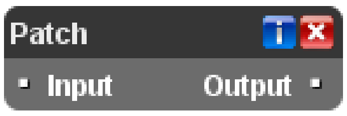


Figure 3: A screenshot of a *patch* with an *input port* and an *output port*.

The modules are called *patches* in the OpenInsightExplorer framework. Figure 3 depicts a screenshot of a simple patch. They operate as independent *black boxes*. That means that the user does not need to know precisely how they work. It is only necessary to know what they do. Since every stage of the visualization pipeline exchanges data with its preceding and/or succeeding stage, patches need to exchange data with each other as well. They have so called *input ports* and *output ports* (see Figure 3). Through the input ports a patch receives data. It processes the data and passes its results to another patch through its output ports.

To create visualizations with OpenInsightExplorer, users only need to find patches with the desired functionality and connect them in the visual editor of the framework (as depicted in Figure 2). Patches can be found in the patch repository window entitled *Patch Bag*. They are sorted in a tree structure by their functionality. When a user selects a patch in the repository, information about the patch is displayed in the right section of the *Patch Bag* window. Selected patches can be dragged in the main editor's window where they can be connected. Visualizations developed with the OpenInsightExplorer are called *compositions*. Using this simple visual programming concept allows users with little programming experience to program custom visualizations [9].

Like all other DFVPLs OpenInsightExplorer supports automatic parallelization of the execution of its modules. Patches can be executed in parallel as soon as they have received all necessary data for an execution, since it uses the same *dataflow execution model* principle on which every DFVPL relies on. Important and partially unique features of OpenInsightExplorer in comparison to the state-of-the-art languages presented in this paper are listed below:

Platform Independence: The framework is written in *Java*, which is a platform independent programming language. Many platforms and architectures support runtime environments for Java (JRE) and can run software written in Java.

Growing Ports: The growing ports mechanism of OpenInsightExplorer is a unique feature which the presented state-of-the-art programming languages do not provide. It allows to add and remove ports dynamically to a patch while editing a visualization. Figure 4 shows a simple example of the mechanism. E.g. a patch that determines the maximum of a set of numbers should be flexible with respect to the number of operands of the function - hence the number of input ports.

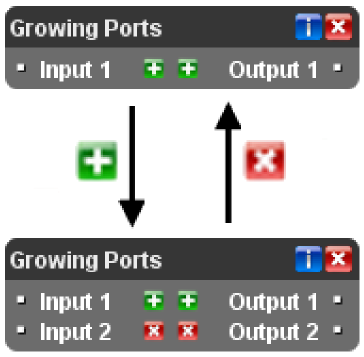


Figure 4: Illustrating the growing ports mechanism. Ports can have *add* and/or *remove* icons which will trigger the mechanism.

Streams: Instead of sending only individual data tokens between patches, OpenInsightExplorer implements the concept of *token streams* [15]. Patches can have special *stream* ports which enable them group data together to a stream. A stream consists of a start token, an ordered sequence of data and a token which will signal the end of a stream. Streams can also be embedded into another stream, which is a big improvement over flat arrays. These streams within streams are called sub streams.

Port Trees: Ports can be organized in trees. Also labels can be added to port trees (Figure 5). This allows to structure the input and output ports of a patch, and to add and remove ports dynamically.

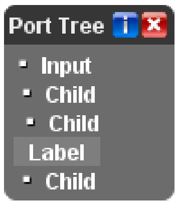


Figure 5: Ports are organized in a tree structure.

Generic Ports: To make patches more flexible, OpenInsightExplorer features generic port types. Patches can have ports, which are not assigned to a certain data type. As soon as they are connected, they can adapt their data type to the type of the connection partner. They can change their data type dynamically. This feature allows to implement patches, which can operate on any desired data type and can be used more frequently.

Patch GUI: Developers can place GUI elements of a patch in three different locations. Patches can have a *running* GUI which is a window that will be visible during the runtime of a visualization. For example the *Renderer* patch providing an OpenGL render surface uses the running GUI window for output purposes. The second possibility to add a GUI is the *configuration* GUI. This window will only be visible during editing a visualization. It is useful to display GUI elements that configure the behaviour of a patch. The third location is the *bound* GUI. It is directly visible between the input and output ports of a patch (Figure 6).

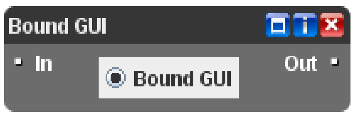
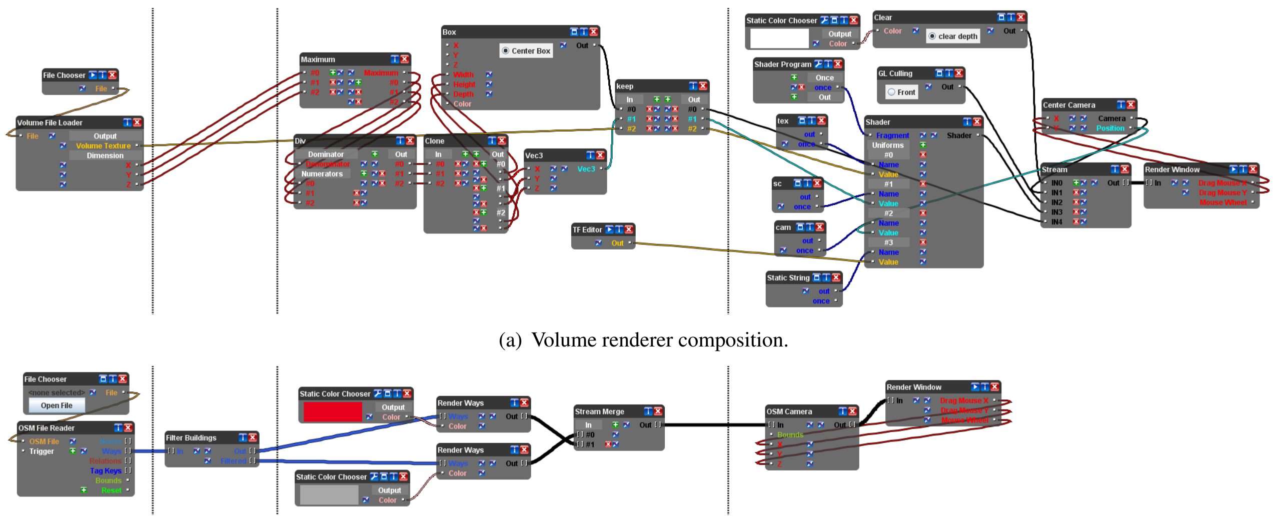


Figure 6: A patch with a *bound* GUI.

Custom Data Types: Unlike some state-of-the-art DFVPLs OpenInsightExplorer allows users to introduce new data types. Ports can be constructed with any arbitrary data type developers of a patch may desire. This is in our opinion a very important feature because visualizations can be build upon very different data types (e.g. volumetric data, data structures that represent graphs).

These classes can contain methods and functions in addition to the data. For example, a class that represents a graph can have a method which returns the names of



(a) Volume renderer composition.

(b) OpenStreetMap visualization composition.

Figure 7: Two example visualization compositions. They are split into following successive stages of the visualization pipeline: *data acquisition*, *filtering*, *mapping* and *rendering*

the nodes of the graph sorted. Furthermore such a class could implement many different interfaces and therefore represents multiple data types at once. For example, a graph class can implement two interfaces at once: one represents an undirected graph and the other one a directed graph.

Delegating Patches: The exchange of classes containing functions enables the development of patches that follow the *delegation* pattern. A patch can call a function of a previously received helper object and therefore delegates certain needed functionality to it. This can greatly enhance the usability of patches.

Type-safety: Ports in the framework support a type-safety mechanism. Every port of a patch is constructed for a certain data type (with the exception of generic ports which were discussed before). It can only send or receive a certain data type it was assigned to. Whenever a user tries to connect two patches in the visual editor OpenInsightExplorer verifies if the data types of the input port and output port are compatible. The color of the name of a indicates hints the data type the port was constructed for.

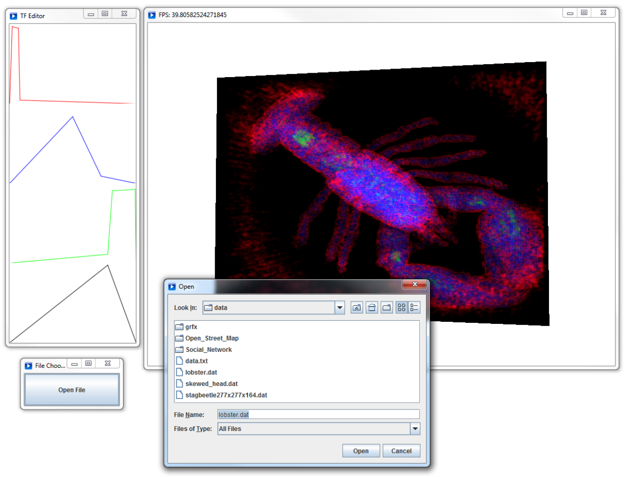
The next section deals about general architecture of the framework and how some of the previous mentioned features are implemented.

4 Implementation

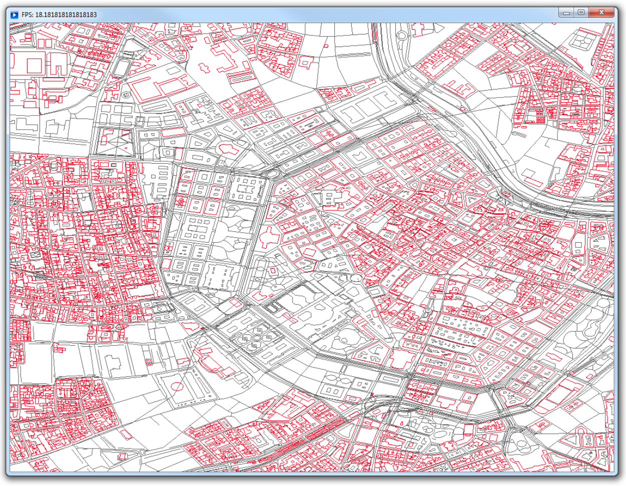
The framework and the visual editor is written in the platform independent programming language Java. OpenInsightExplorer loads platform dependent native libraries at runtime, which makes it possible to port the framework to different operating systems and hardware architectures. However the current version only supports the operating system Windows, since needed native libraries for OpenGL rendering (Jogl [11]) are distributed only for that operating system with the framework. But porting to other platforms / operating systems should be a feasible task.

OpenInsightExplorer can be extended with new features by developing new patches and adding them to the framework. This process was designed to be easy to accomplish, since patches are designed to operate as independent black boxes. To create a new patch, users only need to implement a certain *Patch* interface (see Listing 1). Its design is inspired by the Java's Applet interface. It contains methods to initialize *init()* and *reset()* a patch and the methods *start()* and *stop()* for its execution. In addition it declares methods that return references to the GUI elements of a patch and its input and output port trees. A new patch only gets compiled once and its binary *class* file copied into the patch entitled sub-folder of the framework. At every startup of OpenInsightExplorer the framework scans this folder and displays all found patches in the patch repository.

To send and receive data, patches can instance input or output port objects and add them to their appropriate port tree. Port objects can only be assigned to a certain data type, since it is a class with a generic type parameter. Many different *callback functions* can be registered to ports. These functions enable the implementation of the growing port mechanism and generic ports. They are called depending on different events. E.g. a generic port calls a specific callback function as soon as the user tries to connect this generic port to another port in the visual editor. Developers can implement code into this callback



(a) On the left is the transfer function editor depicted and in front a file open dialogue. The right window is the rendering output window, displaying the volume visualization.



(b) All ways are rendered in gray and buildings in red.

Figure 8: Screenshots of the example visualizations: the volume renderer (a) and the OpenStreetMap visualization (b).

function that will adapt the patch so it can handle the data type of the connection partner or refuse the connection attempt. The same principle is applied to the growing port mechanism (see Figure 4). The add and remove icons trigger different callback function in which a developer can implement code that will add or remove one or more ports to/from the port trees.

```
public interface Patch {

   public void init();
   public void reset();
   public void start();
   public void stop();

   public String getInfo();
   public String getName();

   public void load(Serializable o);
   public Serializable save();

   public Port getInputPorts();
   public Port getOutputPorts();

   public JPanel getBoundGUI();
   public JFrame getConfigurationGUI();
   public JFrame getRunningGUI();

}
```

Listing 1: The *Patch* interface.

The implementation of the framework is hidden from patch developers by applying the *cheshire cat* programming pattern [21]. They are only confronted with functions or methods which are truly necessary for developing patches and cannot (accidentally) access the framework internals. Applying this pattern to the framework ensures that patches can work only as absolute independent black boxes and enforces the interchangeability of patches.

5 Results

To evaluate the usability of the OpenInsightExplorer framework two example visualizations were implemented with it. The first example is a volume renderer, which comes from the field of scientific visualization. To test the frameworks information visualization capabilities, the second example is a collection of different visualizations of the OpenStreetMap project. The example visualizations demand different data types, data transformations and rendering techniques.

5.1 Volume Rendering

The first example visualization which was implemented to evaluate the framework is a simple volume renderer based on raycasting. Figure 8(a) depicts a screenshot of the running visualization. The goal was to use only general purpose patches of the framework whenever possible for the volume visualization. Only two custom patches, the *Volume File Loader* and the *Transfer Function Editor*, had to be implemented in addition. The first one loads the volume data from a file and converts it to a 3D texture. The support for other volume file formats can be easily achieved by developing other patches for those formats. Figure 7(a) illustrates the composition of the volume renderer. The example uses GPU hardware acceleration for image generation. This is achieved by using GLSL fragment shader programs, which implement the ray sampling and composition functions. This example composition contains only 21 patches in total.

5.2 OpenStreetMap Visualization

The second example visualizes data from the *OpenStreetMap (OSM)* [17] project. OpenStreetMap is a collaborative project to create a free editable map of the world. Maps from OpenStreetMap contain information about highways, buildings, public transport and much more. For this example visualization a map of the city of Vienna was used. It was extracted from the OpenStreetMap database.

OpenStreetMap maps can be exported to XML-files. These files are built on only three simple elements: *node*, *way* and *relation*. Each element may have an arbitrary number of properties (*tags*) which are key-value pairs (e.g. highway=primary). Since OpenInsightExplorer allows users, as a feature, to introduce arbitrary data types, these elements can be mapped to specially developed classes.

The visualization shows all streets and buildings of Vienna. Figure 8(b) depicts a screenshot of the visualization: street are visualized in gray and buildings in red. The corresponding composition contains only 10 patches in total (see Figure 7(b)). This example demonstrates the capabilities and the benefits of the usage of individually implemented data structures to create patches. The big advantage is that the user can achieve the desired result with very few lines of code. The example also emphasizes the fact that re-usability is highly given. If a patch is already implemented it does not require much effort for minor modifications to re-use it for another purpose.

6 Limitations and Conclusions

Like most other dataflow languages, OpenInsightExplorer is prone to dataflow network deadlocks. It does not introduce new features for deadlock prevention or recognition to the field of dataflow language research. OpenInsightExplorer follows a coarse grained dataflow approach. This means that the modules are rather complex and such deadlocks seldom occur.

It does not support any kind of structured programming, which nearly all current visual dataflow programming languages do. Also OpenInsightExplorer provides only a basic debugging support in comparison to other state-of-the-art languages. Extending the framework with more sophisticated debugging tools and structured programming support should be a very feasible task.

Despite the existing drawbacks of the framework, OpenInsightExplorer contains unique features in comparison to the presented state-of-the-art languages, like the growing port mechanism and generic ports. Both mechanisms enable the development of more flexible and reuseable modules. Developers can use and introduce arbitrary data types to the framework. Many other existing visual dataflow programming languages are not capable of this.

OpenInsightExplorer cannot be used as a universal tool for non-programmers for developing arbitrary visualizations. But users with programming experience can benefit from the framework. They are able to implement all missing modules and can reuse already existing ones. This speeds up the development process and allows to rapidly prototype rather simple visualizations. Nevertheless OpenInsightExplorer implements features, which could bring great benefits to other visual dataflow languages. They are worthwhile to be adopted by current state-of-the-art languages, which may not possess them, to improve their usability.

References

[1] K. Arvind and D.E. Culler. *Dataflow architectures*, pages 225–253. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

[2] E. Baroth and C. Hartsough. *Visual programming in the real world*, pages 21–42. Manning Publications Co., Greenwich, CT, USA, 1995.

[3] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in information visualization: using vision to think*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[4] E. H. Chi. A taxonomy of visualization techniques using the data state reference model. In *Proceedings of the IEEE Symposium on Information Vizualization 2000*, pages 69–75, Washington, DC, USA, 2000. IEEE Computer Society.

[5] A. L. Davis and R. M. Keller. Data flow program graphs. *Computer*, 15:26–41, 1982.

[6] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.

[7] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. *SIGARCH Computer Architecture News*, 3:126–132, 1974.

[8] G. Fischer, E. Giaccardi, Y. Ye, A. G. Sutcliffe, and N. Mehandjiev. Meta-design: a manifesto for end-user development. *Communication of the ACM*, 47:33–37, 2004.

[9] T. R. G. Green and M. Petre. Usability Analysis of Visual Programming Environments: A ‘cognitive dimensions’ framework. *Journal of visual languages and computing*, 7:131–174, 1996.

[10] M. Haidacher. *Information-based Feature Enhancement in Scientific Visualization*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 2011.

[11] Jogl. http://kenai.com/projects/jogl/pages/Home. Accessed: 2011-02-06.

[12] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36:1–34, 2004.

[13] R. Karp and R. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal*, 14:359–370, 1966.

[14] MeVisLab. http://www.mevislab.de/. Accessed: 2010-11-09.

[15] J. P. Morrison. *Flow-Based Programming, 2nd Edition: A New Approach to Application Development*. CreateSpace, Paramount, CA, 2010.

[16] OpenDX. http://www.opendx.org/index2.php. Accessed: 2011-03-12.

[17] OpenStreetMap. http://www.openstreetmap.org. Accessed: 2011-03-17.

[18] Quartz Composer. http://developer.apple.com/graphicsimaging/quartz/quartzcomposer.html. Accessed: 2011-02-06.

[19] J. Rexilius, J. M. Kuhnigk, H. K. Hahn, and H. O. Peitgen. An application framework for rapid prototyping of clinically applicable software assistants. In Christian Hochberger and Rdiger Liskowsky, editors, *GI Jahrestagung (1)*, volume 93 of *LNI*, pages 522–528. GI, 2006.

[20] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. *The visualization toolkit: an object-oriented approach to 3D graphics*. Prentice-Hall, Inc., 2 edition, 1998.

[21] H. Sutter. *Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions*. Pearson Higher Education, 2004.

[22] Visualization Toolkit. www.vtk.org. Accessed: 2011-06-01.